A polytime complexity analyser for Probabilistic Polynomial Time over imperative stack programs.

J.Y. Moyen\(^1\) and P. Parisen Toldin\(^2\)

1 LIPN, UMR 7030, CNRS, Université Paris 13 F-93430 Villetaneuse, France. Jean-Yves.Moyen@lipn.univ-paris13.fr
2 Dipartimento di Scienze dell’Informazione, Università di Bologna Équipe FOCUS, INRIA Sophia Antipolis Mura Anteo Zamboni 7, 40127 Bologna, Italy. parisent@cs.unibo.it

Abstract

We present \textit{iSAPP} (Imperative Static Analyser for Probabilistic Polynomial Time), a complexity verifier tool that is sound and extensionally complete for the Probabilistic Polynomial Time (PP) complexity class. \textit{iSAPP} works on an imperative programming language for stack machines. The certificate of polynomiality can be built in polytime, with respect to the number of stacks used.

1998 ACM Subject Classification Theory, Verification

Keywords and phrases ICC, Probabilistic Polytime, Static verifier

1 Introduction

One of the crucial problem in program analysis is to understand how much time it takes a program to complete its run. Having a bound on running time or on space consumption is really useful, specially in fields of information technology working with limited computing power. Solving this problem for every program is well known to be undecidable. The best we can do is to create an analyser for a particular complexity class able to say “yes”, “no”, or “don’t know”. Creating such an analyser can be quite easy: the one saying every time “don’t know” is a static complexity analyser. The most important thing is to create one that answers “don’t know” the minimum number of time as possible.

We try to combine this problem with techniques derived from Implicit Computational Complexity (ICC). Such research field combines computational complexity with mathematical logic, in order to give machine independent characterisations of complexity classes. ICC has been successfully applied to various complexity classes such as \text{FP} \cite{2, 11, 4}, \text{PSPACE} \cite{12}, \text{LOGSPACE} \cite{8}.

ICC systems usually work by restricting the constructions allowed in a program. This \textit{de facto} creates a small programming language whose programs all share a given complexity property (such as computing in polynomial time). ICC systems are normally extensionally complete: for each function computable within the given complexity bound, there exists one program in the system computing this function. They also aim at intentional completeness: each program computing within the bound should be recognised by the system. Full intentional completeness, however, is undecidable and ICC systems try to capture as many programs as possible (that is, answer “don’t know” as little time as possible).

Having an ICC system characterising a complexity class \(\mathcal{C}\) is a good starting point for developing a static complexity analyser. There is a large literature on static analysers for complexity bounds. We develop an analysis recalling methods from \cite{9, 3, 10}. Comparatively to these approaches our system works with a more concrete language of stacks, where
variables, constants and commands are defined; we are also sound and complete with respect to the Probabilistic Polynomial time complexity class (PP) [7].

We introduce a probabilistic variation of the LOOP language. Randomised computations are nowadays widely used and most of efficient algorithms are written using stochastic information. There are several probabilistic complexity classes and BPP (which stands for Bounded-error Probabilistic Polytime) [7] is considered close to the informal notion of feasibility. Our work would be a first step into the direction of being able to capture real feasible programs solving problems in BPP (BPP ⊆ PP) [7].

Similar work has been done in [6] with characterisation of complexity class PP; This work gives a characterisation of complexity class PP by using a functional language with safe recursion as in Bellantoni and Cook [2].

Our system is called iSAPP, which stands for Imperative Static Analyser for Probabilistic Polynomial Time. It works on a prototype of imperative programming language based on the LOOP language [14]. The main purpose of this paper is to present a minimal probabilistic polytime certifier for imperative programming languages.

Following ideas from [9, 3] we “type” commands with matrices, while we do not type expressions since they have constant size. The underlying idea is that these matrices express a series of polynomials bounding the size of stacks, with respect to their input size. The algebra on which these matrices and vectors are based is a finite (more or less tropical) semi-ring.

2 Stacks machines

We study stacks machines, a generalisation of the classical counters machines. Informally, a stacks machine work with letters belonging to a finite alphabet and stacks of letters. Letters can be manipulated with operators. Typical alphabet include the binary alphabet {0, 1} or the set long int of 64 bits integers. On the later, typical operators are + or *.

Each machine has a finite number of registers that may hold letters and a finite number of stacks that may hold stacks. Tests can be made either on registers and letters (with boolean operators) or to check whether a given stack holds the empty stack. There are only bounded (for) loops which are controlled by the size of a given stack. That is, it is more alike a foreach (element in the stack) loop. Since there are only bounded loops (and no while), this de facto limits the language to primitive recursive functions. In this way, stack machines are a generalisation of the classical LOOP language [14]. Since our analysis is compositional, we add also functions to the language; their certificates can be computed separately and plugged in the right place when a call is performed.

2.1 Syntax and Semantics

We denote ⟨⟩ the empty stack and ⟨a1...an⟩ the stack with n elements and a1 at top.

Definition 1. A stacks machine consists in:
- a finite alphabet Σ = {a1, ..., an} containing at least two values true and false;
- a finite set of operators, \( \text{op}_i \) of type \( \Sigma^n \rightarrow \Sigma \), containing at least a 0-ary operator \( \text{rand} \), operators whose co-domain is \{true, false\} are predicates noted \text{op}?;
- a finite set of registers \( r \) and stacks, \( S_j \) (the empty stack is noted ⟨⟩);
and a program written in the following syntax:

\[ b \in \text{BooleanExp} ::= \text{true} | \text{false} | \text{op}(e_1, \ldots, e_n) | \text{rand()} | \text{isempty}(S) \]

\[ e \in \text{Expressions} ::= c | \text{op}(e_1, \ldots, e_n) | \text{top}(S) \]

\[ C \in \text{Commands} ::= \text{skip} | r := e | S_1 := S_2 | S := \{c_1 \ldots c_n\} | S_k := \text{call}(f, S_1 \ldots S_n) \]

\[ \text{pop}(S) | \text{push}(e, S); C; C | \text{If } b \text{ Then } C \text{ Else } C \mid \text{loop } S \{ C \} \]

\[ f \in \text{Functions} ::= \text{def } f \text{ in } \{ S_1 \ldots S_n \} \{ C \} \text{ out}(S_j) \]

Note that registers may not appear directly in boolean expressions to avoid dealing with the way non-boolean values are interpreted as booleans. However, it is easy to define a unary predicate which, e.g., sends \text{true} to \text{true} and every other letter to \text{false} to explicitly handle this.

Expressions always return letters (content of registers) while commands modify the state but do not return any value. \text{top}() does not destruct the stack but simply returns its top element while \text{pop}() remove the top element from the stack but does not return anything. It is also possible to assign constant stack to a stack.

The \text{isempty}() predicate returns \text{true} if and only if the stack given in argument holds the empty stack and \text{false} otherwise. The \text{loop } S \{ C \} commands executes \text{C} as many time as the size of \text{S}. Moreover, \text{S} may not appear in \text{C}. It is, however, possible to make a copy beforehand if the content is needed within the loop. Finally, we give the possibility to have function call. The command \text{call}(f, S_1 \ldots S_n) call the function \text{f} passing the actual arguments \text{S}_1 \ldots \text{S}_n and finally return the result stored in the stack \text{S}_j.

### 2.2 Complexity

The set of operators is not specified and may vary from one stacks machine to another (together with the alphabet). This allows for a wide variety of settings parametrised by these. Typical alphabets are the binary one (\{\text{true}, \text{false}\}), together with classical boolean operators (\text{not, and, ...}); or the set \text{long int} of 64 bits integers with a large number of operators such as \text{+, *, <, ...}. Since there is only a finite number of letters and operators all have the alphabet as domain and co-domain, there is only a finite number of operators at each arity. So, without going deep into details, it makes sense to consider that each operator take a constant time to be computed. More precisely, each operator can be computed within a time bounded by a constant. Typically, on \text{long int}, \text{+} can be computed in 64 elementary (binary) additions and \text{**} takes a bit more operations but is still done in bounded time.

Thus, in order to simplify the study, we consider that operators are computed in constant time and we do not need to take individual operators into account when bounding complexity. It is sufficient to consider the number of operators.

The only thing that is unbounded is the size (length) of stacks. Thus, if one want, e.g., to handle large integers (larger than the size of the alphabet), one has to encode them within stacks. The most obvious ways being the unary representation (a number is represented by the size of a stack) and the binary one (a stack of 0 and 1 is interpreted as a binary number with least significant bit on top). Obviously, any other base can be use. In each case, addition (and multiplication) has to be defined for this representation of “large integers” with the tools given by the language (loops). Of course, encoding unbounded value is crucial in order to simulate arbitrary Turing Machines (or even simply \text{Ptime} ones) and is thus required for the completeness part of the result.

Note that copying a whole stack as a single instruction is a bit unrealistic as it would rather takes time proportional to the size of the stack. However, since each stack will
individually be bounded in size by a polynomial, this does not hampers the polynomiality of the program. A clever implementation of stacks with pointers (i.e. as lists) will also allow copy of a whole stack to be implemented as copy of a single pointer, an easy operation.

Since the language only provides bounded loops whose number of execution can be (dynamically) known before executing them, only primitive recursive functions may be computed. This may look like a big restriction but actually is quite common within classical ICC results on $\text{Ptime}$. Notably, Cobham [5] or Bellantoni and Cook [2] both work on restrictions of the primitive recursion scheme; Bonfante, Marion and Moyen [4] split the size analysis (quasi-interpretation) from a termination analysis (termination ordering) which also characterise only primitive recursive programs; and lastly Jones and Kristiansen [9], on which this work is directly based, use the LOOP language which also allows only primitive recursion.

Since loops are bounded by the size of stacks, it is sufficient to bound the size of stacks in order to bound the time complexity of the program. Indeed, if each stack has a size smaller than $p$ and the program has never more than $k$ nested loops, then its runtime cannot be larger than $p^k$. Similarly, in the original $\text{mwp}$ calculus of Jones and Kristiansen, it was sufficient to bound the value of stacks in order to bound the runtime of programs (for the same reasons). Note that to have a large number of iterations, one first has to create a stack of large size, that is when bounding the number of iterations stacks are considered de facto as unary numbers.

For each stack, we keep the dependencies it has from the other stacks. For example, after a copy ($S_1 := S_2$), the size of $S_1$ is the same as the size of $S_2$. Keeping precise dependencies is not manageable, so we only keep the shape of the dependence (e.g. the degree with which it appear in a polynomial). These shapes are collected in a vector (for each stack) and combining all of them gives a matrix certificate expressing the size of the output stacks relatively to the size of the input stacks. The matrix calculus we obtain for the certificates is compositional. This allows for a modular approach of building certificates.

### 3 Algebra

Before going deeply in explaining our system, we need to present the algebra on which it is based. iSAPP is based on a finite algebra of values. The set of scalars is $\text{Values} = \{0, L, A, M\}$ and these are ordered in the following way $0 < L < A < M$. The idea behind these elements is to express how the value of stacks influences the result of an expression. $0$ expresses no-dependency between stack and result; $L$ (stands for “Linear”) expresses that the result linearly depends with coefficient $1$ from this stack. $A$ (stands for “Additive”) expresses the idea of generic affine dependency. $M$ (stands for “Multiplicative”) expresses the idea of generic polynomial dependency.

We define sum, multiplication and union in our algebra as expressed in Table 1. The reader will immediately notice that $L + L$ gives $A$, while $L \cup L$ gives $L$ The operator $\cup$ works as a maximum. Over this semi-ring we create a module of matrices, where values are elements of $\text{Values}$. We define a partial order $\leq$ between matrices of the same size as component wise ordering. Particular matrices are $\textbf{0}$, the one filled with all $0$, and $\textbf{I}$, the identity matrix, where elements of the main diagonal are $L$ and all the others are $0$. If $v \in \text{Values}$, a particular vector is $V^v_i$ that is a column vector full of zeros and having $v$ at $i$-th row. Multiplication and addition between matrices work as usual\footnote{That is: $(A + B)_{i,j} = A_{i,j} + B_{i,j}$ and $(A \times B)_{i,j} = \sum A_{i,k} \times B_{k,j}$} and we define point-wise union between matrices: $(A \cup B)_{i,j} = A_{i,j} \cup B_{i,j}$. Notice that $A \cup B \leq A + B$. \footnote{That is: $(A + B)_{i,j} = A_{i,j} + B_{i,j}$ and $(A \times B)_{i,j} = \sum A_{i,k} \times B_{k,j}$}
As usual, multiplication between a value and a matrix corresponds to multiplying every element of the matrix by that value.

We can now move on and present some new operators and properties of matrices. Given a column vector \( V \) of dimension \( n \), a matrix \( A \) of dimension \( n \times m \) an index \( i (i \leq m) \), we indicate with \( A_i \leftarrow V \) a substitution of the \( i \)-th column of the matrix \( A \) with the vector \( V \).

Next, we need a closure operator. The “union closure” is the union of all powers of the matrix: \( A \cup \mathcal{A} = \bigcup_{i \geq 0} A^i \). It is always defined because the set of possible matrices is finite.

We will need also a “merge down” operator. Its use is to propagate the influence of some stacks to some other and it is used to correctly detect the influence of stacks controlling loops onto stacks modified within the loop (hence, we can also call it “loop correction”). The last row and column of the matrix is treated differently because it will be use to handle constants and not stacks. In the following, \( n \) is size of the vector, \( k < n \) and \( j < n \).

\[
(V_{ik,n}^j)_i = \begin{cases} 
M & \text{if } \exists p < n, p \neq k \text{ such that } V_p \neq 0 \\
0 & \text{otherwise and } V_n = 0 
\end{cases}
\]

\[
(V_{ik,j}^j)_k = \begin{cases} 
L & \text{otherwise and } V_n = L \\
A & \text{otherwise and } V_n \geq A \\
V_k & \text{otherwise} 
\end{cases}
\]

\[
(V_{ik,j}^j)_i = \begin{cases} 
0 & \text{if } i = n \\
M & \text{if } i \neq j, V_i \neq 0 \text{ and } V_j \neq 0 \\
V_i & \text{otherwise} 
\end{cases}
\]

In the following we will use a slightly different notation. Given a matrix \( A \) and an index \( k \), \( A^{ik} \) is the matrix obtained by applying the previous definition of merge down on each column of \( A \). Formally, if \( V \) is the \( j \)-th column of \( A \), then \( j \)-th column of \( A^{ik} \) is \( V_{ik,j} \).

Finally, the last operator that we are going to introduce is the “re-ordering” operator. Given a vector \( V \) we write \( V^{1 \rightarrow i, \ldots, n \rightarrow j} \) to indicate that the result is a vector whose rows are permuted. The first raw goes in the \( i \)-th position and so on till the \( n \)-th to the \( j \)-th. In order to use a short notation, if a row is flowing in its same position, then we don’t explicit it. Formally, if \( U = V_{1 \rightarrow i, \ldots, n \rightarrow j} \), then: \( U_p = \sum_k V_k \mid k \rightarrow p \).

So, in case two or more rows clash on the same final row, we perform a sum between the values. This operator is used for certificate the function calls. Indeed we have to connect the formal parameters with the actual parameters. Therefore, we have to permute the result of the function in order to keep track where the actual parameters has been substituted in place of the formal parameters.
4 Multipolynomials and abstraction

We can now proceed and introduce another fundamental concept for iSAPP: multipolynomials. This concept was firstly presented in [13]. A multipolynomial represents real bounds and its abstraction is a matrix. In the following, we assume that every polynomial have positive coefficients and it is in the canonical form.

First, we need to introduce some operator working on polynomials.

Definition 2 (Union of polynomial). Be \( p, q \) the canonical form of the polynomials \( p, q \) and let \( r, s \) polynomials, \( \alpha, \beta \) natural numbers, we define the operator \( (p \oplus q) \) over polynomials in the following way:

\[
(p \oplus q) = \begin{cases} 
\max (\alpha, \beta) + (r \oplus s) & \text{if } p = \alpha + r \text{ and } q = \beta + s. \\
\max (\alpha, \beta) \cdot X_i + (r \oplus s) & \text{otherwise if } p = \alpha X_i + r \text{ and } q = \beta X_i + s. \\
p + q & \text{otherwise} 
\end{cases}
\]

Let’s see some example. Suppose we have these two polynomials: \( X_1 + 2X_2 + 3X_4^2X_5 \) and \( X_1 + 3X_2 + 3X_4^2X_5 + X_6 \). Call them, respectively \( p \) and \( q \). We have that \( (p \oplus q) \) is \( X_1 + 3X_2 + 3X_4^2X_5 + X_6 \).

First we need to introduce the concept of abstraction of polynomial. Abstraction gives a vector representing the shape of our polynomial and how variables appear inside it.

Definition 3 (Abstraction of polynomial). Let \( p(X) \) a polynomial over \( n \) variables, \( [p(X)] \) is a column vector of size \( n+1 \) such that:

- If \( p(X) \) is a constant \( c > 1 \), then \([p(X)]\) is \( V_n^A \).
- Otherwise if \( p(X) \) is a constant 0 or 1, then \([p(X)]\) is respectively \( V_0 \) or \( V_1 \).
- Otherwise if \( p(X) = X_i \), then \([p(X)]\) is \( V_i \).
- Otherwise if \( p(X) = \alpha X_i \) (for some constant \( \alpha > 1 \)), then \([p(X)]\) is \( V_i^A \).
- Otherwise if \( p(X) = q(X) + r(X) \), then \([p(X)]\) is \( [q(X)] + [r(X)] \).
- Otherwise, \( p(X) = q(X) \cdot r(X) \), then \([p(X)]\) is \( M \cdot [q(X)] \cup M \cdot [r(X)] \).

Size of vectors is \( n+1 \) because \( n \) cells are needed for keeping track of \( n \) different variables and the last cell is the one associated to constants. We can now introduce multipolynomials and their abstraction.

Definition 4 (Multipolynomials). A multipolynomial is a tuple of polynomials. Formally \( P = (p_1, \ldots, p_n) \), where each \( p_i \) is a polynomial.

In the following, in order to refer to a particular polynomial of a multipolynomial we will use an index. So, \( P_i \) refers to the \( i \)-th polynomial of \( P \). Now that we have introduced the definition of multipolynomials, we can go on and present two foundamental operation on them: sum and composition.

Definition 5 (Sum of multipolynomials). Given two multipolynomials \( P \) and \( Q \) over the same set of variables, we define addition in the following way: \( (P \oplus Q)_i = (P_i \oplus Q_i) \).

Definition 6 (Composition of multipolynomial). Given two multipolynomials \( P \) and \( Q \) over the same set of variables, the composition of two multipolynomials is defined as the composition component-wise of each polynomial. Formally we define composition in the following way: \( (P \circ Q)_i = Q_1 \cdot P_1, \ldots, Q_n \cdot P_n \).

Abstracting a multipolynomial naturally gives a matrix where each column is the abstraction of one of the polynomials.
Definition 7. Let $P$ be a multipolynomial, its abstraction $\langle P \rangle$ is a matrix where the $i$-th column is the vector $\langle P_i \rangle$.

In the following, we use polynomials to bound size of single stacks. Since handling polynomials is too hard (i.e. undecidable), we only keep their abstraction. Similarly, we use multipolynomials to bound the size of all the stacks of a program at once. Again, rather than handling the multipolynomials, we only work with their abstractions.

5 Typing and certification

We presented all the ingredients of iSAPP and we are ready to introduce certifying rules. Certifying rules, in figure 1, associate at every command a matrix. We suppose to have $n - 1$ stacks. Notice how expressions are not typed; indeed, we don’t need to type them because their size is fixed.

\[
\begin{array}{c}
\frac{n \geq 1}{\vdash s := (c_1 \ldots c_n) : I \leftarrow V^n_s} \quad \text{(CONST-A)}
\\
\frac{i \in \mathbb{N}}{\vdash i := (c_1) : I \leftarrow V^n_i} \quad \text{(CONST-L)}
\\
\frac{\vdash \text{push}(e, S_1) : I \leftarrow (V^n_1 + V^n_1)}{\vdash (\text{Push})}
\\
\frac{\vdash C : A \quad \vdash C : B}{\vdash C : A \cup B} \quad \text{(CONCAT)}
\\
\frac{\vdash C : A}{\vdash C : B} \quad \text{(ASSIGN)}
\\
\vdash \text{def in}\ (S_1 \ldots S_n) \{C\} \{C\} : A \quad \text{(FUN)}
\\
\vdash \text{call}(f, s_1, \ldots, s_p) : I \leftarrow \langle 1 \to k, \ldots, n \to p \rangle \quad \text{(FUNCALL)}
\end{array}
\]

Figure 1 Typing rules for commands and functions

These matrices tell us about the behaviour of a command and functions. We can think about them as certificates. Certificates for commands tell us about the correlation between input and output stacks. Each column gives the bound of one output stack while each row corresponds to one input stack. Last row and column handle constants.

As example, command \texttt{(SKIP)} tells us that no stack is changed. Concatenation of commands (\texttt{CONCAT}) tells us how to find a certificate for a series of commands. The intrinsic meaning of matrix multiplication is to “connect” output of the first certificate with input of the second. In this way we rewrite outputs of the second certificate respect to inputs of the first one. Notice how the rule for \texttt{(Push)} does not have any hypothesis. Indeed, this command just increase by $+1$ (a constant) the size of the stack $S_i$. When there is a test, taking the union (i.e. maximum) of the certificates means taking the worst possible case between the two branches. The most interesting type rule is the one concerning the \texttt{(Loop)} command. The right premise acts as a guard: an $A$ on the diagonal means that there is a stack $S_i$ such that iterating the loop a certain number of time results in (the size of) $S_i$ depending affinely of itself, e.g. $|S_i| = 2 \times |S_i|$. Obviously, iterating this loop may create an exponential growth, so we stop the analysis immediately. Next, the union closure used as a certificate corresponds to a worst case scenario. We can’t know if the loop will be executed $0, 1, 2, \ldots$ times each corresponding to certificates $A^0, A^1, A^2, \ldots$ Thus we assume the worst
and take the union of these, that is the union closure. Finally, the loop correction (merge down) is here to take into account the fact that the result will also depend on the size of the stack controlling the loop (i.e. the index $k$ is the number of the variable $S_k$ controlling the loop).

Before start to prove the main theorems, let present some examples using the commands \texttt{call()}, \texttt{loop} \{ \}. In the following we will use integer number like 0, 1, 2, ... intending a constant list of size 0, 1, 2, ... This should help the reader.

\textbf{Example 8 (Addition).} We are going to present the function $+$ (a shortcut for the following function). We can check that the analysis of this function is exactly the one expected. The size of the result is the sum of the sizes of the two stacks.

```python
def addition in (S_1, S_2){
    S_3 := S_2
    loop (S_2){
        push(top(S_3), S_1)
        pop(S_3)
    }
} out(S_1)
```

The associate matrix of this function is exactly what we are expecting. Indeed, the matrix is the following one:

\[
\begin{bmatrix}
L & 0 & 0 & 0 \\
L & L & L & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & L
\end{bmatrix}
\]

\textbf{Example 9 (Multiplication).} In the following we present a way to type multiplication between a number and a variable. In the following $S_2$ is multiplied by $n$ and the result is stored in $S_1$.

```python
S_1 := 0
loop (S_2){
    S_1 = S_1 + n
}
```

typed with

\[
\begin{bmatrix}
0 & 0 & 0 \\
A & L & 0 \\
0 & 0 & L
\end{bmatrix}
\]

\textbf{Example 10 (Multiplication).} In this example we show how to type a multiplication between two variables.

```python
def multiplication in (S_1, S_2){
    S_3 := 0
    loop (S_2){
        S_3 := S_1 + S_3
    }
} out(S_1)
```

The loop is typed with the matrix

\[
\begin{bmatrix}
L & 0 & M & 0 \\
0 & L & M & 0 \\
0 & 0 & L & 0 \\
0 & 0 & 0 & L
\end{bmatrix}
\]

So, the entire function is typed with

\[
\begin{bmatrix}
L & 0 & M & 0 \\
0 & L & M & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & L
\end{bmatrix}
\]

as was expected.

\textbf{Example 11 (Subtraction).} In this example we show how to type the subtraction between two variables.
6 Semantics

Semantics of the programs generated by the grammar in def 1 is the usual and expected one. In the following we are using $\sigma$ as the state function associating to each variable a stack and to each register a letter. Semantics for boolean value is labelled with probability, while semantics for expressions ($\rightarrow_a$) is not carrying anything. In figure 2 is shown the semantic for booleans and expressions. Most of boolean operator have probability 1 and operator rand reduced to $true$ or $false$ with probability $\frac{1}{2}$. Notice how there is no semantic associated to operators $op^2()$ and $op()$. Of course, their semantics depends on how they will be implemented.

Since semantics for boolean is labelled with probability, also semantics of commands ($\rightarrow_a$) is labelled with a probability. It tells us the probability to reach a particularly final state after having execute a command from a initial state.

In figure 3 are presented the semantics for commands. Since a compile time all the functions definitions can be collected, we suppose that exists a set of defined function called DefinedFunctions where all the functions defined belong.

Since iSAPP is working on stochastic computations, in order to reach soundness and completeness respect to PP, we need to define a semantics for distribution of final states. We need to introduce some more definitions. Let $\mathcal{D}$ be a distribution of probabilities over states. Formally, $\mathcal{D}$ is a function whose type is $(Stacks \rightarrow Values) \rightarrow \alpha$. Sometimes we will use the following notation $\mathcal{D} = \{\sigma_1^{\alpha_1}, \ldots, \sigma_n^{\alpha_n}\}$ indicating that probability of $\sigma_i$ is $\alpha_i$.

We can so define semantics for distribution; the most important rules are shown in Figure 4. Since semantics for some commands computes with probability equal to 1, the correspondent rule for distributions is not presented. Unions of distributions and multiplication between real number and a distribution have the natural meaning. Notice also how all the final distributions are normalized distributions.

Here we can present our first result.

Theorem 12. A command $C$ in a state $\sigma_1$ reduce to another state $\sigma_2$ with probability equal to $\mathcal{D}(\sigma_2)$, where $\mathcal{D}$ is the distribution of probabilities over states such that $(C, \sigma_1) \rightarrow_{\mathcal{D}} \mathcal{D}$.

Proof is done by structural induction on derivation tree. It is quite easy to check that
iSAPP, a complete polytime complexity verifier tool for PP.

| Command | Semantics |
|---------|-----------|
| (skip, σ) | $\sigma(S) = \langle c_1, c_2, \ldots, c_n \rangle$ |
| (pop(S), σ) | $\sigma(S) = \langle c_2, \ldots, c_n \rangle$ |
| (push(c, S), σ) | $\sigma(S[\langle e/c \rangle])$ |
| (r := e, σ) | $\sigma(r/e)$ |
| (S := S1, σ) | $\sigma(S_1)$ |
| (S := {C}, σ) | $\sigma(S)$ |
| (loop S, σ) | $\sigma(S)$ |
| (if b Then C1 Else C2, σ) | $\sigma(S)$ |

**Figure 3** semantics of commands

| Command | Semantics |
|---------|-----------|
| (S_p := call(myfun, τ), σ) | $\sigma(S_p)$ |
| (C1, σ) | $\sigma(S_1)$ |
| (C1; C2, σ) | $\sigma(S_1)$ |
| (loop S, σ) | $\sigma(S)$ |
| (if b Then C1 Else C2, σ) | $\sigma(S)$ |
| (C1, σ) | $\sigma(S)$ |
| (C1; C2, σ) | $\sigma(S)$ |

**Figure 4** Distributions of output states
this property holds, as the rules in Figure 4 are showing us exactly this statement. The reader should also not be surprised by this property. Indeed, we are not considering just one possible derivation from $\langle C_1, \sigma_1 \rangle$ to $\sigma_2$, but all the ones going from the first to the latter.

7 Soundness

The language recognised by $iSAPP$ is an imperative language where the iteration schemata is restricted and the size of objects (here, stacks) is bounded. These are ingredients of a lot of well known ICC polytime systems. There is no surprise that every program certified by $iSAPP$ runs in probabilistic polytime.

Now we can start to present theorems and lemmas of our system. First we will focus on multipolynomial properties in order to show that the behaviour of these algebraic constructor is similar to the behaviour of matrices in our system. Finally we will link these things together to get polytime bound for $iSAPP$. Here are two fundamental lemmas. Their proofs are straightforward.

▶ **Lemma 13.** Let $p$ and $q$ two positive polynomials, then it holds that $\lceil p \oplus q \rceil = \lceil p \rceil \cup \lceil q \rceil$.

**Proof.** by induction on the size of the two polynomials. By definition the union between two polynomial is defined in 2 as the maximum of the comparable monomials. Let’s analyze the different cases:

- If $p = c_1 + r$ and $q = c_2 + s$. By induction, $\lceil r \oplus s \rceil = [r] \cup [s]$. By definition $p \oplus q$ is $\max (\alpha, \beta) + (r \oplus s)$ and so, by definition $\lceil \max (\alpha, \beta) \rceil + \lceil (r \oplus s) \rceil$. By using induction hypothesis we get $\lceil \max (\alpha, \beta) \rceil + \lceil r \rceil \cup \lceil s \rceil$. It’s clear that $\lceil \max (\alpha, \beta) \rceil$ is equal to $\lceil \max ([\alpha], [\beta]) \rceil$, since the abstraction take in account the value of the constants. We get, so $(\lceil \alpha \rceil \cup \lceil \beta \rceil) + (\lceil r \rceil \cup \lceil s \rceil)$. Notice how the abstractions of the two constants are two column vectors having 0 everywhere except for the last row. We can so rewrite the previous equation as $(\lceil \alpha \rceil + \lceil r \rceil) \cup (\lceil \beta \rceil + \lceil s \rceil)$, that is the thesis.

- If $p = \alpha X_i + r$ and $q = \beta X_i + s$. This case is very similar to the previous one.

- The last case is where the two polynomials are not comparable. In this case, the union is defined as $p + q$. There are two cases:
  - If some variables are present just in one polynomial and not in the other one, then the correspondent rows, for each single variable, is not influenced by the abstraction of the polynomial in which the variable does not appear.
  - If some variables are present in both. In this case it means that the variables appear in at least one monomial with grade greater than one or in a monomial having more than one variable. In both cases the associated abstracted value for both is $M$.

The thesis holds. This concludes the proof.

▶ **Lemma 14.** Let $P$ and $Q$ two positive multipolynomials, then it holds that $\lceil (P \oplus Q) \rceil = \lceil P \rceil \cup \lceil Q \rceil$.

**Proof.** By definition of sum between multipolynomials $\oplus$ we know that sum is defined componentwise, $(P \oplus Q)_i = (P_i \oplus Q_i)$. By lemma 13 we prove the theorem.

▶ **Lemma 15.** Let $P$ and $Q$ two positive multipolynomials (over $n$ variables) in canonical form, then it holds that $\lceil P \cdot Q \rceil \leq \lceil Q \rceil \times \lceil P \rceil$.
**Proof.** We will consider the element in position \( i,j \) and so we have: \( ([Q] \times [P])_{i,j} = \sum_k [Q]_{i,k} \times [P]_{k,j} \). We can start by making some algebraic passages:

\[
[P \cdot Q]_{i,j} = [P(Q_1, \ldots, Q_n)]_{i,j} = [P_j(Q_1, \ldots, Q_n)]_i.
\]

The equality holds because we are considering the element in the \( j \)-th column. Since we are interested at the element in position \( i \)-th we have to understand how the variable \( X_i \) (or constant) in each \( Q_k \) is substituted.

- **Case where** \( i = n + 1 \).
  - If none of the polynomials \( Q_k \) has a constant inside, then the proof is evident, since the only possible constant appearing in the result is the possible constant appearing in \( P_j \). Recall that the element in position \( (n+1, n+1) \) is \( L \) by definition, so \( \sum_k [Q]_{i,k} \times [P]_{k,j} \) contain at least \([P]_{n+1,1}\).
  - Otherwise some constants appear in some \( Q_k \). This means that the expected abstraction for the element at position \( (n + 1, j) \) may be \( A \) or \( L \). If \( L \) is the result, then is clear that and equality holds, since it means that the constant is 1. The inequality hold if the expected result is \( A \), since that on the right side we have to perform the following sum: \( \sum_k [Q]_{i,k} \times [P]_{k,j} \) and we could find an \( A \) or \( M \) value.

- **Case where** \( i < n + 1 \). In this case we are considering how the variable \( X_i \) appears. We have four possibilities:
  - If \( X_i \) does not appear in any \( Q_k \) polynomials. In this case the expected abstract value is 0. Is easy to check that this holds, since on the left side of the inequality we get 0 and on the right side we get \( \sum_k [Q]_{i,k} \times [P]_{k,j} \) that is 0, since all \([Q]_{i,k} \) are 0.
  - In the following we will consider that \( X_i \) appears in some \( Q_k \) polynomials. Call them \( \overline{Q}_X_i \). If some of the polynomials where \( X_i \) appears is substituted in some monomial of \( P_j \) of shape as \( \alpha X_p \cdot q(X) \) in place of some \( X_p \), then for sure on the right side of the inequality we will get a value \( M \). On the right side, considering \( \sum_k [Q]_{i,k} \times [P]_{k,j} \) we will multiply for sure an \( M \) value with the abstracted value for \( X_i \) of \( \overline{Q}_X_i \), where it appears. The result is so for sure \( M \).
  - Otherwise, if some of the polynomials where \( X_i \) appears is substituted in some monomial of \( P_j \) of shape as \( \alpha X_p \) (\( \alpha > 1 \)), then the expected abstract value depends on how \( X_i \) appears in \( \overline{Q}_X_i \). For all the three possible cases of \( X_i \) in \( \overline{Q}_X_i \), the abstracted value obtained on the left side is equal to the value obtained on the right side.
  - Otherwise, \( X_i \) appears is substituted in some monomial of \( P_j \) of shape as \( X_p \); then the substitution gives in output exactly the \( \overline{Q}_X_i \) substituted. The equality holds because on the right side we are going to multiply by \( L \) the abstracted value found for each \( Q_k \).

This concludes the proof.

Let’s now present the results about the probabilistic polytime soundness. The following theorem tell us that at each step of execution of a program, size of variables are polynomially correlated with size of variables in input.

**Theorem 16.** Given a command \( C \) well typed in iSAPP with matrix \( A \), such that \( \langle C, \sigma_1 \rangle \rightarrow^* \sigma_2 \) we get that exists a multipolynomial \( P \) such that for all stacks \( S_i \) we have that \( |\sigma_2(S_i)| \leq P_i(|\sigma_1(S_1)|, \ldots, |\sigma_1(S_n)|) \) and \([P]\) is \( A \).
Proof. By structural induction on typing tree. We will present just the most important cases.

- If the last rule is (Const-0), it means that we have only one stack and its size is 0. The relative vector in the matrix is a \( V^0 \). We can choose the constant polynomial 0, whose abstraction is exactly \( V^0 \). The polynomial 0 bounds the size of the stack.

- If the last rule is one of the following (Skip), (Const-A), (Const-L), (Axiom-Reg), then the proof is trivial.

- If the last rule is (Push), then we know that the size of the stack \( S \) has been increased by 1. The associated vector is a column vector having \( L \) on the \( i \)-th row and \( L \) on the last line. The correspondent polynomial, \( X_i + 1 \), is the correct bounding polynomial for the \( i \)-th stack.

- If the last rule is (Subtyp), then by induction on the hypothesis we can easily find a new polynomial bound.

- If the last rule is (Asgn), then we know that the size of the \( i \)-th stack is equal to the size of the stack \( j \)-th. So, the polynomial bounding the size of the \( i \)-th stack uses at least two variables and the correct one is \( P_i(X_i, X_j) = X_j \).

- If the last rule is (IfThen), then by applying induction hypothesis on the two premises we have multipolynomial bounds \( Q, R \) such that \([Q] = A\) and \([R] = B\). By lemma 14 we get the thesis.

- If the last rule is (Fun), then by applying the induction hypothesis on the premise we directly prove the thesis.

- If the last rule is (FunCall), then by applying the induction hypothesis on the premise we have a polynomial bound \( Q \) such that \([Q] = A\). For all the stacks different from the \( i \)-th, the bounding polynomial is trivial, while for the stack \( S_i \) depends on the result of the function call.

The function return value stored in the \( j \)-th stack used inside the function. According to the actual parameters, the actual polynomial bound is different from the one retrieving by applying the induction hypothesis.

- If last rule is (Concat), then by lemma 15 we can easily conclude the thesis.

- If last rule is (Loop), we are in the following case; so, \( A \) is \((B^\cup)^{\downarrow k}\). The typing and the associated semantic are the following:

\[
\begin{align*}
& \vdash C_1 : B \\
& \forall i, (B^{\cup})_{i,k} < A \\
& \vdash \text{loop} S_k \{C_1\} : (B^{\cup})^{\downarrow k} \quad \text{(Loop)} \\
& \langle C_1, \sigma \rangle \rightarrow^* \sigma_1 \\
& \langle S_k, \sigma \rangle \rightarrow_n \langle c_1 \ldots c_n \rangle \\
& \langle C_1, \sigma_{n-1} \rangle \rightarrow^* \sigma_n \\
& \langle \text{loop} S_k \{C_1\}, \sigma \rangle \rightarrow^\text{loop} \sigma_n
\end{align*}
\]

We consider just the case where \( n > 0 \), since the other one is trivial. By induction on the premise we have a multipolynomial \( P \) bound for command \( C_1 \) such that its abstraction is \( B \). If \( P \) is a bound for \( C_1 \), then \( P \cdot P \) is a bound for \( C_1 ; C_1 \) and \( (P \cdot P) \cdot P \) is a bound for \( C_1 ; C_1 ; C_1 \) and so on. All of these are multipolynomial because we are composing multipolynomials with multipolynomials.

By lemma 15 and knowing that \([P]\) is \( B \) we can easily deduce to have a multipolynomial bound for every iteration of command \( C_1 \). In particularly by lemma 14 we can easily sum up everything and find out a multipolynomial \( Q \) such that \([Q]\) is \((B^{\cup})^{\downarrow k}\). This means that further iterations of sum of powers of \( P \) will not change the abstraction of the result.

So, for every iteration of command \( C_1 \) we have a multipolynomial bound whose abstraction cannot be greater than \((B^{\cup})^{\downarrow k}\). So, we study the worst case; we analyse the matrix \( B^{\cup} \).
Side condition on (Loop) rule tells us to check elements on the main diagonal. Recall that by definition of union closure, elements on the main diagonal are supposed to be greater than 0. We required also to be less than A. Let’s analyse all the possibilities of an element in position i, i:

- Value 0 means no dependencies. If value is L it means that \( Q_i \) concrete bound for such column has shape \( S_i + r(\overline{S}) \), where \( S_i \) does not appear in \( r(\overline{S}) \). Iteration of such assignment gives us polynomial bound increment of the value of variable \( S_i \).
- If value is A could means that \( Q_i \) concrete bound for such column has shape \( \alpha S_i + r(\overline{S}) \) (for some \( \alpha > 1 \)), where \( S_i \) does not appear in \( r(\overline{S}) \). Iteration of such assignment lead us to exponential blow up on the size of \( S_i \).
- Otherwise value is M. This case is worse than the previous one. It’s evident that we could have exponential blow up on the size of \( S_i \).

The abstract bound \( B \) is still not a correct abstract bound for the loop because loop iteration depends on some variable \( S_k \). We need to adjust our bound in order to keep track of the influence of variable \( S_k \) on loop iteration.

We take multipolynomial \( Q \) because we know that further iterations of the algorithm explained before will not change its abstraction \([Q]\). Looking at i-th polynomial of multipolynomial \( Q \) we could have three different cases. We behave in the following way:

- The polynomial has shape \( S_i + p(\overline{S}) \). In this case we multiply the polynomial \( p \) by \( S_k \) because this is the result of iteration. We substitute the i-th polynomial with the canonical form of polynomial \( S_i + p(\overline{S}) \cdot S_k \).
- The polynomial has shape \( S_i + \alpha \), for some constant \( \alpha \). In this case we substitute with \( S_i + \alpha \cdot S_k \).
- The polynomial has shape \( S_i \) or \( S_i \) does not appear in the polynomial. We leave as is.

In this way we generate a new multipolynomial, call it \( R \). The reader should easily check that these new multipolynomial expresses a good bound of iterating \( Q \) a number of times equal to \( S_k \). Should also be quite easy to check that \(|R| \) is exactly \((B^\cup)^{ik}\). This concludes the proof.

Polynomial bound on size of stacks is not enough; we should also prove polynomiality of number of steps. Since all the programs generated by the language terminate and all the stacks are polynomially bounded in their size, the theorem follows straightforward.

- **Theorem 17.** Let \( C \) be a command well typed in iSAPP and \( \sigma_1, \sigma_n \) state functions. If \( \pi : \langle C_1, \sigma_0 \rangle \rightarrow^n \sigma_n \), then there is a polynomial \( p \) such that \(|\pi| \) is bounded by \( p(\sum_{i} |\sigma_0(S_i)|) \).

**Proof.** By induction on the associated semantic proof tree.

### 7.1 Probabilistic Polynomial Soundness

Nothing has been said about probabilistic polynomial soundness. Theorems 16 and 17 tell us just about polytime soundness. Probabilistic part is now introduced. We will prove probabilistic polynomial soundness following idea in [6], by using “representability by majority”.

- **Definition 18 (Representability by majority).** Let \( \overline{\sigma_0}[S/n] \) define as \( \forall S, \sigma_0(S) = n \). Then \( C \) is said to represent-by-majority a language \( L \subseteq \mathbb{N} \) iff:
  1. If \( n \in L \) and \( \langle C, \overline{\sigma_0}[S/n] \rangle \rightarrow \mathcal{D}, \) then \( \mathcal{D}(\sigma_0) \geq \sum_{m>0} \mathcal{D}(\sigma_m) \);
  2. If \( n \notin L \) and \( \langle C, \overline{\sigma_0}[S/n] \rangle \rightarrow \mathcal{D}, \) then \( \sum_{m>0} \mathcal{D}(\sigma_m) > \mathcal{D}(\sigma_0) \).
Observe that every command $C$ in iSAPP represents by majority a language as defined in [18]. In literature [11] it is well known that we can define PP by majority. We say that the probability error should be at most $\frac{1}{2}$ when we are considering string in the language and strictly smaller than $\frac{1}{2}$ when the string is not in the language. So we can easily conclude that iSAPP is sound also respect to probabilistic polytime.

8 Probabilistic Polynomial Completeness

There are several ways to demonstrate completeness respect to some complexity class. We will show that by using language recognised by our system we are able to encode Probabilistic Turing Machines (PTM). We will not be able to encode all possible PTMs but all the ones with particularly shape. This lead us to reach extensional completeness. For every problem in PP there is at least an algorithm solving that problem that is recognised by iSAPP.

A Probabilistic Turing Machine [7] can be seen as non deterministic TM with one tape where at each iteration are able to flip a coin and choose between two possible transition functions to apply.

In order to encode Probabilistic Turing Machines we will proceed with the following steps:
- We show that we are able to encode polynomials. In this way we are able to encode the polynomial representing the number of steps required by the machine to complete.
- We encode the input tape of the machine.
- We show how to encode the transition $\delta$ function.
- We put all together and we have an encoding of a PTM running in polytime.

Should be quite obvious that we can encode polynomials in iSAPP. Grammar and examples [8][9][10][11] give us how encode polynomials.

We need to encode the tape of our PTMs. We subdivide our tape in three sub-tapes. The left part $tape_l$, the head $tape_h$ and the right part $tape_r$. $tape_r$ is encoded right to left, while the left part is encoded as usual left to right.

Let’s move on and present the encoding of transition function of PTMs. Transition function of PTMs, denoted with $\delta$, is a relation $\delta \subseteq (Q \times \Sigma) \times (Q \times \Sigma \times \{←, ↓, →\})$. Given an input state and a symbol it may give in output more tuples of state, a symbol and a direction of the head (left, no movement, right).

In the following we are going to present two procedures to encode movements of the head. It is really important to pay attention on how we encode this operations. Recall that a PTM loops the $\delta$ function and our system requires that the matrix certifying/typing the loop needs to have values of the diagonal less than $A$.

**Definition 19 (Move head to right).** Moving head to right means to concatenate the bit pointed by the head to the left part of the tape; therefore we need to retrieve the first bit of the right part of the tape and associate it to the head. Procedure is presented as algorithm [1] call it $\text{MoveToRight()}$.

The first column of the matrix represents dependencies for variables $tape_l$, the second represents $tape_h$, third is $tape_r$, forth is $M_{state}$ and finally recall that last column is for constants. In the following, columns of matrices are ordered in this way.

Similarly we can encode the procedure for moving the head to left and the possibility of not moving at all, that is a skip command. So, the $\delta$ function is then encoded in the standard way by having nested If-Then-Else commands, checking the value of $\text{rand}$, the state, the symbol on a tape and performing the right procedure.
Algorithm 1 Move head to right

push(top(tape_h), tape_l)
tape_h := ()
push(top(tape_l), tape_h)
pop(tape_l)

Using typing rules we are able to type the algorithm with the following matrix:

\[
\begin{bmatrix}
L & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & L & 0 & 0 \\
0 & 0 & 0 & L & 0 \\
L & A & 0 & 0 & L
\end{bmatrix}
\]

Algorithm 2 Prototype of encoded \(\delta\) function

if rand then
  if equal?(M_state, 1) then
    else
      if equal?(M_state, 2) then
        ... else
          ... end if
        end if
      end if
  else
    ... end if
else
  ... end if
Every matrix could be seen as adjacency matrix of a graph.

As example, the following matrix $A$:

$$
\begin{bmatrix}
L & 0 & 0 & 0 \\
L & L & L & 0 \\
L & 0 & 0 & 0 \\
0 & M & L & L
\end{bmatrix}
$$

has its own representation in the graph on the right side.

Figure 5 Example of graph representing a matrix in iSAPP

Finally, we have to put the encoded $\delta$-function inside a loop. The machine runs in a polynomial number of steps. Since the encoded $\delta$-function is typed with the matrix presented in Alg. 2 we can easily see that the union closure of that matrix fits the constraints of the typing rule of loop $\{\}$. We can therefore conclude that we can encode Probabilistic Turing Machine working in polytime.

9 Polynomiality

In this last session we will discuss why iSAPP is a feasible analyser. We already shown that is sound, so is able to understand whenever a program does not run in Probabilistic Polynomial Time. Moreover, is also complete, respect to PP; this means that iSAPP is able to recognise a lot of programs. At least one for each problem in PP. The final question has to do with the efficiency of our system: “how much time does it take iSAPP to check a program?” Can be shown that iSAPP is running in polytime respect to the number of variables used.

Since the typing rules are deterministic, the key problems lays on the rule (Loop').

$$
\vdash C_1 : A \quad \forall i, (A \cup i)_{i,i} < A \\
\vdash \text{loop } S_k \{ C_1 \} : (A \cup i)^k 
$$

(LOOP)

It is not trivial to understand how much it takes a union closure to be performed. While all the typing rules for all the other commands and expressions are trivial, the one for loop needs some more explanations. By definition, $A \cup i$ is defined as $\cup i A^i$.

In the example in Figure 5 we can easily check that $C$ flows in $S_1$ with $M$ in one step. So, $A^2$ have $M$ in position $(4, 1), (4, 2), (4, 3)$. Indeed, by using the rule of our algebra we can see how dependencies flows in the graph.

How many unions have to be performed in order to calculate $\cup_i A^i$? In order to answer to this question, we can prove the following theorem.

Theorem 20 (Polynomiality). Given a squared matrix $A$ of size $n$ and $B = \bigcup A^i$, we get that $B = \bigcup_{i<n^2} A^i$. Union closure can be calculated by considering just the first $n^2$ matrix power.

Proof. Here is the scratch of the proof. Since the matrix is an encoding of a flow graph, we can see the matrix as a graph of dependencies between stacks size. Recall that the union
is component-wise, so we can focus on a singular element of a matrix. Given two nodes $S_1$ and $S_2$ of our graph, let’s check all the possibilities:

- The expected value is $M$. If so, after no more than $n$ iteration of $A$ we should have found it. If not, there are no possibilities to have $M$ in that position. After $n$ iteration, the information has flown through all the nodes.

- The expected value is $A$. We need to iterate more than $n$ times. Indeed $A$ value can be found also by adding $L + L$. In the flow-graph relation, this means finding two distinct paths from node $S_1$ to $S_2$. This can be easily done by encoding two paths in one. By generating all the possible pairs of nodes, we can easily see that the number of steps to find, if exists, two distinct paths takes $n^2$ number of steps (number of all pairs).

- If after $n^2$ steps no $M$ or $A$ value has been found, the maximum value found is the correct one. Indeed, if no dependence has been found or if just a linear dependence has been found, no further iteration could change the final value.

10 Conclusions

We presented an ICC system characterising the class $PP$. There are several improvements respect to the known systems in literature. We can catalogue them in two sets. First, we extend the known system to probabilistic computations, being able to characterise $PP$. Since the typing requires polynomial time, it is feasible to use iSAPP as a static analyser for complexity. The typing/certificate gives also information about the polynomial bound. On the other hand, respect to sequential computations, we presented a finer analysis. iSAPP works over a concrete language and takes care of constants and function calls. For all of these reasons, we are able to show a program that cannot be typed correctly by Kristiansen and Jones [9].

Algorithm 3 Example of recognised program

\[
\begin{array}{l}
\text{loop } (S_1) \{ \\
\quad S_2 := 0 * S_2 \\
\} \\
\end{array}
\]

That is typed with the identity matrix $I_{2^2} \leftarrow V^0$. For multiplication we use the implementation in def 8.

Since every constant is abstracted as a variable in [9], they cannot for sure recognise that this program runs in polytime and for this reason this program should be rejected. Once abstracted it is impossible to know the value of the constant. Of course, everything depends on how the abstraction is made. In general, every program which deals with constants could appear problematic in [9] [8]: At least, for a lot of programs, their bounds are bigger. Moreover, as they wrote in [8]: “Note that no procedure for inferring complexity will be complete for $L_{concrete}$”, while our procedure is sound and complete for our concrete language.

Finally we would like to point out some future direction:

- Integrating the analysis with new features in order to capture more programs.
- Apply our analysis to a more generic imperative programming language.
- Extending the algebra in such way that the associated certificates would tell more detailed information about the polynomial bounding the complexity.
- Make a finer analysis in order to be sound and complete for $BPP$. 

\[\]
References

1 Sanjeev Arora and Boaz Barak. *Computational Complexity, A Modern Approach*. Cambridge University Press, 2009.

2 Stephen Bellantoni and Stephen A. Cook. A new recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2:97–110, 1992.

3 Amir M. Ben-Amram, Neil D. Jones, and Lars Kristiansen. Linear, polynomial or exponential? complexity inference in polynomial time. In *Proceedings of the 4th conference on Computability in Europe: Logic and Theory of Algorithms*, CiE ’08, pages 67–76, Berlin, Heidelberg, 2008. Springer-Verlag.

4 G. Bonfante, J.-Y. Marion, and J.-Y. Moyen. Quasi-interpretations a way to control resources. *Theoretical Computer Science*, 412(25):2776 – 2796, 2011.

5 Alan Cobham. The intrinsic computational difficulty of functions. In Y. Bar-Hillel, editor, *Logic, Methodology and Philosophy of Science, proceedings of the second International Congress, held in Jerusalem, 1964*, Amsterdam, 1965. North-Holland.

6 Ugo Dal Lago and Paolo Parisen Toldin. A higher-order characterization of probabilistic polynomial time. In R. Peña, M. van Eekelen, and O. Shkaravska, editors, *Proceedings of 2nd International Workshop on Foundational and Practical Aspects of Resource Analysis*, FOPARA 2011, volume 7177 of LNCS. Springer, 2011. To be appeared in.

7 John Gill. Computational complexity of probabilistic turing machines. *SIAM J. Comput.*, 6(4):675–695, 1977.

8 Neil D. Jones. Logspace and ptime characterized by programming languages. *Theoretical Computer Science*, 228:151–174, October 1999.

9 Neil D. Jones and Lars Kristiansen. A flow calculus of mwp-bounds for complexity analysis. *ACM Trans. Comput. Logic*, 10(4):28:1–28:41, August 2009.

10 Lars Kristiansen and Neil D. Jones. The flow of data and the complexity of algorithms. In *Proceedings of the First international conference on Computability in Europe: new Computational Paradigms*, CiE’05, pages 263–274, Berlin, Heidelberg, 2005. Springer-Verlag.

11 Daniel Leivant. Stratified functional programs and computational complexity. In *Principles of Programming Languages, 20th International Symposium, Proceedings*, pages 325–333. ACM, 1993.

12 Daniel Leivant and Jean-Yves Marion. Ramified recurrence and computational complexity II: Substitution and poly-space. In Leszek Pacholski and Jerzy Tiuryn, editors, *Computer Science Logic, 9th International Workshop, Proceedings*, volume 933 of LNCS, pages 486–500. 1995.

13 R. Metnani and J.-Y. Moyen. Equivalence between the mwp and Quasi-Interpretations analysis. In J.-Y. Marion, editor, *DICE’11*, April 2011.

14 Albert R. Meyer and Dennis M. Ritchie. The complexity of loop programs. In *Proceedings of the 1967 22nd national conference*, ACM ’67, pages 465–469, New York, NY, USA, 1967. ACM.