Invariant Checking for Programs with Procedure Calls*

Guillem Godoy[1] and Ashish Tiwari[2]

[1] LSI Department, Technical University of Catalonia
Jordi Girona, 1-3 08034 Barcelona, Spain, ggodoy@lsi.upc.edu
[2] SRI International, 333 Ravenswood Ave, Menlo Park, CA, U.S.A
Tel:+1.650.859.4774, Fax:+1.650.859.2844, tiwari@csl.sri.com

Abstract. Invariants are a crucial component of the overall correctness of programs. We explore the theoretical limits for doing automatic invariant checking and show that invariant checking is decidable for a large class of programs that includes some recursive programs. The proof uses known results like the decidability of Presburger arithmetic and the semilinearity of the Parikh image of a regular language. Removing some of the restrictions on the program model leads to undecidability of the invariant checking problem.

1 Introduction

The ability to generate reliable and correct software depends crucially on the development of tools for automatically verifying the correctness of programs. Modern software development tools support automatic program analysis, but only to a limited extent. Extending these analyses to richer and deeper properties of programs is an active area of research.

Invariants are a crucial component of the overall correctness of programs. An invariant is simply an expression that evaluates to "true" on all executions (paths) of the program. The problem of checking whether a given expression is an invariant is undecidable in general. However, there are simplified program models for which invariant checking is decidable, even efficiently. These decidability results are important in two respects: they help in developing efficient analysis engines and understanding the causes of undecidability, which in turn is useful for identifying places where any analysis engine will necessarily be incomplete.

There are several results on the decidability of invariant checking for restricted program models. These decidability results are parameterized by three choices: (a) the program model, (b) the theory of the expression language used in the program model, and (c) the form of the assertion. A common assumption on the program model is that there are no procedure call nodes [6, 11, 12,

* The first author was supported by Spanish Ministry of Education and Science through the FORMALISM project (TIN2007-66523) and the LOGICTOOLS-2 project (TIN2007-68093-C02-01). The second author was supported in part by NSF grants CNS-0720721 and CNS-0834810 and NASA grant NNX08AB95A.

7, 8]. When procedure call nodes are allowed, it is commonly assumed that variables can take only *finitely* many values [4, 3, 1]. When (restricted) procedure call nodes are allowed, and also infinite domains for data values are allowed [16, 13, 14, 10, 9], then, it is assumed that assertions are always equality of two program expressions. In other words, there are no results for the case when procedure call nodes, infinite data values, and disequality assertions are all allowed. However, note that checking invariance of *disequalities* is equally important; for example, for alias analysis.

In this paper, we show that invariant checking is decidable in a setting which allows (a) recursive procedure call nodes in the program model, (b) infinite domains for values of variables, and (c) any Boolean combination of equality and disequality of program expressions as the assertion. Specifically, we define a programming model in which a program contains a *finite* number of program variables, but each variable takes values over the *infinite* domain of (uninterpreted first-order) terms. The program variables are updated by assignments and the control flow structure consists of non-deterministic conditionals, loops, and (possibly recursive) procedure calls. We identify a subclass of programs in this programming model for which the problem of checking if an equality or disequality (or any Boolean combination thereof) is an invariant is decidable.

In the process of obtaining the main result, we also show that the following problem is solvable: given $N + M$ substitutions, $\sigma_1, \ldots, \sigma_N, \beta_1, \ldots, \beta_M$, $N + M$ integer variables, $n_1, \ldots, n_N, m_1, \ldots, m_M$, and terms x, y, determine if there is a value for the $N + M$ integer variables in a given semilinear set such that

$$\sigma_N^{n_N} \ldots \sigma_1^{n_1}(x) = \beta_M^{m_M} \ldots \beta_1^{m_1}(y).$$

This result can be of independent interest.

2 Preliminaries

Let $\mathcal{T}(\Sigma, \{\boldsymbol{X}\})$ be the set of all the terms constructed over a fixed finite signature Σ and a set of variables $\boldsymbol{X}$. The root symbol of a term t is denoted by $\texttt{root}(t)$. The *positions* $\texttt{Pos}(t)$ in a term t are sequences of positive integers (ϵ, the empty sequence, is the root position). A subterm of t at position p is written $t|_p$. The concatenation of the positions p and q is denoted as $p.q$. A substitution is a mapping from a set of variables $\boldsymbol{X}$ to $\mathcal{T}(\Sigma, \{\boldsymbol{X}\})$. We denote substitutions by σ, θ and $\sigma(t)$ denotes the term obtained by replacing every variable in t by its image by σ. Given substitutions σ and θ, their composition is denoted by juxtaposition $\sigma\theta$, and is defined by $\sigma\theta(x) := \sigma(\theta(x))$.

A *linear* set is any subset of $\mathbb{N}^k$ that can be written in the form $\{\boldsymbol{c_0} + \sum_{i=1}^{n} \alpha_i \boldsymbol{c_i} \mid \alpha_i \in \mathbb{N}\}$ for some fixed $n + 1$ vectors $\boldsymbol{c_0}, \ldots, \boldsymbol{c_n}$ in $\mathbb{N}^k$. A *semilinear* set is a finite union of *linear* sets.

The *Parikh image* of a word $w \in \Sigma^*$ is a vector in $\mathbb{N}^{|\Sigma|}$ that contains the number of occurrences in w of each symbol in Σ. For example, if $\Sigma = \{a_1, a_2\}$ and $w = a_1a_2a_1a_1$, then the Parikh image of w is $\langle 3, 1\rangle$. It is well known that Parikh image of a regular (even context-free) language is semilinear [15].

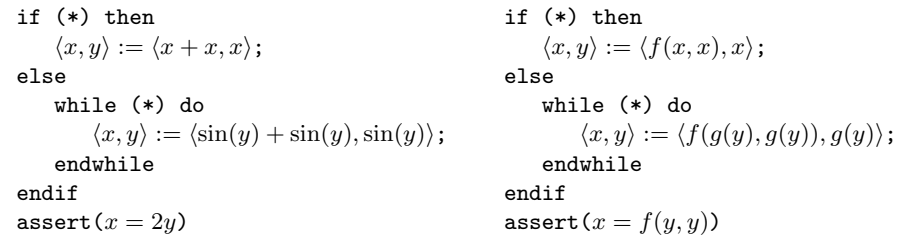

Fig. 1. A simple program (left) and its abstracted version (right). The abstract version is obtained by replacing the interpreted symbols $+, \sin$ by uninterpreted symbols f, g respectively.

A Presburger arithmetic formula is a (possibly quantified) first-order formula over predicate symbols $=$ and $>$, and with terms constructed using the binary symbol $+$ and constant symbols $\mathbb{N}$ (that is, linear arithmetic expressions are allowed). Presburger arithmetic formulas are interpreted over the natural numbers in the standard way. If ϕ is a sentence in Presburger arithmetic, then $\models_{\mathbb{N}} \phi$ denotes validity of ϕ. We will use $\models$ to denote validity in the pure theory of equality over uninterpreted symbols (occasionally, this theory combined with Presburger arithmetic).

Every semilinear set can be represented using a Presburger formula. Hence, it follows that for every regular language L, there exists a Presburger formula ϕ_L whose solutions coincide with the Parikh image of L. In fact, the size of the formula ϕ_L can be bounded using the following result from Seidl et. al. [17].

Theorem 1 (Seidl et. al. [17], Theorem 1). *For any nondeterministic finite automaton A, an existential Presburger formula ϕ_A for the Parikh image of the language $\mathtt{L}(A)$ of A can be constructed in time $\mathtt{O}(|A|)$, where $|A|$ is the number of states plus the number of transitions in A.*

3 Invariant Checking and Related Work

We illustrate the main ideas related to invariant checking via a simple example. Consider the program in Figure 1 (left). Given an assertion, say $x = 2y$, at the end of the program, the problem of invariant checking seeks to find out if the assertion is an invariant of the program, that is, if it evaluates to true for all executions of the program.

Since invariant checking is undecidable for general programs, often the program is abstracted and invariants are checked on the abstracted program. The program in Figure 1 (left) has already abstracted away the actual conditions (that were present in some original program) and replaced them with nondeterministic choices (*). This new program has more behaviors, and hence if an expression is an invariant for this new program, it will be an invariant of the original. For the above example, it is easy to see that the assertion at the end of the program holds under all possible executions of this nondeterministic program.

The left program can be abstracted further by replacing the interpreted functions, + and sin, by uninterpreted functions, say f and g. Again, this is a sound abstraction – the new abstracted program, shown on the right-hand side of Figure 1, has more behaviors. The process of abstraction is attractive since it can give a program that lies in a class of programs for which invariant checking is decidable.

The decidability of the assertion checking problem is parameterized by three choices: (a) the program model, (b) the theory of the expression language used in the program model, and (c) the form of the assertion. We briefly describe the common choices made for obtaining decidability and point to related work.
(a) The program model: First note that including conditional branches in programs quickly leads to undecidability of invariant checking [12, 11]. Hence, a commonly studied program model is one that contains only assignments, *non-deterministic* conditionals and *non-deterministic* loops. The two programs in Figure 1 fall into this category. Since interprocedural analysis is more difficult, procedure call nodes are often disallowed in the program model [6, 11, 12, 7, 8]. For this program model, invariant checking is decidable when the expressions are terms over uninterpreted symbols and assertions are term equalities [6, 8]. For example, the invariant checking problem for the program in Figure 1 (right) falls into this class. However, the invariant checking problem becomes undecidable if we consider disequality assertions, such as $x \neq y$, rather than equality assertions [18]. It remains decidable when the assertion is a disjunction of conjunctions of equalities [8]. For the same program model, the above results also generalize to several other expression languages; the reader is referred to [8] for details.
A useful extension of the program model is obtained by including procedure calls. If the procedure calls are not recursive, the problem can be reduced to the original one by just inlining the procedures. When recursive procedure calls are allowed, the problem becomes more complex, and there are very few results on the decidability of invariant checking [9].
(b) The expression language: It is commonly assumed that infinite data types have been abstracted into finite types and this assumption forms the starting point for several investigations, especially when the program model allows recursive procedure calls [4, 3, 1]. Our work, however, takes a complementary path. We focus on *restricted and simpler control flow paths, but keep the data type domains infinite.*
(c) The form of assertion: There are some results for the case when procedure calls and infinite data types are both allowed [16, 13, 14, 10, 9], but in all these cases, as well as in most of the other works, assertions are restricted to equality between program expressions.

In contrast to all the above mentioned works, we consider equality and disequality assertions in the presence of recursive procedure calls and infinite data types. Since the Post Correspondence Problem (PCP) can be reduced to checking a disequality assertion in a non-deterministic loop containing non-deterministic

```
main () {
   ⟨x, y⟩ := ⟨f(x, x), x⟩;
   call P ;
   y := f(y, y);
   assert(x = y);
}
```

```
P() {
   if (*) then // do nothing
   else
      x := g(x); call P ; y := g(y);
   endif
}
```

Fig. 2. A simple program containing a main procedure (left) and a subprocedure (right).

conditionals (see Figure 3 and [18]), we have to restrict the program model – by disallowing conditionals within loops – to achieve decidability.

Summary of the Main Ideas and Results.

Consider the recursive program in Figure 2. We will view an assignment block $\langle x_1, \ldots, x_n \rangle := \langle s_1, \ldots, s_n \rangle$ as the substitution $\sigma_s = \{x_1 \mapsto s_1, \ldots x_n \mapsto s_n\}$. In the program in Figure 2, the five assignment blocks correspond to the following five substitutions:

$$\sigma_1 = \{x \mapsto f(x,x), y \mapsto x\} \quad \sigma_2 = \{x \mapsto x, y \mapsto f(y,y)\} \quad \sigma_3 = \{x \mapsto x, y \mapsto y\}$$
$$\sigma_4 = \{x \mapsto g(x), y \mapsto y\} \quad \sigma_5 = \{x \mapsto x, y \mapsto g(y)\}$$

The assertion $x = y$ holds at the end of procedure `main` iff x and y have equal values at that point on all program paths. This is equivalent to deciding whether

$$\sigma_1 \underline{\sigma_4^N \sigma_3 \sigma_5^N} \sigma_2(x) = \sigma_1 \underline{\sigma_4^N \sigma_3 \sigma_5^N} \sigma_2(y), \qquad \text{for every } N \geq 0 \tag{1}$$

Note that the underlined composition of substitutions capture the effect of the recursive procedure `P`, and N represents the total number of recursive calls to `P` in a certain execution.

Consider the negation of Condition 1:

$$\sigma_1 \underline{\sigma_4^N \sigma_3 \sigma_5^N} \sigma_2(x) \neq \sigma_1 \underline{\sigma_4^N \sigma_3 \sigma_5^N} \sigma_2(y), \qquad \text{for some } N \geq 0 \tag{2}$$

We view the above literal as an instance of the general dis-unification problem: find all numerical values for the variables $i_1, \ldots, i_n, j_1, \ldots, j_m$ such that

$$\sigma_1^{i_1} \ldots \sigma_n^{i_n}(x) \neq \beta_1^{j_1} \ldots \beta_m^{j_m}(y). \tag{3}$$

where σ_i, β_j's are given substitutions. We prove that the solutions for this dis-unification problem can be expressed with a Presburger arithmetic formula with free variables $i_1, \ldots, i_n, j_1, \ldots, j_m$. As a consequence, we prove decidability of any Boolean formula whose atoms are of Form 3, and in particular, we prove decidability of our original invariant checking problem.

In our example, we construct the Presburger sentence equivalent to Equation 1 by constructing a Presburger formula ϕ for the following "more general" set:

$$\{\langle i_1, \ldots, i_5, j_1, \ldots, j_5 \rangle \in \mathbb{N}^{10} \mid \sigma_1^{i_1} \sigma_4^{i_4} \sigma_3^{i_3} \sigma_5^{i_5} \sigma_2^{i_2}(x) \neq \sigma_1^{j_1} \sigma_4^{j_4} \sigma_3^{j_3} \sigma_5^{j_5} \sigma_2^{j_2}(y)\}$$

The Presburger formula ϕ is obtained in two steps. In the first step, we construct a non-deterministic finite automaton A on an alphabet with 10 symbols such that the Parikh image of the language accepted by A is equal to the above set. This construction is given in Section 5. In the second step, we construct a Presburger formula representing the Parikh image of A.

We characterize the class of programs for which the invariant checking problem can be decided using our approach. A program is in this class if the effect of all its execution paths can be described as a finite union of expressions of the form $\sigma_1^{i_1} \ldots \sigma_n^{i_n}$, with linear conditions relating the i_j's. We also define a general program model and a syntactic subclass, called *Sloopy Programs*, that falls in the decidable class. An important restriction is that Sloopy Programs disallow conditionals within loops.

The outline for the rest of the paper is as follows. In Section 4 we define the notion of *parameterized substitutions*, present our program model and the subset of Sloopy Programs. We show that parameterized substitutions are sufficient to finitely represent the semantics of Sloopy Programs. Thus, the invariant checking problem is reduced to deciding conditions of the form of Equation 1. In Section 5 we show that these conditions are decidable. Finally, we put forward our conclusions in Section 7 and discuss avenues for future research.

4 Program Model, Semantics, and Parameterized Substitutions

The programs considered in this paper do not have input and are non-deterministic. The semantics of a concrete execution of a program is the final value of its variables, which can be viewed as a substitution σ. The semantics of a program can then be defined as the set of substitutions corresponding to all its possible executions. We are interested in programs whose semantics is finitely representable in some formalism. In this setting, the following definition will be useful.

Definition 1. *A* parameterized substitution*, $\theta(n_1, \ldots, n_N)$, or $\theta(\boldsymbol{n})$ in short, is an expression of the form*

$$\sigma_N^{n_N} \cdots \sigma_3^{n_3} \sigma_2^{n_2} \sigma_1^{n_1}$$

where each n_i is a variable (ranging over the natural numbers) and each σ_i is a substitution. A parameterized substitution is succinctly written as $\boldsymbol{\sigma^n}$.

An instance of $\theta(\boldsymbol{n})$ is a substitution obtained by fixing the valuation for the variables $\boldsymbol{n}$. If Θ is a set of parameterized substitutions, then the set $\texttt{Instances}(\Theta)$ *is defined as*

$$\texttt{Instances}(\Theta) := \{\sigma \mid \sigma := \theta(\boldsymbol{c}),\ \ \boldsymbol{c} \in \mathbb{N}^N,\ \ \theta(\boldsymbol{n}) \in \Theta\}$$

An extended parameterized substitution *is a pair $(\theta(\boldsymbol{n}); \chi(\boldsymbol{n}))$ where $\theta(\boldsymbol{n})$ is a parameterized substitution and $\chi(\boldsymbol{n})$ is a Presburger formula with free variables $\boldsymbol{n}$. If Θ is a set of extended parameterized substitutions, then the set*

$\texttt{Instances}(\Theta)$ *is defined as*

$$\texttt{Instances}(\Theta) := \{\sigma \mid \sigma := \theta(\boldsymbol{c}), \quad \boldsymbol{c} \in \mathbb{N}^N, \quad \models_{\mathbb{N}} \chi(\boldsymbol{c}), \quad (\theta(\boldsymbol{n}); \chi(\boldsymbol{n})) \in \Theta\}$$

For a given class of programs, we say that its semantics is effectively representable with extended parameterized substitutions *if, for every program P in the class, a finite set Θ of extended parameterized substitutions can be computed such that the semantics, $[\![P]\!]$, of P is equal to* $\texttt{Instances}(\Theta)$.

Note that a parameterized substitution θ can be written as an extended parameterized substitution $(\theta; \texttt{true})$.

Program Model. We define a general class of programs syntactically and then identify its subclass that is effectively representable with extended parameterized substitutions. Let X be a finite set of variables, called program variables. A *program* is a finite ordered list of *procedures*, $\langle P_0, P_1, \ldots, P_k \rangle$, where a *procedure* is a string defined by the following grammar:

$$\begin{aligned} P ::= \; & \boldsymbol{X} := \boldsymbol{t} \;\mid\; P \;;\; P \;\mid \\ & \texttt{if}\ (*)\ P\ \texttt{else}\ P\ \texttt{endif} \;\mid\; \texttt{while}\ (*)\ P\ \texttt{endwhile} \;\mid\; \texttt{call}\ n \end{aligned}$$

where $n \in \{0, \ldots, k\}$ is an index (referring to the procedure at the n-th position in the ordered list above) and $\boldsymbol{t}$ is a vector of terms (of size exactly equal to $|\boldsymbol{X}|$).

We next define a subclass of programs called Sloopy Programs that only contain *s*imple *loop*s. A Sloopy Procedure is defined as follows:

$$\begin{aligned} \texttt{Intr}_\texttt{n} ::= \; & \boldsymbol{X} := \boldsymbol{t} \;\mid\; \texttt{while}\ (*)\ \boldsymbol{X} := \boldsymbol{t}\ \texttt{endwhile} \;\mid\; \texttt{Intr}_\texttt{n}\ ;\ \texttt{Intr}_\texttt{n} \;\mid \\ & \texttt{if}\ (*)\ \texttt{Intr}_\texttt{n}\ \texttt{else}\ \texttt{Intr}_\texttt{n}\ \texttt{endif} \;\mid\; \texttt{call}\ m \text{ where } m > n \\ \texttt{SProc}_\texttt{n} ::= \; & \texttt{Intr}_\texttt{n} \;\mid\; \texttt{if}\ (*)\ \texttt{Intr}_\texttt{n}\ \texttt{else}\ \boldsymbol{X} := \boldsymbol{t};\ \texttt{call}\ n;\ \ \boldsymbol{X} := \boldsymbol{t'}\ \texttt{endif} \end{aligned}$$

A Sloopy Program is an ordered list of procedures, $\langle P_0, P_1, \ldots, P_k \rangle$, where each P_i is generated by $\texttt{SProc}_\texttt{i}$. We assume that P_0 is the main procedure.

The class of Sloopy Programs has two main restrictions compared to the class of general programs defined above. First, it restricts what can occur inside a nondeterministic while loop. Specifically, it disallows conditionals inside while. If this is allowed, then invariant checking (of disequality assertions) becomes undecidable as shown by the program in Figure 3. The second restriction in Sloopy Programs concerns mutual recursion. Recursive calls are not allowed inside code generated by $\texttt{Intr}_\texttt{n}$, but Procedure P_n (generated by $\texttt{SProc}_\texttt{n}$) can recursively call itself.

Semantics. The semantics of the program constructs `if else endif` and `while endwhile` are standard, with the condition $(*)$ meaning that the control can flow in either direction in a nondeterministic way. The construct `call` n denotes a procedure call where control flows to the procedure with index n —

```
SolvePCP((u_1, v_1), ..., (u_k, v_k)):
 x := u_1(ε); y := v_1(ε);
 while (*) {
    if (*) { x := u_1(x); y := v_1(y); }
    if (*) { x := u_2(x); y := v_2(y); }
    ⋮
    if (*) { x := u_k(x); y := v_k(y); }
 }  assert(x ≠ y)
```

The assertion $x \neq y$ is not an invariant of program `SolvePCP` iff $u_1 u_{i_1} \dots u_{i_m} = v_1 v_{i_1} \dots v_{i_m}$ for some $i_1, \dots, i_m \in \{1, \dots, k\}$. Thus, we can solve PCP by checking if certain disequalities are invariants.

Fig. 3. Undecidability of invariant checking for general programs.

with the understanding that all variables $\boldsymbol{X}$ are global variables. Thus, a program essentially represents a (possibly infinite) collection of paths, where a path is a sequence of assignments.

Definition 2 (Semantics of a program). *The semantics of a path* $\boldsymbol{X} := \boldsymbol{e_1}; \boldsymbol{X} := \boldsymbol{e_2}; \cdots ; \boldsymbol{X} := \boldsymbol{e_k}$; *is the substitution obtained by composing the k substitutions as follows:* $\langle \boldsymbol{X} \mapsto \boldsymbol{e_1} \rangle \langle \boldsymbol{X} \mapsto \boldsymbol{e_2} \rangle \cdots \langle \boldsymbol{X} \mapsto \boldsymbol{e_k} \rangle$. *The semantics of a program* $[\![P]\!]$ *is the collection of the semantics of all its paths.*

The following lemma says that the semantics of the class of Sloopy Programs is effectively representable with extended parameterized substitutions.

Lemma 1. *For any Sloopy Program P, a finite set* Θ *of extended parameterized substitutions can be computed such that* $[\![P]\!] = \texttt{Instances}(\Theta)$.

Since the semantics of a Sloopy Program P is, by definition, the semantics of the main procedure P_0, the proof of Lemma 1 follows immediately from the same claim for Sloopy Procedures generated by $\texttt{SProc}_\texttt{n}$ stated and proved in Appendix A. The intuition behind the proof is that each basic block corresponds to a substitution and the parameters in the parameterized substitution represent the number of times a basic block (which is part of a loop or procedure) is executed. The relationship between these numbers is encoded in the constraint in the extended parameterized substitution. The restrictions on Sloopy Programs ensure that its semantics are representable by extended parameterized substitutions in this way (Lemma 1).

5 Invariant Checking

We define the invariant checking problem for programs as follows. Given a program P and a postcondition ψ, we are interested in testing whether

$$\models \sigma(\psi), \qquad \text{for all } \sigma \in [\![P]\!],$$

where $\sigma(\psi)$ denotes the formula obtained by applying σ to all the terms in ψ, and $\models$ denotes validity in the pure theory of equality. The postcondition formula ψ is

a (quantifier-free) formula built using equalities $t_1 = t_2$ as the atomic formulas, where t_1, t_2 are terms over $\mathcal{T}(\Sigma, \{\boldsymbol{X}\})$. Without loss of generality, we can assume that t_1 and t_2 are variables, since the program P can always introduce two new variables, say x, y, and assignments, $x := t_1;\ y := t_2$, and instead check for $x = y$.

For programs whose semantics is effectively representable with extended parameterized substitutions, the invariant checking problem reduces to the following problem. Given a formula ψ and *one* extended parameterized substitution $(\theta(\boldsymbol{u}); \chi(\boldsymbol{u}))$, determine whether

$$\models \theta(\boldsymbol{c})(\psi), \qquad \text{for all } \boldsymbol{c} \in \mathbb{N}^N \text{ s.t. } \models_{\mathbb{N}} \chi(\boldsymbol{c}),$$

that is, determine whether all solutions $\boldsymbol{c}$ of χ are also solutions of $\theta(\boldsymbol{u})(\psi)$.

We solve this problem by mapping the formula $\theta(\boldsymbol{u})(\psi)$ to a formula in Presburger arithmetic $\phi(\boldsymbol{u})$ such that the two formulas have the same set of solutions; that is, for all $\boldsymbol{c} \in \mathbb{N}^N$, $\models_{\mathbb{N}} \phi(\boldsymbol{c})$ iff $\models \theta(\boldsymbol{c})(\psi)$. We further simplify the proof by first considering only a disequality, $x \neq y$, in place of ψ. We will show that

Lemma 2. *Given a parameterized substitution $\theta(\boldsymbol{u})$ and variables x and y, there is a Presburger arithmetic formula $\phi_{x,y,\theta}(\boldsymbol{u})$ such that for all $\boldsymbol{c} \in \mathbb{N}^N$, it is the case that $\models \theta(\boldsymbol{c})(x) \neq \theta(\boldsymbol{c})(y)$ iff $\models_{\mathbb{N}} \phi_{x,y,\theta}(\boldsymbol{c})$.*

Parameterized Disequation to Automaton We prove Lemma 2 by solving a more general problem. Given substitutions $\sigma_1, \ldots, \sigma_N$ and $\beta_1, \ldots, \beta_M$, all with domain $\{x_1, \ldots, x_k\}$ and range $\mathcal{T}(\Sigma, \{x_1, \ldots, x_k\})$, we consider the problem of characterizing the solutions of the disequation

$$\sigma_N^{n_N} \ldots \sigma_1^{n_1}(x) \neq \beta_M^{m_M} \ldots \beta_1^{m_1}(y),$$

where $n_1, \ldots, n_N, m_1, \ldots, m_M$ are variables ranging over the natural numbers, and x, y are variables in $\{x_1, \ldots, x_k\}$. Our goal is to represent all the solutions of this disequation, written in short as $\boldsymbol{\sigma^n}(x) \neq \boldsymbol{\beta^m}(y)$, as a Presburger arithmetic formula with $N + M$ free variables $\boldsymbol{n}, \boldsymbol{m}$. To this end, we first construct an automaton A that accepts words over an alphabet of vectors $\{0,1\}^{N+M}$ such that the following property holds: $\boldsymbol{\sigma^n}(x)$ will be different from $\boldsymbol{\beta^m}(y)$ iff there exists a word $w := \boldsymbol{v_1} \ldots \boldsymbol{v_l}$ in the language accepted by A such that $Sum(w) := \boldsymbol{v_1} + \ldots + \boldsymbol{v_l}$ is equal to the vector $\langle \boldsymbol{n}, \boldsymbol{m} \rangle \in \mathbb{N}^{N+M}$. In fact, only the vector $\mathbf{0}$ and the unit vectors of the canonical basis $\boldsymbol{e}_1, \ldots, \boldsymbol{e}_{N+M}$ will appear in the definition of A. (Here, $\boldsymbol{e}_j \in \{0,1\}^{N+M}$ is a vector that has 1 in the j-th position and 0's elsewhere.)

Intuitively, the automaton non-deterministically searches for the position in the terms $\boldsymbol{\sigma^n}(x)$ and $\boldsymbol{\beta^m}(y)$ where the two terms are different. Informally, call this position the *point of difference*. We use the notation $\boldsymbol{N}$ to denote the set $\{1, \ldots, N\}$. The non-deterministic automaton $A = (Q, \Sigma_A, q_{\texttt{init}}, Q_F, T)$ is defined as follows:
(1) The alphabet Σ_A is $\{\mathbf{0}, \boldsymbol{e}_1, \ldots, \boldsymbol{e}_{N+M}\}$. Informally, when the automaton A

makes a transition on symbol $\boldsymbol{e}_i$, for $i \in \boldsymbol{N}$, then it means that the automaton A decided to use one more application of σ_i in its search for the "point of difference". Analogously, when it makes a transition on symbol $\boldsymbol{e}_{N+j}$, for $j \in \boldsymbol{M}$, it decided to use one more application of β_j.
(2) The set of states Q of the automaton A is

$$\begin{aligned} Q &= Q_1 \cup Q_F \\ Q_1 &= \{\langle i, s, j, t\rangle \mid i \in \boldsymbol{N}, s \text{ is a subterm of } \{x_1, \ldots, x_k\} \cup \{\sigma_i(x_1), \ldots, \sigma_i(x_k)\}, \\ &\qquad j \in \boldsymbol{M}, t \text{ is a subterm of } \{x_1, \ldots, x_k\} \cup \{\beta_j(x_1), \ldots, \beta_j(x_k)\}\} \\ Q_F &= \{q_{ij} \mid i \in \boldsymbol{N}+\boldsymbol{1}, j \in \boldsymbol{M}+\boldsymbol{1}\} \end{aligned}$$

Here Q_F is the set of accepting states, and $q_{\texttt{init}} = \langle 1, x, 1, y\rangle$ is the initial state. Intuitively, when A is in the state $\langle i, s, j, t\rangle$, then it means that A is currently applying σ_i and β_j, respectively, and currently matching s and t, in its search for the "point of difference".
(3) The transitions T of A are given in Table 1. Informally speaking, in its search for the "point of difference" from state $\langle i, s, j, t\rangle$, the non-deterministic automaton A does the following: (a) if the top function symbols are different, then it moves into an accept state (T_1'), (b) if the top function symbols are identical, then it *guesses* under which subterms the "point of difference" may lie, and moves into the state with these subterms (T_1), (c) if the search reaches a variable, then it non-deterministically chooses to either apply the current substitution (T_2, T_2') and continue the search, or it moves to the next substitution (T_3, T_3'), (d) if the search reaches the last substitutions, then it moves into an accepting state if it finds the "point of difference" (T_4, T_4', T_5), and (e) if A is in an accepting state (that is, it has found the "point of difference"), but it has not used up all the available substitutions, then it accepts all possible choices for the remaining substitutions (T_6, T_6').

The next two lemmas will capture the intuition behind the construction of A. The first lemma states that every run of A corresponds to some instance of x and y and some path on those instances.

Lemma 3. *Let A be an automaton constructed from x, y, $\boldsymbol{\sigma}$ and $\boldsymbol{\beta}$ as before. Let w be a word in Σ_A^*. Let $\langle 1, x, 1, y\rangle \xrightarrow{w} \langle i, s, j, t\rangle$ be a run of the automaton A. Let $Sum(w) = \langle \boldsymbol{n}, \boldsymbol{m}\rangle$, $u = \boldsymbol{\sigma}^{\boldsymbol{n}}(x)$ and $v = \boldsymbol{\beta}^{\boldsymbol{m}}(y)$. Then, there is a position p such that $s = u|_p$, $t = v|_p$, and $\texttt{root}(u|_{p'}) = \texttt{root}(v|_{p'})$ for all positions $p' < p$.*

Proof. We generate the required position p by annotating each state in the given run with a position. The initial state is annotated with position ϵ. If p is the annotation on the current state, then (a) if the next state is obtained using a transition from the set T_1, then the next state is annotated with $p.l$, and (b) if the next state is obtained using any other transition, then the next state is annotated with p.

Now, the lemma follows by induction on the length of the run. In the base case, the claim is clearly true for the initial state. It is easily verified that the claim is preserved on every transition that does not lead to some q_{ij} state. □

$$
\begin{aligned}
T &= T_1 \cup T_1' \cup T_2 \cup T_2' \cup T_3 \cup T_3' \cup T_4 \cup T_4' \cup T_5 \cup T_6 \cup T_6' \\
T_1 &= \{(\langle i, fs_1 \dots s_n, j, ft_1 \dots t_n\rangle, \mathbf{0}, \langle i, s_l, j, t_l\rangle) \mid i \in \boldsymbol{N}, j \in \boldsymbol{M}, l \in \boldsymbol{n}\} \\
T_1' &= \{(\langle i, fs_1 \dots s_n, j, gt_1 \dots t_m\rangle, \mathbf{0}, q_{ij}) \mid i \in \boldsymbol{N}, j \in \boldsymbol{M}, f \not\equiv g\} \\
T_2 &= \{(\langle i, x, j, t\rangle, \boldsymbol{e}_i, \langle i, \sigma_i(x), j, t\rangle) \mid i \in \boldsymbol{N}, j \in \boldsymbol{M}\} \\
T_2' &= \{(\langle i, t, j, x\rangle, \boldsymbol{e}_{N+j}, \langle i, t, j, \sigma_j(x)\rangle) \mid i \in \boldsymbol{N}, j \in \boldsymbol{M}\} \\
T_3 &= \{(\langle i, x, j, t\rangle, \mathbf{0}, \langle i+1, x, j, t\rangle) \mid i \in \boldsymbol{N} - \mathbf{1}, j \in \boldsymbol{M}\} \\
T_3' &= \{(\langle i, t, j, x\rangle, \mathbf{0}, \langle i, t, j+1, x\rangle) \mid i \in \boldsymbol{N}, j \in \boldsymbol{M} - \mathbf{1}\} \\
T_4 &= \{(\langle N, x, j, t\rangle, \mathbf{0}, q_{N+1,j}) \mid j \in \boldsymbol{M}, t \notin \boldsymbol{X}\} \\
T_4' &= \{(\langle i, t, M, x\rangle, \mathbf{0}, q_{i,M+1}) \mid i \in \boldsymbol{N}, t \notin \boldsymbol{X}\} \\
T_5 &= \{(\langle N, s, M, t\rangle, \mathbf{0}, q_{N+1,M+1}) \mid s, t \in \boldsymbol{X}, s \not\equiv t\} \\
T_6 &= \{(q_{ij}, \boldsymbol{e}_l, q_{ij}) \mid i \in \boldsymbol{N}, j \in \boldsymbol{M} + \mathbf{1}, i \le l \le N\} \\
T_6' &= \{(q_{ij}, \boldsymbol{e}_{N+l}, q_{ij}) \mid i \in \boldsymbol{N} + \mathbf{1}, j \in \boldsymbol{M}, j \le l \le M\}
\end{aligned}
$$

Table 1. Transitions of the automaton encoding solutions of a parameterized disequation.

Conversely, we can show that given an instance of x and y and a position p on these instances, we can find a corresponding run of A.

Lemma 4. *Let A be an automaton constructed from x, y, $\boldsymbol{\sigma}$ and $\boldsymbol{\beta}$ as before. Let $\boldsymbol{n}, \boldsymbol{m}$ be $N + M$ natural numbers and let u be $\boldsymbol{\sigma}^{\boldsymbol{n}}(x)$ and v be $\boldsymbol{\beta}^{\boldsymbol{m}}(y)$. If p is a position in u and v such that $\mathtt{root}(u|_{p'}) = \mathtt{root}(v|_{p'})$ for all $p' < p$, then there is a run of A, $\langle 1, x, 1, y\rangle \xrightarrow{w} \langle i, s, j, t\rangle$ such that $Sum(w) = \langle n_1, \dots, n_{i-1}, n', 0, \dots, 0, m_1, \dots, m_{j-1}, m', 0, \dots, 0\rangle$, $n' \le n_i$, $m' \le m_j$, $s := (\sigma_i^{n'} \sigma_{i-1}^{n_{i-1}} \dots \sigma_1^{n_1}(x))|_p$, and $t := (\beta_j^{m'} \beta_{j-1}^{m_{j-1}} \dots \beta_1^{m_1}(y))|_p$.*

Proof. We construct the required run of automaton A by following the path p on terms u and v. We need to keep three auxiliary variables – two indices n', m', and a position p' – to guide this run of A. We just append the 3 auxiliary variables to the state to simplify presentation. The starting state is $\langle\langle 1, x, 1, y\rangle, 0, 0, \epsilon\rangle$.

Suppose that the current (extended) state of A is $\langle\langle i, s, j, t\rangle, n', m', p'\rangle$. The auxiliary variables will satisfy the invariant that $0 \le n' \le n_i$, $0 \le m' \le m_j$, $\epsilon \le p' \le p$, $s = (\sigma_i^{n'} \sigma_{i-1}^{n_{i-1}} \dots \sigma_1^{n_1}(x))|_p$, and $t = (\beta_j^{m'} \beta_{j-1}^{m_{j-1}} \dots \beta_1^{m_1}(y))|_p$. Now, the next state in the required run will be:

(1) $\langle\langle i, s|_l, j, t|_l\rangle, n', m', p'.l\rangle$ (using a transition from T_1), if neither s nor t is a variable and $p'.l \le p$. Note that, by assumption, in this case, $\mathtt{root}(s) = \mathtt{root}(t)$ and hence a transition from T_1 will be enabled.

(2) $\langle\langle i, \sigma_i(s), j, t\rangle, n' + 1, m', p'\rangle$ (using a transition from T_2), if s is a variable and $n' < n_i$.

(2') $\langle\langle i, s, j, \beta_j(t)\rangle, n', m' + 1, p'\rangle$ (using a transition from T_2'), if t is a variable and $m' < m_j$.

(3) $\langle\langle i+1, s, j, t\rangle, 0, m', p'\rangle$ (using a transition from T_3), if s is a variable, $i < N$, and $n' = n_i$.

(3') $\langle\langle i, s, j+1, t\rangle, n', 0, p'\rangle$ (using a transition from T_3'), if t is a variable, $j < M$, and $m' = m_j$.

Using induction on the length of position p, it is easy to prove that the above run has all the desired properties (stated in the lemma). □

We can now state the correctness of the construction of A, but leave the proof to Appendix A.

Lemma 5. *Let A be an automaton constructed from x, y, $\boldsymbol{\sigma}$ and $\boldsymbol{\beta}$ as before. Let $\boldsymbol{n}, \boldsymbol{m}$ be $N + M$ natural numbers.*

Then $\boldsymbol{\sigma}^{\boldsymbol{n}}(x)$ is different from $\boldsymbol{\beta}^{\boldsymbol{m}}(y)$ iff there exists a word w accepted by A such that $Sum(w) = \langle \boldsymbol{n}, \boldsymbol{m}\rangle$.

For an automaton A, let $L(A)$ denote the language accepted by A and let $Sum(L(A))$ denote the set $\{Sum(w) \mid w \in L(A)\}$. Lemma 5 gives us the following result on representing solutions of parameterized disequations.

Theorem 2. *Let $\boldsymbol{X}$ be a finite set of variables and $\boldsymbol{\sigma}, \boldsymbol{\beta}$ be $N+M$ substitutions mapping $\boldsymbol{X}$ to the set of terms $\mathcal{T}(\Sigma, \boldsymbol{X})$. Given $x, y \in \boldsymbol{X}$, there is a finite automaton A such that*

$$Sum(L(A)) = \{\langle \boldsymbol{c}, \boldsymbol{d}\rangle \in \mathbb{N}^{N+M} \mid \boldsymbol{\sigma}^{\boldsymbol{c}}(x) \neq \boldsymbol{\beta}^{\boldsymbol{d}}(y)\}$$

The number of states in A is bounded by $O(InputSize^4)$ and the number of transitions is bounded by $O(InputSize^8)$, where *InputSize* is the size of the input $\boldsymbol{\sigma}, \boldsymbol{\beta}$.

Automaton to Presburger Formula Let A be a finite automaton over the alphabet $\{\mathbf{0}, \boldsymbol{e}_1, \ldots, \boldsymbol{e}_{N+M}\}$. If we treat $\mathbf{0}$ as the ϵ-symbol, then it is obvious that $Sum(L(A))$ is simply the Parikh image of $L(A)$. It follows from Theorem 1 that we can represent $Sum(L(A))$ by a Presburger formula ϕ_A.

Disequation to Arbitrary Formula We can put together Theorems 2 and 1 to immediately get a proof of Lemma 2. In fact, Lemma 2 can now be easily generalized to arbitrary formulas ψ whose atomic formulas are equations between variables.

Theorem 3. *Given an extended parameterized substitution $(\theta(\boldsymbol{u}); \chi(\boldsymbol{u}))$ and a quantifier-free equality formula ψ, there is a Presburger formula $\phi_{\psi,(\theta;\chi)}(\boldsymbol{u})$ such that for all $\boldsymbol{c} \in \mathbb{N}^N$, it is the case that $\models \chi(\boldsymbol{c}) \Rightarrow \theta(\boldsymbol{c})(\psi)$ iff $\models_{\mathbb{N}} \phi_{\psi,(\theta;\chi)}(\boldsymbol{c})$.*

Proof. We proceed by structural induction on ψ. The base case is when ψ is $x = y$. By Lemma 2, we know there is a formula $\phi_{x\neq y,\theta}$ corresponding to $x \neq y$.

Thus, for all $\boldsymbol{c} \in \mathbb{N}^N$, we have the following inference:

$$\begin{array}{lll} & \models \theta(\boldsymbol{c})(x) \neq \theta(\boldsymbol{c})(y) \text{ iff } \phi_{x\neq y,\theta}(\boldsymbol{c}) & \text{Lemma 2} \\ \therefore, & \models \theta(\boldsymbol{c})(x) = \theta(\boldsymbol{c})(y) \text{ iff } \neg\phi_{x\neq y,\theta}(\boldsymbol{c}) & \\ \therefore, & \models \chi(\boldsymbol{c}) \Rightarrow \theta(\boldsymbol{c})(x) = \theta(\boldsymbol{c})(y) \text{ iff } \chi(\boldsymbol{c}) \Rightarrow \neg\phi_{x\neq y,\theta}(\boldsymbol{c}) & \\ \therefore, & \phi_{\psi,(\theta;\chi)} := \phi_{x=y,(\theta;\chi)} := \chi(\boldsymbol{u}) \Rightarrow \neg\phi_{x\neq y,\theta}(\boldsymbol{u}) & \end{array}$$

For the inductive step, if ψ is $\psi_1 \vee \psi_2$, then it is easy to see that $\phi_{\psi,\theta}$ is $\phi_{\psi_1,\theta} \vee \phi_{\psi_2,\theta}$, and similarly for the cases when ψ is $\neg\psi_1$ and when ψ is $\psi_1 \wedge \psi_2$. □

Note that $\phi_{\psi,\theta}$ is always an existentially quantified Presburger formula (with free variables). We can test the validity of the (universal closure) of $\phi_{\psi,\theta}$ and thus decide the invariant checking problem for Sloopy Programs.

Theorem 4. *Let P be a Sloopy Program and ψ be an assertion. The problem of checking if ψ is an invariant for P is decidable.*

Proof. Using Lemma 1, we first get a finite set Θ of extended parameterized substitutions that represent $[\![P]\!]$, that is, $[\![P]\!] = \texttt{Instances}(\Theta)$. For each parameterized substitution $(\theta;\chi) \in \Theta$, we use Theorem 3 to construct a Presburger formula $\phi_{\psi,(\theta;\chi)}$ and test the validity of (the universal closure of) $\phi_{\psi,(\theta;\chi)}$. If all such Presburger formulas are valid, then ψ is an invariant of P; otherwise, it is not. The correctness follows from the following reasoning:

$$\begin{array}{lll} & \psi \text{ is an invariant of } P & \\ \text{iff} & \models \sigma(\psi) \text{ for each } \sigma \in [\![P]\!] & \text{By definition of invariant} \\ \text{iff} & \models \sigma(\psi) \text{ for each } \sigma \in \texttt{Instances}(\Theta) & \text{Lemma 1} \\ \text{iff} & \models \chi(\boldsymbol{c}) \Rightarrow \theta(\boldsymbol{c})(\psi) \text{ for each } (\theta;\chi) \in \Theta, \boldsymbol{c} \in \mathbb{N}^l & \text{Definition of } \texttt{Instances} \\ \text{iff} & \forall \boldsymbol{u}(\phi_{\psi,(\theta;\chi)}(\boldsymbol{u})) \text{ is valid} & \text{Theorem 3} \end{array}$$

This completes the proof. □

6 Discussion

The class of Sloopy Programs has some severe restrictions compared to the class of general programs defined in Section 4. It disallows procedure calls and loops inside a loop and it supports only a limited form of recursion. If we allow arbitrary loops and recursive procedure calls (that is, use the general program model P from Section 4), but restrict to (positive) equations as assertions, then the decidability is not known, although it is known for some subcases [9, 5]. If linear arithmetic is the expression language, then disequality checking remains undecidable (similar proof as Figure 3), but equality checking is known to be decidable [13, 9].

The decidability result for the class of Sloopy Programs actually works more generally. It works for any program in which the mapping that maps a run of that program to the vector of the number of times a basic block is executed in that run is injective. In other words, given the number of times a basic block has executed in a run, it should be possible to extract the exact run of the

program. For any such program, the decidability arguments given in this paper are applicable.

The proof of decidability given here performs two steps. The first step computes the semantics of the program using extended parameterized substitutions. Fixing the parameters fixes a program path. The second step constructs an automaton that characterizes all solutions for the parameters that make an assertion true. We can get a direct proof by merging these two steps. Note that applying a substitution (corresponding to a basic block) to an assertion is the same as computing the weakest precondition of the assertion with respect to the basic block.

Techniques used for deciding equality assertions do not directly apply for deciding disequality assertions. Decidability for equality assertions often relies on the fact that there can be only *finitely* many *non-redundant equations*. This is mostly not true for disequalities.

7 Conclusions

We presented two decidability results in this paper. First, we showed decidability of the following problem: given substitutions $\sigma_1, \ldots, \sigma_N, \beta_1, \ldots, \beta_M$ and terms x, y, is there a vector $(c_1, \ldots, c_N, d_1, \ldots, d_N)$ of natural numbers in a given semilinear set such that $\sigma_N^{c_N} \ldots \sigma_1^{c_1}(x) \neq \beta_M^{d_M} \ldots \beta_1^{d_1}(y)$? We also showed decidability of the above problem when the disequality is replaced by an equality or any Boolean combination of equalities and disequalities. Using the above result, we established decidability of invariant checking for a large class of programs with recursion. Our decidability result is valid for any class of programs whose semantics is effectively representable with extended parameterized substitutions, and Sloopy Programs are just a particular case. It would be interesting to study alternative classes of programs satisfying this property. Moreover, note that our assertions are just quantifier-free boolean formulas. Adding quantification to the assertions will be interesting, since the first order theory of term algebras is known to be decidable [2]. Other variants of the problem can also be considered for future research, such as allowing local variables or parameters in the procedures, and incorporating interpreted symbols in the signature.

Acknowledgments. We thank the reviewers for their helpful comments.

References

1. A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. In *POPL*, pages 62–73, 2003.
2. H. Comon and C. Delor. Equational formulae with membership constraints. *Inf. Comput.*, 112:167–216, 1994.
3. J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithm for model checking pushdown systems. In *CAV*, volume 1885 of *LNCS*, 2000.
4. J. Esparza and J. Knoop. An automata-theoretic approach to interprocedural data-flow analysis. In *FoSSaCS*, volume 1578 of *LNCS*, 1999.

5. A. Gascon, G. Godoy, M. Schmidt-Schauß, and A. Tiwari. Context unification with one context variable. *J. of symbolic computation*, 2009. Submitted.
6. S. Gulwani and G. C. Necula. A polynomial-time algorithm for global value numbering. In *SAS*, volume 3148 of *LNCS*, pages 212–227, 2004.
7. S. Gulwani and A. Tiwari. Assertion checking over combined abstraction of linear arithmetic & uninterpreted functions. In *ESOP*, volume 3924 of *LNCS*, 2006.
8. S. Gulwani and A. Tiwari. Assertion checking unified. In *VMCAI*, volume 4349 of *LNCS*. Springer, 2007.
9. S. Gulwani and A. Tiwari. Computing procedure summaries for interprocedural analysis. In *ESOP*, volume 4421 of *LNCS*, pages 253–267. Springer, 2007.
10. M. Müller-Olm, M. Petter, and H. Seidl. Interprocedurally analyzing polynomial identities. In *STACS*, volume 3884 of *LNCS*, pages 50–67. Springer, 2006.
11. M. Müller-Olm, O. Rüthing, and H. Seidl. Checking Herbrand equalities and beyond. In *VMCAI*, volume 3385 of *LNCS*, pages 79–96, January 2005.
12. M. Müller-Olm and H. Seidl. A note on Karr's algorithm. In *31st ICALP*, pages 1016–1028, 2004.
13. M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In *31st ACM Symposium on POPL*, pages 330–341, January 2004.
14. M. Müller-Olm, H. Seidl, and B. Steffen. Interprocedural Herbrand equalities. In *ESOP*, volume 3444 of *LNCS*, pages 31–45. Springer, 2005.
15. R. J. Parikh. On context-free languages. *J. of the ACM*, 13(4):570–581, 1966.
16. T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *22nd ACM Symposium on POPL*, pages 49–61, 1995.
17. H. Seidl, T. Schwentick, A. Muscholl, and P. Habermehl. Counting in trees for free. In *ICALP*, volume 3142 of *LNCS*, pages 1136–1149, 2004.
18. A. Tiwari and S. Gulwani. Logical interpretation: Static program analysis using theorem proving. In *CADE-21*, volume 4603 of *LNAI*, pages 147–166, 2007.

A Supplementary Lemmas and Proofs

Lemma 6. *For any Sloopy Procedure SP_n, there is a finite set Θ of extended parameterized substitutions such that $[\![SP_n]\!] = \texttt{Instances}(\Theta)$.*

Proof. We prove this by structural induction on the structure of the Sloopy Procedure SP_n. By assumption programs are finite, and hence the structural induction process is well defined. The following equations define the set of extended parameterized substitutions that capture the semantics of Sloopy Procedure SP_n in terms of the the semantics of its components.

$$
\begin{aligned}
[\![\boldsymbol{X} := \boldsymbol{t}]\!] &= \{(\langle \boldsymbol{X} \mapsto \boldsymbol{t}\rangle^n; n = 1)\} \\
[\![\texttt{while } (*) \ \boldsymbol{X} := \boldsymbol{t} \ \texttt{endwhile}]\!] &= \{(\langle \boldsymbol{X} \mapsto \boldsymbol{t}\rangle^n; \texttt{true})\} \\
[\![\texttt{Intr}_\texttt{n} \ ; \ \texttt{Intr}_\texttt{n}{}']\!] &= \{(\theta\theta'; \chi \wedge \chi') \mid (\theta;\chi) \in [\![\texttt{Intr}_\texttt{n}]\!], \\
&\qquad\qquad (\theta';\chi') \in [\![\texttt{Intr}_\texttt{n}{}']\!]\} \\
[\![\texttt{if } (*) \ \texttt{Intr}_\texttt{n} \ \texttt{else} \ \texttt{Intr}_\texttt{n}{}' \ \texttt{endif}]\!] &= [\![\texttt{Intr}_\texttt{n}]\!] \cup [\![\texttt{Intr}_\texttt{n}{}']\!] \\
[\![\texttt{call } m]\!] &= [\![P_m]\!] \\
[\![\texttt{if } (*) \ \texttt{Intr}_\texttt{n}; \ \texttt{else } \boldsymbol{X} := \boldsymbol{t}; \ \texttt{call } n; \ \ \boldsymbol{X} := \boldsymbol{t}' \ \texttt{endif}]\!] & \\
&= \{(\langle \boldsymbol{X} \mapsto \boldsymbol{t}\rangle^{n_1} \theta \langle \boldsymbol{X} \mapsto \boldsymbol{t}'\rangle^{n_2}; \chi \wedge n_1 = n_2) \mid \\
&\qquad (\theta;\chi) \in [\![\texttt{Intr}_\texttt{n}]\!]\}
\end{aligned}
$$

where n, n_1 and n_2 are always new variables ranging over the natural numbers. Recall that the side condition, $m > n$, guarantees that the above inductive way of obtaining parameterized representation of semantics is well defined. The correctness of the above definition is obvious. □

Proof (Lemma 5). $\Rightarrow$**:** Assume that $u = \sigma_N^{n_N} \dots \sigma_1^{n_1}(x)$ is different from $v = \beta_M^{m_M} \dots \beta_1^{m_1}(y)$. Then, there exists a position $p \in \mathtt{Pos}(u) \cap \mathtt{Pos}(v)$ such that $\mathtt{root}(u|_p) \neq \mathtt{root}(v|_p)$. Among all choices for such a p, we choose one with the minimal size/length. Hence, $\mathtt{root}(u|_{p'}) = \mathtt{root}(v|_{p'})$ for all $p' < p$.

Using Lemma 4 on this choice of position p, we infer that there is a run of automaton A, $\langle 1, x, 1, y\rangle \xrightarrow{w} \langle i, s, j, t\rangle$, such that

$$Sum(w) = \langle n_1, \dots, n_{i-1}, n', 0, \dots, 0, m_1, \dots, m_{j-1}, m', 0, \dots, 0\rangle,$$

with $n' \leq n_i$, $m' \leq m_j$, $s = (\sigma_i^{n'} \sigma_{i-1}^{n_{i-1}} \dots \sigma_1^{n_1}(x))|_p$, and $t = (\beta_j^{m'} \beta_{j-1}^{m_{j-1}} \dots \beta_1^{m_1}(y))|_p$. Among all the runs satisfying these conditions, we choose a run r maximum in length. Because of the choice of p, one of the following conditions is satisfied:
(a) *Neither s nor t is a variable*: In this case, a transition from set T_1' is applicable and we can complete the current run to get the following accepting run:

$$\langle 1, x, 1, y\rangle \xrightarrow{w} \langle i, s, j, t\rangle \xrightarrow[T_1']{\mathbf{0}} q_{ij} \xrightarrow[T_6, T_6']{w'} q_{ij}$$

where w' is the word $e_i^{n_i - n'} e_{i+1}^{n_{i+1}} \dots e_N^{n_N} e_{N+j}^{m_j - m'} e_{N+j+1}^{m_{j+1}} \dots e_{N+M}^{m_M}$. By construction, we have $Sum(w\mathbf{0}w') = \langle n_1, \dots, n_N, m_1, \dots, m_M\rangle$ in this accepting run.
(b) *Either s or t is a variable*: Suppose s is a variable, say x', and t is not a variable. If $i \neq N$ or $n' < n_i$, then we can apply a transition of T_2 or T_3 to obtain a larger run than r and with the same conditions, contradicting then the election of r. Hence, $i = N$ and $n' = n_N$. Thus, we can complete the current run as follows:

$$\langle 1, x, 1, y\rangle \xrightarrow{w} \langle i, x', j, t\rangle \;=\; \langle N, x', j, t\rangle \xrightarrow[T_4]{\mathbf{0}} q_{N+1,j} \xrightarrow[T_6']{w'} q_{N+1,j}$$

where w' is the word $e_{N+j}^{m_j - m'} e_{N+j+1}^{m_{j+1}} \dots e_{N+M}^{m_M}$. Again, by construction, we have $Sum(w\mathbf{0}w') = \langle n_1, \dots, n_N, m_1, \dots, m_M\rangle$ in this accepting run. Finally, the other cases, when either t or both s and t are variables can be handled similarly and we get the desired accepting run in each case.

$\Leftarrow$**:** Suppose that A accepts w and $Sum(w) = \langle n_1, \dots, n_N, m_1, \dots, m_M\rangle$. Thus, we have an accepting run of A that can be written in the following form:

$$\langle 1, x, 1, y\rangle \xrightarrow{w'} \langle i, s, j, t\rangle \xrightarrow[T_1, T_4, T_4', T_5]{\mathbf{0}} q_{IJ} \xrightarrow[T_6, T_6']{w''} q_{IJ}$$

It holds that $Sum(w') = \langle n_1, \dots, n_{i-1}, n', 0, \dots, 0, m_1, \dots, m_{j-1}, m', 0, \dots, 0\rangle$ for some $n' \leq n_i$ and $m' \leq m_j$. Let $u := \sigma_i^{n'} \sigma_{i-1}^{n_{i-1}} \dots \sigma_1^{n_1}(x)$ and $v := \sigma_i^{n'} \sigma_{i-1}^{n_{i-1}} \dots \sigma_1^{n_1}(x)$. Using Lemma 3 on the run $\langle 1, x, 1, y\rangle \xrightarrow{w'} \langle i, s, j, t\rangle$, we conclude that there is a position p such that s is $u|_p$, t is $v|_p$, and $\mathtt{root}(u|_{p'}) = \mathtt{root}(v|_{p'})$ for all $p' < p$.

We can now complete the proof depending on whether we used T_1, T_4, T_4', or T_5 in the accepting run above:

T_1 : In this case, u and v are different at a non-leaf position p. Hence, $\sigma_N^{n_N} \dots \sigma_1^{n_1}(x)$ and $\beta_M^{m_M} \dots \beta_1^{m_1}(y)$, which are just instances of u and v, will also differ at position p.

T_4 : In this case, s is a variable and t is not a variable. Furthermore, I is necessarily $N+1$ in this case. Consequently, the only transitions applicable on $q_{I,J}$ are those in T_6', and hence w'' cannot contain $\boldsymbol{e}_i$ for $i \leq N$. Hence, at position p, the term $\sigma_N^{n_N} \dots \sigma_1^{n_1}(x)$ contains the variable s, whereas the term $\beta_M^{m_M} \dots \beta_1^{m_1}(y)|_p$ will not be a variable.

T_4' : This case is similar to the previous case.

T_5 : In this case, $I = N+1, J = M+1$, and hence, at position p, the two terms – $\sigma_N^{n_N} \dots \sigma_1^{n_1}(x)$ and $\beta_M^{m_M} \dots \beta_1^{m_1}(y)$ – have *distinct* variables.

Thus, in all cases, the terms $\sigma_N^{n_N} \dots \sigma_1^{n_1}(x)$ and $\beta_M^{m_M} \dots \beta_1^{m_1}(y)$ are different. □