An Afficient Equivalence-Checking Algorithm for a Model of Programs with Commutative and Absorptive Statements

Vladislav Podymov

National Research University Higher School of Economics
3 Kochnovsky Proezd, Moscow 125319, Russia
valdus@yandex.ru

Abstract. We present an efficient equivalence-checking algorithm for a propositional model of programs with semantics based on (what we call) progressive monoids on the finite set of statements generated by relations of a specific form. We consider arbitrary set of relations for commutativity (relations of the form $ab = ba$ for statements a, b) and left absorption (relations of the form $ab = b$ for statements a, b) properties. The main results are a polynomial-time decidability for the equivalence problem in the considered case, and an explicit description of an equivalence-checking algorithm which terminates in time polynomial in size of programs.

Key words: program models, equivalence checking, semigroups, commutativity, left absorption

1 Introduction

The paper addresses the functional equivalence problem for imperative programs: given two programs, do determine whether they compute the same function. This problem is known to be undecidable in general (follows from Rice's theorem [1]), but is highly important to have means to determine if given programs are equivalent: theoretically, the equivalence problem complexity shows what influence the structure of programs has on their meaning, i.e. which structural properties can be analyzed and which cannot; practically, equivalence-checking algorithms can be used in a huge variety of program analysis problems: optimization, verification, refactoring, obfuscation, and so on — in which any two programs (e.g. original and transformed, or ideal and real) are tested for having the same functionality.

One way to overcome the negativity of Rice's theorem is to provide sufficient conditions for program equivalence. For example, an optimizing compiler performs a sequence of program transformation steps, each step preserves the original program functionality, and sufficient equivalence conditions may be used to prove transformation soundness and provide optimization algorithms known to be sound. The range of sound and efficient optimization capabilities is impressive: compilers have means to remove some unreachable or dead code, to restructure control flow graph and its linear code blocks, to optimize a variety of efficiency characteristics [2]. Nevertheless, there is still a long way to go. For example, the block of code

```
if(P(z)) {x=f(x); y=y+z;} else {x=g(x);x=v+z;}
if(!P(z)) y=y+z; else x=v+z;
```

and a much simpler block `y=y+z; x=v+z;` are equivalent, if we assume that the functions `P`, `f`, `g` are side-effect free and terminate, but simplifications of this kind are unlikely to be done automatically by modern compilers. To prove these blocks are equvalent, it is sufficient to use the following properties:

1. the value of `P(z)` remains unchanged;
2. the order of statements `x = f(x);` and `y = y + z;` has no effect on the result, and the same holds for `x = v + z;` and `y = y + z;`
3. the statement `x = g(x);` is dead if `x = v + z;` acts right after it, and the same holds for `x = f(x);` and `x = v + z;`

We refer to (2) as commutativity, and (3) as left absorption properties for statements, and focus on these two properties only. Suppose we have a statement $a : (x_1, \ldots, x_n) = f(y_1, \ldots, y_k)$: "compute f on arguments $y_1, \ldots, y_k$, and put the result into $x_1, \ldots, x_n$". Let $Mod(a)$ be the set $\{x_1, \ldots, x_n\}$, and $Use(a)$ be $\{y_1, \ldots, y_k\}$. An easy way to extract commutativity and left absorption for statements is to analyze their Use- and Mod-sets: if $Use(a) \cap Mod(b) = Mod(a) \cap Use(b) = Mod(a) \cap Mod(b) = \emptyset$, then a and b are commutative; if $Mod(a) \subseteq Mod(b)$ and $Mod(a) \cap Use(b) = \emptyset$, then b is left-absorptive for a.

To investigate the equivalence problem with respect to these properties we use the following approach: describe a program model which preserves the original program syntax but has a much simpler abstract semantics preserving some original semantical properties (programs in such a model are often called program schemata); propose an equivalence notion for schemata such that if two given schemata are equivalent, then the original programs are also equivalent; solve the equivalence problem for schemata, and use the resulting equivalence-checking algorithm to verify the original program equivalence.

We consider the model called "propositional programs" (PPs for short) proposed in [3, 4]. A PP is basically a finite automaton such that: vertices are labeled with statement symbols and correspond to program control locations in which a current data state is changed; transitions are labeled with all possible valuations for Boolean condition symbols; vertices are connected in the same way as in the original program. An interpretation which defines how a PP operates consists of: a set of abstract data states; a meaning of each statement symbol as a function for data state modification; a meaning of each condition symbol as a truth-value marking of data states. In general case these components are defined in terms of Kripke frames and models [5], but this paper focuses on commutativity and left absorption properties only: in this case data states are elements of a monoid on the set of statements generated by a set of relations of the form $ab = ba$ (for commutativity) and $ab = b$ (for left absorption) where a, b are statement symbols. For short, we call such a monoid a CA-monoid. We call two PPs equvalent in a CA-monoid M if for each interpretation based on M last data states of their executions are equal. All necessary definitions are given in the further sections.

The main goal of the paper is to provide an efficient equivalence-checking algorithm for PPs operating in CA-monoids. We call an algorithm efficient if it terminates

in a polynomial time. Unfortunately, even for the trivial case when no commutativity and left absorption is assumed the equivalence problem for PPs is known to be co-NP-complete [6]. But for a fixed finite signature (alphabet of statements and conditions) for the trivial case PPs are known to be closely related to finite automata [7] and have an equivalence-checking algorithm having a very low time complexity [8]. Thus, hereafter a fixed finite signature is assumed, which means that we study the hardness with respect to structural program features. Another unpleasant fact is that equivalence-checking results for automata and PPs are not interchangeable for nontrivial cases, for example, for arbitrary commutativity the equivalence problem for automata is known to be undecidable [9], while for schemata it is decidable in polynomial time for a syntactically and semantically more general case [10].

CA-monoids considered in this paper are assumed to be progressive: we cannot obtain the same element twice appending non-neutral element to the right via composition. Note that for left-cancellative monoids the progressiveness is the same as the atomicity of the neutral element. The main result of the paper is an improvement of known decidability of the equivalence problem in the considered case [11] to polynomial-time decidability (theorem 2), and an explicit description of a polynomial-time decision algorithm (section 5). Note that if commutativity and left absorption properties are extracted in the way showed above, then the resulting CA-monoid is necessarily progressive. Moreover, [12] provides a simple-to-check progressiveness criterion for CA-monoids.

To make a few remarks on related work, we may mention polynomial-time decidability results on equivalence problems for one-counter automata [13] (NL-completeness), specific context-free languages [14, 15], specific recursive schemata [10], and decidability results for multitape automata [16] and k-valued transducers [17]. Note that there are much more results on the equivalence problem and quite a bit of them are mentioned.

The structure of the paper is as follows. Section 2 provides basic definitions. Section 3 discusses the main idea of the analysis technique and basic properties of the main object to be analyzed — a graph of consistent computations (GCC). The result of section 4 is a deep analysis of a GCC which allows to describe an polynomial-time equivalence-checking algorithm. The main theorem and the algorithm description are given in section 5.

2 Basic Definitions

Hereafter $\mathfrak{A}$ and $\mathfrak{L}$ are finite alphabets of basic assignment statements, or *actions* for short, and *logical conditions* respectively. A *propositional program* (PP)

$$\pi = (L, en, ex, T, B)$$

consists of: a finite set of *control locations* L including *entry* en and *exit* ex; a *transition function* $T : (L \setminus \{ex\}) \times \mathfrak{L} \to L$; a *binding function* $B : L \to \mathfrak{A}$. We write $l \xrightarrow{\delta}_{\pi} l'$ and $l \to_{\pi} l'$ instead of $T(l, \delta) = l'$ for readability. We define the *size* $|\pi|$ of a PP π to be the number of its control locations: $|\pi| = |L|$. The meaning of a PP is briefly discussed in the introduction except of the following: a logical condition may be understood as

a valuation for all Boolean conditions, which means that exactly one logical condition holds for each abstract data state.

To define semantics of PPs in general case we should use Kripke frames and models [5], but in the considered case (commutativity and left absorption for actions) we simplify it using the notation of semigroup theory. A *semigroup* $(S, *)$ consists of an arbitrary set S of *elements*, and a binary *associative operation* on S. A *monoid* $M = (S, *, \varepsilon)$ is a semigroup $(S, *)$ together with its *neutral element* ε. To define how a PP operates we use a monoid *on the set of actions*: each action from $\mathfrak{A}$ is an element of the monoid, and each other element s can be decomposed into actions ($s = a_1 * \cdots * a_k$, $k \geq 0$, $a_i \in \mathfrak{A}$ for $1 \leq i \leq k$). We write $[a_1 \dots a_k]$ to denote such s. Hereafter we use only monoids on the set of actions. An *interpretation* $I = (M, \xi)$ for a PP consists of a monoid $M = (S, *, \varepsilon)$, and a *valuation* $\xi : S \to \mathfrak{L}$ for logical conditions.

A *path* for a program π is a sequence of program locations of the form $l_1 \to_\pi l_2 \to_\pi \dots$. For paths pt', pt'', by $pt' \to pt''$ we denote a path which starts with a path pt' and continues with a path pt''. A path is called a *trace* if l_1 is the entry of π. A path is *full* if it either is infinite or leads to the exit of π. A full trace is called a *run*. A trace $l_1 \xrightarrow{\delta_1}_\pi l_2 \xrightarrow{\delta_2} \dots$ is called an *I-trace* for an interpretation I if $\delta_i = \xi([B(l_1) \dots B(l_i)])$ for $i \geq 1$. To simplify the notation, we write $B(l_1, \dots, l_i)$ along with $B(l_1) \dots B(l_i)$, and $[l_1, \dots, l_1]$ along with $[B(l_1, \dots, l_i)]$. The *result* of a finite I-run rn is the element $[rn]$. An infinite I-run loops and has the *indefinite result* $\perp$. Two programs are *I-equivalent* if the results of their I-runs are equivalent. For a monoid M two programs are *M-equivalent* if they are I-equivalent for each interpretation $I = (M, \xi)$.

The *equivalence problem* investigated in this paper is the M-equivalence problem for PPs where M is a monoid: given two PPs, to check if they are M-equivalent. The problem is solved for specific monoids M, which we call progressive CA-monoids. A monoid is *progressive* if for each triple of its elements s, s_1, s_2, if $s_1 \neq \varepsilon$ or $s_2 \neq \varepsilon$, then $s \neq s * s_1 * s_2$ Note that the elements of a progressive monoid are partially ordered w.r.t. its operation, i.e. $s_1 * s_2 \neq s_1$ for $s_2 \neq \varepsilon$. In other words, if we introduce the following order $\leq$: $s_1 \leq s_2$ iff $\exists s.\ s_2 = s_1 * s$ — then a monoid is ordered iff $\leq$ is a non-strict partial order. Along with $\leq$ we will use naturally induced orders $<$, $\geq$, $>$, as well as $s_1 \| s_2$ to denote the incomparability of s_1, s_2. A *CA-monoid* is a monoid generated by an arbitrary set of relations of the form $ab = ba$ and $ab = b$ ($a, b \in \mathfrak{A}$); the former one is a *commutativity relation*, the latter is a (left) *absorption relation*. A monoid *generated by a set of relations* can be defined as follows:

- each element is a set of words over the alphabet $\mathfrak{A}$;
- $g \in [h]$ iff g can be obtained from h with a finite number of rewritings using generating relations (a subword from the left part of a relation to the right part and vice versa).

The main result of the paper is the polynomial-time decidability of the M-equivalence problem for PPs for an arbitrary progressive CA-monoid M together with an explicit description of a decision algorithm. The description of the algorithm depends on a description of M, i.e. on a set of commutativity and absorption relations defining M. The algorithm is polynomial in time in size of input programs.

3 Graph of Consistent Computations

To simplify notation, hereafter we assume an arbitrary progressive CA-monoid $M = (S, *, \varepsilon)$ to be defined, as well as PPs

$$\pi_1 = (L_1, en_1, ex_1, T_1, B_1)$$

and

$$\pi_2 = (L_2, en_2, ex_2, T_2, B_2).$$

The main idea of how we analyze the M-equivalence problem for PPs is similar to those developed in [3, 4], [10], [17] and can be described as follows. Suppose we have an interpretation $I = (M, \xi)$. We construct the I-traces tr_1, tr_2 of π_1 and π_2 in the following manner: we start with one-element I-traces of π_1, π_2; if $[tr_1] = [tr_2]$, then we add the next location to both traces; if $[tr_1] > [tr_2]$, then we add the next location to tr_2; otherwise we add the next location to tr_1. If any of the traces tr_i is an I-run, then we do not add anything to it. As the result we have a sequence of pairs of traces leading either to the I-runs of tr_1, tr_2, or to an I-run and an I-trace. Consider any such pair $(tr_1 \to l_1, tr_2 \to l_2)$. To find what results can be obtained with computations starting in these traces, all we need to know is last program locations of these traces and data states they lead to. Thus, we replace this pair with a tuple $(l_1, l_2, [tr_1 \to l_1], [tr_2 \to l_2])$.

Hereafter we write $\lfloor s \rfloor$ to denote the set of all *minimal members* of s, $s \in S$: $\{g \mid g \in s,\ |g| = \min\limits_{u \in s} |u|\}$ — and $\|s\|$ to denote the length of words in $\lfloor s \rfloor$. Also, we write $\lfloor h \rfloor$ instead of $\lfloor [h] \rfloor$, and call a word $h \in \mathfrak{A}^*$ a *minimal member* if $h \in \lfloor h \rfloor$.

Let (l_1, l_2, s_1, s_2) be a tuple as described above, and s be a longest common prefix s of s_1, s_2: $s * s_1' = s_1$; $s * s_2' = s_2$; $\|s\|$ has a maximal possible value. To check the equivalence of programs, we do not need the results themselves, but to check if the results are equal. Thus, we replace s_1, s_2 with s_1', s_2' in the tuple. Due to the construction rules for tuples, $\|s_2'\| \leq 1$, which means that $s_2' = [\alpha]$ for $\alpha \in \mathfrak{A} \cup \{\lambda\}$. Then we replace (l_1, l_2, s_1, s_2) with a reduced tuple (l_1, l_2, s_1', α). Looking over all possible interpretations I, we connect all possible tuples in a graph Γ, which we call a graph of consistent computations (GCC) in the following way. The vertices of Γ are all possible tuples (l_1, l_2, s, α), where l_1, l_2 are locations of π_1, π_2, s is a data state, and $\alpha \in \mathfrak{A} \cup \{\lambda\}$. If there exists an interpretation I such that tuples t_1, t_2 are what we get from adjacent pairs of traces obtained with the construction rules for I, then t_1 and t_2 are connected with an edge. To eliminate the enumeration of all interpretations, we define Γ using M, π_1, and π_2 only, and then prove that to analyze Γ is sufficient to understand whether π_1, π_2 are M-equivalent.

To explain the term "consistent", we define the notion of consistency for traces of PPs in the following way. Traces of π_1, π_2 are *(M-)consistent* if they are I-traces for some interpretation $I = (M, \xi)$. Thus, an alternative description of the M-equivalence problem for PPs is: given two PPs, to check if for each pair of M-consistent runs their results are equal. The following proposition shows how to check trace consistency without enumerating of all possible interpretations.

Proposition 1 ([3]). *Let M be an progressive monoid, and tr_1, tr_2 be traces of π_1, π_2. Then tr_1, tr_2 are M-consistent iff for any their prefixes $tr_1' \to_{\pi_1} l_1$, $tr_2' \to_{\pi_2} l_2$ holds: if $[tr_1'] = [tr_2']$, then $\exists \delta$. $tr_1' \xrightarrow{\delta}_{\pi_1} l_1$ and $tr_2' \xrightarrow{\delta}_{\pi_2} l_2$.*

Now we give an alternative definition of Γ, which does not require enumeration of all interpretations. The nodes of Γ are 4-tuples of the form $v = (l_1, l_2, s, \alpha)$ where $l_i \in L_i$, $s \in S$, and $\alpha \in \mathfrak{A} \cup \{\lambda\}$. The *root* of Γ is the node (en_1, en_2, s, α) where: if $[B_1(en_1)] \leq [B_2(en_2)]$, then $s = \varepsilon$, otherwise $s = [B_1(en_2)]$; if $[B_2(en_2)] \leq [B_1(en_1)]$, then $\alpha = \lambda$, otherwise $\alpha = B_2(en_2)$. Some of the nodes (l_1, l_2, s, α) are *terminal* and have no outgoing edges:

1. $l_1 = ex_1$, $\alpha \neq \lambda$;
2. $l_2 = ex_2$, $s \neq \varepsilon$;
3. $l_1 = ex_1$, $l_2 = ex_2$, and either $s_1 \neq \varepsilon$, or $\alpha \neq \lambda$.

For cases (1), (2) (which are not mutually exclusive, but exclude (3)) we call a node *disproving* (the equivalence of PPs); Edges outgoing from a nonterminal node $v = (l_1, l_2, s, \alpha)$ carry the following labels (one edge for each $\delta \in \mathfrak{L}$): (δ, δ) if $l_1 \neq ex_1$, $l_2 \neq ex_2$, and $s = [\alpha] = \varepsilon$; (ϵ, δ) if $l_2 \neq ex_2$, $\alpha = \lambda$, and either $l_1 = ex_1$ or $s \neq \varepsilon$; (δ, ϵ) otherwise. Here ϵ is a special symbol, $\epsilon \notin \mathfrak{L}$. To finish the definition of Γ, we describe the node $v' = (l_1', l_2', s', \alpha')$ s.t. $v \xrightarrow{(\sigma_1, \sigma_2)} v'$. If $\sigma_1 = \epsilon$, then $l_1' = l_1$ and $s'' = s$, otherwise $l_1 \xrightarrow{\sigma_1} l_1'$ and $s'' = s * [B_1(l_1')]$. If $\sigma_2 = \epsilon$, then $l_2' = l_2$ and $\alpha'' = \alpha$, otherwise $l_2 \xrightarrow{\sigma_2} l_2'$ and $\alpha'' = B_2(l_2')$. If $s'' = [\alpha] * s'''$, then $s' = s'''$ and $\alpha' = \lambda$, othewise $s' = s''$ and $\alpha' = \alpha''$.

As the GCC is defined, we should prove that it contains as many traces as needed to check if π_1, π_2 are M-equivalent. To do it, we show the correspondence between paths in Γ and traces of π_1, π_2 (lemmas 1–3), and then show what paths should we search for to check whether π_1, π_2 are M-equivalent (theorem 1).

To shorten the notation, we give three more definitions. An *r-path* is a path in Γ originating from its root. *Projections* $pr_1(\omega)$, $pr_2(\omega)$ of an r-path ω are defined as follows. $pr_i((en_1, en_2, s, \alpha)) = en_i$. Let $\omega = \omega' \xrightarrow{(\sigma_1, \sigma_2)} (l_1, l_2, s, \alpha)$. Then:

- $pr_i(\omega) = pr_i(\omega')$ if $\sigma_i = \epsilon$;
- $pr_i(\omega) = pr_i(\omega') \xrightarrow{\sigma_i} l_i$ otherwise.

For an infinite r-path ω we define $pr_i(\omega)$ as the shortest r-path s.t. for each finite prefix ω'' of ω holds: $pr_i(\omega'')$ is a prefix of $pr_i(\omega)$. An r-path ω is *disproving* in two cases: it leads to a disproving node; it is infinite, and for some its (infinite) tail ω_t and some $i \in \{1, 2\}$ holds: for a label (σ_1, σ_2) of any edge of ω_t holds $\sigma_i = \epsilon$, and for any node (l_1, l_2, s, α) of ω_t the location ex_i is reachable from l_i.

Lemma 1. *Let ω be an r-path. Then $pr_1(\omega)$ and $pr_2(\omega)$ are consistent traces of π_1, π_2. Moreover, if ω leads to a node (l_1, l_2, s, α), then the following holds: there exists $s' \in S$ s.t. $s' * s = [pr_1(\omega)]$ and $s' * [\alpha] = [pr_2(\omega)]$; if $s \neq \varepsilon$ and $\alpha \neq \lambda$, then s and $[\alpha]$ are incomparable.*

Lemma 2. *Let rn_1, rn_2 be consistent runs of π_1, π_2. Then there is an r-path ω s.t. $pr_i(\omega)$ is a prefix of rn_i (for $i \in \{1, 2\}$), and either ω is infinite, or it leads to a terminal vertex.*

Lemma 3. *Let ω be an r-path, $i \in \{1,2\}$, and $rn_i = pr_i(\omega) \to_{\pi_i} pt$ be a run of π_i. Then rn_i and $pr_{3-i}(\omega)$ are consistent traces.*

Lemmas 1, 2 follow from proposition 1, the left-cancellativity of M, and the definition of Γ. To prove lemma 3 by contradiction, we note that if rn_i and $pr_{3-i}(\omega)$ are not consistent, then due to proposition 1 and lemma 1 there exist a proper prefix pt' of a path pt and a proper prefix tr'_{3-i} of $pr_{3-i}(\omega)$ s.t. $[pr_i(\omega) \to_{\pi_i} pt'] = [tr'_{3-i}]$. Then by lemma 1 there does not exist a prefix ω' of ω s.t. $pr_i(\omega) = tr'_{3-i}$, which is impossible by the definition of pr_i.

Theorem 1. *PPs π_1, π_2 are M-equivalent iff Γ contains no disproving paths.*

Proof. Necessity. If Γ contains a finite disproving path ω, then by lemma 3 the traces $pr_1(\omega)$, $pr_2(\omega)$ can be extended to consistent runs of π_1, π_2, and by lemma 2 the results of these runs are distinct. If Γ contains an infinite disproving path ω and i-th components of edge labels of some its tail are equal to ϵ, then the trace $pr_i(\omega)$ can be extended to a finite run rn_i of π_i, while pr_{3-i} is an infinite run of π_{3-i}, and by lemma 3 the runs rn_i and pr_{3-i} are consistent and have distinct results.

Sufficiency. If rn_1, rn_2 are consistent runs of π_1, π_2 with distinct results, then by lemma 2 we obtain an r-path ω which is either finite and leads to a disproving vertex due to lemma 1, or infinite, in which case exactly one of runs rn_1, rn_2 is infinite, and ω is an infinite disproving path.

Note that theorem 1 does not solve the M-equivalence problem for PPs, as Γ is infinite in general case. The next section provide means to build a polynomial-time traversal of Γ sufficient to solve the equivalence problem.

4 Analysis of Graph of Consistent Computations

The main idea of the further GCC analysis is to partition its nodes into a finite number of classes and prove that during the traversal it is sufficient to visit polynomially many nodes in each class, and after it we state either that π_1 and π_2 are not M-equivalent (lemma 7) or that other nodes in the class may be ignored in the further traversal (lemma 8). To do it, we introduce and analyze the notion which we called an absorption effect. An *(absorption) effect* induced by an element $s \in S$ is a function $æ_s : S \to S$ showing how the elements of S are changed under "pressure" of absorption of s appended to the right. Formally, $æ_s(s_1) = \underset{s_2 : s_1 * s = s_2 * s}{\arg\min} \|s_2\|$. To prove the soundness of the definition, we state a (what we call) *commutativity of minimals*: if h, g are minimal members of the same element of M, then g can be obtained from h with commutativity rewritings only (without any absorption). The soundness is provided by the following lemma.

Lemma 4. *An equation $X * s = s'$ has at most one solution s'' on X s.t. $\|s''\|$ has minimal possible value among all solutions.*

Proof. Suppose we have at least two such solutions: s''_1, and s''_2: $s''_1 * s = s' = s''_2 * s$. Let $h_1 \in \lfloor s''_1 \rfloor$, $h_2 \in \lfloor s''_2 \rfloor$, and $h \in \lfloor s \rfloor$. Due to the commutativity of minimals, we

can obtain $h_2 h$ from $h_1 h$ using commutativity relations only. By symmetry of commutativity relation, we get the derivation of $h^- h_2^-$ from $h^- h_1^-$ (g^- is a reversed g). By the left-cancellativity of M and the same symmetry, h_1 can be rewritten to h_2 with commutativity only, which means $s_1'' = s_2''$.

The next step is to introduce a partial order on the set of all effects (which we denote by): $æ_1 \leq æ_2$ iff there exist $s_1, s_2 \in S$ s.t. $s_1 \leq s_2$, $æ_1 = æ_{s_1}$, and $æ_2 = æ_{s_2}$. To prove that $\leq$ is a partial order, we state one more algebraic property of M, which we call the *inductive minimality*: if h is a minimal member, $a \in \mathfrak{A}$, and $[ah] \neq [h]$, then ah is a minimal member.

Lemma 5. *The relation $\leq$ over is a non-strict partial order.*

Proof. By the inductive minimality, an effect æ determines a set of symbols deleted from a minimal member: if $æ(s) = s'$ and $h \in \lfloor s \rfloor$, then there exists $h' \in \lfloor s' \rfloor$ s.t. h' can be obtained from h by deleting some symbols. If there exist $æ_1$, $æ_2$ s.t. $æ_1 \leq æ_2$ and $æ_2 \leq æ_1$, then $æ_1$ deletes at least those symbols which are deleted by $æ_2$ and vice versa. It means that for any minimal member h the effects $æ_1$ and $æ_2$ delete the same symbols from h, and thus $æ_1([h]) = æ_2([h])$.

Another pleasant property of the effects is that they can be (in some sense) computed by a finite automaton. It means that the set is finite, and that having a short description of an effect $æ_s$ and an action a, we can easily compute the description of $æ_{s*[a]}$. An *automaton* $A = (Q, q_0, T_A, S_A, M_A)$ (over the alphabet $\mathfrak{A}$) consists of: a finite set of *states* Q including an *initial state* q_0; a *transition function* $T : Q \times \mathfrak{A} \to Q$; a set of *labels* S_A; a *marking function* $M_A : Q \to S_A$. We denote by $A(q, h)$ a state to which A jumps reading the word $h \in \mathfrak{A}^*$, and use the following shortenings: $A(h)$ is $A(q_0, h)$; $M_A(q, h)$ is $M_A(A(q, h))$; $M_A(h)$ is $M_A(A(h))$. The notion of automaton used in this paper is a minor generalization of a well-known finite deterministic acceptor with the only difference: instead of answers "yes" and "no", here S_A is the set of all possible answers. We put the following meaning in the phrase "to compute an effect": $|S_A| = ||$; $æ_{[h]} = æ_{[g]}$ iff $M_A(h) = M_A(g)$; if $[h] = [g]$, then $A(h) = A(g)$. We call an automaton satisfying all these properties an *æ-automaton*. Note that intuitively the best way to define such an automaton is to say $S_A =$, but formally the set cannot be used explicitly in any algorithm as it contains functions with an infinite domain. The description of an æ-automaton is based on an *automata-based recognition of absorption*: there exists an automaton with the following two properties: reading an input word h, it enters a state labelled with the set $\{a \in \mathfrak{A} \mid [a] * [h^-] = [h^-]\}$; reading words h, g s.t. $[h^-] = [g^-]$, it enters the same state.

Lemma 6. *For any progressive CA-monoid there exists an æ-automaton.*

Proof. We start with an automaton A stated by the automata-based recognition of absorption. Then we divide the states of A into equivalence classes: q and q' are equal iff $\forall h \in \mathfrak{A}^*$. $M_A(q, h) = M_A(q', h)$. Note that we have a finite number N of such classes. Changing the labels of A to $\{1, \ldots, N\}$, enumerating the equivalence classes, and assigning the label i to the i-th class, we get an automaton A' s.t. $M_{A'}(h) = M_{A'}(g)$ iff $æ_{h^-} = æ_{g^-}$. Then by changing the labels again we construct exactly N finite-state

acceptors, the i-th of which says if the original answer is i. Using classical automata-theoretic results [8], we reverse the language of each acceptor, construct their Cartesian product and reassign labels back to $\{1, \ldots, N\}$, and the result is exactly a required æ-automaton: $M_{A'}(h) = M_{A'}(g)$ iff $æ_h = æ_g$, and all states were constructed for elements of S, thus all parts of the definition are satisfied.

Hereafter we assume to be given an æ-automaton $A = (Q, q_0, T_A, S_A, M_A)$. By the definition of A, we write $A([h])$ along with $A(h)$. The last two notions required for the analysis of Γ are an effect evolution and a generalized effect evolution. Intuitively, these notions show how an absorption effect is modified during a program run from some intermediate run location. An *effect evolution* $\dot{æ}_s^h$ along the word $h = a_1 \ldots a_k \in \mathfrak{A}^*$ induced by $s \in S$ is defined as follows. We start with the sequence $(æ_s, æ_{s*[a_1]}, \ldots, æ_{s*[h]})$. Then we delete adjacent duplicates. The result is exactly $\dot{æ}_s^h$. Note that any evolution is a chain in a finite (lemma 6) partially ordered (lemma 5) set, and thus the number of evolutions is finite. A *generalized effect evolution* $\ddot{æ}_q^h$ along the word $h = a_1 \ldots a_k \in \mathfrak{A}^*$ induced by a state $q \in Q$ is defined in the same way except we start with the sequence of pairs $((æ_\varepsilon, æ_{M_A(q)}), (æ_{[a_1]}, æ_{M_A(q,a_1)}), \ldots, (æ_{[h]}, æ_{M_A(q,h)}))$. For the same reasons the set of all generalized evolutions is finite.

There is an important property of evolutions. For any full path pt for a PP $\pi = (L, en, ex, T, B)$ and any $s \in S$ there exists a full path pt' (which we call *reduced*) s.t.: $\dot{æ}_s^{B(pt)} = \dot{æ}_s^{B(pt')}$; if pt is finite, then $|pt'| \leq |L| \cdot |Q|$; if pt is infinite, then $pt' = pt'_1 \to pt''_2 \to pt''_2 \to \ldots$ where $|pt'_1| + |pt''_2| \leq |L| \cdot |Q|$. To prove the existence of a reduced path pt', it is sufficient to: mark pt with states of A starting with $A(s)$ modifying it with the actions of pt; delete loops between two repeated pairs of a program location and an automaton state; for an infinite path, replace an infinite tail corresponding to the last effect in the evolution with the repetition of a loop described above. The same property holds for generalized evolutions with an upper bound $|L| \cdot |Q|^2$ instead of $|L| \cdot |Q|$. The following lemmas are the result of the analysis of Γ, as they provide a finite partitioning of the nodes of Γ required in the equivalence-checking algorithm.

Lemma 7. *Let: $q \in Q$; $l_1 \in L_1$; $l_2 \in L_2$; $\alpha \in \mathfrak{A} \cup \{\lambda\}$; $m = |Q|^3 \cdot |\pi_1| \cdot |\pi_2|^2 + 2$; pairwise-distinct GCC nodes (l_1, l_2, s_i, α), $i \in \{1, \ldots, m\}$, be reachable from the GCC root; $A(s_1) = \cdots = A(s_m) = q$; $l_i \to pt_i$ be a full path of π_i ($i \in \{1, 2\}$), and at least one of these paths is finite; $\ddot{æ}_q^{B_1(pt_1)} = ((æ_1^1, æ_1^2), \ldots, (æ_k^1, æ_k^2))$; $\dot{æ}_\alpha^{B_2(pt_2)} = (\dot{æ}_1, \ldots, \dot{æ}_p)$; $1 \leq k' \leq k$; the elements $æ_{k'}^1(s_i)$, $i \in \{1, \ldots, m\}$, be pairwise distinct; $\{æ_1, \ldots, æ_p\} \cap \{æ_{k'+1}^2, \ldots, æ_k^2\} = \emptyset$.*

Then π_1 and π_2 are not M-equivalent.

Proof. The main idea is to show that at least for one of the nodes w_j traces of a projection of a path ω_j leading to this node can be expanded into consistent runs with different results. We show in details only one case (for other cases a proof technique works in the same way, though it looks a bit more complex): pt_1 and pt_2 are finite, $k' = k$. W.l.o.g. we assume pt_1 and pt_2 to be reduced paths, thus $|pt_i| \leq |\pi_i| \cdot |Q|^{3-i}$. By proposition 1 the only way for the runs not to be consistent is if their proper prefixes lead to a common element s, and next steps are made with distinct conditions δ_1, δ_2

— we call it a collision. By lemma 3, for each ω_p these prefixes are not shorter than the projections of ω_p. Thus, we have $[pr_1(\omega_p) \to pt'_1] = [pr_2(\omega_p) \to pt'_2]$ for some proper prefixes pt'_1, pt'_2 of pt_1 and pt_2. But as the elements $æ^1_k(s_p)$, $æ^1_k(s_r)$ are distinct for $p \neq r$, the elements $[pr_1(\omega_p) \to pt'_1]$ and $[pr_1(\omega_p) \to pt'_1]$ are also distinct (as M is left-cancellative), while $[pr_2(\omega_p) \to pt'_2] = [pr_2(\omega_r) \to pt'_2]$. Thus, a collision for different indexes p, r cannot happen for the same prefixes pt'_1, pt'_2 simultaneously. On the other hand, we have at most $|pt_1| \cdot |pt_2| \leq |\pi_1| \cdot |\pi_2| \cdot |Q|^3$ possible pairs of lengths for these prefixes, which means that at least for two indexes j', j'' there are no collisions: in other words, expansions for these indexes are consistent. Based on the same thought about distinction of $æ^1_k(s_{j'})$, $æ^1_k(s_{j''})$ as above for p, r, we conclude that at least for one of indexes j', j'' the results of computations are distinct.

Lemma 8. *Let: $q \in Q$; $l_1 \in L_1$; $l_2 \in L_2$; $\alpha \in \mathfrak{A} \cup \{\lambda\}$; $m = |Q|^3 \cdot |\pi_1| \cdot |\pi_2|^2 + 2$; pairwise-distinct GCC nodes $v_i = (l_1, l_2, s_i, \alpha)$, $i \in \{1, \ldots, m\}$, are reachable from the GCC root; $A(s_1) = \cdots = A(s_m) = q$; $l_i \to pt_i$ is a full path of π_i, $i \in \{1, 2\}$; $\ddot{æ}_q^{B_1(pt_1)]} = ((æ^1_1, æ^2_1), \ldots, (æ^1_k, æ^2_k))$; $\dot{æ}_\alpha^{B_2(pt_2)} = (æ_1, \ldots, æ_s)$; $1 \leq k' < k$, $1 \leq s' < s$; the elements $æ^1_{k'}(s_i)$, $i \in \{1, \ldots, m\}$, be pairwise different; $æ^1_{k'+1}(s_1) = \cdots = æ^1_{k'+1}(s_m)$; $\{æ_1, \ldots, æ_{s'}\} \cap \{æ^2_{k'+1}, \ldots, æ^2_k\} = \emptyset$; $æ_{s'+1} \in \{æ^2_{k'+1}, \ldots, æ^2_k\}$; ω be a GCC r-path leading to v_m; $pr_1(\omega) \to pt_1$ and $pr_2(\omega) \to pt_2$ be consistent runs having distinct results.*

Then there exists j, $1 \leq j < m$, s.t. v_j belongs to some disproving r-path.

We prove lemma 8 using the same technique as for lemma 7, so due to lack of space we omit the proof.

5 Equivalence-Checking Algorithm

The resulting equivalence-checking algorithm is to traverse Γ in any way. During the traversal visited nodes are partitioned into classes defined by a state q, locations l_1, l_2, and a symbol α (as in lemmas 7, 8): the nodes (l_1, l_2, s, α) and $(l'_1, l'_2, s', \alpha')$ belong to the same class iff $l_1 = l'_1$, $l_2 = l'_2$, $A(s) = A(s')$, and $\alpha = \alpha'$. If the traversal is finished, then the absence of disproving paths can be checked directly, and the answer is given by theorem 1. If $(|Q|^3 \cdot |\pi_1| \cdot |\pi_2|^2 + 2)$ nodes are visited in any class, then either the non-equivalence is stated (lemma 7), or all other nodes in the class are ignored from now on (lemma 8).

Let N be the total program size: $N = |\pi_1| + |\pi_2|$. The number of distinct node classes is $O(N^2)$. Each class contains $O(N^3)$ nodes. Thus, no more than $O(N^5)$ nodes are visited during the traversal. A minimal member $g \in \lfloor h \rfloor$ can be constructed in time $O(|h|)$ using the inductive minimality and lemma 6. Checking if $[h] \leq [g]$ can be done in time $O((|h| + |g|)^2)$: construct a word $g' \in \lfloor g \rfloor$; if $h = \lambda$, then $[h] \leq [g]$; otherwise $h = ah'$, where $a \in \mathfrak{A}$; if $[ag] = [g]$ (it can be checked with A), then delete a from h and repeat the procedure; otherwise check whether g contains a letter a; if not, then $[h] \not\leq [g]$; if yes, then pick the leftmost occurence of a in g: $g = g'ag''$; if each symbol from g' commutes with a, then delete marked a from h and g and repeat the procedure; otherwise $[h] \not\leq [g]$. The soundness of the latter algorithm is *proved in the appendix*. The elimination of a longest prefix can be performed in a similar way,

the difference is that we pick not the leftmost symbol of h but every symbol which can be shifted to the leftmost position with commutativity relations, and every completion is successfull. Each element of M can be encoded as any its member (e.g. constructed by a computation). During the traversal the algorithm constructs words h of length at most $O(N^5)$, and each element comparison can be done in time $O(N^{10})$. For each vertex the algorithm does at most $O(N^3)$ comparisons of elements of M, and this is the bottleneck of the algorithm. Thus, the algorithm performs $O(N^5)$ steps, and each step is done in time $O(N^{13})$, and the total time is $O(N^{18})$. The last upper bound together with theorem 1 and lemmas 7, 8 states the validity of the main theorem of this paper.

Theorem 2. *Let M be a progressive CA-monoid on a finite set of actions. Then the M-equivalence problem for propositional programs is decidable in time polynomial in size of programs.*

6 Conclusion

We want to present three thoughts concluding the paper. The first one is that the considered case: propositional programs, commutativity, and left absorption — appears to be simple enough to have a polynomial-time decidability for the equivalence problem. The second one is that (nevertheless) this case is complex enough to require a rather nontrivial technique for its analysis. Finally, the third one is that it was mentioned in the introduction that there is still a long way to go, and this statement remains unchanged, as even in a simple example in the introduction we need not only commutativity and absorption (but also dependencies between statements and conditions) to prove the equivalence. To be able to prove the equivalence of "real" fragments of code, we should consider other properties of program primitives, which is the topic of future research.

References

1. Rice, H.G.: Classes of recursively enumerable sets and their decision problems. Trans. Amer. Math. Soc. 89, 25–59 (1953)
2. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques and Tools. Pearson Education, Inc., New York (2006)
3. Zakharov, V.A.: An efficient and unified approach to the decidability of equivalence of propositional program schemes. Lect. Notes Comput. Sci. 1443, 247–258 (1998)
4. Zakharov, V.A., Zakharyaschev, I.M.: On the equivalence checking problem for a model of programs related with muti-tape automata. Lect. Notes Comput. Sci. 3317, 293âĂŞ-305 (2005)
5. Harel, D., Kozen, D., Tiuryn J.: Dynamic Logic. The MIT Press (2000)
6. Hunt, H.B., Constable, R.L., Sahni, S.: On the computational complexity of program scheme equivalence. SIAM Journal of Computing. 9, N 2, 396–416 (1980)
7. Rutledge, J.D.: On Ianov's program schemata. J. Assoc. Comp. Mach. 11, 1–9 (1964)
8. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to automata theory, languages, and computation. Addison-Wesley, Boston (2003)
9. Kozen, D.: Kleene Algebra withTests and Commutativity Conditions. In: Tools and Algorithms for Construction and Analysis of Systems, Second International Workshop, pp. 14–33 (1996)

10. Zakharov, V.A.: On the decidability of the equivalence problem for monadic recursive programs. Theoretical Informatics and Applications. 34, N 2, 157–171 (2000)
11. Glushkov, V.M., Letichevskii, A.A.: Theory of algorithms and discrete processors. Adv. Inf. Syst. Sci. 1, 1–58 (1969)
12. Podymov, V.V., Zakharov, V.A.: On a semigroup model of sequential programs specified by means of two-tape automata. Belgorod State University Scientific Bulletin. History. Political science. Economics. Information technologies. 7(78), N 14/1, 94–101 (2010) (in Russian)
13. Bohm, S., Goller, S, Jancar, P.: Equivalence of deterministic one-counter automata is NL-complete. Proceedings of the forty-fifth annual ACM symposium on Theory of computing. 131–140 (2013)
14. Debski, W., Fraczak, W.: Concatenation state machines and simple functions. Lecture Notes in Computer Science. 3317, 113–124 (2004)
15. Bastien, C., Czyzowicz, J., Fraczak, W., Rytter, W.: Prime normal form and equivalence of simple grammars. Theoretical Computer Science. 363, N 2, 124–134 (2006)
16. Harju, T., Karhumäki, J.: The equivalence problem of multitape finite automata. Theoretical Computer Science. 78, N .2, pp. 347–355 (1991)
17. de Souza, R.: On the decidability of the equivalence for k-valued transducers. Lecture Notes in Computer Science. 5257, 252–263 (2008)