Games for complexity of second-order call-by-name programs

Andrzej S. Murawski

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD, UK

Abstract

We use game semantics to show that program equivalence and program approximation in a second-order fragment of Idealized Algol are PSPACE-complete. The result relies on a PSPACE construction of deterministic finite automata representing strategies defined by second-order programs and is an improvement over the at least exponential space bounds implied by work of other authors in which extended regular expressions were used.

The approach makes it possible to study the contribution of various constructs of the language to the complexity of program equivalence and demonstrates a similarity between call-by-name game semantics and call-by-name interpreters.

Key words: game semantics, computational complexity, program analysis

1 Introduction

Game semantics views computation as an exchange of moves between two players, who represent respectively the program under evaluation and the environment in which the program is evaluated. Programs can then be interpreted as strategies for the first player. This approach has led to the construction of first *fully abstract* models for a variety of programming languages, i.e. models in which the interpretations of two programs coincide if and only if the programs are equivalent [1–8]. The game models provide a semantic characterization of program equivalence and make it possible to recast questions about equivalence of programs as semantic problems. However, reasoning about programs with game models is not so easy, especially if one has automation in mind.

Email address: `Andrzej.Murawski@comlab.ox.ac.uk` (Andrzej S. Murawski).

$$\frac{}{\Gamma \vdash \mathit{skip} : \mathit{com}} \qquad \frac{i \in \{\, 0, \cdots, \mathit{max} \,\}}{\Gamma \vdash i : \mathit{exp}} \qquad \frac{}{\Gamma \vdash \Omega_B : B}$$

$$\frac{}{\Gamma, x : T \vdash x : T} \qquad \frac{\Gamma \vdash M : \mathit{exp}}{\Gamma \vdash \mathbf{succ}(M) : \mathit{exp}} \qquad \frac{\Gamma \vdash M : \mathit{exp}}{\Gamma \vdash \mathbf{pred}(M) : \mathit{exp}}$$

$$\frac{\Gamma \vdash M : \mathit{exp} \quad \Gamma \vdash N_0 : B \quad \Gamma \vdash N_1 : B}{\mathbf{ifzero}\, M N_0 N_1 : B}$$

$$\frac{\Gamma \vdash M : \mathit{com} \quad \Gamma \vdash N : B}{M; N : B} \qquad \frac{\Gamma \vdash M : \mathit{exp} \quad \Gamma \vdash N : \mathit{com}}{\mathbf{while}\ M\ \mathbf{do}\ N : \mathit{com}}$$

$$\frac{\Gamma \vdash M : \mathit{var}}{\Gamma \vdash\ !M : \mathit{exp}} \qquad \frac{\Gamma \vdash M : \mathit{var} \quad \Gamma \vdash N : \mathit{exp}}{\Gamma \vdash M := N : \mathit{com}}$$

$$\frac{\Gamma, X : \mathit{var} \vdash M : B}{\mathbf{new}\ X\ \mathbf{in}\ M : B} \qquad \frac{\Gamma \vdash M : \mathit{exp} \to \mathit{com} \quad \Gamma \vdash N : \mathit{exp}}{\Gamma \vdash \mathbf{mkvar}(M, N) : \mathit{var}}$$

$$\frac{\Gamma, x : B \vdash M : T}{\Gamma \vdash \lambda x^B . M : B \to T} \qquad \frac{\Gamma \vdash M : B \to T \quad \Gamma \vdash N : B}{\Gamma \vdash MN : T}$$

Fig. 1. Syntax of IA_2.

Firstly, to achieve full abstraction, equivalence classes of strategies need to be considered instead of strategies, and in general the relation involved (the so-called *intrinsic* preorder) is very intricate. Secondly, positions arising in game semantics are not merely sequences of moves. In addition, they are endowed with pointers that connect moves subject to a number of combinatorial constraints.

The case of Idealized Algol in which expressions may have side effects is much more satisfying. There, the above-mentioned quotient set admits a direct characterization based on *complete* plays - plays that correspond to terminating computations. Consequently, the first obstacle is removed: questions about program equivalence (respectively approximation) can be restated as equivalence (respectively containment) queries for the induced sets of complete positions. Moreover, when one restricts the language to second order, positions can be treated as strings of moves, because the pointer structure is uniquely reconstructible and hence redundant. Then it turns out that complete plays generated by second-order programs form regular languages [9], which immediately implies decidability of second-order program equivalence and approximation, because the problems of equivalence and containment of regular languages are decidable.

Two expositions of the regular game semantics exist [9,10], both employing a class of semi-extended regular expressions with intersections to describe the sets of complete plays generated by programs. Because the equivalence and containment problems for such expressions are known to be EXPSPACE-complete, one might suspect that the corresponding problems concerning programs will inherit this complexity (intersections are crucial for modelling state). In this paper we show that this is not the case: program approximation as well as program equivalence in the fragment of Idealized Algol considered in these papers are in fact both PSPACE-complete.

Our approach consists of a direct construction of deterministic automata which represent the game semantics of programs. In order to avoid the use of exponential space this process has two stages: first we construct the automaton corresponding to programs in which state changes are not observed, then we refine it so that state changes are respected. Because the construction is conducted in polynomial space, and both equivalence and containment of deterministic automata are NL-complete, one can obtain a PSPACE algorithm for program approximation and equivalence by combining the two in a careful way.

To our knowledge this is the first time a complexity result like this has been proved using a denotational model.

1.1 Idealized Algol

Idealized Algol (IA) is the canonical language combining functional and imperative programming. We shall concern ourselves with its fragment, called IA_2, in which free identifiers are of base type or (first-order) function type and arguments to procedures are of base type. IA_2 types (denoted by T) are generated by the following grammar:

$$B ::= \quad com \quad | \quad exp \quad | \quad var \qquad\qquad T ::= \quad B \quad | \quad B \rightarrow T.$$

Those generated from B are called base types. *com* is the type of commands, *exp* is the type of expressions. We assume that values of type *exp* are taken from a finite initial segment $\{ 0, \cdots, max \}$ of natural numbers ($max > 0$). *var* is the type of mutable variables in which only values of type *exp* can be stored. In what follows we will continue to use B if we want to stress that a certain type is a base type; otherwise we will use T.

IA_2 typing judgments are of the form $\Gamma \vdash M : T$ where $\Gamma = \{ x_1 : T_1, \cdots, x_n : T_n \}$. All the typing rules are shown in Figure 1. Given $\Gamma \vdash M : T$ where $T = B_1 \rightarrow \ldots \rightarrow B_k \rightarrow B$ we will say that the *arity* of M is k (which will be written as $ar(M) = k$). $|M|$ will denote the size of M. Where X is a set, $|X|$ means its cardinality; if s is a sequence of characters (or moves), $|s|$ is its length. $\mathrm{FV}(M)$ will denote the set of free identifiers of M, i.e. $\{ x_1, \cdots, x_n \}$.

We consider the *active* variant of Idealized Algol in which commands may be combined with other terms of base types to generate side effects (the more restrictive version in which expressions cannot have side effects cannot be characterized using complete plays [8]). It is also possible to generate variable objects with **mkvar** so that they have non-standard writing and reading 'methods'. We assume that the initial value of a mutable cell is 0, $\mathbf{pred}(0)$ and $\mathbf{succ}(max)$ are undefined but other conventions (e.g. $\mathbf{pred}(0) = 0, \mathbf{succ}(max) = 0$) can be accommodated with ease. The operational semantics of the full language is based on call-by-name evaluation and can be found in [3]. For instance, in order to evaluate $\mathbf{ifzero}\, MN_0N_1$ one must evaluate M first and if the result is i, $N_{i \bmod 2}$ should be evaluated next to yield the final result for $\mathbf{ifzero}\, MN_0N_1$. We write $M \Downarrow$ if M is a closed term of type *com* which evaluates to *skip*.

Definition 1 Two terms $\Gamma \vdash M_1, M_2 : T$ are *equivalent* ($\Gamma \vdash M_1 \cong M_2$) if for any context $C[\cdot]$ such that $C[M_1], C[M_2]$ are closed terms of type *com*, we have $C[M_1] \Downarrow$ if and only if $C[M_2] \Downarrow$. Similarly, M_1 *approximates* M_2 ($\Gamma \vdash M_1 \precsim M_2$) iff for all contexts satisfying the properties above whenever $C[M_1]$ terminates so does $C[M_2]$.

Note that the contexts may come from outside IA_2, which is necessary to test procedures. It turns out that the presence of **mkvar** in the context does not make a difference as far as equivalence is concerned, but it does affect program approximation [11].

1.2 Game semantics

We give a brief overview of the game model of IA [3] focussing on the elements relevant to modelling IA_2 (for a more complete tutorial introduction we recommend [12]).

The games used to model IA types are two-player games between O (Opponent) and P (Proponent) in which the players make moves alternately. Opponent is the player to be associated with the environment (he begins), whereas Proponent makes moves representing actions of the program. There are two kinds of moves: *questions* and *answers*. Each question comes with a set of possible answers. Whenever an answer-move is played, it must be an answer to the latest unanswered question - this is called the *well-bracketing* condition. The games corresponding to IA types are built from the games interpreting base types using the product and function space constructions. In the game $[\![com]\!]$, interpreting the command type, O can play *run* to which P may only reply with *done*. In $[\![exp]\!]$ after O plays the initial question q, P can play any $i \in \{0, \cdots, max\}$ as an answer. In $[\![var]\!]$ there are two kinds of plays: $write(i)\ ok$ and $read\ j$ ($i, j = 0, \cdots, max$) which are used to model assignment and dereferencing respectively. In general, in order to define positions and various game constructions one needs to use justification pointers (from each non-initial move of one player to a previous move of the other), but in the second-order case they are uniquely reconstructible and can be omitted.

Function types are interpreted using the function space game $A \Rightarrow B$, which involves moves from both A and B as a disjoint sum: those from B are still assigned to the same players, those from A change owners (any O-move in A becomes a P-move in $A \Rightarrow B$ and vice versa). Each play of $A \Rightarrow B$ begins in B and consists of a play in B intertwined with plays of A, but it is only P who can switch between the plays in A or between a play in A and a play in B. Product games are used for modelling contexts: in $A \times B$ all moves from A and B are available (again as a disjoint sum). They belong to the same players as in the original games. Plays in $A \times B$ are either plays from A or plays from B. It is the initial move that decides in which subgame the play will proceed. However, in the game $A \times B \Rightarrow C$, many plays from $A \times B$ may already occur: some of them may be from A and some from B. The games $A \times B \Rightarrow C$ and $A \Rightarrow (B \Rightarrow C)$ are actually identical.

Strategies for a given game A (written as $\sigma : A$) are prefix-closed subsets of plays which indicate P's responses. For IA only deterministic strategies need to be considered: whenever $sp_1, sp_2 \in \sigma$ and p_1, p_2 are P-moves, we have $p_1 = p_2$. In contrast, all possible O-moves are taken into account in the specification of a strategy: if $s \in \sigma$, $|s|$ is even and s can be extended (to a valid play) with an O-move o, then $so \in \sigma$. IA terms $x_1 : T_1, \cdots, x_n : T_n \vdash M : T$ are interpreted by strategies (denoted $[\![x_1 : T_1, \cdots, x_n : T_n \vdash M : T]\!]$) for the game $[\![T_1]\!] \times \cdots \times [\![T_n]\!] \Rightarrow [\![T]\!]$. We present most of the special strategies used to interpret IA_2 in Figure 2, where m_q, m_a stand for any question-answer pair available in the relevant game.

Games and strategies form a category where morphisms between two games A and B are strategies for the game $A \Rightarrow B$. The identity strategy $id_A : A \Rightarrow A$ simply tells P to copy moves made by O between the two copies of A (since the first move can only occur on the right, P will then copy it to the left instance of A). An *interaction sequence* of two strategies $\sigma : A \Rightarrow B$ and $\tau : B \Rightarrow C$ is a sequence of moves from A, B and C such that when moves from A are erased one gets a position from τ and when moves from C are erased one gets an interleaving of several positions from σ. The strategy $\sigma; \tau : A \Rightarrow C$ is then defined by positions that arise from interaction sequences after erasing moves from B. The product game indeed defines products: pairing $(\langle \sigma, \tau \rangle : C \Rightarrow A \times B)$ of two strategies $\sigma : C \Rightarrow A$ and $\tau : C \Rightarrow B$ amounts to taking $\sigma + \tau$. Similarly, the function space construction makes the category cartesian closed. Because $A \times B \Rightarrow C$ and $A \Rightarrow (B \Rightarrow C)$ are identical, the currying and uncurrying operations are essentially identities. With the structure outlined above, IA_2 terms can be interpreted compositionally by using the identity strategies for free identifiers and interpreting other constructs $\Gamma \vdash \mathbf{op}(M_1, \cdots, M_k)$ by

$$\langle\, [\![\Gamma \vdash M_1]\!], \cdots, [\![\Gamma \vdash M_k]\!]\,\rangle; [\![\mathbf{op}]\!]$$

where $[\![\mathbf{op}]\!]$ is a suitable strategy from Figure 2. For **while**, one uses the strategy $[\![\mathbf{while}]\!] : [\![exp]\!] \times [\![com]\!]_1 \Rightarrow [\![com]\!]_2$ with positions of the shape

$$run_2\ (q\ (\sum_{i=1}^{max} i)\ run_1\ done_1)^*\ q\ 0\ done_2$$

where the subscripts refer to the origin of the moves: $[\![com]\!]_1$ or $[\![com]\!]_2$.

Example 2 Any IA term $\Gamma, X : var \vdash M : B$ defines a strategy for $G = [\![\Gamma]\!] \times [\![var]\!] \Rightarrow [\![B]\!]$. Each play of G, restricted to the $[\![var]\!]$ subgame, is a sequence of $write(i)\, ok$ and $read\ j$ segments and there is no connection between *read*'s and preceding *write*'s. Variable binding (**new**) is interpreted by constraining $[\![\Gamma, X : var \vdash M : B]\!]$ to sequences in which each *read* is followed by the value used in the most recent $write(i)$ move (or 0 if no *write* has taken place yet) and subsequently hiding (erasing) all the $read, i, write(j), ok$ moves.

A non-empty position s is called *complete* if all questions in s are answered (for games generated by IA types this is equivalent to maximality). Given a strategy σ we denote its subset of complete positions by $\mathsf{comp}(\sigma)$. As we have mentioned at the very beginning, such positions characterize IA program approximation and equivalence.

Theorem 3 ([3]) *Suppose* $\Gamma \vdash M_1, M_2 : T$. *Then we have:*

$$\begin{array}{rcl} \Gamma \vdash M_1 \sqsubsetsim M_2 & \mathit{iff} & \mathsf{comp}([\![\Gamma \vdash M_1]\!]) \subseteq \mathsf{comp}([\![\Gamma \vdash M_2]\!]), \\ \Gamma \vdash M_1 \cong M_2 & \mathit{iff} & \mathsf{comp}([\![\Gamma \vdash M_1]\!]) = \mathsf{comp}([\![\Gamma \vdash M_2]\!]). \end{array}$$

If we can represent positions as words of a language, then program approximation and equivalence correspond to the well-studied problems of language containment and equivalence. Complete plays induced by IA_2 programs turn out to be representable by regular languages [9]. Hence, IA_2 program approximation (respectively equivalence) can be shown decidable by a reduction to the containment (respectively equivalence) problem for a class of extended regular expressions [9,10]. In order to estimate the complexity of the algorithms implied by these papers, we review the relevant results about regular languages ([13] contains a compendium of such results and original references).

Theorem 4 *The language containment and equivalence problems are:*

- *NL-complete for deterministic finite automata,*

	$[\![skip]\!] : [\![com]\!]$
O	run
P	$done$

	$[\![i]\!] : [\![exp]\!]$
O	q
P	i

	$[\![\Omega_B]\!] : [\![B]\!]$
O	m_q

	$[\![\mathbf{succ}]\!] : [\![exp]\!]$	$\Rightarrow [\![exp]\!]$
O		q
P	q	
O	i	
P		$i+1$

	$[\![\mathbf{pred}]\!] : [\![exp]\!]$	$\Rightarrow [\![exp]\!]$
O		q
P	q	
O	$i+1$	
P		i

$[\![\mathbf{ifzero}_B]\!]$:

	$[\![exp]\!]$	$\times [\![B]\!]$	$\times [\![B]\!]$	$\Rightarrow [\![B]\!]$
O				m_q
P	q			
O	0			
P		m_q		
O		m_a		
P				m_a

	$[\![exp]\!]$	$\times [\![B]\!]$	$\times [\![B]\!]$	$\Rightarrow [\![B]\!]$
O				m_q
P	q			
O	$i+1$			
P			m_q	
O			m_a	
P				m_a

	$[\![;_B]\!] : [\![com]\!]$	$\Rightarrow [\![B]\!]$	$\Rightarrow [\![B]\!]$
O			m_q
P	run		
O	$done$		
P		m_q	
O		m_a	
P			m_a

	$[\![!]\!] : [\![var]\!]$	$\Rightarrow [\![exp]\!]$
O		q
P	$read$	
O	i	
P		i

	$[\![:=]\!] : [\![var]\!]$	$\Rightarrow [\![exp]\!]$	$\Rightarrow [\![com]\!]$
O			run
P		q	
O		i	
P	$write(i)$		
O	ok		
P			$done$

Note that P does not reply to the initial question in $[\![\Omega_B]\!]$. Similarly, because we assumed that $\mathbf{succ}(max)$ and $\mathbf{pred}(0)$ are undefined, P will not respond to $q\,q\,max$ and $q\,q\,0$ when following $[\![\mathbf{succ}]\!]$ and $[\![\mathbf{pred}]\!]$ respectively.

Fig. 2. Strategies used to interpret IA_2 and their maximal positions.

- *PSPACE-complete for nondeterministic finite automata,*
- *PSPACE-complete for regular expressions,*
- *EXPSPACE-complete for regular expressions with intersection,*
- *EXPSPACE-complete for regular expressions with squaring ($L^2 = L \cdot L$).*

The PSPACE and EXPSPACE bounds for equivalence of regular expressions are proved by following standard automata constructions. For regular expressions they produce automata of linear size with respect to the size of the interpreted expression. However, intersection requires the use of a product automaton whose size is the product of sizes of the two component automata. Similarly, automata must be duplicated to interpret squaring. As both constructions need access to all states of the component automata, the components must be stored in their entirety for the sake of future constructions (this should be contrasted with the constructions for concatenation, Kleene star and sum, which can be conducted using just the initial and final states). Thus, nested occurrences of intersections or squaring will require exponential space. Indeed, that use of strictly superpolynomial space cannot be eliminated, as the equivalence problems are EXPSPACE-complete and it is known that PSPACE≠EXPSPACE [14].

1.3 Earlier work and outline of the new results

Now we are ready to estimate the complexity of algorithms obtained by following the recursive assignments of extended regular expressions to IA_2 terms presented in [9,10]. Both papers use intersections to enforce the causality between reads and writes to variables, which seems rather unavoidable. In addition, a number of auxiliary operations such as substitution, restriction and various homomorphic images are employed. If we want to account for IA_2 terms, squaring must also be handled because

$$\mathsf{comp}(\llbracket(\lambda x.x;x)M\rrbracket) = \mathsf{comp}(\llbracket M\rrbracket)\cdot\mathsf{comp}(\llbracket M\rrbracket)$$

(this is done as a special case of intersection in [10]). Thus, assuming that all the auxiliary operations do not make complexity worse, we can extract an exponential space algorithm provided the size of the extended regular expressions is linear in the size of the analyzed term. It turns out however that some care is still needed here, because even the natural descriptions in [9] yield expressions of exponential size. For instance, any of the two rules below (used iteratively) can produce this effect

$$\begin{aligned} (\!|\,\text{if } M \text{ then } N_0 \text{ else } N_1|\!)_i &= (\!|M|\!)_{\mathrm{tt}}\cdot(\!|N_0|\!) + (\!|M|\!)_{\mathrm{ff}}\cdot(\!|N_1|\!) \\ (\!|\mathbf{while}\ M\ \mathbf{do}\ N|\!) &= ((\!|M|\!)_{\mathrm{tt}}\cdot(\!|N|\!))^*\cdot(\!|M|\!)_{\mathrm{ff}} \end{aligned}$$

because $(\!|M|\!)$ occurs twice on the right. The translation from [10] does give rise to expressions whose size is linear in the size of the program, but the induced automata are often larger than one could expect. For example, because products are used to model any application, the size of the automaton representing **ifzero** MN_0N_1 or $fM_1 \cdots M_n$ is equal to the product of the sizes of the automata being combined, although intuitively it should be closer to their sum. In any case, an EXPSPACE algorithm is implicit in [10] and due to EXPSPACE-completeness of containment and equivalence for extended regular expressions one might suspect that program equivalence and approximation share this complexity.

This turns out *not* to be the case. We will prove that IA_2 program approximation as well as program equivalence are PSPACE-complete, which shows that regular expressions are not the ideal way to represent game semantics if intuitions about complexity are to be conveyed. The discrepancy seems to come from the fact that game semantics is deterministic whereas regular expressions can also account for nondeterminism.

Our results can be seen as a continuation of Jones and Muchnick's work on finite memory programs (FMPs) [15]. FMPs were considerably simpler than IA_2 programs. They lacked type structure, did not allow for function definitions and their relation to finite deterministic automata was more apparent. Moreover, the notion of equivalence considered in [15] was rather crude and, like for automata, based on equivalence of accepted inputs.

The approach we take consists of several steps. First, given a term P, we will find another term P' whose game semantics can be thought of as a symbolic representation of state changes caused by P. Roughly, P' will be obtained from P by ignoring the occurrences of **new** (thereby eliminating some problematic product constructions). In general this does not yield an equivalent program and Section 2 shows how to mend the defects so that a 'correct' P', without any occurrences of **new**, can be found.

In Section 3 we define a procedure called IA2DFA which produces a deterministic automaton for **new**-free IA_2 terms. The size of the automaton can still be exponential (because nested applications of a λ-abstraction can cause the squaring effect) but we will show how to carry out the computation on a PSPACE transducer.

Definition 5 A *transducer* is a Turing machine equipped with a read-only input tape, write-only output tape, and readable and writable work tape. A PSPACE transducer never uses more than $p(|s|)$ *work* space on any input s, for some polynomial p.

PSPACE transducers terminating on all inputs may still produce output of exponential size but that is the limit since each computation must end after

$$
\begin{aligned}
\textbf{succ}(\textbf{new } X \textbf{ in } M) &\cong \textbf{new } X \textbf{ in } \textbf{succ}(M) \\
\textbf{pred}(\textbf{new } X \textbf{ in } M) &\cong \textbf{new } X \textbf{ in } \textbf{pred}(M)
\end{aligned}
$$

$$
\begin{aligned}
\textbf{ifzero}\,(\textbf{new } X \textbf{ in } M)N_0N_1 &\cong \textbf{new } X \textbf{ in } (\textbf{ifzero}\, MN_0N_1) \\
\textbf{ifzero}\, M(\textbf{new } X \textbf{ in } N_0)N_1 &\cong \textbf{new } X \textbf{ in } (\textbf{ifzero}\, MN_0N_1) \\
\textbf{ifzero}\, MN_0(\textbf{new } X \textbf{ in } N_1) &\cong \textbf{new } X \textbf{ in } (\textbf{ifzero}\, MN_0N_1)
\end{aligned}
$$

$$
\begin{aligned}
(\textbf{new } X \textbf{ in } M); N &\cong \textbf{new } X \textbf{ in } (M; N) \\
M; (\textbf{new } X \textbf{ in } N) &\cong \textbf{new } X \textbf{ in } (M; N)
\end{aligned}
$$

$$
\begin{aligned}
M := (\textbf{new } X \textbf{ in } N) &\cong \textbf{new } X \textbf{ in } M := N \\
(\textbf{new } X \textbf{ in } M) := N &\cong \textbf{new } X \textbf{ in } M := N
\end{aligned}
$$

Fig. 3. Some equivalences

an exponential number of steps.

Section 5 describes how the automaton produced in the previous round can be refined by taking state changes into account. The outcome will be a deterministic automaton accepting precisely the complete plays induced by the analyzed term. Since we want the resultant algorithm to be implementable by a PSPACE tranducer as well, the integration of IA2DFA must be carried out with caution so as to avoid the storage of the full output tape.

Finally, for approximation or equivalence testing we need to submit the two PSPACE computable description of automata to the containment or equivalence checking algorithms. As we recalled in Theorem 4 this check can be implemented in nondeterministic logarithmic space, but since the input is actually of exponential size with respect to the size of the initial program 'logarithmic' means 'polynomial'. As before, the problem of storing the intermediate result (which may be of exponential size) must be addressed but once this is done we get a nondeterministic PSPACE verification procedure. Since NPSPACE=PSPACE (see e.g. [14]) the approximation and equivalence problems for IA_2 terms are in PSPACE.

In Section 6 we show that they are also PSPACE-hard and hence PSPACE-complete. Finally, we discuss the complexity of equivalence for a number of fragments of IA_2 and conclude with some optimizing suggestions.

2 Moving variable bindings

This section begins the description of a PSPACE algorithm which, given an IA_2 typing judgment $x_1 : T_1, \cdots, x_n : T_n \;\vdash\; P : T$, produces an automaton accepting $\mathsf{comp}(\llbracket x_1 : T_1, \cdots, x_n : T_n \ \vdash \ P : T \rrbracket)$. From now on we will always use P to refer to the original program.

In order to simplify the computation of the automaton we will first move all **new**-bindings to the topmost level (consequently losing all information about local scope). This must be done in such a way that program equivalence is preserved. In particular the following two problems must be addressed.

Firstly, moving bindings outwards is not always a well-defined operation on the syntax (which only allows terms of the shape **new** X **in** M if M is of base type). Therefore, we need to define what **new** X **in** M means when M is of function type: given $M : T' \to T''$, **new** X **in** M will be shorthand for $\lambda x^{T'}.$**new** X **in** Mx. It worth noting that, regardless of T, $\llbracket \Gamma \ \vdash$ **new** X **in** $M : T\rrbracket$ is actually calculated in the same way as for base types, by cutting down $\llbracket \Gamma, X : \mathit{var} \ \vdash \ M : T \rrbracket$ to sequences with the 'good variable' behaviour in which the $\mathit{write}(i)$, ok, read, i moves are hidden.

In many cases the expansion of scope produces equivalent terms as shown in Figure 3. In fact, the terms displayed on the left-hand side in the figure are interpreted by the same strategies as those on the left. Unfortunately some desirable equivalences fail:

$$\begin{aligned}
\textbf{while } (\textbf{new } X \textbf{ in } M) \textbf{ do } N &\ncong \textbf{new } X \textbf{ in while } M \textbf{ do } N\\
\textbf{while } M \textbf{ do } (\textbf{new } X \textbf{ in } N) &\ncong \textbf{new } X \textbf{ in while } M \textbf{ do } N\\
M(\textbf{new } X \textbf{ in } N) &\ncong \textbf{new } X \textbf{ in } MN,
\end{aligned}$$

because the expression in scope of the variable X on the left might be evaluated several times. Then the terms on the right behave differently, because the second evaluation would inherit the state from the first one (in the third case this is due to call-by-name evaluation).

Example 6 Here is a concrete example illustrating the difference:

$$\begin{aligned}
&(\lambda x.\textbf{ifzero}\, xxx)(\textbf{new } X \textbf{ in } (X := \textbf{ifzero}\, !X10); !X) \cong 1,\\
&\textbf{new } X \textbf{ in } (\lambda x.\textbf{ifzero}\, xxx)((X := \textbf{ifzero}\, !X10); !X) \cong 0.
\end{aligned}$$

We will address the failures by explicit initialization and replace each subterm of P of the form **new** X **in** M with **new** X **in** $(X := 0; M)$. Obviously the addition of the (redundant) explicit initializations yields an equivalent program. This syntactic operation should be carried out as a preprocessing pass and combined with renaming identifiers in order to avoid name clashes when the bindings are removed. The former might double the size of the program in the worst case, the latter may add a logarithmic factor, but in any case the new term can be stored in polynomial space (with respect to the original size of P). After explicit initialization the removal of **new** to the outermost level turns out to preserve equivalence.

Definition 7 A term-in-context $\Gamma, X : var \vdash M : T$ is *explicitly initialized* with respect to X iff in each position of $[\![\Gamma, X : var \vdash M : T]\!]$ the first move made in the designated $[\![var]\!]$ subgame is $write(0)$.

Lemma 8 *Suppose* $\Gamma, X : var \vdash M : T$ *is explicitly initialized with respect to* X *and* $\Gamma, [\cdot] : T \vdash C[\cdot] : T'$ *is an* IA_2 *context (in particular this means that* X *does not occur in* $C[\cdot]$*). Then:*

(i) $C[M]$ *is explicitly initialized with respect to* X*,*
(ii) $[\![C[\mathbf{new}\ X\ \mathbf{in}\ M]]\!] = [\![\mathbf{new}\ X\ \mathbf{in}\ C[M]]\!]$.

PROOF. *(i)* holds because of the way strategies are composed. *(ii)* can be proved by induction on the structure of C using *(i)*. For the cases shown in Figure 3 and

$$\begin{aligned}
[\![\lambda x.(\mathbf{new}\ X\ \mathbf{in}\ C[M])]\!] &= [\![\mathbf{new}\ X\ \mathbf{in}\ \lambda x.C[M]]\!] \\
[\![(\mathbf{new}\ X\ \mathbf{in}\ C[M])N]\!] &= [\![\mathbf{new}\ X\ \mathbf{in}\ (C[M]N)]\!] \\
[\![\mathbf{mkvar}(M, \mathbf{new}\ X\ \mathbf{in}\ C[N])]\!] &= [\![\mathbf{new}\ X\ \mathbf{in}\ \mathbf{mkvar}(M, C[N])]\!] \\
[\![\mathbf{mkvar}(\mathbf{new}\ X\ \mathbf{in}\ C[M], N)]\!] &= [\![\mathbf{new}\ X\ \mathbf{in}\ \mathbf{mkvar}(C[M], N)]\!]
\end{aligned}$$

the assumption that M (and consequently $C[M]$) is explicitly initialized is irrelevant. However, it is essential to turning the inequivalences identified on the previous page into equivalences. $\Box$

Example 9 We revisit Example 6 after adding explicit initialization:

$$\begin{aligned}
(\lambda x.\mathbf{ifzero}\, xxx)(\mathbf{new}\ X\ \mathbf{in}\ (X := 0; X := \mathbf{ifzero}\, !X 1 0; !X)) &\cong 1, \\
\mathbf{new}\ X\ \mathbf{in}\ (\lambda x.\mathbf{ifzero}\, xxx)(X := 0; X := \mathbf{ifzero}\, !X 1 0; !X) &\cong 1.
\end{aligned}$$

By the above Lemma, since $\Gamma, X : var \vdash (X := 0; M) : B$ is explicitly initialized we have $[\![C[\mathbf{new}\ X\ \mathbf{in}\ (X := 0; M)]]\!] = [\![\mathbf{new}\ X\ \mathbf{in}\ C[X := 0; M]]\!]$. If we apply this fact for each occurrence of **new** in P we arrive at

Corollary 10 *For any* IA_2 *term* $\Gamma \vdash P : T$ *there exists a* **new***-free* IA_2 *term* $\Gamma, X_1 : var, \cdots, X_m : var \vdash P' : T$ *such that*

$$[\![\Gamma \vdash P]\!] = [\![\Gamma \vdash \mathbf{new}\ X_1, \cdots, X_m\ \mathbf{in}\ P']\!],$$

$m \leq |P|$ *and* $|P'| = O(\,|P| \log |P|\,)$.

The Corollary amounts to a simple proof of the factorization theorem for IA_2 without the need to encode positions as in the general proof [3]. We are going use it to simplify the generation of an automaton accepting $\mathsf{comp}(\llbracket\Gamma \vdash P : T\rrbracket)$: we will construct an automaton for P' and convert it to one for P. In the language of regular expressions, this corresponds to moving the intersections corresponding to **new** to the outermost level, which greatly simplifies their translation. The automaton for P' may still have exponential size though.

Remark 11 The globalization of variables conducted for IA_2 programs cannot be extended to full IA. Let M_1, M_2 be the terms $\lambda f^{com\to com}.f(f\, skip)$ and respectively

$$X := 0;\; x\,; X := \mathbf{succ}(!X); \mathbf{ifzero}\,(\mathbf{pred}(!X))\,(skip)\,\Omega_{com}.$$

Then $M_1(\lambda x^{com}.\mathbf{new}\ X\ \mathbf{in}\ M_2) \cong skip$ but $\mathbf{new}\ X\ \mathbf{in}\ M_1(\lambda x^{com}.M_2) \cong \Omega_{com}$.

3 Algorithm for new-free programs

From now on, for brevity, we shall write $\llbracket\Gamma \vdash M : T\rrbracket$ meaning $\mathsf{comp}(\llbracket\Gamma \vdash M : T\rrbracket)$. Assuming $T = B_1 \to \ldots \to B_k \to B$, $\llbracket\Gamma \vdash M : T\rrbracket$ can be decomposed in one of the following ways depending on B and the initial and final moves:

$$\begin{aligned}
B = com:&\quad \llbracket\ldots\rrbracket = run \cdot (\!|\ldots|\!) \cdot done,\\
B = exp:&\quad \llbracket\ldots\rrbracket = q \cdot \textstyle\sum_{i=0}^{max}((\!|\ldots|\!)_i \cdot i),\\
B = var:&\quad \llbracket\ldots\rrbracket = read \cdot \textstyle\sum_{i=0}^{max}((\!|\ldots|\!)_i^r \cdot i) + \sum_{i=0}^{max}(write(i) \cdot (\!|\ldots|\!)_i^w) \cdot ok.
\end{aligned}$$

We make a few auxiliary definitions: for $B = exp$ we define $(\!|\ldots|\!) = \sum_{i=0}^{max}(\!|\ldots|\!)_i$, for $B = var$ we let $(\!|\ldots|\!)^r = \sum_{i=0}^{max}(\!|\ldots|\!)_i^r$. The generated automata will represent $(\!|\ldots|\!)$, $(\!|\ldots|\!)^r$, $(\!|\ldots|\!)_i^w$ respectively in a way to be specified soon. The alphabet $\mathcal{A}$ will consist of moves defined by the types occurring in the typing judgment. We use identifier names to 'implement' the disjoint sums inherent in the construction of $(\prod_{i=1}^n \llbracket T_i\rrbracket) \Rightarrow \llbracket T\rrbracket$: the names will be attached to moves of the component base type games and in addition, for function types, we will add numerical indices to moves originating from the types of arguments. Suppose $T_i = B_{i,1} \to \ldots \to B_{i,k_i} \to B_i'$ $(i = 1, \cdots, n)$ and $T = B_1 \to \ldots \to B_k \to B$. Then we set

$$\begin{aligned}
\mathcal{A} = \textstyle\bigcup_{i=1}^n(&\quad \textstyle\bigcup_{j=1}^{k_i}\{\, c_{x_{i,j}} \mid c \in \mathcal{A}(B_{i,j})\,\} \quad \cup \quad \{\, c_{x_i} \mid c \in \mathcal{A}(B_i')\,\} \quad) \cup\\
&\textstyle\bigcup_{i=1}^k\{\, c_i \mid c \in \mathcal{A}(B_i)\,\} \,\cup\, \mathcal{A}(B)
\end{aligned}$$

where

$$\begin{array}{ll}\mathcal{A}(\mathit{exp}) & = \{\, q, 0, \cdots, \mathit{max} \,\}, \\ \mathcal{A}(\mathit{com}) & = \{\, \mathit{run}, \mathit{done} \,\}, \\ \mathcal{A}(\mathit{var}) & = \{\, \mathit{read}, 0, \cdots, \mathit{max}, \mathit{write}(0), \cdots, \mathit{write}(\mathit{max}), \mathit{ok} \,\}.\end{array}$$

For any IA_2 term our algorithm will generate a *semantic automaton*, which is essentially a partial deterministic automaton with ϵ-transitions.

Definition 12 $A = \langle\, Q, s, \delta, L \,\rangle$ is a *semantic automaton* providing

- $Q \subseteq \mathbb{N}$, $s \in Q$, L is a list of states from Q (called the *final list*), and
- Q, δ can be decomposed as $Q = Q_{\mathcal{A}} + Q_\epsilon$ and $\delta = \delta_{\mathcal{A}} + \delta_\epsilon$ respectively such that $\delta_{\mathcal{A}} : Q_{\mathcal{A}} \times \mathcal{A} \rightharpoonup Q$ and $\delta_\epsilon : Q_\epsilon \times \{\, \epsilon \,\} \rightharpoonup Q$.

Note that whenever there is an ϵ-transition, it is unique and no other transitions involving characters from the alphabet are possible.

Definition 13 Let $A = \langle\, Q, s, \delta, L \,\rangle$ be a semantic automaton and suppose $T = B_1 \to \ldots \to B_k \to B$.

- If $B = \mathit{com}$ we say that A accepts $(\!|\Gamma \vdash M : T|\!)$ if $L = [s']$ and $\langle\, Q, s, \delta, \{\, s' \,\} \,\rangle$ accepts $(\!|\Gamma \vdash M : T|\!)$ in the standard sense.
- If $B = \mathit{exp}$ we say that A accepts $(\!|\Gamma \vdash M : T|\!)$ if $L = [s_0, \cdots, s_{max}]$ and for any $0 \leq i \leq \mathit{max}$ the automaton $\langle\, Q, s, \delta, \{\, s_i \,\} \,\rangle$ accepts $(\!|\Gamma \vdash M : T|\!)_i$.
- If $B = \mathit{var}$ we define the acceptance of $(\!|\Gamma \vdash M : T|\!)^r$ like for exp and that of each $(\!|\Gamma \vdash M : T|\!)^w_i$ $(0 \leq i \leq \mathit{max})$ like for com.

Given semantic automata for $(\!|\ldots|\!)$ it is very easy to construct those accepting $[\![\ldots]\!]$ by following the decomposition patterns.

Semantic automata will be generated by scanning the input program, in the opposite order to that normally used for evaluation. This leads to quite a concise procedure, shown in Figure 5, which does not generate any unnecessary ϵ-transitions for stitching the automata resulting from recursive calls. The automata will be generated back-to-front: we specify the list of final states first, then pass it as an argument to the generating procedure IA2DFA and wait for the initial state to be returned (recall that states are natural numbers). The alternative approach to output the final states given the initial state is more problematic: in order to interpret **ifzero** MN_0N_1 we would have to 'unify' the final states resulting from N_0 and N_1 either by adding ϵ-transitions and effectively merging the states, or by maintaining sets of final states (which might grow exponentially large).

IA2DFA takes two arguments, an IA_2 term and a list of states (meant to be the final list), and returns the initial state of the semantic automaton corresponding to the analyzed term. For $\Gamma \vdash P' : T$ such that $T = B_1 \to \ldots \to B_k \to B$, the initial call will depend on B:

- for $B = com$ we call IA2DFA$(P', [0])$,
- for $B = exp$ we call IA2DFA$(P', [0, \cdots, max])$,
- for $B = var$ we can call either IA2DFA$(P', [0, \cdots, max])^r$ (to get $(\!|\ldots|\!)^r$) or IA2DFA$(P', [0])_i^w$ (to get $(\!|\ldots|\!)_i^w$) for any $0 \le i \le max$.

Transitions of the automaton will be output at runtime by PRINT instructions as $s_1 \xrightarrow{c} s_2$, where $s_1, s_2 \in \mathbb{N}$ and $c \in \mathcal{A} \cup \{\,\epsilon\,\}$. To simplify proofs we assume that P' is of base type. This is an insignificant restriction: instead of P' we can always consider $\Gamma, y_1 : B_1, \cdots, y_k : B_k \vdash P' y_1 \cdots y_k : B$ instead. The game semantics of $P' y_1 \cdots y_k$ and P' are almost identical (the η-law is valid) except that the moves labelled with y_i for $P' y_1 \cdots y_h$ should be labelled with i for P'. This distinction can be easily integrated into our procedure later and does not affect complexity since the typing judgments submitted for analysis contain type information about free and bound variables anyway.

IA2DFA relies on certain information about function arguments in P', which should be extracted before IA2DFA is called. In IA_2, functions can be defined either as λ-abstractions or as first-order identifiers. Therefore, each argument to a function can be associated either with an occurrence of λ or with an occurrence of a first-order variable f and an index $1 \le i \le ar(f)$. We will differentiate between occurrences of the same first-order variable f by annotating them with subscripts $(f_1, f_2, f_3, \cdots)$. Similarly, we assume that no two bound variables have the same name. Thus, for a given term, each function argument can be specified either by the name of a base-type identifier or by a pair (f_z, j), where $1 \le j \le ar(f)$ and f_z is an occurrence of f in P'. The function *arg* will assign the actual argument to each such specification if possible (some functions may not be applied inside the term, e.g. M in $\mathbf{mkvar}(M, N)$). We can define *arg* by running the procedure ARGS shown in Figure 4 ([ ] is the empty list, : denotes concatenation). In all other cases not mentioned in the Figure the call to ARGS should be propagated so that all subterms are examined. Values of *arg* are defined only inside the rule for application.

Example 14 Suppose $f : com \to com \to com \vdash M : com$ where M is of the shape $\lambda x.\lambda y.f_1((\lambda z.f_2 M_1 M_2) M_3)$ for some M_1, M_2, M_3. Then we have ARGS$(M) = [x, y, (f_1, 2)]$ and

$$\begin{array}{ll} arg(f_1, 1) = (\lambda z.f_2 M_1 M_2) M_3 & arg(z) = M_3 \\ arg(f_2, 1) = M_1 & arg(f_2, 2) = M_2. \end{array}$$

Remark 15 It is worth observing that the value of $arg(x)$ is a subterm which occurs to the right of any occurrence of x in P'. This ensures that recursive definitions using $arg(x)$ will not be circular.

$$\text{ARGS}(i) = [\,] \qquad \text{ARGS}(\mathit{skip}) = [\,] \qquad \text{ARGS}(\Omega_B) = [\,]$$

$$\begin{aligned}
\text{ARGS}(f_z) &= [(f_z, 1), \cdots, (f_z, ar(f))] \\
\text{ARGS}(\lambda x.M) &= (x : \text{ARGS}(M)) \\
\text{ARGS}(MN) &= (h : t) = \text{ARGS}(M); \\
&\quad\ \text{DEFINE}(\mathit{arg}(h) = N); \\
&\quad\ [\,] = \text{ARGS}(N); \\
&\quad\ \text{RETURN } t \\
\text{ARGS}(\mathbf{mkvar}(M, N)) &= [h] = \text{ARGS}(M) \quad h \text{ is the write parameter} \\
&\quad\ \text{ARGS}(N) \\
&\quad\ \text{RETURN } [\,]
\end{aligned}$$

Fig. 4. Computing *arg*.

Given a term of arity k ARGS returns a list of length k (corresponding to the k arguments). The lemma below makes this precise. Consequently, ARGS(N) in the rule for application always returns the empty list and ARGS(M), for **mkvar**(M, N), returns a singleton list $[h]$. We call h the associated *write parameter* of the occurrence of **mkvar**. The set of write parameters occurring in P' will be referred to as WPAR(P').

Lemma 16 *Let* $\Gamma \vdash M : T$ *such that* $ar(M) = m$. *Then* $ARGS(M) = [y_1, \cdots, y_m]$ *and for some* $0 \leq l \leq m$ *we have*

- y_i *is a base-type identifier for all* $1 \leq i \leq l$,
- *there exists* $z \in \mathbb{N}$ *such that* $y_i = (f_z, ar(f) + i - m)$ *for all* $l < i \leq m$.

Because we assumed that P' is of base type, ARGS(P') = [ ].

Lemma 17 *Let* M *be a subterm of* P'.

- *If* $y \in ARGS(M)$ *then either* $arg(y)$ *is defined after* $ARGS(P')$ *is complete or* $y \in WPAR(P')$ *(the first* $\in$ *means list membership).*
- *If* $x \in FV(M)$ *then either* $arg(x)$ *is defined after* $ARGS(P')$ *is complete, or* $x \in WPAR(P')$, *or* $x \in FV(P')$.

The statically gathered information suffices to generate automata for all elements of the syntax except **mkvar**. Each occurrence of **mkvar** comes with an associated write parameter, which cannot be defined statically. Instead, its *arg* value will be determined at runtime as necessary.

Note that ARGS runs in polynomial time and the generated *arg* function can be stored in polynomial space for future reference. *arg* contains information about function arguments and will be used in IA2DFA to transfer control to them once they have to be processed. In this respect IA2DFA operates very much like a call-by-name evaluator. Thanks to the ability to make 'jumps' to arguments, IA2DFA will not have to use exponential space, even though the generated automaton might be of exponential size.

The definition of IA2DFA for constants and composite terms is presented in Figure 5. Note that hardly any transitions get printed out since moves correspond to free identifiers. IA2DFA for identifiers is defined in the next two figures (6 and 7) respectively for base and first-order types. The clause for $\mathbf{mkvar}(M, N)$ will ensure that $\arg y$ for $y \in \mathrm{WPAR}(P')$ will always be defined before it is needed. States of semantic automata are natural numbers. We use $\mathit{fresh}(s)$ to generate a yet unused natural number. This can be implemented via a global natural number which is incremented during each call to *fresh*. $\mathit{fresh}(s_1, \cdots, s_n)$ will be shorthand for $\mathit{fresh}(s_1), \cdots, \mathit{fresh}(s_n)$. We use ∞ to denote a special state from which no transitions will be possible. The definitions of $\mathrm{IA2DFA}(\cdots, l)^r$, $\mathrm{IA2DFA}(\cdots, l)_i^w$ for $\mathbf{ifzero}\, M N_0 N_1$, $M; N$, MN, $\lambda x.M$ (although not shown explicitly in the figure) are identical to those presented there for $\mathrm{IA2DFA}(\cdots, l)$.

4 Analysis of the algorithm

IA2DFA never diverges because each recursive branch it generates could be viewed as a left-to-right scan of P': at each call a subterm of the currently analyzed term is visited or a jump is made following *arg*. By Remark 15 the jump is always to the right and visiting subterms also corresponds to proceeding right in P'. Thus the depth of the recursive stack is bounded by $|P'|$ and we can reason by induction on the depth. As *arg* is not always defined for write parameters it is important to show that all *arg* values are defined when they are needed. After settling this, we will prove that IA2DFA generates a semantic automaton and, finally, that the automaton represents the game semantics of P'. In what follows we shall often make statements about $\mathrm{IA2DFA}(M, l)$ meaning all the various types of call like $\mathrm{IA2DFA}(M, l)^r$ and $\mathrm{IA2DFA}(M, l)_i^w$.

Definition 18 Let M be a subterm of P' and $\mathrm{FV}(M)\backslash\mathrm{FV}(P') = \{\, v_1, \cdots, v_k \,\}$. Suppose that for any write parameter h if $\mathit{arg}(h)$ is defined then $\mathit{arg}(h) \in \{\, 0, \cdots, \mathit{max} \,\}$. Then we define $\overline{M}$ to be $M[\overline{\mathit{arg}(v_1)}/v_1, \cdots, \overline{\mathit{arg}(v_k)}/v_k]$.

Note that the definition depends on values of *arg* for write parameters and $\overline{M}$ might change when *arg* is modified. By Remark 15 and the fact that $\mathit{arg}(h) \in \{\, 0, \cdots, \mathit{max} \,\}$ for write parameters, the definition is not circular. We consider

$$
\begin{array}{ll}
\mathrm{IA2DFA}(\mathit{skip}, [s]) = & \mathrm{RETURN}\ s \\
\mathrm{IA2DFA}(i, [s_0, \cdots, s_{max}]) = & \mathrm{RETURN}\ s_i \\
\mathrm{IA2DFA}(\Omega_B, l) = & \mathrm{RETURN}\ \infty \\
& \\
\mathrm{IA2DFA}(\mathbf{succ}(M), [s_0, \cdots, s_{max}]) = & \mathrm{IA2DFA}(M, [s_1, \cdots, s_{max}, \infty]) \\
\mathrm{IA2DFA}(\mathbf{pred}(M), [s_0, \cdots, s_{max}]) = & \mathrm{IA2DFA}(M, [\infty, s_0, \cdots, s_{max-1}]) \\
& \\
\mathrm{IA2DFA}(\mathbf{ifzero}\, M N_0 N_1, l) = & s_0 = \mathrm{IA2DFA}(N_0, l) \\
& s_1 = \mathrm{IA2DFA}(N_1, l) \\
& \mathrm{IA2DFA}(M, [s_0, \underbrace{s_1, \cdots, s_1}_{max}]) \\
& \\
\mathrm{IA2DFA}(\mathbf{while}\ M\ \mathbf{do}\ N, [s]) = & \mathit{fresh}(s_N) \\
& s_1 = \mathrm{IA2DFA}(N, [s_N]) \\
& s_M = \mathrm{IA2DFA}(M, [s, \underbrace{s_1, \cdots, s_1}_{max}]) \\
& \mathrm{PRINT}(s_N \xrightarrow{\epsilon} s_M) \\
& \mathrm{RETURN}\ s_M \\
& \\
\mathrm{IA2DFA}(M; N, l) = & s = \mathrm{IA2DFA}(N, l) \\
& \mathrm{IA2DFA}(M, [s]) \\
& \\
\mathrm{IA2DFA}(M := N, [s]) = & \mathbf{for}\ i = 0\ \mathbf{to}\ max\ \mathbf{do} \\
& \qquad s_i = \mathrm{IA2DFA}(M, [s])^w_i \\
& \mathrm{IA2DFA}(N, [s_0, \cdots, s_{max}]) \\
& \\
\mathrm{IA2DFA}(!M, [s_0, \cdots, s_{max}]) = & \mathrm{IA2DFA}(M, [s_0, \cdots, s_{max}])^r \\
& \\
\mathrm{IA2DFA}(MN, l) = & \mathrm{IA2DFA}(M, l) \\
\mathrm{IA2DFA}(\lambda x.M, l) = & \mathrm{IA2DFA}(M, l) \\
& \\
\mathrm{IA2DFA}(\mathbf{mkvar}(M, N), [s_0, \cdots, s_{max}])^r = & \\
& \mathrm{IA2DFA}(N, [s_0, \cdots, s_{max}]) \\
& \\
\mathrm{IA2DFA}(\mathbf{mkvar}(M, N), [s])^w_i = & \text{let } h \text{ be the associated} \\
& \text{write parameter} \\
& \\
& \mathrm{DEFINE}(\mathit{arg}(h) = i) \\
& s' = \mathrm{IA2DFA}(M, [s]) \\
& \mathrm{UNDEFINE}\ (\mathit{arg}(h)) \\
& \mathrm{RETURN}\ s'
\end{array}
$$

Fig. 5. IA2DFA

$\underline{arg(x) \text{ defined}}$

$$\begin{array}{ll}
\text{IA2DFA}(x : com, [s]) = & \text{IA2DFA}(arg(x), [s]) \\
\text{IA2DFA}(x : exp, [s_0, \cdots, s_{max}]) = & \text{IA2DFA}(arg(x), [s_0, \cdots, s_{max}]) \\
\text{IA2DFA}(x : var, [s_0, \cdots, s_{max}])^r = & \text{IA2DFA}(arg(x), [s_0, \cdots, s_{max}])^r \\
\text{IA2DFA}(x : var, [s])^w_i = & \text{IA2DFA}(arg(x), [s])^w_i
\end{array}$$

$\underline{arg(x) \text{ undefined}}$ (then $x \in \text{FV}(P')$ by Lemmas 17, 21)

$$\begin{array}{ll}
\text{IA2DFA}(x : com, [s]) = & fresh(s', s'') \\
 & \text{PRINT}(s' \xrightarrow{run_x} s'',\ s'' \xrightarrow{done_x} s) \\
 & \text{RETURN } s' \\
 & \\
\text{IA2DFA}(x : exp, [s_0, \cdots, s_{max}]) = & fresh(s', s'') \\
 & \text{PRINT}(s' \xrightarrow{run_x} s'') \\
 & \textbf{for } i = 0 \textbf{ to } max \textbf{ do} \\
 & \qquad\qquad \text{PRINT}(s'' \xrightarrow{i_x} s_i) \\
 & \text{RETURN } s' \\
 & \\
\text{IA2DFA}(x : var, [s_0, \cdots, s_{max}])^r = & fresh(s', s'') \\
 & \text{PRINT}(s' \xrightarrow{read_x} s'') \\
 & \textbf{for } i = 0 \textbf{ to } max \textbf{ do} \\
 & \qquad\qquad \text{PRINT}(s'' \xrightarrow{i_x} s_i) \\
 & \text{RETURN } s' \\
 & \\
\text{IA2DFA}(x : var, [s])^w_i = & fresh(s', s'') \\
 & \text{PRINT}(s' \xrightarrow{write(i)_x} s'',\ s'' \xrightarrow{ok_x} s)
\end{array}$$

Fig. 6. The interpretation of base-type identifiers.

$\overline{M}$ to be undefined if some $arg(v_i)$ is not defined.

Lemma 19 *Let M be a subterm of P'. Then $FV(\overline{M}) \subseteq FV(P')$.*

Definition 20 Let M be a subterm of P'. Recall that $|\text{ARGS}(M)| = ar(M)$. Define $\widehat{M}$ as $\overline{M}\ \overline{arg(\text{ARGS}(M)[1])}\ \cdots\ \overline{arg(\text{ARGS}(M)[ar(M)])}$.

Note that $\widehat{M}$ is also dependent on values of arg for write parameters.

Lemma 21 *After the initial call IA2DFA(P', l), whenever IA2DFA(M, l) is called, $\widehat{M}$ is defined. Moreover, when IA2DFA(M, l) returns, arg is the same as at the moment IA2DFA(M, l) was called.*

$$
\begin{array}{lll}
B = com: \mathrm{IA2DFA}(f_z,[s]) = & & \mathit{fresh}(s',s'') \\
& & \mathrm{PRINT}(s' \xrightarrow{run_f} s'',\ s'' \xrightarrow{done_f} s) \\
& & \text{goto REST} \\
B = exp: & \mathrm{IA2DFA}(f_z,[s_0,\cdots,s_{max}]) = & \mathit{fresh}(s',s'') \\
& & \mathrm{PRINT}(s' \xrightarrow{q_f} s'') \\
& & \textbf{for } k = 0 \textbf{ to } n \textbf{ do} \\
& & \qquad \mathrm{PRINT}(s'' \xrightarrow{k_f} s_k) \\
& & \text{goto REST} \\
B = var: & \mathrm{IA2DFA}(f_z,[s_0,\cdots,s_{max}])^r = & \mathit{fresh}(s',s'') \\
& & \mathrm{PRINT}(s' \xrightarrow{read_f} s'') \\
& & \textbf{for } k = 0 \textbf{ to } n \textbf{ do} \\
& & \qquad \mathrm{PRINT}(s'' \xrightarrow{k_f} s_k) \\
& & \text{goto REST} \\
& \mathrm{IA2DFA}(f_z,[s])^w_i = & \mathit{fresh}(s',s'') \\
& & \mathrm{PRINT}(s' \xrightarrow{write(i)_f} s'',\ s'' \xrightarrow{ok_f} s) \\
& & \text{goto REST}
\end{array}
$$

REST :

for $j = 1$ **to** $ar(f)$ **do**

$$
\begin{array}{ll}
B_j = com: & \mathit{fresh}(u^j) \\
& s^j = \mathrm{IA2DFA}(arg(f_z,j),[u^j]) \\
& \mathrm{PRINT}(u^j \xrightarrow{done_{f,j}} s'', s'' \xrightarrow{run_{f,j}} s^j) \\
B_j = exp: & \textbf{for } k = 0 \textbf{ to } max \textbf{ do } \mathit{fresh}(u^j_k) \\
& s^j = \mathrm{IA2DFA}(arg(f_z,j),[u^j_0,\cdots,u^j_{max}]) \\
& \textbf{for } k = 0 \textbf{ to } max \textbf{ do } \ \mathrm{PRINT}(u^j_k \xrightarrow{k_{f,j}} s'') \\
& \mathrm{PRINT}(s'' \xrightarrow{q_{f,j}} s^j) \\
B_j = var: & \textbf{for } k = 0 \textbf{ to } max \textbf{ do } \mathit{fresh}(u^j_k) \\
& s^j = \mathrm{IA2DFA}(arg(f_z,j),[u^j_0,\cdots,u^j_{max}])^r \\
& \textbf{for } k = 0 \textbf{ to } max \textbf{ do } \ \mathrm{PRINT}(u^j_k \xrightarrow{k_{f,j}} s'') \\
& \mathrm{PRINT}(s'' \xrightarrow{read_{f,j}} s^j) \\
& \mathit{fresh}(u^j) \\
& \textbf{for } k = 0 \textbf{ to } max \textbf{ do } s^j_k = \mathrm{IA2DFA}(arg(f_z,j),[u^j])^w_k \\
& \textbf{for } k = 0 \textbf{ to } max \textbf{ do } \mathrm{PRINT}(s'' \xrightarrow{write(k)_{f,j}} s^j_k) \\
& \mathrm{PRINT}(u^j \xrightarrow{ok_{f,j}} s'')
\end{array}
$$

RETURN s'

Fig. 7. The interpretation of first-order identifiers $f : B_1 \to \ldots \to B_{ar(f)} \to B$.

PROOF. We start from the second statement. Note that only a call to IA2DFA($\mathbf{mkvar}(M, N), [s])_i^w$ can modify *arg*. Because of our initial remark in this section about the algorithm working like a left-to-right scan, only one call for the same occurrence of **mkvar** can be active at the same time. Hence, when the value of $arg(h)$ is undefined at the end of IA2DFA($\mathbf{mkvar}(M, N), [s])_i^w$, the uniquely determined previous definition is reversed. This ensures that executing IA2DFA($\mathbf{mkvar}(M, N), l$) leaves *arg* unchanged.

For the first part we use induction on the order determined by the tree of recursive calls to IA2DFA following IA2DFA(P', l), where the root corresponds to the base case. For the initial call we have $\widehat{P'} = P'$. For the inductive step, we assume that when IA2DFA(M, l) is called $\widehat{M}$ is defined and we shall prove (by case analysis of M) that the immediate recursive calls made from IA2DFA(M, l) also have this property.

For (occurrences of) base-type identifiers $x : B$, $\widehat{x}$ is defined by induction hypothesis. Thus, either $x \in \mathrm{FV}(P')$ and $\widehat{x} = x$, or $x \notin \mathrm{FV}(P')$ and $\overline{arg(x)}$ is defined. In the first case there is nothing to prove because no recursive calls are made, in the second case there is a call for $arg(x)$, but then we have $\widehat{arg(x)} = \widehat{x}$.

For first-order identifiers $f : B_1 \to \ldots \to B_{ar(f)} \to B$ we know by induction hypothesis that $\widehat{f} = f\,\overline{arg(f, 1)} \cdots \overline{arg(f, ar(f))}$ is defined. Therefore, so is $\widehat{arg(f, i)} = \overline{arg(f, i)}$ for $1 \le i \le ar(f)$ (note that the calls for $arg(f, i)$ do not affect *arg* so we can still appeal to the induction hypothesis).

For **ifzero**, $\widehat{\mathbf{ifzero}\, M N_0 N_1}$ is defined (at call time). Because $\widehat{\mathbf{ifzero}\, M N_0 N_1} = \mathbf{ifzero}\, \widehat{M}\widehat{N_0}\widehat{N_1}$, each of $\widehat{M}$,$\widehat{N_0}$,$\widehat{N_1}$ is also defined then. Since the inner calls to IA2DFA do not change *arg* all these values are also defined when IA2DFA is called on each of them. The argument for **succ**, **pred**, **ifzero**, **while**, $M; N$, $M := N$, $!M$ and IA2DFA($\mathbf{mkvar}(M, N), l)^r$ is analogous.

For application it suffices to observe that $\widehat{MN} = \widehat{M}$ and appeal to the inductive hypothesis. For λ-abstraction note that $ar(\lambda x.M) = ar(M) + 1$ and $\mathrm{ARGS}(\lambda x.M)[1] = x$. By inductive hypothesis $\widehat{\lambda x.M}$ is defined, so $\overline{M_i}$ is defined for any $1 \le i \le ar(\lambda x.M)$, where M_i denotes $arg(\mathrm{ARGS}(\lambda x.M)[i])$. Besides, we have $\mathrm{ARGS}(M)[j] = M_{j+1}$ for $1 \le j \le ar(M)$, so

$$\begin{aligned} \widehat{\lambda x.M} &= \overline{(\lambda x.M)}\,\overline{M_1}\,\overline{M_2} \cdots \overline{M_{ar(\lambda x.M)}}, \\ \widehat{M} &= \overline{M}\,\overline{M_2} \cdots \overline{M_{ar(\lambda x.M)}} \qquad \text{(if defined)}. \end{aligned}$$

Therefore, to show that $\widehat{M}$ is defined, we need to demonstrate that $\overline{M}$ is defined. Let $\{ w_1, \cdots, w_l \} = \mathrm{FV}(M) \setminus \{ x \}$. Then

$$\begin{aligned} \overline{\lambda x.M} &= \lambda x.M[\overline{arg(w_1)}/w_1, \cdots, \overline{arg(w_l)}/w_l], \\ \overline{M} &= M[\overline{arg(w_1)}/w_1, \cdots, \overline{arg(w_l)}/w_l][\overline{arg(x)}/x] \qquad \text{(if defined).} \end{aligned}$$

Now note that $\overline{\lambda x.M}$ is defined (so $\overline{arg(w_j)}$ is defined for $1 \leq j \leq l$) and so is $\overline{arg(x)}$ (because $arg(x) = M_1$ and $\overline{M_1}$ is defined). Hence, $\overline{M}$ and $\widehat{M}$ are defined when IA2DFA(M, l) is called from IA2DFA$(\lambda x.M, l)$.

Finally, for IA2DFA$(\mathbf{mkvar}(M, N), [s])_i^w$, we know from the induction hypothesis that $\overline{M}$ is defined, because $\widehat{\mathbf{mkvar}(M, N)}$ is defined and $\widehat{\mathbf{mkvar}(M, N)} = \mathbf{mkvar}(\overline{M}, \overline{N})$. Since ARGS$(M)[1] = h$ and $arg(h) = i$, when IA2DFA$(M, [s])$ is called, we have $\widehat{M} = \overline{M}i$. $\square$

The Theorem shows in particular that IA2DFA never blocks because of undefinability of some value of *arg*. Therefore, having printed out a set of transitions, it always terminates and returns a state as a result.

Lemma 22 *IA2DFA(P', l) produces a semantic automaton (we take the returned state as the initial one and l as the final list).*

PROOF. First we show that during the execution of IA2DFA(M, l) no transitions from the final states in l are generated. Let us first look at the interpretation of *free* identifiers.

For $x : com$ we get $s' \xrightarrow{run_x} s'' \xrightarrow{done_x} s$.

For $x : exp$ the result is $s' \xrightarrow{q_x} s'' \xrightarrow{i_x} s_i$, where $0 \leq i \leq max$.

For $x : var$, IA2DFA$(x, [s_0, \cdots, s_{max}])^r$ and IA2DFA$(x, [s])_i^w$ produce

$$s' \xrightarrow{read_x} s'' \xrightarrow{j_x} s_j \qquad \text{and} \qquad s' \xrightarrow{write(i)_x} s'' \xrightarrow{ok_x} s$$

respectively.

For $f_z : B_1 \to \ldots \to B_{ar(f)} \to B$ one of the following groups of transitions is generated first as for base-type identifiers (depending on B):

$$\begin{array}{ll} s' \xrightarrow{run_f} s'' \xrightarrow{done_f} s & \qquad s' \xrightarrow{q_f} s'' \xrightarrow{i_f} s_i \\ s' \xrightarrow{read_f} s'' \xrightarrow{i_f} s_i & \qquad s' \xrightarrow{write(i)_f} s'' \xrightarrow{ok_f} s. \end{array}$$

After that, loops of one of the shapes below are created for each $1 \le j \le ar(f)$.

$$\begin{tikzcd}
s'' \arrow[d, bend right, "run_{f,j}"'] \\
s^j = \mathrm{IA2DFA}(arg(f_z, j), [u^j]) \arrow[u, bend right, "done_{f,j}"']
\end{tikzcd}$$

$$\begin{tikzcd}
s'' \arrow[d, bend right, "q_{f,j}"'] \\
s^j = \mathrm{IA2DFA}(arg(f_z, j), [u^j_0, \cdots, u^j_k, \cdots, u^j_{max}]) \arrow[u, bend right, "k_{f,j}"']
\end{tikzcd}$$

$$\begin{tikzcd}
s^j = \mathrm{IA2DFA}(arg(f_z, j), [u^j_0, \cdots, u^j_k, \cdots, u^j_{max}])^r \arrow[d, bend left, "k_{f,j}"] \\
s'' \arrow[u, bend left, "read_{f,j}"] \arrow[d, bend right, "write(k)_{f,j}"'] \\
s^j_k = \mathrm{IA2DFA}(arg(f_z, j), [u^j])^w_k \arrow[u, bend right, "ok_{f,j}"']
\end{tikzcd}$$

In each of the above cases only transitions ending in the final state are produced and for the other cases a simple recursive argument suffices. Hence, $\mathrm{IA2DFA}(M, l)$ can produce transitions involving the states from l or 'fresh' states but no outgoing transitions from the states in l will be printed out at this stage. In particular, there will be no outgoing transition from ∞.

Now we can prove by induction on the order defined by the recursive tree of calls to IA2DFA (where leaves correspond to the base cases) that the generated automata are deterministic in the sense of Definition 12. It is clear that the automata generated for constants (no transitions) and base-type free identifiers are deterministic. The cases relying on a single recursive call are easy too, because a single appeal to the induction hypothesis will suffice.

For first-order identifiers the recursive calls produce disjoint automata because the final lists passed as arguments are disjoint. Because of the way the automata are combined (see diagrams above) nondeterminism will never arise.

For **ifzero**, the first two calls have access to the same final list but, since the recursive calls do not define transitions leading from final states, the two automata put together still define a deterministic automaton. For the same reason the third call using s cannot break determinacy. Virtually the same argument applies to $M; N$ and $M := N$.

For **while**, the two calls might share s and s_N (if $s_N = s_1$) but like before no transitions from s or s_N are then defined. Consequently, the automaton produced in the two calls is deterministic. Finally, the ϵ-transition is deterministic (in the sense of the definition of semantic automata) because no transitions from s_N could have been defined in previous stages. $\square$

In order to formulate an invariant applicable to intermediate IA2DFA calls we will need to suppress some optimizations in the code from Figure 5. This is necessary to specify the meaning of the results for terms of type *exp*. To be able to describe it precisely we have to make sure that when IA2DFA is called, the final list contains $max + 1$ different states. It should be clear that with the new definitions given in Figure 8 the initial call to IA2DFA (for P') will produce an automaton which is equivalent to that generated by the original IA2DFA. Recall that the shape of the initial call to IA2DFA depends on B.

Proposition 23 *Following the initial call to IA2DFA:*

- *IA2DFA*(M, l) *outputs an automaton accepting* $(\!|\Gamma \vdash \widehat{M}|\!)$,
- *IA2DFA*$(M, l)^r$ *outputs an automaton accepting* $(\!|\Gamma \vdash \widehat{M}|\!)^r$,
- *IA2DFA*$(M, l)_i^w$ *outputs an automaton accepting* $(\!|\Gamma \vdash \widehat{M}|\!)_i^w$,

as explained in Definition 13. $\widehat{M}$ *is to be calculated at the moment when IA2DFA*(M, l) *is called (but we already know that* $\widehat{M}$ *will remain the same until IA2DFA*(M, l) *is completed).*

PROOF. We use induction on the order determined by the tree of recursive calls. If M is a constant the result is obvious. If M is a free base-type identifier the generated automaton is shown in the proof of Lemma 22 (and can be seen to be correct by comparison with [9,10]). If M is a base-type identifier but is not free, then by Lemma 21 $arg(x)$ is defined when IA2DFA(x, l) is called. Then we have $\widehat{x} = \widehat{arg(x)}$ so the Theorem holds by induction hypothesis. For **succ**, **pred**, **ifzero**, **while**, $M; N$, $M := N$, $!M$ and first-order variables the result follows from the induction hypothesis and the fact that the composite automata are combined in the right way (see [9,10] for comparison; for first-order identifiers use the figures in the proof of Lemma 22). For application a direct appeal to the induction hypothesis does the job since $\widehat{MN} = \widehat{M}$. For λ-abstraction the proof of Lemma 21 shows that $\widehat{\lambda x.M}$ is β-equivalent to $\widehat{M}$. Because the game semantics of β-equivalent terms is identical, it suffices to appeal to the induction hypothesis again. Finally, since $(\!|\Gamma \vdash \mathbf{mkvar}(M, N)|\!)^r = (\!|\Gamma \vdash N|\!)$, the defining clause for IA2DFA$(\mathbf{mkvar}(M, N), l)^r$ is correct. Similarly, as $(\!|\Gamma \vdash \mathbf{mkvar}(M, N)|\!)_i^w = (\!|\Gamma \vdash Mi|\!)$ holds, it suffices to verify that IA2DFA$(M, [s])$ produces an automaton for $(\!|\Gamma \vdash \overline{M}i|\!)$. By induction hypothesis this is indeed the case, because $arg(h) = i$ will hold throughout its runtime

$$\begin{array}{ll}
\text{IA2DFA}(\mathbf{succ}(M), [s_0, \cdots, s_{max}]) = & \mathit{fresh}(s_\infty) \\
& \text{PRINT}(s_\infty \xrightarrow{\epsilon} \infty) \\
& \text{IA2DFA}(M, [s_1, \cdots, s_{max}, s_\infty])
\end{array}$$

$$\begin{array}{ll}
\text{IA2DFA}(\mathbf{pred}(M), [s_0, \cdots, s_{max}]) = & \mathit{fresh}(s_\infty) \\
& \text{PRINT}(s_\infty \xrightarrow{\epsilon} \infty) \\
& \text{IA2DFA}(M, [s_\infty, s_0, \cdots, s_{max-1}])
\end{array}$$

$$\begin{array}{ll}
\text{IA2DFA}(\mathbf{ifzero}\, M N_0 N_1, l) = & s_0 = \text{IA2DFA}(N_0, l) \\
& s_1 = \text{IA2DFA}(N_1, l) \\
& \mathit{fresh}(u_1, u_2, \cdots, u_{max}) \\
& \mathbf{for}\ i = 1\ \mathbf{to}\ max\ \mathbf{do} \\
& \qquad \text{PRINT}(u_i \xrightarrow{\epsilon} s_1) \\
& \text{IA2DFA}(M, [s_0, u_1, u_2, \cdots, u_{max}])
\end{array}$$

$$\begin{array}{ll}
\text{IA2DFA}(\mathbf{while}\ M\ \mathbf{do}\ N, [s]) = & \mathit{fresh}(s_N) \\
& s_1 = \text{IA2DFA}(N, [s_N]) \\
& \mathit{fresh}(u_2, \cdots, u_{max}) \\
& \mathbf{for}\ i = 2\ \mathbf{to}\ max\ \mathbf{do} \\
& \qquad \text{PRINT}(u_i \xrightarrow{\epsilon} s_1) \\
& s_M = \\
& \quad \text{IA2DFA}(M, [s, s_1, u_2, \cdots, u_{max}]) \\
& \text{PRINT}(s_N \xrightarrow{\epsilon} s_M) \\
& \text{RETURN } s_M
\end{array}$$

$$\begin{array}{ll}
\text{IA2DFA}(M := N, [s]) = & \mathbf{for}\ i = 0\ \mathbf{to}\ max\ \mathbf{do} \\
& \qquad s_i = \text{IA2DFA}(M, [s])^w_i \\
& \mathit{fresh}(u_1, \cdots, u_{max}) \\
& \mathbf{for}\ i = 1\ \mathbf{to}\ max\ \mathbf{do} \\
& \qquad \text{PRINT}(u_i \xrightarrow{\epsilon} s_i) \\
& \text{IA2DFA}(N, [s_0, u_1 \cdots, u_{max}])
\end{array}$$

Fig. 8. Less efficient IA2DFA.

and so $\widehat{M}$ will be equal to $\overline{M}i$ when IA2DFA$(M, [s])$ is called. □

Theorem 24 (Correctness) *Suppose* $\Gamma \vdash P' : B$.

- *For* $B = com$, *IA2DFA*$(P', [0])$ *outputs a semantic automaton accepting* $(\!|\Gamma \vdash P'|\!)$.
- *For* $B = exp$, *IA2DFA*$(P', [0, \cdots, max])$ *outputs a semantic automaton accepting* $(\!|\Gamma \vdash P'|\!)$.
- *For* $B = var$, *IA2DFA*$(P', [0, \cdots, max])^r$ *and IA2DFA*$(P', [0])^w_i$ $(0 \leq i \leq max)$ *produce semantic automata accepting* $(\!|\Gamma \vdash P'|\!)^r$ *and* $(\!|\Gamma \vdash P'|\!)^w_i$

$(0 \leq i \leq max)$ respectively.

PROOF. Since $\widehat{P'} = P'$, it suffices to appeal to the preceding Proposition. $\square$

Complexity

Both ARGS and IA2DFA use subterms of P' as arguments. We can represent each such subterm by the index of its leftmost character in P', which will require $O(\log|P'|)$ space.

ARGS is based on a simple traversal of the syntactic tree of P', so the depth of the recursion cannot exceed $|P'|$. To implement it, we need to store the argument (a subterm of P') and the intermediate result for each recursive call. The former can be done in $O(\log|P'|)$ space, for the latter $O(|P'|\,\log|P'|)$ will suffice, because we need to store a list with up to $|P'|$ entries each of which is an occurrence of x or (f_z, j), where f_z is an occurrence of a first-order identifier (and the occurrences can be represented in $O(\log|P'|)$ space). Because $O(|P'|\,\log|P'|)$ space is needed for each call and the depth of the recursion does not exceed $|P'|$, ARGS can be implemented to run in $O(|P'|^2\,\log|P'|)$ space. Additionally, we must preserve the results of the DEFINE clauses for future use by IA2DFA. But *arg* is a function from x or (f_z, j) to subterms of P', so we will be able to do that in $O(|P'|\,\log|P'|)$ space.

We have already remarked that the recursion stack used in IA2DFA never gets deeper than $|P'|$. Observe that each call to IA2DFA can make $O(|P'|)$ direct recursive calls and generate $O(|P'|)$ fresh states: the worst case is the code labelled REST in Figure 7 where $ar(f) \leq |P'|$ iterations are made; in all other cases the number of calls and new states are both uniformly bounded by a multiple of max, so they contribute only $O(1)$ calls and fresh states. Let c be the larger constant implied by the two $O(|P'|)$ estimates. Then the tree produced by recursive calls of IA2DFA has at most $(c|P'|)^{|P'|}$ nodes. Since at most $c|P'|$ fresh states can be created at each node, in total IA2DFA can produce up to $(c|P'|)^{|P'|}(c|P'|)$ states. They are natural numbers so $O(|P'|\,\log|P'|)$ space will be needed to store each of them and to support fresh-name generation. Consequently, one needs $O(|P'|^2\,\log|P'|)$ space to implement IA2DFA, because the stack will have at most $|P'|$ frames and for each call we have to remember the arguments (a subterm of P' requiring $O(\log|P'|)$ space plus a list of up to max states requiring $O(|P'|\,\log|P'|)$ space) and sometimes a bounded number of states generated inside the call for future use (again $O(|P'|\,\log|P'|)$ space). Note that the automaton produced by IA2DFA can be of exponential size but since it is printed out on the output tape we have:

Theorem 25 *Recall the notation used in Corollary 10. Let* $\Gamma' = \Gamma, X_1 : var, \cdots, X_m : var$*. The semantic automaton accepting* $(\!|\Gamma' \vdash P'|\!)$*,* $(\!|\Gamma' \vdash P'|\!)^r$*,* $(\!|\Gamma' \vdash P'|\!)_i^w$ *(where applicable) can be computed by a PSPACE transducer.*

The automata for $(\!|\ldots|\!)$ can easily be modified to accept $[\![\Gamma \vdash P']\!]$. It suffices to introduce two new states $start, end \in \mathbb{N}$, which we designate as the initial and final states of the new automaton, and to add transitions of the following shapes (as appropriate):

$$\begin{array}{cccc} start \xrightarrow{run} & start \xrightarrow{q} & start \xrightarrow{read} & start \xrightarrow{write(i)} \\ 0 \xrightarrow{done} end & i \xrightarrow{i} end & i \xrightarrow{i} end & 0 \xrightarrow{ok} end. \end{array}$$

The targets of the transitions from *start* are the states returned by IA2DFA. For $P' : var$, before the automata produced by IA2DFA$(P', [0, \cdots, max])^r$ and IA2DFA$(P', [0])_i^w$ $(0 \leq i \leq max)$ are combined, one has to make sure that they are disjoint (e.g. by attaching different tags to states). The resulting (semantic) automaton will be referred to as $A_{P'} = \langle\, Q \cup \{\, start, end \,\}, start, \delta, [end] \,\rangle$.

5 Producing the stateful automaton

Using Corollary 10, we will now show how to construct a deterministic automaton accepting $[\![\Gamma \vdash P]\!]$, also in PSPACE. Recall that $|P'| = O(|P| \log |P|)$. Since $A_{P'}$ may already be of exponential size, it cannot be stored. Instead, each time we need to look up a transition from $A_{P'}$, we will call IA2DFA from scratch and wait until the relevant information is printed out. All the other transitions that are output will be ignored (rather than stored).

The states of $A_{P'}$ can be partitioned into those from which O is to move (*O-states*) and those from which only P-moves can follow (*P-states*), e.g. *start* is an O-state. Transitions of $A_{P'}$ always involve states belonging to different players with the exception of ϵ-transitions, which are between two P-states. Since the strategies we consider are deterministic, at most one transition is available from a P-state. The distinction between O-states and P-states will help to create the automaton corresponding to **new** $X_1, \cdots, X_m$ **in** P' $(m \leq |P|)$, in which state changes are respected and hidden. Recall that the transitions generated by IA2DFA are of the form $s_1 \xrightarrow{c} s_2$, where $s_1, s_2 \in Q \subseteq \mathbb{N}$. In this section we will add state information to them, so each new transition will have one of the following shapes:

$$start \xrightarrow{c} s_2^{\overbrace{(0, \cdots, 0)}^{m}} \qquad \text{or} \qquad s_1^{\vec{x}} \xrightarrow{c} s_2^{\vec{y}} \qquad \text{or} \qquad s_1^{\vec{x}} \xrightarrow{c} end,$$

$\mathit{find}(S_1, \vec{x}, \vec{y}, S_2) =$

Because S_2 is a P-state, there exists at most one pair (c, S_3) such that $S_2 \xrightarrow{c} S_3$ is a transition from $A_{P'}$. If none exists HALT.

(1) If c is not a state move:
PRINT($S_1^{\vec{x}} \xrightarrow{c} S_3^{\vec{y}}$) if $S_3 \neq \mathit{end}$,
PRINT($S_1^{\vec{x}} \xrightarrow{c} \mathit{end}$) otherwise,
in either case HALT afterwards.

(2) If $c = \epsilon$ then $\mathit{find}(S_1, \vec{x}, \vec{y}, S_3)$.

(3) If $c = \mathit{write}(i)_{X_j}$, let S_4 be the unique state such that $S_3 \xrightarrow{ok_{X_j}} S_4$ is a transition in $A_{P'}$ and call

$$\mathit{find}(S_1, \vec{x}, (y_1, \cdots, y_{j-1}, i, y_{j+1}, \cdots, y_m), S_4).$$

(4) If $c = \mathit{read}_{X_j}$, let S_4 be the unique state such that $S_3 \xrightarrow{y_{j\,X_j}} S_4$ is a transition of $A_{P'}$ and call $\mathit{find}(S_1, \vec{x}, \vec{y}, S_4)$.

Fig. 9. The *find* procedure.

---

where $\vec{x} = (x_1, \cdots, x_m)$ and $\vec{y} = (y_1, \cdots, y_m)$ are elements of $\{0, \cdots, \mathit{max}\}^m$ and c is not a *state move* (by state moves we mean ϵ and any of $\mathit{write}(i)_{X_j}$, ok_{X_j}, read_{X_j}, i_{X_j} for $1 \leq j \leq m$, $0 \leq i \leq \mathit{max}$). $\vec{x}$ and $\vec{y}$ will reflect the state changes caused by playing c. Hence, the states of the new automaton will be *start*, *end* and $q^{\vec{x}}$ for any $q \in Q, \vec{x} \in \{0, \cdots, \mathit{max}\}^m$. *start* and *end* will remain the initial and final states respectively. In order to define the new transitions we proceed as follows.

(i) For any $A_{P'}$-transition $s_1 \xrightarrow{c} s_2$, if s_1 is an O-state and c is not a state move, then for all $\vec{x} \in \{0, \cdots, \mathit{max}\}^m$, PRINT($s_1^{\vec{x}} \xrightarrow{c} s_2^{\vec{x}}$) if $s_1 \neq \mathit{start}$, otherwise PRINT($\mathit{start} \xrightarrow{c} s_2^{(0,\cdots,0)}$).

(ii) For all P-states s and $\vec{x} \in \{0, \cdots, \mathit{max}\}^m$ call $\mathit{find}(s, \vec{x}, \vec{x}, s)$, where

$$\mathit{find}(S_1, (x_1, \cdots, x_m), (y_1, \cdots, y_m), S_2)$$

is defined in Figure 9. The arguments S_1, S_2 will always be P-states.

Complexity

find works by following paths in $A_{P'}$. Its definition is tail-recursive and it can be executed as a loop. We will show that, like before, the new transitions can be printed out by a PSPACE transducer. At each state some information about $A_{P'}$ will be needed so we will need to run IA2DFA. Because we cannot store the whole $A_{P'}$ in polynomial space, we will only allocate space for one transition so each PRINT instruction will overwrite the previous one. In this way we can still observe the output of IA2DFA without violating the PSPACE bound.

Let us discuss part (i) first. To implement (i), we need to generate the requisite transitions without repetition, which can be done by calling IA2DFA repeatedly and memorizing the last transition processed. After IA2DFA prints out a transition starting from an O-state we simply adorn it with all possible tuples, which can be done in PSPACE using $m \leq |P|$ nested loops.

Part (ii) is more complicated. To implement the *find* loop for a given s and $\vec{x}$, we need to store $S_1, \vec{X}$ (which are always equal to $s, \vec{x}$) and $S_2, \vec{y}$ (which do change). $O(|P'| \log |P'|)$ space is sufficient for the states (see the previous complexity section), $\vec{x}$ can be stored in $O(|P|)$ space. At each iteration one transition from $A_{P'}$ will be needed, which we can get by calling IA2DFA and waiting until it is printed out (if it exists). Therefore, for a given $s, \vec{x}$ the *find* loop can be implemented to run in polynomial space. Unfortunately, the loop might not terminate in general. However, since the number of all possible configurations is $(max + 1)^m |Q|$, divergence can be detected with the help of a counter of polynomial size (then we simply stop without generating any transition). Thus *find* is implementable in PSPACE, but we have to iterate the process for all $\vec{x}$ and all P-states. The former can be done via nested loops (as in (i)), the latter requires us to memorize the previously processed P-state in order to avoid repetitions (a P-state is a source of a unique transition).

Theorem 26 *For any* IA_2 *term* P*,* $[\![\Gamma \vdash P : T]\!]$ *is accepted by a deterministic automaton (without* ϵ*-transitions) which is computable by a PSPACE transducer.*

To test equivalence or approximation we need to port the above transducer with the Turing machines (from Theorem 4) that decide respectively equivalence and containment of deterministic finite automata. Moreover, this should be done in polynomial space, so the obvious sequencing of the machines will not do. Instead, we will compose the two machines in the same way as that in which two logarithmic-space reductions are combined to produce a logarithmic space reduction [16]. We sketch the solution briefly. Obviously we cannot afford to store the whole output tape of the PSPACE transducer but, since it runs in PSPACE, it will produce output of size $O(2^{|P|^k})$ for some $k \in \mathbb{N}$. But the logarithmic space acceptor must be able to scan the whole tape and, to accommodate that, we can represent its input head by a counter c of size $O(|P|^k)$. Then, each time the head symbol is needed, we will rerun the transducer until it outputs the cth symbol. By composing an NL acceptor with a PSPACE transducer in this fashion we obtain an NPSPACE acceptor, which can be converted to a PSPACE acceptor using Savitch's theorem [16].

Theorem 27 *Program equivalence and approximation of* IA_2 *terms can be decided in polynomial space.*

6 Hardness

We show that some classic problems about boolean formulas can be reduced to questions about program equivalence or approximation in various fragments of IA_2. Let us write $\mathsf{IA}_2^{\mathrm{min}}$ for the sublanguage of IA_2 consisting of all constants, **succ**, **pred**, **ifzero**, **new**, assignment and dereferencing. $\mathsf{IA}_2^{\mathrm{min}}$ could be viewed as a minimal language for programming with state.

Boolean formulas are generated by the grammar $F ::= X_i \mid F \vee F \mid F \wedge F \mid \neg F$, where $i \in \mathbb{N}$. We write $F(X_1, \cdots, X_k)$ if the variables occurring in F are among $\{X_1, \cdots, X_k\}$. It is well-known that that the decision problem TAUTOLOGY (to decide whether a given boolean formula is a tautology) is coNP-complete (see e.g. [14]).

Given a boolean formula $F(X_1, \cdots, X_k)$ let us define a corresponding $\mathsf{IA}_2^{\mathrm{min}}$ term $X_1 : \mathit{var}, \cdots, X_k : \mathit{var} \vdash M_F : \mathit{exp}$ by

$$\begin{aligned}
M_X &= \mathbf{ifzero}\,!X\,0\,1,\\
M_{F_1 \vee F_2} &= \mathbf{ifzero}\,M_{F_1} M_{F_2} 1,\\
M_{F_1 \wedge F_2} &= \mathbf{ifzero}\,M_{F_1} 0 M_{F_2},\\
M_{\neg F} &= \mathbf{ifzero}\,M_F 1\,0.
\end{aligned}$$

Theorem 28 *F is a tautology if and only if*

$$x : \mathit{exp} \vdash \mathbf{new}\ X_1, \cdots, X_k\ \mathbf{in}\ (X_1 := x; \cdots; X_k := x; M_F) : \mathit{exp}$$

is equivalent to (or approximates)

$$x : \mathit{exp} \vdash \mathbf{new}\ X_1, \cdots, X_k\ \mathbf{in}\ (X_1 := x; \cdots; X_k := x; 1) : \mathit{exp}.$$

PROOF. The encoding relies on the fact that the value of x may vary in the k assignments, which can be viewed as repeated evaluations of x. The first term corresponds to evaluating F for an assignment of truth values to its free variables, so F is a tautology if and only if M_F always yields 1. The second term uses x in the same way as the first one but it will always return 1 like any tautology would. The argument can easily be formalized using Theorem 3. □

Consequently, program equivalence and approximation in $\mathsf{IA}_2^{\mathrm{min}}$ are coNP-hard. Conversely, a close look at IA2DFA reveals that without first-order identifiers, application and **while** the generated automaton has linear size and no loops. Thus a trace certifying inequivalence of two $\mathsf{IA}_2^{\min}$ terms can be guessed and verified in polynomial time.

Theorem 29 $\mathsf{IA}_2^{\min}$ *program equivalence and approximation are coNP-complete.*

The inclusion of procedures, first-order identifiers or loops makes the problems PSPACE-hard.

Definition 30 Let $\Box X$ denote either $\forall X$ or $\exists X$. Any formula of the shape

$$\Box X_1.\cdots\Box X_k.F(X_1,\cdots,X_k)$$

where F is a boolean formula, is called a *totally quantified boolean formula.*

A totally quantified boolean formula is either valid or invalid and the problem TQBF of deciding validity is PSPACE-complete (see e.g. [14]). Below we present three reductions of TQBF to IA_2 program equivalence. In the first two cases for each totally quantified boolean formula G we define a closed term $M_G : \mathit{exp}$. Testing validity is then equivalent to checking whether M_G is equivalent to $1 : \mathit{exp}$ (equivalently, whether M_G approximates 1).

Using **while** we can extend the previous inductive assignment of IA_2 terms to formulas with

$$\begin{aligned} M_{\forall X.G} = \mathbf{new}\ X, Z\ \mathbf{in}\ &(Z := 1; X := 2;\\ &\quad \mathbf{while}\ (!X)\ \mathbf{do}\ (X := \mathbf{pred}(!X); \mathbf{ifzero}\, M_G (Z := 0) \mathit{skip});\\ &!Z),\end{aligned}$$

$$\begin{aligned} M_{\exists X.G} = \mathbf{new}\ X, Z\ \mathbf{in}\ &(Z := 0; X := 2;\\ &\quad \mathbf{while}\ (!X)\ \mathbf{do}\ (X := \mathbf{pred}(!X); \mathbf{ifzero}\, M_G\, \mathit{skip} (Z := 1));\\ &!Z).\end{aligned}$$

This works because each loop makes two iterations and stores respectively $G(0) \wedge G(1)$ and $G(0) \vee G(1)$ in Z.

λ-abstraction and application can be used to replace **while**:

$$\begin{aligned} &M_{\forall X.G} = \mathbf{new}\ X, Z\ \mathbf{in}\ (Z := 1;\\ &(\lambda x^{\mathit{exp}}.\mathbf{ifzero}\, (X := 0; x)(Z := 0)(\ \mathbf{ifzero}\, (X := 1; x)(Z := 0)\mathit{skip}\ ))\ M_G;\\ &\qquad !Z),\end{aligned}$$

$$\begin{aligned} &M_{\exists X.G} = \mathbf{new}\ X, Z\ \mathbf{in}\ (Z := 0;\\ &(\lambda x^{\mathit{exp}}.\mathbf{ifzero}\, (X := 0; x)(\ \mathbf{ifzero}\, (X := 1; x)\ \mathit{skip}\ (Z := 1)\ )(Z := 1))\ M_G;\\ &\qquad !Z).\end{aligned}$$

If first-order variables are available yet another reduction is possible, but now it produces a term $f : com \to com \vdash M_G : exp$:

$$\begin{aligned}
M_{\forall X.G} = \;& \textbf{new}\ X, Z\ \textbf{in}\ (Z := 1; X := 2; \\
& \quad f(\ X := \textbf{pred}(!X);\ \textbf{ifzero}\, M_G(Z := 0)\mathit{skip}\ ); \\
& \textbf{ifzero}\,(!X)\ \mathit{skip}\, \Omega_{com}; \\
& !Z),
\end{aligned}$$

$$\begin{aligned}
M_{\exists X.G} = \;& \textbf{new}\ X, Z\ \textbf{in}\ (Z := 0; X := 2; \\
& \quad f(\ X := \textbf{pred}(!X);\ \textbf{ifzero}\, M_G\mathit{skip}(Z := 1)\ ); \\
& \textbf{ifzero}\,(!X)\ \mathit{skip}\, \Omega_{com}; \\
& !Z).
\end{aligned}$$

As before we force the 'function' f to investigate its argument precisely twice. In this case a totally quantified formula G is valid iff $f : com \to com \vdash M_G \cong M'_G$ or $M_G \sqsubsetapprox M'_G$, where M'_G is the same as M_G except that for the outermost quantifier $!Z$ is replaced with 1.

All three reductions are of polynomial time (logarithmic space) complexity, because none of the encodings duplicates M_G for modelling quantification. By Theorem 27 we have:

Theorem 31 *Program equivalence and approximation in* IA_2 *are PSPACE-complete.*

7 Optimizations

The PSPACE algorithm leading to Theorem 27 relies on constructions which make it naive to expect polynomial runtime even for simple programs. This is because at many stages the generating procedure must be run again and again to save space, which in turn increases runtime in a significant way (this idea underlies the passage from the automaton for P' to that for P, the composition with the nondeterministic verifier as well as Savitch's Theorem). Therefore, it seems that for practical purposes a possibly exponential space algorithm should be used. We can suggest several improvements to IA2DFA and *find* so that our algorithm leads to better time complexity.

For instance, in the first stage all information about variable scope is forgotten, whereas it could be recorded and taken advantage of in the *find* procedure. Then one would not have to generate m-tuples but only tuples corresponding to the variables whose scope actually extends over the given subterm. In IA2DFA the clause for **while** could also be optimized to detect simple divergences: before the PRINT instruction **if** $s_M = s_N$ **then** RETURN ∞ could be added. This would detect some terms equivalent to **while** 1 **do** *skip* without the need to create a loop in the automaton. In *find* one could also employ a better mechanism to detect divergence and try to generate only transitions which are actually reachable. A natural way to do that seems to be a depth-first search of the automaton produced by IA2DFA.

The automata corresponding to strategies are very sparse. Therefore one can count on a considerable reduction of space consumption if an economical representation scheme is used [17].

8 Conclusion

We have investigated the complexity of a simple imperative programming language IA_2 using its game model. Our results (Theorems 29, 31) are summarized in the table below, where the right column refers to the complexity of program equivalence in the respective fragment (in each case it turned out that program approximation had the same complexity as program equivalence).

Fragment	Complexity
$\mathsf{IA}_2^{\min}$	coNP
$\mathsf{IA}_2^{\min}$+**while**	PSPACE
$\mathsf{IA}_2^{\min}+\lambda x^B+$ application	PSPACE
$\mathsf{IA}_2^{\min}+$ first-order identifiers	PSPACE
IA_2	PSPACE

One might also ask how the presence of state affects the complexity. For purely functional programs (without !, := or **new**) our approach still implies a PSPACE algorithm, since the automata involved can have exponential size because of procedures. Without them however, a PTIME algorithm can be extracted. On the other hand, it is known that IA_2 enriched with a **let** construct for procedures (i.e. IA_2 with λ-abstraction and application extended to all IA_2 types) can also be captured by regular languages [18]. After inlining the **let**'s each IA_2+**let** term becomes a (potentially exponentially larger) IA_2 term, so our approach would yield an EXPSPACE algorithm in this case. We were unable to prove completeness in these cases though.

As future work we plan to investigate the complexity of call-by-value programs. The categorical framework for modelling call-by-value [4] is more complicated than that of call-by-name models and the game model is not understood as well as for call-by-name. However, call-by-value fragments with regular semantics have already been found in [19] (for block-allocated variables) and in [20] (for a fragment of ML).

Acknowledgements

The author gratefully acknowledges support from EPSRC (GR/R88861/01) and St John's College, Oxford.

References

[1] S. Abramsky, R. Jagadeesan, P. Malacaria, Full abstraction for PCF, Information and Computation 163 (2000) 409–470.

[2] J. M. E. Hyland, C.-H. L. Ong, On full abstraction for PCF: I. Models, observables and the full abstraction problem, II. Dialogue games and innocent strategies, III. A fully abstract and universal game model, Information and Computation 163(2) (2000) 285–408.

[3] S. Abramsky, G. McCusker, Linearity, sharing and state: a fully abstract game semantics for Idealized Algol with active expressions, in: P. W. O'Hearn, R. D. Tennent (Eds.), Algol-like languages, Birkhaüser, 1997, pp. 297–329.

[4] S. Abramsky, G. McCusker, Call-by-value games, in: Proceedings of CSL, Vol. 1414 of Lecture Notes in Computer Science (1997) 1–17.

[5] K. Honda, N. Yoshida, Game-theoretic analysis of call-by-value computation (extended abstract), in: Proceedings of ICALP, Vol. 1256 of Lecture Notes in Computer Science (1997) 225–236.

[6] J. Laird, A semantic analysis of control, Ph.D. thesis, University of Edinburgh, 1998.

[7] S. Abramsky, K. Honda, G. McCusker, Fully abstract game semantics for general references, in: Proceedings of IEEE Symposium on Logic in Computer Science (1998) 334–344.

[8] S. Abramsky, G. McCusker, Full abstraction for Idealized Algol with passive expressions, Theoretical Computer Science 227 (1999) 3–42.

[9] D. R. Ghica, G. McCusker, Reasoning about Idealized Algol using regular expressions, in: Proceedings of ICALP, Vol. 1853 of Lecture Notes in Computer Science (2000) 103–115.

[10] S. Abramsky, Algorithmic games semantics: a tutorial introduction, in: H. Schwichtenberg, R. Steinbruggen (Eds.), Proof and System Reliability, Kluwer Academic Publishers, 2001, pp. 21–47.

[11] G. McCusker, On the semantics of Idealized Algol without the bad-variable constructor., in: Proceedings of MFPS, Electronic Notes in Theoretical Computer Science (2003).

[12] S. Abramsky, G. McCusker, Game semantics, in: H. Schwichtenberg, U. Berger (Eds.), Logic and Computation, Springer-Verlag, 1998, pp. 1–56.

[13] M. R. Garey, D. S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, Freeman, 1979.

[14] M. Sipser, Introduction to the Theory of Computation, PWS Publishing Company, 1997.

[15] N. D. Jones, S. S. Muchnick, Even simple programs are hard to analyze, JACM 24(2) (1977) 338–350.

[16] C. H. Papadimitriou, Computational Complexity, Addison-Wesley, 1994.

[17] G. A. Kiraz, Compressed storage of sparse finite-state transducers, in: Proceedings of WIA, Vol. 2214 of Lecture Notes in Computer Science (1999) 109–121.

[18] D. R. Ghica, G. McCusker, The regular language semantics of second-order Idealized Algol, Theoretical Computer Science 309 (2003) 469–502.

[19] D. R. Ghica, Regular-language semantics for a call-by-value programming language, in: Proceedings of MFPS, Vol. 45 of Electronic Notes in Computer Science (2001).

[20] A. S. Murawski, Functions with local state: regularity and undecidability, Theoretical Computer Science, to appear.