Third-Order Idealized Algol with Iteration is Decidable

Andrzej S. Murawski[1⋆] and Igor Walukiewicz[2⋆⋆]

[1] Oxford University Computing Laboratory, Parks Road, Oxford OX1 3QD, UK
[2] LaBRI, Université Bordeaux-1, 351, Cours de la Libération, 33 405, Talence, France

Abstract. The problems of contextual equivalence and approximation are studied for the third-order fragment of Idealized Algol with iteration (IA_3^*). They are approached via a combination of game semantics and language theory. It is shown that for each IA_3^*-term one can construct a pushdown automaton recognizing a representation of the strategy induced by the term. The automata have some additional properties ensuring that the associated equivalence and inclusion problems are solvable in PTIME. This gives an EXPTIME decision procedure for contextual equivalence and approximation for β-normal terms. EXPTIME-hardness is also shown in this case, even in the absence of iteration.

1 Introduction

In recent years game semantics has provided a new methodology for constructing fully abstract models of programming languages. By definition, such models capture the notions of contextual equivalence and approximation and so offer a semantic framework in which to study these two properties. In this paper we focus on the game semantics of Idealized Algol, a language proposed by Reynolds as a synthesis of functional and imperative programming [1]. It is essentially the simply-typed λ-calculus extended with constants for modelling arithmetic, assignable variables and recursion. This view naturally determines fragments of the language when the typing framework is constrained not to exceed a particular order. Many versions of Algol have been considered in the literature. Typically, for decidability results, general recursion has to be left out completely or restricted to iteration, e.g. in the form of while-loops as will be the case in this paper. For similar reasons, base types are required to be finite.

In game models, terms of a programming language are modelled by strategies. These in turn can sometimes be represented by formal languages, i.e. sets of finite words, such that equivalence and approximation are established by verifying respectively equality and inclusion of the induced languages. This approach to modelling semantics is interesting not only because it gives new insights into the semantics but also because it opens up the possibility of applying existing algorithms and techniques developed for dealing with various families of formal

⋆ Supported by British EPSRC (GR/R88861) and St John's College, Oxford.
⋆⋆ Supported by the European Community Research Training Network GAMES.

languages [2]. Therefore, it is essential that the class of languages one uses is as simple as possible – ideally its containment problem should be decidable and of relatively low complexity.

In this paper we show how to model terms of third-order Idealized Algol with iteration (IA_3^*) using variants of visibly pushdown automata [3]. One of the advantages of taking such specialized automata is that the instances of the containment problem relevant to us will be decidable in PTIME. Another is the relative simplicity of the inductive constructions of automata for the constructs of the language. We give the constructions only for terms in β-normal form taking advantage of the fact that each term can be effectively normalized. The automata constructed by our procedure have exponential size with respect to the size of the term, which leads to an exponential-time procedure for checking approximation and equivalence of such terms. We also provide the matching lower bound by showing that equivalence of third-order terms, even without iteration, is EXPTIME-hard.

Ghica and McCusker [4] were the first to show how certain strategies can be modelled by languages. They have defined a procedure which constructs a regular language for every term of second-order Idealized Algol with iteration. Subsequently, Ong [5] has shown how to model third-order Idealized Algol without iteration using deterministic pushdown automata. Our work can be seen as an extension of his in two directions: a richer language is considered and a more specialized class of automata is used (the latter is particularly important for complexity issues). In contrast to the approach of [5], we work exclusively with the standard game semantics and translate terms directly into automata, while the translation in [5] relies on an auxiliary form of game semantics (with explicit state) in which strategies are determined by *view-functions*. In the presence of iteration these functions are no longer finite and the approach does not work any more (in yet unpublished work Ong proposes to fix this deficiency by considering view-functions whose domains are regular sets and which act uniformly with respect to the regular expressions representing these sets). It should also be noted that our construction yields automata without pushdowns for terms of order two, hence it also subsumes the construction by Ghica and McCusker.

Our results bring us closer to a complete classification of decidable instances of Idealized Algol and their complexity. Since the fourth-order fragment without iteration was shown undecidable in [6], the only unresolved cases seem to be those of second- and third-order fragments with recursively defined terms of base types (of which iteration is a special case). Recursive functions lead to undecidability at order two as shown in [5].

The outline of the paper is as follows. We present Idealized Algol and its third-order fragment IA_3^* in Section 2. Then we recapitulate the game model of the language. Next the class of *simple* terms is defined. These are terms that induce plays in which pointers can be safely omitted which makes it possible to represent their game semantics via languages. In Section 4 we introduce our particular class of automata and give an inductive construction of such an automaton for every simple term in β-normal form. In Section 5 we show how to deal with terms that are not simple. The last section concerns the EXPTIME lower bound for the complexity of equivalence in IA_3^*.

2 Idealized Algol

We consider a finitary version IA_f of Idealized Algol with active expressions [7]. It can be viewed as a simply typed λ-calculus over the base types *com*, *exp*, *var* (of commands, expressions and variables respectively) augmented with the constants

$skip : com$	$i : exp \quad (0 \leq i \leq max)$	$\Omega_{\mathcal{B}} : \mathcal{B}$
$\mathbf{succ} : exp \to exp$	$\mathbf{pred} : exp \to exp$	$\mathbf{ifzero}_{\mathcal{B}} : exp \to \mathcal{B} \to \mathcal{B} \to \mathcal{B}$
$\mathbf{seq}_{\mathcal{B}} : com \to \mathcal{B} \to \mathcal{B}$	$\mathbf{deref} : var \to exp$	$\mathbf{assign} : var \to exp \to com$
$\mathbf{cell}_{\mathcal{B}} : (var \to \mathcal{B}) \to \mathcal{B}$	$\mathbf{mkvar} : (exp \to com) \to exp \to var$	

where $\mathcal{B}$ ranges over base types and $exp = \{\, 0, \cdots, max \,\}$. Each of the constants corresponds to a different programming feature. For instance, sequential composition of M and N is expressed as $\mathbf{seq}_{\mathcal{B}} MN$, assignment of N to M is represented by $\mathbf{assign} MN$ and $\mathbf{cell}_{\mathcal{B}}(\lambda x.M)$ amounts to creating a local variable x visible in M. Other features can be added in a similar way, e.g. **while**-loops will be introduced via the constant $\mathbf{while} : exp \to com \to com$. In order to gain control over multiple occurrences of free identifiers during typing (cf. Definition 9) we shall use a linear form of the application rule and the contraction rule:

$$\frac{\Gamma \vdash M : T \to T' \quad \Delta \vdash N : T}{\Gamma, \Delta \vdash MN : T'} \qquad \frac{\Gamma, x_1 : T, x_2 : T \vdash M : T'}{\Gamma, x : T \vdash M[x/x_1, x/x_2] : T'}.$$

The linear application simply corresponds to composition: in any cartesian-closed category $[\![\Gamma, \Delta \vdash MN : T']\!]$ is equal (up to currying) to

$$\begin{array}{ccc} [\![\Delta \vdash N : T]\!] & ; & [\![\, \vdash \lambda x^T.\lambda\Gamma.Mx : T \to (\Gamma \to T')]\!] \\ [\![\Delta]\!] \Rightarrow [\![T]\!] & & [\![T]\!] \Rightarrow ([\![\Gamma]\!] \Rightarrow [\![T']\!]). \end{array}$$

Thanks to the applicative syntax and the above decomposition the process of interpreting the language can be divided into simple stages: the modelling of base constructs (free identifiers and constants), composition, contraction and currying.

The operational semantics of IA_f can be found in [7]; we will write $M \Downarrow$ if M reduces to *skip*. We study the induced equivalence and approximation relations.

Definition 1. *Two terms $\Gamma \vdash M_1, M_2 : T$ are* equivalent *($\Gamma \vdash M_1 \cong M_2$) if for any context $C[-]$ 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 ($\Gamma \vdash M_1 \lessapprox M_2$) iff for all contexts satisfying the properties above whenever $C[M_1] \Downarrow$ then $C[M_2] \Downarrow$.*

In general, equivalence of IA_f terms is not decidable [6]. To obtain decidability one has to restrict the order of types, which is defined by:

$$\mathsf{ord}(\mathcal{B}) = 0 \quad \text{and} \quad \mathsf{ord}(\mathcal{T} \to \mathcal{T}') = \max(\mathsf{ord}(\mathcal{T}) + 1, \mathsf{ord}(\mathcal{T}')).$$

Definition 2. *An* IA_{f} *term* $\Gamma \vdash M : T$ *is an* ith-order term *provided its typing derivation uses sequents in which the types of free identifiers are of order less than* i *and the type of the term has order at most* i. *The collection of* i*th-order* IA_{f} *terms will be denoted by* IA_i.

To establish decidability of program approximation or equivalence of ith-order terms it suffices to consider ith-order terms in β-normal form. To type such terms, one only needs a restricted version of the application rule in which the function term M is either a constant or a term of the form $f M_1 \cdots M_k$, where $f : T$ is a free identifier (and so $\mathsf{ord}(T) < i$).

In this paper we will be concerned with IA_3 enriched with **while**, which we denote by IA_3^* for brevity.

3 Game semantics

Here we recall the basic notions of game semantics and discuss how to code strategies in terms of languages. To that end we investigate when it is not necessary to represent pointers in plays and obtain the class of simple terms for which pointers can be disregarded. We use the game semantics of Idealized Algol as described in [7]. The games are defined over arenas which specify the available moves and the relationship between them.

Definition 3. *An* arena A *is a triple* $\langle M_A, \lambda_A, \vdash_A \rangle$, *where* M_A *is the set of moves,* $\lambda_A : M_A \to \{ O, P \} \times \{ q, a \}$ *indicates whether a move is an O-move or a P-move and whether it is a question or an answer, and* $\vdash_A \subseteq (M_A + \{ \star \}) \times M_A$ *is the* enabling *relation which must satisfy the following two conditions.*

- *For all* $m, n \in M_A$ *if* $m \vdash_A n$ *then* m *and* n *belong to different players and* m *is a question.*
- *If* $\star \vdash_A m$ *then* m *is an O-question which is not enabled by any other move. Such moves are called* initial*; the set containing them will be denoted by* I_A.

The permitted scenarios in a given arena are required to be *legal justified sequences* of moves. A justified sequence s over an arena A is a sequence of moves from M_A equipped with pointers so that every non-initial move n (in the sense of Definition 3) in s has a pointer to an earlier move m in s with $m \vdash_A n$ (m is then called the *justifier* of n). Given a justified sequence s, its O-view $\llcorner s \lrcorner$ and P-view $\ulcorner s \urcorner$ are defined as follows, where o and p stand for an O-move and a P-move respectively:

$$\llcorner \epsilon \lrcorner = \epsilon \quad \llcorner s o \lrcorner = \llcorner s \lrcorner o \qquad\qquad \llcorner s\, o\, t\, p \lrcorner = \llcorner s \lrcorner\, o\, p$$
$$\ulcorner \epsilon \urcorner = \epsilon \quad \ulcorner s o \urcorner = o \quad (\text{if } o \text{ is initial}) \quad \ulcorner s p \urcorner = \ulcorner s \urcorner p \quad \ulcorner s\, p\, t\, o \urcorner = \ulcorner s \urcorner\, p\, o.$$

Definition 4. *A justified sequence* s *is* legal *if it satisfies the following:*

- *players alternate (O begins),*
- *the* visibility *condition holds: in any prefix* tm *of* s *if* m *is a non-initial O-move then its justifier occurs in* $\llcorner t \lrcorner$ *and if* m *is a P-move then its justifier is in* $\ulcorner t \urcorner$,

– *the* bracketing *condition holds: for any prefix tm of s if m is an answer then its justifier must be the last unanswered question in t.*

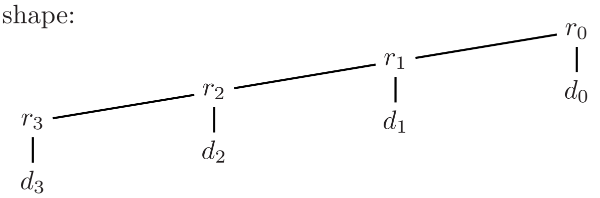
The set of legal sequences over arena A is denoted by L_A.

Formally, a game will be an arena together with a subset of L_A. This makes it possible to define different games over the same arenas.

Definition 5. *A* game *is a tuple $\langle M_A, \lambda_A, \vdash_A, P_A \rangle$ such that $\langle M_A, \lambda_A, \vdash_A \rangle$ is an arena and P_A is a non-empty, prefix-closed subset of L_A (called the set of* positions *or* plays *in the game)*[3].

Games can be combined by a number of constructions, notably $\times, \otimes, !, \multimap, \Rightarrow$. We describe them briefly below. In the first three cases the enabling relation is simply inherited from the component games. As for plays, we have $P_{A\times B} = P_A + P_B$ in the first case. In contrast, each play in $P_{A\otimes B}$ is an interleaving of a play from A with a play from B (and only O can switch between them). Similarly, positions in $P_{!A}$ are interleavings of a finite number of plays from P_A (again only O can jump between them). The $\multimap$ construction is more complicated: we have $M_{A\multimap B} = M_A + M_B$ but the ownership of A moves in $M_{A\multimap B}$ is reversed. The enabling relation is defined by $\vdash_{A\multimap B}=\vdash_A + \vdash_B + \{\, (b, a) \mid b \in I_B \wedge a \in I_A \,\}$ and plays of $A \multimap B$ are interleavings of single plays from A and B. This time, however, each such play has to begin in B and only P can switch between the interleaved plays. The game $A \Rightarrow B$ is defined as $!A \multimap B$.

Example 1. The underlying arena of $((\llbracket com \rrbracket \Rightarrow \llbracket com \rrbracket) \Rightarrow \llbracket com \rrbracket) \Rightarrow \llbracket com \rrbracket$ has the following shape:



Definition 6. *In arenas corresponding to* IA_f *types we can define the* order of a move *inductively (we denote it by $\mathsf{ord}_A(m)$). The initial O-questions have order 0. For all other questions q we define $\mathsf{ord}_A(q)$ to be $\mathsf{ord}_A(q') + 1$ where $q' \vdash q$ (this definition is never ambiguous for the arenas in question). Answers inherit their order from the questions that enable them. The* order of an arena *is the maximal order of a question in it.*

For instance, in the example above r_3 is a third-order move. We will continue to use subscripts to indicate the order of a move.

The next important definition is that of a strategy. Strategies determine unique responses for P (if any) and do not restrict O-moves.

Definition 7. *A* strategy *in a game A (written as $\sigma : A$) is a prefix-closed subsets of plays in A such that: (i) whenever $sp_1, sp_2 \in \sigma$ and p_1, p_2 are P-moves then $p_1 = p_2$; (ii) whenever $s \in \sigma$ and $so \in P_A$ for some O-move o then*

[3] P_A also has to satisfy a closure condition [7] which we omit here.

so $\in \sigma$*. We write* $\mathsf{comp}(\sigma)$ *for the set of non-empty* complete plays *in* σ*, i.e. plays in which all questions have been answered.*

An IA_f term $\Gamma \vdash M : T$, where $\Gamma = x_1 : T_1, \cdots, x_n : T_n$, is interpreted by a strategy (denoted by $[\![\Gamma \vdash M : T]\!]$) for the game

$$[\![\Gamma \vdash T]\!] = [\![T_1]\!] \times \cdots \times [\![T_n]\!] \Rightarrow [\![T]\!] = !([\![T_1]\!]) \otimes \cdots \otimes !([\![T_n]\!]) \multimap [\![T]\!].$$

Remark 1. From the definitions of the $\otimes$ and $\multimap$ constructions we can deduce the following *switching properties*. A play in $[\![\Gamma \vdash T]\!]$ always starts with an initial O-question in $[\![T]\!]$. Subsequently, whenever P makes a move in $[\![T_i]\!]$ or $[\![T]\!]$, O must also follow with a move in $[\![T_i]\!]$ or $[\![T]\!]$ respectively. We also note that the arenas used to interpret ith-order terms are of order i.

The interpretation of terms presented in [7] gives a fully abstract model in the sense made precise below.

Theorem 1. *$\Gamma \vdash M_1 \mathrel{\underset{\sim}{\sqsubset}} M_2$ iff $\mathsf{comp}([\![\Gamma \vdash M_1]\!]) \subseteq \mathsf{comp}([\![\Gamma \vdash M_2]\!])$. Consequently, $\Gamma \vdash M_1 \cong M_2$ iff $\mathsf{comp}([\![\Gamma \vdash M_1]\!]) = \mathsf{comp}([\![\Gamma \vdash M_2]\!])$.*

In the sections to follow we will show how to represent strategies defined by β-normal IA_3^*-terms via languages. The simplest, but not always faithful, representation consists in taking the underlying set of moves.

Definition 8. *Given $P \subseteq P_G$ we write $\mathcal{L}(P)$ for the language over the alphabet M_G consisting of the sequences of moves of the game G underlying positions in P.*

While in $\mathcal{L}(P)$ we lose information about pointers, the structure of the alphabet M_G remains unchanged; in particular each letter has an order as it is a move from M_G.

Some β-normal IA_3^* terms define strategies σ for which $\mathcal{L}(\sigma)$ constitutes a faithful representation. This will be the case if pointers are uniquely reconstructible. To identify such terms it is important to understand when pointers can be ignored in positions over third-order arenas and when they have to be represented explicitly in some way. Due to the well-bracketing condition, pointers from answer-moves always lead to the last unanswered question, hence they are uniquely determined by the underlying sequence of moves. The case of questions is more complicated. Initial questions do not have pointers at all, however all non-initial ones do, which is where ambiguities might arise. Nevertheless it turns out that in the positions of interest pointers leading from first-order and second-order questions are determined uniquely, because only one unanswered enabler will occur in the respective view. Third-order questions do need pointers though, the standard example [8] being $\lambda f.f(\lambda x.f(\lambda y.x))$ and $\lambda f.f(\lambda x.f(\lambda y.y))$. The terms define the following positions respectively:

$$q_0\ q_1\ q_2\ q_1\ q_2\ q_3 \qquad\qquad q_0\ q_1\ q_2\ q_1\ q_2\ q_3\,.$$

Here pointers from third-order questions cannot be omitted, because several potential justifiers occur in the P-view. To get around the difficulties we will first focus on terms where the ambiguities for third-order questions cannot arise.

Definition 9. *A β-normal* IA_3^**-term will be called* simple *iff it can be typed without applying the contraction rule to identifiers of second-order types.*

Clearly, the two terms above are not simple.

Lemma 1. *Suppose $\Gamma \vdash M : T$ is simple and $sq_3 \in [\![\Gamma \vdash M : T]\!]$. Then $\ulcorner s \urcorner$ contains exactly one unanswered occurrence of an enabler of q_3.*

Consequently, the justifiers of all third-order moves in positions generated by simple terms are uniquely determined so, if σ denotes a simple term, $\mathcal{L}(\sigma)$ uniquely determines σ. In the next section we focus on defining automata accepting $\mathcal{L}(\mathsf{comp}(\sigma))$.

4 Automata for simple terms

This section presents the construction of automata recognizing the languages defined by simple terms. The construction proceeds by induction on the term structure. The only difficult case is application. We have chosen to pass through the intermediate step of linear composition to make the technical details more transparent.

4.1 Automata model

The pushdown automata we are going to use to capture simple terms are specialized deterministic visibly pushdown automata [3]. Their most important feature is the dependence of stack actions on input letters. Another important point in the following definitions is that the automata will use the stack only when reading third-order moves.

Definition 10. *A* strategy automaton *is a tuple*

$$\mathcal{A} = \langle\, Q, M_{push}, M_{pop}, M_{noop}, \Gamma, i, \delta_{push}, \delta_{pop}, \delta_{noop}, F \,\rangle$$

where Q is a finite set of states; $(M_{push}, M_{pop}, M_{noop})$ is the partition of the input alphabet into push, pop and noop (no stack change) letters; Γ is the stack alphabet; i is the initial state and $F \subseteq Q$ is the set of final states. The transitions are given by the partial functions:

$$\delta_{push} : Q \times M_{push} \dot{\to} Q \times \Gamma \quad \delta_{pop} : Q \times M_{pop} \times \Gamma \dot{\to} Q \quad \delta_{noop} : Q \times M_{noop} \dot{\to} Q.$$

Observe that while doing a push or a noop move the automaton does not look at the top symbol of the stack. We will sometimes use an arrow notation for transitions: $s \xrightarrow{a/x} s'$ for $\delta_{push}(s, a) = (s', x)$, $s \xrightarrow{a,x} s'$ for $\delta_{pop}(s, a, x) = s'$, and $s \xrightarrow{a} s'$ for $\delta_{noop}(s, a) = s'$.

The definitions of a configuration and a run of a strategy automaton are standard. A *configuration* is a word from $Q\Gamma^*$. The *initial configuration* is i (the initial state and the empty stack). The transition functions define transitions between configurations, e.g. the transition $s \xrightarrow{a/x} s'$ of the automaton gives transitions $sv \xrightarrow{a} s'xv$ for all $v \in \Gamma^*$. A *run* on a word $w = w_1 \ldots w_n$ is a sequence of configurations: $c_0 \xrightarrow{w_1} c_1 \xrightarrow{w_2} \ldots \xrightarrow{w_n} c_n$ where $c_0 = i$ is the initial configuration. A run is *accepting* if the state in c_n is from F. We write $L(\mathcal{A})$ for the set of words accepted by $\mathcal{A}$.

Since we want to represent sequences that are not necessarily positions, notably interaction sequences, we make the next definition general enough to account for both cases.

Definition 11. *Let ρ be a prefix-closed subset of sequences over a set of moves M, and let $\mathsf{comp}(\rho)$ be the subset of ρ consisting of non-empty sequences with an equal number of question- and answer-moves*[4]*. We say that a strategy automaton $\mathcal{A}$ is* proper for ρ *if the following conditions hold.*

(A1) $L(\mathcal{A}) = \mathsf{comp}(\rho)$.
(A2) *Every run of $\mathcal{A}$ corresponds to a sequence from ρ (as $\mathcal{A}$ is deterministic each run uniquely specifies the input sequence).*
(A3) *The alphabets M_{push} and M_{pop} consist of third-order questions and answers from M respectively.*

$\mathcal{A}$ *is* almost proper for ρ *if $L(\mathcal{A}) = \{\,\epsilon\,\} \cup \mathsf{comp}(\rho)$ and* **(A2)** *is satisfied.*

Remark 2. If $\mathcal{A}$ is proper or almost proper for $\rho = \mathcal{L}(\sigma)$ then thanks to **(A2)** we can then make a number of useful assumptions about its structure.

1. If there is a transition on a P-move from a state, then it is either the unique transition from this state or it is a pop transition and the other transitions are pop transitions on different stack letters. This is because strategies are deterministic and the push and noop moves do not look at the contents of the stack.
2. If the game in question is well-opened, i.e. none of its plays contains two initial moves, then $\mathcal{A}$ will never return to the initial state. Otherwise σ would contain just such a play. Hence, we can assume that the initial state does not have any incoming transitions and that it does not have any outgoing pop transitions.

Our first goal will be to model simple terms. The following remark summarizes what needs to be done.

Remark 3. Recall the linear application rule from Section 2. Whenever it is applied when typing β-normal IA_3^* terms we have $\mathsf{ord}(T) \leq 1$ and if $\mathsf{ord}(T) = 1$ then M is $\mathbf{cell}_{\mathcal{B}}$, **mkvar** or a term of the shape $f M_1 \cdots M_k$ where the order of f's type is at most 2. Consequently, the corresponding instances of composition are restricted accordingly. To sum up, the following semantic elements are needed to model β-normal simple IA_3^*-terms.

[4] Note that this coincides with the concept of a complete play when $\rho = \mathcal{L}(\sigma)$ for some strategy σ.

- A strategy for each of the constants.
- Identity strategies $\mathsf{id}_{[\![T]\!]}$ ($\mathsf{ord}(T) \leq 2$).
- Composition of $\sigma : [\![T]\!] \Rightarrow [\![T']\!]$ and $\tau : [\![T']\!] \Rightarrow [\![T'']\!]$ where $\mathsf{ord}(T) \leq 2$, $\mathsf{ord}(T') \leq 1$ and $\mathsf{ord}(T'') \leq 3$; moreover, if $\mathsf{ord}(T') = 1$ then either $\tau = [\![\mathbf{cell}_{\mathcal{B}}]\!]$, or $\tau = [\![\mathbf{mkvar}]\!]$, or $\tau = [\![\lambda x.\lambda\Gamma.f M_1 \cdots M_k x]\!]$.
- A way of modelling contraction up to order 1.

We have not included (un)currying in the list because in the games setting they amount to identities (up to the associativity of the disjoint sum).

The strategies for the constants and identities up to order 1 do not contain third-order moves and it is easy to construct finite automata (without stack) which are proper for each of them. The strategy automata for identity strategies at order 2 can be constructed using the $\dagger$ construction (to be introduced shortly) and the equality $\mathsf{id}_{A\Rightarrow B} = \mathsf{id}_A^\dagger \multimap \mathsf{id}_B$. Contraction up to order 1 can be interpreted simply by relabelling, so in the remainder of this section we concentrate on composition.

4.2 Composition

Let $\sigma : A \Rightarrow B$ and $\tau : B \Rightarrow C$. Recall that $A \Rightarrow B = !A \multimap B$ and $B \Rightarrow C = !B \multimap C$. In order to compose the strategies, one first defines $\sigma^\dagger : !A \multimap !B$ by

$$\sigma^\dagger = \{\, s \in L_{!A\multimap !B} \mid \text{for all initial } m,\ s \upharpoonright m \in \sigma \,\},$$

where $s \upharpoonright m$ stands for the subsequence of s (pointers included) whose moves are hereditarily justified by m. Then $\sigma;\tau : A \Rightarrow C$ is taken to be $\sigma^\dagger ;_{\mathrm{lin}} \tau$, where $;_{\mathrm{lin}}$ is discussed below.

The *linear composition* $\sigma ;_{\mathrm{lin}} \tau : A \multimap C$ of two strategies $\sigma : A \multimap B$ and $\tau : B \multimap C$ is defined in the following way. Let u be a sequence of moves from arenas A, B and C with justification pointers from all moves except those initial in C. The set of all such sequences will be denoted by $\mathsf{int}(A, B, C)$. Define $u \upharpoonright B, C$ to be the subsequence of u consisting of all moves from B and C (pointers between A-moves and B-moves are ignored). $u \upharpoonright A, B$ is defined analogously (pointers between B and C are then ignored). Finally, define $u \upharpoonright A, C$ to be the subsequence of u consisting of all moves from A and C, but where there was a pointer from a move $m_A \in M_A$ to an initial move $m_B \in M_B$ extend the pointer to the initial move in C which was pointed to from m_B. Then given two strategies $\sigma : A \multimap B$ and $\tau : B \multimap C$ the composite strategy $\sigma ;_{\mathrm{lin}} \tau : A \multimap C$ is defined in two steps:

$$\begin{aligned} \sigma || \tau &= \{\, u \in \mathsf{int}(A, B, C) \mid u \upharpoonright A, B \in \sigma,\ u \upharpoonright B, C \in \tau \,\}, \\ \sigma ;_{\mathrm{lin}} \tau &= \{\, u \upharpoonright A, C \mid u \in \sigma || \tau \,\}. \end{aligned}$$

Thus in order to carry out the composition of two strategies we will study separately: the dagger construction $\sigma^\dagger$, interaction sequences $\sigma||\tau$, and finally the hiding operation leading to $\sigma ;_{\mathrm{lin}} \tau$.

4.3 Dagger

Recall from Remark 3 that to model β-normal IA_3^*-terms we only need to apply $\dagger$ for $B = [\![T]\!]$ where $\mathsf{ord}(T) \leq 1$. It is possible to describe precisely what this construction does in this case; we will write q_i, a_i to refer to any ith-order question and answer from B ($i = 0, 1$). The definition of $\sigma^\dagger$ describes it as an interleaving of plays in σ but much more can be said about the way the copies of σ are intertwined thanks to the switching conditions, cf. Remark 1, controlling the play on $!A \multimap !B$. For instance, only O will be able to switch between different copies of σ and this can only happen after P plays in B. Consequently, if $\mathsf{ord}(T) = 0$ (no q_1, a_1 is available then) a new copy of σ can be started only after P plays a_0, i.e. when the previous one is completed. Thus $\sigma^\dagger$ in this case consists of iterated copies of σ. If $\mathsf{ord}(T) = 1$ then a new copy of σ can be started by O each time P plays q_1 or a_0. An old copy of σ can be revisited with a_1, which will then answer some unanswered occurrence of q_1. However, due to the bracketing condition, this will be possible only after all questions played after that q_1 have been answered, i.e. when all copies of σ opened after q_1 are completed. Thus, $\sigma^\dagger$ contains "stacked" copies of σ. Thanks to this we can then characterize $K = \{\,\epsilon\,\} \cup \mathsf{comp}(\sigma^\dagger)$ by the (infinite) recursive equation

$$K = \{\varepsilon\} \cup \bigcup \{q_0\Box q_1 K a_1 \Box \ldots q_1 K a_1 \Box a_0 K : q_0 \Box q_1 a_1 \Box \ldots q_1 a_1 \Box a_0 \in \mathsf{comp}(\sigma)\},$$

where $\Box$'s stand for (possibly empty and possibly different) segments of moves from A. Note that q_1 is always followed by a_1 in a position of σ due to switching conditions and the fact that B represents a first-order type.

Lemma 2. *Let $T' = B_k \to \cdots \to B_1 \to B_0$ be a type of order at most 1. If there exists a proper automaton $\mathcal{A}$ for $\sigma : ![\![T]\!] \multimap [\![T']\!]$ then there exists an almost proper automaton $\mathcal{A}^\dagger$ for $\sigma^\dagger$. In this automaton the questions and answers from $M_{[\![B_k]\!]}, \cdots, M_{[\![B_1]\!]}$ become push and pop letters respectively.*

Proof. We will refer to the questions and answers of $[\![B_0]\!]$ by q_0, a_0 respectively and to those from $[\![B_i]\!]$ ($i > 0$) by q_1 and a_1. Let $L = \mathsf{comp}(\sigma)$ and $K = \{\,\epsilon\,\} \cup \mathsf{comp}(\sigma^\dagger)$. Recall that K satisfies the equation given above.

Let i and f be the initial and final states of $\mathcal{A}$ respectively. As $\mathcal{A}$ is proper for σ, we can assume that there are no transitions to i (Remark 2(2.)). Because $\mathcal{A}$ accepts only well-opened plays we can assume that all the transitions to f are of the form $s \xrightarrow{a_0} f$ and there are no transitions from f. In order to define $\mathcal{A}^\dagger$ we first "merge" f with i or, more precisely, change each transition as above to $s \xrightarrow{a_0} i$ and make i the final state. This produces an automaton accepting L^* (observe that $L^* \subseteq K$). Then we make the following additional modifications:

$$\text{replace } s \xrightarrow{q_1} s' \text{ by } s \xrightarrow{q_1/s'} i \qquad \text{and} \qquad \text{replace } s' \xrightarrow{a_1} s'' \text{ by } i \xrightarrow{a_1, s'} s''.$$

The intuition behind the construction of $\mathcal{A}^\dagger$ is quite simple. When $\mathcal{A}^\dagger$ reads q_1 it goes to the initial state and stores the return state s' on the stack (the return state is the state $\mathcal{A}$ would go to after reading q_1). After this $\mathcal{A}^\dagger$ is ready to process a new copy of K. When finished with this copy it will end up in the state i. From this state it can read a_1 and at the same time the return state from the stack which will let it continue the simulation of $\mathcal{A}$. Consequently, it is not difficult to see that $\mathcal{A}^\dagger$ satisfies **(A2)**.

Next we argue that $\mathcal{A}^\dagger$ is deterministic. Because $\mathcal{A}$ was, the modifications involving a_0 could not introduce nondeterminism. Those using q_1 and a_1 might, if $\mathcal{A}$ happened to have an outgoing noop transition from i on a_1. However, since $!\llbracket T \rrbracket \multimap \llbracket T' \rrbracket$ is well-opened, by Remark 2 (2.) we can assume that this is not the case.

Finally, observe that $\mathcal{A}^\dagger$ currently accepts a superset of K. To be precise, it accepts a word from K iff both a final state is entered and the stack is empty. Thus, in order to accept by final state only, we have to make the automaton aware of whether the stack is empty. The solution is quite simple. The automaton starts with the empty stack. When it wants to put the first symbol onto the stack it actually puts this symbol with a special marker. Now, when popping, the automaton can realize that there is a special marker on the symbol being popped and learn this way that the stack becomes empty. This information will then be recorded in the state. The solution thus requires doubling the number of stack symbols (one normal copy and one marked copy) and doubling the number of states (information whether stack is empty or not is kept in the state).

Note that by **(A3)** $\mathcal{A}$ does not change the stack when reading q_1 and a_1 (which are first-order moves). In $\mathcal{A}^\dagger$ these letters become push and pop letters respectively. $\square$

4.4 Interaction sequences: $\sigma^\dagger || \tau$

The next challenge in modelling composition is to handle the interaction of two strategies. Recall from Remark 3 that in all instances of composition that we need to cover we have $B = \llbracket T \rrbracket$, where either $\mathsf{ord}(T) = 0$ or $\mathsf{ord}(T) = 1$ and $\tau = \llbracket \mathbf{cell}_\mathcal{B} \rrbracket, \llbracket \mathbf{mkvar} \rrbracket, \llbracket \lambda x.\lambda \Gamma. f M_1 \cdots M_k x \rrbracket$.

Lemma 3. *Suppose $\tau :!B \multimap C$ is as above. Let q_1, a_1 denote any first-order question and answer from B respectively (note that in $!B \multimap C$ they are second-order moves). If $\tau = \llbracket \mathbf{cell}_\mathcal{B} \rrbracket, \llbracket \mathbf{mkvar} \rrbracket$ then, in positions from τ, q_1 is always followed by a_1 and a_1 is always preceded by q_1. In the remaining case, q_1 will be followed by a third-order question from C and each third-order answer to that question will be followed immediately by a_1.*

Lemma 4. *Suppose there exist proper automata for $\sigma :!A \multimap B$ and $\tau :!B \multimap C$. If τ is as before then there exists a proper automaton $\mathcal{A}_{||}$ for $\sigma^\dagger || \tau$. Moreover, if there is a transition on a B move from a state of $\mathcal{A}_{||}$ then it is a noop transition and there is no other transition from that state.*

Proof. Let $\mathcal{A}_1$ be the almost proper automaton for $\sigma^\dagger :!A \multimap !B$ constructed in Lemma 2 and let $\mathcal{A}_2$ be proper for $\tau :!B \multimap C$. We use indices 1 and 2 to distinguish between the components of $\mathcal{A}_1$ and $\mathcal{A}_2$. The set of states and the stack alphabet of $\mathcal{A}_{||}$ will be given by

$$Q = (Q_1 \times Q_2) \cup (\{i_1\} \times Q_1 \times Q_2) \quad \text{and} \quad \Gamma = \Gamma_1 \cup \Gamma_2 \cup (\Gamma_2 \times Q_1).$$

$i = (i_1, i_2)$ and $F = F_1 \times F_2$ will be respectively the initial state and the set of final states. The alphabet of $\mathcal{A}_{||}$ will be partitioned in the following way.

$$M_{push} = (M^1_{push} - M_B) \cup M^2_{push} \qquad M_{pop} = (M^1_{pop} - M_B) \cup M^2_{pop}$$
$$M_{noop} = M^1_{noop} \cup M^2_{noop}$$

The definitions are not symmetric because first-order moves from B are push or pop letters for $\mathcal{A}_1$ and noop letters for $\mathcal{A}_2$. Note that moves from B are in M_{noop}. Finally, we define the transitions of $\mathcal{A}_{||}$ in several stages starting from those on A- and C-moves:

$$(s_1, s_2) \xrightarrow{m\Box} (s'_1, s_2) \qquad \text{if } m \in M_A \text{ and } s_1 \xrightarrow{m\Box} s'_1,$$
$$(s_1, s_2) \xrightarrow{m\Box} (s_1, s'_2) \qquad \text{if } m \in M_C \text{ and } s_2 \xrightarrow{m\Box} s'_2\ .$$

$\Box$ denotes an arbitrary stack action (push, pop or noop). Intuitively, for the letters considered above $\mathcal{A}_{||}$ just simulates the move of the appropriate component.

Next we deal with moves from B. Moves of order 0 are noop letters both for $\mathcal{A}_1$ and $\mathcal{A}_2$. So, we can simulate the transitions componentwise:

$$(s_1, s_2) \xrightarrow{m} (s'_1, s'_2) \qquad \text{if } s_1 \xrightarrow{m} s'_1,\ s_2 \xrightarrow{m} s'_2,\ m \in M_B \text{ and } \mathsf{ord}_B(m) = 0.$$

First-order moves from B are noop letters in $\mathcal{A}_2$ but push or pop letters in $\mathcal{A}_1$. We want them to be noop letters in $\mathcal{A}_{||}$, so we memorize the push operation in the state:

$$(s_1, s_2) \xrightarrow{q_1} (i_1, s, s'_2) \qquad \text{if } q_1 \in M_B,\ \mathsf{ord}_B(q_1) = 1,\ s_1 \xrightarrow{q_1/s} i_1 \text{ and } s_2 \xrightarrow{q_1} s'_2,$$
$$(i_1, s, s_2) \xrightarrow{a_1} (s'_1, s'_2) \qquad \text{if } a_1 \in M_B,\ \mathsf{ord}_B(a_1) = 1,\ i_1 \xrightarrow{a_1, s} s'_1 \text{ and } s_2 \xrightarrow{a_1} s'_2.$$

Observe that we know that the transition on q_1 in $\mathcal{A}_1$ is a push transition leading to the initial state i_1, because $\mathcal{A}_1$ comes from Lemma 2. In order for the construction to work the information recorded in the state has to be exploited by the automaton in future steps. By Lemma 3, q_1 is always followed either by a_1 or by a third-order question from C. The above transitions take care of the first case. In the second case we will arrange for the symbol to be preserved on the stack together with the symbol pushed by the third-order question. Dually, when processing third-order answers we should be ready to process the combined symbols and decompress the information back into the state to be used by the following a_1. Thus we add the following transitions

$$(i_1, s, s_2) \xrightarrow{q_3/(X,s)} (i_1, s'_2) \qquad \text{if } q_3 \in M_C \text{ and } s_2 \xrightarrow{q_3/X} s'_2,$$
$$(i_1, s_2) \xrightarrow{a_3,(X,s)} (i_1, s, s'_2) \qquad \text{if } a_3 \in M_C \text{ and } s_2 \xrightarrow{a_3, X} s'_2,$$

which complete the definition of $\mathcal{A}_{||}$. It is not difficult to verify that $\mathcal{A}_{||}$ is proper for $\sigma^\dagger||\tau$. Note that for each state (s_1, s_2) with an outgoing transition on a B-move m there is no other transition, because m is always a P-move either for $\mathcal{A}_1$ or for $\mathcal{A}_2$ and we can then appeal to Remark 2 for that automaton. $\square$

4.5 Rounding up

We are now ready to interpret the linear application rule introduced in Section 2. Assuming we have proper automata for $\sigma = [\![\Delta \vdash N : T]\!] : [\![\Delta]\!] \Rightarrow [\![T]\!]$ and $\tau = [\![\lambda x^T.\lambda\Gamma.Mx]\!] : [\![T]\!] \Rightarrow ([\![\Gamma]\!] \Rightarrow [\![T']\!])$ respectively, we would like to find an automaton $\mathcal{A}_{\text{lin}}$ which is proper for $\sigma^\dagger;_{\text{lin}}\tau = [\![\Gamma, \Delta \vdash \lambda\Gamma.MN : \Gamma \to T']\!]$. To that end it suffices to consider the automaton $\mathcal{A}_{||}$ from Lemma 4 and hide the moves from $[\![T]\!]$. Recall that by Lemma 4 if there exists a transition on a move from $[\![T]\!]$ from a state of $\mathcal{A}_{||}$ then it is a noop transition and no other transitions leave that state. Hence, the automaton for $\sigma^\dagger;_{\text{lin}}\tau$ can be obtained by "collapsing" the sequences of $[\![T]\!]$ transitions in $\mathcal{A}_{||}$. This can be done by first replacing each transition $s_0 \xrightarrow{m\square} s_1$ by $s_0 \xrightarrow{m\square} s_{k+1}$ when there is a sequence of transitions in $\mathcal{A}_{||}$ of the form:

$$s_0 \xrightarrow{m\square} s_1 \xrightarrow{m_1} s_2 \xrightarrow{m_2} \dots \xrightarrow{m_k} s_{k+1}$$

where m is not from $[\![T]\!]$, $m_1, \dots, m_k$ are from $[\![T]\!]$, and s_{k+1} does not have an outgoing transition on a move from $[\![T]\!]$ (note that k is bounded by the number of states in $\mathcal{A}_{||}$). After this it is enough to remove all the transitions on letters from $[\![T]\!]$. It is easy to see that the resulting automaton $\mathcal{A}_{\text{lin}}$ is proper for $\sigma^\dagger;_{\text{lin}}\tau$.

This completes the description of the construction of automata for simple terms. It remains to calculate the size of the resulting automata. For us the size of an automaton, denoted $|\mathcal{A}|$, will be the sum of the number of states and the number of stack symbols. We ignore the size of the alphabet because it is determined by types present in a sequent and hence is always linear in the size of the sequent. The number of transitions is always bounded by a polynomial in the size of the automaton.

The strategy automata for simple terms have been constructed from automata for base strategies using composition and contraction (λ-abstraction being the identity operation). Contraction does not increase the size of the automaton so it remains to calculate the increase due to composition. Suppose we have two automata $\mathcal{A}_\sigma$ and $\mathcal{A}_\tau$. Let Q_σ, Γ_σ (Q_τ, Γ_τ) stand for the sets of states and stack symbols of $\mathcal{A}_\sigma$ ($\mathcal{A}_\tau$). Examining the dagger construction we have that $|Q_\sigma^\dagger| = 2|Q_\sigma|$ and $|\Gamma_\sigma^\dagger| = 2(|\Gamma_\sigma| + |Q_\sigma|)$. For $\mathcal{A}_{||}$ we have $|Q_{||}| = 2|Q_\sigma^\dagger \times Q_\tau|$ and $|\Gamma_{||}| = |\Gamma_\sigma^\dagger| + |\Gamma_\tau| + |\Gamma_\tau \times Q_\sigma|$. Putting the two together and approximating both the number of states and stack symbols with $|\mathcal{A}_\sigma|$ and $|\mathcal{A}_\tau|$ we obtain: $|Q_{\text{lin}}| \leq 4|\mathcal{A}_\sigma||\mathcal{A}_\tau|$ and $|\Gamma_{\text{lin}}| \leq 5|\mathcal{A}_\sigma||\mathcal{A}_\tau|$. Thus $|\mathcal{A}_{\text{lin}}| \leq 9|\mathcal{A}_\sigma||\mathcal{A}_\tau|$ which gives us:

Lemma 5. *For every simple term $\Gamma \vdash M : T$ there exists an automaton which is proper for $[\![\Gamma \vdash M : T]\!]$ and whose size is exponential in the size of $\Gamma \vdash M : T$.*

5 Beyond simple terms

In this section we address the gap between simple terms and other β-normal IA_3^*-terms.

Lemma 6. *Any IA_3^*-term $\Gamma \vdash M : T$ in β-normal form can be obtained from a simple term $\Gamma' \vdash M' : T'$ by applications of the contraction rule for second-order identifiers followed by λ-abstractions.*

Hence, in order to account for all β-normal terms we only need to show how to interpret contraction at second order, because λ-abstraction is easy to interpret by renaming. As already noted at the end of Section 3, interpreting contraction will require an explicit representation scheme for pointers from third-order moves. Given a position sq_3 ending in a third-order move q_3 let us write $\alpha(s)$ (resp. $\alpha(s, q_3)$) for the number of open second- and third-order questions in s (resp. between q_3 and its justifier in s; if the justifier occurs immediately before q_3 then $\alpha(s, q_3) = 0$).

Definition 12. *Suppose $\sigma = [\![\Gamma \vdash M : T]\!]$, where $\Gamma \vdash M : T$ is an IA_3^*-term. The languages $\mathcal{P}(\sigma)$ and $\mathcal{P}'(\sigma)$ over $M_{[\![\Gamma \vdash T]\!]} + \{\, check, hit \,\}$ are defined as follows:*

$$\mathcal{P}(\sigma) = \{\, s\ check^{\alpha(s,q_3)}\ hit\ check^{\alpha(s)-\alpha(s,q_3)-1} \mid sq_3 \in \mathcal{L}(\sigma) \,\}$$
$$\mathcal{P}'(\sigma) = \{\, s\ check^{\alpha(s,q_3)}\ hit\ check^{\alpha(s)-\alpha(s,q_3)-1} \mid \exists s'.\ sq_3 s' \in \mathcal{L}(\mathsf{comp}(\sigma)) \,\}.$$

Note that q_3 is always a P-move, so s uniquely determines q_3. Clearly, $\mathcal{L}(\sigma) \cup \mathcal{P}(\sigma)$ represents σ faithfully in the sense that equality of representations coincides with equality of strategies. The subtlety is that we should compare only complete positions in strategies. This is why we introduce $\mathcal{P}'(\sigma)$. Using the results from the previous section, we first show how to construct automata recognizing $\mathcal{L}(\mathsf{comp}(\sigma)) \cup \mathcal{P}(\sigma)$ and $\mathcal{L}(\mathsf{comp}(\sigma)) \cup \mathcal{P}'(\sigma)$, where σ denotes a simple term. For this we will need to consider the nondeterministic version of strategy automata defined in the obvious way by allowing transition relations in place of functions.

By Lemma 1, in any position from σ the pointer from a third-order move q_3 points to the unique unanswered enabler visible in the P-view and hence is uniquely determined. Below we give a different characterization of the justifier relative to the whole position rather than to its P-view.

Lemma 7. *If $sq_3 \in [\![\Gamma \vdash M : T]\!]$, where $\Gamma \vdash M : T$ is simple, and q_3 is a third-order question then q_3's justifier in sq_3 is the last open enabler of q_3 in s.*

Lemma 8. *For any simple term $\Gamma \vdash M : T$ let $\sigma = [\![\Gamma \vdash M : T]\!]$. Then there exist a strategy automaton recognizing $\mathcal{L}(\mathsf{comp}(\sigma)) \cup \mathcal{P}(\sigma)$ and a nondeterministic strategy automaton accepting $\mathcal{L}(\mathsf{comp}(\sigma)) \cup \mathcal{P}'(\sigma)$ such that the push and pop letters are respectively questions and answers of order at least 2 and $check, hit$ are pop letters. Their sizes are exponential in the size of $\Gamma \vdash M : T$.*

Proof. By Lemma 5 there exists a proper automaton $\mathcal{A}$ for $\mathcal{L}(\sigma)$. First we modify $\mathcal{A}$ so that second-order questions are pushed on the stack when read and taken off the stack when the corresponding second-order answers are processed. Note that the resulting automaton, let us call it $\mathcal{A}'$, still accepts $\mathcal{L}(\mathsf{comp}(\sigma))$, because σ satisfies the bracketing condition. Due to the modification above, the symbols present on the stack during a run of $\mathcal{A}'$ will correspond exactly to the unanswered second- and third-order questions in the sequence of moves read by the automaton (of course in the case of second-order questions these symbols are the questions themselves).

Next we modify $\mathcal{A}'$ to recognize $\mathcal{L}(\mathsf{comp}(\sigma)) \cup \mathcal{P}(\sigma)$. We add new transitions so that when the new automaton sees a *check* letter while being in state s it enters into a special mode. If $\mathcal{A}'$ could not read a third-order question q_3 from s, the new automaton rejects immediately. Otherwise there is precisely one question q_3 that can be read from s (Remark 2 (1.)). By Lemma 7 it suffices to make the new automaton read *check* letters and pop the stack as long as the stack symbol is not an enabler of q_3. When the first one comes, the automaton should read *hit* and subsequently continue accepting *check* as long as the stack is not empty.

The construction of a nondeterministic automaton accepting $\mathcal{L}(\mathsf{comp}(\sigma)) \cup \mathcal{P}'(\sigma)$ is similar except that while reading *check* and *hit* the automaton will need to guess how to extend sq_3 to a complete position accepted by $\mathcal{A}$. For this the automaton uses a pre-calculated table of triples (s_1, x, s_2) such that there is a computation of $\mathcal{A}$ from the state s_1 with only x on the stack to the state s_2 with the empty stack. The nondeterministic automaton uses this table during the last phase to guess a possible extension of the computation of $\mathcal{A}$.

As all these modifications increase the size of the automaton only by a linear factor we obtain the complexity bound required by the lemma. $\Box$

Lemma 8 can be extended to all IA_3^*-terms in β-normal form. By Lemma 6, it suffices to be able to interpret λ-abstraction and contraction. Both can now be done by a suitable relabelling. Note that by identifying moves originating from the two distinguished copies of T in the contraction rule we do not lose information about pointers any more, because these are now represented explicitly.

Theorem 2. *For any IA_3^*-term $\Gamma \vdash M : T$ in β-normal form there exist a strategy automaton accepting $\mathcal{L}(\mathsf{comp}(\sigma)) \cup \mathcal{P}(\sigma)$ and a nondeterministic strategy automaton accepting $\mathcal{L}(\mathsf{comp}(\sigma)) \cup \mathcal{P}'(\sigma)$, where $\sigma = [\![\Gamma \vdash M : T]\!]$. Their sizes are exponential in the size of the term.*

Suppose the strategies σ_1, σ_2 denote two β-normal IA_3^*-terms. Observe that $\mathsf{comp}(\sigma_1) \subseteq \mathsf{comp}(\sigma_2)$ is equivalent to $\mathcal{L}(\mathsf{comp}(\sigma_1)) \cup \mathcal{P}'(\sigma_1) \subseteq \mathcal{L}(\mathsf{comp}(\sigma_2)) \cup \mathcal{P}(\sigma_2)$. We can verify the containment in the same way as for deterministic finite automata using complementation and intersection. Because the strategy automaton representing the rhs is deterministic, complementation does not incur an exponential increase in size. For intersection we can construct a product automaton in the obvious way because stack operations are determined by the input and, for a given input letter, will be of the same kind in both automata. From this observation and the above theorem we obtain our main result.

Corollary 1. *The problems of contextual equivalence and approximation for IA_3^* terms in β-normal form are in* EXPTIME.

6 Lower bound

We show EXPTIME-hardness of the equivalence problem for IA_3^* terms in β-normal form. This implies EXPTIME-hardness of the approximation problem. We use a reduction of the equivalence problem of nondeterministic automata on binary trees [9].

Labelled binary trees will be represented by positions of the game $[\![exp \to ((com \to com) \to com) \to com]\!]$. The sequence of moves $\mathcal{S}(t)$ corresponding to a given binary tree t is defined as follows

$$\mathcal{S}(x) = r_2\, q\, x\, d_2 \qquad \mathcal{S}(y(t_1, t_2)) = r_2\, q\, y\, r_3\, \mathcal{S}(t_1)\, d_3 r_3\, \mathcal{S}(t_2)\, d_3\, d_2$$

where x, y range over nullary and binary labels respectively. Observe that $\mathcal{S}(t)$ corresponds to a left-to-right depth-first traversal of t. Note that the term $\lambda f.f(\lambda x.x; x)$ defines complete positions of the shape $r_0 r_1 U d_1 d_0$ where $U ::= \epsilon \mid r_2 r_3 U d_3 r_3 U d_3 d_2$, i.e. $\lambda f.f(\lambda x.x; x)$ generates all possible sequences of r_i, d_i $(0 \le i \le 3)$ corresponding to trees. In order to represent a given tree automaton we can decorate the term with code that asks for node labels and prevents the positions incompatible with trees from developing into complete ones.

Lemma 9. *For any tree automaton $\mathcal{A}$ there exists a β-normal IA_3 term $M_{\mathcal{A}}$ such that $\mathsf{comp}([\![M_{\mathcal{A}}]\!]) = \{\, r_0\, r_1\, \mathcal{S}(t)\, d_1\, d_0 \mid t \in T(\mathcal{A})\,\}$, where $T(\mathcal{A})$ is the set of trees accepted by $\mathcal{A}$.*

Corollary 2. *The contextual equivalence and approximation problems for β-normal IA_3-terms are* EXPTIME*-hard. Thus the two problems for IA_3^* terms in β-normal form are* EXPTIME*-complete.*

References

1. Reynolds, J. C.: The essence of Algol. In: Algorithmic Languages. North Holland (1981) 345–372
2. Abramsky, S., Ghica, D. R., Murawski, A. S., Ong, C.-H. L.: Applying game semantics to compositional software modelling and verification. In Proc. of TACAS, LNCS **2988** (2004) 421–435
3. Alur, R., Madhusudan, P.: Visibly pushdown languages. Proc. of STOC (2004) 202–211
4. Ghica, D. R., McCusker, G.: Reasoning about Idealized Algol using regular expressions. Proc. of ICALP, LNCS **1853** (2000) 103–115
5. Ong, C.-H. L.: Observational equivalence of 3rd-order Idealized Algol is decidable. Proc. of LICS (2002) 245–256
6. Murawski, A. S.: On program equivalence in languages with ground-type references. In Proc. of LICS (2003) 108–117
7. Abramsky, S., McCusker, G.: Linearity, sharing and state: a fully abstract game semantics for Idealized Algol with active expressions. In: Algol-like languages. Birkhaüser (1997) 297–329
8. Hyland, J. M. E., Ong, C.-H. L.: On full abstraction for PCF. Information and Computation **163(2)** (2000) 285–408
9. Seidl, H.: Deciding equivalence of finite tree automata. SIAM J. Comput. **19(3)** (1990) 424–437

A Games for base types

Here are the arenas for base types. The lines indicate the enabling relation and moves at the top are initial.

com	*exp*	*var*
run	q	*read* *write*(0) $\cdots$ *write*(*max*)
done	$0 \;\cdots\; max$	$0 \;\cdots\; max$ *ok*

The games denoting base types are built on top of the arenas presented above. They have the following sets of plays:

$$
\begin{aligned}
P_{[\![com]\!]} &= \{\, \epsilon, run, run\ done \,\},\\
P_{[\![exp]\!]} &= \{\, \epsilon, q \,\} \cup \textstyle\bigcup_{i=0}^{max} \{\, q\, i \,\},\\
P_{[\![var]\!]} &= \{\, \epsilon, read \,\} \cup \textstyle\bigcup_{i=0}^{max} \{\, read\, i, write(i), write(i)\ ok \,\}.
\end{aligned}
$$

B Game constructions

$$
\begin{aligned}
M_{!A} &= M_A\\
\lambda_{!A} &= \lambda_A\\
\vdash_{!A} &= \vdash_A\\
P_{!A} &= \{\, s \in L_{!A} \mid \text{for all } m \in I_A,\ s \upharpoonright m \in P_A \,\}
\end{aligned}
$$

$$
\begin{aligned}
M_{A\times B} &= M_A + M_B\\
\lambda_{A\times B} &= [\lambda_A, \lambda_B]\\
\vdash_{A\times B} &= \vdash_A + \vdash_B\\
P_{A\times B} &= \{\, s \in L_{A\times B} \mid s \upharpoonright A \in P_A \ \wedge\ s \upharpoonright B = \epsilon \,\} \cup\\
&\ \{\, s \in L_{A\times B} \mid s \upharpoonright A = \epsilon \ \wedge\ s \upharpoonright B \in P_B \,\}
\end{aligned}
$$

$$
\begin{aligned}
M_{A\otimes B} &= M_A + M_B\\
\lambda_{A\otimes B} &= [\lambda_A, \lambda_B]\\
\vdash_{A\otimes B} &= \vdash_A + \vdash_B\\
P_{A\otimes B} &= \{\, s \in L_{A\otimes B} \mid s \upharpoonright A \in P_A \ \wedge\ s \upharpoonright B \in P_B \,\}
\end{aligned}
$$

$$
\begin{aligned}
M_{A\multimap B} &= M_A + M_B\\
\lambda_{A\multimap B} &= [\langle\, \overline{\lambda}_A, \lambda_B \,\rangle]\\
\vdash_{A\multimap B} &= \vdash_A \ + \ \vdash_B \ + \{\, (b,a) \mid b \in I_B \ \wedge\ a \in I_A \,\}\\
P_{A\multimap B} &= \{\, s \in L_{A\multimap B} \mid s \upharpoonright A \in P_A \ \wedge\ s \upharpoonright B \in P_B \,\}
\end{aligned}
$$

$\overline{\lambda}_A(m) = (x, y)$ iff $\lambda_A(m) = (\overline{x}, y)$, where $\overline{O} = P$ and $\overline{P} = O$.

C Special strategies

Below we list the strategies used to interpret the constants of IA_3^* and give the regular expressions describing the associated sets of complete positions. All other positions are prefixes of the complete ones.

$[\![\mathbf{skip}]\!] : [\![com]\!]$ $\qquad$ $run\ done$

$[\![i]\!] : [\![exp]\!]$ $\qquad$ $q\ i$

$[\![\Omega_{\mathcal{B}}]\!] : [\![\mathcal{B}]\!]$ $\qquad$ $\emptyset$

$[\![\mathbf{succ}]\!] : [\![exp]\!]_l \Rightarrow [\![exp]\!]_r$ $\qquad$ $q_r\ q_l\ \sum_{i=0}^{max} i_l\ ((i+1) \bmod max)_r$

$[\![\mathbf{pred}]\!] : [\![exp]\!]_l \Rightarrow [\![exp]\!]_r$ $\qquad$ $q_r\ q_l\ \sum_{i=0}^{max} i_l\ ((i-1) \bmod max)_r$

$[\![\mathbf{ifzero}_{\mathcal{B}}]\!] : [\![exp]\!]_l \Rightarrow [\![\mathcal{B}]\!]_0 \Rightarrow [\![\mathcal{B}]\!]_1 \Rightarrow [\![\mathcal{B}]\!]_r$

$$\sum_{q \vdash_{[\![\mathcal{B}]\!]} a} q_r\ q_l\ 0_l\ q_0\ a_0\ a_r + \sum_{q \vdash_{[\![\mathcal{B}]\!]} a} q_r\ q_l\ (\sum_{i=1}^{max} i_l)\ q_1\ a_1\ a_r$$

$[\![\mathbf{seq}_{\mathcal{B}}]\!] : [\![com]\!]_{l1} \Rightarrow [\![\mathcal{B}]\!]_{l2} \Rightarrow [\![\mathcal{B}]\!]_r$ $\qquad$ $\sum_{q \vdash_{[\![\mathcal{B}]\!]} a} q_r\ run_{l1}\ done_{l1}\ q_{l2}\ a_{l2}\ a_r$

$[\![\mathbf{deref}]\!] : [\![var]\!]_l \Rightarrow [\![exp]\!]_r$ $\qquad$ $q_r\ read_l\ \sum_{i=0}^{max} i_l\ i_r$

$[\![\mathbf{assign}]\!] : [\![var]\!]_{l1} \Rightarrow [\![exp]\!]_{l2} \Rightarrow [\![com]\!]_r$ $\qquad$ $run_r\ q_{l2}\ \sum_{i=0}^{max} i_{l2}\ write(i)_{l1}\ ok_{l1}\ done_r$

$[\![\mathbf{cell}_{\mathcal{B}}]\!] : ([\![var]\!]_{l1} \Rightarrow [\![\mathcal{B}]\!]_{l2}) \Rightarrow [\![\mathcal{B}]\!]_r$

$$\sum_{q \vdash_{[\![\mathcal{B}]\!]} a} q_r q_{l2} (read_{l1}\ 0_{l1})^* (\sum_{i=0}^{max} write(i)_{l1}\ ok_{l1} (read_{l1}\ i_{l1})^*)^* a_{l2} a_r$$

$[\![\mathbf{mkvar}]\!] : [\![exp \to com]\!]_{l1} \Rightarrow [\![exp]\!]_{l2} \Rightarrow [\![var]\!]_r$

$$read_r\ q_{l2}\ (\sum_{i=0}^{max} i_{l2}\ i_r) + \sum_{i=0}^{max} write(i)_r\ run_{l1}\ (q_{l1}\ i_{l1})^*\ done_{l1}\ ok_r$$

$[\![\mathbf{while}]\!] : [\![exp]\!]_l \Rightarrow [\![com]\!]_l \Rightarrow [\![com]\!]_r$ $\qquad$ $run_r\ q_l\ (\sum_{i=1}^{max} i_l\ run_l\ done_l q_l)^*\ 0_l\ done_r$

D Contraction at order 0 and 1

Contraction at order at most 1 is interpreted by relabelling transitions of the automaton, because moves originating from the two copies of T are now represented by the same alphabet. It is easy to see that the conditions **(A1)-(A3)** still hold but it is not obvious that relabelling preserves determinacy. Actually, it may be necessary to remove some transitions but only those that would never be used anyway. Indeed suppose we have a state with two executable transitions that violate determinacy of the automaton. Then we know by **(A2)** that before contraction we have had $sm_x \in \sigma$ and $sm_y \in \sigma$, where m_x, m_y are moves from the two different copies of $[\![T]\!]$. Because σ is deterministic m_x, m_y must be O-moves. However, this leads to a contradiction with switching rules (Remark 1) which state that only P can switch between different copies of $[\![T]\!]$.

E Identities

In this section we discuss the construction of automata for second-order identities. For types of order at most 1 the required automata have already been presented in [4]. Thus we concentrate on $\mathsf{id}_{[\![T]\!]}$ for $\mathsf{ord}(T) = 2$. We will use Lemma 2 to to simplify the description of an automaton for $\mathsf{id}_{[\![T]\!]}$.

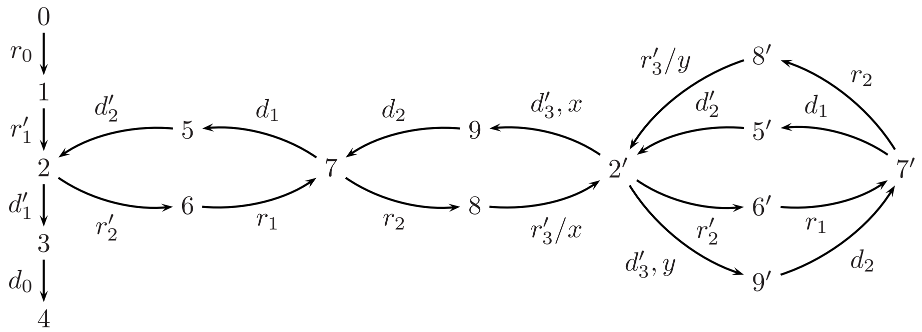
Each second-order type T has the shape $T_k \to \ldots \to T_1 \to T_0$, where $\mathsf{ord}(T_0) = 0$ and $\mathsf{ord}(T_i) \leq 1$ $(i > 0)$. Because the games $A \times B \Rightarrow C$ and $A \Rightarrow (B \Rightarrow C)$ are essentially the same, $\mathsf{id}_{[\![T]\!]}$ is the same as id_G, where $G = G' \Rightarrow [\![T_0]\!]$ and $G' = [\![T_k]\!] \times \cdots \times [\![T_1]\!]$. Because of the decomposition $A \Rightarrow B = !A \multimap B$, we have $\mathsf{id}_G = \mathsf{id}_{!(G')} \multimap \mathsf{id}_{[\![T_0]\!]}$. Further, because $\mathsf{id}_{!G'} = \mathsf{id}^\dagger_{G'}$ (Proposition 6 in [7]), we have $\mathsf{id}_G = \mathsf{id}^\dagger_{G'} \multimap \mathsf{id}_{[\![T_0]\!]}$. We deal with $\dagger$ in Subsection 4.3. Given that, it is easy to complete the construction of a proper automaton for $\mathsf{id}_{[\![T]\!]}$ due to the characterization

$$\mathsf{comp}(\sigma \multimap \mathsf{id}_{[\![T_0]\!]}) = \bigcup_{q_0 q_1 a_1 a_0 \in \mathsf{id}_{[\![T_0]\!]}} q_0 q_1 (\{\,\epsilon\,\} \cup \mathsf{comp}(\sigma)) a_1 a_0$$

and the fact that each position of $\mathsf{id}_{[\![T]\!]}$ is a prefix of some complete position. Note that if $T_0 = var$ the automaton will have to remember whether q_0 was *read* or *write* (to allow only the right a_1 later). This can done by using two copies of the almost proper automaton for $\mathsf{id}^\dagger_{G'}$.

Finally, observe that moves which have become push-letters and pop-letters during the construction of $\mathsf{id}^\dagger_{G'}$ (Lemma 2) are now third-order moves in $!T \multimap T$.

Example 2. Here is the proper automaton for $\mathsf{id}_{[\![(com \to com) \to com]\!]}$. We write m' for the moves from the left copy of $[\![(com \to com) \to com]\!]$.



F Proof of Lemma 7

The lemma is vacuously true for all base cases (no third-order moves are involved) except second-order variables. Thus we need to show that the associated identity strategy satisfies the lemma. This follows from the detailed description of the strategy we already gave, in particular the fact that $\mathsf{id}^\dagger_T$ $(\mathsf{ord}(T) \leq 1)$ is employed in the construction.

Contraction of identifiers of order 0 and 1 is easily seen to preserve the lemma (it does not affect third-order moves), so it remains to show that so does (linear) composition. For that it suffices to show that if σ satisfies the lemma then so will $\sigma^\dagger$, but this follows from the fact that $\sigma^\dagger$ is a "stack" of copies of σ.

G Proof of Lemma 9

A binary-tree automaton (BTA) $\mathcal{A}$ is a quadruple $\langle Q, \Sigma, \delta_0, \delta_2, F \rangle$, where Q is the finite set of states, $F \subseteq Q$ contains the final states, $\Sigma = \Sigma_0 + \Sigma_2$ is the input alphabet partitioned into the sets of nullary and binary symbols and $\delta_0 : \Sigma_0 \to 2^Q$, $\delta_2 : Q \times Q \times \Sigma_2 \to 2^Q$ are the transition functions. The automaton processes the tree from leaves to the root and accepts by final state. $T(\mathcal{A})$ will denote the set of trees accepted by $\mathcal{A}$.

We start by writing down the term and will explain its behaviour later. Let us define $M_{\mathcal{A}}$ to be

$$
\begin{array}{l}
\lambda y.\lambda f.\ \mathbf{new}\ \overline{RESULT} := \overline{0},\ MODE := down\ \mathbf{in} \\
\quad f(\lambda x.\ \mathbf{new}\ \overline{X_l},\ \overline{X_r},\ Z\ \mathbf{in} \\
\qquad [!MODE = down]; Z := y; \\
\qquad \mathbf{if}\ (!Z \in \Sigma_0)\ \mathbf{then}\ (\overline{RESULT} := \delta_0(!Z); MODE := up); \\
\qquad \mathbf{if}\ (!Z \in \Sigma_2)\ \mathbf{then} \\
\qquad\quad (x; [!MODE = up]; \overline{X_l} := !\overline{RESULT}; \\
\qquad\quad MODE := down; \\
\qquad\quad x; [!MODE = up]; \overline{X_r} := !\overline{RESULT}; \\
\qquad\quad \overline{RESULT} := \delta_2(!\overline{X_l}, !\overline{X_r}); MODE := up) \\
\quad); [\overline{RESULT} \cap F \neq \emptyset]
\end{array}
,
$$

where $[guard]$ stands for (**if** $guard$ **then** $skip$ **else** Ω).

The term has two arguments: y and f. IA_3^* functions call their arguments by name, hence a variable $y : exp$ can be considered as an input channel: each time a program asks for its value, it can be different. The role of f is to act as a kind of iterator. The function f takes another function as an argument and the intention is that it calls this argument in a pattern corresponding to the depth-first traversal of the hypothetical tree accepted by the automaton. In each invocation of the argument of f the value of y is checked, which corresponds to a question about the letter labelling the node. Assertions of the shape $[guard]$ along with the special variable $MODE$ cause divergence when O tries to explore positions not corresponding to a tree (e.g. when it tries to visit a child after processing a nullary symbol). The rest of the term calculates the run of the automaton.

The automaton is nondeterministic but we want to calculate its run in a deterministic way. The solution is simple: use a vector of variables to calculate all possible runs of the automaton at the same time. We have used vectors $\overline{X} = \langle X_q \rangle_{q \in Q}$ of variables to represent sets of states that the automaton can reach after processing a subtree (we assume that the value of X_q will be 0 or 1 depending on whether q is to be reachable or not; $\overline{0}$ stands for the vector of zeros). To model the behaviour of a tree automaton their values have to be passed from children-nodes to the parent node once the parent node has both results from its children. We implement this passing mechanism by a "global" vector $\overline{RESULT}$ which is set by the child node using δ_0 or δ_2 and read and saved by the parent in $\vec{X_l}, \vec{X_r}$ during the next moves. $\delta_2(!\overline{X_l}, !\overline{X_r})$ represents the extension of δ_2 to sets of states.