Saturating Automata for Game Semantics

Alex Dixon[a] Andrzej S. Murawski[b,1]

[a] *Department of Computer Science*
University of Warwick
Coventry, UK

[b] *Department of Computer Science*
University of Oxford
Oxford, UK

Abstract

Saturation is a fundamental game-semantic property satisfied by strategies that interpret higher-order concurrent programs. It states that the strategy must be closed under certain rearrangements of moves, and corresponds to the intuition that program moves (P-moves) may depend only on moves made by the environment (O-moves).

We propose an automata model over an infinite alphabet, called saturating automata, for which all accepted languages are guaranteed to satisfy a closure property mimicking saturation.

We show how to translate the finitary fragment of Idealized Concurrent Algol (FICA) into saturating automata, confirming their suitability for modelling higher-order concurrency. Moreover, we find that, for terms in normal form, the resultant automaton has linearly many transitions and states with respect to term size, and can be constructed in polynomial time. This is in contrast to earlier attempts at finding automata-theoretic models of FICA, which did not guarantee saturation and involved an exponential blow-up during translation, even for normal forms.

Keywords: automata over infinite alphabets, Finitary Idealized Concurrent Algol, game semantics, higher-order concurrency

1 Introduction

Game semantics is a versatile modelling theory that interprets computation as interaction between two players, called O (Opponent) and P (Proponent). The two players represent the environment and the program respectively, so programs can be interpreted as strategies for P. Although initially game models concerned functional sequential computation, notably the language PCF [2,11], it did not take long for the methodology to be extended to other programming constructs such as state [3,1], control operators [14], and, soon afterwards, concurrency. Some of the game models were presented in the interleaving tradition of models of concurrency [15,16,10], while others were built in the spirit of partial-order methods (true concurrency) [6].

In the interleaving approach, the aim is to construct strategies in such a way that they will contain all possible sequential observations of parallel interactions. Within game semantics, this led to the realisation that strategies must be closed under certain rearrangements of moves, to reflect the limited power of programs to observe and control the actual ordering of concurrent actions. Critically, a program can wait

[1] This research was funded in whole or in part by EPSRC EP/T006579. For the purpose of Open Access, the author has applied a CC BY public copyright licence to any Author Accepted Manuscript (AAM) version arising from this submission.

until an environment action occurs before proceeding, but it does not have any influence over environment actions or its own concurrent actions beyond those stipulated by the game. To express this constraint, one requires that strategies should be closed under certain move swaps. More specifically, consecutive $m_1 m_2$ can be swapped as long as the swap still leads to a valid play and it is *not* the case that m_1 is an O-move and m_2 is a P-move.

In game semantics, this condition first appeared in a model of Idealized CSP [15], and was named *saturation* in [10]. In game models based on event structures [20], an analogous condition can be expressed more directly using event structures with polarity [20]. Variants of saturation also occur in other contexts in the theory of concurrency. For example, they have been used to describe propagation of signals across wires in delay-insensitive circuits [23] or to specify the relationship between input and output in asynchronous systems with channels [12].

More recently, there have been attempts at defining automata-theoretic formalisms that provide support for representing plays in concurrent game semantics [8,9]. At the technical level, plays are sequences of moves connected by pointers, which poses a challenge for standard automata theory based on finite alphabets. However, an infinite alphabet is ideal for this purpose, especially if it has tree structure, so that the parent relation (link from child to parent) can provide a natural way of representing game-semantic pointers. Although the proposed formalisms were shown to accommodate the game semantics of higher-order concurrent programs, notably, that of a finitary version of Idealized Concurrent Algol (FICA) [10], they do not capture natively the saturation condition: in addition to interpretations of FICA terms (which are guaranteed to satisfy saturation), they are also capable of accepting many other languages, which need not be closed under any kind of swaps.

In contrast, in this paper, we define an automata model over infinite alphabets, called *saturating automata*, for which any accepted language is guaranteed to satisfy (a language variant of) the saturation condition. It is achieved through carefully tailored transitions, which in particular restrict the way that siblings may communicate with each other through parents, and minimise direct communication between other generations.

The new design turns out to bring another technical advantage over existing translations. Saturating automata corresponding to FICA terms in normal form have linearly many states and transitions (with respect to term size), and can be generated in at most quadratic time. This is an improvement over the exponential complexity inherent in earlier translations, which was due to either the fact that memory was modelled through control states [8] or the use of product constructions to handle parallel composition [9]. In view of the ubiquity of the saturation condition, we believe that this makes saturating automata into a point of interest in the design space of automata models, which deserves further study in connection with game semantics or other areas mentioned above.

Related work

In addition to the papers already mentioned, the combination of game semantics and automata theory over infinite alphabets appeared in research into sequential computation, e.g. to handle call-by-value computation [7], ground references [18] and objects [17]. More broadly, our results are related to encodings of higher-order computation in process calculi [22,21,4] (where the role of infinite alphabets would be played by a set of names), and to abstract machines [13]. It would also be interesting to find connections between our work and trace theory over partially commutative alphabets [5], though there the commutation relation is typically symmetric, unlike in our case.

2 Finitary Idealised Concurrent Algol (FICA)

Idealised Concurrent Algol [10] is a paradigmatic call-by-name language combining higher-order computation with imperative constructs in the style of Reynolds [19], extended to concurrency with parallel composition ($||$) and binary semaphores. We consider its finitary variant, FICA, defined over a finite datatype $\{0, \ldots, max\}$ ($max \geqslant 0$), with no recursion, but with iteration. Its types θ are generated by the grammar

$$\theta ::= \beta \mid \theta \to \theta \qquad\qquad \beta ::= \mathbf{com} \mid \mathbf{exp} \mid \mathbf{var} \mid \mathbf{sem}$$

$$\frac{}{\Gamma \vdash \mathbf{skip} : \mathbf{com}} \qquad \frac{}{\Gamma \vdash \mathbf{div}_\theta : \theta} \qquad \frac{0 \leqslant i \leqslant \mathit{max}}{\Gamma \vdash i : \mathbf{exp}} \qquad \frac{\Gamma \vdash M : \mathbf{exp}}{\Gamma \vdash \mathbf{op}(M) : \mathbf{exp}}$$

$$\frac{\Gamma \vdash M : \mathbf{com} \qquad \Gamma \vdash N : \beta}{\Gamma \vdash M; N : \beta} \qquad \frac{\Gamma \vdash M : \mathbf{com} \qquad \Gamma \vdash N : \mathbf{com}}{\Gamma \vdash M || N : \mathbf{com}}$$

$$\frac{\Gamma \vdash M : \mathbf{exp} \qquad \Gamma \vdash N_1, N_2 : \beta}{\Gamma \vdash \mathbf{if}\, M \,\mathbf{then}\, N_1 \,\mathbf{else}\, N_2 : \beta} \qquad \frac{\Gamma \vdash M : \mathbf{exp} \qquad \Gamma \vdash N : \mathbf{com}}{\Gamma \vdash \mathbf{while}\, M \,\mathbf{do}\, N : \mathbf{com}}$$

$$\frac{}{\Gamma, x : \theta \vdash x : \theta} \qquad \frac{\Gamma, x : \theta \vdash M : \theta'}{\Gamma \vdash \lambda x.M : \theta \to \theta'} \qquad \frac{\Gamma \vdash M : \theta \to \theta' \qquad \Gamma \vdash N : \theta}{\Gamma \vdash MN : \theta'}$$

$$\frac{\Gamma \vdash M : \mathbf{var} \qquad \Gamma \vdash N : \mathbf{exp}}{\Gamma \vdash M := N : \mathbf{com}} \qquad \frac{\Gamma \vdash M : \mathbf{var}}{\Gamma \vdash\, !M : \mathbf{exp}} \qquad \frac{\Gamma, x : \mathbf{var} \vdash M : \mathbf{com}, \mathbf{exp}}{\Gamma \vdash \mathbf{newvar}\, x \,\mathbf{in}\, M : \mathbf{com}, \mathbf{exp}}$$

$$\frac{\Gamma \vdash M : \mathbf{sem}}{\Gamma \vdash \mathbf{release}(M) : \mathbf{com}} \qquad \frac{\Gamma \vdash M : \mathbf{sem}}{\Gamma \vdash \mathbf{grab}(M) : \mathbf{com}} \qquad \frac{\Gamma, s : \mathbf{sem} \vdash M : \mathbf{com}, \mathbf{exp}}{\Gamma \vdash \mathbf{newsem}\, s \,\mathbf{in}\, M : \mathbf{com}, \mathbf{exp}}$$

Fig. 1. FICA typing rules

where **com** is the type of commands; **exp** that of $\{0, \ldots, \mathit{max}\}$-valued expressions; **var** that of assignable variables; and **sem** that of semaphores. The typing judgments are displayed in Figure 1. Here, **skip** and $\mathbf{div}_\theta$ are constants representing termination and divergence respectively, i ranges over $\{0, \ldots, \mathit{max}\}$, and **op** represents unary arithmetic operations, such as successor or predecessor (since we work over a finite datatype, operations of bigger arity can be defined using conditionals). Variables and semaphores can be declared locally via **newvar** and **newsem**. Variables are dereferenced using $!M$, and semaphores are manipulated using two (blocking) primitives, $\mathbf{grab}(s)$ and $\mathbf{release}(s)$, which grab and release the semaphore respectively. We assume that variables are initialised to 0 and semaphores are initially released.

In reduction rules, it will be convenient to use the syntax $\mathbf{newvar}\, x := i \,\mathbf{in}\, M$ and $\mathbf{newsem}\, x := i \,\mathbf{in}\, M$, which allows us to specify initial values more flexibly, i.e. $\mathbf{newvar}\, x \,\mathbf{in}\, M$ and $\mathbf{newsem}\, x \,\mathbf{in}\, M$ should be viewed as $\mathbf{newvar}\, x := 0 \,\mathbf{in}\, M$ and $\mathbf{newsem}\, x := 0 \,\mathbf{in}\, M$ respectively.

The operational semantics is defined using a (small-step) transition relation $\mathcal{V} \vdash M, s \longrightarrow M', s'$, where $\mathcal{V}$ is a set of variable names denoting active *memory cells* and *semaphore locks*. s, s' are states, i.e. functions $s, s' : \mathcal{V} \to \{0, \cdots, \mathit{max}\}$, and M, M' are terms. We write $s \otimes (v \mapsto i)$ for the state obtained by augmenting s with $(v \mapsto i)$, assuming $v \notin \mathsf{dom}(s)$. The basic reduction rules are given in Figure 2, where c stands for any language constant (i or **skip**) and $\widehat{\mathbf{op}} : \{0, \cdots, \mathit{max}\} \to \{0, \cdots, \mathit{max}\}$ is the function corresponding to **op**. In-context reduction is given by the schemata:

$$\frac{\mathcal{V}, v \vdash M[v/x], s \otimes (v \mapsto i) \longrightarrow M', s' \otimes (v \mapsto i') \qquad M \neq c}{\mathcal{V} \vdash \mathbf{newvar}\, x := i \,\mathbf{in}\, M, s \longrightarrow \mathbf{newvar}\, x := i' \,\mathbf{in}\, M'[x/v], s'}$$

$$\frac{\mathcal{V}, v \vdash M[v/x], s \otimes (v \mapsto i) \longrightarrow M', s' \otimes (v \mapsto i') \qquad M \neq c}{\mathcal{V} \vdash \mathbf{newsem}\, x := i \,\mathbf{in}\, M, s \longrightarrow \mathbf{newsem}\, x := i' \,\mathbf{in}\, M'[x/v], s'}$$

$$\frac{\mathcal{V} \vdash M, s \longrightarrow M', s'}{\mathcal{V} \vdash \mathcal{E}[M], s \longrightarrow \mathcal{E}[M'], s'}$$

where reduction contexts $\mathcal{E}[-]$ are produced by the grammar:

$$\begin{aligned} \mathcal{E}[-] ::= {} & [-] \mid \mathcal{E}; N \mid (\mathcal{E} \,||\, N) \mid (M \,||\, \mathcal{E}) \mid \mathcal{E}N \mid \mathbf{op}(\mathcal{E}) \mid \mathbf{if}\, \mathcal{E} \,\mathbf{then}\, N_1 \,\mathbf{else}\, N_2 \\ & \mid\, !\mathcal{E} \mid \mathcal{E} := m \mid M := \mathcal{E} \mid \mathbf{grab}(\mathcal{E}) \mid \mathbf{release}(\mathcal{E}). \end{aligned}$$

We say that a term $\vdash M : \mathbf{com}$ *may terminate*, written $M \Downarrow$, if $\emptyset \vdash \emptyset, M \longrightarrow^* \emptyset, \mathbf{skip}$.

FICA terms can be compared using a notion of *contextual (may-)equivalence*, denoted $\Gamma \vdash M_1 \cong M_2$. Two terms of the same type and with the same free variables are equivalent if they cannot be distinguished with respect to termination by any context: for all contexts $\mathcal{C}$ such that $\vdash \mathcal{C}[M_1] : \mathbf{com}$, we have $\mathcal{C}[M_1] \Downarrow$

$$\begin{array}{rlrl}
\mathcal{V} \vdash \textbf{skip}||\textbf{skip},\, s \longrightarrow \textbf{skip},\, s & & \mathcal{V} \vdash \textbf{if}\, i\, \textbf{then}\, N_1\, \textbf{else}\, N_2,\, s \longrightarrow N_1,\, s, \quad i \neq 0 & \\
\mathcal{V} \vdash \textbf{skip}; c,\, s \longrightarrow c,\, s & & \mathcal{V} \vdash \textbf{if}\, 0\, \textbf{then}\, N_1\, \textbf{else}\, N_2,\, s \longrightarrow N_2,\, s & \\
\mathcal{V} \vdash \textbf{op}(i),\, s \longrightarrow \widehat{\textbf{op}}(i),\, s & & \mathcal{V} \vdash (\lambda x.M)N,\, s \longrightarrow M[N/x],\, s & \\
\mathcal{V} \vdash \textbf{newvar}\, x := i\, \textbf{in}\, c,\, s \longrightarrow c,\, s & & \mathcal{V} \vdash\, !v,\, s \otimes (v \mapsto i) \longrightarrow i,\, s \otimes (v \mapsto i) & \\
\mathcal{V} \vdash \textbf{newsem}\, x := i\, \textbf{in}\, c,\, s \longrightarrow c,\, s & & \mathcal{V} \vdash v := i',\, s \otimes (v \mapsto i) \longrightarrow \textbf{skip},\, s \otimes (v \mapsto i') &
\end{array}$$

$$\begin{array}{r}
\mathcal{V} \vdash \textbf{grab}(v),\, s \otimes (v \mapsto 0) \longrightarrow \textbf{skip},\, s \otimes (v \mapsto 1) \\
\mathcal{V} \vdash \textbf{release}(v),\, s \otimes (v \mapsto i) \longrightarrow \textbf{skip},\, s \otimes (v \mapsto 0), \quad i \neq 0 \\
\mathcal{V} \vdash \textbf{while}\, M\, \textbf{do}\, N,\, s \longrightarrow \textbf{if}\, M\, \textbf{then}\, (N; \textbf{while}\, M\, \textbf{do}\, N)\, \textbf{else}\, \textbf{skip},\, s
\end{array}$$

Fig. 2. Reduction rules for FICA

if and only if $\mathcal{C}[M_2] \Downarrow$. Using game semantics, one can reduce $\cong$ to equality of the associated sets of complete plays (Theorem 3.5).

Example 2.1 Consider the term

$$f : \textbf{com} \to \textbf{com}, c : \textbf{com} \vdash \textbf{newvar}\, X\, \textbf{in}\, (f\ (X\ := 1)\ ||\ \textbf{if}\ !X\, \textbf{then}\ c\ \textbf{else}\ \textbf{div}_{\textbf{com}}); !X : \textbf{exp}$$

The free variable f can be viewed as representing an unknown function, to be bound to concrete code by a context. Since we work in a call-by-name setting, that function may evaluate its argument arbitrarily many times, including none. If the function does not use its argument, the value of X will always be 0 (we assume that local variables are initialised to 0) and the term will never terminate, because the right term inside $||$ will always diverge, preventing the whole term from terminating. On the other hand, as long as f evaluates its argument at least once and terminates, and the right-hand side of $||$ is scheduled after the assignment $X := 1$ (and code bound to c terminates) then the whole term will terminate too, returning 1.

In the next section we sketch the game semantics of FICA.

3 Game semantics

In this section, we briefly present the fully abstract game model for FICA from [10], which we rely on in the paper. Game semantics for FICA involves two players, called Opponent (O) and Proponent (P), and the sequences of moves made by them can be viewed as interactions between a program (P) and a surrounding context (O). The games are defined using an auxiliary concept of an arena.

Definition 3.1 An *arena* A is a triple $\langle M_A, \lambda_A, \vdash_A \rangle$ where:

- M_A is a set of *moves*;
- $\lambda_A : M_A \to \{O, P\} \times \{Q, A\}$ is a function determining for each $m \in M_A$ whether it is an *Opponent* or a *Proponent move*, and a *question* or an *answer*; we write $\lambda_A^{OP}, \lambda_A^{QA}$ for the composite of λ_A with respectively the first and second projections;
- $\vdash_A$ is a binary relation on M_A, called *enabling*, satisfying: if $m \vdash_A n$ for no m then $\lambda_A(n) = (O, Q)$, if $m \vdash_A n$ then $\lambda_A^{OP}(m) \neq \lambda_A^{OP}(n)$, and if $m \vdash_A n$ then $\lambda_A^{QA}(m) = Q$.

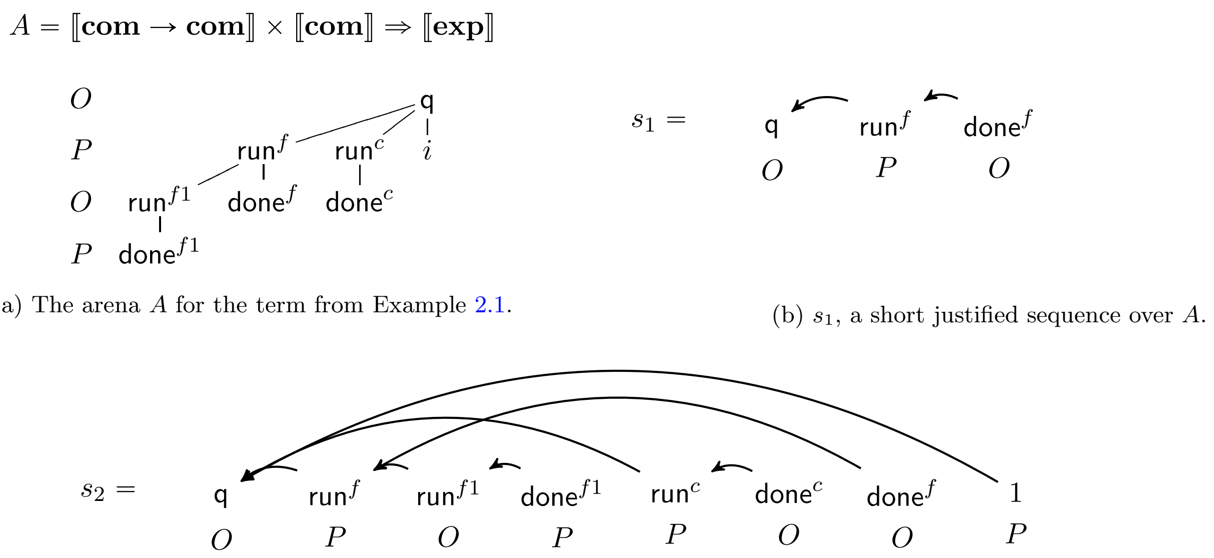
We shall write I_A for the set of all moves of A which have no enabler; such moves are called *initial.* Note that an initial move must be an O-question (OQ). In arenas used to interpret base types all questions are initial - the possible P-answers (PA) are listed below ($0 \leqslant \mathsf{i} \leqslant max$).

Arena	OQ	PA
$[\![\textbf{com}]\!]$	run	done
$[\![\textbf{var}]\!]$	read	i
	write(i)	ok

Arena	OQ	PA
$[\![\textbf{exp}]\!]$	q	i
$[\![\textbf{sem}]\!]$	grb	ok
	rls	ok

$$\begin{array}{rcl@{\qquad\qquad}rcl}
M_{A\times B} &=& M_A + M_B & M_{A\Rightarrow B} &=& M_A + M_B \\
\lambda_{A\times B} &=& [\lambda_A, \lambda_B] & \lambda_{A\Rightarrow B} &=& [\langle \lambda_A^{PO}, \lambda_A^{QA}\rangle, \lambda_B] \qquad (\lambda_A^{PO}(m) = O \text{ iff } \lambda_A^{OP}(m) = P) \\
\vdash_{A\times B} &=& \vdash_A + \vdash_B & \vdash_{A\Rightarrow B} &=& \vdash_A + \vdash_B + \{\,(b,a) \mid b \in I_B \text{ and } a \in I_A\}
\end{array}$$

Fig. 3. Arena constructions (+ and $[\cdots]$ stand for the disjoint union of sets and functions respectively; $\langle\cdots\rangle$ denotes pairing).



(a) The arena A for the term from Example 2.1.

(b) s_1, a short justified sequence over A.

(c) s_2, a longer justified sequence over A.

Fig. 4. Arenas and justified sequences

More complicated types are interpreted inductively using the *product* ($A \times B$) and *arrow* ($A \Rightarrow B$) constructions, given in Figure 3.

We write $[\![\theta]\!]$ for the arena corresponding to type θ. In Figure 4a, we give (the enabling relation of) the arena $A = ([\![\mathbf{com} \to \mathbf{com}]\!] \times [\![\mathbf{com}]\!]) \Rightarrow [\![\mathbf{exp}]\!]$, which needs to be constructed to interpret the term from Example 2.1. We use superscripts to distinguish copies of the same move (the use of superscripts is consistent with our future convention, which will be introduced in Definition 6.1).

Given an arena A, we specify next what it means to be a legal play in A. For a start, the moves that players exchange will have to form a *justified sequence*, which is a finite sequence of moves of A equipped with pointers. Its first move is always initial and has no pointer, but each subsequent move n must have a unique pointer to an earlier occurrence of a move m such that $m \vdash_A n$. We say that n is (explicitly) *justified by* m or, when n is an answer, that n *answers* m. If a question does not have an answer in a justified sequence, we say that it is *pending* in that sequence. In Figures 4b, 4c we give two justified sequences s_1 and s_2 over A.

Not all justified sequences are valid. In order to constitute a legal play, a justified sequence must satisfy a well-formedness condition that reflects the "static" style of concurrency of our programming language: any started sub-processes must end before the parent process terminates. This is formalised as follows, where the letters q and a to refer to question- and answer-moves respectively, while m denotes arbitrary moves.

Definition 3.2 The set P_A of *plays over* A consists of the justified sequences s over A that satisfy the two conditions below.

FORK : In any prefix $s' = \cdots \overset{\curvearrowleft}{q \cdots m}$ of s, the question q must be pending when m is played.

WAIT : In any prefix $s' = \cdots \overset{\curvearrowleft}{q \cdots a}$ of s, all questions justified by q must be answered.

It is easy to check that the justified sequences s_1, s_2 from Figures 4b and 4c are plays.

Remark 3.3 It is worth noting that the notion of play is stable with respect to swaps of adjacent moves except when the swaps involve occurrences of moves $m_1 m_2$ related by the pointer structure: $\overset{\curvearrowleft}{m_1 m_2}$ or m_1, m_2 are answers to questions q_1, q_2 such that q_2 justifies q_1.

A subset σ of P_A is *O-complete* if $s \in \sigma$ and $so \in P_A$ imply $so \in \sigma$, when o is an O-move.

Definition 3.4 A *strategy* on A, written $\sigma : A$, is a prefix-closed O-complete subset of P_A.

Suppose $\Gamma = \{x_1 : \theta_1, \cdots, x_l : \theta_l\}$ and $\Gamma \vdash M : \theta$ is a FICA-term. Let us write $[\![\Gamma \vdash \theta]\!]$ for the arena $[\![\theta_1]\!] \times \cdots \times [\![\theta_l]\!] \Rightarrow [\![\theta]\!]$. In [10] it is shown how to assign a strategy on $[\![\Gamma \vdash \theta]\!]$ to any FICA-term $\Gamma \vdash M : \theta$. We write $[\![\Gamma \vdash M]\!]$ to refer to that strategy. For example, $[\![\Gamma \vdash \mathbf{div}]\!] = \{\epsilon, \mathsf{run}\}$ and $[\![\Gamma \vdash \mathbf{skip}]\!] = \{\epsilon, \mathsf{run}, \overset{\curvearrowleft}{\mathsf{run}\,\mathsf{done}}\}$. The plays s_1, s_2 turn out to belong to the strategy that interprets the term from Example 2.1. Given a strategy σ, we denote by $\mathsf{comp}(\sigma)$ the set of non-empty *complete* plays of σ, i.e. those in which all questions have been answered. For example, s_1 (Figure 4b) is not complete, but s_2 (Figure 4c) is.

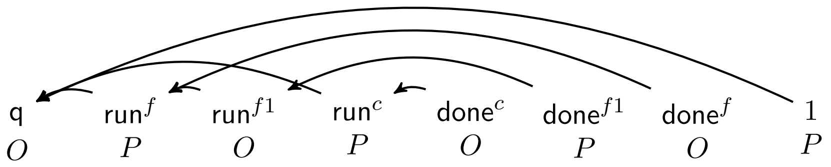
The game-semantic interpretation $[\![\cdots]\!]$ can be viewed as a faithful record of all possible interactions between the term and its contexts. It provides a fully abstract model in the sense that contextual equivalence is characterized by the sets of non-empty complete plays.

Theorem 3.5 ([10]) *We have* $\Gamma \vdash M_1 \cong M_2$ *if and only if* $\mathsf{comp}([\![\Gamma \vdash M_1]\!]) = \mathsf{comp}([\![\Gamma \vdash M_2]\!])$.

The strategies corresponding to FICA terms turn out to be closed under swaps of adjacent moves as long as the earlier move is a P-move or the later one is an O-move, and the swap produces a play. Formally, for any arena A, let us define $\succeq \subseteq P_A \times P_A$ to be the least preorder satisfying $s\,m\,o\,s' \succeq s\,o\,m\,s'$ and $s\,p\,m\,s' \succeq s\,m\,p\,s'$, where m, o, p range over moves, O-moves and P-moves respectively. In the pairs of plays above, we assume that, during a swap, the justification pointers from the two moves also move with them.

Example 3.6 Consider the following play.

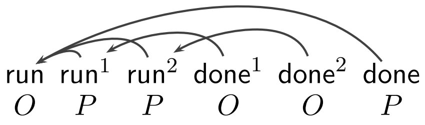
$s_3 =$



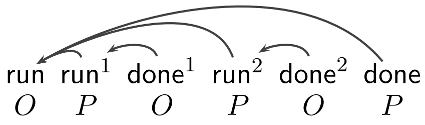
Observe that $s_2 \succeq s_3$, where s_2 is the play from Figure 4c, because the P-move done^{f1} moved to the right past a P-move (run^c) and an O-move (done^c). In contrast, we do not have $s_3 \succeq s_2$, as this would involve moving a P-move (done^{f1}) left past an O-move (done^c).

Example 3.7 Consider the plays s_4, s_5 given below (in the arena $[\![\mathbf{com} \to \mathbf{com} \to \mathbf{com}]\!]$), which correspond to parallel and sequential composition respectively. Observe that $s_4 \succeq s_5$. Note that the witnessing swap involves swapping run^2 (P-move) with done^1 (O-move), which is permitted by the definition of $\succeq$.

$s_4 =$



$s_5 =$



Definition 3.8 A strategy $\sigma : A$ is *saturated* if, for all $s, s' \in P_A$, if $s \in \sigma$ and $s \succeq s'$ then $s' \in \sigma$.

Remark 3.9 Definition 3.8 states that saturated strategies are stable under $\succeq$. Note that $s\,o\,p\,s' \not\succeq s\,p\,o\,s'$, while other o/p combinations are allowed in $\succeq$. Thus, saturated strategies allow one to express causal dependencies of P-moves on O-moves. This aspect of strategies is captured explicitly in concurrent games based on event structures [6].

Theorem 3.10 ([10]) *For any* FICA*-term* $\Gamma \vdash M$*, the strategy* $[\![\Gamma \vdash M]\!]$ *is saturated.*

In the next section we will introduce an automata-theoretic model for representing plays. In contrast to earlier attempts, languages accepted by the automata will satisfy a language-theoretic equivalent of the saturation condition.

4 Saturating automata (SATA)

The automata to be introduced will accept the so-called data languages, i.e. languages over an alphabet of the form $\Sigma \times \mathcal{D}$, where Σ is a finite alphabet and $\mathcal{D}$ is a infinite alphabet of data values. In our case, the dataset $\mathcal{D}$ will have the structure of a countably infinite forest. This structure will be helpful when representing game semantics. In particular, it will be used to encode justification pointers and enforce the WAIT condition.

Definition 4.1 $\mathcal{D}$ is a countably infinite set equipped with a function $\mathit{pred} : \mathcal{D} \to \mathcal{D} \cup \{\perp\}$ (the *parent* function) such that the following conditions hold.

- Infinite branching: $\mathit{pred}^{-1}(\{d_\perp\})$ is infinite for any $d_\perp \in \mathcal{D} \cup \{\perp\}$.
- Well-foundedness: for any $d \in \mathcal{D}$, there exists $i \in \mathbb{N}$, called the *level of* d, such that $\mathit{pred}^{i+1}(d) = \perp$. Level-0 data values are called *roots*.

We say that $T \subseteq \mathcal{D}$ is a subtree of $\mathcal{D}$ if and only if T is closed ($\forall x \in T\colon \mathit{pred}(x) \in T \cup \{\perp\}$) and rooted ($\exists! x \in T\colon \mathit{pred}(x) = \perp$).

Example 4.2 Suppose Σ consists of moves used in Figure 4a, $\mathit{pred}(d_0) = \perp$, $\mathit{pred}(d_1) = \mathit{pred}(d_1') = d_0$ and $\mathit{pred}(d_2) = d_1$. The play s_2 (Figure 4c) can be represented by the following word over $\Sigma \times \mathcal{D}$: $(\mathsf{q}, d_0)(\mathsf{run}^f, d_1)(\mathsf{run}^{f1}, d_2)(\mathsf{done}^{f1}, d_2)(\mathsf{run}^c, d_1')(\mathsf{done}^c, d_1')(\mathsf{done}^f, d_1)(1, d_0)$.

We use subtrees of $\mathcal{D}$ to represent configurations. Their nodes will be annotated with additional information.

- Each even-level node will be annotated with a multiset of control states, and zero or more memory cells. This information will be allowed to evolve during runs. Intuitively, it represents the multiset of states of a group of processes.
- Nodes at odd levels will be labelled with single control states, which will not change.

In a single transition, the automaton will be able to add or remove leaves from its configuration using very limited information. When adding a leaf as a child of node n, only the state at n will be available. When removing a leaf, in addition to the state at the leaf, only the parent state will be accessed, if at all. The automaton will also feature ϵ-transitions, which do not modify the shape of the configuration, but can be used to update annotations at even levels, while possibly accessing memory cells at ancestor nodes.

The automata will be parameterized by k and N. The parameter k is the maximal depth of the data used by the automaton, while N is the maximal number of memory cells at any node. A memory cell will store an element from $V = \{0, \ldots, \mathit{max}\}$. The set of control states will be partitioned into sets $C^{(i)}$, for $0 \leqslant i \leqslant k$, dedicated to representing run-time information at the corresponding level i.

Definition 4.3 A *saturating automaton* (SATA) is a tuple $\mathcal{A} = \langle \Sigma, k, N, C, \delta \rangle$, where:

- $\Sigma = \Sigma_{OQ} + \Sigma_{PQ} + \Sigma_{OA} + \Sigma_{PA}$ is a finite alphabet, partitioned into O/P-questions and O/P-answers (we use q_O, q_P, a_O, a_P respectively to range over the elements of the four components);
- $k \geqslant 0$ is the depth parameter and $N \geqslant 0$ is the local memory capacity;
- $C = \Sigma_{i=0}^{k} C^{(i)}$ is a finite set of *control states*, partitioned into sets $C^{(i)}$ of level-i control states;
- transitions in δ are partitioned according to their type (ADD, DEL or EPS) and level on which they operate; their shapes are listed below, where $c^{(i)}, d^{(i)}, e^{(i)} \in C^{(i)}$ and $D^{(2i)}, E^{(2i)} \in \mathfrak{M}(C^{(2i)})$, where $\mathfrak{M}(X)$ denotes the set of multisets over X.
  - · ADD($2i$) transitions have the form $c^{(2i-1)} \xrightarrow{q_O} D^{(2i)}$ or $\dagger \xrightarrow{q_O} D^{(0)}$ for the special case of $i = 0$;
  - · ADD($2i+1$) transitions have the form $c^{(2i)} \xrightarrow{q_P} d^{(2i+1)}$;
  - · DEL($2i$) transitions have the form $D^{(2i)} \xrightarrow{a_P} \dagger$;
  - · DEL($2i+1$) transitions have the form $c^{(2i+1)} \xrightarrow{a_O} d^{(2i)}$;
  - · EPS($2i$) transitions have the form $D^{(2i)} \xrightarrow{\epsilon} E^{(2i)}$;
  - · EPS($2j, 2i$) transitions read $v \in V$ from memory cell $h \in \{1, \ldots, N\}$ at level $2j \leqslant 2i$ and update it to $v' \in V$, but do not read the input: $(2j, h, v, c^{(2i)}) \xrightarrow{\epsilon} (v', d^{(2i)})$.

Remark 4.4 The ADD($2i$) transitions map exactly onto O-questions from the game semantics. We may view them as spawning a finite number of jobs (hence the use of multisets to represent those jobs' states). Dually, the DEL($2i$) transition maps onto P-answers which answer those O-questions; correspondingly with WAIT, the DEL($2i$) transition is only firable when all jobs have reached their "terminal conditions". Each job created via ADD($2i$) can evolve separately via ADD($2i+1$) or DEL($2i+1$), by EPS($2j, 2i$) (internal state change plus memory operation), or as part of a group via EPS($2i$).

Definition 4.5 A SATA *configuration* is a tuple (D, E, f, m), where D is a finite subset of $\mathcal{D}$ (consisting of data values that have been encountered so far), E is a finite subtree of $\mathcal{D}$ (the shape of the configuration), $f : E \to \sum_{0<2i-1\leqslant k} C^{(2i-1)} + \sum_{0\leqslant 2i\leqslant k} \mathfrak{M}(C^{(2i)})$ is such that

- if d is a level-$2i$ data value then $f(d) \in \mathfrak{M}(C^{(2i)})$,
- if d is a level-$(2i-1)$ data value then $f(d) \in C^{(2i-1)}$,

and $m : E \rightharpoonup V^N$ is a partial function whose domain is the set of even-level nodes of E.

A SATA $\mathcal{A}$ starts from the empty configuration $\kappa_0 = (\varnothing, \varnothing, \varnothing, \varnothing)$ and proceeds according to its transitions δ, as detailed below. We write $\kappa = (D, E, f, m)$ and $\kappa' = (D', E', f', m')$ for the current and the successor configurations respectively.

ADD

We shall have $\kappa \xrightarrow{(t,d)} \kappa'$ provided $t \in \Sigma_{OQ} + \Sigma_{PQ}$, $d \notin D$, $pred(d) \in E$, $D' = D \cup \{d\}$, $E' = E \cup \{d\}$, and if the transition-specific constraints from the table below are satisfied[2]. We write $f[\cdots]$ to extend or update f.

t	transition	pre-condition	f'	m'
q_O	$\dagger \xrightarrow{q_O} D^{(0)}$	$D = \varnothing$	$\{d \mapsto D^{(0)}\}$	$\{d \mapsto 0^N\}$
q_O	$c^{(2i-1)} \xrightarrow{q_O} D^{(2i)}$	$f(pred(d)) = c^{(2i-1)}$	$f[d \mapsto D^{(2i)}]$	$m[d \mapsto 0^N]$
q_P	$c^{(2i)} \xrightarrow{q_P} d^{(2i+1)}$	$c^{(2i)} \in_m f(pred(d))$	$f \left[\begin{array}{c} pred(d) \mapsto f(pred(d)) \backslash_m \{c^{(2i)}\} \\ d \mapsto d^{(2i+1)} \end{array} \right]$	m

Note that, in the first two cases, memory is initialised at the new node. In the last case, $c^{(2i)}$ is removed from $f(pred(d))$, i.e. if a job starts evolving via ADD($2i+1$), it is removed from the list of current jobs.

DEL

We shall have $\kappa \xrightarrow{(t,d)} \kappa'$ provided $t \in \Sigma_{OA} + \Sigma_{PA}$, d is a leaf in E, $D' = D$, $E' = E \backslash \{d\}$, $m' = m$, and the transition-specific constraints listed below are satisfied.

t	transition	pre-condition	f'
a_O	$c^{(2i+1)} \xrightarrow{a_O} d^{(2i)}$	$f(d) = c^{(2i+1)}$	$f[pred(d) \mapsto f(pred(d)) \cup_m \{d^{(2i)}\}]$
a_P	$D^{(2i)} \xrightarrow{a_P} \dagger$	$f(d) = D^{(2i)}$	f

Note that, in the first case, the leaf will contribute a new state to the parent node. For simplicity, we do not "garbage-collect" f', since the leaf removal is already recorded via E'.

[2] Given a multiset $(X, \mu : X \to \mathbb{N})$, we write $x \in_m (X, \mu)$ to mean $\mu(x) > 0$. Given two multisets (X, μ_i) $(i = 1, 2)$, we write $(X, \mu_1) \backslash_m (X, \mu_2)$, $(X, \mu_1) \cup_m (X, \mu_2)$ to stand for (X, μ^-) and (X, μ^+) respectively, where $\mu^-(x) = \max(\mu_1(x) - \mu_2(x), 0)$ and $\mu^+(x) = \mu_1(x) + \mu_2(x)$. Similarly, $(X, \mu_1) \subseteq (X, \mu_2)$ denotes $\mu_1(x) \leqslant \mu_2(x)$ for all $x \in X$.

EPS

We shall have $\kappa \xrightarrow{\varepsilon} \kappa'$ provided $D' = D$, $E' = E$ and there exists an even-level datum d satisfying the transition-specific constraints discussed below.

- For $D^{(2i)} \xrightarrow{\epsilon} E^{(2i)}$, we require $D^{(2i)} \subseteq_m f(d)$, $f' = f[d \mapsto (f(d) \backslash_m D^{(2i)}) \cup_m E^{(2i)}]$ and $m' = m$.
- For $(2j, h, v, c^{(2i)}) \xrightarrow{\mathsf{e}} (v', d^{(2i)})$, we require $c^{(2i)} \in_m f(d)$ and $m(\mathit{pred}^{2i-2j}(d))(h) = v$, $f' = f[d \mapsto (f(d) \backslash_m \{c^{(2i)}\}) \cup_m \{d^{(2i)}\}]$ and $m' = m[\mathit{pred}^{2i-2j}(d)(h) \mapsto v']$.

Note that, in the second case, $m(\mathit{pred}^{2i-2j}(d))(h)$ refers to the hth memory cell of d's ancestor at level $2j$ and only the content of this cell may be modified by the transition.

Definition 4.6 A *trace* of a SATA $\mathcal{A}$ is a word $w \in (\Sigma \times \mathcal{D})^*$ such that $\kappa_0 \xrightarrow{l_1} \kappa_1 \ldots \kappa_{h-1} \xrightarrow{l_h} \kappa_h$, where $\kappa_0 = (\varnothing, \varnothing, \varnothing, \varnothing)$, $l_i \in \{\epsilon\} \cup (\Sigma \times \mathcal{D})$ $(1 \leqslant i \leqslant h)$ and $w = l_1 \cdots l_h$. A configuration $\kappa = (D, E, f, m)$ is *accepting* if E is empty. A trace w is accepted by $\mathcal{A}$ if there is a non-empty sequence of transitions as above with κ_h accepting. The set of traces (resp. accepted traces) of $\mathcal{A}$ is denoted by $\mathit{Tr}(\mathcal{A})$ (resp. $L(\mathcal{A})$).

It follows that each data value can occur in a trace at most twice. The first occurrence (if any) must be related to a question, whereas the second one will necessarily be an answer. The fact that answers can be read only if the corresponding node becomes a leaf is analogous to the game-semantic WAIT condition. Note that E is empty in accepting configurations. This means that in every word that is accepted, each question q_O/q_P (corresponding to leaf creation) will have a corresponding answer a_P/a_O (corresponding to leaf removal), and they will be paired up with the same data value. Such words resemble complete plays (Theorem 3.5) under the convention that a justification pointer from an answer to a question is represented by using the data value introduced by the question. Indeed, we will rely on this when representing plays in Section 6.

Example 4.7 The SATA $\mathcal{A} = \langle \Sigma, 2, 1, C, \delta \rangle$ specified below recognises plays of the FICA term from Example 2.1. It is trace- and language-equivalent to the one that would be derived by the translation given in the proof of Theorem 6.4. We have $\Sigma_{OQ} = \{\mathsf{q}, \mathsf{run}^{f1}\}$, $\Sigma_{PQ} = \{\mathsf{run}^{f}, \mathsf{run}^{c}\}$, $\Sigma_{OA} = \{\mathsf{done}^{f}, \mathsf{done}^{c}\}$ and $\Sigma_{PA} = \{\mathsf{done}^{f1}, 0, \cdots, \mathit{max}\}$, $C^{(0)} = \{l_1^{(0)}, l_2^{(0)}, r_1^{(0)}, r_2^{(0)}, r_3^{(0)}, r_4^{(0)}\}$, $C^{(1)} = \{l_1^{(1)}, r_1^{(1)}\}$, $C^{(2)} = \{l_1^{(2)}, l_2^{(2)}\}$. δ is given below.

$$\begin{array}{lllll}
\text{ADD(0), DEL(0):} & \dagger \xrightarrow{\mathsf{q}} \{l_1^{(0)}, r_1^{(0)}\} & \{l_2^{(0)}, r_4^{(0)}\} \xrightarrow{1} \dagger & & \\
\text{ADD(1), DEL(1):} & l_1^{(0)} \xrightarrow{\mathsf{run}^{f}} l_1^{(1)} & l_1^{(1)} \xrightarrow{\mathsf{done}^{f}} l_2^{(0)} & r_3^{(0)} \xrightarrow{\mathsf{run}^{c}} r_1^{(1)} & r_1^{(1)} \xrightarrow{\mathsf{done}^{c}} r_4^{(0)} \\
\text{ADD(2), DEL(2):} & l_1^{(1)} \xrightarrow{\mathsf{run}^{f1}} \{l_1^{(2)}\} & \{l_2^{(2)}\} \xrightarrow{\mathsf{done}^{f1}} \dagger & & \\
\text{EPS(0,0):} & (0,1,0,r_1^{(0)}) \xrightarrow{\varepsilon} (0, r_2^{(0)}) & (0,1,i,r_1^{(0)}) \xrightarrow{\varepsilon} (i, r_3^{(0)}) \quad (0 < i \leqslant \mathit{max}) & & \\
\text{EPS(0,2):} & (0,1,i,l_1^{(2)}) \xrightarrow{\varepsilon} (1, l_2^{(2)}) \quad (0 \leqslant i \leqslant \mathit{max}) & & &
\end{array}$$

We will now show a possible transition sequence for $\mathcal{A}$. For the sake of simplicity, data values from $\mathcal{D}$ will be subscripted with a number corresponding to their level, and superscripted with zero or more primes to distinguish within each level. Configurations are denoted as a tree of nodes, reflecting the subtree of $\mathcal{D}$ currently maintained in the automaton.

Notes at even levels $2i$ are written $d(X, m)$, where d is a level-$2i$ data value, $X \in \mathcal{M}(C^{(2i)})$ and m represents the memory values maintained at that node (in this case always a single number). Nodes at odd levels $2i-1$ have the form $d(X)$, where d is a level-$(2i-1)$ data value and $X \in C^{(2i-1)}$. The complete transition sequence is given in Figure 5. It witnesses the acceptance of a data word corresponding to the play s_2 from Figure 4c.

5 Saturation

In this section we define a language variant of saturation and show that languages traced and accepted by SATA satisfy it. $d_1, d_2 \in \mathcal{D}$ will be called *independent* if neither $d_1 = \mathit{pred}^k(d_2)$ nor $d_2 = \mathit{pred}^k(d_1)$ for $k \geqslant 0$, i.e. the data lie on different branches. Let $\Sigma_O = \Sigma_{OQ} + \Sigma_{OA}$ and $\Sigma_P = \Sigma_{PQ} + \Sigma_{PA}$.

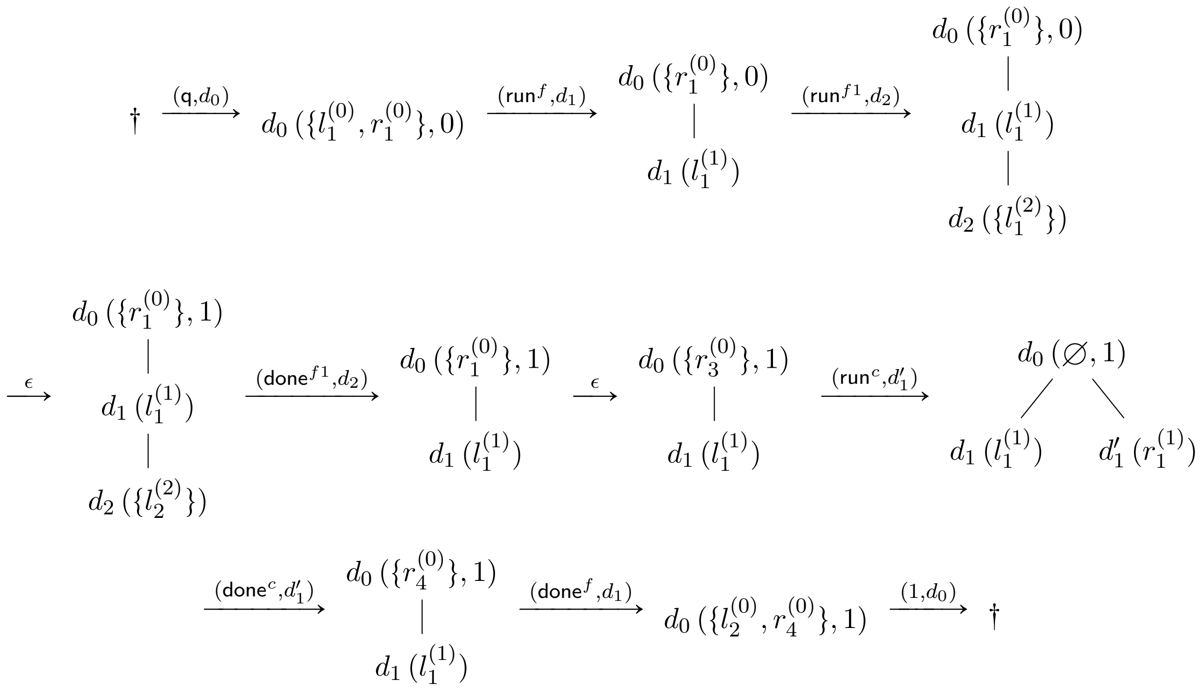


Fig. 5. A transition sequence corresponding to s_2 (Figure 4c).

Definition 5.1 We shall say that $L \subseteq (\Sigma \times \mathcal{D})^*$ is *saturated* iff, for any $w \in L$ and independent d_1, d_2, $w = w_1(t_1, d_1)(t_2, d_2)w_2 \in L$ implies $w_1(t_2, d_2)(t_1, d_1)w_2 \in L$ whenever $t_1 \in \Sigma_P$ or $t_2 \in \Sigma_O$.

Remark 5.2 The condition "$t_1 \in \Sigma_P$ or $t_2 \in \Sigma_O$" is the negation of "$t_1 \in \Sigma_O$ and $t_2 \in \Sigma_P$", i.e. the swap is allowed unless the first letter is from Σ_O and the second one from Σ_P. Note that this is analogous to the game-semantic saturation condition (Definition 3.8). The definition above uses independent d_1, d_2. It would not make sense to extend it to any dependent cases: one can show that in such cases the swap will never result in a trace (Appendix A, cf. Remark 3.3).

To show that saturating automata are bound to produce saturated sets of traces/accepted words, we establish a series of lemmas about commutativity between various kinds of transitions.

Lemma 5.3 ($\epsilon O \mapsto O\epsilon$) *If* $\kappa_1 \xrightarrow{\epsilon} \kappa_2 \xrightarrow{(t,d)} \kappa_3$ *and* $t \in \Sigma_O$ *then* $\kappa_1 \xrightarrow{(t,d)} \kappa_2' \xrightarrow{\epsilon} \kappa_3$ *for some* κ_2'.

Proof. We need to consider all combinations of the transitions listed below.

ϵ			O		
$D^{(2i)} \xrightarrow{\epsilon} E^{(2i)}$	or	$(2j, h, v, c^{(2i)}) \xrightarrow{\epsilon} (v', d^{(2i)})$	$c^{(2i'-1)} \xrightarrow{q_O} D^{(2i')}$	or	$c^{(2i'+1)} \xrightarrow{a_O} d^{(2i')}$

Observe that the EPS transitions do not modify states at odd levels or add nodes. Thus, the Σ_O transitions could be fired from κ_1. Now note that the Σ_O transitions cannot prevent the EPS transitions from being executed next, because they do not change states at even levels (though they add new ones). □

Remark 5.4 The converse to Lemma 5.3 is false. If a Σ_O transition is followed by an EPS transition, it may be impossible to swap them, because the latter could rely on states introduced by the former.

Lemma 5.5 ($P\epsilon \mapsto \epsilon P$) *If* $\kappa_1 \xrightarrow{(t,d)} \kappa_2 \xrightarrow{\epsilon} \kappa_3$ *and* $t \in \Sigma_P$ *then* $\kappa_1 \xrightarrow{\epsilon} \kappa_2' \xrightarrow{(t,d)} \kappa_3$ *for some* κ_2'.

Proof. We inspect the shape of the relevant rules, which are listed below.

P			ϵ		
$c^{(2i)} \xrightarrow{q_P} d^{(2i+1)}$	or	$D^{(2i)} \xrightarrow{a_P} \dagger$	$D^{(2i')} \xrightarrow{\epsilon} E^{(2i')}$	or	$(2j, h, v, c^{(2i')}) \xrightarrow{\epsilon} (v', d^{(2i')})$

Observe that the ϵ transitions do not depend on any information introduced by transitions on Σ_P. Hence, they are executable from κ_1. Note also that they will not destroy any information needed to execute the Σ_P transitions when fired, as there must already have been enough copies of any information to fire the transitions in the original order. □

Remark 5.6 The converse to Lemma 5.5 is false: an ϵ transition may well be followed by a transition on Σ_P that relies on the states introduced by the ϵ transition.

Remark 5.7 One can use Lemmata 5.3 and 5.5 to replace sequences of transitions of the form $\kappa \xrightarrow{(t_1,d_1)} (\xrightarrow{\epsilon})^* \xrightarrow{(t_2,d_2)} \kappa'$ with sequences of transitions between the same configurations such that the transitions on (t_1, d_1) and (t_2, d_2) will be adjacent.

- If $t_1 \in \Sigma_P$ then, using Lemma 5.5 repeatedly, one can obtain $\kappa_1 (\xrightarrow{\epsilon})^* \xrightarrow{(t_1,d_1)} \xrightarrow{(t_2,d_2)} \kappa'$.
- If $t_2 \in \Sigma_O$ then, using Lemma 5.3 this time, one can obtain $\kappa_1 \xrightarrow{(t_1,d_1)} \xrightarrow{(t_2,d_2)} (\xrightarrow{\epsilon})^* \kappa'$.

Note that these transformations require either $t_1 \in \Sigma_P$ or $t_2 \in \Sigma_O$, so they cannot be carried out if $t_1 \in \Sigma_O$ and $t_2 \in \Sigma_P$.

Next we examine permutability of consecutive transitions involving independent data values.

Lemma 5.8 *Suppose d_1, d_2 are independent and $\kappa_1 \xrightarrow{(t_1,d_1)} \kappa_2 \xrightarrow{(t_2,d_2)} \kappa_3$, where $t_1 \in \Sigma_P$ or $t_2 \in \Sigma_O$. Then there exists κ_2' such that $\kappa_1 \xrightarrow{(t_2,d_2)} \kappa_2' \xrightarrow{(t_1,d_1)} \kappa_3$.*

Proof. Recall that non-ϵ transitions rely only on two consecutive levels of the configuration tree. Consequently, if d_1, d_2 are independent and $\mathit{pred}(d_1) \neq \mathit{pred}(d_2)$ then the transitions operate on disjoint regions of the configuration and can be swapped.

Now suppose $\mathit{pred}(d_1) = \mathit{pred}(d_2)$ and note that, because of independence, we have $d_1 \neq d_2$. Consequently, the transitions must operate at the same level and concern different children of the same node.

- If the level is even, we need to consider the following combinations of transitions: ADD($2i$) ADD($2i$), DEL($2i$) ADD($2i$), DEL($2i$) DEL($2i$) (other cases can be ignored due to the $t_1 \in \Sigma_P$ or $t_2 \in \Sigma_O$ constraint). Recalling that ADD($2i$) and DEL($2i$) transitions have the form $c^{(2i-1)} \xrightarrow{q_O} D^{(2i)}$ and $D^{(2i)} \xrightarrow{a_P} \dagger$ respectively, we can confirm that the Lemma holds, because the state $c^{(2i-1)}$ associated with $\mathit{pred}(d_1) = \mathit{pred}(d_2)$ is not modified and there is no scope for interference between the transitions.
- If the level is odd, we need to consider the following combinations of transitions: ADD($2i+1$)ADD($2i+1$), ADD($2i+1$)DEL($2i+1$), DEL($2i+1$)DEL($2i+1$) (other cases can be ignored due to the $t_1 \in \Sigma_P$ or $t_2 \in \Sigma_O$ constraint). Recalling that ADD($2i+1$) and DEL($2i+1$) transitions have the form $c^{(2i)} \xrightarrow{q_P} d^{(2i+1)}$ and $c^{(2i+1)} \xrightarrow{a_O} d^{(2i)}$ respectively, we can confirm that the Lemma holds, because the transitions will not interfere. In particular, due to $d_1 \neq d_2$, the DEL($2i+1$) transition in ADD($2i+1$)DEL($2i+1$) cannot use the state introduced by the preceding ADD($2i+1$) transition.

□

Remark 5.9 Note that the "$t_1 \in \Sigma_P$ or $t_2 \in \Sigma_O$" condition is necessary: in the DEL($2i+1$) ADD($2i+1$) case (i.e. $a_O q_P$), it is possible for the latter transition to use the target state of the former.

Theorem 5.10 *For any* SATA *$\mathcal{A}$, the sets $\mathit{Tr}(\mathcal{A}), L(\mathcal{A})$ are saturated.*

Proof. Consider t_1, t_2, d_1, d_2 such that $t_1 \in \Sigma_P$ or $t_2 \in \Sigma_O$, d_1, d_2 are independent and $w_1(t_1, d_1)(t_2, d_2)w_2 \in \mathit{Tr}(\mathcal{A})$. Thus, there exist κ_1, κ_2 such that $\kappa_1 \xrightarrow{(t_1,d_1)} (\xrightarrow{\epsilon})^* \xrightarrow{(t_2,d_2)} \kappa_2$. By Remark 5.7, we can rearrange the transitions to get $\kappa_1 (\xrightarrow{\epsilon})^* \xrightarrow{(t_1,d_1)} \xrightarrow{(t_2,d_2)} (\xrightarrow{\epsilon})^* \kappa_2$. By Lemma 5.8, we then obtain $\kappa_1 (\xrightarrow{\epsilon})^* \xrightarrow{(t_2,d_2)} \xrightarrow{(t_1,d_1)} (\xrightarrow{\epsilon})^* \kappa_2$, i.e. $w_1(t_2, d_2)(t_1, d_1)w_2 \in \mathit{Tr}(\mathcal{A})$. Hence, $\mathit{Tr}(\mathcal{A})$ is saturated. As $L(\mathcal{A})$ is a subset of $\mathit{Tr}(\mathcal{A})$ in which all questions have answers, $L(\mathcal{A})$ is also saturated, because the swaps do not affect membership in $L(\mathcal{A})$. □

Remark 5.11 Earlier proposals for automata models of FICA [8,9] failed to satisfy saturation. In retrospect, this was because they allowed for too much communication between control states at various levels.

Leafy automata [8] could access the whole branch of the configuration tree at each transition and modify it during transition. In particular, each move could access and update the state at the root. This feature could easily be used to define leafy automata that are very rigid and not closed under any kind of transition swaps. Local leafy automata, also introduced in [8], restrict access only to the local part of the branch but still allow communication (thus preventing swaps) between nodes sharing a parent or great-grandparent.

Split automata [9] in turn featured restricted access to control states at various levels, but their transitions still allowed for state-based communication between siblings, through transitions $c^{(2i)} \xrightarrow{q_P} (d^{(2i)}, d^{(2i+1)})$ and $(c^{(2i)}, c^{(2i+1)}) \xrightarrow{a_O} d^{(2i)}$. The first rule could be used to create two child nodes in a specific order only, violating Lemma 5.8 for $t_1, t_2 \in \Sigma_P$. The second rule could be used to delete child nodes in a specific order only, violating the same lemma for $t_1, t_2 \in \Sigma_O$. Finally, the fact that the two rules can communicate through level $2i$ means that we can make the second one conditional on the first one, meaning that Lemma 5.8 would be violated for $t_1 \in \Sigma_P$ and $t_2 \in \Sigma_O$. Consequently, split automata did not offer native support for saturation, regardless of the polarity of letters.

6 From FICA to SATA

In this section we provide an inductive translation from FICA to SATA. The main result states that, for terms in normal form, the construction can be carried out in quadratic time and the automata have linearly many states and transitions (with respect to term size).

First, we describe how to encode justification pointers in plays using data and a special indexing scheme. Recall from Section 3 that, to interpret base types, game semantics uses moves from the set

$$\begin{aligned} \mathcal{M} &= M_{[\![\mathbf{com}]\!]} \cup M_{[\![\mathbf{exp}]\!]} \cup M_{[\![\mathbf{var}]\!]} \cup M_{[\![\mathbf{sem}]\!]} \\ &= \{\, \mathsf{run}, \mathsf{done}, \mathsf{q}, \mathsf{read}, \mathsf{grb}, \mathsf{rls}, \mathsf{ok} \,\} \cup \{\, i, \mathsf{write}(i) \mid 0 \leqslant i \leqslant \max \,\}. \end{aligned}$$

The game-semantic interpretation of a term-in-context $\Gamma \vdash M : \theta$ is a strategy over the arena $[\![\Gamma \vdash \theta]\!]$, which is obtained through product and arrow constructions, starting from arenas corresponding to base types. As both constructions rely on the disjoint sum, the moves from $[\![\Gamma \vdash \theta]\!]$ are derived from the base types present in types inside Γ and θ. To indicate the exact occurrence of a base type from which each move originates, we will annotate elements of $\mathcal{M}$ with a specially crafted scheme of superscripts. Suppose $\Gamma = \{x_1 : \theta_1, \cdots, x_l : \theta_l\}$. The superscripts will have one of the two forms, where $\boldsymbol{i} \in \mathbb{N}^*$ and $\rho \in \mathbb{N}$:

- $(\boldsymbol{i}, \rho)$ will represent moves from θ;
- $(x_v \boldsymbol{i}, \rho)$ will represent moves from θ_v $(1 \leqslant v \leqslant l)$.

The annotated moves will be written as $m^{(\boldsymbol{i},\rho)}$ or $m^{(x_v\boldsymbol{i},\rho)}$, where $m \in \mathcal{M}$. We will sometimes omit ρ on the understanding that this represents $\rho = 0$. Similarly, when $\boldsymbol{i}$ is omitted, the intended value is ϵ, e.g. m stands for $m^{(\epsilon,0)}$ and m^x for $m^{(x,0)}$. The next definition explains how the $\boldsymbol{i}$ superscripts are linked to moves from $[\![\theta]\!]$. Given $X \subseteq \{m^{(\boldsymbol{i},\rho)} \mid \boldsymbol{i} \in \mathbb{N}^*, \rho \in \mathbb{N}\}$ and $y \in \mathbb{N} \cup \{x_1, \cdots, x_l\}$, we let $yX = \{m^{(y\boldsymbol{i},\rho)} \mid m^{(\boldsymbol{i},\rho)} \in X\}$.

Definition 6.1 Given a type θ, the corresponding alphabet $\mathcal{T}_\theta$ is defined as follows

$$\begin{aligned} \mathcal{T}_\beta &= \{\, m^{(\epsilon,\rho)} \mid m \in M_{[\![\beta]\!]},\ \rho \in \mathbb{N} \,\} \qquad \beta = \mathbf{com}, \mathbf{exp}, \mathbf{var}, \mathbf{sem} \\ \mathcal{T}_{\theta_l \to \ldots \to \theta_1 \to \beta} &= \textstyle\bigcup_{u=1}^{l} (u \mathcal{T}_{\theta_u}) \cup \mathcal{T}_\beta \end{aligned}$$

For $\Gamma = \{x_1 : \theta_1, \cdots, x_l : \theta_l\}$, the alphabet $\mathcal{T}_{\Gamma \vdash \theta}$ is defined to be $\mathcal{T}_{\Gamma \vdash \theta} = \bigcup_{v=1}^{l} (x_v \mathcal{T}_{\theta_v}) \cup \mathcal{T}_\theta$.

Example 6.2 Given $\Gamma = \{f : \mathbf{com} \to \mathbf{com}, c : \mathbf{com}\}$, we have

$$\mathcal{T}_{\Gamma \vdash \mathbf{exp}} = \{\mathsf{run}^{(f1,\rho)}, \mathsf{done}^{(f1,\rho)}, \mathsf{run}^{(f,\rho)}, \mathsf{done}^{(f,\rho)}, \mathsf{run}^{(c,\rho)}, \mathsf{done}^{(c,\rho)}, \mathsf{q}^{(\epsilon,\rho)}, i^{(\epsilon,\rho)} \mid 0 \leqslant i \leqslant max,\ \rho \in \mathbb{N}\,\}.$$

Note that $\mathcal{T}_{\Gamma\vdash\theta}$ admits a natural partitioning into X-questions and X-answers ($X \in \{O, P\}$), depending on whether the underlying move is an X-question or an X-answer. To represent the game semantics of terms-in-context $\Gamma \vdash M : \theta$, we will represent plays as words over $\Sigma \times \mathcal{D}$, where Σ is a finite subset of $\mathcal{T}_{\Gamma\vdash\theta}$. Only a finite subset will be needed, because ρ will be bounded.

Next we explain how ρ and data will be used to represent justification pointers. Because no data value can be used twice with a question, occurrences of questions correspond to unique data values. A justification pointer from an answer to a question can then be represented simply by pairing up the same data value with the answer. Pointers from question-moves will be represented with the help of the index ρ. Initial question-moves do not have a pointer and to represent such questions we simply use $\rho = 0$. To represent moves with justification pointers, we will rely on ρ on the understanding that $(m^{(y,\rho)}, d)$ represents a pointer to the unique question-move that introduced $\mathit{pred}^{\rho+1}(d)$. The reader may wish to check that Example 4.2 does follow this convention (therein m^x stands for $m^{(x,0)}$). Below we give another example involving $\rho > 0$, which may arise in our translation for certain P-moves.

Example 6.3 The play $\begin{tikzcd}[column sep=small] \mathsf{q} & \mathsf{run}^f \arrow[l, bend right] & \mathsf{run}^{f1} \arrow[l, bend right] & \mathsf{run}^c \arrow[lll, bend right] \end{tikzcd}$ can be represented by $(\mathsf{run}^{(\epsilon,0)}, d_0)$ $(\mathsf{run}^{(f,0)}, d_1)$ $(\mathsf{run}^{(f1,0)}, d_2)$ $(\mathsf{run}^{(c,2)}, d_3)$, given $\mathit{pred}(d_{i+1}) = d_i$ $(0 \leqslant i \leqslant 2)$.

Below we state the main result linking FICA with saturating automata. Question-moves in this translation are handled with ADD transitions: ADD($2i$) and ADD($2i+1$) correspond to O- and P-questions respectively. Answer-moves are processed with DEL transitions: DEL($2i$) for P-answers and DEL($2i+1$) for O-answers.

Theorem 6.4 *For any* FICA *term* $\Gamma \vdash M : \theta$ *there exists a* SATA $\mathcal{A}_M$ *over a finite subset of* $\mathcal{T}_{\Gamma\vdash\theta}$ *such that the set of plays represented by words from* $Tr(\mathcal{A}_M)$ *is* $[\![\Gamma \vdash M : \theta]\!]$*, and* $L(\mathcal{A}_M)$ *represents* $\mathsf{comp}([\![\Gamma \vdash M : \theta]\!])$*. Moreover, when* M *is in* β*-normal* η*-long form,* $\mathcal{A}_M$ *has linearly many states and transitions, and can be constructed in quadratic time.*

Proof. [Sketch] It is well known that any FICA term can be reduced to an equivalent term in β-normal η-long form. The argument proceeds by induction on the structure of such forms. When referring to the inductive hypothesis for a subterm M_i, we use the subscript i to refer to the automata components, e.g. $C_i^{(j)}$, $\xrightarrow{m}_i$ etc. In contrast, $C^{(j)}$, $\xrightarrow{m}$ (without subscripts) will refer to the automaton that is being constructed. Inference lines —— indicate that the transitions listed under the line should be added to the new automaton provided the transitions listed above the line are present in the automaton obtained from the inductive hypothesis.

The following three invariants that strengthen the inductive hypothesis help us establish correctness and the requisite complexity. They concern labelled transitions only.

- **OA** (OA determinacy): if $c^{(2i+1)} \xrightarrow{a_O} d_1^{(2i)}$ and $c^{(2i+1)} \xrightarrow{a_O} d_2^{(2i)}$ then $d_1^{(2i)} = d_2^{(2i)}$.
- **PQ** (PQ pre-determinacy): if $c_1^{(2i)} \xrightarrow{q_P} d^{(2i+1)}$ and $c_2^{(2i)} \xrightarrow{q_P} d^{(2i+1)}$ then $c_1^{(2i)} = c_2^{(2i)}$.
- **FA** (final readiness): for every $D^{(0)} \xrightarrow{a_P} \dagger$, i.e. where a_P is a *final answer*, whenever the automaton reaches a configuration (D, E, f, m) with $D^{(0)} \subseteq f(r)$, where r is the root, then the transition can be executed, i.e. r has no children and $f(r) = D^{(0)}$.

Below we discuss the most interesting cases, referring the reader to Appendix B for other details.

$\mathbf{M \equiv M_1 || M_2}$:

This construction needs to interleave M_1 and M_2 while gluing the initial and final moves. Accordingly, we take $k = \max(k_1, k_2)$, $N = N_1 + N_2$, $C^{(0)} = C_1^{(0)} + C_2^{(0)} + \{\circ_1, \circ_2, \bullet_1, \bullet_2\}$, $C^{(i)} = C_1^{(i)} + C_2^{(i)}$ $(0 < i \leqslant k$, assuming $C_u^{(i)} = \varnothing$ for $i > k_u)$. All transitions from $\mathcal{A}_{M_1}$ and $\mathcal{A}_{M_2}$ other than ADD(0), DEL(0), EPS($0, 2i$) are simply embedded into the new automaton. ADD(0) and DEL(0) need to be synchronised, as shown below.

$$\frac{}{\dagger \xrightarrow{\mathsf{run}} \{\circ_1, \circ_2\}} \qquad \frac{\dagger \xrightarrow{\mathsf{run}}_u D_u^{(0)} \quad u \in \{1, 2\}}{\{\circ_u\} \xrightarrow{\epsilon} D_u^{(0)}} \qquad \frac{D_u^{(0)} \xrightarrow{\mathsf{done}}_u \dagger \quad u = 1, 2}{D_u^{(0)} \xrightarrow{\epsilon} \{\bullet_u\}} \qquad \frac{}{\{\bullet_1, \bullet_2\} \xrightarrow{\mathsf{done}} \dagger}$$

$N = N_1 + N_2$ reflects the need to combine local memories of the two automata. This need arises only at level 0, as memory at other levels will be disjoint. Consequently, we need to adjust memory indices for $\mathrm{EPS}(0, 2i)$ transitions from $\mathcal{A}_{M_2}$:

$$\frac{(0, h, v, c^{(2i)}) \xrightarrow{\epsilon}{}_{u} (v', d^{(2i)})}{(0, N_1 + h, v, c^{(2i)}) \xrightarrow{\epsilon} (v', d^{(2i)})}.$$

$\mathbf{M = f(M_1)}$:

This case is interesting, because this is where labelled transitions are created rather than inherited. We discuss the simplest instance $f : \mathbf{com} \to \mathbf{com}$. We take $k = 2 + k_1$, $N = N_1$, $C^{(0)} = \{0_{\mathsf{run}}, 0_{\mathsf{done}}\}$, $C^{(1)} = \{1_{\mathsf{run}}\}$, $C^{(j+2)} = C_1^{(j)}$ $(0 \leqslant j \leqslant k_1)$. First we add transitions corresponding to calling and returning from f:

$$\dagger \xrightarrow{\mathsf{run}^{(\epsilon,0)}} \{0_{\mathsf{run}}\} \qquad 0_{\mathsf{run}} \xrightarrow{\mathsf{run}^{(f,0)}} 1_{\mathsf{run}} \qquad 1_{\mathsf{run}} \xrightarrow{\mathsf{done}^{(f,0)}} 0_{\mathsf{done}} \qquad \{0_{\mathsf{done}}\} \xrightarrow{\mathsf{done}^{(\epsilon,0)}} \dagger.$$

In state 1_{run} we want to allow the environment to spawn an unbounded number of copies of the strategy for $\Gamma \vdash M_1 : \mathbf{com}$:

$$\frac{\dagger \xrightarrow{\mathsf{run}^{(\epsilon,0)}}{}_{1} D_1^{(0)}}{1_{\mathsf{run}} \xrightarrow{\mathsf{run}^{(f1,0)}} D_1^{(0)}} \qquad \frac{D_1^{(0)} \xrightarrow{\mathsf{done}^{(\epsilon,0)}}{}_{1} \dagger}{D_1^{(0)} \xrightarrow{\mathsf{done}^{(f1,0)}} \dagger}.$$

Note that the copies will run two levels lower than in $\mathcal{A}_{M_1}$.

The remaining moves related to M_1 originate from Γ, i.e. are of the form $m^{(x_v \boldsymbol{i},\rho)}$, where $(x_v : \theta_v) \in \Gamma$. The associated transitions need to be embedded into the new automaton, but P-question-moves of the form $m^{(x_v,\rho)}$ (corresponding to initial moves of $[\![\theta_v]\!]$) need to have their pointer adjusted so that they point at the move tagged with $\mathsf{run}^{(\epsilon,0)}$ (leaving ρ unchanged in this case would mean pointing at $\mathsf{run}^{(f1,0)}$). To achieve this, it suffices to add 2 to ρ in this case. Otherwise ρ can remain unchanged, because the pointer structure is preserved. Below we use $\square_L, \square_R$ to refer to arbitrary left/right-hand sides of transition rules.

$$\frac{\square_L \xrightarrow{m^{(x_v,\rho)}}{}_{1} \square_R \qquad m \in \Sigma_Q}{\square_L \xrightarrow{m^{(x_v,\rho+2)}} \square_R} \qquad \frac{\square_L \xrightarrow{m^{(x_v \boldsymbol{i},\rho)}}{}_{1} \square_R \qquad \boldsymbol{i} \neq \epsilon \text{ or } (\boldsymbol{i} = \epsilon \text{ and } m \in \Sigma_A)}{\square_L \xrightarrow{m^{(x_v \boldsymbol{i},\rho)}} \square_R}$$

Memory-related transitions are also copied, while adjusting the depth of the level that is being accessed by adding 2:

$$\frac{(2j, h, v, c^{(2i)}) \xrightarrow{\epsilon}{}_{1} (v', d^{(2i)})}{(2j + 2, h, v, c^{(2i)}) \xrightarrow{\epsilon} (v', d^{(2i)})}.$$

$\mathbf{M \equiv newvar\, x\, in\, M_1}$:

According to [10], it suffices to consider plays from M_1 in which $\mathsf{read}^{(x,\rho)}$ and $\mathsf{write}(j)^{(x,\rho)}$ moves are immediately followed by answers, and the sequences obey the "good variable" discipline (a value that is read corresponds to the most recently written value). To implement this recipe in an automaton, we add an extra cell at level 0 to store values of x along with explicit initialisation (to facilitate automata re-use in loops). To this end, we take $k = k_1$, $N = N_1 + 1$, $C^{(0)} = C_1^{(0)} + \{\circ, \bullet\}$, $C^{(i)} = C_1^{(i)}$ $(0 < i \leqslant k)$. All transitions from $\mathcal{A}_{M_1}$ can be copied over except $\mathrm{ADD}(0), \mathrm{DEL}(0)$ and those with superscripts of the form (x, ρ), i.e. related to x. ADD(0) and DEL(0) are handled as specified below.

$$\frac{m \in I_{[\![\beta]\!]}}{\dagger \xrightarrow{m} \{\circ\}} \qquad \frac{0 \leqslant v \leqslant max}{(0, N, v, \circ) \xrightarrow{\epsilon} (0, \bullet)} \qquad \frac{\dagger \xrightarrow{q}{}_{1} D_1^{(0)}}{\{\bullet\} \xrightarrow{\epsilon} D_1^{(0)}} \qquad \frac{D_1^{(0)} \xrightarrow{a}{}_{1} \dagger}{D_1^{(0)} \xrightarrow{a} \dagger}$$

Note that in this case $\beta = \mathbf{com}, \mathbf{exp}$, so $I_{[\![\beta]\!]} = \{\mathsf{run}\}$ or $I_{[\![\beta]\!]} = \{\mathsf{q}\}$.

For transitions related to x we proceed as follows.

$$\frac{c^{(2i)} \xrightarrow{\mathsf{write}(j)^{(x,\rho)}}_1 d^{(2i+1)} \xrightarrow{\mathsf{ok}^{(x,0)}}_1 e^{(2i)} \quad 0 \leqslant v \leqslant max}{(0, N, v, c^{(2i)}) \xrightarrow{\epsilon} (j, e^{(2i)})} \qquad \frac{c^{(2i)} \xrightarrow{\mathsf{read}^{(x,\rho)}}_1 d^{(2i+1)} \xrightarrow{j^{(x,0)}}_1 e^{(2i)}}{(0, N, j, c^{(2i)}) \xrightarrow{\epsilon} (j, e^{(2i)})}$$

Thanks to **OA**, the construction will add (at most) $max + 1$ new transitions for each transition $c^{(2i)} \xrightarrow{\mathsf{write}(j)^{(x,\rho)}}_1 d^{(2i+1)}$. Observe that they have the shape $(0, N, v, c^{(2i)}) \xrightarrow{\epsilon} (j, e^{(2i)})$ $(0 \leqslant v \leqslant max)$, and could be represented succinctly by writing $(0, N, ?, c^{(2i)}) \xrightarrow{\epsilon} (j, e^{(2i)})$, where ? is a wildcard representing an arbitrary value. So, each $c^{(2i)} \xrightarrow{\mathsf{write}(j)^{(x,\rho)}}_1 d^{(2i+1)}$ gives rise to a single transition with a wildcard. As the only modifications on EPS$(2j, 2i)$ transitions are of the kind discussed above (adding to the first two components, but never values), this representation with wildcards can be propagated in further steps. Similarly, thanks to **PQ**, each transition $d^{(2i+1)} \xrightarrow{j^{(x,0)}}_1 e^{(2i)}$ gives rise to (at most) one new transition $(0, N, j, c^{(2i)}) \xrightarrow{\epsilon} (j, e^{(2i)})$.

Complexity analysis

The constructions produce an automaton in which there are linearly many states, memory cells and transitions, with respect to term size. For states, it suffices to observe that each construction adds at most a fixed number of new states to those obtained from IH. The same applies to memory cells.

The case of transitions is harder, as there are several ways in which transitions are added to the new automaton. The easiest case is when a transition is simply copied from an automaton obtained through IH without any changes to transition labels. Other cases, represented by inference rules, are based on single premises (old transitions) and generate new single transitions. As the old ones are not included in the new automaton, such rules preserve the number of transitions. **newvar in** relies on a rule with two premises but, as discussed, the outcome could still be viewed as a single transition with a wildcard. Finally, when transitions cannot be traced back to old ones, their number is always bounded by a constant (we regard *max* as a constant too).

Hence, we can conclude that the number of transitions (possibly with wildcards) will be linear. Because each transition with a wildcard represents $max + 1$ transitions without wildcards, by instantiating them we still obtain a linear number of transitions. It is also worth noting that each transition involves at most three states: whenever sets of states are involved in transitions, they contain at most two elements.

Finally, we assess the time complexity of the constructions. A typical case consists of invoking IH and performing a bounded number of linear-time operations on the results to implement the constructions, such as retagging to implement the disjoint sum and relabelling. The combinations of transitions mentioned in **newvar** can also be considered in linear time after some preprocessing that guarantees constant-time access to incoming and outgoing transition of a given state. Overall, this could be viewed as a linear number of linear-time operations, yielding quadratic time complexity. □

7 Conclusion

We have introduced saturating automata, a new model of computation over infinite alphabets. Unlike earlier proposals [8,9], the automata accept only languages that satisfy a closure property corresponding to saturation, a property that naturally emerges in concurrent interactions between programs and their environment. Consequently, the automata can be claimed to provide a more intrinsic model of such interactions.

We also showed that saturating automata can be used to represent the game semantics of FICA, a paradigmatic language combining higher-order functions, state and concurrency. In contrast to previous translations, one does not incur an exponential penalty for using saturating automata to interpret FICA terms in normal form, which further confirms their fit with FICA.

The opportunity for further exploration of saturating automata remains, with a view to finding verification routines that can capitalise on saturation.

References

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

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

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

[4] Berger, M., K. Honda and N. Yoshida, *Sequentiality and the pi-calculus*, in: *Proceedings of TLCA*, LNCS **2044**, Springer, 2001 pp. 29–45.

[5] Cartier, P. and D. Foata, *Problèmes combinatoires de commutation et réarrangements*, Lecture Notes in Mathematics **85** (1969).
URL https://www.emis.de/journals/SLC/books/cartfoa.html

[6] Castellan, S., P. Clairambault, S. Rideau and G. Winskel, *Games and strategies as event structures*, Log. Meth. Comput. Sci. **13** (2017).

[7] Cotton-Barratt, C., A. S. Murawski and C. L. Ong, *ML, visibly pushdown class memory automata, and extended branching vector addition systems with states*, ACM Trans. Program. Lang. Syst. **41** (2019), pp. 11:1–11:38.

[8] Dixon, A., R. Lazic, A. S. Murawski and I. Walukiewicz, *Leafy automata for higher-order concurrency*, in: *Proceedings of FoSSaCS*, LNCS, 2021, pp. 184–204.
URL http://arxiv.org/abs/2101.08720

[9] Dixon, A., R. Lazic, A. S. Murawski and I. Walukiewicz, *Verifying higher-order concurrency with data automata*, in: *Proceedings of LICS*, 2021, pp. 1–13.

[10] Ghica, D. R. and A. S. Murawski, *Angelic semantics of fine-grained concurrency*, Ann. Pure Appl. Log. **151(2-3)** (2008), pp. 89–114.

[11] Hyland, J. M. E. and 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), pp. 285–408.

[12] Jifeng, H., M. B. Josephs and C. A. R. Hoare, *A theory of synchrony and asynchrony*, in: *Programming Concepts and Methods*, Elsevier, 1990 pp. 459–473.

[13] Lago, U. D., R. Tanaka and A. Yoshimizu, *The geometry of concurrent interaction: handling multiple ports by way of multiple tokens*, in: *Proceedings of LICS*, 2017, pp. 1–12.

[14] Laird, J., *Full abstraction for functional languages with control*, in: *Proceedings of 12th IEEE Symposium on Logic in Computer Science*, 1997, pp. 58–67.

[15] Laird, J., *A game semantics of Idealized CSP*, in: *Proceedings of MFPS'01*, Elsevier, 2001 pp. 1–26, ENTCS, Vol. 45.

[16] Laird, J., *Game semantics for higher-order concurrency*, in: *FSTTCS*, Lecture Notes in Computer Science **4337**, 2006, pp. 417–428.

[17] Murawski, A. S., S. J. Ramsay and N. Tzevelekos, *Game semantic analysis of equivalence in IMJ*, in: *Proceedings of ATVA*, Lecture Notes in Computer Science **9364** (2015), pp. 411–428.

[18] Murawski, A. S. and N. Tzevelekos, *Algorithmic games for full ground references*, Formal Methods Syst. Des. **52** (2018), pp. 277–314.

[19] Reynolds, J. C., *The essence of Algol*, in: J. W. de Bakker and J. van Vliet, editors, *Algorithmic Languages*, North Holland, 1978 pp. 345–372.

[20] Rideau, S. and G. Winskel, *Concurrent strategies*, in: *Proceedings of LICS'11* (2011), pp. 409–418.

[21] Röckl, C. and D. Sangiorgi, *A pi-calculus process semantics of Concurrent Idealised ALGOL*, in: *Proceedings of FoSSaCS*, Lecture Notes in Computer Science **1578** (1999), pp. 306–321.

[22] Sangiorgi, D., *Expressing mobility in process algebras: First-order and higher-order paradigms*, Technical Report CST-99-93, University of Edinburgh (1993), phD thesis.

[23] Udding, J. T., *A formal model for defining and classifying delay-insensitive circuits and systems*, Distributed Computing **1(4)** (1986), pp. 197–204.

A Additional material for Section 5

A.1 Traces are not closed under swaps for dependent values

The tree-guided question/answer discipline in SATA also implies the following technical condition about adjacent data values. Let us call $d_1, d_2 \in \mathcal{D}$ *independent* if neither $d_1 = \mathit{pred}^k(d_2)$ nor $d_2 = \mathit{pred}^k(d_1)$ for $k \geqslant 0$.

Lemma A.1 *Suppose $w = w_1(t_1, d_1)(t_2, d_2)$ is a SATA trace. Then one of the following holds: $d_1 = d_2$, $d_1 = \mathit{pred}(d_2)$, $d_2 = \mathit{pred}(d_1)$, or d_1, d_2 are independent.*

Proof. It suffices to show that $d_1 = \mathit{pred}^k(d_2)$ or $d_2 = \mathit{pred}^k(d_1)$ implies $k \in \{0, 1\}$.

Suppose $d_1 = \mathit{pred}^k(d_2)$. Then t_1 cannot be an answer, because then (t_2, d_2) would be illegal. So t_1 must be a question and d_1 is the first occurrence of d_1 inside the trace. Consequently, we must have either $d_1 = d_2$ or $d_1 = \mathit{pred}(d_2)$.

Suppose $d_2 = \mathit{pred}^k(d_1)$. Then t_2 cannot be a question, because then (t_1, d_1) would be illegal. So t_2 must be an answer. Thus, when (t_2, d_2) is played, d_2 must be a leaf, so we must have $d_1 = d_2$ or $d_2 = \mathit{pred}(d_1)$. □

Remark A.2 If d_1, d_2 are not independent, i.e. $d_1 = \mathit{pred}^k(d_2)$ or $d_2 = \mathit{pred}^k(d_1)$, then it follows from the Lemma above that if $w_1(t_1, d_1)(t_2, d_2)$ is a trace then $w_1(t_2, d_2)(t_1, d_1)$ is never a trace. We can show this by cases. If $k = 0$, i.e. $d_1 = d_2$, then t_1 must be a question and t_2 its answer, and so swapping them would not produce a trace—thus we need only consider $k = 1$. If t_1, t_2 are both questions then $\mathit{pred}(d_2) = d_1$ and so swapping the letters would cause d_2 to be added to the tree before its parent, which is not possible. If t_1, t_2 are both answers, then $\mathit{pred}(d_1) = d_2$ and so swapping the letters would cause d_2 to be removed from the tree before its child, which is not possible. In the other cases ($t_1 \in \Sigma_{\mathsf{Q}} \wedge t_2 \in \Sigma_{\mathsf{A}}$ and vice versa) w would not be a trace.

Consequently, it only makes sense to consider swapping letters relying on independent data.

B Additional material for Section 6

B.1 Proof of Theorem 6.4

We construct the corresponding automaton by induction on the structure of β-normal η-long forms. The following three invariants that strengthen the inductive hypothesis will help us establish correctness and the requisite complexity. They concern labelled transitions only.

- **OA** (OA determinacy): for any $a_O \in \Sigma_{OA}$, if $c^{(2i+1)} \xrightarrow{a_O} d_1^{(2i)}$ and $c^{(2i+1)} \xrightarrow{a_O} d_2^{(2i)}$ then $d_1^{(2i)} = d_2^{(2i)}$.
- **PQ** (PQ pre-determinacy): for any $q_P \in \Sigma_{PQ}$, if $c_1^{(2i)} \xrightarrow{q_P} d^{(2i+1)}$ and $c_2^{(2i)} \xrightarrow{q_P} d^{(2i+1)}$ then $c_1^{(2i)} = c_2^{(2i)}$.
- **FA** (final readiness): for every transition of the form $D^{(0)} \xrightarrow{a_P} \dagger$, i.e. where a_P is a *final answer*, whenever the automaton reaches a configuration (D, E, f, m) with $D^{(0)} \subseteq f(r)$, where r is the root, then the transition can be executed, i.e. r has no children and $f(r) = D^{(0)}$.

When referring to the inductive hypothesis, i.e. the automaton constructed for some subterm M_i, we will use the subscript i to refer to its components, e.g. $C_i^{(j)}$, $\xrightarrow{m}_i$ etc. In contrast, we shall use $C^{(j)}$, $\xrightarrow{m}$ to refer to the automaton that is being constructed. The construction will often use inference lines —— to indicate that the transitions listed under the line should be added to the new automaton as long as the transitions listed above the line are present in an automaton given by the inductive hypothesis.

In the first three cases, the corresponding automaton merely needs to respond to the initial question with a suitable answer or not respond at all (for $\mathbf{div}_\theta$).

- $\mathbf{M} \equiv \mathbf{skip}$: $k = 0$, $N = 0$, $C^{(0)} = \{0\}$, δ consists of $\dagger \xrightarrow{\mathsf{run}} \{0\}$ and $\{0\} \xrightarrow{\mathsf{done}} \dagger$.
- $\mathbf{M} \equiv \mathbf{i}$: $k = 0$, $N = 0$, $C^{(0)} = \{0\}$, δ consists of $\dagger \xrightarrow{\mathsf{q}} \{0\}$ and $\{0\} \xrightarrow{\mathsf{i}} \dagger$.

- $\mathbf{M} \equiv \mathbf{div}_\theta$: $k = 0$, $N = 0$, $C^{(0)} = \{0\}$. Supposing $\theta \equiv \theta_l \to \cdots \to \theta_1 \to \beta$, recall that $I_{[\![\beta]\!]}$ stands for the set of initial questions in $[\![\beta]\!]$. δ is then given by

$$\frac{x \in I_{[\![\beta]\!]}}{\dagger \xrightarrow{x} \{0\}}$$

PQ and **OA** holds vacuously, as these cases do not feature the relevant transitions. **FA** clearly holds.

- $\mathbf{M} \equiv \mathbf{op}(\mathbf{M_1})$: $k = k_1$, $N = N_1$, $C^{(j)} = C_1^{(j)}$ $(0 \leqslant j \leqslant k)$. In this case, we only need to adjust the final answers, i.e. we take all transitions for M_1 except DEL(0), and modify the DEL(0) transitions as follows.

$$\frac{D^{(0)} \xrightarrow{\mathsf{i}}_1 \dagger}{D^{(0)} \xrightarrow{\widehat{\mathbf{op}}(i)} \dagger}$$

The above relabelling does not concern transitions relevant to **OA** and **PQ**, so the properties are simply inherited from $\mathcal{A}_{M_1}$. **FA** holds by appeal to IH.

- $\mathbf{M} \equiv \mathbf{M_1} || \mathbf{M_2}$: This construction needs to interleave M_1 and M_2 while gluing the initial and final moves. Accordingly, we take $k = \max(k_1, k_2)$, $N = N_1 + N_2$, $C^{(0)} = C_1^{(0)} + C_2^{(0)} + \{\circ_1, \circ_2, \bullet_1, \bullet_2\}$, $C^{(i)} = C_1^{(i)} + C_2^{(i)}$ $(0 < i \leqslant k$, assuming $C_u^{(i)} = \varnothing$ for $i > k_u)$.

All operations other than ADD(0), DEL(0), EPS$(2j, 2i)$ can be simply embedded into the new automaton.

ADD(0) and DEL(0) need to be modified to enable synchronisation. For ADD(0), we proceed as shown below.

$$\frac{}{\dagger \xrightarrow{\mathsf{run}} \{\circ_1, \circ_2\}} \qquad \frac{\dagger \xrightarrow{\mathsf{run}}_u D_u^{(0)} \quad u \in \{1, 2\}}{\{\circ_u\} \xrightarrow{\epsilon} D_u^{(0)}}$$

For DEL(0), we first direct the constituent automata to dedicated states $\bullet_1$ and $\bullet_2$, which are then used in a single new DEL(0) rule.

$$\frac{D_u^{(0)} \xrightarrow{\mathsf{done}}_u \dagger \quad u = 1, 2}{D_u^{(0)} \xrightarrow{\epsilon} \{\bullet_u\}} \qquad \frac{}{\{\bullet_1, \bullet_2\} \xrightarrow{\mathsf{done}} \dagger}$$

It follows from IH that this will preserve **FA** for done. **OA** and **PQ** are preserved too, because the construction does not affect the relevant transitions.

$N = N_1 + N_2$ reflects the need to combine memories of the two automata. This need arises only at level 0, as memory at other levels will be disjoint. Consequently, for $j > 0$, we can simply use (disjoint) copies of EPS$(2j, 2i)$ from both automata. For $j = 0$, we arrange for M_1 to use indices from 1 to N_1 and for M_2 to use indices from $N_1 + 1$ to $N_1 + N_2$. Thus, we add N_1 when embedding EPS$(0, 2i)$ transitions from M_2:

$$\frac{(2j, h, v, c^{(2i)}) \xrightarrow{\epsilon}_u (v', d^{(2i)}) \quad j > 0}{(2j, h, v, c^{(2i)}) \xrightarrow{\epsilon} (v', d^{(2i)})} \qquad \frac{(0, h, v, c^{(2i)}) \xrightarrow{\epsilon}_u (v', d^{(2i)})}{(0, h_u, v, c^{(2i)}) \xrightarrow{\epsilon} (v', d^{(2i)})}$$

where $h_1 = h$ and $h_2 = N_1 + h$.

- $\mathbf{M} \equiv \mathbf{M_1}; \mathbf{M_2} : \mathbf{com}$

Here we need to let $\mathcal{A}_{M_1}$ run to completion and then direct the computation to $\mathcal{A}_{M_2}$. We take $k = \max(k_1, k_2)$, $N = N_1 + N_2$, $C^{(0)} = C_1^{(0)} + C_2^{(0)} + \{\circ\}$, $C^{(i)} = C_1^{(i)} + C_2^{(i)}$ $(0 < i \leqslant k)$.

We modify the ADD(0) and DEL(0) transitions as follows.

$$\frac{\dagger \xrightarrow{\mathsf{run}}_1 D^{(0)}}{\dagger \xrightarrow{\mathsf{run}} D^{(0)}} \qquad \frac{D_1^{(0)} \xrightarrow{\mathsf{done}}_1 \dagger}{D_1^{(0)} \xrightarrow{\epsilon} \{\circ\}} \qquad \frac{\dagger \xrightarrow{\mathsf{run}}_2 D_2^{(0)}}{\{\circ\} \xrightarrow{\epsilon} D_2^{(0)}} \qquad \frac{D_2^{(0)} \xrightarrow{\mathsf{done}}_2 \dagger}{D_2^{(0)} \xrightarrow{\mathsf{done}} \dagger}$$

The remaining transitions are simply copies of other transitions from $\mathcal{A}_{M_1}$, $\mathcal{A}_{M_2}$, with the proviso that in EPS$(0, 2j)$ transitions from $\mathcal{A}_{M_2}$ we add N_1 to the index of the memory cell that is accessed.

For correctness, we need to appeal to **FA** for M_1, which tells us that reaching a configuration in which the root is labelled with $D_1^{(0)}$ amounts to the termination of M_1. As before, the construction does not modify transitions relevant to **OA**, **PQ**, so the properties are simply inherited from M_1 and M_2.

- $\mathbf{M \equiv M_1; M_2 : exp}$
  This case is very similar to the previous one. The only difference is how ADD(0) and DEL(0) transitions are handled.

$$\frac{\dagger \xrightarrow{\mathsf{run}}_1 D^{(0)}}{\dagger \xrightarrow{\mathsf{q}} D^{(0)}} \qquad \frac{D_1^{(0)} \xrightarrow{\mathsf{done}}_1 \dagger}{D_1^{(0)} \xrightarrow{\epsilon} \{\circ\}} \qquad \frac{\dagger \xrightarrow{\mathsf{q}}_2 D_2^{(0)}}{\{\circ\} \xrightarrow{\epsilon} D_2^{(0)}} \qquad \frac{D_2^{(0)} \xrightarrow{\mathsf{i}}_2 \dagger}{D_2^{(0)} \xrightarrow{\mathsf{i}} \dagger}$$

- $\mathbf{M \equiv M_1; M_2 : var}$
  This case is more complicated as there are multiple initial questions and, while $\mathcal{A}_{M_1}$ is running, it is necessary to remember which question was played. We use two extra memory cells (indexed by $N_1 + N_2 + 1$ and $N_1 + N_2$ respectively) to record whether the question is read (1 in cell $N_1 + N_2 + 1$) or $\mathsf{write}(i)$ (0 in cell $N_1 + N_2 + 1$ and i in $N_1 + N_2 + 2$). States of the form $\circ_x$ ($x \in I_{[\![\mathbf{var}]\!]}$) will support this process. Once $\mathcal{A}_{M_1}$ terminates, we use states of the form $\bullet_x$ to access the information from memory before starting $\mathcal{A}_{M_2}$.
  Formally, we take $k = \max(k_1, k_2)$, $N = N_1 + N_2 + 2$, $C^{(0)} = C_1^{(0)} + C_2^{(0)} + \{\circ_x, \bullet_x \mid x \in I_{[\![\mathbf{var}]\!]}\} + \{\circ, \bullet, \bullet_w\}$, $C^{(i)} = C_1^{(i)} + C_2^{(i)}$ $(0 < i \leqslant k)$.
  ADD(0) and DEL(0) transitions are handled as detailed below, whereas other classes of transitions are copied into the automaton using the same memory-offset procedure.

$$\frac{x \in I_{[\![\mathbf{var}]\!]}}{\dagger \xrightarrow{x} \{\circ_x\}} \qquad \frac{}{(0, N_1 + N_2 + 1, 0, \circ_{\mathsf{read}}) \xrightarrow{\epsilon} (1, \circ)} \qquad \frac{}{(0, N_1 + N_2 + 2, 0, \circ_{\mathsf{write}(i)}) \xrightarrow{\epsilon} (i, \circ)}$$

$$\frac{\dagger \xrightarrow{\mathsf{run}}_1 D^{(0)}}{\{\circ\} \xrightarrow{\epsilon} D^{(0)}} \qquad \frac{D_1^{(0)} \xrightarrow{\mathsf{done}}_1 \dagger}{D_1^{(0)} \xrightarrow{\epsilon} \{\bullet\}} \qquad \frac{}{(0, N_1 + N_2 + 1, 1, \bullet) \xrightarrow{\epsilon} \bullet_{\mathsf{read}}}$$

$$\frac{}{(0, N_1 + N_2 + 1, 0, \bullet) \xrightarrow{\epsilon} \bullet_w} \qquad \frac{0 \leqslant i \leqslant max}{(0, N_1 + N_2 + 2, i, \bullet_w) \xrightarrow{\epsilon} \bullet_{\mathsf{write}(i)}}$$

$$\frac{\dagger \xrightarrow{x}_2 D_2^{(0)} \quad x \in I_{[\![\mathbf{var}]\!]}}{\{\bullet_x\} \xrightarrow{\epsilon} D_2^{(0)}} \qquad \frac{D_2^{(0)} \xrightarrow{a_P}_2 \dagger}{D_2^{(0)} \xrightarrow{a_P} \dagger}$$

Correctness and invariant preservation follow from IH, as in previous cases.

- $\mathbf{M \equiv M_1; M_2 : sem}$
  This case is simpler than the previous one, because there are only two initial moves. It can be dealt with analogously and one extra cell will suffice to remember the initial move.

- $\mathbf{M \equiv if\, M_1\, then\, M_2\, else\, M_3 : com}$
  This case is similar to $M_1; M_2$. Once $\mathcal{A}_{M_1}$ terminates, we must activate $\mathcal{A}_{M_2}$ or $\mathcal{A}_{M_3}$, depending on the label of the final transition in $\mathcal{A}_{M_1}$. Below we describe what modifications are needed for ADD(0), DEL(0) rules. We use special states $\circ_0, \circ_1$ to indicate the value of the guard M_1. Formally, $k = \max(k_1, k_2, k_3)$, $N = N_1 + \max(N_2, N_3)$, $C^{(0)} = C_1^{(0)} + C_2^{(0)} + C_3^{(0)} + \{\circ_0, \circ_1\}$, $C^{(i)} = C_1^{(i)} + C_2^{(i)} + C_3^{(i)}$ $(0 < i \leqslant k)$.

$$\frac{\dagger \xrightarrow{\mathsf{q}}_1 D_1^{(0)}}{\dagger \xrightarrow{\mathsf{run}} D_1^{(0)}} \qquad \frac{D_1^{(0)} \xrightarrow{0}_1 \dagger}{D_1^{(0)} \xrightarrow{\epsilon} \{\circ_0\}} \qquad \frac{D_1^{(0)} \xrightarrow{i}_1 \dagger \quad i > 0}{D_1^{(0)} \xrightarrow{\epsilon} \{\circ_1\}}$$

$$\frac{\dagger \xrightarrow{\mathsf{run}}_2 D_2^{(0)}}{\{\circ_1\} \xrightarrow{\epsilon} D_2^{(0)}} \qquad \frac{\dagger \xrightarrow{\mathsf{run}}_3 D_3^{(0)}}{\{\circ_0\} \xrightarrow{\epsilon} D_3^{(0)}} \qquad \frac{D_u^{(0)} \xrightarrow{\mathsf{done}}_u \dagger \quad u \in \{2, 3\}}{D_u^{(0)} \xrightarrow{\mathsf{done}} \dagger}$$

All other transitions must be copied from $\mathcal{A}_{M_1}, \mathcal{A}_{M_2}$ and, as in previous cases, memory indices in EPS$(0, 2i)$ transitions from $\mathcal{A}_{M_2}, \mathcal{A}_{M_3}$ must be increased by N_1.

The cases corresponding to $\beta = \mathbf{exp}, \mathbf{var}, \mathbf{sem}$ are also handled analogously to ;. For $\mathbf{var}, \mathbf{sem}$, it is necessary to remember the initial move.

- $\mathbf{M} \equiv \mathbf{while}\, \mathbf{M_1}\, \mathbf{do}\, \mathbf{M_2}$

  In this case we need to run $\mathcal{A}_{M_1}$ and, depending on the final move, direct it to $\mathcal{A}_{M_2}$ or terminate. When $\mathcal{A}_{M_2}$ is about to finish, we redirect to $\mathcal{A}_{M_1}$. Because the root will be deleted only at the very end, (local) memory at level 0 must be re-initialised on each iteration. This will be taken care of in the **newvar**/**newsem** cases, where we will add explicit transitions that initialise memory/semaphores as soon as the associated block starts.

  $k = \max(k_1, k_2)$, $N = N_1 + N_2$, $C^{(0)} = C_1^{(0)} + C_2^{(0)} + \{\circ, \bullet\}$, $C^{(i)} = C_1^{(i)} + C_2^{(i)}$ $(i > 0)$. As before, at level 0, we let M_1 use the left segment of memory, while M_2 will work on the right one. At other levels, the automata can use their original indexing.

  First we handle ADD(0) and DEL(0) transitions.

$$\frac{\dagger \xrightarrow{\mathsf{q}}{}_1 D_1^{(0)}}{\dagger \xrightarrow{\mathsf{run}} D_1^{(0)}} \qquad \frac{D_1^{(0)} \xrightarrow{0}{}_1 \dagger}{D_1^{(0)} \xrightarrow{\mathsf{done}} \dagger} \qquad \frac{D_1^{(0)} \xrightarrow{i}{}_1 \dagger \quad i > 0}{D_1^{(0)} \xrightarrow{\epsilon} \{\circ\}} \qquad \frac{\dagger \xrightarrow{\mathsf{run}}{}_2 D_2^{(0)}}{\{\circ\} \xrightarrow{\epsilon} D_2^{(0)}}$$

$$\frac{D_2^{(0)} \xrightarrow{\mathsf{done}}{}_2 \dagger}{D_2^{(0)} \xrightarrow{\epsilon} \{\bullet\}} \qquad \frac{\dagger \xrightarrow{\mathsf{q}} D_1^{(0)}}{\{\bullet\} \xrightarrow{\epsilon} D_1^{(0)}}$$

  All transitions from $\mathcal{A}_{M_1}$, $\mathcal{A}_{M_2}$ that are different from ADD(0) and DEL(0) are embedded into the new automaton with the proviso that, in EPS$(0, 2i)$ transitions from $\mathcal{A}_{M_2}$, we augment h by N_1.

  For correctness, we need to appeal to **FA** for M_1, M_2: they guarantee that the move to $\mathcal{A}_{M_2}$ happens only after $\mathcal{A}_{M_1}$ is definitely finished and, dually, the move to $\mathcal{A}_{M_1}$ takes place only after $\mathcal{A}_{M_2}$ is finished.

  The construction does not affect transitions relevant to **OA** and **PQ**, so they are preserved. By appealing to IH, we can conclude that **FA** is preserved too.

- $\mathbf{M} \equiv \mathbf{newvar}\, \mathbf{x}\, \mathbf{in}\, \mathbf{M_1} : \beta$

  $k = k_1$, $N = N_1 + 1$, $C^{(0)} = C_1^{(0)} + \{\circ, \bullet\}$, $C^{(i)} = C_1^{(i)}$ $(0 < i \leqslant k)$. Here we add an extra cell at level 0 to store values of x along with explicit initialisation (to facilitate automata re-use).

  All transitions from $\mathcal{A}_{M_1}$ can be copied over except ADD(0), DEL(0) and those labelled with x (for reading and writing). They will be handled as specified below. Note that in this case $\beta = \mathbf{com}, \mathbf{exp}$, so $I_{[\![\beta]\!]} = \{\mathsf{run}\}$ or $I_{[\![\beta]\!]} = \{\mathsf{q}\}$.

$$\frac{x \in I_{[\![\beta]\!]}}{\dagger \xrightarrow{x} \{\circ\}} \qquad \frac{0 \leqslant v \leqslant max}{(0, N, v, \circ) \xrightarrow{\epsilon} (0, \bullet)} \qquad \frac{\dagger \xrightarrow{x} D_1^{(0)}}{\{\bullet\} \xrightarrow{\epsilon} D_1^{(0)}} \qquad \frac{D_1^{(0)} \xrightarrow{a}{}_1 \dagger}{D_1^{(0)} \xrightarrow{a} \dagger}$$

$$\frac{c^{(2i)} \xrightarrow{\mathsf{write}(j)^{(x,\rho)}}{}_1 d^{(2i+1)} \xrightarrow{\mathsf{ok}^{(x,0)}}{}_1 e^{(2i)} \quad 0 \leqslant v \leqslant max}{(0, N, v, c^{(2i)}) \xrightarrow{\epsilon} (j, e^{(2i)})} \qquad \frac{c^{(2i)} \xrightarrow{\mathsf{read}^{(x,\rho)}}{}_1 d^{(2i+1)} \xrightarrow{j^{(x,0)}}{}_1 e^{(2i)}}{(0, N, j, c^{(2i)}) \xrightarrow{\epsilon} (j, e^{(2i)})}$$

  Note that, thanks to **OA**, the construction will add (at most) $max + 1$ new transitions for each transition $c^{(2i)} \xrightarrow{\mathsf{write}(j)^{(x,\rho)}}{}_1 d^{(2i+1)}$. Observe that each of them has the shape $(0, N, v, c^{(2i)}) \xrightarrow{\epsilon} (j, e^{(2i)})$ and the whole group could be represented succinctly by writing $(0, N, ?, c^{(2i)}) \xrightarrow{\epsilon} (j, e^{(2i)})$, where ? is a wildcard representing an arbitrary value. So, we could say that each $c^{(2i)} \xrightarrow{\mathsf{write}(j)^{(x,\rho)}}{}_1 d^{(2i+1)}$ gives rise to a single transition with a wildcard. Similarly, thanks to **PQ**, each transition $d^{(2i+1)} \xrightarrow{j^{(x,0)}}{}_1 e^{(2i)}$ gives rise to (at most) one new transition $(0, N, j, c^{(2i)}) \xrightarrow{\epsilon} (j, e^{(2i)})$.

  Correctness follows from the fact that it suffices to restrict the work of M_1 to traces in which the relevant moves follow each other [10]. Further, by Lemma 5.3, it suffices to consider scenarios in which the associated transitions follow each other.

  As before, **OA**, **PQ** are preserved, because no new relevant transitions are introduced. **FA** follows by appealing to IH.

- $\mathbf{M \equiv newsem\, s\, in\, M_1}$

  This case is very similar to the previous one but only two values are possible: 0 (the initial one) or 1. The transitions for grabbing and releasing the semaphore are introduced as shown below. Thanks to **OA** (or **PQ**) and the nature of semaphores, only one transition will be added in each case.

$$\frac{c^{(2i)} \xrightarrow{\mathsf{grb}^{(s,\rho)}}_1 d^{(2i+1)} \xrightarrow{\mathsf{ok}^{(s,0)}}_1 e^{(2i)}}{(0,N,0,c^{(2i)}) \xrightarrow{\epsilon} (1,e^{(2i)})} \qquad \frac{c^{(2i)} \xrightarrow{\mathsf{rls}^{(s,\rho)}}_1 d^{(2i+1)} \xrightarrow{\mathsf{ok}^{(s,0)}}_1 e^{(2i)}}{(0,N,1,c^{(2i)}) \xrightarrow{\epsilon} (0,e^{(2i)})}$$

- $\mathbf{M \equiv f M_l \cdots M_1}$

  Suppose $\Gamma \vdash f M_l \cdots M_1 : \beta$ with $(f : \theta_l \to \cdots \to \theta_1 \to \beta) \in \Gamma$. Given $q \in I_{[\![\beta]\!]}$, we write A_q for the set of corresponding answers.

  We take $k = 2 + \max_{1 \leqslant i \leqslant l} k_i$, $N = \max_{1 \leqslant i \leqslant l} N_i$ (assuming that max taken over the empty range is 0).

$$\begin{aligned} C^{(0)} &= \{0_q \mid q \in I_{[\![\beta]\!]}\} + \{0_a \mid q \in I_{[\![\beta]\!]}, a \in A_q\} \\ C^{(1)} &= \{1_q \mid q \in I_{[\![\beta]\!]}\} \\ C^{(j+2)} &= \sum_{i=1}^{l} C_i^{(j)} \quad (0 \leqslant j \leqslant k). \end{aligned}$$

First we add transitions corresponding to calling and returning from f:

$$\frac{q \in I_{[\![\beta]\!]}}{\dagger \xrightarrow{q} \{0_q\}} \qquad \frac{q \in I_{[\![\beta]\!]}}{0_q \xrightarrow{q^f} 1_q} \qquad \frac{q \in I_{[\![\beta]\!]} \quad a \in A_q}{1_q \xrightarrow{a^f} 0_a} \qquad \frac{}{\{0_a\} \xrightarrow{a} \dagger}$$

Observe that these transitions satisfy **OA**, **PQ** and **FA**.

If $l \geqslant 1$ then in states 1_q we want to enable the environment to spawn an unbounded number of copies of each of $\Gamma \vdash M_u : \theta_u$ $(1 \leqslant u \leqslant l)$. This is done through the following rules, which embed the actions of the automata for M_u while relabelling the moves. We use $\square_L, \square_R$ to refer to arbitrary lhs and rhs of transitions, which are to be copied by the rule.

· Moves from M_u corresponding to θ_u obtain an additional annotation fu, as they are now the uth argument of $f : \theta_l \to \cdots \to \theta_1 \to \beta$.

$$\frac{\dagger \xrightarrow{m^{(\epsilon,0)}}_u D_u^{(0)}}{1_q \xrightarrow{m^{(fu,0)}} D_u^{(0)}} \qquad \frac{D_u^{(0)} \xrightarrow{m^{(\epsilon,0)}}_u \dagger}{D_u^{(0)} \xrightarrow{m^{(fu,0)}} \dagger} \qquad \frac{\square_L \xrightarrow{m^{(\boldsymbol{i},\rho)}}_u \square_R \quad \boldsymbol{i} \neq \epsilon}{\square_L \xrightarrow{m^{(fu\boldsymbol{i},\rho)}} \square_R}$$

The pointer structure is simply inherited in this case, but an additional pointer needs to be created to q^f from formerly initial moves for M_u (i.e. $m^{(\epsilon,0)}$), which did not have a pointer earlier. Fortunately, because we also use $\rho = 0$ in initial moves to represent the lack of a pointer, by copying 0 now we indicate that the move $m^{(fu,0)}$ points one level up, i.e. at the q^f move, as required.

· Moves from M_u that originate from Γ, i.e. moves of the form $m^{(x_v \boldsymbol{i},\rho)}$, where $(x_v \in \theta_v) \in \Gamma$, need no relabelling except for question-moves $m^{(x_v,\rho)}$ that need to point at the initial move $m^{(\epsilon,0)}$. Leaving ρ unchanged in this case would mean pointing at $m^{(fu,0)}$, whereas we need to point at $m^{(\epsilon,0)}$ instead. To readjust such pointers, we simply add 2 to ρ in the relevant moves, and preserve ρ in other moves.

$$\frac{\square_L \xrightarrow{m^{(x_v,\rho)}}_u \square_R \qquad m \text{ is a question}}{\square_L \xrightarrow{m^{(x_v,\rho+2)}} \square_R} \qquad \frac{\square_L \xrightarrow{m^{(x_v\boldsymbol{i},\rho)}}_u \square_R \qquad \boldsymbol{i} \neq \epsilon \text{ or } (\boldsymbol{i} = \epsilon \text{ and } m \text{ is an answer})}{\square_L \xrightarrow{m^{(x_v\boldsymbol{i},\rho)}} \square_R}$$

· All EPS transitions from M_i can be embedded in the new automaton but, because all states are now two levels deeper, we need to adjust the level of memory related transitions as follows.

$$\frac{(2j,h,v,c^{(2i)}) \xrightarrow{\epsilon}_u (v',d^{(2i)})}{(2j+2,h,v,c^{(2i)}) \xrightarrow{\epsilon} (v',d^{(2i)})}$$

The preservation of **OA** and **PQ** follows from the construction and IH, as the old transitions are simply copied in and relabelled injectively.

- $\mathbf{M} \equiv \lambda \mathbf{x}.\mathbf{M_1} : \theta_l \to \cdots \to \theta_1 \to \beta$

  This case is dealt with simply by renaming labels in the automaton for $\Gamma, x : \theta_l \vdash M_1 : \theta_{l-1} \to \cdots \to \theta_1 \to \beta$: labels of the form $m^{(xi,\rho)}$ must be renamed as $m^{(li,\rho)}$. **OA**, **PQ** and **FA** are inherited.

- $\mathbf{M} \equiv !\mathbf{M_1}$: Here we need to perform the same transitions as the automaton for M_1 would when started from read except that read needs to be relabelled to q. Consequently, it suffices to use all transitions from the automaton for M_1 except ADD(0), which needs to be modified as follows.

$$\frac{\dagger \xrightarrow{\mathsf{read}}_1 C^{(0)}}{\dagger \xrightarrow{\mathsf{q}} C^{(0)}}$$

  Clearly, this preserves **OA**, **PQ** and **FA**.

- $\mathbf{M} \equiv \mathbf{M_1} := \mathbf{M_2}$

  This case is similar to ;. First we direct the computation into $\mathcal{A}_{M_2}$ and, depending on the final move i, continue to $\mathcal{A}_{M_1}$, as if write(i) was played.

  $k = \max(k_1, k_2)$, $N = N_1 + N_2$, $C^{(0)} = C_1^{(0)} + \{\circ_i \,|\, 0 \leqslant i \leqslant max\}$, $C^{(i)} = C_1^{(i)} + C_2^{(i)}$ $(0 < i \leqslant k)$

$$\frac{\dagger \xrightarrow{\mathsf{q}}_2 D_2^{(0)}}{\dagger \xrightarrow{\mathsf{run}} D_2^{(0)}} \qquad \frac{D_2^{(0)} \xrightarrow{i}_2 \dagger}{D_2^{(0)} \xrightarrow{\epsilon} \{\circ_i\}} \qquad \frac{\dagger \xrightarrow{\mathsf{write}(i)}_1 D_1^{(0)}}{\{\circ_i\} \xrightarrow{\epsilon} D_1^{(0)}} \qquad \frac{D_1^{(0)} \xrightarrow{\mathsf{ok}}_1 \dagger}{D_1^{(0)} \xrightarrow{\mathsf{done}} \dagger}$$

  All transitions different from ADD(0) and DEL(0) need to copied (while respecting disjointness). For EPS$(0, 2i)$ transitions from $\mathcal{A}_{M_2}$, we need to add N_1 to indices of memory cells, as for ;.

- $\mathbf{M} \equiv \mathbf{grab}(\mathbf{M_1})$: Here we want to perform the same transitions as the automaton for M_1 would when started from grb. At the same time, grb and the corresponding answer ok have to be relabelled to run and done respectively. Consequently, it suffices to preserve all transitions from $\mathcal{A}_{M_1}$ except ADD(0) and DEL(0), which are modified as follows.

$$\frac{\dagger \xrightarrow{\mathsf{grb}}_1 D^{(0)}}{\dagger \xrightarrow{\mathsf{run}} D^{(0)}} \qquad \frac{D^{(0)} \xrightarrow{\mathsf{ok}}_1 \dagger}{D^{(0)} \xrightarrow{\mathsf{done}} \dagger}$$

  Clearly, **OA**, **PQ** and **FA** are inherited in this case.

- $\mathbf{M} \equiv \mathbf{release}(\mathbf{M_1})$

  This case is the same as the previous one but rls should be used instead of grb.

Complexity

Finally, we discuss complexity. We claim is that our constructions produce an automaton in which there are linearly many states, memory cells and transitions, with respect to term size.

For states, we observe that each construction adds at most a fixed number of new states to those obtained from IH. The same applies to memory cells.

The case of transitions is harder, as there are several ways in which transitions are added to the new automaton.

- The easiest case is when a transition is simply copied from an automaton obtained through IH without any changes to transition labels.
- Most other cases, represented by inference rules, are based on single premises (old transitions) and generate new single rules. As the old transitions are not included in the new automaton, such rules preserve the number of transitions.

- **newvar in** relies on a rule with two premises, but as we discussed the outcome could still be viewed as a single transition with a wildcard. Because other rules for transforming EPS transitions will not modify the position of the wildcard, all future transformations can be performed in the presence of wildcards.

  The case of reading is easier, as **OA** and **PQ** guarantee that only one transition will be produced per application of the rule. This will not increase the number of transitions in the automaton.
- When transitions cannot be traced back to old ones, their number is always bounded by a constant (we regard *max* as a constant too).

Overall we can conclude that the number of transitions (possibly with wildcards) is linear. Because each transition with a wildcard represents $max + 1$ transitions without wildcards, by instantiating them we also obtain a linear number of transitions. It is also worth noting that each transition involves at most three states: whenever sets of states are involved in transitions, they contain at most two elements.

Finally, we assess the time complexity of the constructions. A typical case consists of invoking IH and performing a bounded number of linear-time operations on the results to implement the constructions, such as retagging to implement the disjoint sum and relabelling. The combinations of transitions mentioned in **newvar** can also be considered in linear time after some preprocessing that guarantees constant-time access to incoming and outgoing transition of a given state. Overall, this could be viewed as a linear number of linear-time operations, yielding quadratic time complexity.