Saturating Automata for Game Semantics

Alex Dixon\textsuperscript{a}  Andrzej S. Murawski\textsuperscript{b,1}

\textsuperscript{a} Department of Computer Science  
University of Warwick  
Coventry, UK  
\textsuperscript{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 \cite{2,11}, it did not take long for the methodology to be extended to other programming constructs such as state \cite{3,1}, control operators \cite{14}, and, soon afterwards, concurrency. Some of the game models were presented in the interleaving tradition of models of concurrency \cite{15,16,10}, while others were built in the spirit of partial-order methods (true concurrency) \cite{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...
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 moves $m_1 m_2$ 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 $\{0,\ldots, \mathit{max}\} (\mathit{max} \geq 0)$, with no recursion, but with iteration. Its types $\theta$ are generated by the grammar

$$\theta ::= \beta \mid \theta \to \theta \quad \beta ::= \mathit{com} \mid \mathit{exp} \mid \mathit{var} \mid \mathit{sem}$$
where \( \text{com} \) is the type of commands; \( \text{exp} \) that of \( \{0, \ldots, \text{max}\} \)-valued expressions; \( \text{var} \) that of assignable variables; and \( \text{sem} \) that of semaphores. The typing judgments are displayed in Figure 1. Here, \( \text{skip} \) and \( \text{div}_\theta \) are constants representing termination and divergence respectively, \( i \) ranges over \( \{0, \ldots, \text{max}\} \), and \( \text{op} \) 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 \( \text{newvar} \) and \( \text{newsem} \). Variables are dereferenced using \( \text{!}\) and, semaphores are manipulated using two (blocking) primitives, \( \text{grab}(s) \) and \( \text{release}(s) \), 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 \( \text{newvar} \ x := i \in M \) and \( \text{newsem} \ x := i \in M \), which allows us to specify initial values more flexibly, i.e. \( \text{newvar} \ x \in M \) and \( \text{newsem} \ x \in M \) should be viewed as \( \text{newvar} \ x := 0 \in M \) and \( \text{newsem} \ x := 0 \in M \) respectively.

The operational semantics is defined using a (small-step) transition relation \( \mathcal{V} \vdash M, s \rightarrow M', s' \). Figure 2, where \( \mathcal{V} \) is a set of variable names denoting active memory cells and semaphore locks, \( v \in \mathcal{V} \), are states, i.e.

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

\[
\begin{align*}
\mathcal{E}[-] := \ & \text{[-]} \ | \ \mathcal{E} ; \mathcal{N} \ | \ (\mathcal{E} || \mathcal{N}) \ | \ (\mathcal{M} || \mathcal{E}) \ | \ \mathcal{E}\mathcal{N} \ | \ \text{op}(\mathcal{E}) \ | \ \text{if } \mathcal{E} \ \text{then } N_1 \ \text{else } N_2 \\
& | \ \mathcal{E} \ | \ \mathcal{E} := m \ | \ M := \mathcal{E} \ | \ \text{grab}(\mathcal{E}) \ | \ \text{release}(\mathcal{E}).
\end{align*}
\]

We say that a term \( \vdash M : \text{com} \) may terminate, written \( M \Downarrow \), if \( \emptyset \vdash \emptyset, M \rightarrow^* \emptyset, \text{skip} \).

\( \text{FICA} \) terms can be compared using a notion of contextual (may-)equivalence, denoted \( \vdash M_1 \simeq M_2 \). 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 \( C \) such that \( \vdash C[M_1] : \text{com} \), we have \( C[M_1] \Downarrow \).
if and only if \( \mathcal{C}[M_2] \downarrow \). Using game semantics, one can reduce \( \equiv \) to equality of the associated sets of complete plays (Theorem 3.5).

**Example 2.1** Consider the term

\[
f : \mathbf{com} \rightarrow \mathbf{com}, \ c : \mathbf{com} \vdash \mathbf{newvar} \ X \mathbf{in} (f \ (X := 1) \ | \ if \ !X \ then \ c \ else \ \mathbf{div}_{\mathbf{com}}); \!X : \mathbf{exp}
\]

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 \( \lambda_A^O, \lambda_A^Q \) for the composite of \( \lambda_A \) 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 \( \lambda_A^O(m) \neq \lambda_A^Q(n) \), and if \( m \vdash_A n \) then \( \lambda_A^Q(m) = Q \).

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 (\( 0 \leq i \leq \text{max} \)).

<table>
<thead>
<tr>
<th>Arena</th>
<th>OQ</th>
<th>PA</th>
</tr>
</thead>
<tbody>
<tr>
<td>[com]</td>
<td>run</td>
<td>done</td>
</tr>
<tr>
<td>[var]</td>
<td>read</td>
<td>done</td>
</tr>
<tr>
<td>write(i)</td>
<td>ok</td>
<td></td>
</tr>
<tr>
<td>[exp]</td>
<td>q</td>
<td>i</td>
</tr>
<tr>
<td>[sem]</td>
<td>grb</td>
<td>ok</td>
</tr>
<tr>
<td>rls</td>
<td>ok</td>
<td></td>
</tr>
</tbody>
</table>
\[
M_{A \times B} = M_A + M_B \quad M_{A \Rightarrow B} = M_A + M_B
\]
\[
\lambda_{A \times B} = [\lambda_A, \lambda_B] \quad \lambda_{A \Rightarrow B} = [\langle \lambda_A^{PO}, \lambda_A^O \rangle, \lambda_B]
\]
\[
\vdash_{A \times B} = \vdash_A + \vdash_B \quad \vdash_{A \Rightarrow B} = \vdash_A + \vdash_B + \{ (b, a) \mid b \in I_B \text{ and } a \in I_A \}
\]

Fig. 3. Arena constructions (+ and [· · · ] stand for the disjoint union of sets and functions respectively; ⟨· · · ⟩ denotes pairing).

\[
A = [\text{com} \rightarrow \text{com}] \times [\text{com}] \Rightarrow [\text{exp}]
\]

\[
O \quad q \quad \text{run}_f \quad \text{done}_f
\]

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

\[
O \quad P \quad \text{run}_f \quad \text{run}_c \quad \text{done}_f
\]

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

\[
s_2 = \quad q \quad \text{run}_f \quad \text{run}_f \quad \text{done}_f \quad \text{run}_c \quad \text{done}_c \quad \text{done}_c \quad \text{done}_f \quad 1
\]

(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 \( \theta \). In Figure 4a, we give (the enabling relation of) the arena \( A = ([\text{com} \rightarrow \text{com}] \times [\text{com}]) \Rightarrow [\text{exp}] \), 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 \( s' = \cdots q \cdots m \) of \( s \), the question \( q \) must be pending when \( m \) is played.

**WAIT**: In any prefix \( s' = \cdots q \cdots a \) 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: $m_1 m_2$ or $m_1, m_2$ are answers to questions $q_1, q_2$ such that $q_2$ justifies $q_1$.

A subset $\sigma$ 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 $\Gamma = \{x_1 : \theta_1, \ldots , x_i : \theta_i\}$ and $\Gamma \vdash M : \theta$ is a FICA-term. Let us write $[\Gamma \vdash \theta]$ for the arena $[\theta_1] \times \cdots \times [\theta_i] \Rightarrow [\theta]$. 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, $[\Gamma \vdash \text{div}] = \{\epsilon, \text{run}\}$ and $[\Gamma \vdash \text{skip}] = \{\epsilon, \text{run}, \epsilon, \text{run done}\}$. The plays $s_1, s_2$ turn out to belong to the strategy that interprets the term from Example 2.1. Given a strategy $\sigma$, we denote by $\text{comp}(\sigma)$ the set of non-empty complete plays of $\sigma$, 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 $\Gamma \vdash M_1 \equiv M_2$ if and only if $\text{comp}([\Gamma \vdash M_1]) = \text{comp}([\Gamma \vdash M_2])$.

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 $\geq P_A \times P_A$ to be the least preorder satisfying $sm os' \geq sm os'$ and $sp ms' \geq sp ms'$, 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.

![Diagram](image)

Observe that $s_2 \geq s_3$, where $s_2$ is the play from Figure 4c, because the $P$-move $\text{done}_1$ moved to the right past a $P$-move ($\text{run}_c$) and an $O$-move ($\text{done}_c$). In contrast, we do not have $s_3 \geq s_2$, as this would involve moving a $P$-move ($\text{done}_1$) left past an $O$-move ($\text{done}_c$).

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

![Diagram](image)

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

**Remark 3.9** Definition 3.8 states that saturated strategies are stable under $\geq$. Note that $s o p s' \geq s o p s'$, while other $o/p$ combinations are allowed in $\geq$. 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 $\Sigma \times D$, where $\Sigma$ is a finite alphabet and $D$ is an infinite alphabet of data values. In our case, the dataset $D$ 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** $D$ is a countably infinite set equipped with a function $\text{pred} : D \rightarrow D \cup \{\bot\}$ (the parent function) such that the following conditions hold.

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

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

**Example 4.2** Suppose $\Sigma$ consists of moves used in Figure 4a, $\text{pred}(d_0) = \bot$, $\text{pred}(d_1) = \text{pred}(d'_1) = d_0$ and $\text{pred}(d_2) = d_1$. The play $s_2$ (Figure 4c) can be represented by the following word over $\Sigma \times D$: $(q, d_0)(\text{run}^f, d_1)(\text{run}^f, d_2)(\text{done}^f, d_2)(\text{run}^c, d'_2)(\text{done}^c, d'_1)(\text{done}^f, d_1)(1, d_0)$.

We use subtrees of $D$ 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 $\epsilon$-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 $V = \{0, \ldots, \text{max}\}$. The set of control states will be partitioned into sets $C^{(i)}$, for $0 \leq i \leq k$, dedicated to representing run-time information at the corresponding level $i$.

**Definition 4.3** A saturating automaton (SATA) is a tuple $A = (\Sigma, k, N, C, \delta)$, 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);
- $k \geq 0$ is the depth parameter and $N \geq 0$ is the local memory capacity;
- $C = \sum_{i=0}^{k} C^{(i)}$ is a finite set of control states, partitioned into sets $C^{(i)}$ of level-$i$ control states;
- transitions in $\delta$ 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 $D^{(2i)}, E^{(2i)} \in \mathcal{M}(C^{(2i)})$, where $\mathcal{M}(X)$ denotes the set of multisets over $X$.
- ADD($2i$) transitions have the form $c^{(2i-1)} \xrightarrow{\text{ADD}} D^{(2i)}$ or $\xrightarrow{\text{ADD}} D^{(0)}$ for the special case of $i = 0$;
- ADD($2i + 1$) transitions have the form $c^{(2i)} \xrightarrow{\text{ADD}} d^{(2i+1)}$;
- DEL($2i$) transitions have the form $D^{(2i)} \xrightarrow{\text{DEL}}$;
- DEL($2i + 1$) transitions have the form $c^{(2i+1)} \xrightarrow{\text{DEL}} d^{(2i)}$;
- EPS($2i$) transitions have the form $D^{(2i)} \xrightarrow{\text{EPS}} E^{(2i)}$;
- EPS($2j, 2i$) transitions read $v \in V$ from memory cell $h \in \{1, \ldots, N\}$ at level $2j \leq 2i$ and update it to $v' \in V$, but do not read the input: $(2j, h, v, c^{(2i)}) \xrightarrow{\epsilon} (v', d^{(2i)})$. 

7
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 number 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), \(f : E \rightarrow \sum_{0<2i-1 \leq k} C^{(2i-1)} + \sum_{0 \leq 2i \leq k} \mathfrak{M}(C^{(2i)})\) 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 \(m : E \rightarrow V^N\) is a partial function whose domain is the set of even-level nodes of \(E\).

A SATA \(\mathcal{A}\) starts from the empty configuration \(\kappa_0 = (\emptyset, \emptyset, \emptyset, \emptyset)\) and proceeds according to its transitions \(\delta\), 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 \(t \in \Sigma_{OQ} + \Sigma_{PQ}, d \notin D, \text{ pred}(d) \in E, D' = D \cup \{d\}, E' = E \cup \{d\}\), and if the transition-specific constraints from the table below are satisfied\(^2\). We write \(f[\cdots]\) to extend or update \(f\).

<table>
<thead>
<tr>
<th>(t)</th>
<th>transition</th>
<th>pre-condition</th>
<th>(f')</th>
<th>(m')</th>
</tr>
</thead>
<tbody>
<tr>
<td>(q_0)</td>
<td>(\uparrow^{\text{OQ}} \cdot D^{(0)})</td>
<td>(D = \emptyset)</td>
<td>({d \mapsto D^{(0)}})</td>
<td>({d \mapsto 0_N})</td>
</tr>
<tr>
<td>(q_0)</td>
<td>(c^{(2i-1)} \quad \text{ or } \quad D^{(2i)})</td>
<td>(f(\text{ pred}(d)) = c^{(2i-1)})</td>
<td>(f[d \mapsto D^{(2i)}])</td>
<td>(m[d \mapsto 0_N])</td>
</tr>
<tr>
<td>(q_P)</td>
<td>(c^{(2i)} \quad \text{ or } \quad d^{(2i+1)})</td>
<td>(c^{(2i)} \in_m f(\text{ pred}(d)))</td>
<td>(f)</td>
<td>(m)</td>
</tr>
</tbody>
</table>

Note that, in the first two cases, memory is initialised at the new node. In the last case, \(c^{(2i)}\) is removed from \(f(\text{ pred}(d))\), 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 \(E, D' = D, E' = E \setminus \{d\}, m' = m\), and the transition-specific constraints listed below are satisfied.

<table>
<thead>
<tr>
<th>(t)</th>
<th>transition</th>
<th>pre-condition</th>
<th>(f')</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a_O)</td>
<td>(c^{(2i+1)} \quad \text{ or } \quad d^{(2i)})</td>
<td>(f(d) = c^{(2i+1)})</td>
<td>(f[\text{ pred}(d) \mapsto f(\text{ pred}(d)) \cup_m {d^{(2i)}}])</td>
</tr>
<tr>
<td>(a_P)</td>
<td>(D^{(2i)} \quad \text{ or } \quad \uparrow)</td>
<td>(f(d) = D^{(2i)})</td>
<td>(f)</td>
</tr>
</tbody>
</table>

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 \((X, \mu : X \rightarrow \mathbb{N})\), we write \(x \in_m (X, \mu)\) to mean \(\mu(x) > 0\). Given two multisets \((X, \mu_i)\) \((i = 1, 2)\), we write \((X, \mu_1)_m (X, \mu_2), (X, \mu_1) \cup_m (X, \mu_2)\) to stand for \((X, \mu^-)\) and \((X, \mu^+)\) 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 \(\mu_1(x) \leq \mu_2(x)\) 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 \( D^{(2i)} \xrightarrow{\varepsilon} E^{(2i)} \), we require \( D^{(2i)} \subseteq_m f(d), f' = f[d \mapsto (f(d))^m D^{(2i)}] \cup_m E^{(2i)} \) and \( m' = m \).
- For \( (2j, h, v, c^{(2i)}) \xrightarrow{\varepsilon} (v', d^{(2i)}) \), we require \( c^{(2i)} \subseteq_m f(d) \) and \( m(\text{pred}^{2i-2}(d)(h)) = v, f' = f[d \mapsto (f(d))_m \{c^{(2i)}\}] \cup_m \{d^{(2i)}\} \) and \( m' = m[\text{pred}^{2i-2}(d)(h) \mapsto v'] \).

Note that, in the second case, \( m(\text{pred}^{2i-2}(d)(h)) \) refers to the \( h \)th 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 \( w \in (\Sigma \times D)^* \) such that \( \kappa_0 \xrightarrow{1} \kappa_1 \ldots \kappa_{h-1} \xrightarrow{h} \kappa_h \), where \( \kappa_0 = (\emptyset, \emptyset, \emptyset, \emptyset), l_i \in \{\epsilon\} \cup (\Sigma \times D) \) (1 \( \leq i \leq h \)) and \( w = l_1 \ldots l_h \). 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 \( \kappa_h \) accepting. The set of traces (resp. accepted traces) of \( \mathcal{A} \) is denoted by \( \text{Tr}(\mathcal{A}) \) (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 \( a_P/a_Q \) (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 \( \Sigma_OQ = \{q, \text{run}^f\}, \Sigma_{PQ} = \{\text{run}^f, \text{run}^c\}, \Sigma_OA = \{\text{done}^f, \text{done}^c\} \) and \( \Sigma_PA = \{\text{done}^f, 0, \ldots, \text{max}\} \). \( C^{(0)} = \{t_1^{(0)}, t_2^{(0)}, t_1^{(0)}, t_2^{(0)}, r_3^{(0)}, r_4^{(0)}\}, C^{(1)} = \{t_1^{(1)}, t_2^{(1)}\}, C^{(2)} = \{t_1^{(2)}, t_2^{(2)}\} \). \( \delta \) is given below.

\[
\begin{align*}
\text{ADD}(0), \text{DEL}(0): & \quad \uparrow q \rightarrow \{t_1^{(0)}, r_1^{(0)}\} \quad \{t_2^{(0)}, r_4^{(0)}\} \rightarrow \uparrow \\
\text{ADD}(1), \text{DEL}(1): & \quad \uparrow t_1^{(0)} \rightarrow \{t_1^{(1)}\} \quad \{t_2^{(0)}\} \rightarrow \{t_1^{(1)}, t_2^{(1)}\} \rightarrow \{t_2^{(0)}\} \rightarrow \{t_1^{(0)}, t_2^{(0)}\} \rightarrow \{t_2^{(1)}\} \rightarrow t_1^{(1)} \rightarrow \{t_2^{(0)}\} \rightarrow \{t_1^{(0)}, t_2^{(0)}\} \rightarrow \{t_2^{(1)}\} \\
\text{ADD}(2), \text{DEL}(2): & \quad \uparrow t_1^{(1)} \rightarrow \{t_2^{(1)}\} \rightarrow \{t_2^{(1)}\} \rightarrow \{t_2^{(1)}\} \rightarrow \{t_2^{(1)}\} \rightarrow \{t_2^{(1)}\} \rightarrow \{t_2^{(1)}\} \\
\text{EPS}(0,0): & \quad (0, 1, 0, r_1^{(0)}) \rightarrow (0, r_2^{(0)}) \quad (0, 1, i, r_1^{(0)}) \rightarrow (i, r_3^{(0)}) \quad (0 < i \leq \text{max}) \\
\text{EPS}(0,2): & \quad (0, 1, i, t_1^{(2)}) \rightarrow (1, t_2^{(2)}) \quad (0 < i \leq \text{max})
\end{align*}
\]

We will now show a possible transition sequence for \( \mathcal{A} \). For the sake of simplicity, data values from \( D \) 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 \( D \) currently maintained in the automaton.

Notes at even levels \( 2i \) are written \( d(X, m) \), where \( d \) is a level-\( 2i \) data value, \( X \in M(C^{(2i)}) \) 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. \( d_1, d_2 \in D \) will be called independent if neither \( d_1 = \text{pred}^k(d_2) \) nor \( d_2 = \text{pred}^k(d_1) \) for \( k \geq 0 \), i.e. the data lie on different branches. Let \( \Sigma_O = \Sigma_{OQ} + \Sigma_{OA} \) and \( \Sigma_P = \Sigma_{PQ} + \Sigma_{PA} \).
We inspect the shape of the relevant rules, which are listed below.

Definition 5.1 We shall say that \( L \subseteq (\Sigma \times D)^* \) 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 \( \Sigma_A \) and the second one from \( \Sigma_P \). 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 \((\epsilon \Omega \Rightarrow O \epsilon)\) 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 \( \kappa_2' \).

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

\[
\begin{array}{c|c|c}
\epsilon & D^{(2i)} \xrightarrow{\epsilon} E^{(2i)} & O \\
\hline
\end{array}
\]

Observe that the EPS transitions do not modify states at odd levels or add nodes. Thus, the \( \Sigma_O \) transitions
could be fired from \( \kappa_1 \). Now note that the \( \Sigma_O \) 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 \( \Sigma_O \) 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 \((P \epsilon \Rightarrow \epsilon P)\) 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 \( \kappa_2' \).

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

\[
\begin{array}{c|c|c}
P & D^{(2i)} \xrightarrow{\epsilon} E^{(2i)} & \epsilon \\
\hline
\end{array}
\]

\[\epsilon \]
\[
(2j, h, v, c^{(2i)}) \xrightarrow{\epsilon} (v', d^{(2i)})
\]
Observe that the $\epsilon$ transitions do not depend on any information introduced by transitions on $\Sigma_P$. Hence, they are executable from $\kappa_1$. Note also that they will not destroy any information needed to execute the $\Sigma_P$ 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 $\epsilon$ transition may well be followed by a transition on $\Sigma_P$ that relies on the states introduced by the $\epsilon$ transition.

**Remark 5.7** One can use Lemmata 5.3 and 5.5 to replace sequences of transitions of the form $\kappa_1 \xrightarrow{(t_1, d_1)} \cdot \xrightarrow{\epsilon} \cdot \xrightarrow{(t_2, d_2)} \kappa'$ 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 $\kappa_1 \xrightarrow{(t_1, d_1)} \xrightarrow{(t_2, d_2)} \kappa'$.
- 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 $\kappa_2'$ such that $\kappa_1 \xrightarrow{(t_2, d_2)}' \xrightarrow{(t_1, d_1)} \kappa_3'$.

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

Now suppose $\text{pred}(d_1) = \text{pred}(d_2)$ 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: $\text{ADD}(2i) \text{ADD}(2i)$, $\text{DEL}(2i) \text{ADD}(2i)$, $\text{DEL}(2i) \text{DEL}(2i)$ (other cases can be ignored due to the $t_1 \in \Sigma_P$ or $t_2 \in \Sigma_O$ constraint). Recalling that $\text{ADD}(2i)$ and $\text{DEL}(2i)$ transitions have the form $c^{(2i-1)} \xrightarrow{\delta_{\sigma_1}} \text{DEL}(2i)$ and $\text{ADD}(2i) \xrightarrow{\delta_{\sigma_1}}$ respectively, we can confirm that the Lemma holds, because the state $c^{(2i-1)}$ associated with $\text{pred}(d_1) = \text{pred}(d_2)$ 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: $\text{ADD}(2i+1) \text{ADD}(2i+1)$, $\text{ADD}(2i+1) \text{DEL}(2i+1)$, $\text{DEL}(2i+1) \text{DEL}(2i+1)$ (other cases can be ignored due to the $t_1 \in \Sigma_P$ or $t_2 \in \Sigma_O$ constraint). Recalling that $\text{ADD}(2i+1)$ and $\text{DEL}(2i+1)$ transitions have the form $c^{(2i)} \xrightarrow{\delta_{\sigma_1}} d^{(2i+1)}$ and $c^{(2i+1)} \xrightarrow{\delta_{\sigma_1}} d^{(2i)}$ respectively, we can confirm that the Lemma holds, because the transitions will not interfere. In particular, due to $d_1 \neq d_2$, the $\text{DEL}(2i+1)$ transition in $\text{ADD}(2i+1) \text{DEL}(2i+1)$ cannot use the state introduced by the preceding $\text{ADD}(2i+1)$ transition.

**Remark 5.9** Note that the “$t_1 \in \Sigma_P$ or $t_2 \in \Sigma_O$” condition is necessary: in the $\text{DEL}(2i+1) \text{ADD}(2i+1)$ 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 $A$, the sets $\text{Tr}(A), L(A)$ 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 $w_1(t_1, d_1)(t_2, d_2)w_2 \in \text{Tr}(A)$. Thus, there exist $\kappa_1, \kappa_2$ 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 $\kappa_1 \xrightarrow{(t_1, d_1)} \xrightarrow{\epsilon} \xrightarrow{(t_2, d_2)} \kappa_2$. By Lemma 5.8, we then obtain $\kappa_1 \xrightarrow{(t_1, d_1)} \xrightarrow{\epsilon} \xrightarrow{(t_2, d_2)} \kappa_2$, i.e. $w_1(t_2, d_2)(t_1, d_1)w_2 \in \text{Tr}(A)$. Hence, $\text{Tr}(A)$ is saturated. As $L(A)$ is a subset of $\text{Tr}(A)$ in which all questions have answers, $L(A)$ is also saturated, because the swaps do not affect membership in $L(A)$.
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 \( c^{(2i)} \xrightarrow{dp} (d^{(2i)}, d^{(2i+1)}) \) and \( (c^{(2i)}, c^{(2i+1)}) \xrightarrow{ao} d^{(2i)} \). 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 2l 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

\[
\mathcal{M} = M_{\text{[com]}} \cup M_{\text{[exp]}} \cup M_{\text{[var]}} \cup M_{\text{[sem]}}
\]

\[= \{ \text{run, done, q, read, grb, rlis, ok} \} \cup \{ i, \text{write}(i) \mid 0 \leq i \leq \max \} \]

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 \( \Gamma \) and \( \theta \). 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 \( \Gamma = \{ x_1 : \theta_1, \ldots, x_l : \theta_l \} \). The superscripts will have one of the two forms, where \( i \in \mathbb{N}^* \) and \( \rho \in \mathbb{N} \):

- \((i, \rho)\) will represent moves from \( \theta \);
- \((x_i, i, \rho)\) will represent moves from \( \theta_v \) (\( 1 \leq v \leq l \)).

The annotated moves will be written as \( m^{(i,\rho)} \) or \( m^{(x_i, i, \rho)} \), where \( m \in \mathcal{M} \). We will sometimes omit \( \rho \) on the understanding that this represents \( \rho = 0 \). Similarly, when \( i \) is omitted, the intended value is \( \varepsilon \), e.g. \( m \) stands for \( m^{(\varepsilon,0)} \) and \( m^x \) for \( m^{(x,0)} \). The next definition explains how the \( i \) superscripts are linked to moves from \( [\theta] \).

Given \( X \subseteq \{ m^{(i,\rho)} \mid i \in \mathbb{N}^*, \rho \in \mathbb{N} \} \) and \( y \in \mathbb{N} \cup \{ x_1, \cdots, x_l \} \), we let \( y X = \{ m^{(y,\rho)} \mid m^{(k,\rho)} \in X \} \).

Definition 6.1 Given a type \( \theta \), the corresponding alphabet \( T_\theta \) is defined as follows

\[
T_\beta = \{ m^{(\varepsilon,\rho)} \mid m \in M_{[\beta]}, \rho \in \mathbb{N} \} \quad \beta = \text{com, exp, var, sem}
\]

\[
T_{\theta_1 \rightarrow \cdots \rightarrow \theta_l \rightarrow \beta} = \bigcup_{\alpha=1}^{l}(uT_{\theta_\alpha}) \cup T_\beta
\]

For \( \Gamma = \{ x_1 : \theta_1, \cdots, x_l : \theta_l \} \), the alphabet \( T_{\Gamma \vdash \theta} \) is defined to be \( T_{\Gamma \vdash \theta} = \bigcup_{\alpha=1}^{l}(x_iT_{\theta_\alpha}) \cup T_\theta \).

Example 6.2 Given \( \Gamma = \{ f : \text{com} \rightarrow \text{com}, c : \text{com} \} \), we have

\[
T_{\Gamma \vdash \text{exp}} = \{ \text{run}(f^{(1,\rho)}, \text{done}(f^{(1,\rho)}), \text{run}(f, \rho), \text{done}(f, \rho), \text{run}(c, \rho), \text{done}(c, \rho), q^{(\varepsilon,\rho)}, i^{(\varepsilon,\rho)} \mid 0 \leq i \leq \max, \rho \in \mathbb{N} \}.
\]
Note that $T_{\Gamma\theta}$ 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 $\Sigma$ is a finite subset of $T_{\Gamma\theta}$. Only a finite subset will be needed, because $\rho$ will be bounded.

Next we explain how $\rho$ and $\omega$ 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 value can be used twice with a question, occurrences of questions correspond to unique data values. A

**Example 6.3** The play $q \xrightarrow{\text{run}^e} \text{run}^f \xrightarrow{\text{run}^c}$ can be represented by $(\text{run}^{(e,0)}, d_0) (\text{run}^{(f,0)}, d_1) (\text{run}^{(j,0)}, d_2)$, given $\text{pred}(d_{i+1}) = d_i$ ($0 \leq i \leq 2$).

Below we state the main result linking FICA with saturating automata. Question-moves in this translation are handled with ADD transitions: $\text{ADD}(2i)$ and $\text{ADD}(2i + 1)$ correspond to O- and P-questions respectively. Answer-moves are processed with DEL transitions: $\text{DEL}(2i)$ for P-answers and $\text{DEL}(2i + 1)$ 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 $T_{\Gamma\theta}$ such that the set of plays represented by words from $\text{Tr}(\mathcal{A}_M)$ is $[\Gamma \vdash M : \theta]$, and $L(\mathcal{A}_M)$ represents $\text{comp}(\Gamma \vdash M : \theta)$. Moreover, when $M$ is in $\beta$-normal $\eta$-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 $\beta$-normal $\eta$-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. $C_{i}^{(j)}, \quad m \mapsto$ (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 $c^{(2i+1)} \xrightarrow{a} d_1^{(2i)}$ and $c^{(2i+1)} \xrightarrow{a} d_2^{(2i)}$ then $d_1^{(2i)} = d_2^{(2i)}$.
- **PQ** (PQ pre-determinacy): if $c_1^{(2i)} \xrightarrow{q} d_1^{(2i+1)}$ and $c_2^{(2i)} \xrightarrow{q} d_2^{(2i+1)}$ then $c_1^{(2i)} = c_2^{(2i)}$.
- **FA** (final readiness): for every $D^{(0)} \xrightarrow{a} \uparrow$, 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.

$M \equiv M_1 \cup M_2$.

This construction needs to interleave $M_1$ and $M_2$ while gluing the initial and final moves. Accordingly, we take $k = \max(k_1, k_2)$, $N = N_1 + N_2$, $C^{(i)} = C_1^{(i)} + C_2^{(i)} + \{c_1, c_2, \{1\}, \{2\}\}$, $C^{(i)} = C_{1}^{(i)} + C_{2}^{(i)} (0 < i \leq k)$, assuming $C_{i}^{(i)} = \emptyset$ for $i > k_u$. All transitions from $\mathcal{A}_M$ and $\mathcal{A}_M$ 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.

\[
\begin{align*}
\uparrow \xrightarrow{\text{run}^u} \{c_1, c_2\} & \quad \downarrow \xrightarrow{\text{run}^u} D_u^{(0)} \quad u \in \{1, 2\} \\
\{c_1\} \xrightarrow{\text{run}^u} D_u^{(0)} & \quad \{c_1\} \rightarrow D_u^{(0)} \\
\{c_u\} \rightarrow D_u^{(0)} \quad u = 1, 2 & \quad D_u^{(0)} \text{ done} \quad \& \quad u = 1, 2 \quad \{\{1\}, \{2\}\} \text{ done} \quad \uparrow
\end{align*}
\]
\(N = N_1 + N_2\) 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 \(\text{EPS}(0, 2i)\) transitions from \(\mathcal{A}_{M_2}\):

\[
\frac{(0, h, v, c(2i))}{(0, N_1 + h, v, c(2i))} \xrightarrow{\epsilon} (v', d(2i))
\]

\(M = f(M_1)\):

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

\[
\begin{align*}
\uparrow_{\text{run}(x,0)} & \rightarrow \{0_{\text{run}}\} \\
0_{\text{run}} & \rightarrow 1_{\text{run}} \\
1_{\text{run}} & \rightarrow 0_{\text{done}} \rightarrow \{0_{\text{done}}\} \rightarrow \text{done}(x,0) \uparrow.
\end{align*}
\]

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

\[
\begin{align*}
\uparrow_{\text{run}(x,0)} & \rightarrow D_1^{(0)} \\
1_{\text{run}} & \rightarrow D_1^{(1)} \rightarrow D_1^{(0)} \\
D_1^{(0)} & \rightarrow \text{done}(x,0) \uparrow.
\end{align*}
\]

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

The remaining moves related to \(M_1\) originate from \(\Gamma\), i.e. are of the form \(m^{(x, i, \rho)}\), 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 \(m^{(x, \rho)}\) (corresponding to initial moves of \([\theta_v]\)) need to have their pointer adjusted so that they point at the move tagged with \(\text{run}(x,0)\) (leaving \(\rho\) unchanged in this case would mean pointing at \(\text{run}(f,0)\)). To achieve this, it suffices to add 2 to \(\rho\) in this case. Otherwise \(\rho\) can remain unchanged, because the pointer structure is preserved. Below we use \(\sqcap_L, \sqcap_R\) to refer to arbitrary left/right-hand sides of transition rules.

\[
\begin{align*}
\sqcap_L m^{(x, i, \rho)} & \rightarrow 1 \sqcap_R \\
\sqcap_L m^{(x, i, \rho + 2)} & \rightarrow \sqcap_R
\end{align*}
\]

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

\[
\begin{align*}
\frac{(2j, h, v, c(2i))}{(2j + 2, h, v, c(2i))} \xrightarrow{\epsilon} (v', d(2i))
\end{align*}
\]

\(M \equiv \text{newvar } x \in M_1:\)

According to [10], it suffices to consider plays from \(M_1\) in which \(\text{read}(x, \rho)\) and \(\text{write}(j)(x, \rho)\) 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 \(k = k_1\), \(N = N_1 + 1\), \(C^{(0)} = C^{(0)} + \{\cdot, \cdot\}, C^{(i)} = C^{(i)} (0 < i < k)\). All transitions from \(\mathcal{A}_{M_1}\) can be copied over except \(\text{ADD}(0), \text{DEL}(0)\) and those with superscripts of the form \((x, \rho)\), i.e. related to \(x\). \(\text{ADD}(0)\) and \(\text{DEL}(0)\) are handled as specified below.

\[
\begin{align*}
\frac{m \in I_{[\beta]}}{\uparrow m \rightarrow \{\cdot\}} \\
0 \leq v \leq \text{max} & \rightarrow (0, N, v, \cdot) \rightarrow (0, \cdot) \\
\frac{0 \rightarrow 1 \downarrow D_1^{(0)}}{\{\cdot\} \rightarrow \cdot \rightarrow D_1^{(0)} \\
D_1^{(0)} & \rightarrow a \rightarrow 1 \uparrow \\
D_1^{(0)} & \rightarrow a \rightarrow \uparrow
\end{align*}
\]

Note that in this case \(\beta = \text{com, exp}\), so \(I_{[\beta]} = \{\text{run}\} \) or \(I_{[\beta]} = \{\text{q}\}\).
For transitions related to $x$ we proceed as follows.

\[
\begin{align*}
\frac{c^{(2i)} \text{write}(j)^{(x,\rho)} \rightarrow_1 d^{(2i+1)} \text{ok}^{(x,0)} \rightarrow_1 e^{(2i)}}{(0, N, v, c^{(2i)}) \rightarrow^\epsilon (j, e^{(2i)})} & \quad 0 \leq v \leq \max \\
\frac{c^{(2i)} \text{read}^{(x,\rho)} \rightarrow_1 d^{(2i+1)} j^{(x,0)} \rightarrow_1 e^{(2i)}}{(0, N, j, c^{(2i)}) \rightarrow^\epsilon (j, e^{(2i)})}
\end{align*}
\]

Thanks to OA, the construction will add (at most) $\max + 1$ new transitions for each transition $c^{(2i)} \text{write}(j)^{(x,\rho)} \rightarrow_1 d^{(2i+1)}$. Observe that they have the shape $(0, N, v, c^{(2i)}) \rightarrow^\epsilon (j, e^{(2i)})$ $(0 \leq v \leq \max)$, and could be represented succinctly by writing $(0, N, ?, c^{(2i)}) \rightarrow^\epsilon (j, e^{(2i)}), \text{where} \ ? \text{is a wildcard representing an arbitrary value. So, each} c^{(2i)} \text{write}(j)^{(x,\rho)} \rightarrow_1 d^{(2i+1)} \text{gives rise to a single transition with a wildcard. As the only modifications on EPS}(2j, 2i) \text{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)} j^{(x,0)} \rightarrow_1 e^{(2i)} \text{gives rise to (at most) one new transition} (0, N, j, c^{(2i)}) \rightarrow^\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 earlier proposals. We have introduced saturating automata, a new model of computation over infinite alphabets. Unlike previous proposals, 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.

7 Conclusion

We have introduced saturating automata, a new model of computation over infinite alphabets. Unlike earlier proposals, 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.

15
DIXON, MURAWSKI

References


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 $d_1, d_2 \in D$ independent if neither $d_1 = \text{pred}^k(d_2)$ nor $d_2 = \text{pred}^k(d_1)$ for $k \geq 0$.

**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: $d_1 = d_2$, $d_1 = \text{pred}(d_2)$, $d_2 = \text{pred}(d_1)$, or $d_1, d_2$ are independent.

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

Suppose $d_1 = \text{pred}^k(d_2)$. 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 $d_1 = \text{pred}(d_2)$.

Suppose $d_2 = \text{pred}^k(d_1)$. 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 $d_2 = \text{pred}(d_1)$.$ \square$

**Remark A.2** If $d_1, d_2$ are not independent, i.e. $d_1 = \text{pred}^k(d_2)$ or $d_2 = \text{pred}^k(d_1)$, 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 $\text{pred}(d_2) = d_1$ 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 $\text{pred}(d_1) = d_2$ 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 ($t_1 \in \Sigma_Q \land t_2 \in \Sigma_A$ 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 $\beta$-normal $\eta$-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 $c^{(2i+1)} a_O d^{(2i)}_1$ and $c^{(2i+1)} a_O d^{(2i)}_2$ then $d^{(2i)}_1 = d^{(2i)}_2$.
- **PQ** (PQ pre-determinacy): for any $q_p \in \Sigma_{PQ}$, if $c^{(2i)}_1 q_p d^{(2i+1)}$ and $c^{(2i)}_2 q_p d^{(2i+1)}$ then $c^{(2i)}_1 = c^{(2i)}_2$.
- **FA** (final readiness): for every transition of the form $D^{(0)} \xrightarrow{a_p} \uparrow$, 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. $C^{(j)}_i$, $m^*_i$ etc. In contrast, we shall use $C^{(j)}$, $m^*$ to refer to the automaton that is being constructed. The construction will often use inference lines $\xrightarrow{\cdot}$ 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 div$_\theta$).

- **M ⊢ skip :** $k = 0$, $N = 0$, $C^{(0)} = \{0\}$, $\delta$ consists of $\uparrow \text{run} \rightarrow \{0\}$ and $\{0\} \text{ done} \rightarrow \uparrow$.
- **M ⊢ i :** $k = 0$, $N = 0$, $C^{(0)} = \{0\}$, $\delta$ consists of $\uparrow q \rightarrow \{0\}$ and $\{0\} i \rightarrow \uparrow$. 17
• $M \equiv \text{div}_\theta : k = 0, N = 0, C^{(0)} = \{0\}$. Supposing $\theta \equiv \theta_1 \rightarrow \cdots \rightarrow \theta_1 \rightarrow \beta$, recall that $I[\beta]$ stands for the set of initial questions in $[\beta]$. $\delta$ is then given by

$$
\begin{align*}
 x \in I[\beta] \\
 \uparrow \ x \rightarrow \{0\}
\end{align*}
$$

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

• $M \equiv \text{op}(M_1) : k = k_1, N = N_1, C^{(j)} = C_1^{(j)} (0 \leq j \leq k)$. 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.

$$
\frac{D^{(0)} \overset{p}{\longrightarrow} \overset{q}{\longrightarrow} \uparrow_{\text{op}(p)}}{D^{(0)} \overset{\text{op}(q)^*}{\longrightarrow} \uparrow}
$$

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

• $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 $k = \max(k_1, k_2), N = N_1 + N_2, C^{(0)} = C_1^{(0)} + C_2^{(0)} + \{c_1, c_2, \bullet_1, \bullet_2\}, C^{(i)} = C_1^{(i)} + C_2^{(i)} (0 < i \leq k)$, assuming $C^{(i)} = \emptyset$ 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.

$$
\frac{\uparrow_{\text{run}} \{\circ_1, \circ_2\} \quad \uparrow_{\text{run}} \in \{1, 2\}}{\begin{array}{c}
\{\circ_u \in D_u^{(0)} \}
\end{array} \quad \{\circ_u \in D_u^{(0)} \}}
$$

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.

$$
\frac{\{\bullet_v \in D_v^{(0)} \} \quad \{\bullet_v \in D_v^{(0)} \}}{\begin{array}{c}
\{\bullet_v \in D_v^{(0)} \}
\end{array} \quad \{\bullet_v \in D_v^{(0)} \}}
$$

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 $M_2$:

$$
\begin{align*}
(2j, h, u, v, c^{(2i)}) & \xrightarrow{\epsilon} (v', d^{(2i)}) & j > 0 \\
(2j, h, u, v, c^{(2i)}) & \xrightarrow{\epsilon} (v', d^{(2i)}) \\
(0, h_u, v, c^{(2i)}) & \xrightarrow{\epsilon} (v', d^{(2i)}) \\
(0, h_u, v, c^{(2i)}) & \xrightarrow{\epsilon} (v', d^{(2i)})
\end{align*}
$$

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

• $M \equiv M_1; M_2 : \text{com}$

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

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

$$
\frac{\uparrow_{\text{run}} 1_{D_1^{(0)}}}{\downarrow \uparrow_{\text{run}} 1_{D_1^{(0)}}} \quad \frac{\{\circ\} \xrightarrow{\epsilon} \{\circ\}}{\{\circ\} \xrightarrow{\epsilon} \{\circ\}} \quad \frac{\{\circ\} \xrightarrow{\epsilon} \{\circ\}}{\{\circ\} \xrightarrow{\epsilon} \{\circ\}}
$$

The remaining transitions are simply copies of other transitions from $A_{M_1}, A_{M_2}$, with the proviso that in EPS(0, 2j) transitions from $A_{M_2}$ we add $N_1$ to the index of the memory cell that is accessed.
For correctness, we need to appeal to $\text{FA}$ for $M_1$, which tells us that reaching a configuration in which the new state is labelled with $D_1^{(0)}$ amounts to the termination of $M_1$. As before, the construction does not modify transitions relevant to $\text{OA}$, $\text{PQ}$, so the properties are simply inherited from $M_1$ and $M_2$.

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

  \[
  \begin{array}{c}
  \uparrow \text{run}_1 \rightarrow I_1 \rightarrow D_1^{(0)} \\
  \uparrow \text{q} \rightarrow D_1^{(0)} \\
  \end{array}
  \quad
  \begin{array}{c}
  D_1^{(0)} \xrightarrow{0} \rightarrow 1 \\
  D_1^{(0)} \xrightarrow{\epsilon} \{q_0\} \\
  \end{array}
  \quad
  \begin{array}{c}
  \uparrow \text{q} \rightarrow 2 \rightarrow D_2^{(0)} \\
  \{q_0\} \xrightarrow{\epsilon} D_2^{(0)} \\
  \end{array}
  \quad
  \begin{array}{c}
  D_2^{(0)} \xrightarrow{i} \rightarrow 2 \\
  \end{array}
  \]

- **$M \equiv M_1; M_2 : \text{var}$**
  This case is more complicated as there are multiple initial questions and, while $A_{M_1}$ 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 $\text{read}$ (1 in cell $N_1 + N_2 + 1$) or $\text{write}$ ($0$ in cell $N_1 + N_2 + 1$ and $i$ in $N_1 + N_2 + 2$). States of the form $\{q_0, q_x \mid x \in I_{[\text{var}]}\}$ will support this process. Once $A_{M_1}$ terminates, we use states of the form $\{q_0, q_x \mid x \in I_{[\text{var}]}\}$ to access the information from memory before starting $A_{M_2}$.

  Formally, we take $k = \max(k_1, k_2)$, $N = N_1 + N_2 + 2$, $C^{(0)} = C^{(0)}_1 + C^{(0)}_2 + \{q_x, q_x \mid x \in I_{[\text{var}]}\}$, $C^{(i)} = C^{(i)}_1 + C^{(i)}_2 + \{q_x, q_x \mid x \in I_{[\text{var}]}\}$, $C^{(i)} = C^{(i)}_1 + C^{(i)}_2 + \{q_x, q_x \mid x \in I_{[\text{var}]}\}$, $0 < i \leq k$.

  $\text{ADD}(0)$ and $\text{DEL}(0)$ transitions are handled as detailed below, whereas other classes of transitions are copied into the automaton using the same memory-offset procedure.

  \[
  \begin{array}{c}
  x \in I_{[\text{var}]} \\
  \uparrow \epsilon \rightarrow \{q_x\} \\
  \end{array}
  \quad
  \begin{array}{c}
  (0, N_1 + N_2 + 1, 0, \text{read}) \xrightarrow{\epsilon} (1, i) \\
  \end{array}
  \quad
  \begin{array}{c}
  (0, N_1 + N_2 + 2, 0, \text{write}) \xrightarrow{\epsilon} (i, i) \\
  \end{array}
  \quad
  \begin{array}{c}
  \end{array}
  \]

  \[
  \begin{array}{c}
  \{q_0\} \xrightarrow{\epsilon} D_1^{(0)} \\
  \end{array}
  \quad
  \begin{array}{c}
  D_1^{(0)} \xrightarrow{\epsilon} \{q_0\} \\
  \end{array}
  \quad
  \begin{array}{c}
  \{q_0\} \xrightarrow{\epsilon} D_2^{(0)} \\
  \end{array}
  \quad
  \begin{array}{c}
  D_2^{(0)} \xrightarrow{\epsilon} \{q_0\} \\
  \end{array}
  \]

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

- **$M \equiv M_1; M_2 : \text{sem}$**
  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.

- **$M \equiv M_1$ then $M_2$ else $M_3 : \text{com}$**
  This case is similar to $M_1; M_2$. Once $A_{M_1}$ terminates, we must activate $A_{M_2}$ or $A_{M_3}$, depending on the label of the final transition in $A_{M_1}$. Below we describe what modifications are needed for $\text{ADD}(0), \text{DEL}(0)$ rules. We use special states $\text{c}_0, \text{c}_1$ to indicate the value of the guard $M_1$. Formally, $k = \max(k_1, k_2, k_3)$, $N = N_1 + \max(N_2, N_3)$, $C^{(0)} = C^{(0)}_1 + C^{(0)}_2 + C^{(0)}_3 + \{\text{c}_0, \text{c}_1\}$, $C^{(i)} = C^{(i)}_1 + C^{(i)}_2 + C^{(i)}_3$ ($0 < i \leq k$).

  \[
  \begin{array}{c}
  \uparrow \text{q} \rightarrow D_1^{(0)} \\
  \uparrow \text{run}_1 \rightarrow D_1^{(0)} \\
  \end{array}
  \quad
  \begin{array}{c}
  D_1^{(0)} \xrightarrow{0} \rightarrow 1 \\
  D_1^{(0)} \xrightarrow{\epsilon} \{q_0\} \\
  \end{array}
  \quad
  \begin{array}{c}
  \uparrow \text{q} \rightarrow 2 \rightarrow D_2^{(0)} \\
  \{q_0\} \xrightarrow{\epsilon} D_2^{(0)} \\
  \end{array}
  \quad
  \begin{array}{c}
  \uparrow \text{run}_2 \rightarrow D_2^{(0)} \\
  \{q_0\} \xrightarrow{\epsilon} D_2^{(0)} \\
  \end{array}
  \quad
  \begin{array}{c}
  D_2^{(0)} \xrightarrow{i} \rightarrow 2 \\
  \end{array}
  \quad
  \begin{array}{c}
  \end{array}
  \]

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

19
The cases corresponding to \( \beta = \exp, \var, \sem \) are also handled analogously to \( \cdot \). For \( \var, \sem \), it is necessary to remember the initial move.

- **M \equiv \text{while } M_1 \text{ do } M_2**

  In this case we need to run \( A_{M_1} \) and, depending on the final move, direct it to \( A_{M_2} \) or terminate. When \( A_{M_2} \) is about to finish, we redirect to \( A_{M_1} \). 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.

  \( k = \max(k_1, k_2) \), \( N = N_1 + N_2 \), \( C(0) = C_1(0) + C_2(0) + \{e, \bullet\} \), \( C(i) = C_1(i) + C_2(i) (i > 0) \). 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.

  \[
  \begin{array}{c|c|c|c}
  \hline
  & D^{(0)}_1 & D^{(0)}_2 & \text{run} \\
  \hline
  \uparrow a & \downarrow 0 & \downarrow 0 & i > 0 \\
  \downarrow \text{run} & \uparrow \text{run} & \uparrow \text{run} & \uparrow \text{run} \\
  \hline
  \end{array}
  \]

  All transitions from \( A_{M_1} \), \( A_{M_2} \) that are different from ADD(0) and DEL(0) are embedded into the new automaton with the proviso that, in \( E(0, 2v) \) transitions from \( A_{M_2} \), we augment \( h \) by \( N_1 \).

  For correctness, we need to appeal to \( \mathbf{FA} \) for \( M_1, M_2 \): they guarantee that the move to \( A_{M_2} \) happens only after \( A_{M_1} \) is definitely finished and, dually, the move to \( A_{M_1} \) takes place only after \( A_{M_2} \) is finished.

  The construction does not affect transitions relevant to \( \mathbf{OA} \) and \( \mathbf{PQ} \), so they are preserved. By appealing to III, we can conclude that \( \mathbf{FA} \) is preserved too.

- **M \equiv \text{newvar } x \text{ in } M_1 : \beta**

  \( k = k_1, N = N_1 + 1 \), \( C(0) = C_1(0) + \{e, \bullet\} \), \( C(i) = C_1(i) (0 < i \leq k) \). 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 \( A_{M_1} \) 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 \( \beta = \text{com, exp} \), so \( I_{[\beta]} = \{\text{run}\} \) or \( I_{[\beta]} = \{q\} \).

  \[
  \begin{array}{c|c|c|c|c|c|c}
  \hline
  & \{e\} & 0 \leq v \leq \max & \{e\} & D^{(0)}_1 & \{e\} & D^{(0)}_1 \\
  \hline
  x \in I_{[\beta]} & x \rightarrow \{e\} & (0, N, v, e) \xrightarrow{e} (0, \bullet) & x \rightarrow \{e\} & \downarrow \text{run} \rightarrow \text{run} & \downarrow \text{run} \rightarrow \text{run} \\
  \hline
  \end{array}
  \]

  \( c^{(2i)} \xrightarrow{\text{write}} (j, e^{(2i)}) \xrightarrow{\text{read}} (0, N, v, c^{(2i)}) \xrightarrow{e} (\bullet, e^{(2i)}) \xrightarrow{e} (0, N, j, e^{(2i)}) \xrightarrow{e} (j, e^{(2i)}) \xrightarrow{e} (0, N, j, e^{(2i)}), \ x \rightarrow \{e\} \xrightarrow{e} (j, e^{(2i)}), \ x \rightarrow \{e\} \xrightarrow{e} (j, e^{(2i)}), \ x \rightarrow \{e\} \xrightarrow{e} (j, e^{(2i)}), \ x \rightarrow \{e\} \xrightarrow{e} (j, e^{(2i)}) \)

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

  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, \( \mathbf{OA}, \mathbf{PQ} \) are preserved, because no new relevant transitions are introduced. \( \mathbf{FA} \) follows by appealing to III.
• $M \equiv \text{newsem}s \in M_1$

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.

\[
\begin{align*}
&c^{(2)}(1) \xrightarrow{\text{get}(s, \rho)} d^{(2i+1)} \xrightarrow{o_k(s, 0)} \epsilon^{(2i)}(1, e^{(2i)}) \\
&c^{(2)}(2i) \xrightarrow{\text{rel}(s, \rho)} d^{(2i+1)} \xrightarrow{o_k(s, 0)} \epsilon^{(2i)}(0, e^{(2i)})
\end{align*}
\]

• $M \equiv fM_1 \cdots M_1$

Suppose $\Gamma \vdash fM_1 \cdots M_1 : \beta$ with $(f : \theta_1 \to \cdots \to \theta_1 \to \beta) \in \Gamma$. Given $q \in I_{[\beta]}$, we write $A_q$ for the set of corresponding answers.

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

\[
\begin{align*}
C^{(0)} &= \{0_q | q \in I_{[\beta]}\} + \{0| q \in I_{[\beta]}, a \in A_q\} \\
C^{(1)} &= \{1_q | q \in I_{[\beta]}\} \\
C^{(j+2)} &= \sum_{i=1}^{l} C^{(j)}_i \ (0 \leq j \leq k).
\end{align*}
\]

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

\[
\begin{align*}
\uparrow q &\xrightarrow{q'} \{0_q\} & q &\xrightarrow{q} 1_q & q &\xrightarrow{a} \{0| a \} \\
0_q &\xrightarrow{q} 1_q & a &\xrightarrow{q} 0_a
\end{align*}
\]

Observe that these transitions satisfy OA, PQ and FA.

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

- Moves from $M_u$ corresponding to $\theta_u$ obtain an additional annotation $fu$, as they are now the $u$th argument of $f : \theta_l \to \cdots \to \theta_1 \to \beta$.

\[
\begin{align*}
\uparrow m^{(\epsilon, 0)} &\xrightarrow{D_u^{(0)}} D_u^{(0)} & D_u^{(0)} &\xrightarrow{\epsilon} \uparrow \\
1_q &\xrightarrow{m^{(\epsilon, 0)}} D_u^{(0)} & m^{(\epsilon, 0)} &\xrightarrow{\epsilon} D_u^{(0)} & D_u^{(0)} &\xrightarrow{m^{(\epsilon, 0)}} \uparrow
\end{align*}
\]

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 $m^{(\epsilon, 0)}$ points one level up, i.e. at the $q^f$ move, as required.

- Moves from $M_u$ that originate from $\Gamma$, i.e. moves of the form $m^{(x, \epsilon, \rho)}$, where $(x, \epsilon, \rho) \in \Gamma$, need no relabelling except for question-moves $m^{(x, \epsilon, \rho)}$ that need to point at the initial move $m^{(\epsilon, 0)}$. Leaving $\rho$ unchanged in this case would mean pointing at $m^{(\epsilon, 0)}$, whereas we need to point at $m^{(\epsilon, 0)}$ instead.

To readjust such pointers, we simply add 2 to $\rho$ in the relevant moves, and preserve $\rho$ in other moves.

\[
\begin{align*}
\Box_L m^{(x, \epsilon, \rho)} &\xrightarrow{\epsilon} D_u^{(0)} & m \text{ is a question} \\
\Box_L m^{(x, \epsilon, \rho+2)} &\xrightarrow{\epsilon} D_u^{(0)} & \Box_L m^{(x, \epsilon, \rho)} &\xrightarrow{\epsilon} D_u^{(0)} & \epsilon \neq \epsilon \text{ or } (\epsilon = \epsilon \text{ and } m \text{ is an answer})
\end{align*}
\]

- 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.

\[
\begin{align*}
(2j, h, v, c^{(2i)}) &\xrightarrow{\epsilon} (u', d^{(2i)}) \\
(2j + 2, h, v, c^{(2i)}) &\xrightarrow{\epsilon} (u', d^{(2i)})
\end{align*}
\]
The preservation of \( \text{OA} \) and \( \text{PQ} \) follows from the construction and IH, as the old transitions are simply copied in and relabelled injectively.

- \( M \equiv \lambda x. M_1 : \theta_1 \to \cdots \to \theta_1 \to \beta \)

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

- \( M \equiv \mathbf{M}_1 \) : Here we need to perform the same transitions as the automaton for \( M_1 \) would when started from \( \text{read} \) except that \( \text{read} \) needs to be relabelled to \( \theta \). Consequently, it suffices to use all transitions from the automaton for \( M_1 \) except \( \text{ADD}(0) \), which needs to be modified as follows.

  \[
  \begin{array}{c}
  \text{read} : \uparrow^1 \downarrow^1 \\
  \theta : \uparrow^0 \downarrow^0 \\
  \end{array}
  \]

  Clearly, this preserves \( \text{OA}, \text{PQ} \) and \( \text{FA} \).

- \( M \equiv \mathbf{M}_1 \equiv \mathbf{M}_2 \)

  This case is similar to \( \vdash \). 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 \( \text{write}(i) \) was played.

  \[ k = \max(k_1, k_2), \ N = N_1 + N_2, \ C^{(0)} = C^{(0)}_1 + \{ q_i | 0 \leq i \leq \text{max} \}, \ C^{(i)} = C^{(i)}_1 + (0 < i \leq k) \]

  \[
  \begin{array}{c}
  \uparrow^q \downarrow^q \uparrow^2 \downarrow^2 \\
  \uparrow^\text{run} \downarrow^\text{run} \\
  \end{array}
  \]

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

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

  \[
  \begin{array}{c}
  \text{grb} : \uparrow^\text{grb} \downarrow^\text{grb} \\
  \text{run} : \uparrow^\text{run} \downarrow^\text{run} \\
  \end{array}
  \]

  Clearly, \( \text{OA}, \text{PQ} \) and \( \text{FA} \) are inherited in this case.

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

  This case is the same as the previous one but \( \text{rls} \) should be used instead of \( \text{grb} \).

**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.