Verifying higher-order concurrency with data automata

Alex Dixon*, Ranko Lazić*, Andrzej S. Murawski† and Igor Walukiewicz‡

*Department of Computer Science, University of Warwick
†Department of Computer Science, University of Oxford
‡CNRS, Université de Bordeaux

***Abstract*—Using a combination of automata-theoretic and game-semantic techniques, we propose a method for analysing higher-order concurrent programs. Our language of choice is Finitary Idealised Concurrent Algol (FICA) due to its relatively simple fully abstract game model.**

Our first contribution is an automata model over a tree-structured infinite data alphabet, called *split automata*, whose distinctive feature is the separation of control and memory. We show that every FICA term can be translated into such an automaton. Thanks to the structure of split automata, we are able to observe subtle aspects of the underlying game semantics.

This enables us to identify a fragment of FICA with iteration and limited synchronisation (but without recursion), for which, in contrast to the whole FICA, a variety of verification problems turn out to be decidable.

I. Introduction

We investigate how to use game semantics for verification of concurrent higher-order programs. Game semantics makes it possible to investigate properties such as "two programs are equivalent in all contexts", or "there is a context where a program behaves in a particular way". The quantification over contexts offered by game semantics is particularly interesting for concurrent programs when we want to understand the possible behaviours of a program in an environment we do not control. Our research objective is to find effective ways of using game semantics to analyse programming languages with concurrency.

The use of game semantics comes with a price. It uses sequences with pointers, which is a major obstacle to applying standard automata-theoretic techniques. In some cases, pointers can be ignored or encoded in the structure of sequences. In the concurrent setting, where closure under shuffle is hardly avoidable, such approaches do not seem promising.

We use data to encode pointers: roughly, two positions with related data are linked by a pointer. This approach requires using data automata to describe the semantics of programs. The challenge is to find a sufficiently expressive class of data automata with good algorithmic properties.

We propose a model of data automata, called *split automata*, expressive enough to encode the game semantics of Finitary Idealised Concurrent Algol (FICA). We identify a subclass of these automata for which the emptiness problem is decidable. We show that this subclass corresponds to restricting the use of semaphores in FICA.

978-1-6654-4895-6/21/$31.00

Finitary Idealised Concurrent Algol is a prototypical programming language combining functional, imperative, and concurrent computation. It is a call-by-name language with higher-order features, side-effects, and concurrency implemented by a parallel composition operator and semaphores. It is finitary since, as is common in this context, base types are restricted to finite domains. The fully abstract game semantics of FICA is relatively simple [1], making it an appealing candidate for our study.

Split automata work with data values organised in an infinite tree. The data values represent occurrences of moves and the tree structure helps to capture nesting dependencies implied by pointers. The automata accept data words, which we use to represent plays in game semantics. We show how to translate a FICA term to a split automaton accepting precisely the representations of plays in the semantics of the term. Due to this close connection, the emptiness problem is undecidable for split automata, as the corresponding problem for FICA is undecidable [2].

Nevertheless, the structure of split automata leads us to uncover a decidable fragment of FICA. The name, split automata, emphasises their structure, where the control and memory used to store values of the program are separated. This separation allows us to discover a restriction on the use of semaphores that makes automatic verification possible. In *restricted-semaphore* FICA (rsFICA), subterms of the form $fM_1 \dots M_l$ cannot contain free semaphore variables. Intuitively, this means that semaphore variables cannot be passed to unspecified functions. On the other hand, we put no restrictions on the scope of memory variables, and we allow the iteration construct.

At the level of split automata, the decidable restriction takes the form of an idempotency requirement on transitions. It captures in a succinct way the difference between read/write operations and grab/release operations. We prove that the emptiness problem for idempotent automata is decidable. Technically, the argument makes an interesting link with parametric verification.

Several verification problems become decidable thanks to the decidability result and the reduction from rsFICA to idempotent automata. Unlike standard program analysis, we do not address the problem of verifying if a property holds

for all executions, but rather whether the property holds for all executions in all possible contexts. It is game semantics that permits us to handle the additional quantification over all possible contexts. For example, we can decide if there is a context where a given variable in a given program is assigned value 13, or dually, it is never assigned value 13. The two questions are different because one uses existential and the other universal quantification on contexts and executions. Since the method uses finite automata to express properties, we can also express standard questions from program analysis, such as "is a given variable live at a given program point?" or "is a given variable invariant in some loop?".

Related work: Concurrency, even with only first-order recursion, leads to undecidability [3]. The first decidable fragment of FICA is Syntactic Control of Concurrency (SCC) [2]. It imposes bounds on the number of threads in which arguments can be used. This restriction makes it possible to represent the game semantics of programs by finite automata.

Another very recent fragment is local FICA [4] (LFICA). It forbids **while**, and requires the binding between declaration and use of any variable or semaphore to "cross" at most one free identifier. The decidability argument for LFICA starts from an automata model related to split automata. However, the latter are more refined, allowing for a more direct translation of FICA into automata. Moreover, the decidability arguments are completely different in the two cases. The one from [4] uses a reduction to reachability in Petri Nets. Our fragment, rsFICA, does not encompass LFICA, because the latter allows the use of semaphores at (applicative) depth 2. In rsFICA we have unbounded computations thanks to **while**, and unrestricted use of state, making it a more natural fragment than LFICA.

Our decidability argument relies on the observation that, with limited use of semaphores, a program cannot tell how many copies of code it interacts with. This situation is well known in parametric verification [5], [6], [7], [8], [9], [10]. For example, the above-cited undecidability argument [3] is based on the fact that two communicating pushdown automata can simulate a Turing machine. When one of the two pushdown automata is duplicated an indeterminate number of times, and communication uses reads and writes to shared variables, the model becomes decidable [6]. In our model, parameterisation is not postulated in order to get decidability but it occurs naturally in the associated game semantics; intuitively, it comes from the quantification over all contexts.

Split automata are a model of computation over an infinite alphabet. Models of this kind have been researched intensively in recent years, not least due to connections with database theory, notably XML [11]. Nested data were first considered in [12], where the authors discuss shuffle expressions. After that, data automata [13] and class memory automata [14] have been adapted to nested data in [15], [16]. For most models over nested data, the emptiness problem is undecidable. To achieve decidability, the authors in [15], [16] relax the acceptance conditions so that the emptiness problem can eventually be recast as a coverability problem for a well-structured transition system. In [17], this result was used to show decidability of equivalence for a first-order (sequential) fragment of Reduced ML. On the other hand, in [12] the authors relax the order of letters in words, which leads to an analysis based on semi-linear sets. Both of these restrictions are too strong to represent the semantics of FICA, because of the game-semantic WAIT condition, which corresponds to waiting until all subprocesses terminate. In split automata, this is reflected by making answers conditional on the success of zero tests.

Yet another related strand of work on concurrent higher-order programs is based on higher-order recursion schemes [18], [19]. Unlike FICA, they feature recursion but the computation is purely functional over a single atomic type o.

Structure of the paper: We start with a presentation of FICA and its game semantics. Next, we introduce split automata. In section V we describe how plays are represented by data words, and how to translate FICA terms to split automata. In Section VI we introduce rsFICA, and show that split automata obtained from rsFICA terms are idempotent. In the following section we present the decidability of the emptiness problem for idempotent automata. In section VIII we describe how some verification problems for rsFICA can be effectively reduced to the emptiness problem for idempotent automata.

II. Finitary Idealised Concurrent Algol (FICA)

Idealised Concurrent Algol [1] is a paradigmatic language combining higher-order with imperative computation in the style of Reynolds [20], extended to concurrency with parallel composition ($||$) and binary semaphores. We consider its finitary variant, FICA, where the datatype is finite $\{0, \ldots, max\}$ ($max \geqslant 0$), there is no recursion, but there is iteration instead. Its types θ are generated by the grammar

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

where $\mathbf{com}$ is the type of commands; $\mathbf{exp}$ that of $\{0, \ldots, max\}$-valued expressions; $\mathbf{var}$ that of assignable variables; and $\mathbf{sem}$ that of semaphores. The typing judgments are displayed in Figure 1. Here, $\mathbf{skip}$ and $\mathbf{div}_\theta$ are constants representing termination and divergence respectively, i ranges over $\{0, \ldots, max\}$, and $\mathbf{op}$ 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 $\mathbf{newvar}$ and $\mathbf{newsem}$. 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. A term $\vdash M : \mathbf{com}$ is said to terminate, written $M \Downarrow$, if there exists a terminating evaluation sequence from M to $\mathbf{skip}$.

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 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, $\mathcal{C}[M_1] \Downarrow$ if and only if $\mathcal{C}[M_2] \Downarrow$. Due to quantification over all contexts, even very simple instances of equivalence, like equivalence with $\mathbf{div}_\theta$, are undecidable. Intuitively, to show inequivalence of

$$\frac{}{\Gamma \vdash \mathbf{skip} : \mathbf{com}} \qquad \frac{}{\Gamma \vdash \mathbf{div}_\theta : \theta} \qquad \frac{}{\Gamma \vdash i : \mathbf{exp}} \qquad \frac{\Gamma \vdash M : \mathbf{exp}}{\Gamma \vdash \mathbf{op}(M) : \mathbf{exp}} \qquad \frac{\Gamma \vdash M : \mathbf{com} \qquad \Gamma \vdash N : \beta}{\Gamma \vdash M; N : \beta}$$

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

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

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

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

Fig. 1. FICA typing rules

a term $\Gamma \vdash M : \theta$ with $\mathbf{div}_\theta$, we need to find a terminating interaction of M with a context. Using game semantics, this can be reduced to the existence of a so-called complete play. If one can then find a class of automata to represent such plays, this can be further reduced to an emptiness problem.

Example 1. In Figure 2, we give two examples of problematic shapes of FICA terms, where s, x stands for a code fragment that uses the variable x and semaphore s. Using the methodology of [2], it is possible to construct terms of this form to represent two-counter machines. The terms then represent such machines in the sense that inequivalence with $\mathbf{div}$ coincides with the halting problem. The expressive power comes from the use of the free identifier $f : \mathbf{com} \to \mathbf{com}$, corresponding to an unspecified procedure, which can investigate its argument in an unbounded number of concurrent threads. We write $M_1 + M_2$ for non-deterministic choice, which can be coded in FICA as $\mathbf{newvar}\, x\, \mathbf{in}\, ((x := 0 || x := 1); \mathbf{if}\, !x\, \mathbf{then}\, M_1\, \mathbf{else}\, M_2)$, so the loop corresponds to an arbitrary number of iterations.

In this paper, we will identify a class of terms, called *restricted-semaphore* FICA (rsFICA), for which equivalence with $\mathbf{div}_\theta$ will turn out to be decidable. We will also show how to use the decidability procedure to verify properties of potential interactions of rsFICA terms with contexts. rsFICA will allow for iteration ($\mathbf{while}$) and unrestricted use of assignable variables, but forbid use of semaphores in subterms of the form $fM_1 \cdots M_k$. Consequently, the *overlined* uses of semaphores in Figure 2 will be illegal in rsFICA.

On the other hand, in local FICA [4], which is another recently established decidable fragment of FICA, $\mathbf{while}$ is completely forbidden, and the use of both semaphores and variables is restricted in such a way that the *underlined* parts are banned.

III. Game semantics

In this section, we briefly present the fully abstract game model for FICA from [1], 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 2. 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
$[\![\mathbf{com}]\!]$	run	done
$[\![\mathbf{var}]\!]$	read	i
	write(i)	ok

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

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 4, we give (the enabling relations of) $A_1 = [\![\mathbf{com} \to \mathbf{com} \to \mathbf{com}]\!]$ and $A_2 = [\![(\mathbf{var} \to \mathbf{com}) \to \mathbf{com}]\!]$ respectively, using superscripts to distinguish copies of the same move (the use of superscripts is consistent with our future use of tags in Definition 10).

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

$$f : \mathbf{com} \to \mathbf{com} \;\vdash\; \mathbf{newvar}\, x\, \mathbf{in}\, \mathbf{newsem}\, s\, \mathbf{in}\; \underline{\mathbf{while}}\,(0+1)\,\mathbf{do}\,(\,f(\,\overline{s},x\,);\,s,x\,)\;\|\;\underline{\mathbf{while}}\,(0+1)\,\mathbf{do}\,(\,f(\,\overline{s},x\,);\,s,x\,)$$

$$f : \mathbf{com} \to \mathbf{com} \;\vdash\; \mathbf{newvar}\, x\, \mathbf{in}\, \mathbf{newsem}\, s, s_1, s_2\, \mathbf{in}\; f(\,\mathbf{grab}(\overline{s_1});\, f(\,\underline{\overline{s}},\underline{x}\,);\, \overline{s},x;\, \mathbf{release}(\overline{s_1})\,)\;\|\;f(\,\mathbf{grab}(\overline{s_2});\, f(\,\underline{\overline{s}},\underline{x}\,);\, \overline{s},x;\, \mathbf{release}(\overline{s_2}))$$

Fig. 2. Problematic FICA terms

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

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

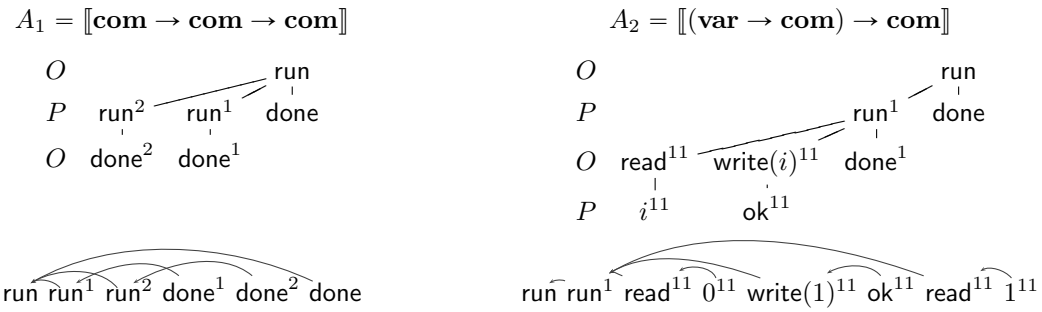


Fig. 4. Arenas and justified sequences

question does not have an answer in a justified sequence, we say that it is *pending* in that sequence. In Figure 4 we give two justified sequences from A_1 and A_2 respectively.

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. 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 $s' = \cdots q \overset{\frown}{\cdots} m$ of s, the question q must be pending when m is played.

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

It is easy to check that the justified sequences given above are plays. 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 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 [1] 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 $[\![\Gamma \vdash \mathbf{skip}]\!] = \{\epsilon, \mathsf{run}, \mathsf{run} \overset{\frown}{\mathsf{done}}\}$. 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. 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 5 ([1])**.** *We have* $\Gamma \vdash M_1 \cong M_2$ *if and only if* $\mathsf{comp}([\![\Gamma \vdash M_1]\!]) = \mathsf{comp}([\![\Gamma \vdash M_2]\!])$.

In particular, since $\mathsf{comp}([\![\Gamma \vdash \mathbf{div}_\theta]\!]) = \varnothing$, we have that $\Gamma \vdash M : \theta$ is contextually equivalent to $\mathbf{div}_\theta$ if and only if $\mathsf{comp}([\![\Gamma \vdash M]\!]) = \varnothing$.

IV. Split automata

We propose a class of automata recognising sequences of tags with data values. Every letter in a sequence is a pair from $\Sigma \times \mathcal{D}$, where Σ is a finite set of tags, and $\mathcal{D}$ is an infinite set of data values. Tags will represent moves in game semantics, and data values will encode pointers. We shall show that the automata are expressive enough to express the game semantics of FICA. Namely, for every term we shall be able to construct an automaton accepting a representation of the set of plays corresponding to the term.

Because the automata separate control states from memory, we call them *split automata*. This structure will allow us to

understand how a term accesses the memory cells. This will enable us to identify a decidable fragment of FICA.

Our dataset $\mathcal{D}$ has the structure of a countably infinite forest. This structure will be instrumental for representing game semantics, specifically to encode justification pointers and to enforce the WAIT condition.

Definition 6 (Dataset)**.** $\mathcal{D}$ is a countably infinite set equipped with a function $pred : \mathcal{D} \to \mathcal{D} \cup \{\bot\}$ (the *parent* function) such that the following conditions hold.

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

A configuration of a split automaton is a finite subtree of $\mathcal{D}$ labelled with states (consisting of a control state with zero or more memory cells). We say that $T \subseteq \mathcal{D}$ is a subtree of $\mathcal{D}$ if and only if T is closed ($\forall x \in T\colon pred(x) \in T \cup \{\bot\}$) and rooted ($\exists! x \in T\colon pred(x) = \bot$). The automaton can add or remove leaves from its configuration. When doing so, it can only refer to the control state at the parent level. Moreover, it has transitions that do not modify the shape of its configuration but manipulate the control state of a node and the memory content of one of its ancestors at the same time.

A split automaton has two parameters (k, 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 each node. The set of control states of the automaton is partitioned into sets $C^{(i)}$, for $0 \leqslant i \leqslant k$.

Data at odd levels will be labelled only with control states, which will not change during runs. Data at even levels will be labelled with control states (which may change during runs), as well as with memory stores consisting of N cells, each storing an element from $V = \{0, \dots, max\}$. Accordingly, in configurations, even-level data will be labelled with elements of $C^{(i)} \times V^N$.

Definition 7. A *split automaton* (SA) is a tuple $\mathcal{A} = \langle \Sigma, k, N, C, \delta \rangle$, where:

- $\Sigma = \Sigma_{\mathsf{Q}} + \Sigma_{\mathsf{A}}$ is a finite alphabet, partitioned into questions and answers;
- $k \geqslant 0$ is the depth parameter;
- $N \geqslant 0$ is the local memory capacity;
- $C = \bigcup\{C^{(i)} : i = 0, \dots, k\}$ 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 and level on which they operate (below $c^{(i)}, d^{(i)} \in C^{(i)}$):
  - ADD(i) transitions are $c^{(i-1)} \xrightarrow{q} (d^{(i-1)}, d^{(i)})$, and $\dagger \xrightarrow{q} c^{(0)}$ for the special case of $i = 0$, with $q \in \Sigma_{\mathsf{Q}}$,
  - DEL(i) transitions are $(c^{(i-1)}, c^{(i)}) \xrightarrow{a} d^{(i-1)}$, and $c^{(0)} \xrightarrow{a} \dagger$ for the special case of $i = 0$, with $a \in \Sigma_{\mathsf{A}}$,
  - EPS($2j, 2i$) transitions read $v \in V$ from memory cell $h \in \{1, \dots, N\}$ at level $2j \leqslant 2i$ and update it to $v' \in V$, but do not read the input: $(2j, h, v, c^{(2i)}) \xrightarrow{\epsilon} (v', d^{(2i)})$.

Transitions cannot modify control states at odd layers: if $c^{(2i-1)} \xrightarrow{q} (d^{(2i-1)}, d^{(2i)}) \in \delta$ or $(c^{(2i-1)}, c^{(2i)}) \xrightarrow{a} d^{(2i-1)} \in \delta$ then $c^{(2i-1)} = d^{(2i-1)}$.

A *configuration* of a split automaton 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), $f : E \to C$ is a level-preserving function, i.e. if d is a level-i data value then $f(d) \in C^{(i)}$, and $m : E \rightharpoonup V^N$ is a partial function whose domain is the even-level nodes of E.

A split automaton $\mathcal{A}$ starts from the empty configuration $\kappa_0 = (\varnothing, \varnothing, \varnothing, \varnothing)$ and proceeds according to its transitions δ as explained below. Let the current configuration be $\kappa = (D, E, f, m)$.

An ADD transition from κ is possible on a letter (t, d) when $t = q \in \Sigma_{\mathsf{Q}}$ and $d \notin D$ is a fresh level-i datum such that $pred(d) \in E$ (the parent of d is in the configuration). In this case, the automaton adds a new leaf d to the configuration and updates the control state. The new leaf gets the control state determined by the transition, moreover if it is on an even level its memory is initialised. Formally, $\mathcal{A}$ goes from κ to $\kappa' = (D \cup \{d\}, E \cup \{d\}, f', m')$ provided one of the following conditions holds:

- On transition $c^{(i-1)} \xrightarrow{q} (d^{(i-1)}, d^{(i)})$ when $f(pred(d)) = c^{(i-1)}$, $f' = f[pred(d) \mapsto d^{(i-1)}, d \mapsto d^{(i)}]$, and $m' = m$ (if i is odd) or $m' = m[d \mapsto 0^N]$ (if i is even).
- On transition $\dagger \xrightarrow{q} d^{(0)}$ when $D = \varnothing$, $f' = [d \mapsto d^{(0)}]$, and $m' = [d \mapsto 0^N]$.

We write $f[\cdots]$ to extend or update f.

On reading a letter (t, d) with $t = a \in \Sigma_{\mathsf{A}}$ and $d \in E$ a level-i datum, a transition is possible only if d is a leaf in E. A DEL transition deletes d and updates the neighbouring control state without modifying the associated memory (if any). Formally, the automaton changes its configuration to $\kappa' = (D, E\backslash\{d\}, f'\backslash\{d\}, m\backslash\{d\})$ provided one of the following conditions holds:

- On transition $(c^{(i-1)}, c^{(i)}) \xrightarrow{a} d^{(i)}$ when $f(pred(d)) = c^{(i-1)}$, $f(d) = c^{(i)}$, and $f' = f[pred(d) \mapsto c^{(i-1)}]$.
- On transition $c^{(0)} \xrightarrow{a} \dagger$ when $f' = f$.

Observe that the last transition is possible only when d is a leaf of E and at level 0 at the same time, the result of the transition is the empty tree E.

Transitions from EPS are silent. They apply at even levels only and do not modify the shape of configurations. However, they may change the associated control state and read/write one memory location situated at the same level or another even level above. Formally, the automaton can go to $\kappa' = (D, E, f', m')$ on transition $(2j, h, v, c^{(2i)}) \xrightarrow{\varepsilon} (v', d^{(2i)})$ if there is a level-$2i$ datum $d \in E$ such that $m(pred^{2i-2j}(d))(h) = v$, $f(d) = c^{(2i)}$, $f' = f[d \mapsto d^{(2i)}]$ and $m'(pred^{2i-2j}(d)))(h) = v'$, and m' is the same as m otherwise.

Definition 8. A *trace* of a split automaton $\mathcal{A}$ is a word $w \in (\Sigma \times \mathcal{D})^*$ such that $\kappa_0 \xrightarrow{l_1} \kappa_1 \dots \kappa_{h-1} \xrightarrow{l_h} \kappa_h$, 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 $Tr(\mathcal{A})$ (resp. $L(\mathcal{A})$).

Remark 9. A related model, called *leafy automata*, was recently proposed in [4]. Leafy automata can modify all states along the relevant branch in a single step. Split automata are more constrained in that regard. Their odd levels never change, and transitions have more restricted access to information on the branch: control states can be modified only up to one level up and, at memory access, control states lying strictly above cannot be accessed at all. This more refined structure is crucial to identifying a decidable family within split automata.

Although split automata appear more restrictive, we will next show that they can still express the game semantics of FICA. Indeed, all the restrictions are motivated by a good fit with the translation. Thus, split automata are also able to accommodate the semantics of the undecidable terms discussed in Section II. Consequently, the associated emptiness problem must be undecidable.

V. From FICA to split automata

We present a translation from FICA to split automata. For this, we first describe how we encode pointers in plays using both a special indexing scheme and data. Then we present the translation that proceeds by induction on term structure.

Recall from Section III that, to interpret base types, game semantics uses moves from the set

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

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 $\vec{i} \in \mathbb{N}^*$ and $\rho \in \mathbb{N}$:

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

The annotated moves will be written as $m^{(\vec{i},\rho)}$ or $m^{(x_v\vec{i},\rho)}$, where $m \in \mathcal{M}$. We will sometimes omit ρ on the understanding that this represents $\rho = 0$. Similarly, when $\vec{i}$ 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 $\vec{i}$ superscripts are linked to moves from $[\![\theta]\!]$. Given $X \subseteq \{m^{(\vec{i},\rho)} \mid \vec{i} \in \mathbb{N}^*,\ \rho \in \mathbb{N}\}$ and $y \in \mathbb{N} \cup \{x_1, \cdots, x_l\}$, we let $yX = \{m^{(y\vec{i},\rho)} \mid m^{(\vec{i},\rho)} \in X\}$.

Definition 10. Given a type θ, the corresponding alphabet $\mathcal{T}_\theta$ is defined as follows

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

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

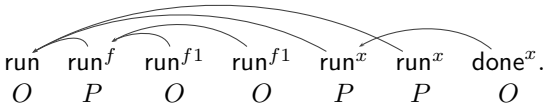
For example, $\mathcal{T}_{f:\mathbf{com}\to\mathbf{com}, x:\mathbf{com} \vdash \mathbf{com}}$ is $\{\ \mathsf{run}^{(f1,\rho)}, \mathsf{done}^{(f1,\rho)}, \mathsf{run}^{(f,\rho)}, \mathsf{done}^{(f,\rho)}, \mathsf{run}^{(x,\rho)}, \mathsf{done}^{(x,\rho)}, \mathsf{run}^{(\epsilon,\rho)}, \mathsf{done}^{(\epsilon,\rho)} \mid \rho \in \mathbb{N}\,\}$.

To represent the game semantics of terms-in-context $\Gamma \vdash M : \theta$, we shall use *finite subsets* of $\mathcal{T}_{\Gamma \vdash \theta}$ as alphabets. They will be finite because ρ will be bounded. Note that $\mathcal{T}_{\Gamma \vdash \theta}$ admits a natural partitioning into questions and answers, depending on whether the underlying move is a question or an answer.

We will represent plays using data words in which the underlying sequence of tags comes from (a finite subset of) the alphabet defined above. Next we explain how superscripts and data are 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 $pred^{\rho+1}(d)$.

Example 11. Suppose that $d_0 = pred(d_1), d_1 = pred(d_2) = pred(d_2'), d_2 = pred(d_3)$, $d_2' = pred(d_3')$. Then the data word $(\mathsf{run}^{(\epsilon,0)}, d_0)$ $(\mathsf{run}^{(f,0)}, d_1)$ $(\mathsf{run}^{(f1,0)}, d_2)$ $(\mathsf{run}^{(f1,0)}, d_2')$ $(\mathsf{run}^{(x,2)}, d_3)$ $(\mathsf{run}^{(x,2)}, d_3')$ $(\mathsf{done}^{(x,0)}, d_3)$ represents the play



Note that a play may have several different representations; the last three moves of the above play could also be represented by $(\mathsf{run}^{(x,0)}, d_1')$ $(\mathsf{run}^{(x,0)}, d_1'')$ $(\mathsf{done}^{(x,0)}, d_1')$, with $pred(d_1') = pred(d_1'') = d_0$.

Example 12. Consider the SA $\mathcal{A} = \langle Q, 3, 0, \Sigma, \delta \rangle$, where $Q^{(0)} = \{0, 1, 2\}$, $Q^{(1)} = \{3\}$, $Q^{(2)} = \{4, 5, 6\}$, $Q^{(3)} = \{7\}$, $\Sigma_{\mathsf{Q}} = \{\mathsf{run}^{(\epsilon,0)}, \mathsf{run}^{(f,0)}, \mathsf{run}^{(f1,0)}, \mathsf{run}^{(x,2)}\}$, $\Sigma_{\mathsf{A}} = \{\mathsf{done}^{(\epsilon,0)}, \mathsf{done}^{(f,0)}, \mathsf{done}^{(f1,0)}, \mathsf{done}^{(x,0)}\}$, and δ is given by

$$\dagger \xrightarrow{\mathsf{run}^{(\epsilon,0)}} 0 \qquad 0 \xrightarrow{\mathsf{run}^{(f,0)}} (1,3) \qquad (1,3) \xrightarrow{\mathsf{done}^{(f,0)}} 2$$
$$2 \xrightarrow{\mathsf{done}^{(\epsilon,0)}} \dagger \qquad 3 \xrightarrow{\mathsf{run}^{(f1,0)}} (3,4) \qquad 4 \xrightarrow{\mathsf{run}^{(x,2)}} (5,7)$$
$$(5,7) \xrightarrow{\mathsf{done}^{(x,0)}} 6 \qquad (3,6) \xrightarrow{\mathsf{done}^{(f1,0)}} 3$$

Then traces from $Tr(\mathcal{A})$ represent all plays from $\sigma = [\![f : \mathbf{com} \to \mathbf{com}, x : \mathbf{com} \vdash fx : \mathbf{com}]\!]$, including the play from Example 11, and $L(\mathcal{A})$ represents $\mathsf{comp}(\sigma)$.

One may wonder why we did not choose to use the parent structure of $\mathcal{D}$ to represent justification pointers (this would correspond to $\rho = 0$ in all cases). Unfortunately, this simplified scheme would not work with split automata: in the above example, the number of $\mathsf{run}^{(x,2)}$ moves has to be the same as the number of $\mathsf{run}^{(f1,0)}$ moves. If we used level-1 data values for run^x, we would not be able to use the automaton to enforce this property.

Below we state the main result linking FICA with split automata. Question-moves in this translation are handled with ADD transitions (at even levels for O and odd levels for P). Answer-moves are handled by DEL transitions (at odd levels for O and even levels for P). The structure of split automata makes it possible to decouple the interpretation of memory-related operations from the rest.

Theorem 13. *For any* FICA *term* $\Gamma \vdash M : \theta$*, there exists a split automaton* $\mathcal{A}_M$ *over a finite subset of* $\mathcal{T}_{\Gamma\vdash\theta}$ *such that the set of plays represented by data words from* $Tr(\mathcal{A}_M)$ *is exactly* $[\![\Gamma \vdash M : \theta]\!]$*. Moreover,* $L(\mathcal{A}_M)$ *represents* $\mathsf{comp}([\![\Gamma \vdash M : \theta]\!])$.

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. $Q_i^{(j)}$, $\xrightarrow{m}_i$ etc. In contrast, $Q^{(j)}$, $\xrightarrow{m}$ 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 via the inductive hypothesis. Below we discuss the most interesting cases, giving several representative steps in each case.

$(\mathbf{M = M_1 || M_2})$ To model interleaving, we will use pairs of level-0 control states from both automata with memory big enough to accommodate both automata. We take $k = \max(k_1, k_2)$, $N = N_1 + N_2$, $C^{(0)} = C_1^{(0)} \times C_2^{(0)}$ and $C^{(i)} = C_1^{(i)} + C_2^{(i)}$ $(i > 0)$. All it takes then is to embed transitions from $\mathcal{A}_{M_1}$ and $\mathcal{A}_{M_2}$ suitably.

ADD and DEL transitions that can assess level-0 control need to preserve the control state of the other component, e.g. $\dfrac{c_1^{(0)} \xrightarrow{\ell}_1 (d_1^{(0)}, d_1^{(1)}) \qquad c \in C_2^{(0)}}{(c_1^{(0)}, c) \xrightarrow{\ell} ((d_1^{(0)}, c), d_1^{(1)})}$. EPS transitions from M_2 need to be adjusted (by adding N_1) so that their refer to the memory dedicated to M_2, e.g. $\dfrac{c \in C_1^{(0)} \qquad (0, h, v, c^{(0)}) \xrightarrow{\epsilon}_2 (v', d^{(0)})}{(0, N_1{+}h, v, (c, c^{(0)})) \xrightarrow{\epsilon} (v', (c, d^{(0)}))}$. Finally, at the start and at the end the automata need to be synchronised: $\dfrac{\dagger \xrightarrow{\mathsf{run}}_1 c_1^{(0)} \qquad \dagger \xrightarrow{\mathsf{run}}_2 c_2^{(0)}}{\dagger \xrightarrow{\mathsf{run}} (c_1^{(0)}, c_2^{(0)})}$, $\dfrac{c_1^{(0)} \xrightarrow{\mathsf{done}}_1 \dagger \qquad c_2^{(0)} \xrightarrow{\mathsf{done}}_2 \dagger}{(c_1^{(0)}, c_2^{(0)}) \xrightarrow{\mathsf{done}} \dagger}$.

$(\mathbf{M = newvar\, x\, in\, M_1})$ According to [1], 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 will add a memory cell at level 0 to keep track of the current value of x. To this end, we take $k = k_1$, $N = N_1 + 1$, $C^{(i)} = C_1^{(i)}$ $(0 \leqslant i \leqslant k)$.

Transitions that do not use the moves discussed above can be copied into the new automaton without changes. The remaining transitions are added according to the following rules. Note that in the case for writing we add transitions for *every* old value v.

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

$$\frac{c^{(2i)} \xrightarrow{\mathsf{read}^{(x,\rho)}}_1 (d^{(2i)}, d^{(2i+1)}) \xrightarrow{j^{(x,0)}}_1 e^{(2i)}}{(0, N, j, c^{(2i)}) \xrightarrow{\epsilon} (j, e^{(2i)})}$$

Altogether the transitions use the Nth memory cell to store the value of x.

$(\mathbf{M = newsem\, s\, in\, M_1})$ This case is very similar to the previous one: we need to restrict plays from M_1 to those in which $\mathsf{grb}^{(x,\rho)}$ and $\mathsf{rls}^{(x,\rho)}$ moves are immediately followed by answers, and the sequence of such moves obeys the "good semaphore" discipline (grabs follow releases and vice versa). To implement this behaviour in an automaton, we will add a memory cell at level 0 to keep track of the current value of x. Thus, we take $k = k_1$, $N = N_1 + 1$, $C^{(i)} = C_1^{(i)}$ $(0 \leqslant i \leqslant k)$.

Transitions that do not use these special moves are copied over without changes and the remaining transitions are added by following the rules above, which keep track of the current state of the semaphore.

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

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

$(\mathbf{M = f(M_1)})$ Here we only discuss the simplest case of $f : \mathbf{com} \to \mathbf{com}$. We take $k = 2 + k_1$, $N = N_1$, $Q^{(0)} = \{0, 1, 2\}$, $Q^{(1)} = \{3\}$, $Q^{(j+2)} = Q^{(j)}$ $(0 \leqslant j \leqslant k_1)$. First we add transitions corresponding to calling and returning from f: $\dagger \xrightarrow{\mathsf{run}^{(\epsilon,0)}} 0$, $0 \xrightarrow{\mathsf{run}^{(f,0)}} (1, 3)$, $(1, 3) \xrightarrow{\mathsf{done}^{(f,0)}} 2$, $2 \xrightarrow{\mathsf{done}^{(\epsilon,0)}} \dagger$.

In control state 3 we want to allow the environment to spawn an unbounded number of copies of the strategy for $\Gamma \vdash M_1 : \mathbf{com}$: $\dfrac{\dagger \xrightarrow{\mathsf{run}^{(\epsilon,0)}}_1 c^{(0)}}{3 \xrightarrow{\mathsf{run}^{(f1,0)}} (3, c^{(0)})}$, $\dfrac{c^{(0)} \xrightarrow{\mathsf{done}^{(\epsilon,0)}}_1 \dagger}{(3, c^{(0)}) \xrightarrow{\mathsf{done}^{(f1,0)}} 3}$. Note that 3 is immutable.

Other moves related to M_1 originate from Γ, and have the form $m^{(x_v \vec{i}, \rho)}$, where $(x_v \in \theta_v) \in \Gamma$. The associated transitions are copied over but question-moves of the form $m^{(x_v,\rho)}$ (i.e. 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 $\square_L$ and $\square_R$ to refer to arbitrary left- and right-hand sides of transition rules.

$$\frac{\square_L \xrightarrow{m^{(x_v,\rho)}}_1 \square_R \qquad m \text{ is a question}}{\square_L \xrightarrow{m^{(x_v,\rho+2)}} \square_R}$$

$$\frac{\Box_L \xrightarrow{m^{(x_v \vec{i},\rho)}}_1 \Box_R \qquad \vec{i} \neq \epsilon \text{ or } (\vec{i} = \epsilon \text{ and } m \text{ is an answer})}{\Box_L \xrightarrow{m^{(x_v \vec{i},\rho)}} \Box_R}$$

Memory-related transitions are also copied, while adjusting the depth of the level that is being accessed by adding 2: $\frac{(2j,h,v,c^{(2i)}) \xrightarrow{\epsilon}_1 (v',d^{(2i)})}{(2j+2,h,v,c^{(2i)}) \xrightarrow{\epsilon} (v',d^{(2i)})}$. □

VI. Restricted-semaphore FICA

We introduce rsFICA, a fragment of FICA with a restricted use of semaphores. The translation of FICA to split automata allows us to observe structural properties of the automata generated in this case. We discover that they satisfy an idempotency property. In the next section we prove that emptiness is decidable for idempotent automata. This gives a decision procedure for verifying a range of properties of rsFICA terms.

Definition 14. *Restricted-semaphore* FICA (rsFICA) consists of FICA terms $\Gamma \vdash M : \theta$, whose β-normal η-long form is such that subterms of the form $fM_1 \cdots M_l$ ($l > 0$) do not contain free variables of type $\mathbf{sem}$.

Note that rsFICA retains all sequential features of FICA, such as unrestricted types, state and loops. On top of this, it allows for a "shallow" use of semaphores, i.e., not in subterms of the form $f(N)$. For example, the usage in $\mathbf{newsem}\, s\, \mathbf{in}\, \mathbf{grab}(s); f(\mathbf{skip}); \mathbf{release}(s)$ would be shallow, while the usage in $\mathbf{newsem}\, s\, \mathbf{in}\, f(\mathbf{grab}(s))$ is not. In our automata translation, shallow uses occur at the same level as the semaphore declaration and, consequently, can be interpreted using control states alone.

We now explain how the restriction is reflected in the structure of automata obtained by the translation. We identify two properties that will be important for us in the decidability argument of the next section.

The first property, called *local boundedness*, is actually a general property of FICA. It concerns the branching degree of trees in configurations of split automata at *even* levels only.

Definition 15. A split automaton is *locally bounded* if there exists B such that in all reachable configurations (D, E, f, m) every *even-level* node in E has at most B children.

By inspecting our constructions one can confirm that the generated automata are locally bounded, e.g. the bounds add up for the $||$ construct, and in all other constructions it suffices to take the maximum of the bounds for subterms.

Remark 16. In Algol, local boundedness is related to the lack of recursion, which leads to undecidability even without semaphores. If we wanted to extend our translation to FICA with recursion, local boundedness would have to be violated.

A strictly stronger restriction, called (global) boundedness, was considered in [4]. It put a bound on a total number of children that can be created below an even-level node during a run. The difference is that in local boundedness we are interested in the number of children present at any given time. This allows us to accommodate iteration.

The second property, called *restricted-semaphore*, is specific to the rsFICA fragment. It talks about ways the memory is manipulated by an automaton. Recall that the memory is manipulated by EPS transitions that check for a value of some memory cell and modify it in an atomic transition. This enables us to check if a semaphore is free and grab it, if possible. In contrast, read and write operations either just check the value without modifying it, or just modify the value without checking it.

Let us now explain this at the level of split automata. EPS transitions have the form $(2j, h, v, c^{(2i)}) \xrightarrow{\epsilon} (v', d^{(2i)})$. Such a transition finds a data value d labelled with control state $c^{(2i)}$, checks if for the ancestor of d at level $2j$ the h-th cell of memory has value v, and if so, changes this value to v', and changes the control state at d to $d^{(2i)}$. Consequently, if the use of semaphores is restricted then we only have transitions that either check for v and do not modify it, or update the cell to v' without checking the previous value of the cell. This is formalised in the following definition.

Definition 17. In a *restricted-semaphore* split automaton, if there exists a transition $(2j, h, v, c^{(2i)}) \xrightarrow{\epsilon} (v', d^{(2i)})$ for some $v \neq v'$, then there also exist transitions $(2j, h, v'', c^{(2i)}) \xrightarrow{\epsilon} (v', d^{(2i)})$ for all $v'' \in V$.

Some instances of semaphore use can still be translated to restricted-semaphore split automata. This is exactly the origin of the rsFICA fragment. A shallow use of a semaphore is translated into $(0, N, j, c^{(0)}) \xrightarrow{\epsilon} (1 - j, e^{(0)})$ for $j = 0, 1$, i.e. only level-0 states are involved. Such transitions can be simulated by using the level-0 control state as memory. This can be done by taking $C^{(0)} = C_1^{(0)} \times \{0, 1\}$, $C^{(i)} = C_1^{(i)}$ ($i > 0$), $N = N_1 + 1$, initializing the second component of the control state to 0 in ADD(0), and propagating it in other transitions that have access to it, i.e. ADD(1), DEL(1) and EPS(0,0). The transition above can then be replaced by $(0, N, 0, (c^{(0)}, j)) \xrightarrow{\epsilon} (0, (e^{(0)}, 1 - j))$ for $j = 0, 1$. Note that here the memory component is used in a dummy way: 0 is read and not modified.

Corollary 18. *Split automata corresponding to* rsFICA *terms are locally bounded and restricted-semaphore.*

In the next section we show how to decide emptiness for split automata with these two properties. To simplify the decidability proof we formulate the restricted-semaphore property in a more abstract way as a kind of idempotence of transitions of the automaton.

Recall that even-level data are labelled both by a control state of the automaton and memory contents. We call the pair $\langle m^{(2i)}, c^{(2i)} \rangle$ a *combined state*. In terms of combined states, a transition $(2j, h, v, c^{(2i)}) \xrightarrow{\epsilon} (v', d^{(2i)})$ can be written as

$$(\langle m^{(2j)}[h \mapsto v], c^{(2j)} \rangle, \langle m^{(2i)}, c^{(2i)} \rangle) \xrightarrow{\epsilon} (\langle m^{(2j)}[h \mapsto v'], c^{(2j)} \rangle, \langle m^{(2i)}, d^{(2i)} \rangle)$$

If the automaton is restricted-semaphore, we also have

$$(\langle m^{(2j)}[h \mapsto v'], c^{(2j)} \rangle, \langle m^{(2i)}, c^{(2i)} \rangle) \xrightarrow{\epsilon} (\langle m^{(2j)}[h \mapsto v'], c^{(2j)} \rangle, \langle m^{(2i)}, d^{(2i)} \rangle).$$

because of $(2j, h, v', c^{(2i)}) \xrightarrow{\epsilon} (v', d^{(2i)})$. Thus, using $q^{(2i)}, r^{(2i)}$ to range over combined states, we have discovered that the restricted-semaphore condition implies the following *idempotence property*:

$$\begin{array}{ll} \text{if} & (q^{(2j)}, q^{(2i)}) \xrightarrow{\epsilon} (r^{(2j)}, r^{(2i)}) \\ \text{then} & (r^{(2j)}, q^{(2i)}) \xrightarrow{\epsilon} (r^{(2j)}, r^{(2i)}). \end{array}$$

If we interpret the first transition as putting constraints on when $q^{(2i)}$ can be changed to $r^{(2i)}$, the idempotence property says that if this change can be done in one node of the configuration tree, it can be done in an arbitrary number of nodes. This property is crucial for the decidability argument in the next section.

VII. Idempotent automata

The aim of the section is to show that emptiness of locally bounded and restricted-semaphore split automata is decidable. This will allow us to answer certain verification questions about rsFICA, since it compiles to this type of automata.

We prove the decidability result using a more abstract kind of automata called idempotent. Separating memory from control was instrumental to express properties of the translation from FICA to automata. For the decidability proof, all we need is local boundedness and idempotency discussed in the previous section. For this reason, we will work with abstract states that represent combined states of split automata, i.e., pairs $\langle m^{(i)}, c^{(i)} \rangle$ (where, for odd i, the absence of memory is encoded by $m^{(i)}$ having zero cells). We will use $q^{(i)}$ and $r^{(i)}$ to range over combined states. For convenience, we reformulate the definition of split automata in this new notation, including the local boundedness and idempotency properties.

Definition 19. An *idempotent automaton* is a tuple $\langle Q, k, \Sigma, \delta \rangle$ where k is the depth parameter, $Q = \bigcup\{Q^{(i)} : i = 0, \ldots, k\}$ is a finite set of states, $\Sigma = \Sigma_{\mathsf{Q}} + \Sigma_{\mathsf{A}}$ is the alphabet, and δ contains transitions of the shape given below, where $i > 0$, $q \in \Sigma_{\mathsf{Q}}$, $a \in \Sigma_{\mathsf{A}}$ and $q^{(i)}, r^{(i)} \in Q^{(i)}$.

$$\begin{array}{rcl} \mathrm{ADD}(0) & \dagger \xrightarrow{q} r^{(0)} & \\ \mathrm{DEL}(0) & q^{(0)} \xrightarrow{a} \dagger & \\ \mathrm{ADD}(2i) & q^{(2i-1)} \xrightarrow{q} (q^{(2i-1)}, r^{(2i)}) & \\ \mathrm{DEL}(2i) & (q^{(2i-1)}, q^{(2i)}) \xrightarrow{a} q^{(2i-1)} & \\ \mathrm{ADD}(2i+1) & q^{(2i)} \xrightarrow{q} (r^{(2i)}, r^{(2i+1)}) & \\ \mathrm{DEL}(2i+1) & (q^{(2i)}, q^{(2i+1)}) \xrightarrow{a} r^{(2i)} & \\ \mathrm{EPS}(2j, 2i) & (q^{(2j)}, q^{(2i)}) \xrightarrow{\epsilon} (r^{(2j)}, r^{(2i)}) & j \leqslant i \end{array}$$

We require two conditions:

- idempotence: if $(q^{(2j)}, q^{(2i)}) \xrightarrow{\epsilon} (r^{(2j)}, r^{(2i)})$ then $(r^{(2j)}, q^{(2i)}) \xrightarrow{\epsilon} (r^{(2j)}, r^{(2i)})$;
- local boundedness: there is a bound B such that in every reachable configuration every even-level node has at most B children.

The definitions of a run and acceptance are as for split automata. It should be clear that restricted-semaphore locally-bounded split automata can be simulated by idempotent automata by using their states to represent control states combined with memory. As discussed in the previous section, the bound B in the local boundedness condition can be derived from the syntax of the program. Observe that, due to the shape of transitions, the states at odd levels never change.

We are interested in the emptiness problem: does a given idempotent automaton accept a word? We show its decidability via a level reduction lemma. The lemma gives an effective elimination of two levels of data values for an idempotent automaton. Its repeated application reduces the problem to the emptiness problem for standard finite automata.

Lemma 20. *For every idempotent automaton $\mathcal{A}$ with $2i + 2$ levels, one can construct an idempotent automaton $\mathcal{A}^{\uparrow}$ with $2i$ levels such that the language of $\mathcal{A}$ is non-empty if and only if the language of $\mathcal{A}^{\uparrow}$ is nonempty.*

Proof. Suppose B is the bound on the number of children a node at an even level can have. The bound comes from the local boundedness property of $\mathcal{A}$. The states of $\mathcal{A}^{\uparrow}$ are the states of $\mathcal{A}$, except for at level $2i$:

$$Q^{\uparrow(2i)} = Q^{(2i)} \times (\{1, \ldots, B\} \rightharpoonup (Q^{(2i+1)} \times \mathcal{P}(Q^{(2i+2)})))$$

where the second argument is the set of partial functions as displayed. We write $\varnothing$ for the empty function, and $g(i) = \perp$ when g is not defined at i.

The intention is to suplement the state at level $2i$ with a second component, describing subtrees at level $2i + 1$ that are eliminated by the construction. A node of level $2i$ can have at most B children, hence the second component is a partial function with domain $\{1, \ldots, B\}$. The values of this function are representations of subtrees rooted at the children of the node: the label of a node at level $2i + 1$, and the set of labels of children of this node. The representation loses information about the precise number of children with each label. Consequently, there are fintely many representations of subtrees at level $2i + 1$.

For a configuration (D, E, f) and a datum $d \in E$, we write $\overline{fE}(d)$ for the labelled subtree rooted at d. A pair $(q^{(2i+1)}, S) \in Q^{(2i+1)} \times \mathcal{P}(Q^{(2i+2)})$ represents a subtree rooted at a node of level $2i+1$: the root is labelled by $q^{(2i+1)}$, and the set of labels of the children of the root is S. We write $setrep(\overline{fE}(d^{(2i+1)}))$ for this representation of the subtree rooted at $d^{(2i+1)}$.

A state $(q^{(2i)}, g) \in Q^{\uparrow(2i)}$ represents a subtree rooted at a node of level $2i$ with $q^{(2i)}$ the label of the node, and g representing, at most B, subtrees rooted at the children of the node as described above.

The transitions of $\mathcal{A}^{\uparrow}$ reflect this representation of subtrees by states. The way to modify the transitions is presented in Figure 5. It gives a set of rules saying that if a transition above the line is present in $\mathcal{A}$ then it is replaced by the transition below the line in $\mathcal{A}^{\uparrow}$. Observe that only transitions involving levels $2i, 2i + 1, 2i + 2$ are modified.

The intuition behind these rules follows the intuition behind the definition of $Q^{\uparrow(2i)}$ we have seen earlier. For example, the first rule, ADD$(2i)$ of $\mathcal{A}$, creates a new node at level $2i$ labelled by $r^{(2i)}$. The rule is changed to an ADD$(2i)$ rule that

$$\text{ADD}(2i) \quad \frac{q^{(2i-1)} \xrightarrow{q} (q^{(2i-1)}, r^{(2i)})}{q^{(2i-1)} \xrightarrow{q} (q^{(2i-1)}, (r^{(2i)}, \varnothing))} \qquad \text{DEL}(2i) \quad \frac{(q^{(2i-1)}, q^{(2i)}) \xrightarrow{a} q^{(2i-1)}}{(q^{(2i-1)}, (q^{(2i)}, \varnothing)) \xrightarrow{a} q^{(2i-1)}}$$

$$\text{EPS}(2j, 2i) \quad \frac{(q^{(2j)}, q^{(2i)}) \xrightarrow{\epsilon} (r^{(2j)}, r^{(2i)})}{(q^{(2j)}, (q^{(2i)}, g)) \xrightarrow{\epsilon} (r^{(2j)}, (r^{(2i)}, g))}$$

$$\text{ADD}(2i+1) \quad \frac{q^{(2i)} \xrightarrow{q} (r^{(2i)}, r^{(2i+1)})}{(q^{(2i)}, g[l \mapsto \bot]) \xrightarrow{q} (r^{(2i)}, g[l \mapsto (r^{(2i+1)}, \varnothing)])}$$

$$\text{DEL}(2i+1) \quad \frac{(q^{(2i)}, q^{(2i+1)}) \xrightarrow{a} r^{(2i)}}{(q^{(2i)}, g[l \mapsto (q^{(2i+1)}, \varnothing)]) \xrightarrow{a} (r^{(2i)}, g[l \mapsto \bot]))}$$

$$\text{ADD}(2i+2) \quad \frac{q^{(2i+1)} \xrightarrow{q} (q^{(2i+1)}, r^{(2i+2)})}{(q^{(2i)}, g[l \mapsto (q^{(2i+1)}, S)]) \xrightarrow{q} (q^{(2i)}, g[l \mapsto (q^{(2i+1)}, S \cup \{r^{(2i+2)}\})])}$$

$$\text{DEL}(2i+2) \quad \frac{(q^{(2i+1)}, q^{(2i+2)}) \xrightarrow{a} q^{(2i+1)}}{(q^{(2i)}, g[l \mapsto (q^{(2i+1)}, S \cup \{q^{(2i+2)}\})]) \xrightarrow{a} (q^{(2i)}, g[l \mapsto (q^{(2i+1)}, S)])}$$

$$\text{EPS}(2j, 2i+2) \quad \frac{(q^{(2j)}, q^{(2i+2)}) \xrightarrow{\epsilon} (r^{(2j)}, r^{(2i+2)}) \qquad 2j < 2i}{(q^{(2j)}, (q^{(2i)}, g[l \mapsto (q^{(2i+1)}, S \cup \{q^{(2i+2)}\})])) \xrightarrow{\epsilon} (r^{(2j)}, (q^{(2i)}, g[l \mapsto (q^{(2i+1)}, S \cup \{r^{(2i+2)}\})]))}$$

$$\text{EPS}(2i, 2i+2) \quad \frac{(q^{(2i)}, q^{(2i+2)}) \xrightarrow{\epsilon} (r^{(2i)}, r^{(2i+2)})}{(q^{(2i)}, g[l \mapsto (q^{(2i+1)}, S \cup \{q^{(2i+2)}\})]) \xrightarrow{\epsilon} (r^{(2i)}, g[l \mapsto (q^{(2i+1)}, S \cup \{r^{(2i+2)}\})])}$$

$$\text{EPS}(2i+2, 2i+2) \quad \frac{q^{(2i+2)} \xrightarrow{\epsilon} r^{(2i+2)}}{(q^{(2i)}, g[l \mapsto (q^{(2i+1)}, S \cup \{q^{(2i+2)}\})]) \xrightarrow{\epsilon} (q^{(2i)}, g[l \mapsto (q^{(2i+1)}, S \cup \{r^{(2i+2)}\})])}$$

Fig. 5. Translation rules

creates a new node labelled by $(r^{(2i)}, \varnothing)$, where $\varnothing$ is the empty function representing that the newly created node has no children. Another example is the last rule EPS$(2i+2, 2i+2)$. The transition of automaton $\mathcal{A}$ changes the state at a node of level $2i+2$. This is replaced by the change of one of the elements in the set S. The notation we use actually describes two possible ways to instantiate the rule. In one, there may be no $q^{(2i+2)}$ in S, meaning that $q^{(2i+2)}$ is replaced by $r^{(2i+2)}$ in S. In the other, there may still be $q^{(2i+2)}$ remaining in S, so the rule adds $r^{(2i+2)}$ to S without removing $q^{(2i+2)}$. The two ways are useful: the first corresponds to a situation when there is only one child labelled by $q^{(2i+2)}$, the second when there is more than one.

Observe that all $(2i+1)$ and $(2i+2)$ transitions are translated to EPS$(2i, 2i)$ transitions, except for EPS$(2j, 2i+2)$ for $2j < 2i+2$. The idempotence property of the resulting automaton $\mathcal{A}^\uparrow$ then follows from that of $\mathcal{A}$.

We need to show that $\mathcal{A}$ accepts some data word if and only if $\mathcal{A}^\uparrow$ accepts some, possibly different, data word. The two data words will be different when the one accepted by $\mathcal{A}$ uses data of levels below $2i$ as these are not accessible for $\mathcal{A}^\uparrow$.

For the proof we introduce a concept of *indexed runs of* $\mathcal{A}$, namely we add a fourth component to configurations that assigns numbers to some nodes. An indexed configuration is (D, E, f, ind) where $ind : E \rightharpoonup \{1, \ldots, B\}$ is a partial function defined for all data of level $2i+1$ in E. Intuitively, ind gives unique identifiers to siblings at level $2i+1$. When a new node at level $2i+1$ is created it gets the smallest index different from the indices of its siblings. It keeps this index till it is removed. Since a node at level $2i$ can have at most B children we have enough indices. An indexed accepting run has the form:

$$(\varnothing, \varnothing, \varnothing, \varnothing) \xrightarrow{b_1} \ldots (D_l, E_l, f_l, ind_l) \xrightarrow{b_l} \ldots (\varnothing, \varnothing, \varnothing, \varnothing)$$

where as before every b_l is either ϵ or a letter (t_l, d_l) consisting of a tag $t_l \in \Sigma$ and a datum $d_l \in \mathcal{D}$. The indexing functions allow us to define $state(\overline{f_l E_l}(d^{(2i)}), ind_l) \in Q^{\uparrow(2i)}$ for every $d^{(2i)} \in E_l$:

$$state(\overline{f_l E_l}(d^{(2i)}), ind_l) = (f_l(d^{(2i)}), g)$$

where $g(ind_l(d^{(2i+1)})) = setrep(\overline{f_l E_l}(d^{(2i+1)}))$ for every child $d^{(2i+1)}$ of $d^{(2i)}$ in E_l.

From an indexed run ν of $\mathcal{A}$, we construct a run of $\mathcal{A}^\uparrow$ by induction. We suppose that we have constructed a run $\mu^\uparrow$ of $\mathcal{A}^\uparrow$ corresponding to a prefix μ of the run of $\mathcal{A}$. Run $\mu^\uparrow$ reaches a configuration $(D^\uparrow, E^\uparrow, f^\uparrow)$ while run μ reaches (D, E, f, ind). We assume that the following invariant holds:

(1) When restricted to levels $\leqslant 2i$, set $D^\uparrow$ is the same as D, and $E^\uparrow$ is the same as E.
(2) For every $d \in E^\uparrow$ of level $< 2i$ we have $f^\uparrow(d) = f(d)$.
(3) For every $d^{(2i)} \in E^\uparrow$ (of level $2i$), we have $f^\uparrow(d^{(2i)}) = state(\overline{fE}(d^{(2i)}), ind)$.

We consider the next transition on the run ν. If it does not concern level $2i$ or below then the same transition can be

executed by $\mathcal{A}^\uparrow$. If it does we examine the possible cases and show that the corresponding transition of $\mathcal{A}^\uparrow$ preserves the invariant. This way we prolong μ and $\mu^\uparrow$ while keeping the invariant. Property (1) implies that $\mu^\uparrow$ is accepting if μ is.

For the other direction we consider an accepting run $\nu^\uparrow$ of $\mathcal{A}^\uparrow$. Let ℓ be the length of $\nu^\uparrow$. By induction, for every prefix $\mu^\uparrow$ of this run we construct a run μ satisfying the same invariant as above, and moreover:

(4) Every node of level $2i{+}2$ in (D, E, F) has at least $2^{\ell-|\mu^\uparrow|}$ siblings with the same label.

The construction of μ is by cases depending on the type of transitions in the run of $\mathcal{A}^\uparrow$. We need sufficiently big multiplicities of leaves to simulate set operations where the state on the left hand-side does not disappear (cf. our discussion above concerning EPS$(2i+2, 2i+2)$) rule). We can get arbitrary multiplicities thanks to the idempotency of the rules.

Let us examine a representative case of the EPS$(2j, 2i+2)$ rule for $2j < 2i$. Suppose $\mathcal{A}^\uparrow$ applies at a node $d^{(2i)}$ a rule:

$$(q^{(2j)}, (q^{(2i)}, g[l \mapsto (q^{(2i+1)}, S \cup \{q^{(2i+2)}\})])) \xrightarrow{\epsilon} (r^{(2j)}, (q^{(2i)}, g[l \mapsto (q^{(2i+1)}, S \cup \{r^{(2i+2)}\})]))$$

By the invariant $f^\uparrow(d^{(2i)}) = state(\overline{fE}(d^{(2i)}), ind)$, consider $d^{(2i+1)}$ with $ind(d^{(2i+1)}) = l$. We have that it has a child labeled $q^{(2i+2)}$. By the fourth invariant it has at least $2^{\ell-|\mu^\uparrow|}$ children labeled $q^{(2i+2)}$. On the side of automaton $\mathcal{A}$ we can then do the corresponding EPS$(2j, 2i+2)$ transition $(q^{(2j)}, q^{(2i+2)}) \xrightarrow{\epsilon} (r^{(2j)}, r^{(2i+2)})$ followed by some number of transitions $(r^{(2j)}, q^{(2i+2)}) \xrightarrow{\epsilon} (r^{(2j)}, r^{(2i+2)})$. If $q^{(2i+2)}$ does not appear in $S \cup \{r^{(2i+2)}\}$ then the above rule is used to change all occurrences of $q^{(2i+2)}$ below $d^{(2i+1)}$ to $r^{(2i+2)}$. If it appears then $2^{(\ell-|\mu^\uparrow|)-1}$ occurrences are changed, leaving the rest. This reestablishes the fourth invariant both for $q^{(2i+2)}$ and $r^{(2i+2)}$. Observe that if invariant (4) talked about $(\ell - |\mu^\uparrow|)$ siblings instead of $2^{\ell-|\mu^\uparrow|}$ then it would not be possible to reestablish it at this point. □

Repeated applications of Lemma 20 reduce the emptiness problem of an idempotent automaton to the emptiness problem of a standard finite automaton. Indeed, an idempotent automaton whose depth parameter is 0 is just a finite automaton.

Theorem 21. *Emptiness of idempotent automata is decidable.*

VIII. Verification of stuttering invariant properties

We show some examples of properties that we can verify using our construction. As described in previous sections, we can translate a given program fragment into an automaton accepting the representation of complete plays in the game semantics. The automaton works on a data tree of some depth, which reflects the syntactic structure of the underpinning λ-term. We fix some level $2i$ of the data tree, and for every $2i$ node we look at the sequence of states appearing at the node during a run. We check if some/every such sequence satisfies a given regular property. Later we will show how some standard program analysis properties can be expressed in this way.

We first describe automata constructions and then give some applications. We try to give an idea of the constructions without going to excessive details that are not difficult but tedious. To fix the notation, consider an idempotent automaton $\mathcal{A}$, and a level $2i$ of the dataset. We assume that we are given a standard finite automaton $\mathcal{C}$ defining interesting sequences of states at level $2i$. So the alphabet of $\mathcal{C}$ is $Q^{(2i)}$, i.e. the set of states of $\mathcal{A}$ at level $2i$. We use q^c to refer to states of $\mathcal{C}$. The initial state of $\mathcal{C}$ is q^c_{init}, and the set of accepting states of $\mathcal{C}$ is $Fin^{\mathcal{C}}$.

We want to check if there is an accepting run of $\mathcal{A}$ such that every node at level $2i$ goes through a sequence of states accepted by $\mathcal{C}$. In other words, if there is an accepting run such that for every datum d of level $2i$ appearing in the run the sequence of states that label d is accepted by $\mathcal{C}$. We then say *there is an accepting run where every $2i$ sub-run satisfies $\mathcal{C}$*.

We convert an idempotent automaton $\mathcal{A}$ into an idempotent automaton $\mathcal{A}^{\forall\mathcal{C}}$ such that: $\mathcal{A}^{\forall\mathcal{C}}$ has an accepting run if and only if $\mathcal{A}$ has an accepting run where every $2i$ sub-run satisfies $\mathcal{C}$. This reduces the question to the emptiness of idempotent automata that, as we have seen, is decidable.

For this construction to work we require that automaton $\mathcal{C}$ describes a property that is *stuttering invariant*: automaton $\mathcal{C}$ accepts some word $w_1 b w_2$ if and only if it accepts the word $w_1 b b w_2$. We assume below that $\mathcal{C}$ is a minimal deterministic automaton. Observe that if $\mathcal{C}$ is a stuttering invariant minimal deterministic automaton then, for every transition $q^c_1 \xrightarrow{b} q^c_2$, it has also the transition $q^c_2 \xrightarrow{b} q^c_2$.

To construct $\mathcal{A}^{\forall\mathcal{C}}$ we modify states at level $2i$ of $\mathcal{A}$, and transitions involving this level. The new set of sates at level $2i$ is $Q^{(2i)} \times Q^c$; namely we add states of $\mathcal{C}$ as another component. We modify transitions:

$$\mathrm{ADD}(2i)\,\frac{q^{(2i-1)} \xrightarrow{q} (q^{(2i-1)}, r^{(2i)})}{q^{(2i-1)} \xrightarrow{q} (q^{(2i-1)}, (r^{(2i)}, q^c))} \quad \text{if } q^c_{init} \xrightarrow{r^{(2i)}} q^c.$$

$$\mathrm{DEL}(2i)\,\frac{(q^{(2i-1)}, q^{(2i)}) \xrightarrow{a} r^{(2i-1)}}{(q^{(2i-1)}, (q^{(2i)}, q^c)) \xrightarrow{a} q^{(2i-1)}} \quad \text{if } q^c \in Fin^{\mathcal{C}}$$

ADD$(2i)$ transitions initialise the $\mathcal{C}$ component. DEL$(2i)$ transitions allow to remove a $2i$ node provided the $\mathcal{C}$ component is in an accepting state.

We modify other transitions involving level $2i$ so that the $\mathcal{C}$ component is updated as expected. Supposing $q^c_1 \xrightarrow{r^{(2i)}} q^c_2$ is a transition of $\mathcal{C}$:

$$\mathrm{EPS}(2j, 2i)\,\frac{(q^{(2j)}, q^{(2i)}) \xrightarrow{\epsilon} (r^{(2j)}, r^{(2i)})}{(q^{(2j)}, (q^{(2i)}, q^c_1)) \xrightarrow{\epsilon} (r^{(2j)}, (r^{(2i)}, q^c_2))}$$

$$\mathrm{ADD}(2i+1)\,\frac{q^{(2i)} \xrightarrow{q} (r^{(2i)}, r^{(2i+1)})}{(q^{(2i)}, q^c_1) \xrightarrow{q} ((r^{(2i)}, q^c_2), r^{(2i+1)})}$$

$$\mathrm{DEL}(2i+1)\,\frac{(q^{(2i)}, q^{(2i+1)}) \xrightarrow{a} r^{(2i)}}{((q^{(2i)}, q^c_1), q^{(2i+1)}) \xrightarrow{a} (r^{(2i)}, q^c_2)}$$

$$\mathrm{EPS}(2i, 2j)\,\frac{(q^{(2i)}, q^{(2j)}) \xrightarrow{\epsilon} (r^{(2i)}, r^{(2j)})}{((q^{(2i)}, q^c_1), q^{(2j)}) \xrightarrow{\epsilon} ((r^{(2i)}, q^c_2), r^{(2j)})}$$

Observe that we have two types of EPS transitions to handle, depending on whether $2i$ is the lower or the upper level. Note that EPS transitions of the first kind are idempotent in $\mathcal{A}^{\forall\mathcal{C}}$ if they were in $\mathcal{A}$. Similarly for the second kind, but here additionally we rely on stuttering invariance of $\mathcal{C}$.

Lemma 22. *If $\mathcal{A}$ is an idempotent automaton and $\mathcal{C}$ is a stuttering invariant minimal deterministic automaton then $\mathcal{A}^{\forall\mathcal{C}}$ is an idempotent automaton. Moreover, $\mathcal{A}$ has an accepting run whose all $2i$ sub-runs satisfy $\mathcal{C}$ if and only if $\mathcal{A}^{\forall\mathcal{C}}$ has an accepting run.*

We can modify the above construction to answer another question: is there an accepting run with at least one $2i$ sub-run satisfying $\mathcal{C}$? Automaton $\mathcal{A}^{\exists\mathcal{C}}$ for this question is constructed from $\mathcal{A}^{\forall\mathcal{C}}$. The idea is that once one of the sub-runs at level $2i$ reaches an accepting state, it puts this information in the root state. At the same time we should ensure that this sub-run terminates after this action. In the universal case we could use termination detection to collect acceptance information: in an accepting run all components at level $2i$ needed to disappear, and they could disappear only when they reached an accepting state. In the existential case we need to implement an ad hoc notification mechanism.

In $\mathcal{A}^{\exists\mathcal{C}}$, the set of states at level 0 becomes $Q^{(0)} \cup \{tt, ff\}$; the second component indicating if there is a $2i$ sub-run satisfying $\mathcal{C}$. The modification of transitions reflects this intuition:

$$\mathrm{ADD}(0)\frac{\dagger \xrightarrow{q} r^{(0)}}{\dagger \xrightarrow{q} (r^{(0)}, ff)} \qquad \mathrm{DEL}(0)\frac{q^{(0)} \xrightarrow{a} \dagger}{(q^{(0)}, tt) \xrightarrow{a} \dagger}$$

The ADD transition says that we initialize the second component to ff, the DEL transition says that an accepting run should end with the second component set to tt.

For other transitions of $\mathcal{A}^{\exists\mathcal{C}}$, we take transitions of $\mathcal{A}^{\forall\mathcal{C}}$ with some modifications. Transitions involving level 0 are modified so that they leave the second component unchanged. The second component can be changed only by a new ϵ-transition at level $2i$ that we introduce now. In $\mathcal{A}^{\exists\mathcal{C}}$ the set of states at level $2i$ is $Q^{(2i)} \cup (Q^{(2i)} \times Q^c)$. So a state at level $2i$ of $\mathcal{A}^{\exists\mathcal{C}}$ is either a state of $\mathcal{A}^{\forall\mathcal{C}}$ or a state of $\mathcal{A}$ at this level. The idea is that at the end of a computation at some node at level $2i$, the automaton can set the component at level 0 to tt if the computation finished in an accepting state of $\mathcal{C}$. We have the following for arbitrary states and arbitrary $\alpha \in \{tt, ff\}$.

$$\mathrm{EPS}(0, 2i)\ ((q^{(0)}, \alpha), (q^{(2i)}, q^c)) \xrightarrow{\varepsilon} ((q^{(0)}, tt), q^{(2i)}) \text{ if } q^c \in F$$

$$\mathrm{EPS}(0, 2i)\ ((q^{(0)}, \alpha), (q^{(2i)}, q^c)) \xrightarrow{\varepsilon} ((q^{(0)}, \alpha), q^{(2i)})$$

It is clear that these transitions are idempotent. The other transitions are as in automaton $\mathcal{A}^{\forall\mathcal{C}}$ except for the transitions of type $\mathrm{DEL}(2i)$ that are those from $\mathcal{A}$: $(q^{(2i-1)}, q^{(2i)}) \xrightarrow{a} r^{(2i-1)}$ The idea is that $\mathrm{DEL}(2i)$ can be performed only after one of the above $\mathrm{EPS}(0, 2i)$ transitions.

Lemma 23. *If $\mathcal{A}$ is an idempotent automaton and $\mathcal{C}$ is a stuttering invariant minimal deterministic automaton then $\mathcal{A}^{\exists\mathcal{C}}$ is an idempotent automaton. Moreover, $\mathcal{A}$ has an accepting run whose some $2i$ sub-run satisfies $\mathcal{C}$ if and only if $\mathcal{A}^{\exists\mathcal{C}}$ has an accepting run.*

Proposition 24. *The following questions are decidable, for idempotent automata $\mathcal{A}$, data levels $2i$, and stuttering invariant finite automata $\mathcal{C}$:*

- *Is there an accepting run of $\mathcal{A}$ whose all $2i$ sub-runs satisfy $\mathcal{C}$?*
- *Is there an accepting run of $\mathcal{A}$ whose some $2i$ sub-run satisfies $\mathcal{C}$?*
- *Do all accepting runs of $\mathcal{A}$ have some $2i$ sub-run satisfying $\mathcal{C}$?*
- *Do all accepting runs of $\mathcal{A}$ have all $2i$ sub-runs satisfying $\mathcal{C}$?*

The first two items are solved by the two lemmas above. The remaining two are obtained by considering the dual question for the complement automaton. Observe that the complement of a stuttering invariant language is also stuttering invariant. In fact, this is the only point where we need stuttering invariance of $\mathcal{C}$. In the rest of the arguments till now we need only that $\mathcal{C}$ is closed under stuttering expansion.

This proposition allows the verification of rsFICA terms to be reduced to emptiness checking of idempotent automata. Given a term we use the translation from the proof of Theorem 13 to obtain a split automaton. This automaton is an idempotent automaton by Corollary 18. We can then take an automaton $\mathcal{C}$ expressing a property of interest. With stuttering invariant finite automata $\mathcal{C}$ we can express such properties as:

- Is a given variable set to a given value?
- Is a given variable invariant in a loop?
- Is a given variable read after being set?

As reads are silent, in order to check the last property, it will be necessary to instrument the program to perform a special write on every relevant read.

Proposition 24 gives a method to verify these properties for all quantification combinations: every/exists accepting run and every/exists sub-run of this run. Indeed, the proposition effectively reduces verification of all these questions to emptiness checking of a suitable idempotent automaton. The latter is decidable by Theorem 21.

IX. Conclusions

This work identifies a fragment of Finitary Idealized Concurrent Algol, called rsFICA, having good algorithmic properties with respect to verification. Thanks to the use of game semantics we can verify properties talking about executions in all contexts. In rsFICA we restrict the use of semaphores and we do not permit recursion. Removing one of these restrictions makes the verification problems we have considered here undecidable. Observe that we admit iteration in rsFICA, so there is a striking difference between the power of iteration and recursion in this context.

We find it interesting that parametrisation appears in our setting. In existing literature parametrisation was postulated as a way to get around undecidability. Here, it arises naturally in the game-semantic interpretation of rsFICA terms. In our decidability proof we work in a setting similar to that of [5], but we have a hierarchy of parameterised systems, that moreover can test for termination. Downward closure arguments are an efficient way to get decidability for parametric systems [21], [9]. Unfortunately they are not applicable in the context of termination [7], [22]. This is where the notion of idempotent transitions comes very useful. It nicely captures

"no test-and-set" intuition, and allows for a relatively direct inductive argument on tree levels.

As future work, we would like to investigate the equivalence problem for idempotent automata. If decidable, this could be used in testing contextual equivalence of rsFICA terms. The test would give false negatives though, as our encoding of plays via data words is not bijective: several different encodings may represent the same play. Other related challenges include applying the concurrent games framework [23] in verification and investigating contextual equivalence with respect to semaphore-free contexts [24]. It would also be interesting to look for connections with abstract machines [25], the Geometry of Interaction [26], and the π-calculus [27].

We also do not know the complexity of emptiness checking for idempotent automata. Our procedure has complexity of a tower of exponentials whose height depends on the idempotent automaton's depth parameter.

References

[1] D. R. Ghica and A. S. Murawski, "Angelic semantics of fine-grained concurrency," *Ann. Pure Appl. Log.*, vol. 151(2-3), pp. 89–114, 2008. doi:10.1016/j.apal.2007.10.005

[2] D. R. Ghica, A. S. Murawski, and C.-H. L. Ong, "Syntactic control of concurrency," *Theor. Comp. Sci.*, pp. 234–251, 2006. doi:10.1016/j.tcs.2005.10.032

[3] G. Ramalingam, "Context-sensitive synchronization-sensitive analysis is undecidable," *ACM Trans. Program. Lang. Syst.*, vol. 22, no. 2, pp. 416–430, 2000. doi:10.1145/349214.349241

[4] A. Dixon, R. Lazic, A. S. Murawski, and I. Walukiewicz, "Leafy automata for higher-order concurrency," in *Proceedings of FoSSaCS*, ser. LNCS, 2021. [Online]. Available: http://arxiv.org/abs/2101.08720

[5] S. A. German and P. A. Sistla, "Reasoning about systems with many processes," *J. ACM*, vol. 39, no. 3, pp. 675–735, 1992. doi:10.1145/146637.146681

[6] M. Hague, "Parameterised pushdown systems with non-atomic writes," in *Proceedings of FSTTCS*, ser. LIPIcs, 2011, pp. 457–468. doi:10.4230/LIPIcs.FSTTCS.2011.457

[7] P. Ganty and R. Majumdar, "Algorithmic verification of asynchronous programs," *ACM Trans. Program. Lang. Syst.*, vol. 34, no. 1, pp. 6:1–6:48, 2012. doi:10.1145/2160910.2160915

[8] J. Esparza, P. Ganty, and R. Majumdar, "Parameterized verification of asynchronous shared-memory systems," *J. ACM*, vol. 63, no. 1, p. 10, 2016. doi:10.1145/2842603

[9] S. L. Torre, A. Muscholl, and I. Walukiewicz, "Safety of parametrized asynchronous shared-memory systems is almost always decidable," in *Proceedings of CONCUR*, ser. LIPIcs, vol. 42, 2015, pp. 72–84. doi:10.4230/LIPIcs.CONCUR.2015.72

[10] P. A. Abdulla, M. F. Atig, and R. Rezvan, "Parameterized verification under TSO is pspace-complete," *Proc. ACM Program. Lang.*, vol. 4, no. POPL, pp. 26:1–26:29, 2020. doi:10.1145/3371094

[11] T. Schwentick, "Automata for XML - A survey," *J. Comput. Syst. Sci.*, vol. 73, no. 3, pp. 289–315, 2007. doi:10.1016/j.jcss.2006.10.003

[12] H. Björklund and M. Bojańczyk, "Shuffle expressions and words with nested data," in *Proceedings of MFCS*, ser. LNCS, vol. 4708, 2007, pp. 750–761. doi:10.1007/978-3-540-74456-6_66

[13] M. Bojańczyk, C. David, A. Muscholl, T. Schwentick, and L. Segoufin, "Two-variable logic on data words," *ACM Trans. Comput. Log.*, vol. 12, no. 4, pp. 27:1–27:26, 2011. doi:10.1145/1970398.1970403

[14] H. Björklund and T. Schwentick, "On notions of regularity for data languages," *Theor. Comput. Sci.*, vol. 411, no. 4-5, pp. 702–715, 2010. doi:10.1016/j.tcs.2009.10.009

[15] N. Decker, P. Habermehl, M. Leucker, and D. Thoma, "Ordered navigation on multi-attributed data words," in *Proceedings of CONCUR*, ser. LNCS, vol. 8704, 2014, pp. 497–511. doi:10.1007/978-3-662-44584-6_34

[16] C. Cotton-Barratt, A. S. Murawski, and C. L. Ong, "Weak and nested class memory automata," in *Proceedings of LATA*, ser. LNCS, vol. 8977, 2015, pp. 188–199. doi:10.1007/978-3-319-15579-1_14

[17] C. Cotton-Barratt, D. Hopkins, A. S. Murawski, and C. L. Ong, "Fragments of ML decidable by nested data class memory automata," in *Proceedings of FOSSACS*, ser. LNCS, vol. 9034, 2015, pp. 249–263. doi:10.1007/978-3-662-46678-0_16

[18] M. Hague, "Saturation of concurrent collapsible pushdown systems," in *Proceedings of FSTTCS*, ser. LIPIcs, vol. 24, 2013, pp. 313–325. doi:10.4230/LIPIcs.FSTTCS.2013.313

[19] N. Kobayashi and A. Igarashi, "Model-checking higher-order programs with recursive types," in *Proceedings of ESOP*, ser. LNCS, vol. 7792, 2013, pp. 431–450. doi:10.1007/978-3-642-37036-6_24

[20] J. C. Reynolds, "The essence of Algol," in *Algorithmic Languages*, J. W. de Bakker and J. van Vliet, Eds. North Holland, 1978, pp. 345–372. doi:10.1007/978-1-4612-4118-8_4

[21] G. Zetzsche, "An approach to computing downward closures," in *Proceedings of ICALP*, ser. LNCS, 2015, pp. 440–451. doi:10.1007/978-3-662-47666-6_35

[22] M. Fortin, A. Muscholl, and I. Walukiewicz, "Model-checking linear-time properties of parametrized asynchronous shared-memory pushdown systems," in *Proceedings of CAV*, ser. LNCS, vol. 10427, 2017, pp. 155–175. doi:10.1007/978-3-319-63390-9_9

[23] S. Castellan, P. Clairambault, S. Rideau, and G. Winskel, "Games and strategies as event structures," *Log. Meth. Comput. Sci.*, vol. 13, no. 3, 2017. doi:10.23638/LMCS-13(3:35)2017

[24] A. S. Murawski, "Full abstraction without synchronization primitives," in *Proceedings of MFPS*, ser. ENTCS, vol. 265, 2010, pp. 423–436. doi:10.1016/j.entcs.2010.08.025

[25] O. Fredriksson and D. R. Ghica, "Abstract machines for game semantics, revisited," in *Proceedings of LICS*, 2013, pp. 560–569. doi:10.1109/LICS.2013.63

[26] U. D. Lago, 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. doi:10.1109/LICS.2017.8005112

[27] M. Berger, K. Honda, and N. Yoshida, "Sequentiality and the pi-calculus," in *Proceedings of TLCA*, ser. LNCS, 2001, vol. 2044, pp. 29–45. doi:10.1007/3-540-45413-6_7