Logical Methods in Computer Science
Volume 22, Issue 2, 2026, pp. 5:1–5:30
https://lmcs.episciences.org/

Submitted Dec. 20, 2024
Published Apr. 16, 2026

EXPRESSIVITY OF AUDALA: TURING COMPLETENESS AND POSSIBLE EXTENSIONS

TOM T.P. FRANKEN AND THOMAS NEELE

Eindhoven University of Technology, The Netherlands
e-mail address: {t.t.p.franken, t.s.neele}@tue.nl

Abstract. AuDaLa is a recently introduced programming language that follows the new data autonomous paradigm. In this paradigm, small pieces of data execute functions autonomously. Considering the paradigm and the design choices of AuDaLa, it is interesting to determine the expressivity of the language. In this paper, we implement Turing machines in AuDaLa and prove that implementation correct. This proves that AuDaLa is Turing complete, giving an initial indication of AuDaLa's expressivity. Additionally, we give examples of how to add extensions to AuDaLa to increase its practical expressivity and to better match conventional parallel languages, allowing for a more straightforward and performant implementation of algorithms.

1. Introduction

Nowadays, performance gains are increasingly obtained through parallelism. To make use of this, there are many developments in how to get the hardware to process the program efficiently. Languages are often designed around that, focusing on threads and processes. Recently, AuDaLa [FNG23, FNG25] was introduced, which completely abstracts away from threads. In AuDaLa, data is *autonomous*, meaning that the data executes its own functions. It follows the new data autonomous paradigm [FNG23, FNG25], which abstracts away from active processor and memory management for parallel programming and instead focuses on the innate parallelism of data. This paradigm encourages parallelism by making running code in parallel the default setting, instead of requiring functions to be explicitly called in parallel. The paradigm also promotes separation of concerns and a bottom-up design process.

The simplicty, structure and focus on data and understandable programs relates AuDaLa to domain specific languages and brings up the question to which extent AuDaLa is generally applicable. It is therefore interesting to establish how applicable and expressive AuDaLa is, as AuDaLa is meant to be general-purpose.

The applicability of AuDaLa has been studied in other work by creating a compiler [Lee23] that can translate AuDaLa programs to CUDA [G+08]. It proved beneficial to base the compiler on a new *weak-memory semantics* [LFN25] of AuDaLa. We furthermore formalised the relation of the weak-memory semantics with the original semantics of [FNG23]. The

Key words and phrases: AuDaLa and Verification and Expressiveness.
Extension of 'AuDaLa is Turing Complete' as published in the proceedings of FORTE 2024 [FN24].

DOI:10.46298/LMCS-22(2:5)2026

performance of AuDaLa programs compiled under the weak-memory semantics was sufficient for us to consider AuDaLa fit for parallel programming in practice.

Then, to establish the expressivity of AuDaLa, we used the original semantics to establish that AuDaLa can simulate every Turing machine [FN24]. It follows that AuDaLa is *Turing complete* and can therefore compute all effectively computable functions following the Church-Turing thesis [Cop24]. This also gives an indication for the theoretical expressivity of the data-autonomous paradigm in general.

The current paper extends our paper from FORTE 2024 [FN24] in the following ways:

- We provide a self-contained account of AuDaLa's semantics, including a running example, and recap a few results from [FNG25] required for our proofs.
- We give the full proof for the Turing completeness of AuDaLa, extending parts of the proof which were only sketched before.
- We explore of the *practical expressivity* of AuDaLa: the extent to which AuDaLa enables the expression of the user's intentions in the program directly. We do this by providing three possible extensions to AuDaLa, including the necessary additions to the syntax and semantics. Two of the extensions deal with extended looping mechanisms and the other implements arrays in AuDaLa.
- We discuss the adaptations, some of the design choices and other adaptations that are worth considering in the future.

We remark that for all results, we use the original, sequentially consistent semantics [FNG23]. This semantics has been proven equivalent to the weak-memory semantics [LFN25] when no read-write race conditions occur, which is sufficient for the Turing completeness proof in this paper. Though we only define the semantic adaptations of this paper for the original semantics of AuDaLa, extending them to the weak-memory semantics is reasonably straightforward.

Overview. In the current paper, we first give a recap of Turing machines and AuDaLa in Section 2, followed by the implementation of a Turing machine in AuDaLa in Section 3. We then give some properties of AuDaLa and some important corollaries for proving programs in AuDaLa in Section 4, which is based on [FNG25, Section 7]. These are used to prove that AuDaLa is Turing complete in Section 4.2. Then, in Section 6, we propose three extensions to AuDaLa. We conclude in Section 7.

Related Work. AuDaLa is a *data-autonomous* language and related to other data-focused languages, like standard data-parallel languages (CUDA [G$^+$08] and OpenCL [CDK14]), languages which apply local parallel operations on data structures (Halide [RK$^+$17], ReLaCS [RL93]) and actor-based languages (Ly [UA10], A-NETL [BY95]).

Though the expressivity of actor languages has been studied before [dB$^+$12] and there is research into suitable Turing machine-like models for concurrency [QYZG17, Koz76, Wie84], there does not seem to be a large focus on proving Turing completeness of parallel languages. This can have multiple reasons. One of these reasons can be that many of these languages extend other languages, e.g., CUDA and OpenCL are built upon C++. For these languages, Turing completeness is inherited from their base language. Other languages, for example domain specific languages like Halide [RK$^+$17], are simple by design, and if the functionality is more important than the applicability, Turing completeness may be of lesser importance. Some languages are also not Turing complete on purpose [Gib15, DK98], for example to make automated verification decidable. As AuDaLa is not an extension of a sequential

language, Turing completeness for AuDaLa is not automatic. It has been noted that Turing completeness also does not establish as much about expressivity as it does in the domain of sequential languages, as Turing completeness does not account for the effects of concurrent and distributive operations, which can lead to differences in behaviour in two Turing complete concurrent languages [DG09]. Though we agree with this, the fact stands that the full behaviour of a language is a superset of the behaviour of the language in the sequential setting, so establishing AuDaLa to be Turing complete is a good starting point for analysing the expressivity of AuDaLa. By proving AuDaLa to be Turing complete, we furthermore establish that even though its simple design and rigid structure may make it seem like AuDaLa is simple enough to be decidable or domain specific, in reality, AuDaLa is no less complex than other Turing complete, general languages.

Our proof follows the same line as the proof for the Turing completeness of Circal [DD02]. Other parallel systems that have been proven Turing complete include water systems [HND+21] and asynchronous non-camouflage cellular automata [Y+17].

2. Basic Concepts

In this section, we discuss the basics of Turing machines and AuDaLa.

2.1. Turing Machines.

We define a Turing machine following the definition of Hopcroft *et al.* [HMU01]. Let $\mathbb{D} = \{L, R\}$ be the set of the two directions *left* and *right*. A Turing machine T is a 7-tuple $T = (Q, q_0, F, \Gamma, \Sigma, B, \delta)$, with a finite set of control states Q, an initial state $q_0 \in Q$, a set of accepting states $F \subseteq Q$, a set of tape symbols Γ, a finite set of input symbols $\Sigma \subseteq \Gamma$, a blank symbol $B \in \Gamma \setminus \Sigma$ (the initial symbol of all cells not initialized) and a partial transition function $\delta : Q \times \Gamma \nrightarrow Q \times \Gamma \times \mathbb{D}$.

Every Turing machine T operates on an infinite *tape* divided into *cells*. Initially, this tape contains an input string $Z = z_0 \dots z_n$ with symbols from Σ, but is otherwise blank. The cell the Turing machine operates on is called the *head*. We represent the tape as a function $t : \mathbb{Z} \to \Gamma$, where cell i contains symbol $t(i) \in \Gamma$. In this function, cell 0 is the head, cells i s.t. $i < 0$ are the cells left from the head and cells i s.t. $i > 0$ are the cells right from the head. We restrict ourselves to deterministic Turing machines. We also assume the input string is not empty, without loss of generality.

We define a *configuration* to be a tuple (q, t), with q the current state of the Turing machine and t the current tape function. Given input string $Z = z_0 \dots z_n$, the *initial configuration* of a Turing machine T is (q_0, t_Z), with q_0 as defined for T, and $t_Z(i) = z_i$ for $0 \leq i \leq n$ and $t_Z(i) = B$ otherwise.

During the execution, a Turing machine T performs *transitions*, defined as:

Definition 2.1 (Turing machine transition)**.** Let $T = (Q, q_0, F, \Gamma, \Sigma, B, \delta)$ be a Turing machine with input string Z and let (q, t) be a configuration such that $\delta(q, t(0)) = (q', z', D)$, with $D \in \mathbb{D}$. Then $(q, t) \to (q', t')$, where t' is defined as

$$t'(i) = \begin{cases} z' & \text{if } i = 1 \\ t(i-1) & \text{otherwise} \end{cases} \text{ if } D{=}L \text{ and } t'(i) = \begin{cases} z' & \text{if } i = -1 \\ t(i+1) & \text{otherwise} \end{cases} \text{ if } D{=}R.$$

We say a Turing machine T *accepts* a string Z iff, starting from (q_0, t_Z) and taking transitions while possible, T halts in a configuration (q, t) s.t. $q \in F$.

2.2. **AuDaLa.** In this section, we give a recap of AuDaLa concepts, program layout and semantics relevant to this paper. As an example throughout the text, we use Listing 1, depicting a program for computing reachability on a small directed graph. In essence, AuDaLa is a single instruction-multiple data programming language, which abstracts away from explicit memory management and thread management and puts a large focus on the structure of parallel program design. By doing this, it keeps the design of the program modular and simple.

An AuDaLa program contains three parts, which are neatly separated in the code. Firstly, the definitions of the data types and their parameters are expressed as *structs*. In Listing 1, we are computing something on a graph, so we define structs for nodes and edges, where edges have a source and target (given as the parameters *in* and *out*) and nodes can either be reachable or not reachable. During execution, these structs will be instantiated to *struct instances*, which represent the parallel data elements of the system. For example, in the reachability example, every node in the graph is be represented by a struct instance of the struct *Node*. The second part are *steps*, which are defined in context of a struct. These are functions to be executed in parallel. The steps are *simple*, in the sense that they cannot contain loops. For example, in Listing 1, the struct *Edge* contains two steps; one for propagating the reachability property, and the other for initialization. Lastly, a *schedule* separate from the data system dictates in which order steps are executed, and which loops exist during execution. The looping mechanism used in AuDaLa is called a *fixpoint* loop, which repeatedly executes the embedded schedule until the entire system is *stable*. This is the case when in the last iteration of the loop, no new struct instances have been created and no parameters have been changed. For example, the schedule of Listing 1 first calls the initialization step once, before calling a fixpoint loop over the *reachability* step. This loop executes the reachability step until the entire system is stable, which in this case means that no more nodes are set to reachable. This is the case when nodes one to four are set to reachable, as node five is not reachable.

In the schedule, step calls and fixpoints are separated by *barriers* ($<$), s.t. $A < B$ means that subschedule A is executed and only when every struct instance is finished, B is executed. Step calls can be global step calls or local step calls. For a step F, a global step call is denoted by the appearance of F in the schedule, and will cause every struct instance of which the struct has a definition of F to execute F in parallel as defined for their struct. Local step calls are denoted as $\theta.F$ for some struct type θ, and will cause only the struct instances of θ to execute F.

Steps consist of statements, which can be if-then statements, variable assignments (of the form $T\ x := val$ for some new variable x of type T) variable updates (of the form $X := val$ for some (possibly referenced) variable X) and *constructor statements*, which create new struct instances. In the statements, one can use most conventional expressions, like applying binary operators, using brackets and negation, refer to **null** and **this**, create struct instances and introduce literals or access variables. Variables are accessed using references; variable $x.y$ returns the value of y in the element reached through the reference x.

Henceforth, we only consider AuDaLa program that are *well-formed*. Well-formed AuDaLa programs are well-typed and satisfy the following requirements:

(1) Identifiers may not be keywords.
(2) A step name is declared at most once within each struct definition.
(3) A parameter name is used at most once within each struct definition.

```
struct Node (reach: Bool) {}

struct Edge (in: Node, out: Node) {
  reachability {
    if (in.reach = true) then {
      out.reach := true;
    }
  }
  init {
    Node node1 := Node(true);
    Node node2 := Node(false);
    Node node3 := Node(false);
    Node node4 := Node(false);
    Node node5 := Node(false);

    Edge edge12 := Edge(node1, node2);
    Edge edge13 := Edge(node1, node3);
    Edge edge23 := Edge(node2, node3);
    Edge edge34 := Edge(node3, node4);
    Edge edge51 := Edge(node5, node1);
  }
}

init < Fix(reachability)
```

Listing 1: AuDaLa code for a reachability program on a small graph

(4) Names of local variables do not overlap with parameter names of the surrounding struct definition.
(5) Variable assignment statements are only used to declare new local variables.
(6) A local variable is only used after its declaration in a variable assignment statement.

Our example in Listing 1 is well-formed, as it is well-typed and fulfills all of the above conditions. A type system that formalises the requirements on well-formed programs can be found in [FNG25, Section 4].

2.3. **AuDaLa Semantics.** In this section, we explain the semantics of AuDaLa, using the reachability program of Listing 1 as a running example. In the scope of this section, we denote that program as the program $\mathcal{R}$. In the semantics, we often use lists of the form $a; \ldots; z$, with empty list ε. Here, it is important to note that for any list L, $L = L; \varepsilon$. We also use function updates: a function $f[x \mapsto y]$ is created from a function f, where $f[x \mapsto y](x) = y$ and $f[x \mapsto y](z) = f(z)$ for all $z \neq x$. We allow the shorthand $f[\{a_1 \mapsto b_1, a_2 \mapsto b_2, \ldots\}]$ for multiple function updates where all left-hand sides are pairwise distinct, and use the indexed notation $f[a, i \mapsto b]$ if $f(a)$ is a tuple and we want to only change the element at index i of that tuple. We also assume that $\mathcal{R}$ is parsed and that we have its abstract syntax tree. In general, this means that polymorphic elements in $\mathcal{R}$, like 42 or *null* are annotated with their type, as for example $42_{\mathbb{Z}}$, when 42 is an integer, or $null_T$ for some type T. We divide the explanation of the semantics in three parts: *semantic concepts and sets*, *commands and the state* and *the transition relation*.

Semantic Concepts and Sets. To reason about the syntax, the semantics uses a group of syntactic sets: ID for variable names and other names from the syntax, LT for the set of literals, ST for statements, E for expressions, and O, containing all syntactic binary operators. Though we allow all possible identifiers in the set ID, we only use a specific subset. For $\mathcal{R}$, we therefore further refine the set ID to the sets $Var_{\mathcal{R}}$, containing variable names used in $\mathcal{R}$, $Par_{\mathcal{R}}$, containing parameter names used in $\mathcal{R}$ and $\Theta_{\mathcal{R}}$, which contains all struct type names used in $\mathcal{R}$. Concretely, $Var_{\mathcal{R}} = \{\text{node1}, \ldots, \text{node5}, \text{edge12}, \ldots, \text{edge51}\}$, $Par_{\mathcal{R}} = \{\text{reach}, \text{in}, \text{out}\}$ and $\Theta_{\mathcal{R}} = \{\text{Node}, \text{Edge}\}$.

The set of all syntactic types is $\mathcal{T} = \{\texttt{Nat}, \texttt{Int}, \texttt{Bool}, \texttt{String}\} \cup ID$, which has an associated set of semantic types $\mathbb{T} = \{\mathbb{N}, \mathbb{Z}, \mathbb{B}, \mathit{String}, \mathcal{L}\}$. Here, $\mathcal{L}$ is the set of *labels*. In the AuDaLa semantics, at any time during the execution of $\mathcal{R}$ every existing struct instance has a unique identifying *label* which is unique to that struct instance. The set $\mathcal{L}$ is assumed adequately large to label all potential struct instances that can exist during an execution of $\mathcal{R}$.

In the semantics of $\mathcal{R}$, we define that every single struct type, in this case *Node* and *Edge*, has a *null-instance*. This instance represents the default element of the struct type, and has two main purposes. Firstly, the null-instances are by definition *stable*: their parameter values cannot be changed by writes. This makes it easier to design fixpoints, because writes to null-instances do not impact stability and reads from null-instances have a deterministic effect. Secondly, null-instances are able to execute functions just like normal instances, which is useful during initialization. In $\mathcal{R}$, the step *init* will be executed by the null-instance of *Edge*, which exists from the initial state of the program.

From the syntax of $\mathcal{R}$, it follows that $\mathcal{L}$ should contain at least 10 labels, to facilitate the two null-instances and the eight struct instances created by the null-instance of *Edge* when executing *init*. We also define the special subset $\mathcal{L}^0 \subseteq \mathcal{L}$ to contain the *null-labels*, which represent the null-instances.

The set of all values is $\mathcal{V} = \mathcal{L} \sqcup \mathbb{Z} \sqcup \mathbb{B} \sqcup \mathit{String}$, with $\sqcup$ denoting the disjoint union. Note that we consider $\mathbb{N} \subset \mathbb{Z}$. For a literal g, the semantic value is denoted $\mathit{val}(g)$. We consider the semantic values of literals straightforwardly defined. For example, the semantic value of `5` is $\mathit{val}(\texttt{5})$, which we assume to be 5. The semantic value of an occurrence of `this` in the syntax is defined as the label of the currently executing struct instance, and the semantic value of the null-labels is referred to by ℓ^0_θ for some struct type θ. In $\mathcal{R}$, the null-labels are therefore ℓ^0_{Node} and ℓ^0_{Edge}. We also define a null-value for every syntactic type: 0 for `Nat` and `Int`, *false* for `Bool`, ε, the empty string, for `String` and ℓ^0_T for if $T \in ID$, with T the syntactic type. These null-values can be accessed through a function *DefaultVal*.

Commands and the State. During execution, when executing a step F, we transform the syntax of F to an intermediate language to make the atomic actions executed in the execution of $\mathcal{R}$ explicit, which allows us to keep the operational semantics of AuDaLa simple with regards to step execution. We express these atomic actions as *commands*, which are ordered in the intermediate language in a variant of polish notation, which puts the operators after the values. These commands are then given to the struct instances to execute.

Formally, a struct instance s with some label ℓ_s is a tuple $\langle \theta, \gamma, \chi, \xi \rangle$, where θ is the struct type, γ is a list of commands to be executed, χ is a value stack where intermediate values are stored during execution, and ξ is the variable environment. The commands are given below, with simplified descriptions of the requirements and the actions associated with them (a complete formalisation is given later in this section).

(1) The push command, **push**(v) with value v, causes s to push *val* to its value stack χ. There is a special variant, **push**(**this**), for which s pushes ℓ_s to χ.
(2) The read command, **rd**(x) with variable $x \in Var_{\mathcal{R}} \cup Par_{\mathcal{R}}$, requires a label ℓ on χ to denote the location of v. The associated action is that s reads the variable x in the variable environment of the struct instance of ℓ and puts the resulting value on χ.
(3) The write command, **wr**(x) with variable $x \in Var_{\mathcal{R}} \cup Par_{\mathcal{R}}$, requires a label ℓ on χ to denote the location of x and a value v on χ. The associated action is that s writes the value v to the entry for x in the variable environment of the struct instance of ℓ.
(4) The constructor command, **cons**(θ') with some $\theta' \in \Theta_{\mathcal{R}}$, requires that for every parameter $p_1, \ldots, p_n$ of θ, there is a value $v_1, \ldots, v_n$ on the stack. The associated action is that s initialises a new struct instance $\langle \theta, \varepsilon, \varepsilon, \xi' \rangle$ s.t. $\xi'(p_i) = v_i$ for all $1 \leq i \leq n$ with label ℓ' and puts ℓ' on χ.
(5) The if command, **if**(C) with a command list C, requires a boolean value on χ. The associated action is that s adds the commands in C to γ iff the top value on the value of χ is the value *true*.
(6) The not command, **not**, requires a boolean value on χ. The associated action is that s negates the top value of the value stack.
(7) The operator command, **op**(o) with some binary operator $o \in O$, requires two values compatible with o on χ. The associated action is that s applies o to the top two values of χ.

The set of all commands is denoted $\mathcal{C}$. Note that commands assume certain values on the top of the stack χ. These values are guaranteed to be present if a program is well-formed, as shown in [FNG25, Theorem 7.14]. If the values are not there, the command cannot execute.

The syntax is transformed into commands through the following function:

Definition 2.2 (Interpretation function [FNG25])**.** Let $x, x_1, \ldots, x_n \in ID$ be variables, $E, E_1, \ldots, E_m \in E$ expressions, $g \in LT$ a literal, $\theta \in ID$ a struct type, $S \in ST$ a statement, $\mathcal{S} \in ST^*$ a list of statements, $T \in \mathcal{T}$ a type and $\circ \in O$ an operator from the syntax. Let the list $x_1; ...; x_n$ be the list of n variables from x_1 to x_n. We define the *interpretation function* $[\![\cdot]\!] : ST^* \cup E \to \mathcal{C}^*$ transforming a list of statements or expressions into a list of commands:

$$
\begin{aligned}
[\![g]\!] &= \mathbf{push}(val(g)) \\
[\![\texttt{this}]\!] &= \mathbf{push}(\mathbf{this}) \\
[\![\texttt{null}_T]\!] &= \mathbf{push}(defaultVal(T)) \\
[\![x_1.\cdots.x_n]\!] &= \mathbf{push}(\mathbf{this}); \mathbf{rd}(x_1); \ldots; \mathbf{rd}(x_n) \\
[\![!E]\!] &= [\![E]\!]; \mathbf{not} \\
[\![E_1 \,\texttt{op}\, E_2]\!] &= [\![E_1]\!]; [\![E_2]\!]; \mathbf{op}(\circ) \\
[\![\texttt{if}\ E\ \texttt{then}\{\mathcal{S}\}]\!] &= [\![E]\!]; \mathbf{if}([\![\mathcal{S}]\!]) \\
[\![T\, x := E]\!] &= [\![x_1.\cdots.x_0.x := E]\!] \\
[\![x_1.\cdots.x_n.x := E]\!] &= [\![E]\!]; [\![x_1.\cdots.x_n]\!]; \mathbf{wr}(x) \\
[\![\theta(E_1, \ldots, E_m)]\!] &= [\![E_1]\!]; \ldots; [\![E_m]\!]; \mathbf{cons}(\theta) \\
[\![\varepsilon]\!] &= \varepsilon \\
[\![S; \mathcal{S}]\!] &= [\![S]\!]; [\![\mathcal{S}]\!]
\end{aligned}
$$

Note that if $n = 0$, $[\![x_1.\cdots.x_n]\!] = \mathbf{push}(\mathbf{this})$ as a special case. We require $\mathbf{push}(\mathbf{this})$ to give us the label of the current executing struct instance at the start of dereferencing any pointer.

As an example, interpreting the syntax of the step *reachability* in $\mathcal{R}$ leads to the following commands:

$$\begin{aligned}&\mathbf{push}(\mathbf{this}); \mathbf{rd}(\text{in}); \mathbf{rd}(\text{reach}); \mathbf{push}(\mathit{true}); \mathbf{op}(=); \mathbf{if}(\\ &\quad \mathbf{push}(\mathit{true}); \mathbf{push}(\mathbf{this}); \mathbf{rd}(\text{out}); \mathbf{wr}(\text{reach})\\ &)\end{aligned}$$

Executions are sequences of transitions between *states*. To reflect the current state of the program, states are a tuple $\langle Sc, \sigma, s\chi\rangle$. Here, Sc is the currently still-to-be-executed schedule of the program. The second element σ is a partial function from labels to struct instances, the *struct environment*, which contains all the current struct instance tuples and therefore also keeps track of the variables, their values and which commands still have to be executed by which struct instance. This struct environment is what formally links the labels to the struct instance tuples. The last element $s\chi$ of the state is the *stability stack*, which is used to keep track of the currently executing fixpoints and whether they are stable. States are collected in the *state space* $S_{\mathcal{G}}$.

When starting execution of a fixpoint, a new Boolean value is placed on the stability stack corresponding to this fixpoint. This value is set to true every time the fixpoint iterates, and is set to false whenever a parameter is changed during the execution of the fixpoint. If at the end of a fixpoint iteration the value is still true, the fixpoint terminates and removes the variable from the stability stack.

For example, in $\mathcal{R}$, the first time the schedule reaches $\mathit{Fix}(\mathit{reachability})$, the value *true* will be placed on the stability stack. Then *reachability* is executed. If for some struct instance, out.reach := *true* is executed, the top of the stability stack will be set to *false*. Then at the end of executing *reachability*, another iteration will be started depending on whether the top of the stability stack is *false*. If not, the fixpoint is stable and will terminate.

For every program $\mathcal{P}$, we define an initial state $P^0_{\mathcal{P}}$:

Definition 2.3 (Initial state [FNG25])**.** The *initial state* of $\mathcal{P}$ is $P^0_{\mathcal{P}} = \langle Sc_{\mathcal{P}}, \sigma^0_{\mathcal{P}}, \varepsilon\rangle$, where $\sigma^0_{\mathcal{P}}(\ell^0_\theta) = \langle \theta, \varepsilon, \varepsilon, \xi^0_\theta\rangle$ for all $\theta \in \Theta_{\mathcal{P}}$ and $\sigma^0_{\mathcal{P}}(\ell) = \bot$ for all other labels.

Here, ξ^0_θ is defined as $\xi^0_\theta(p) = \mathit{defaultVal}(T)$ for all $p \in \mathit{Par}(\theta, \mathcal{P})$ where T is the type of p and $\mathit{Par}(\theta, \mathcal{P})$ refers to the parameters of θ in $\mathcal{P}$. For all other variables x, $\xi^0_\theta(x)$ is arbitrary.

In our example, ξ^0_{Node} yields *false* for *reach* and an arbitrary value for all other inputs, and ξ^0_{Edge} yields ℓ^0_{Node} for both *in* and *out* and an arbitrary value for all other inputs. The initial state for $\mathcal{R}$ is then $P^0_{\mathcal{R}} = \langle \mathit{init} < \mathit{Fix}(\mathit{reachability}), \sigma^0_{\mathcal{R}}, \varepsilon\rangle$, where $\sigma^0_{\mathcal{R}}(\ell^0_{\mathit{Node}}) = \langle \mathit{Node}, \varepsilon, \varepsilon, \xi^0_{\mathit{Node}}\rangle$ and $\sigma^0_{\mathcal{R}}(\ell^0_{\mathit{Edge}}) = \langle \mathit{Edge}, \varepsilon, \varepsilon, \xi^0_{\mathit{Edge}}\rangle$ and $\sigma^0_{\mathcal{R}}(\ell) = \bot$ for all other labels.

The Transitions. The execution of a program is expressed as a sequence of transitions between states. The *transitions* are expressed as a transition relation $\Rightarrow_{\mathcal{P}}$ containing fifteen transition rules, defined relative to the executing program $\mathcal{P}$. These come in two categories: transition rules for executing commands and transition rules for dealing with the schedule. They are shown in Figure 1. For the transitions of the schedule, we remark that the schedule is interpreted as a list, even though the rules use the syntax of AuDaLa for schedule notation.

Therefore, for example, the **FixInit** rule is also applicable on a schedule $Fix(sc)$, even though it is not followed by a barrier with another schedule behind it.

Figure 1: The semantics of AuDaLa, ordered by category.

We use (occurrences of) x for variables, p for parameters, v for values and b for Boolean values. The notation $\circ$ refers to a syntactic operator, and we assume that o is a corresponding semantic operator. The notation C refers to a list of commands. $Par(\theta, \mathcal{P})$ refers to the parameters of θ in the syntax of $\mathcal{P}$. If σ is note defined for ℓ, this is denoted as $\sigma(\ell) = \bot$. Steps are denoted with F and occurrences of sc are schedules, as is Sc. The symbol S^F_θ denotes the list of statements in step F in struct θ in $\mathcal{P}$. The symbol $aFix$ is a special fixpoint notation that refers to a fixpoint encountered before but not yet terminated.

```
1  struct TapeCell (left: TapeCell, right: TapeCell, symbol: Int){} //def. of TapeCell
2
3  struct Control (head: TapeCell, state: Int, accepting: Bool) {
4    transition {
5      see Listing 3 and 4        //definition of the step "transition"
6    }
7    init {
8      see Listing 5              //definition of the step "init"
9    }
10 }
11
12 init < Fix(transition)        //schedule: run "init" once and then iterate "transition"
```

Listing 2: The AuDaLa Turing machine structure

The predicate $Done(\sigma)$ is defined as:

$$Done(\sigma) = \forall \ell.(\sigma(\ell) = \bot \vee \exists \theta, \chi, \xi.\sigma(\ell) = \langle \theta, \varepsilon, \chi, \xi \rangle)$$

which intuitively means that no existing struct instance still has a command to execute.

The transition relation for $\mathcal{R}$ is therefore $\Rightarrow_{\mathcal{R}}$. Which program the transition system is defined for impacts the rules **ComWr**, **ComWrNSkip**, **InitG** and **InitL**, as those are dependent on information from the program (parameters and statements in steps respectively).

With this transition system, we can give the full semantics of any program $\mathcal{P}$, which is defined as the tuple $\langle S_{\mathcal{G}}, \Rightarrow_{\mathcal{P}}, P^0_{\mathcal{P}} \rangle$. The semantics for $\mathcal{R}$ is thus $\langle S_{\mathcal{G}}, \Rightarrow_{\mathcal{R}}, P^0_{\mathcal{R}} \rangle$.

3. The Implementation of a Turing Machine in AuDaLa

In this section, we describe the implementation of a Turing machine in AuDaLa. Let $T = (Q, q_0, F, \Gamma, \Sigma, B, \delta)$ be a Turing machine and Z an input string. We implement T and initialize the tape to Z in AuDaLa. We assume w.l.o.g. that $Q \subseteq \mathbb{Z}$ with $q_0 = 0$ and that $\Gamma \subseteq \mathbb{Z}$ with $B = 0$.

We model a cell of T's tape by a struct *TapeCell*, with a left cell (parameter *left*), a right cell (*right*) and a cell symbol (*symbol*). The control of T is modeled by a struct *Control*, which saves a tape head (variable *head*), a state $q \in Q$ (*state*) and whether $q \in F$ (*accepting*). See Listing 2. The step *transition* in the *Control* struct models the transition function δ. For every pair $(q, z) \in Q \times \Gamma$ s.t. $\delta(q, z) = (q', z', D)$ with $D \in \mathbb{D}$, *transition* contains a clause as shown in Listing 3 (assuming $D = R$). This clause updates the state and symbol, and saves whether the new state is accepting. It also moves the head and creates a new *TapeCell* if there is no next *TapeCell* instance, which we check in line 6. Note that we also check whether the head is not null, for while the null-instance of head cannot update its own parameter, it can still make new *TapeCell*s, which is undesirable. For this, as z can be *null*, we need to explicitly check whether *head* is a *null*-instance. Note that $B = 0$, and that if $D = L$ the code only minimally changes.

The clauses for the transitions are combined using an if-else if structure (syntactic sugar for a combination of ifs and variables), so only one clause is executed each time *transition* is executed. See Listing 4. In the step *init* in the *Control* struct, we create a *TapeCell* for every symbol $z \in Z$ from left to right, which are linked together to create the tape. We

```
if (state == q && head.symbol == z) then {
  head.symbol := z';          //update the head symbol
  state := q';                //update the state
  accepting := (q' ∈ F);      //the new state is accepting or rejecting

  if (head != null && head.right == null) then {
    head.right := TapeCell(head, null, 0); //call constructor to create a new TapeCell
  }
  head := head.right;         //move right
}
```

Listing 3: A clause for $\delta(q, z) = (q', z', R)$.

```
transition {
  if (state == q1 && head.symbol == z1) then{
    /*clause 1*/
  }
  else if (state == q2 && head.symbol == z2) then {
    /*clause 2*/
  }
  else if (state == q3 && head.symbol == z3) then {
    /*clause 3*/
  }
  // etc.
}
```

Listing 4: The *transition* step. The shown pairs all have an output in δ.

```
init {
  TapeCell cell0 := TapeCell(null, null, z0);          // initialize the tape
  TapeCell cell1 := TapeCell(null, null, z1);
  TapeCell cell2 := TapeCell(null, null, z2);

  cell1.left := cell0;                                 // connect the tape
  cell0.right := cell1;
  cell2.left := cell1;
  cell1.right := cell2;

  Control(cell0, 0, (q0 ∈ F));                         //initialize the control
}
```

Listing 5: Initializing input string Z.

also create a *Control*-instance. Listing 5 shows this for an example tape $Z = z_0, z_1, z_2$. A call of *init* in the schedule causes the *null*-instance of *Control* to initialize the tape. It also initializes a single non-*null* instance of *Control*. The schedule will then make that instance of *Control* run the *transition* step until the program stabilizes. Listing 2 shows the final structure of the program.

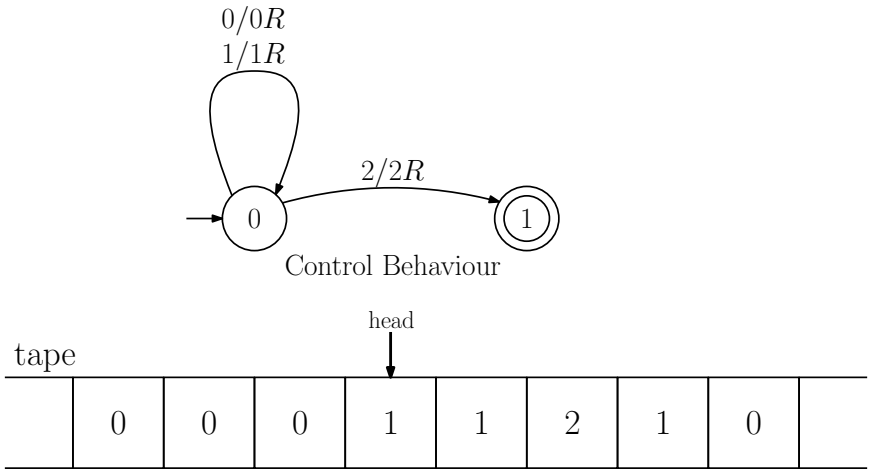

Figure 2: The Turing Machine of Example 3.1 with input string '1121'.

Example 3.1. As an example, consider a Turing machine $T_{ex} = (\{0,1\}, 0, \{1\}, \{0,1,2\}, \{1,2\}, 0, \delta)$, with $\delta(0,0) = (0,0,R)$, $\delta(0,1) = (0,1,R)$ and $\delta(0,2) = (1,2,R)$, with no transitions defined for state 1. This Turing machine walks through the input and stops after finding a 2. In Listing 6, we have created an AuDaLa program for this Turing machine with input string '1121'. The Turing machine with input string '1121' is visualised in Figure 2.

4. Properties of AuDaLa Programs

This section discusses some properties of AuDaLa programs that are relevant for the Turing completeness proof. This section is a partial recap of Section 7 of [FNG25]. In Section 4.1, we recap some standard properties of AuDaLa programs. In Section 4.2 we summarize those results of [FNG25, Theorem 7.14] which are relevant to this paper. This theorem is not included in full because it is out of the scope of this paper.

4.1. Standard Properties.

First, as AuDaLa uses a reference notation of the form $x.y$ to access variable y from the struct instance saved under variable x, we define an *semantic reference notation*, so that we can use the notation freely in both the syntactic and semantic contexts.

Definition 4.1 (Reference Notation)**.** Let σ be a structure environment and let x be a struct instance with label ℓ_x such that $\sigma(\ell_x) = \langle \theta, \gamma, \chi, \xi \rangle$. We define the notation $\ell_x.a^i$ inductively on i, with a variable a:
(1) $\ell_x.a^1 = \xi(a)$,
(2) $\ell_x.a^i = \xi'(a)$ for $i > 1$, with $\ell_x.a^{i-1} \in \mathcal{L}$ and $\sigma(\ell_x.a^{i-1}) = \langle \theta', \gamma', \chi', \xi' \rangle$.
We write $\ell_x.a$ for $\ell_x.a^1$ and $x.a$ for $\ell_x.a$, where $\sigma(\ell_x) = x$ and a is a parameter of x.

To give the following two definitions, we first define that an *execution* of a step F from P is a chain of transitions starting with an **Init** transition for F from P and ending at the first state with struct environment σ s.t. $\mathit{Done}(\sigma)$ holds. We say that P can execute F if such a chain exists from P. Additionally, note that while the AuDaLa semantics does not define which labels get assigned to new struct instances beyond that they should be

```
struct TapeCell (left: TapeCell, right: TapeCell, symbol: Int){} //def. of TapeCell
struct Control (head: TapeCell, state: Int, accepting: Bool) {
  transition {
    if (state == 0 && head.symbol == 0) then{ // transition δ(0,0)
      head.symbol := 0;
      state := 0;
      accepting := false;
      if (head != null && head.right == null) then {
        head.right := TapeCell(head, null, 0);
      }
      head := head.right;
    }
    else if (state == 0 && head.symbol == 1) then{ // transition δ(0,1)
      head.symbol := 1;
      state := 0;
      accepting := false;
      if (head != null && head.right == null) then {
        head.right := TapeCell(head, null, 0);
      }
      head := head.right;
    }
    else if (state == 0 && head.symbol == 2) then{ // transition δ(0,2)
      head.symbol := 2;
      state := 1;
      accepting := true;
      if (head != null && head.right == null) then {
        head.right := TapeCell(head, null, 0);
      }
      head := head.right;
    }
  }
  init {
    TapeCell cell0 := TapeCell(null, null, 1);          // initialize the tape
    TapeCell cell1 := TapeCell(null, null, 1);
    TapeCell cell2 := TapeCell(null, null, 2);
    TapeCell cell3 := TapeCell(null, null, 1);
    cell1.left := cell0;                                 // connect the tape
    cell0.right := cell1;
    cell2.left := cell1;
    cell1.right := cell2;
    cell3.left := cell2;
    cell2.right := cell3;
    Control(cell0, 0, false);
  }
}
init < Fix(transition)      //schedule: run "init" once and then iterate ‘‘transition"
```

Listing 6: The AuDaLa program for T_{ex} with input string 1121.

new, the exact labels which get assigned to new struct instances do not have an impact on the program: they aren't null-labels, and labels which are not null-labels are only used abstractly. We therefore define determinism to not take newly assigned labels into account. The definition of determinism for AuDaLa is then:

Definition 4.2 (Determinism)**.** Let F be a *step* in an AuDaLa program. Then F is *deterministic* for some state P iff P can execute F and there exists exactly one state that is reached by executing F modulo the labels newly assigned to struct instances during F.

We follow it up with a definition for *race conditions* in AuDaLa. We consider an *access* of a variable x to be an application of **ComRd** or **ComWr** with x as the parameter of the command that induced that transition.

Definition 4.3 (Race Conditions)**.** Let F be a step of some AuDaLa program $\mathcal{P}$. Let P be a state of $\mathcal{P}$. Then F contains a *race condition* starting in P iff there is an enabled **InitL** or **InitG** transition for F from P and during the execution of F after taking this **Init** transition, there exist a parameter x which is accessed by two distinct struct instances a and b, with at least one of these accesses writing to x. We call a race condition between writes a *write-write* race condition, and a race condition between a read and a write a *read-write* race condition.

We combine these two concept in the following lemma.

Lemma 4.4 (AuDaLa Determinism [FNG25])**.** *An AuDaLa step F is* deterministic *for some state P if F does not contain a race condition starting in P.*

It follows that when a step is deterministic we can ignore interleaving of struct instances during the execution of the step when determining the effects of the step.

4.2. **Properties of Well-formed AuDaLa Programs.** When programs are well-formed, we know even more about them. In this section, give some results derived from Theorem 7.14 in [FNG25], which are important for the Turing completeness proof. Theorem 7.14 is omitted, as stating it and its proof fully is expansive and has no purpose in this paper.

We define that the state after the execution of an expression E or a statement S by some struct instance s is the state resulting from the transition of the last command from $[\![E]\!]$ or $[\![S]\!]$ as put in the command list of s. Then we know that for any well-formed program $\mathcal{P}$, the following properties hold:

Corollary 4.5 (Progress)**.** *The execution of a well-formed program never gets stuck unless it terminates: every sequence of transitions from a state P_1 with struct environment σ_1 s.t. $Done(\sigma_1) = false$ eventually reaches a state P_1' with struct environment σ_1' s.t. $Done(\sigma_1') = true$, and every sequence of transitions from a state P_2 with struct environment σ_2 s.t. $Done(\sigma_2) = true$ eventually reaches a state P_2' with struct environment σ_2 where either the schedule is empty or $Done(\sigma_2') = false$.*

Corollary 4.6 (Expression Results)**.** *Every expression E, when executed by s during a step F in a well-formed program $\mathcal{P}$, results in a single value v on the stack of s. This value is of the semantic type expected from the syntax of E and it is deterministic if there are no read-write race conditions.*

Corollary 4.7 (Execution Effects)**.** *Let S be a statement executed during the execution of a step F in a well-formed program $\mathcal{P}$ by a struct instance s. Let P' be the state resulting from the transition induced by the last command of $[\![S]\!]$ for s. Then:*

(1) *If S is an update statement $x_1.\ldots.x_n.x := E$, then, with x the variable in the struct instance belonging to a result of an execution of $x_1.\ldots.x_n$, if x is not a parameter of a null-instance, x will be updated to a result of an execution of E in P', and all values in the stability stack will be reset to false if this means that a parameter has changed. If x is a parameter of a null-instance, E is executed and its effects are still present in P', but x has not been updated.*
(2) *If S is a variable assignment statement $T\ x := E$, then it has the same effect as executing the update statement $x := E$.*
(3) *If S is an if-statement, in P', E will be executed to a result b. If $b = true$, the if-clause will be taken. If $b = false$, the if-clause will be skipped.*
(4) *If S is a constructor statement $\theta(E_1, \ldots, E_m)$, then in P', there will be a new struct instance for a fresh label ℓ with type θ and its parameters set to results of executions of $E_1, \ldots, E_m$, and all values in the stability stack will be reset to false. Additionally, ℓ will be the last value on the stack of s.*

Additionally, let $\mathcal{E}$ be the expressions executed as a part of S. For any expression $E \in \mathcal{E}$, if E is a constructor expressions $\theta(E_1, \ldots, E_m)$, its result will be a fresh label ℓ' s.t. in P', there is a struct instance for ℓ' with type θ and its parameters set to results of executions of $E_1, \ldots, E_m$. Furthermore, if there exists a constructor expression E in $\mathcal{E}$, all values in the stability stack will be reset to false. Lastly, the effects of a statement will be deterministic if all results from all expressions in S are deterministic.

These corollaries and results allow us to simplify the proof of AuDaLa programs by keeping the proof on the level of the syntax.

5. Turing Completeness

In this section, we show that AuDaLa is Turing complete. To do this, we establish an equivalence between the results of transitions taken by T with input string Z and the execution the *transition* step of its implementation. Henceforth, let $\mathcal{P}_{(T,Z)}$ be the implementation of a Turing machine T with an input string $Z = z_0 \ldots z_n$ as specified in Section 3. As $\mathcal{P}_{(T,Z)}$ is well-typed and satisfies the requirements given in Section 2.2, $\mathcal{P}_{(T,Z)}$ is well-formed.

As $\mathcal{P}_{(T,Z)}$ is well-formed, it does not get stuck, according to Corollary 4.5. It follows that the execution of $\mathcal{P}_{(T,Z)}$ consists of the execution of a step *init* from the initial state, and then executions of the step *transition*. Between these executions, there will be one or more state with a struct environment σ s.t. $Done(\sigma)$ holds. These states we call *idle states*. For these idle states, we define *implementation configurations*:

Definition 5.1 (Implementation Configuration)**.** Let P be an idle state of $\mathcal{P}_{(T,Z)}$ containing a single non-*null* instance c of *Control*. Then we define the *implementation configuration* of P as a tuple (q_P, t_P) s.t. q_P is the value of the *state* parameter of c and $t_P : \mathbb{Z} \to \mathbb{Z}$ defined as:

$$t_P(i) = \begin{cases} c.head.symbol & \text{if } i = 0 \\ c.head.left^{-i}.symbol & \text{if } i < 0 \\ c.head.right^{i}.symbol & \text{if } i > 0 \end{cases},$$

For our equivalence, we compare the Turing machine configuration of T with input string Z after doing transitions to the implementation configuration of $\mathcal{P}_{(T,Z)}$. As the implementation configuration is defined on the struct environment of idle states only, we will abstract from which idle state we take between executions of the *transition* step, as they will all have the same configuration: there is no semantic transition that changes the struct environment enabled when the *Done*-predicate holds. We therefore abstract to collections of idle states between step executions.

To prove AuDaLa Turing complete, we have to prove, as a base case, that after the initialization we find ourselves into a collection of idle states with the initial configuration of T with Z as its implementation configuration. We also have to prove as a step case that every transition taken by either T with Z or $\mathcal{P}_{(T,Z)}$ can be exactly mirrored by $\mathcal{P}_{(T,Z)}$ or T with Z s.t. after the transition, the implementation configuration is again equal to the Turing machine configuration of T with Z. To do this, we also have to prove that after any transition taken, there is only a single non-null *Control* instance in the idle states reached. From this, it follows by induction that $\mathcal{P}_{(T,Z)}$ and T with Z have equivalent behaviour, so $\mathcal{P}_{(T,Z)}$ is a correct implementation of T with Z and AuDaLa is Turing complete.

To prove that after initialization, the collection of idle states reached have the initial Turing machine configuration of T with Z as their implementation configuration, we first prove that the step *init* is deterministic. We then use this to prove the exact result of executing *init*, by using Corollary 4.7.

Lemma 5.2. *The execution of* init *in* $\mathcal{P}_{(T,Z)}$ *is deterministic.*

Proof. The step *init* is only executed once, at the start of the program, by the null-instance of *Control* (as no other instances exist). As only one instance exists, there cannot be a race condition between two struct instances. Therefore *init* contains no race conditions. It then follows by Lemma 4.4 that *init* is deterministic. □

Lemma 5.3 (Executing *init* in the initial state)**.** *Let* $P^0_{\mathcal{P}_{(T,Z)}}$ *be the initial state at the start of executing* $\mathcal{P}_{(T,Z)}$ *and let the input string* $Z = z_0 \dots z_n$*. Executing the step init on* $P^0_{\mathcal{P}_{(T,Z)}}$ *results in a state* P_1 *with a single non-null Control instance such that* (q_0, t_Z) *is the implementation configuration of* P_1*.*

Proof. First, note that the initial state for $\mathcal{P}_{(T,Z)}$, as defined in the AuDaLa semantics, contains the schedule of $\mathcal{P}_{(T,Z)}$, the null-instances of all structs, and a stability stack. The stability stack has no bearing on this proof, and will be disregarded. From Lemma 5.2, we know that as only the null-instance of *Control* executes *init*, the execution of *init* is deterministic.

Then let P_1 be an idle state reached by executing *init* from $P^0_{\mathcal{P}_{(T,Z)}}$. As written in line 43 of $\mathcal{P}_{(T,Z)}$, the null-instance of *Control* will create one additional *Control* instance. As no other struct instances execute *init*, it follows that there will be a single non-null *Control* instance in P_1. Let this instance be c. From line 43 of Listing 6, we also know that c will have its state set to 0, which is the representation of q_0.

We then prove that the function made according to Definition 5.1 in P_1 from the *TapeCells* is t_Z. Due to Lemma 5.2, we can walk through Listing 5 sequentially. Then, through multiple applications of Corollary 4.7, we know that the first part makes one *TapeCell* instance for every $z_i \in Z$, and the second part then connects these to the correct *left* and *right* neighbours. It follows that in P_1, there exists a *TapeCell* for all symbols $z_i \in Z$, and no others, s.t. every *TapeCell* z_i is be connected to z_{i-1} and z_{i+1} (if they exist) through

parameters *left* and *right* respectively. The last line of *init* then sets the *head* parameter of c to the *TapeCell* for z_0. It follows that the function made according to Definition 5.1 is indeed t_Z.

Then the lemma holds: the implementation configuration of P_1 is (q_0, t_Z). □

We then prove that taking a transition in the $\mathcal{P}_{(T,Z)}$ has the same effect as taking the same transition in T with input string Z. We do this by first establishing that the step *transition* taken from a state with a single non-null *Control* instance is deterministic, after which we prove that all executions of *transition* taken in $\mathcal{P}_{(T,Z)}$ are deterministic (as there will always be only a single *Control* instance). We then prove that executing some transition from a configuration (q, t) in $\mathcal{P}_{(T,Z)}$ has the same resulting configuration as executing the same transition from (q, t) in T with Z. As a part of this, we also prove that no new *Control* instances are made, as that would mean that the idle states reached would not have an implementation configuration.

Lemma 5.4. *Let P be an idle state reachable in $\mathcal{P}_{(T,Z)}$ with a single non-null Control instance. Any execution of* transition *executed from P is deterministic.*

Proof. As per Lemma 4.4, it suffices to prove that the execution contains no race conditions. Due to the definition of *transition* (Listing 3), only one clause of the step will be executed. Let c be this clause. As there is only one non-null *Control* instance, let c_0 be the null-instance of *Control* and let c_1 be the non-null instance of *Control*.

During the execution of *transition* by the null-instance of *Control* c_0, all of the updates executed by c_0 will be updates to parameters of null-instances (recall that $c_0.head = null$). As there is only one non-null *Control* instance c_1, any race condition must involve both c_0 and c_1, and must then be a race for a null-parameter. However, race conditions require at least one write to be performed to the variable and writes to null-parameters are not performed (as per the semantic rule **ComWrNSkip**). It follows that the execution contains no race conditions. □

Lemma 5.5. *Every* transition *step executed in $\mathcal{P}_{(T,Z)}$ is deterministic.*

Proof. We prove by induction that every execution of *transition* in $\mathcal{P}_{(T,Z)}$ starts from a state with a single non-null *Control* instance. As a base case, the first execution of *transition* starts from P_1 as defined in Lemma 5.3, which has a single non-null *Control* instance.

Then, let P_i be a state with a single non-null *Control* instance from which *transition* is executed for the ith time. Due to Lemma 5.4, we know that the execution of *transition* is deterministic, so we can consider the sequential execution of *transition*. The step *transition* is made up of multiple mutually exclusive clauses by definition. Let C be the clause executed, w.l.o.g., for this execution of *transition*. By the definition of C, in none of the statements in C a *Control* instance is created, as seen in Listing 3. It follows that the state P_i' resulting from the execution of *transition* will also have only a single non-null *Control* instance. By Corollary 4.5, we know that eventually, from P_i', a state P_{i+1} is reached s.t. either the schedule is empty or *transition* is executed again. If the schedule is empty in P_{i+1}, every execution of the step *transition* in $\mathcal{P}_{(T,Z)}$ will have been executed from a state with a single non-null *Control* instance. Else, note that the Schedule Transitions as defined in the semantics of $\mathcal{P}_{(T,Z)}$ (Figure 1) do not change the struct environment, so P_{i+1} will again have a single non-null *Control* instance.

It follows that every step *transition* executed in $\mathcal{P}_{(T,Z)}$ will be executed from a state with a single non-null *Control* instance. By Lemma 5.4, the lemma follows. □

Lemma 5.6 (Effect of a *transition* execution)**.** *Let P be an idle state of $\mathcal{P}_{(T,Z)}$ from which* transition *can be executed and let (q,t) be the implementation configuration of P. Let (q,t) also be the current Turing machine configuration of T with input string Z. Then the result of a transition in T is a configuration (q',t') iff the result of executing the* transition *step from P in $\mathcal{P}_{(T,Z)}$ is an idle state P' such that (q',t') is its implementation configuration.*

Proof. Let P and (q,t) be as defined in the lemma. Then by definition of $\mathcal{P}_{(T,Z)}$, there exists a single transition that can be executed from P for $\mathcal{P}_{(T,Z)}$ iff $\delta(q,t)$ is defined. Additionally, δ is a partial function, so if $\delta(q,t)$ is defined for (q,t) it is always uniquely defined for (q,t).

Let $\delta(q,t) = (q',s',D)$. Let $D = R$ (the proof of $D = L$ is analogous). Then as a transition in T with Z is deterministic, by Definition 2.1, the resulting Turing machine configuration of taking a transition from (q,t) is (q',t'), with

$$t'(i) = \begin{cases} z' & \text{if } i = -1 \\ t(i+1) & \text{otherwise} \end{cases}.$$

Taking the transition from P for $\mathcal{P}_{(T,Z)}$ is also deterministic (Lemma 5.5), and therefore we can walk through the statements of the clause to determine its effect. By definition of $\mathcal{P}_{(T,Z)}$, this clause is based on the template shown in Listing 3. Let c be the single *Control* instance that exists in P, and let h be the *TapeCell* instance which is referenced in the *head* parameter of c. As per multiple applications of Corollary 4.7, the result of the transition is that the *state* of c is updated to q', the *symbol* of h is updated to z', the *accepting* parameter of c is updated to whether $q' \in F$ and that the *head* parameter of c shifts one *TapeCell* to the right (making a new *TapeCell* if required).

Creating a function of the *TapeCells* as in Definition 5.1 then results in function t'', s.t. $t''(-1) = z'$ and $t''(i) = t(i+1)$ for all $i \neq -1$. It follows that $t'' = t'$. Then, by Definition 5.1, the implementation configuration of the resulting state is (q',t'). □

Theorem 5.7. *AuDaLa is Turing complete.*

Proof. We prove this by induction, with as base case that after initialization, both the $\mathcal{P}_{(T,Z)}$ and the turing machine T with input string Z have configuration (q_0, t_Z). For T with Z, this follows by definition, and for $\mathcal{P}_{(T,Z)}$, this follows from Lemma 5.3. Then, as the step case, we prove that when T with Z and $\mathcal{P}_{(T,Z)}$ both start from configuration (q,t) and take the same transition, both end up in configuration (q',t'). This follows from Lemma 5.6. We conclude that T with Z and $\mathcal{P}_{(T,Z)}$ have the same behaviour, and therefore $\mathcal{P}_{(T,Z)}$ is a correct implementation of T with Z in AuDaLa. Therefore, AuDaLa is Turing complete. □

6. Adaptations

In this section, we discuss several potential adaptations for the AuDaLa semantics which increase practical expressivity. First, we discuss the adaptation of the schedule to allow for more diverse looping structures, and then we will discuss the addition of array-like structures to AuDaLa. This will include additions to the syntax and semantics of AuDaLa. Here, we consider the adaptations in this section separately from each other, and that for all adaptations we extend the original semantics of AuDaLa, as discussed in Section 2.2. At the end of the section, we include a short discussion on these additions.

6.1. **Schedule Adaptations.** In AuDaLa, the schedule controls the flow of the program, using steps, barriers and fixpoints, as discussed in Section 2.2.

The fixpoint has two main components as a looping structure. First, it requires synchronisation between all struct instances in each iteration of the loop. Second, it requires that the whole system is stable for the loop to terminate. For the schedule adaptations, we will first relax the second requirement, introducing a possible addition of *parameter-specific fixpoints*, which only require certain parameters to be stable. Then, we will relax the first requirement and give a possible addition of *iterators*, which removes the synchronisation requirement for fixpoints.

6.1.1. *Parameter-specific fixpoints.* In AuDaLa, fixpoints require that after the execution of the fixpoint every parameters in the entire system is stable. This is convenient theoretically, as this is a strong guarantee for the contents of a state after fixpoint termination. In practice, however, it could be useful to limit the relevant parameters for the fixpoint.

To illustrate this, we have created a program based on the reachability example given earlier. It is displayed in Listing 7. The goal of this program is not only to compute which nodes are reachable, but also to count the iterations necessary to do so. The program does this by counting the iterations of the fixpoint in a parameter *count*.

This program, however, will never become stable, as *count* is a parameter. Furthermore, as the value of *count* stays relevant between iterations of the fixpoint, *count* must be a parameter. The only way to mitigate that this program never becomes stable, is to make the program more complicated by computing stability of the graph during the step reachability and somehow making that available to all edges, which would be undesirable. To that end, it may be useful to limit the relevant parameters for the fixpoint to exclude the parameter *count*, which is not stable by design.

To limit the relevant parameters for a fixpoint, we add *parameter-specific fixpoints*: special fixpoints which allow us to specify the parameters the fixpoint requires to be stable. Note that we still consider fixpoints unstable when new struct instances are made during the execution of an iteration of the fixpoint. For this, we expand the syntax of schedules to allow fixpoints of the form: "$Fix(sc : x_1, x_2, \ldots)$", where sc is the subschedule to be executed and $x_1, x_2, \ldots$ are the variables which to be stable for the fixpoint to terminate. Note that this could be straightforwardly extended in the case the variables need to be uniquely identified with their struct types. In our example in Listing 7, we replace the fixpoint with "$Fix(reachability : reach)$", which makes the program work as intended.

To allow the parameter-specific fixpoints in the syntax of AuDaLa, we have to add an additional entry '`Fix`'`(`'⟨*Sched*⟩'`:`'⟨*Vars*⟩'`)`' to the syntax for the schedule, which represents a parameter specific fixpoint. Here, ⟨*Vars*⟩ gives the parameters relevant for this fixpoint, with ⟨*Vars*⟩ ::= ⟨*Var*⟩⟨*AVars*⟩ and ⟨*AVars*⟩ ::= '`,`'⟨*Var*⟩⟨*AVars*⟩ | ε.

Let $\mathcal{P}$ be a program for which the semantics is defined. We then change the definition of the semantics as follows to be able to use parameter-specific fixpoints. First, we adapt the state to include a *stability function*, which is a total function which keeps track of which fixpoints depend on which variables. This function has type $\mathbb{N} \to Par_{\mathcal{P}} \to \mathbb{B}$ and is initialized as *false* for all input pairs $(i, x) \in \mathbb{N} \times Par_{\mathcal{P}}$. This function is called during a write to check whether this write is relevant for the stability of the fixpoints. To avoid unnecessary complication, we assume that every parameter name is unique; to remove this assumption, make the stability function type $\mathbb{N} \to (Type \times Par_{\mathcal{P}}) \to \mathbb{B}$ instead. The function sf can be seen as a stack of functions, where $sf(k)$ denotes which parameters are relevant

```
struct Node (reach: Bool) {}

struct Edge (in: Node, out: Node, diam: Int) {
  reachability {
    count := count + 1;
    if (in.reach = true) then {
      out.reach := true;
    }
  }
  init {
    Node node1 := Node(true);
    Node node2 := Node(false);
    Node node3 := Node(false);
    Node node4 := Node(false);
    Node node5 := Node(false);

    Edge edge12 := Edge(node1, node2, −1);
    Edge edge13 := Edge(node1, node3, −1);
    Edge edge23 := Edge(node2, node3, −1);
    Edge edge34 := Edge(node3, node4, −1);
    Edge edge51 := Edge(node5, node1, −1);
  }
}

init < Fix(reachability)
```

Listing 7: Counting Reachability.

for the kth fixpoint currently initialized. The new state is then defined as a four-tuple $\langle Sc, \sigma, s\chi, sf \rangle$, with Sc the schedule, σ the struct type environment, $s\chi$ the stability stack and $sf : \mathbb{N} \to Par_{\mathcal{P}} \to \mathbb{B}$ the *stability function*.

For our initial state, we define the function sf^0, which returns false for all input pairs $(i, x) \in \mathbb{N} \times Par_{\mathcal{P}}$, which is added as the fourth element of the initial state.

For the transition rules, we define the function $sf_t^0 : Par_{\mathcal{P}} \to \mathbb{B}$, which returns true for all inputs $x \in Par_{\mathcal{P}}$. We change the rule **ComWr** to the following rule **ComWrN**, which ensures that the stability stack is only reset when the relevant variables are changed. The changed parts of the rule are highlighted.

$$(\textbf{ComWrN})\ \frac{\begin{array}{c}\sigma(\ell) = \langle \theta, \mathbf{wr}(x); \gamma, \chi; v; \ell', \xi \rangle \\ \sigma(\ell') = \langle \theta', \gamma', \chi', \xi' \rangle \\ \ell' \notin \mathcal{L}^0 \vee x \notin Par(\theta', \mathcal{P}) \\ su = (x \notin Par(\theta', \mathcal{P}) \vee \xi'(x) = v)\end{array}}{\begin{array}{l}\langle Sc, \sigma, s\chi, \boxed{sf} \rangle \Rightarrow \langle Sc, \sigma[\ell \mapsto \langle \theta, \gamma, \chi, \xi \rangle][\ell', 4 \mapsto \xi'[x \mapsto v]], \\ \qquad s\chi_1 \wedge \boxed{(su \vee \neg sf(1)(x))}; \ldots; s\chi_{|s\chi|} \wedge \boxed{(su \vee \neg sf(|s\chi|)(x))}, \boxed{sf} \rangle\end{array}}$$

In **ComWrN**, as in **ComWr**, su is true when x is not a parameter changed by the application of this rule. However, due to the addition of parameter-specific fixpoints, even if x is a changed parameter, whether the entries of the stability stack need to be reset to *false* also depends on whether x is relevant for the entries. Let $s\chi_k$ be one of the entries of the

stability stack. Then whether x is relevant for the kth currently initialized fixpoint is saved in $sf(k)(x)$: if x is relevant, $sf(k)(x) = true$. Then $s\chi_k$ should only be reset to false (if it isn't false already) if x is a relevant, changed parameter, which is reflected in updating $s\chi_k$ to $s\chi_k \wedge (su \vee sf(k)(x))$.

We also adapt the rule **FixInit** into the rules **FixInitG** (for the general case) and **FixInitS** (for parameter-specific fixpoints), which properly initiates the function sf for every fixpoint. Let $X \subseteq Par_{\mathcal{P}}$ be some list of parameters. The differences between the new rules and **FixInit** are highlighted.

$$(\textbf{FixInitG})\ \frac{Done(\sigma) \quad k = |s\chi| + 1}{\langle Fix(F); F_1, \sigma, s\chi, \ sf \ \rangle \Rightarrow \langle F; aFix(F); F_1, \sigma, s\chi; true, \ sf[k \mapsto sf^0_t] \rangle}$$

$$(\textbf{FixInitS})\ \frac{Done(\sigma) \quad k = |s\chi| + 1 \quad X = x_1, \ldots, x_n \quad x_1, \ldots, x_n \in Par_{\mathcal{P}} \quad sf' = sf[x \mapsto sf^0_f[\{x_1 \mapsto true, \ldots, x_n \mapsto true\}]]}{\langle Fix(F : X); F_1, \sigma, s\chi, \ sf \ \rangle \Rightarrow \langle F; aFix(F); F_1, \sigma, s\chi; true, \ sf' \ \rangle}$$

In these rules, we correctly initialize normal fixpoints to fixpoints dependent on every parameter (as shown by setting $sf(k)$ to sf^0_t) and parameter specific fixpoints to fixpoints only dependent on the given parameters and no others (as shown by the definition of sf' and overwriting $sf(k)$ with sf'. We adapt the rules **FixIter** and **FixTerm** into the rules **FixIterN** and **FixTermN** as follows. Differences are again highlighted.

$$(\textbf{FixIterN})\ \frac{Done(\sigma)}{\langle aFix(sc) < sc_1, \sigma, s\chi; false, \ sf \ \rangle \Rightarrow_{\mathcal{P}} \quad \langle sc < aFix(sc) < sc_1, \sigma, s\chi; true, \ sf \ \rangle}$$

$$(\textbf{FixTermN})\ \frac{Done(\sigma)}{\langle aFix(sc) < sc_1, \sigma, s\chi; true, \ sf \ \rangle \Rightarrow_{\mathcal{P}} \langle sc_1, \sigma, s\chi, \ sf \ \rangle}$$

These rules are not significantly different from **FixIter** and **FixTerm**, as both resetting and initializing levels of sf happens simultaneously in **FixInitG** and **FixInitS**. For the command rules, we then also add sf to both the state before and after the transition, much like in **FixIterN** and **FixTermN**.

6.1.2. *Iterators.* In AuDaLa, fixpoints require that between iterations, there is a synchronisation for all struct instances, through the use of the *Done*-predicate in the transition rules. This is due to the fact that the rules **FixIter** and **FixTerm** require the *Done*-predicate to be true for the state it is taken from in the semantics of AuDaLa (see also Section 2.2). This makes the program more understandable, as it provides a strong guarantee for the status of variables between the executions of the fixpoint.

However, this also inhibits performance, as synchronisation leads to overhead and there are programs for which a strong guarantee on the status of variables between executions is not necessary for understandability. For example, consider again Listing 1 for computing reachability. While we could synchronise between the different iterations of the reachability step, this does not lead to a better understanding, less write-write race conditions or other benefits. Though a straightforward extension may be an alternative barrier which does not

require synchronisation, we believe that this will remove much of the structure and simplicity of AuDaLa. Therefore, we instead introduce an unsynchronised loop: the *iterator*, where the lack of synchronisation is contained within the iterator only.

The iterator takes the form $\mathit{Iter}(F_1; F_2; F_3; \ldots)$, with every occurrence F_i a step name. For this extension, we do not allow iterators or fixpoint nested into iterators (though an iterator nested into a fixpoint is allowed). During execution, the iterator will loop over these steps. With $\mathcal{S}_i$ the statements of the step F_i, every iteration, all struct instances execute the commands generated by $[\![\mathcal{S}_1; \mathcal{S}_2; \mathcal{S}_3; \ldots]\!]$, if they have any. If any struct instances which had commands are finished executing their commands, the iterator can check whether the system is stable. If it is not, then the iterator will make sure that for all struct instances that have to execute commands, there is at least one more iteration of the commands at the end of the command list. The iterator terminates when there are no more commands to execute and the system is stable. This results in a loop where struct instances asynchronously execute the sequence of steps included in the iterator until the system is stable.

For example, if we were to execute the step *reachability* with an iterator, we would put "*Iter*(*reachability*)" in the schedule. This would cause the struct instances to asynchronously execute multiple iterations of the step *reachability*, until the system is stable. Here, as the struct instances of *Node* do not execute any commands for the step *reachability*, they are not taken into account for the iterator. If we had another step *step*, we could then also call "*Iter*(*reachability*; *step*)". This would cause the struct instances to asynchronously execute multiple iterations of the statements of *reachability* followed by the statements of *step* (if the struct instance has commands for these steps), until the system is stable.

To implement our iterator, we extend AuDaLa's syntax by adding the option '`Iter`' '`(`'⟨*StepList*⟩'`)`' for the schedule, where ⟨*StepList*⟩ ::= ⟨*Id*⟩ | ⟨*Id*⟩; ⟨*StepList*⟩, where the ⟨*Id*⟩ occurrences are type checked to be step names. This step list holds the steps to be executed during the iterator, which are executed without synchronisation in between.

In the remainder of this section, let $\mathcal{P}$ be some program. We need to add transition rules to be able to implement the iterator, which are based on the transition rules for fixpoints. First, we add a rule to initialize iterators. This rule, **IterInit**, initialises the iterator and also introduces the meta-schedule element *aIter*, which denotes an iterator which has been started but has not finished yet. We use F^+ to denote a list of one or more steps. We define the function $I(\ell, F^+)$ with $\ell \in \mathcal{L}$ and $F^+ = F_1; \ldots; F_n$, which returns the list of commands $[\![S^{F_1}_{\theta_\ell}]\!]; \ldots; [\![S^{F_n}_{\theta_\ell}]\!]$. Recall from Section 2.2 that $\mathcal{C}$ is the set of all commands. Changes from the rule **FixInit** are highlighted.

$$(\textbf{IterInit})\ \frac{\mathit{Done}(\sigma)}{\langle\ \mathit{Iter}(F^+)\ < sc_1, \sigma, s\chi\rangle \Rightarrow_{\mathcal{P}} \langle\ \mathit{aIter}(F^+)\ < sc_1,\ \sigma[\{\ell \mapsto \langle\theta_\ell,\ I(\ell, F^+)\ , \varepsilon, \xi_\ell, \mathit{true}\rangle \mid \sigma(\ell) = \langle\theta_\ell, \gamma_\ell, \chi_\ell, \xi_\ell\rangle\}], s\chi; \mathit{true}\rangle}$$

In the second rule, we define when an iterator needs to do another iteration. This rule works as follows: When a struct instance is done executing all steps of the iterator, and the stability stack is false, then either some other struct instance caused this instability or the current struct instance did. As we cannot discount either option, it follows that all struct instances should execute the steps from the iterator at least one more time before the iterator can be considered stable. Therefore, the rule tops up the commands of all struct instance s.t. their command lists end in exactly one full list of commands as generated by the steps in the

iterator. The rule also sets the stability stack to *true*. This is safe: every struct instance will execute the steps at least one more time, so the execution of the iterator cannot stop before at least one full iteration has happened in which no parameter was changed and no struct instances were created. This means that the iterator successfully computes a fixed point when it terminates with this iteration transition rule. Note that in this rule, ℓ' is universally quantified.

$$(\textbf{IterIter})\frac{\exists \ell \in \mathcal{L}.(\sigma(\ell) = \langle \theta, \varepsilon, \chi, \xi\rangle \wedge I(\ell, F^+) \neq \varepsilon)}{\begin{array}{l}\langle aIter(F^+) < sc_1, \sigma, s\chi; false\rangle \Rightarrow_{\mathcal{P}} \langle aIter(F^+) < sc_1, \\ \quad \sigma[\{\ell' \mapsto \langle \theta_{\ell'}, \gamma_{\ell'}; I(\ell', F^+), \chi_{\ell'}, \xi_{\ell'}\rangle \mid \\ \quad \sigma(\ell') = \langle \theta_{\ell'}, \gamma_{\ell'}, \chi_{\ell'}, \xi_{\ell'}\rangle \wedge \nexists \gamma' \in \mathcal{C}^*.(\gamma_{\ell'} = \gamma'; I(\ell', F^+))\}], s\chi; true\rangle\end{array}}$$

In the third rule, which can be taken when the iterator is stable and every struct instance is done, the iterator terminates.

$$(\textbf{IterTerm})\frac{Done(\sigma)}{\langle aIter(F^+) < sc_1, \sigma, s\chi; true\rangle \Rightarrow_{\mathcal{P}} \langle sc_1, \sigma, s\chi\rangle}$$

With the first rule, we satisfy that the iterator requires a synchronisation before starting its execution. In the second rule, we allow struct instances to start new iterations of the iterator step without synchronisation, and with the third rule, we require a synchronisation before the loop terminates. This satisfies the first requirement. The commands put on the command lists of struct instances in the first and second rules are generated from the entire subschedule inside the iterator. As commands are run asynchronously, it follows that the second requirement also holds. The third requirement holds because AuDaLa syntax does not permit fixpoints or iterators nested in iterators when expanded with the syntax additions given above. It follows that these extensions correctly implement the iterator as intended into the semantics of AuDaLa.

We can use the iterator to create better code for AuDaLa reachability, by changing Listing 1's schedule to the schedule given below:

```
1 init < Iter(reachability)
```

Listing 8: New Schedule for Listing 1

6.2. **Arrays.** Arrays offer a quick way of grouping elements which allows for quick access due to insights in how things are stored in the memory. AuDaLa does not naturally support dynamically-sized arrays. This is due to the fact that while AuDaLa does not assume a memory implementation, it does require that all struct instances only have a constant-sized amount of memory to their disposal, as AuDaLa focuses on small data elements.

As arrays are widely used in other programming languages and algorithms, supporting arrays may make it easier for people used to those other languages and algorithms to adopt AuDaLa. Furthermore, it may make it easier to translate existing implementations to AuDaLa. It would allow open up the possibility to leverage the constant time access arrays are known for in AuDaLa. Therefore, we propose one way to extend AuDaLa with support

for arrays. This would allow people to use arrays with AuDaLa, and would allow people to leverage the constant time access inherent to them as well.

To adopt arrays in AuDaLa, we introduce a new, special *array instance*. The idea is that this array instance owns an array and saves the array starting address and size. As with conventional arrays, while the size may be chosen dynamically upon initialization, it may not be changed after the array has been initialized. The array is then filled with labels of struct instances, functioning as links to those struct instances. Each element can be accessed through indirect access. Note that this keeps the assumption of constant-sized memory intact for non-array instances, which we prefer over just building in arrays by abolishing the assumption entirely. In this extension, we also consider every entry of an array to be a parameter, to keep the extension simple.

First, we add the new expression '`array`'($\langle Exp \rangle$) to the syntax of AuDaLa. This expression reflects the creation of a new array. The expression contained in it represents the size of the array. When type checking, the inner expression should resolve to a natural number larger than 0, and the whole program should again be well-typed.

When accessing variables, we allow for variable accesses of the type $a[E]$, where $a[E]$ is an array access with E an expression. Accesses can be chained with other variable accesses when necessary. This is reflected by changing the syntax for variables to

$$\langle Var \rangle ::= \langle Var \rangle.\langle Id \rangle \mid \langle Var \rangle\text{`[']}\langle Exp \rangle\text{`]'} \mid \langle Id \rangle$$

We add the type checking requirement that the expression in the square brackets has to resolve to a natural number and that if there is an array access $a[E_1] \cdots [E_n]$ in the syntax, a must be of type $Array_1(\cdots(Array_n(T)))$ for some T. We also require updates of arrays to be well-typed. Lastly, we consider "`array`" to be a protected word in the syntax.

We also allow for a new expression $\langle Var \rangle$'`.s`', which returns the size of the array given for $\langle Var \rangle$. During type checking, we require $\langle Var \rangle$ to refer to an array.

Note that in this extension, we have not included array initializations like "Int[2] $a := \{0, 1\}$", as we would like to keep our commands as simple as possible. We are therefore not inclined to introduce some kind of multiwrite command. This means that the commands generated from a statement like the one given above would not be significantly different from the statements "Array(Int) $a :=$ array(2); $a[0] := 0$; $a[1] := 1$;".

Let $\mathcal{P}$ be an arbitrary program. We create a new set $Types^a_{\mathcal{P}} = \Theta_{\mathcal{P}} \cup \{\mathbb{N}, \mathbb{Z}, \mathbb{B}, String\} \cup Types^1_{\mathcal{P}} \cup Types^2_{\mathcal{P}} \cup \ldots$, where

$$Types^1_{\mathcal{P}} = \{Array(\theta) \mid \theta \in \Theta_{\mathcal{P}}\} \cup \{Array(\mathbb{N}), Array(\mathbb{Z}), Array(\mathbb{B}), Array(String)\}$$

and $Types^i_{\mathcal{P}} = \{Array(T) \mid T \in Types^{i-1}_{\mathcal{P}}\}$ in the semantics. We also define that in the semantics, arrays are labelled using the set $\mathcal{L}^a \subset \mathcal{L}$.

Arrays are dependent on the memory. To that end, and to keep the struct environment separate from hardware dependent memory, we add a *memory function* $\mathcal{M} : \mathcal{A} \rightharpoonup \mathcal{V}$ to the semantic state, with $\mathcal{A}$ a set of memory addresses and $\mathcal{V}$ the set of all values (with $\mathcal{A} \subset \mathbb{N}$). This function models the hardware memory, where the addresses for which it is defined are allocated addresses. We also define that our set $\mathbb{N}$ must include the number 0. We consider the empty address to be an abstract, non-existent address α_0.

We define an *array instance* a as a tuple $\langle \alpha, s \rangle$, which contains the address α at which a starts and the size s of a. We save array instances in the struct environment (and extend

the struct environment to facilitate this). Lastly, we extend the *null*-instances with a *null*-instance $\langle \alpha_0, 0 \rangle$, with label ℓ_0^a. We extend *defaultVal* such that $\mathit{defaultVal}(\texttt{Array}(T)) = \ell_0^a$ for any $T \in \Theta_{\mathcal{P}}^a$.

To work with arrays, we create the commands **rdA** and **wrA** with $x \in \mathit{Var}_{\mathcal{P}} \cup \mathit{Par}_{\mathcal{P}}$ to read from and write to a location of an array. We also add the new commands $\mathbf{arr}(T)$, with $T \in \Theta_{\mathcal{P}}^a$, and **asize** which takes a variable $x \in \mathit{Var}_{\mathcal{P}} \cup \mathit{Par}_{\mathcal{P}}$ and retrieves the size of the array saved under x.

For the interpretation function, we remove the interpretation function clauses $[\![x_1.\cdots.x_n]\!]$, $[\![T\ x := E]\!]$ and $[\![x_1, \cdots, x_n.x := E]\!]$ and we add the following clauses for variables, with expressions E and E', sequences of variables X, X_1 and X_2 and single variable x. Here, X is a variable expression as constructed by the syntax above. We use the notation $X[E]$ to denote that the last $\langle \mathit{Var} \rangle$ element of X is an array access element "$[E]$" and we use the notation $X.x$ to denote that the last $\langle \mathit{Var} \rangle$ element of X is a variable access element "$.x$".

$$
\begin{aligned}
[\![x]\!] &= \mathbf{push}(\mathbf{this}); \mathbf{rd}(x) \\
[\![X[E]]\!] &= [\![X]\!]; [\![E]\!]; \mathbf{rdA} \\
[\![X.x]\!] &= [\![X]\!]; \mathbf{rd}(x) \\
[\![x := E']\!] &= [\![E']\!]; \mathbf{push}(\mathbf{this}); \mathbf{wr}(x) \\
[\![X[E] := E']\!] &= [\![E']\!]; [\![X]\!]; [\![E]\!]; \mathbf{wrA} \\
[\![X.x := E']\!] &= [\![E']\!]; [\![X]\!]; \mathbf{wr}(x) \\
[\![T\ X := E]\!] &= [\![X := E]\!]
\end{aligned}
$$

We also add the following clauses to deal with array creation and requesting the size of an array:

$$
\begin{aligned}
[\![\text{`}\texttt{array}\text{'}(E)]\!] &= [\![E]\!]; \mathbf{arr}(T) \\
[\![X.\mathbf{s}]\!] &= [\![X]\!]; \mathbf{asize}
\end{aligned}
$$

We then add new operational semantics rules to reflect reading from an array (**ComRdA**), writing to an array (**ComWrA**), getting the size of the array (**ComArrs**) and creating an array (**ComArr**). We also adapt the other operational semantics to include $\mathcal{M}$ into the state. The rule **ComRdA** given below is based on the rule **ComRd**, with changes to access the memory instead of variable environments and to use the command **RdA**. Note that the location is realized as a displacement from the memory address of the array. Changes from **ComRd** are highlighted.

$$
(\mathbf{ComRdA})\ \frac{\sigma(\ell) = \langle \theta, \boxed{\mathbf{rdA}}; \gamma, \chi; \ell'; v \boxed{,} \xi \rangle \qquad \boxed{\sigma(\ell') = \langle \alpha, s \rangle} \qquad v < s \qquad \mathcal{M}(\alpha + v) \text{ is defined}}{\langle Sc, \sigma, \boxed{\mathcal{M},} s\chi \rangle \Rightarrow_{\mathcal{P}} \langle Sc, \sigma[\ell \mapsto \langle \theta, \gamma, \chi; \boxed{\mathcal{M}(\alpha + v),} \xi \rangle], \boxed{\mathcal{M},} s\chi \rangle}
$$

The rule **ComWrA** given below is based on **ComWr**. Like **ComWr**, it keeps track of whether it has to update the stability stack if the array is changed and the array is a parameter, though this means that in the end, three entries of the struct environment are

integral to the functioning of this command.

$$\textbf{(ComWrA)}\ \frac{\sigma(\ell) = \langle \theta, \textbf{wrA}; \gamma, \chi; v_1; \ell'; v_2, \xi\rangle \qquad \sigma(\ell') = \langle \alpha, s\rangle \qquad \ell' \notin \mathcal{L}^0 \qquad v_2 < s \qquad su = (\mathcal{M}(\alpha + v_2) = v_1)}{\langle Sc, \sigma, \mathcal{M}, s\chi\rangle \Rightarrow \langle Sc, \sigma[\ell \mapsto \langle \theta, \gamma, \chi, \xi\rangle], \mathcal{M}[(\alpha + v_2) \mapsto v_1], s\chi_1 \wedge su; \ldots; s\chi_{|s\chi|} \wedge su\rangle}$$

The rule **ComAsize** given below is also based on **ComRd**, and returns the size of an array. The differences with **ComRd** are highlighted.

$$\textbf{(ComAsize)}\ \frac{\sigma(\ell) = \langle \theta, \textbf{asize}; \gamma, \chi; \ell', \xi\rangle \qquad \sigma(\ell') = \langle \alpha, s\rangle}{\langle Sc, \sigma, \mathcal{M}, s\chi\rangle \Rightarrow \langle Sc, \sigma[\ell \mapsto \langle \theta, \gamma, \chi; s, \xi\rangle], \mathcal{M}, s\chi\rangle}$$

The last new rule is **ComArr**, which creates a new array. This rule is based on **ComCons**, and differences with that rule are highlighted. Note that ℓ' is a fresh label in this rule, signified by $\sigma(\ell') = \bot$.

$$\textbf{(ComArr)}\ \frac{\sigma(\ell) = \langle \theta, \textbf{arr}(T); \gamma, \chi; s, \xi\rangle \qquad s \in \mathbb{N} \qquad s \geq 1 \qquad \mathcal{M} \text{ is not defined for } \alpha, \ldots, (\alpha + s) \qquad \sigma(\ell') = \bot}{\langle Sc, \sigma, \mathcal{M}, s\chi\rangle \Rightarrow \langle Sc, \sigma[\{\ell \mapsto \langle \theta, \gamma, \chi; \ell', \xi\rangle, \ell' \mapsto \langle \alpha, s\rangle\}], \mathcal{M}[\{\alpha_i \mapsto \mathit{DefaultVal}(T) \mid \alpha_i \in \alpha, \ldots, (\alpha + s)\}], s\chi\rangle}$$

With the rule **ComArr**, we can now make array instances, allocating them a sequence of unallocated addresses. We can also get the size of the array through the addition `.s` behind an array variable `x`, which is then handled by the transition **ComAsize**, induced by the **asize** command. We can lastly read and write from those arrays using conventional block-parenthesis notation, and the locations we read and write to can be determined dynamically. This satisfies our expectations for an array. Note that this extension does not include the destruction and deallocation of arrays.

With this extension, we are able to reimplement reachability to use arrays, as shown in Listing 9 (based on Listing 1). As you can see in the Listing, arrays make the system smaller, but as loops are contained in the schedule, one cannot employ a for-loop to walk through all array instances in a single step, and the code must be adapted for this.

6.3. **Discussion.** In this section, we presented three extensions for AuDaLa to allow for more streamlined and conventional programming in AuDaLa. These extensions are reasonably simple and therefore follow AuDaLa's goals of being a small programming language. Of these extensions, we think that the extension for parameter-specific fixpoints has the most merit, as it simplifies AuDaLa code and allows for more concise programs, which plays into AuDaLa's goals. We also estimate that it should be relatively easy to implement this extension and not have too much overhead in the execution time of AuDaLa programs, as part of the stability function can probably be derived at compile time.

The iterator increases the amount of parallelism in the schedule and therefore follows the goals of AuDaLa to have as much parallelism as possible, but for this extension in particular

```
struct Node (reach: Bool, succ: Array(Node), done: Nat) {
  reachability {
    if (reach = true) then {
      if (done < succ.s){
        succ[done].reach := true;
        done := done + 1;
      }
    }
  }

  init {
    Node node1 := Node(true, array(2), 0);
    Node node2 := Node(false, array(1), 0);
    Node node3 := Node(false, array(1), 0);
    Node node4 := Node(false, null, 0);

    node1.succ[0] := node2;
    node1.succ[1] := node3;
    node2.succ[0] := node3;
    node3.succ[0] := node4;
  }
}
init < Fix(reachability)
```

Listing 9: AuDaLa code for a reachability program on a small graph using arrays

it would be important to test the extension to find out the impact it has on the speed of execution of AuDaLa iterators compared to fixpoints before adding it. If it does not have too much of a negative impact on execution speed, this would be a good extension for AuDaLa.

The array extension is directly and theoretically useful to convert older parallel algorithms to AuDaLa and because many programmers will already be used to them. However, the impact on the efficiency of AuDaLa will have to be tested. Additionally, using arrays may lead to less parallel code: traversing an array is a sequential process, which can lead to a delayed application of new information to elements in the array. For example, where in Listing 1 the edges propagated reachability information obtained the next step without extra delay, the first Node in Listing 9 will only try to set Node 3 to reachable in the second step. As separating steps from loops is a fundamental design choice of AuDaLa, we furthermore do not think it is a good idea to solve this problem by introducing limited loops in the steps. However, even if it were solved that way, such a simple array traversal loop would make step execution less uniform and more sequential, as the struct instance with the longest array walked through now has to finish before the next step can be executed, and it could be that this array is significantly larger than the other arrays. Lastly, our current array extension considers all elements of any array to be parameters, which is a problem for fixpoints if a step uses a local array. To solve this, one will most likely have to start counting per array which parameters it is the value of, leading to more complicated extensions and read and write procedures. We therefore estimate that to make an array-like structure work in AuDaLa, more extensive extensions outside of the scope of this paper would be required,

and would require a consideration of how arrays should be defined beyond the standard to work best in AuDaLa.

We have kept ourselves to extensions that increase the practical expressivity in some way. Two important extensions that this paper did not cover are an extension for AuDaLa to do garbage collection and also remove struct instances, and an extension to include inheritance in AuDaLa and to be able to import and create packages and templates of structs in some way. As both of these extensions are significant and will most likely impact the basis of AuDaLa (for example, garbage collection and destroying struct instances will most likely impact the corollaries used in the Turing completeness proof), they will require a careful approach to implement them and we therefore deemed them outside of the scope of this paper. A similarly significant extension that does impact expressivity is to have an asynchronous separator which does separate steps but does not require synchronisation in between, which will at the very least impact the structured approach of AuDaLa and requires careful design outside of the scope of this paper. These three extensions are thus left for future work.

For this paper, we have not implemented and tested these extensions. We consider this the limitation of the currently expressed extensions, as the practical usability of AuDaLa is important, despite it not being our foremost concern. Additionally, to formalize the exact effects of the extensions on AuDaLa executions, we could develop a formal proof system for AuDaLa, extend it for each of the extensions and prove it sound.

7. Conclusion

In this paper, we have proven AuDaLa Turing complete by giving a method to implement any Turing machine in AuDaLa. Additionally, we have explored the practical expressivity of AuDaLa by seeking out the limits of AuDaLa's syntax and presented extensions to AuDaLa to overcome these limits. With this paper and the papers by Leemrijse *et al.* [LFN25, Lee23], we have made a case that AuDaLa is practically feasible, general-purpose and reasonably adaptable to specific purposes. This sets the basis for AuDaLa to be generally applicable.

However, more work can be done to streamline AuDaLa's use enough to make it a more attractive option for applications. First of all, we should make some theoretical applications that capitalize on AuDaLa's theoretical foundation, like a proof system for AuDaLa. On the practical side, there are at least two larger extensions that AuDaLa could benefit from, garbage collection and package management, which could be incorporated into AuDaLa. Additionally, we can continue our search for useful AuDaLa extensions and optimize the practical expressivity of AuDaLa more.

Acknowledgment

The authors would like to thank Gijs Leemrijse for his work on implementing algorithms in AuDaLa.

References

[BY95] T. Baba and T. Yoshinaga. A-NETL: a language for massively parallel object-oriented computing. In *PMMPC Proc.*, pages 98–105. IEEE, 1995. doi:10.1109/PMMPC.1995.504346.

[CDK14] Nathan Chong, Alastair F. Donaldson, and Jeroen Ketema. A Sound and Complete Abstraction for Reasoning about Parallel Prefix Sums. *SIGPLAN Not.*, 49(1):397–409, 2014. doi:10.1145/2578855.2535882.

[Cop24] B. Jack Copeland. The Church-Turing Thesis. In Edward N. Zalta and Uri Nodelman, editors, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Winter 2024 edition, 2024.

[dB+12] Frank S. de Boer et al. Decidability Problems for Actor Systems. In *CONCUR 2012 – Concurrency Theory*, volume 10 of *Logical Methods in Computer Science*, pages 562–577. Springer, 2012. doi:10.1007/978-3-642-32940-1_39.

[DD02] Jérémie Detrey and Oliver Diessel. *A Constructive Proof of the Turing Completeness of Circal*. School of Computer Science and Engineering, University of New South Wales, Australia, 2002.

[DG09] Cinzia Di Giusto. *Expressiveness of Concurrent Languages*. PhD thesis, Alma Mater Studiorum Università di Bologna, 2009.

[DK98] Arie Van Deursen and Paul Klint. Little languages: little maintenance? *Journal of Software Maintenance: Research and Practice*, 10:75–92, 1998. doi:10.1002/(SICI)1096-908X(199803/04)10:2<75::AID-SMR168>3.0.CO;2-5.

[FN24] Tom T. P. Franken and Thomas Neele. AuDaLa is Turing Complete. In *FORTE 2024 Proc.*, volume 14678 of *LNCS*, pages 221–229. Springer Nature Switzerland, 2024. doi:10.1007/978-3-031-62645-6_12.

[FNG23] Tom T. P. Franken, Thomas Neele, and Jan Friso Groote. An Autonomous Data Language. In *Theoretical Aspects of Computing – ICTAC 2023*, volume 14446 of *LNCS*, pages 158–177. Springer International Publishing, 2023.

[FNG25] Tom T. P. Franken, Thomas Neele, and Jan Friso Groote. The Autonomous Data Language – Concepts, design and formal verification. *Theoretical Computer Science*, 1057:115560, 2025. URL: https://www.sciencedirect.com/science/article/pii/S0304397525004980, doi:10.1016/j.tcs.2025.115560.

[G+08] Michael Garland et al. Parallel Computing Experiences with CUDA. *IEEE Micro*, 28(4):13–27, 2008. doi:10.1109/MM.2008.57.

[Gib15] Jeremy Gibbons. Functional Programming for Domain-Specific Languages. In *CEFP 2013*, LNCS, pages 1–28. Springer International Publishing, 2015. doi:10.1007/978-3-319-15940-9_1.

[HMU01] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation*. Addison-Wesley, Boston, 2nd edition, 2001.

[HND+21] Alec Henderson, Radu Nicolescu, Michael J. Dinneen, T. N. Chan, Hendrik Happe, and Thomas Hinze. Turing completeness of water computing. *J Membr Comput*, 3:182–193, 2021. doi:10.1007/s41965-021-00081-3.

[Koz76] Dexter Kozen. On parallelism in Turing machines. In *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*, pages 89–97. IEEE, 1976. doi:10.1109/SFCS.1976.20.

[Lee23] Gijs Leemrijse. Towards relaxed memory semantics for the Autonomous Data Language, 2023. MSc. thesis, Eindhoven University of Technology.

[LFN25] Gijs P. Leemrijse, Tom T. P. Franken, and Thomas Neele. Formalisation of a New Weak Semantics for AuDaLa. In *Automated Technology for Verification and Analysis*, pages 93–116. Springer Nature Switzerland, 2025. doi:10.1007/978-3-031-78750-8_5.

[QYZG17] Peng Qu, Jin Yan, You-Hui Zhang, and Guang R. Gao. Parallel Turing Machine, a Proposal. *J. Comput. Sci. Technol.*, 32:269–285, 2017. doi:10.1007/s11390-017-1721-3.

[RK+17] Jonathan Ragan-Kelley et al. Halide: decoupling algorithms from schedules for high-performance image processing. *Commun. ACM*, 61:106–115, 2017. doi:10.1145/3150211.

[RL93] F. Raimbault and D. Lavenier. RELACS for systolic programming. In *ASAP Proc.*, pages 132–135. IEEE, 1993. doi:10.1109/ASAP.1993.397128.

[UA10] David Ungar and Sam S. Adams. Harnessing emergence for manycore programming: early experience integrating ensembles, adverbs, and object-based inheritance. In *OOPSLA Proc.*, pages 19–26. ACM, 2010. doi:10.1145/1869542.1869546.

[Wie84] Juraj Wiedermann. *Parallel Turing machines*. Department of Computer Science, University of Utrecht The Netherlands, 1984.

[Y+17] Tatsuya Yamashita et al. Turing-Completeness of Asynchronous Non-camouflage Cellular Automata. In *Cellular Automata and Discrete Complex Systems*, LNCS, pages 187–199. Springer International Publishing, 2017. doi:10.1007/978-3-319-58631-1_15.