AuDaLa is Turing Complete

Tom T.P. Franken[0000−0002−1168−5450] and Thomas Neele[0000−0001−6117−9129]

Eindhoven University of Technology, Eindhoven, The Netherlands
{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 expressiveness of the language and to create verification methods for it. In this paper, we take our first steps to such a verification method by implementing Turing machines in AuDaLa and proving that implementation correct. This also proves that AuDaLa is Turing complete.

Keywords: AuDaLa · Verification · Turing Complete

1 Introduction

Nowadays, performance gains are increasingly obtained through parallelism. The focus is often on how to get the hardware to process the program efficiently and languages are often designed around that, focusing on threads and processes. Recently, AuDaLa [9] 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 [9], 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. A compiler for AuDaLa [15] enables execution of AuDaLa on GPUs.

AuDaLa is built to be simple and focusses fully on parallel data elements. This design principle relates AuDaLa to domain specific languages, which are often less expressive than general purpose languages. It is therefore relevant to establish the expressiveness of AuDaLa, as AuDaLa is built as a general purpose language. Additionally, establishing the expressiveness of AuDaLa also indicates how expressive the data-autonomous paradigm is. AuDaLa has a fully defined semantics, unlike many other languages, which we can use to answer this question.

Turing completeness is a well known property in computer science, which applies to a language or system that can simulate Turing machines. As a Turing machine can compute all effectively computable functions following the Church-Turing thesis [5], a Turing complete language or system can do the same. Two

approaches to showing Turing completeness are implementing a Turing machine in the target language [4, 16] and implementing μ-recursive functions [6, 12].

To prove AuDaLa's expressiveness, we prove AuDaLa Turing complete. We do this by implementing a Turing machine in AuDaLa (Section 2.2). We then give the intuition of the proof that this implementation is correct (Section 3). Constructing this implementation to exhibit correct behaviour is intricate due to AuDaLa's view on the behaviour of data elements and proofs (specifically those in Appendix A) involve detailed reasoning about the semantics and the inference rules defined in it and lay the foundation for proving AuDaLa programs correct.

Related Work. AuDaLa is a *data-autonomous* language and related to other data-focussed languages, like standard data-parallel languages (CUDA [10] and OpenCL [3]), languages which apply local parallel operations on data structures (Halide [18], ReLaCS [19]) and actor-based languages (Ly [20], A-NETL [1]).

Though the expressivity of actor languages has been studied before [2] and there is research into suitable Turing machine-like models for concurrency [14, 17, 21], there does not seem to be a large focus on proving Turing completeness of parallel languages. We estimate that this is because 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. Furthermore, parallel domain specific languages such as Halide [18] are simple by design, only focussing on their domain. Languages may also not be Turing complete on purpose [8, 11], for example to make automated verification decidable.

The proof for the Turing completeness of Circal [7] follows the same line of our proof. Other parallel systems that have been proven Turing complete include water systems [12] and asynchronous non-camouflage cellular automata [22].

2 The Turing Machine Implementation

2.1 Basic Concepts

We define a Turing machine following the definition of Hopcroft *et al.* [13]. 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 $\delta : (Q \setminus F) \times \Gamma \nrightarrow Q \times \Gamma \times \mathbb{D}$.

Every Turing machine T operates on an infinite *tape* divided into *cells*. Initially, this tape contains an input string $S = s_0 \dots s_n$ 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 $S = s_0 \dots s_n$, the *initial configuration* of a Turing machine T is (q_0, t_S), with q_0 as defined for T, and $t_S(i) = s_i$ for $0 \leq i \leq n$ and $t_S(i) = B$ otherwise.

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

Definition 1 (Turing machine transition). *Let $T = (Q, q_0, F, \Gamma, \Sigma, B, \delta)$ be a Turing machine and let (q, t) be a configuration such that $\delta(q, t(0)) = (q', s', D)$, with $D \in \mathbb{D}$. Then $(q, t) \to (q', t')$, where t' is defined as*

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

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

2.2 The Implementation of a Turing Machine in AuDaLa

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

An AuDaLa program contains three parts: the definitions of the data types and their parameters are expressed as *structs*, functions to be executed in parallel are given to these data types as *steps*, and these steps are ordered into the execution of a method by a *schedule* separate from the data system. *Steps* cannot include loops, which are instead managed by the schedule.

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 1.1.

```
1 struct TapeCell (left: TapeCell, right: TapeCell, symbol: Int){} //def. of TapeCell
2 struct Control (head: TapeCell, state: Int, accepting: Bool) {
3   transition {see Listing 1.2 and 1.3}      //definition of the step "transition"
4   init {see Listing 1.4}                    //definition of the step "init"
5 }
6 init < Fix(transition)       //schedule: run "init" once and then iterate "transition"
```

Listing 1.1: The AuDaLa program structure

The step *transition* in the *Control* struct models the transition function δ. For every pair $(q, s) \in Q \times \Gamma$ s.t. $\delta(q, s) = (q', s', D)$ with $D \in \mathbb{D}$, *transition* contains a clause as shown in Listing 1.2 (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 element, which we check in line 5. For this, as s can be *null*, we need to explicitly check whether *head* is a *null*-element. Note that $B = 0$, and that if $D = L$ the code only minimally changes.

```
if (state == q && head.symbol == s) then {
  head.symbol := s'; //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 1.2: A clause for $\delta(q, s) = (q', s', R)$.

```
transition {
  if (state == q1 && head.symbol == s1) then{ /*clause 1*/ }
  else if (state == q2 && head.symbol == s2) then { /*clause 2*/ }
  else if (state == q3 && head.symbol == s3) then { /*clause 3*/ }
  // etc.
}
```

Listing 1.3: The *transition* step. The shown pairs all have an output in δ.

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 1.3. In the step *init* in the *Control* struct, we create a *TapeCell* for every symbol $s \in S$ from left to right, which are linked together to create the tape. We also create a *Control*-instance. Listing 1.4 shows this for an example tape $S = s_0, s_1, s_2$.

In the semantics of AuDaLa [9], the initial state of any program contains only the special *null*-element of each struct. All parameters of the *null*-element are fixed to a *null*-value. They can create other elements but cannot write to their own parameters. Therefore, the call of *init* in the schedule causes the *null*-element of *Control* to initialize the tape. It also initializes a single non-*null* element of *Control*. The schedule will then have that element of *Control* run the *transition* step until the program stabilizes. Listing 1.1 shows the final structure of the program.

3 Turing Completeness

In this section, we show why AuDaLa is Turing Complete. We establish an equivalence between the configurations of a Turing machine and the configurations that can be extracted from the semantics of the corresponding AuDaLa program. We use the fact that the steps executed by the implementation are deterministic,

```
1  init {
2    TapeCell cell0 := TapeCell(null, null, s_0); // initialize the tape
3    TapeCell cell1 := TapeCell(null, null, s_1);
4    TapeCell cell2 := TapeCell(null, null, s_2);
5    cell1.left := cell0; // connect the tape
6    cell0.right := cell1;
7    cell2.left := cell1;
8    cell1.right := cell2;
9    Control(cell0, 0, (q_0 ∈ F)); //initialize the control
10 }
```

Listing 1.4: Initializing input string S.

as there is at most one non-*null* *Control* structure that executes the steps. We omit the full proof of Lemmas 3, 5 and 6, which can be found in Appendix A[1].

Henceforth, let P_{TS} be the implementation of a Turing machine T and an input string $S = s_0 \dots s_n$ as specified in Section 2.2. In AuDaLa's semantics, a *struct instance* is a data element instantiated from a struct during runtime. For the proof we consider a specific kind of AuDaLa state, the *idle state*, which has the property that none of its the struct instances are in the process of executing a step. In AuDaLa, the next step to be executed from an idle state is determined by the schedule. With this we define *implementation configurations*:

Definition 2 (Implementation Configuration). *Let p be an idle state of P_{TS} 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:*

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

where the dot notation $x.p$ indicates the value of parameter p in x and, for $i \geq 1$, $x.p^i$ is inductively defined as $x.p.p^{i-1}$ (with $x.p^0 = x$).

Note that an implementation configuration is also a Turing machine configuration. Next we define determinism for AuDaLa, as well as *data races*.

Definition 3 (Determinism). *Let s be a step in an AuDaLa program. Then s is deterministic iff for all states that can execute s, there exists exactly one state that is reached by executing s.*

Definition 4 (Data Race). *Let s be a step of P_{TS}. Let p be an idle state. Then s contains a data race starting in p iff p can execute s (according to its schedule) and during this execution, there exist a parameter x which is accessed*

[1] If the paper is accepted, a version including appendices will be put on ArXiv

by two distinct struct instances a and b, with one of these accesses writing to x. We call a data race between writes a write-write *data race, and a data race between a read a* read-write *data race.*

We use this to prove the following lemma:

Lemma 1 (AuDaLa Determinism). *An AuDaLa step s is deterministic if it cannot be executed by an idle state p in the execution of P_{TS} s.t. s contains a data race starting in p.*

Proof. If s contains no data races but is not deterministic, then some parameter x can have multiple possible values after executing s from some idle state p. As the operational semantics of AuDaLa do not allow interleaving by a single struct instance (as defined in the semantics of AuDaLa [9]), x must have been accessed by multiple struct instances during execution. The semantics also do not allow randomness, which means that all non-determinism in AuDaLa results from data races. These struct instances must then be in a data race. This is a contradiction. □

In practice, when a step is deterministic we can ignore interleaving of struct instances during the execution of the step.

Lemma 2. *The execution of init in P_{TS} is deterministic.*

Proof. To prove this we need to prove that the execution of *init* contains no data races (Lemma 1). 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 data race between two struct instances. □

Lemma 3 (Executing *init* in the initial state). *Let p_0 be the idle state at the start of executing P_{TS} and let the input string $S = s_0 \dots s_n$. Executing the step init on p_0 results in a state p_1 with a single non-null Control instance such that (q_0, t_S) is the implementation configuration of p_1.*

Proof. The proof consists of sequentially walking through the statements of *init* when executed from the initial state (which is idle) of P_{TS} as defined in the semantics of AuDaLa, processing the statements using those semantics. □

Lemma 4. *Let p be an idle state reachable in P_{TS} with a single non-null Control instance. Any execution of transition executed from p is deterministic.*

Proof. As per Lemma 1, we prove that the execution contains no data races. Let c be an arbitrary clause in the *transition* step (Listing 1.2). If *transition* has a data race during the execution of c, this data race must occur between the one non-*null* instance and the *null*-instance of *Control*. Let the non-*null* instance be x_0 and let x_1 be the *null*-instance of *Control*. Then the parameter which is accessed must be shared by both. This can only be *head.symbol*, as x_0 will not get through the if-statement and the other parameters are relative to x_0 and x_1.

However, as $x_0.head = null$, this means $head.symbol$ cannot be written to, as parameters of *null*-instances cannot be written to in AuDaLa. This contradicts that it can be in a data race. □

Lemma 5. *Every transition step executed in P_{TS} is deterministic.*

Proof. By induction. As a base case, the first execution of *transition* happens from p_1 as defined in Lemma 3, which has only one non-*null Control* instance.

Then consider the execution of *transition* from an idle state p' with one non-*null Control* instance, resulting in idle state p. Due to Lemma 4, we know that the execution of *transition* is deterministic, so we can consider the sequential execution of *transition*. As *transition* is made up of multiple mutually exclusive clauses, considering only a single clause suffices. As in none of the statements a *Control* instance is created, as seen in Listing 1.2, it follows that p will also have only a single non-*null Control* instance. □

Lemma 6 (Effect of a *transition* execution). *Let p be an idle state of P_{TS} from which transition can be executed and let (q, t) be the implementation configuration of p. Assume that (q, t) is also a configuration of T. Then the result of a transition in T is a configuration (q', t') iff the result of executing the transition step from p in P_{TS} is an idle state p' such that (q', t') is its implementation configuration.*

Proof. We know from Lemma 5 that p has one non-*null Control* instance. The proof consists of walking through the statements of *transition* starting at p. □

By induction, using Lemma 3 as base case and Lemma 6 as step, any idle state of P_{TS} after executing *init* corresponds directly to a state (q, t) of T, including terminating and accepting states. We conclude:

Theorem 1. *AuDaLa is Turing complete.*

4 Conclusion

In this paper, we have proven AuDaLa Turing complete by implementing a sequential Turing machine. In future work, we hope to extend the principles here to a full system to prove AuDaLa programs correct. We may also extend the proofs to the weak memory model variant of the AuDaLa semantics [15].

References

1. Baba, T., Yoshinaga, T.: A-NETL: a language for massively parallel object-oriented computing. In: PMMPC Proc. pp. 98–105. IEEE (1995). https://doi.org/10.1109/PMMPC.1995.504346

2. de Boer, F.S., et al.: Decidability Problems for Actor Systems. In: CONCUR 2012 – Concurrency Theory. pp. 562–577. Springer (2012). https://doi.org/10.1007/978-3-642-32940-1_39
3. Chong, N., Donaldson, A.F., Ketema, J.: A sound and complete abstraction for reasoning about parallel prefix sums. SIGPLAN Not. **49**(1), 397–409 (2014). https://doi.org/10.1145/2578855.2535882
4. Churchill, A., Biderman, S., Herrick, A.: Magic: The Gathering Is Turing Complete. In: 10th International Conference on Fun with Algorithms (FUN 2021). Leibniz International Proceedings in Informatics (LIPIcs), vol. 157, pp. 9:1–9:19. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2020). https://doi.org/10.4230/LIPIcs.FUN.2021.9
5. Copeland, B.J.: The Church-Turing Thesis (1997), https://plato.stanford.edu/ENTRIES/church-turing/, last Modified: 2017-11-10
6. Date, P., Potok, T., Schuman, C., Kay, B.: Neuromorphic Computing is Turing-Complete. In: Proceedings of the International Conference on Neuromorphic Systems 2022. pp. 1–10. ICONS '22, Association for Computing Machinery (2022). https://doi.org/10.1145/3546790.3546806
7. Detrey, J., Diessel, O.: A Constructive Proof of the Turing Completeness of Circal. School of Computer Science and Engineering, University of New South Wales, Australia (2002)
8. Deursen, A.V., Klint, P.: Little languages: little maintenance? Journal of Software Maintenance: Research and Practice **10**, 75–92 (1998). https://doi.org/10.1002/(SICI)1096-908X(199803/04)10:2<75::AID-SMR168>3.0.CO;2-5
9. Franken, T.T.P., Neele, T., Groote, J.F.: An Autonomous Data Language. In: Theoretical Aspects of Computing – ICTAC 2023. LNCS, vol. 14446, pp. 158–177. Springer International Publishing (2023)
10. Garland, M., et al.: Parallel Computing Experiences with CUDA. IEEE Micro **28**(4), 13–27 (2008). https://doi.org/10.1109/MM.2008.57
11. Gibbons, J.: Functional Programming for Domain-Specific Languages. In: CEFP 2013, pp. 1–28. LNCS, Springer International Publishing (2015). https://doi.org/10.1007/978-3-319-15940-9_1
12. Henderson, A., Nicolescu, R., Dinneen, M.J., Chan, T.N., Happe, H., Hinze, T.: Turing completeness of water computing. J Membr Comput pp. 182–193 (2021). https://doi.org/10.1007/s41965-021-00081-3
13. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to automata theory, languages, and computation. Addison-Wesley, Boston, 2nd ed edn. (2001)
14. Kozen, D.: On parallelism in turing machines. In: 17th Annual Symposium on Foundations of Computer Science (sfcs 1976). pp. 89–97 (1976). https://doi.org/10.1109/SFCS.1976.20
15. Leemrijse, G.: Towards relaxed memory semantics for the Autonomous Data Language (2023), MSc. thesis, Eindhoven University of Technology
16. Pitt, L.: Turing Tumble is Turing-Complete. Theoretical Computer Science **948**, 113734 (2023). https://doi.org/10.1016/j.tcs.2023.113734
17. Qu, P., Yan, J., Zhang, Y.H., Gao, G.R.: Parallel Turing Machine, a Proposal. J. Comput. Sci. Technol. **32**, 269–285 (2017). https://doi.org/10.1007/s11390-017-1721-3
18. Ragan-Kelley, J., et al.: Halide: decoupling algorithms from schedules for high-performance image processing. Commun. ACM **61**, 106–115 (2017). https://doi.org/10.1145/3150211
19. Raimbault, F., Lavenier, D.: RELACS for systolic programming. In: ASAP Proc. pp. 132–135. IEEE (1993). https://doi.org/10.1109/ASAP.1993.397128

20. Ungar, D., Adams, S.S.: Harnessing emergence for manycore programming: early experience integrating ensembles, adverbs, and object-based inheritance. In: OOPSLA Proc. pp. 19–26. ACM (2010). https://doi.org/10.1145/1869542.1869546
21. Wiedermann, J.: Parallel turing machines. Department of Computer Science, University of Utrecht The Netherlands (1984)
22. Yamashita, T., et al.: Turing-Completeness of Asynchronous Non-camouflage Cellular Automata. In: Cellular Automata and Discrete Complex Systems. pp. 187–199. LNCS, Springer International Publishing (2017). https://doi.org/10.1007/978-3-319-58631-1_15

A Proofs

In this appendix, we first present some auxiliary lemmas in Section A.1, after which we present the in detail proofs to support Lemmas 3 and 6 in Section A.2.

The auxiliary lemmas in Section A.1 use the AuDaLa semantics to prove that update statements actually perform updates, constructor statements actually construct new data elements and so forth. We recommend reading it only to those familiar with the semantics of AuDaLa. Section A.2 does not use any concepts other than those introduced in the paper and references to lemmas from Section A.1 and should not prove a challenge for those who read this paper.

A.1 Auxiliary AuDaLa lemmas

The lemmas presented here are generally useful for any program to be proven correct. They make use of the AuDaLa semantics, so familiarity with these semantics [9] is assumed.

For the rest of this section, let $\mathcal{P}$ be an AuDaLa program without read-write data races. Let $P = \langle Sc, \sigma, s\chi \rangle$ be a state of $\mathcal{P}$, and let $\ell_p \in \mathcal{L}$ be a label and p a struct instance s.t. $p = \sigma(\ell_p) = \langle sL_p, [\![E]\!]; \gamma_p, \chi_p, \xi_p \rangle$. In the first lemma, we prove the result of resolving references in AuDaLa:

Lemma 7 (AuDaLa reference resolution). *Let E be a variable expression of the form "$A.a$" in AuDaLa, where A has the syntax "$a_1.\cdots.a_n$" with variable identifiers $a, a_1, a_2, a_3, \ldots, a_n$. Let the state $P' = \langle Sc, \sigma', s\chi' \rangle$ be a state resulting from the last transition to resolve A, s.t. $\sigma(\ell_p) = \langle sL_p, \gamma'_p; \chi'_p, \xi_p \rangle$. Then $\chi'_p = \chi_p; \ell$, where*

$$\ell = \begin{cases} \ell_p & \text{if } n = 0 \\ p.a_1.\cdots.a_n & \text{otherwise,} \end{cases}$$

Proof. First, E is either a variable that has to be read from, or a variable that has to be written to. In both cases, $[\![A]\!] = \mathbf{push}(\mathbf{this}); \mathbf{rd}(a_1); \ldots; \mathbf{rd}(a_n)$. We prove by induction on n that for any n, the value on the stack resulting from the resolution of A is a single label. As induction hypothesis we take that after the reads up till and including $\mathbf{rd}(a_j)$, with $\chi_{p,j}$ being the stack after those reads, $\chi_{p,j} = \chi_p; \ell_j$, where ℓ_j is the only possible label that can be read from the reads so far.

$n = 0$: If $n = 0$, $[\![A]\!] = \textbf{push}(\textbf{this})$, which is resolved using the derivation rule **ComPushThis** and results in $\chi'_p = \chi_p; \ell_p$. Therefore, the base case holds.

$n = i$: By the induction hypothesis, we know that after the $i-1$th read, $\chi_{p,i-1} = \chi_p; p.a_1.\cdots.a_{i-1}$. Then due to the form of E and by the well-typedness of the syntax of P, we know that $p.a_1.\cdots.a_{i-1}.a_i \in \mathcal{L}$. As there are no read-write data races, we know that there is only a single possible label $p.a_1.\cdots.a_{i-1}.a_i$. Then it follows by the transition **ComRd** that the read of $p.a_1.\cdots.a_{i-1}.a_i$ removes $p.a_1.\cdots.a_{i-1}$ from the stack and adds $p.a_1.\cdots.a_i$ to it. Therefore, $\chi_{p,i} = \chi_p; p.a_1.\cdots.a_i$ and the step holds.

As the induction holds, the lemma holds. □

We then prove the effect of executing an expression:

Lemma 8 (AuDaLa expression execution). *Let E be an AuDaLa expression. Let the state $P' = \langle Sc, \sigma', s\chi' \rangle$ be a state resulting from the transition of the last command of $[\![E]\!]$, with $p' = \sigma'(\ell_p) = \langle sL_p, \gamma'_p; \chi'_p, \xi'_p \rangle$. Then there exists a value v s.t. $\chi'_p = \chi_p; v$. Moreover:*

1. *If $E =$ "`this`", $v = \ell_p$.*
2. *If $E =$ "`null`", and T is the type as determined by the context of E, $v = defaultVal(T)$*
3. *If $E = lit$, where $lit \in LT$, with semantic value $val(lit)$, then $v = val(lit)$.*
4. *If $E = x_1.\cdots.x_n.x$, with $x_1, \ldots, x_n \in ID$, then $v = x_1.\cdots.x_n.x$.*
5. *If $E =$ "sL `(` $E_1, \ldots, E_n$ `)`", for a struct type sL with parameters $par_1, \ldots, par_n$, then $v \in \mathcal{L}$ s.t. $\sigma(v) = \bot$ and $\sigma'(v) = \langle sL, \varepsilon, \varepsilon, \xi \rangle$. Moreover, $\xi = \xi^0_{sL}[par_1 \mapsto [\![E_1]\!], \ldots, par_n \mapsto [\![E_n]\!]]$.*
6. *If $E =$ "`!` E'", then $v = \neg[\![E']\!]$.*
7. *If $E =$ "`(` E' `)`", then $v = [\![E']\!]$.*
8. *If $E =$ "E_1 `o` E_2", for syntactic operator o with semantic equivalent $\circ$, then $v = [\![E_1]\!] \circ [\![E_2]\!]$.*
9. *If E creates a new struct instance, $s\chi' = false^{|s\chi|}$.*

Proof. We prove the lemma by implicit structural induction, with our induction hypothesis being that the lemma holds for any subexpression encountered in cases 5-8, and with as base cases the cases 1-4. Due to our assumption that P has no read-write data races, we can assume that there is only one possible value read from any variable. We then prove the cases separately:

1. If $E =$ "`this`", then according to the interpretation function, this is interpreted as the command **push**(**this**), which is then resolved using the derivation rule **ComPushThis**, executed by p. The result of this derivation rule is that p pushes ℓ_p on it's stack, so this base case holds and $v = \ell_p$.
2. If $E =$ "`null`", and T is the type as determined by the context of E, then according to the interpretation function, this is interpreted as the command **push**(()$defaultVal(T)$). This is then resolved using the derivation rule **ComPush** executed by p, which pushes $defaultVal(T)$ on the stack. Therefore, this base case holds and $v = defaultVal(T)$.

3. If $E = lit$, where $lit \in LT$, with semantic value $val(lit)$, then according to the interpretation function, this is interpreted as the command $\mathbf{push}(val(lit))$. This is resolved using the derivation rule **ComPush** executed by p, which pushes $val(lit)$ to the stack. Therefore, this base case holds and $v = val(lit)$.
4. If $E = x_1.\cdots.x_n.x$, with $x_1, \ldots, x_n \in V$, then according to the interpretation function, this is interpreted as the commands $\mathbf{push}(\mathbf{this}); \mathbf{rd}(x_1); \ldots; \mathbf{rd}(x_n); \mathbf{rd}(x)$. Then, let P_1 be the state after resolving $\mathbf{push}(\mathbf{this}); \mathbf{rd}(x_1); \ldots; \mathbf{rd}(x_n)$ with struct environment σ_1 s.t. $\sigma_1(\ell_p)$ has stack χ_1. From Lemma 7 we know that $\chi_1 = \chi; \ell_p.x_1.\cdots.x_n$ and that $\ell_p.x_1.\cdots.x_n \in \mathcal{L}$. Then, by **ComRd**, we know that $v = \ell_p.x_1.\cdots.x_n.x$. Therefore, this base case holds.
5. If $E =$ "sL `(` $E_1, \ldots, E_n$ `)`", for a struct type sL with parameters $par_1, \ldots, par_n$, then from the interpretation function, we know that E is interpreted as $[\![E_1]\!]; \ldots; [\![E_n]\!]; \mathbf{cons}(sL)$. By the structural induction hypothesis, we know that $[\![E_1]\!]; \ldots; [\![E_n]\!]$ results in the sequence of values $v_1; \ldots; v_n$ at the end of the stack of p. Then by the derivation rule **ComCons**, we know that there exists a label ℓ s.t. $\sigma(\ell) = \bot$ and $\sigma'(\ell) = \langle sL, \varepsilon, \varepsilon, \xi \rangle$, where $\xi = \xi^0_{sL}[par_1 \mapsto v_1, \ldots, par_n \mapsto v_n]$. Also by **ComCons**, we know that $v = \ell$. Therefore, this case holds
6. If $E =$ "`!` E'", then by the interpretation function, this gets interpreted as $[\![E']\!]; \mathbf{not}$. Then by the structural induction hypothesis, we know that $[\![E']\!]$ results in a value v' at the end of the stack of s. Then by derivation rule **ComNot**, we know that $v = \neg v'$, so this case holds.
7. If $E =$ "`(` E' `)`", then as the concrete syntax gets converted into an abstract syntax tree, $[\![E]\!] = [\![E']\!]$, as $[\![E']\!]$ pushes a value v' to the stack as per the structural induction hypothesis, it follows that $[\![E]\!]$ also pushes v' to the stack, so $v = v'$. Therefore, this case holds.
8. If $E =$ "E_1 `o` E_2", for syntactic operator o with semantic equivalent $\circ$, this is interpreted by the interpretation function as $[\![E_1]\!]; [\![E_2]\!]; \mathbf{op}(o)$. Then by the structural induction hypothesis, we know that $[\![E_1]\!]; [\![E_2]\!]$ results in the values $v_1; v_2$ on the stack of p. Then by derivation rule **ComOp**, we know that $v = v_1 \circ v_2$. Therefore, this case holds.
9. If E creates a new struct instance, then either $E =$ "sL `(` $E_1, \ldots, E_n$ `)`" or a subexpression of E creates a new struct instance. In the second case, $s\chi' = false^{|s\chi|}$ by the structural induction hypothesis. In the first case, due to the execution of **ComCons** during the resolution of E, $s\chi' = false^{|s\chi|}$.

As the structural induction and all cases within it hold, the lemma holds. □

We can use this lemma to prove the effects of statement executions. Firstly, for constructor statements, note the following:

Corollary 1 (AuDaLa constructor execution). *A constructor statement has the same effects as a constructor expression, as defined in case 5 of Lemma 8, and also resets the stability stack as defined in case 9 of Lemma 8.*

We then prove the update statement execution effects:

Lemma 9 (AuDaLa update execution). *Let Z be an AuDaLa statement of the form "$A.a$:= E", where A has the syntax "$a_1.\cdots.a_x$" with $a, a_1, a_2, a_3, \ldots, a_n \in ID$ and E is an expression. Let label ℓ_α be uniquely defined as*

$$\ell_\alpha = \begin{cases} p.a_1.a_2.a_3.\cdots.a_n & \text{if } n > 0 \\ \ell_p & \text{if } n = 0 \end{cases}.$$

Let the state $P' = \langle Sc, \sigma', s\chi' \rangle$ be the state resulting from the transition of the last command of $[\![Z]\!]$. Let v be the value pushed to the stack as a result of resolving $[\![E]\!]$ (as per Lemma 8). Let $\sigma(\ell_\alpha) = \langle sL_\alpha, \gamma_\alpha, \chi_\alpha, \xi_\alpha \rangle$ and let $\sigma'(\ell_\alpha) = \langle sL_\alpha, \gamma'_\alpha, \chi'_\alpha, \xi'_\alpha \rangle$.

Then, if $\ell_\alpha \neq \mathcal{L}_0$, $\xi'_\alpha(a) = v$ and if $\xi_\alpha(a) \neq v \wedge a \in Par_{sL_\alpha}$, $s\chi' = false^{|s\chi|}$.

Proof. We know that $[\![Z]\!] = [\![E]\!]; [\![a_1; \cdots; a_x]\!]; \mathbf{wr}(a)$ by the definition of the interpretation function. From Lemma 8 we know that through $[\![E]\!]$, v is put on the stack first. Then, by Lemma 7, we know that the result of $[\![a_1; \cdots; a_x]\!]$ is that ℓ_α is pushed on the stack. Then if $\ell_\alpha \neq \mathcal{L}_0$, we know through the derivation rule **ComWr** that $\xi_\alpha(a) = v$. If $\xi_\alpha(a) \neq v \wedge a \in Par_{sL_\alpha}$, we know that the value of a before **ComWr** can either still be $\xi_\alpha(a)$ or it can have been written to by another struct instance p', also using a **ComWr** transition. In the first case, it follows from **ComWr** that $s\chi' = false^{|s\chi|}$. In the second case, if the other struct instance writes v to a, $s\chi' = false^{|s\chi|}$ due to the **ComWr** transition done by p', and if not, then $s\chi' = false^{|s\chi|}$ due to the **ComWr** transition of p. In any case, $s\chi' = false^{|s\chi|}$. □

The above lemma also suffices for assignment statements:

Corollary 2 (Assignment statements). *Lemma 9 also holds for statements of the form "T a := E", with $T \in \mathcal{T}$ and "a := E".*

Proof. As $[\![$"T a := E"$]\!] = [\![$"a := E"$]\!]$, the effects of executing "T a := E" are the same as executing "a := E" (with $n = 0$). The stability stack will not be updated, as a cannot be a parameter according to our static syntax requirements. □

Lastly, we prove the effects of executing an if-then statement:

Lemma 10 (If-then Statements). *Let Z be a statement of the form "*`if` E `then { }` S `}`*", where S is a list of statements and E is an expression.*

Let the state $P' = \langle Sc, \sigma', s\chi' \rangle$ be the state resulting from the transition of the last command of $[\![Z]\!]$, and let $\sigma'(\ell_p) = sL_p, \gamma'_p, \chi'_p, \xi'_p \rangle$. Let v be the value pushed to the stack as a result of resolving $[\![E]\!]$ (as per Lemma 8).

Then either $v = true$ and $\gamma'_p = [\![S]\!]; \gamma_p$ or $v = false$ and $\gamma'_p = \gamma_p$.

Proof. We know that $[\![Z]\!] = [\![E]\!]; \mathbf{if}([\![S]\!])$ by the definition of the interpretation function. By our assumption of well-typedness, we know that the value v to which $[\![E]\!]$ resolves is a boolean value, and therefore the value at the end of the stack after resolving $[\![E]\!]$ will be either *true* or *false*. Then if $v = true$, we know

by the derivation rule **ComIfT** that $\gamma'_p = [\![S]\!]; \gamma_p$, and if $v = \mathit{false}$, we know by the derivation rule **ComIfF** that $\gamma'_p = \gamma_p$. □

We have now proven the effects of every type of statement. For if-statements and constructor statements, the result of the statement are permanent during the execution of a step Q. Assignment statements can only work with local variables, of which the values are irrelevant at the end of Q. We do however need to prove what we can guarantee about updated parameters after the execution of an update:

Lemma 11 (AuDaLa update results). *Let Q be a step in $\mathcal{P}$ and let Z be an update statement s.t. p executing Z updates a parameter $p'.x$ with type T of some struct instance p' to a value b during Q (along Lemma 9). Let a be the original value of $p'.x$. Let P be a state during Q after the execution of Z by p and before the execution of the statement after Z by p. Then all of the following holds:*

a. If p' is a null-instance, $p'.x = a = \mathit{defaultVal}(T)$.
b. If p' is not a null-instance:
  i. If $p'.x$ is not involved in a write-write data race, $p'.x = b$.
  ii. If $p'.x$ is involved in a write-write data race, let N be the set of all values written to $p'.x$ during T by all data elements involved in the write-write data race. Then $p'.x \in N$.

Additionally, we know that if $a \neq b$, the stability stack is reset.

Proof. If p' is a *null*-instance, then by the rule **ComWrSkip** and by the initialization of *null*-instances, we know that $p'.x = a = \mathit{null}$ after the execution of Z by p. If p' is not a *null*-instance, and $p'.x$ is not involved in a write-write data race, $p'.x$ is not involved in any data race, as $\mathcal{P}$ does not have read-write data races. Then as no other element other than p can have written to $p'.x$ during or after the execution of Z by p and b is deterministic during the execution of Z (as there are no read-write data races), $p'.x = b$. If p' is not a *null*-instance and $p'.x$ is in a write-write data race, as AuDaLa does not allow for nondeterminism in a single data element, this data race must be between different data elements. As these can execute their update statements in any order, any value in N can be the last value written to $p'.x$ before p executes the statement after Z, so $p'.x \in N$. □

This extends to step executions:

Corollary 3 (AuDaLa parameters after a step). *Let Q be a step in $\mathcal{P}$ and let P be a state resulting from an execution of Q. Let Z be the last update statement in Q of some parameter $p'.x$ with type T of some struct instance p' by p, which updates $p'.x$ to a value b. Let the value of $p'.x$ before the execution of Q be a. Then in P:*

a. If p' is a null-instance, $p'.x = a = \mathit{defaultVal}(T)$.
b. If p' is not a null-instance:

i. *If $p'.x$ is not involved in a write-write data race, $p'.x = b$.*
ii. *If $p'.x$ is involved in a write-write data race, let N be the set of all values written to $p'.x$ during Q by all data elements involved in the write-write data race. Then $p'.x \in N$.*

Additionally, we know that if $p'.x$ has had its value changed during the execution of Q, the stability stack has been reset during the execution of Q.

A.2 Turing Complete Lemmas Proofs

In this section, we will prove Lemma 3 and 6 in more detail, using the auxiliary lemmas of the previous section. Recall Lemma 3:

Lemma 3 (Executing *init* in the initial state). *Let p_0 be the idle state at the start of executing P_{TS} and let the input string $S = s_0 \dots s_n$. Executing the step init on p_0 results in a state p_1 with a single non-null Control instance such that (q_0, t_S) is the implementation configuration of p_1.*

Proof. First, note that the idle state at the start of executing P_{TS} is the initial state of P_{TS}. The initial state for P_{TS}, as defined in the AuDaLa semantics, contains the schedule of P_{TS}, the *null*-instances of all structs, and a stability stack. The stability stack has no bearing on this proof, and will be disregarded.

We need to show that p_1 has implementation configuration (q_0, t_S). To show that, we first prove that p_1 contains only one non-*null* struct instance of *Control* and that its *state* parameter is set to q_0. To prove this, we can assume the *init*-code is executed without nondeterministic behaviour, due to Lemma 2. Only the *null*-instance of *Control* executes *init* (as it is the only instance to exist in p_0). The step code makes only a single *Control*-instance, and as the code is deterministic and the *null*-instance executes it, we know that this means only one *Control*-instance is present in p_1, following Lemma 1, which we will call c. Also following Lemma 1, we know that the *state* parameter of c is set to q_0 (represented by integer 0).

We then prove that the function made according to Definition 2 in p_1 from the *TapeCells* is t_S. To do this, we first prove that in p_1, there exists a *TapeCell* for all symbols $s_i \in S$, and no others, s.t. every *TapeCell* s_i is be connected to s_{i-1} and s_{i+1} (if they exist) through parameters *left* and *right* respectively. Then we prove that the *head* parameter of c will be set to the *TapeCell* of s_0.

The first follows from the template in Listing 1.4, which we can follow sequentially due to Lemma 2. According to Lemma 2, the first part makes one *TapeCell* instance for every $s_i \in S$, and according to Corollary 11 and Lemma 2, these are then connected to the correct *left* and *right* neighbours. It follows from the listing and Lemma 1 that *head* parameter of c will be set to the *TapeCell* for s_0.

Then the lemma holds: the implementation configuration of p_1 is (q_0, t_S). □

Now recall Lemma 6:

Lemma 6 (Effect of a *transition* execution). *Let p be an idle state of P_{TS} from which transition can be executed and let (q, t) be the implementation configuration of p. Assume that (q, t) is also a configuration of T. Then the result of a transition in T is a configuration (q', t') iff the result of executing the transition step from p in P_{TS} 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 P_{TS}, there exists a single transition in p for P_{TS} iff $\delta(q, t)$ is defined. Additionally, δ is a function, so it is always uniquely defined for (q, t).

Let $\delta(q, t) = (q', s', D)$. W.l.o.g., let $D = R$ (the proof of $D = L$ is analogous). Then as a transition in T is deterministic, by Definition 1, the resulting state of taking a transition from (q, t) is the state (q', t'), with

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

Taking the transition in p for P_{TS} is also deterministic (Lemma 5), and therefore we can walk through the statements of the clause to determine its effect. By definition of P_{TS}, this clause is based on the template shown in Listing 1.2. Let c be the single *Control* instance of p, and let h be the *TapeCell* instance which is referenced in the *head* parameter of c. Due to Lemma 11, the result of the transition is that the *state* of c is updated to q', the *symbol* of h is updated to s', 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, by Lemma 10 and Lemma 11).

Creating a function of the *TapeCells* as in Definition 2 then results in function t'', s.t. $t''(-1) = s'$ and $t''(i) = t(i+1)$ for all $i \neq -1$, which is equal to t'. Then, by Definition 2, the implementation configuration of the resulting state is (q', t'). Therefore the lemma holds. □