The Autonomous Data Language – Concepts, Design and Formal Verification

Tom T.P. Franken, Thomas Neele, Jan Friso Groote

[a] *Eindhoven University of Technology The Netherlands*

Abstract

Nowadays, the main advances in computational power are due to parallelism. However, most parallel languages have been designed with a focus on processors and threads. This makes dealing with data and memory in programs hard, which distances the implementation from its original algorithm. We propose a new paradigm for parallel programming, the *data-autonomous* paradigm, where computation is performed by autonomous data elements. Programs in this paradigm are focused on making the data collaborate in a highly parallel fashion. We furthermore present AuDaLa, the first data autonomous programming language, and provide a full formalisation that includes a type system and operational semantics. Programming in AuDaLa is very natural, as illustrated by examples, albeit in a style very different from sequential and contemporary parallel programming. Additionally, it lends itself for the formal verification of parallel programs, which we demonstrate.

Keywords: Data-Autonomous, Programming Language, Operational Semantics, Formal Verification

1. Introduction

As increasing the speed of sequential processing becomes more difficult [1], exploiting parallelism has become one of the main means of obtaining further performance improvements in computing. Thus, languages and frameworks aimed at parallel programming play an increasingly important role in computation. Many existing parallel languages use a *task-parallel* or a *data-parallel* paradigm [2].

Task-parallelism mostly focuses on the computation carried out by individual threads, scheduling tasks to threads depending on which threads are idle. In data-parallelism, threads execute the same function but are distributed over the data, thus performing a parallel computation on the collection of all data.

In a shared memory setting, programs in both paradigms require careful design of memory layout, memory access and movement of data to facilitate the threads used by the program. Examples of this are the use of barriers and data access based on thread id's, as well as access protocols. Not only is extensive data movement costly and hinders some performance optimizations [3, 4], the memory handling necessary throughout the entire program due to the focus on threads only widens the gap between algorithms and implementation as noted by, for instance, Leiserson *et al.* [1]. Therefore, to promote memory locality and more algorithmic code, a new data-focused paradigm is needed.

In the conference version of this paper [5], we proposed the new *data-autonomous* paradigm, where *data elements* not only locally store data and references, but also execute their own computations. Computations are always carried out in parallel by all data elements; this is governed by a *schedule*. Data elements can cooperate through stored references. The paradigm completely abstracts away from processors and memory and is fully focused on data, compared to task- and data-parallelism (see Figure 1).

The paradigm provides several benefits. First, it results in a *separation of concerns*: code concerning data structures, algorithms and orchestration is properly separated. Furthermore, parallelism is encouraged by always running computations concurrently on groups of data elements. Finally, the paradigm promotes

Email addresses: t.t.p.franken@tue.nl (Tom T.P. Franken), t.s.neele@tue.nl (Thomas Neele), j.f.groote@tue.nl (Jan Friso Groote)

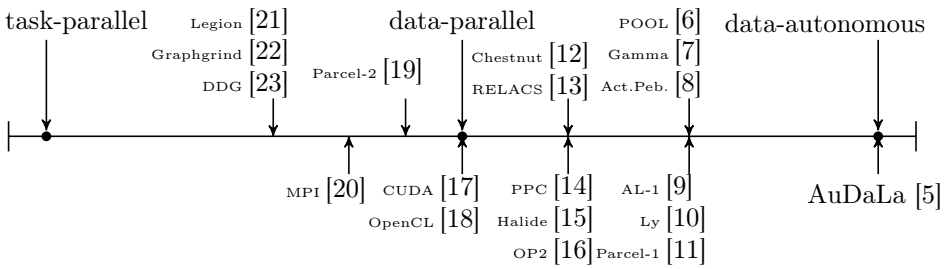

Figure 1: Approximate placement of related work on an axis from process-focused (left) to data-focused (right) paradigms.

a bottom-up design process, from data structure to computations to schedule. Through these benefits, the paradigm and languages based on the paradigm facilitate a more intuitive understanding of how the data interacts in a parallel program, and therefore what that program actually does. We foresee that the data-autonomous paradigm is particularly well suited for graph algorithms and other procedures that operate on unstructured data.

We also introduced AuDaLa [5], the *Autonomous Data Language*, the first data-autonomous programming language, which explores what programming in the data-autonomous paradigm would be like. As opposed common programming languages like C++/CUDA [17, 24] or theoretical frameworks like the PRAM [25] or PPM [26], the focus of AuDaLa is neither the performance of parallel programming, nor is (mainly) the expression of parallel algorithms. Instead, AuDaLa's focus is to be a simple yet implementable and usable parallel programming language which follows the data autonomous paradigm and promotes the creation of accessible and understandable parallel programs. AuDaLa facilitates the understanding of the effects of a parallel program by its enforced structure, and its decoupling of loops from parallelism. By keeping AuDaLa compact and formally defining AuDaLa's semantics, AuDaLa is small and independent of specific hardware for both execution and formal verification.

In AuDaLa, *structs*, *steps* and a *schedule* are responsible for data, computation and orchestration, respectively. We explained AuDaLa's design principles using an example, and gave further examples of AuDaLa programs implementing basic parallel methods. AuDaLa's behaviour is completely formalised in an operational semantics, enabled by its compact syntax. The semantics enabled us to show [27] that AuDaLa is Turing complete.

The conference version of the paper focused mostly on the requirements of keeping AuDaLa simple, accessible and understandable. While the operational semantics allowed for an implementation, we provide an alternative semantics suitable for execution on systems with a weak memory model in [28], thereby making AuDaLa more feasible for implementation on most modern processors. The two semantics are equivalent under the absence of read-write race conditions. A prototype compiler from AuDaLa to CUDA is presented in [29], along with other implemented algorithms.

This work extends [5] by providing a type system, additional example programs and correctness proofs. In particular, we make the following contributions:

- We provide a formal type system for AuDaLa, which determines when programs are *well-formed*.
- We extend the set of examples with three additional AuDaLa programs. The first is a program for sorting with a worst case time complexity of $O(\log m)$, where m is the length of the to be sorted list. This program is followed by two programs for the *3SUM* problem, one of which uses a nested fixpoint.
- We provide a theorem which specifies the exact effect of the execution of commands in well-formed programs, as well as some other properties (Theorem 7.14). This significantly simplifies the notation in later theorems and proofs.
- We prove that both our AuDaLa programs for *prefix sum* and the new $O(\log m)$ sorting algorithm are correct (Theorems 8.8 and 8.19, respectively). These proofs demonstrate that AuDaLa's separation of concerns also helps to separate the proofs into two independent parts.

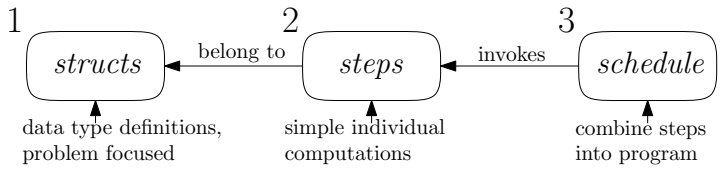

Figure 2: The three main components of an AuDaLa program.

Our basic lemmas and correctness proofs serve as inspiration for the techniques required for reasoning about AuDaLa programs. They can thus form the basis for a future proof system.

Overview. We first discuss the concepts of AuDaLa in Section 2. We then give the syntax of AuDaLa in Section 3, the type system in Section 4 and a semantics in Section 5. We discuss more examples in Section 6. We then prove some properties of AuDaLa in Section 7 before proving two example programs correct in Section 8. Lastly, we review related work in Section 9 and conclude in Section 10.

All sections benefit from having read Section 2. Sections 4, 5 and 6 require knowledge of Section 3, and Sections 7 and 8 require knowledge of Sections 3, 4 and 5. Section 8 also requires familiarity with the code for prefix sum presented in Section 2 and the code for the $O(\log m)$ sorting algorithm presented in Section 6.3. Sections 9 and 10 benefit from any section, but only require the introduction to have been read.

2. Concepts And Motivating Example

In this section, we first discuss the concepts of AuDaLa, and subsequently we design a program for the prefix sum problem in AuDaLa as a motivating example.

AuDaLa has three main components: *structs*, *steps* and a *schedule*. The relation between these components is shown in Figure 2. Structs are data type definitions from which data elements are instantiated during runtime. They contain the name of the data type and the parameters available to data elements of that type. See Listing 3 for an example of a struct definition.

When starting an AuDaLa program, a special *null*-instance is created for every struct, which is a data element representing non-existing instances of that struct. It can be considered a constant *default* instance of a data type, which is inherently stable in the values of its parameters. This is distinct from the use of null-pointers in other common languages, which do not point to an existing address. Among other things null-instances can be used for initialisation, since it already exists when launching the program. Note that this means that reading a null-instance is a valid operation in AuDaLa and does not warrant an interruption to halt the program, like reading null-pointers does for other programming languages.

Each struct contains zero or more *steps*, which represent operations a data element instantiated from that struct can do. A step contains simple, algorithmic code, consisting of conditions and assignments, without loops. This makes steps finite and easy to reason about. Within a step, it is possible to access the parameters of the surrounding struct and also to follow references stored in those parameters. Since these access patterns are known at compile-time, we can increase memory locality by grouping struct instances in a suitable manner.

The *schedule* prescribes an execution order on the steps. It contains step references and fixpoint operators (*Fix*). The occurrences of step references and fixpoint operators are separated by synchronization barriers ('<'). Execution only proceeds past a barrier when all computations that precede the barrier have concluded. Whenever a step occurs in the schedule, it is executed in parallel by all data elements which contain that step, although it is also possible to invoke a step for data elements of a specific type. AuDaLa programs are thus inherently parallel.

We do not make assumptions about a global execution order of statements executed in parallel. In particular, code is not necessarily executed by multiple struct instances in *lock-step*. Furthermore, we allow the occurrence of race conditions within one step, see also Section 6. Thus, barriers (and implicit barriers, see below) are the main method of synchronisation.

```
kernel void koggeStone(const local T *in, local T *out) {
  out[tid] = in[tid];
  barrier();
  for (unsigned offset = 1; offset < n; offset *= 2){
    T temp;
    if (tid ≥ offset) temp = out[tid − offset];
    barrier();
    if (tid ≥ offset) out[tid] = temp ⊕ out[tid];
    barrier();
}}
```

Listing 1: OpenCL kernel for Prefix Sum (from [18])

```
__global__ void scan(float *g_odata, float *g_idata, int n){
  extern __shared__ float temp[];
  int thid = threadIdx.x;
  int pout = 0, pin = 1;
  temp[pout*n + thid] = (thid > 0) ? g_idata[thid−1] : 0;
  __syncthreads();
  for (int offset = 1; offset < n; offset *= 2){
    pout = 1 − pout; // swap double buffer indices
    pin = 1 − pout;
    if (thid >= offset)
      temp[pout*n+thid] += temp[pin*n+thid − offset];
    else
      temp[pout*n+thid] = temp[pin*n+thid]
    __syncthreads();
  }
  g_odata[thid] = temp[pout*n+thid];
}
```

Listing 2: CUDA kernel for Prefix Sum (or Scan) (from [24])

Iterative behaviour is achieved through a fixpoint operator, which executes its body repeatedly until an iteration occurs in which no data is changed. At this point, a fixpoint is reached and the schedule continues past the fixpoint operator. Between the iterations of a fixpoint, there is an implicit synchronisation barrier. For an example schedule, see Listing 4.

To give an example of these components in action, we consider the *prefix sum* problem: given a sequence of integers $x_1, \ldots, x_n$, we compute for each position $1 \leq k \leq n$ the sum $\Sigma_{i=1}^{k} x_i$. We have included OpenCL and CUDA implementations of the problem that previously occurred in the literature [18, 24], see Listings 1 and 2. Here, we omit the initialization to focus on the kernels. In Listing 1, the kernel first copies the input array to the output array, after which it uses an offset variable to check whether an element of the array needs to keep updating itself with other elements. It uses an auxiliary element "temp" to facilitate this update. In Listing 2, the kernel uses a single array which is twice as big as the original array to alternate the updates between the front half and the back half of the array. The method is similar to the method of the the OpenCL example, but it does need to synchronise less due to the alternation in the array. Both kernels require synchronization barriers in their code, as well as an offset variable to check which data needs to be operated on, against which the thread ids need to be checked multiple times per execution. This makes the intended use of the program opaque to those not initially familiar with CUDA or OpenCL.

To design a corresponding AuDaLa program, we follow the design structure suggested in Figure 2. As before, we omit the initialization. In the prefix sum problem, the input is a sequence of integers. We model an element of this sequence with a struct *Position* containing a value *val*. We also give every *Position* a

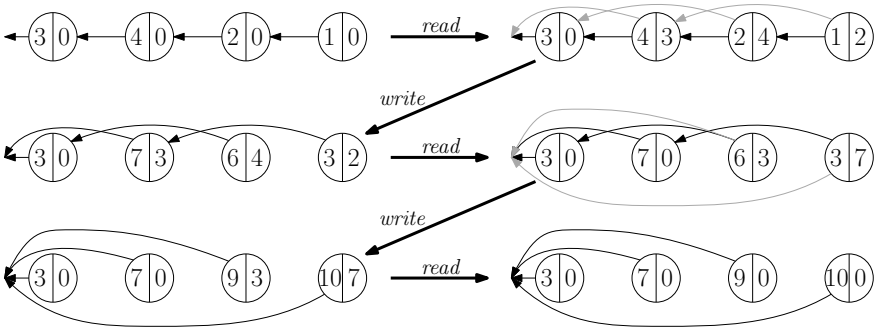

Figure 3: Execution of Prefix Sum on a small list. The left side of a list element holds the parameter *val*, while the right side holds the parameter *auxval*. The parameter *prev* is shown as unmarked black arrows, while the parameter *auxprev* is shown as unmarked grey arrows.

reference to the preceding *Position*, contained in parameter *prev*, as seen in Listing 3. This is needed to compute the prefix sum. The value of *prev* for the first position in the list is set to *null*, referencing the null-*Position*. This null-instance has the values 0 for *val* and *null* for *prev*. In Listing 4 the steps *read* and

```
1  struct Position(val: Int, prev: Position){ [...] }
```

Listing 3: Partial AuDaLa code for the structs for Prefix Sum

write of the *Position* struct are shown. These steps are based on the method for computing prefix sum in parallel, as shown in Figure 3, which was introduced by Hillis and Steele [30]. Every *Position* first reads *prev.prev* and *prev.val* from their predecessor in the step *read*, and after synchronisation, every position updates their *prev* to *prev.prev* and their *val* to *val* + *prev.val* in the step *write*. As the scope of local variables in AuDaLa does not exceed a step, the use of additional parameters *auxprev* and *auxval* in the *read* step is required to recover the value in the *write* step. The steps do not need an *offset* variable like the CUDA and OpenCL kernels, as *Positions* which reached the beginning of the list have a null-instance as predecessor and can still execute the steps.

For our program schedule, we want to repeat *read* and then *write* until all *Positions* have reached the beginning of the list, which results in the schedule as shown in Listing 4. Eventually, all *Positions* will have *null* as their predecessor and no parameters will change further (as *null.val* = 0 and *null.prev* = *null*), causing the fixpoint to terminate.

```
1  struct Position(val: Int, prev: Position, auxval: Int,  auxprev: Position){
2    read {                                                          /*step definition*/
3      auxval := prev.val;
4      auxprev := prev.prev;
5    }
6    write {                                                         /*step definition*/
7      val := val + auxval;
8      prev := auxprev;
9  }}
10
11 Fix(read < write)                                                 /*schedule*/
```

Listing 4: AuDaLa code for Prefix Sum with steps and a schedule

As illustrated by Listing 4 and by Figure 2, AuDaLa has a high *separation of concerns*: structs model data and their attributes, steps contain the algorithmic code and the schedule contains the execution. This

approach requires no synchronization barriers in the user code for the steps, no variables to find the right indices for memory access and no offset variables to avoid going out of bounds. While it may still be opaque to people not familiar with AuDaLa, we are confident that the threshold for attaining this familiarity is generally lower than hardware-focused languages like CUDA and OpenCL.

3. Syntax

In this section, we highlight the most important parts of the concrete syntax of AuDaLa. In the definitions below, non-terminals are indicated with ⟨–⟩ and symbols with quotes; the empty word is ε. The non-terminal *Id* describes identifiers, and the non-terminal *Type* describes type names, which are either `Int`, `Nat` (natural number), `Bool`, `String` or an identifier (the name of a struct).

An AuDaLa *Program* consists of a list of definitions of *structs* and a schedule:

$$\langle Program\rangle ::= \langle Defs\rangle\ \langle Sched\rangle$$
$$\langle Defs\rangle ::= \langle Struct\rangle \mid \langle Struct\rangle\ \langle Defs\rangle$$

A *struct* definition gives the struct a type name (*Id*), a list of parameters (*Pars*) and a number of steps (*Steps*):

$$\langle Struct\rangle ::= \text{`}\texttt{struct}\text{'}\ \langle Id\rangle\ \text{`}\texttt{(}\text{'}\ \langle Pars\rangle\ \text{`}\texttt{)}\text{'}\ \text{`}\texttt{\{}\text{'}\ \langle Steps\rangle\ \text{`}\texttt{\}}\text{'},$$
$$\langle Pars\rangle ::= \langle Par\rangle\langle ParList\rangle \mid \varepsilon$$
$$\langle ParList\rangle ::= \text{`}\texttt{,}\text{'}\ \langle Par\rangle\langle ParList\rangle \mid \varepsilon$$
$$\langle Par\rangle ::= \langle Id\rangle\ \text{`}\texttt{:}\text{'}\ \langle Type\rangle$$

Steps are defined with a step name (*Id*) and a list of statements:

$$\langle Steps\rangle ::= \langle Id\rangle\ \text{`}\texttt{\{}\text{'}\ \langle Stats\rangle\ \text{`}\texttt{\}}\text{'}\ \langle Steps\rangle \mid \varepsilon$$
$$\langle Stats\rangle ::= \langle Stat\rangle\ \langle Stats\rangle \mid \varepsilon$$

A *statement* adheres to the following syntax:

⟨*Stat*⟩ ::=	`if` ⟨*Exp*⟩ `then` `{` ⟨*Stats*⟩ `}`	if-then statement
	⟨*Type*⟩ ⟨*Id*⟩ `:=` ⟨*Exp*⟩ `;`	variable assignment
	⟨*Var*⟩ `:=` ⟨*Exp*⟩ `;`	variable update
	⟨*Id*⟩ `(` ⟨*Exps*⟩ `)` `;`	constructor statement

The *Id* in the variable assignment is a variable name. The constructor statement spawns a new data element of the type determined by *Id*, with parameter values determined by the expressions *Exps*. The syntax of *Exps* is similar to that of *Pars*, using *ExpList* and *Exp*. The syntax for a single *expression Exp* is as follows:

⟨*Exp*⟩ ::=	⟨*Exp*⟩ ⟨*BOp*⟩ ⟨*Exp*⟩	binary operator expression
	`(` ⟨*Exp*⟩ `)`	brackets
	`!` ⟨*Exp*⟩	negation
	⟨*Id*⟩ `(` ⟨*Exps*⟩ `)`	constructor expression
	⟨*Var*⟩	variable expression
	⟨*Literal*⟩	literal expression
	`null`	null expression
	`this`	this expression

We consider `this` to refer to the data element from which the code which includes `this` is executed.

Binary Operators follow the syntax

$$\langle BOp\rangle ::= \text{`}\texttt{=}\text{'} \mid \text{`}\texttt{!=}\text{'} \mid \text{`}\texttt{<=}\text{'} \mid \text{`}\texttt{>=}\text{'} \mid \text{`}\texttt{<}\text{'} \mid \text{`}\texttt{>}\text{'} \mid \text{`}\texttt{*}\text{'} \mid \text{`}\texttt{/}\text{'} \mid \text{`}\texttt{\%}\text{'} \mid \text{`}\texttt{+}\text{'} \mid \text{`}\texttt{-}\text{'} \mid \text{`}\texttt{\^{}}\text{'} \mid \text{`}\texttt{\&\&}\text{'} \mid \text{`}\texttt{||}\text{'},$$

which represent the binary operations for which the notation is commonly used.

A *variable* reference follows the syntax:

$$\langle Var \rangle ::= \langle Id \rangle \text{ `.'} \langle Var \rangle \mid \langle Id \rangle,$$

where in both cases *Id* is the name of a variable. Through the first case, one can access the parameters of parameters. For example, *prev.prev.val* would have been valid AuDaLa in Listing 4, and would access the value of the *Position* before the previous *Position* of the current *Position*.

Lastly, the schedule consists of the variants as given in the following syntax:

$\langle Sched \rangle ::=$	$\langle Id \rangle$	step execution
	$\mid \langle Id \rangle$ `.` $\langle Id \rangle$	typed step execution
	$\mid \langle Sched \rangle$ `<` $\langle Sched \rangle$	barrier composition
	$\mid$ `Fix` `(` $\langle Sched \rangle$ `)`	fixpoint calculation

The *Id* in the step execution is a step name. In the typed step execution, the first *Id* is a type name, while the second is a step name. The typed step execution is used to schedule a step executed by only one specific struct type.

4. Well-Formedness of AuDaLa programs

Like many other programming languages, we put type constraints and other syntactical constraints on the syntax of an acceptable AuDaLa program. To formalise what is and what is not considered well-formed for an AuDaLa program, we give a *type system* [31, 32, 33, 34] for AuDaLa in this section. With the type system of AuDaLa, we root out standard *type errors*, where there is a mismatch between the expected type of an expression or a variable and the actual type presented, as well as any breach of 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.
4. Names of local variables do not overlap with parameter names of the surrounding struct definition.
5. Variable assignment statements (as defined in the syntax) are only used to declare new local variables.
6. A local variable is only used after its declaration in a variable assignment statement.

In Table 1, we give the type rules of the type system. These rules use a *syntax variable environment* Γ and a *struct type environment* Ω, both of which are ordered lists separated by ;. The environment Γ consists of pairs of variable names and their types; the variable names present in Γ can be accessed using $dom(\Gamma)$. The environment Ω saves struct type definitions and their parameters, of which the struct types can be accessed using $dom(\Omega)$.

We denote struct types using variations of the letter θ. We assume that for every occurrence of `this`, the struct in which it occurs is recorded in such a way that every occurrence of `this` is annotated with the name of its struct, like $\texttt{this}_\theta$. We also assume that any list, like a list of parameters, is separated by semicolons even if the syntax specifies commas. We use the shorthand *Pars* to denote a list of parameters $p_1 : T_1; \ldots$, as per the non-terminal $\langle Pars \rangle$. To make working with the schedule easier, we treat occurrences of s_θ for some step s as a variable name for a variable of a special newly introduced type `Step`.

We use *Str* to denote any sequence of characters admissible by the syntax and let N_1 and N_2 s.t. $N_1, N_2 \in \{\texttt{Nat}, \texttt{Int}\}$. Additionally, let S be a statement and let $\mathcal{S}$ be a sequence of statements separated with ;. Let E, E_1, E_2 be expressions, let x be a single variable name and let X be a variable following the non-terminal $\langle Var \rangle$. Lastly, let Sc, Sc_1 and Sc_2 be schedules and let $K = \{\texttt{this}, \texttt{if}, \texttt{then}, \texttt{null}, \texttt{Fix}, \texttt{struct}\}$ be the set of keywords. Let ε_{Sc} be the empty schedule.

There are 7 categories of type rules. The first category, *Environment*, deals with the well-formedness of the environment and the types and struct types contained in it. The category *Types* deals with checking whether types are well-formed and are admitted by the environment. Note that the rule **DefStruct** uses the

Environment

$$(\textbf{Empty})\ \frac{}{\emptyset, \emptyset \vdash \diamond} \qquad (\textbf{AddType})\ \frac{\Gamma, \Omega \vdash \diamond \quad \theta \notin dom(\Omega) \quad \theta \notin K \quad \forall p_i, p_j \in (p_1, \ldots, p_n).(p_i \neq p_j)}{\Gamma, \Omega; \theta(p_1 : T_1, \ldots, p_n : T_n) \vdash \diamond}$$

$$(\textbf{AddVar})\ \frac{\Gamma, \Omega \vdash T \quad v \notin dom(\Gamma) \quad v \notin K}{\Gamma; v : T, \Omega \vdash \diamond}$$

Types

$$(\textbf{TBasic})\ \frac{\Gamma, \Omega \vdash \diamond \quad T \in \{\texttt{Nat}, \texttt{Int}, \texttt{Bool}, \texttt{String}, \texttt{Step}\}}{\Gamma, \Omega \vdash T}$$

$$(\textbf{TStruct})\ \frac{\Gamma, \Omega; \theta(Pars); \Omega' \vdash \diamond}{\Gamma, \Omega; \theta(Pars); \Omega' \vdash \theta} \qquad (\textbf{DefStruct})\ \frac{\Gamma, \Omega \vdash \theta \quad \forall_{1 \leq i \leq n} T_i.\Gamma, \Omega \vdash T_i}{\Gamma, \Omega \vdash \theta(p_1 : T_1, \ldots, p_n : T_n)}$$

Standard Values

$$(\textbf{Bool})\ \frac{\Gamma, \Omega \vdash \diamond \quad x \in \{true, false\}}{\Gamma, \Omega \vdash x : \texttt{Bool}} \qquad (\textbf{String})\ \frac{\Gamma, \Omega \vdash \diamond}{\Gamma, \Omega \vdash \text{“}Str\text{”} : \texttt{String}} \qquad (\textbf{NInt})\ \frac{\Gamma, \Omega \vdash \diamond \quad c < 0}{\Gamma, \Omega \vdash c : \texttt{Int}}$$

$$(\textbf{PNat})\ \frac{\Gamma, \Omega \vdash \diamond \quad c \geq 0}{\Gamma, \Omega \vdash c : \texttt{Nat}} \qquad (\textbf{Null})\ \frac{\Gamma, \Omega \vdash T}{\Gamma, \Omega \vdash \texttt{null} : T} \qquad (\textbf{This})\ \frac{\Gamma, \Omega \vdash \theta}{\Gamma, \Omega \vdash \texttt{this}_\theta : \theta}$$

Expressions

$$(\textbf{Eq})\ \frac{\Gamma, \Omega \vdash E_1 : T \wedge \Gamma, \Omega \vdash E_2 : T \quad \circ \in \{=, !=\}}{\Gamma, \Omega \vdash E_1 = E_2 : \texttt{Bool}} \qquad (\textbf{Comp})\ \frac{\Gamma, \Omega \vdash E_1 : N_1 \quad \Gamma, \Omega \vdash E_2 : N_2 \quad \circ \in \{<=, >=, <, >, =, !=\}}{\Gamma, \Omega \vdash E_1 \circ E_2 : \texttt{Bool}}$$

$$(\textbf{NArith})\ \frac{\Gamma, \Omega \vdash E_1 : \texttt{Nat} \quad \Gamma, \Omega \vdash E_2 : \texttt{Nat} \quad \circ \in \{*, /, \%, +, \hat{}\}}{\Gamma, \Omega \vdash E_1 \circ E_2 : \texttt{Nat}}$$

$$(\textbf{IArith})\ \frac{\Gamma, \Omega \vdash E_1 : N_1 \quad \Gamma, \Omega \vdash E_1 : N_2 \quad \circ \in \{*, /, \%, +, \hat{}, -\}}{\Gamma, \Omega \vdash E_1 \circ E_2 : \texttt{Int}} \qquad (\textbf{BinLog})\ \frac{\Gamma, \Omega \vdash E_1 : \texttt{Bool} \quad \Gamma, \Omega \vdash E_2 : \texttt{Bool} \quad \circ \in \{\&\&, ||\}}{\Gamma, \Omega \vdash E_1 \circ E_2 : \texttt{Bool}}$$

$$(\textbf{Brackets})\ \frac{\Gamma, \Omega \vdash E : T}{\Gamma, \Omega \vdash (E) : T}$$

$$(\textbf{Neg})\ \frac{\Gamma, \Omega \vdash E : \texttt{Bool}}{\Gamma, \Omega \vdash !E : \texttt{Bool}} \qquad (\textbf{ConsE})\ \frac{\forall_{0 \leq i \leq n}.\Gamma, \Omega \vdash E_i : T_i}{\Gamma, \Omega; \theta(p_1 : T_1, \ldots p_n : T_n); \Omega' \vdash \theta(E_1, \ldots, E_n) : \theta} \qquad (\textbf{VarS})\ \frac{\Gamma_1; (x : T); \Gamma_2, \Omega \vdash \diamond}{\Gamma_1; (x : T); \Gamma_2, \Omega \vdash x : T}$$

$$(\textbf{VarR})\ \frac{\Gamma, \Omega; \theta(p_1 : T_1, \ldots p_n : T_n); \Omega' \vdash X : \theta \quad \Gamma, \Omega; \theta(p_1 : T_1, \ldots p_n : T_n); \Omega' \vdash T_i}{\Gamma, \Omega; \theta(p_1 : T_1, \ldots p_n : T_n); \Omega' \vdash X.p_i : T_i}$$

Statements

$$(\textbf{IfThen})\ \frac{\Gamma, \Omega \vdash E : \texttt{Bool} \quad \Gamma, \Omega \vdash \mathcal{S}}{\Gamma, \Omega \vdash \text{if } E \text{ then}\{\mathcal{S}\}} \qquad (\textbf{LVar})\ \frac{\Gamma, \Omega \vdash E : T \quad \Gamma_1; (x : T); \Gamma_2, \Omega \vdash \diamond}{\Gamma, \Omega \vdash Tx := E} \qquad (\textbf{Update})\ \frac{\Gamma, \Omega \vdash X : T \quad \Gamma, \Omega \vdash E : T}{\Gamma, \Omega \vdash X := E}$$

$$(\textbf{ConsS})\ \frac{\forall_{0 \leq i \leq n}.\Gamma, \Omega \vdash E_i : T_i}{\Gamma, \Omega; \theta(p_1 : T_1, \ldots p_n : T_n); \Omega' \vdash \theta(E_1, \ldots, E_n)} \qquad (\textbf{Seq})\ \frac{\Gamma, \Omega \vdash S \quad \Gamma, \Omega \vdash \mathcal{S}}{\Gamma, \Omega \vdash S; \mathcal{S}} \qquad (\textbf{SeqLVar})\ \frac{\Gamma, \Omega \vdash Tx := E \quad \Gamma; (x : T), \Omega \vdash \mathcal{S}}{\Gamma, \Omega \vdash Tx := E; \mathcal{S}}$$

$$(\textbf{SeqIT})\ \frac{\Gamma, \Omega \vdash \text{if } E \text{ then}\{\mathcal{S}_1\} \quad \Gamma, \Omega \vdash \mathcal{S}_2}{\Gamma, \Omega \vdash \text{if } E \text{ then}\{\mathcal{S}_1\}\ \mathcal{S}_2}$$

Schedule

$$(\textbf{SBar})\ \frac{\Gamma, \Omega \vdash sc_1 \quad \Gamma, \Omega \vdash sc_2}{\Gamma, \Omega \vdash sc_1 < sc_2} \qquad (\textbf{SFix})\ \frac{\Gamma, \Omega \vdash sc}{\Gamma, \Omega \vdash \texttt{Fix}(sc)} \qquad (\textbf{STypCall})\ \frac{\Gamma, \Omega \vdash F_\theta : \texttt{Step}}{\Gamma, \Omega \vdash \theta.F} \qquad (\textbf{SNormCall})\ \frac{\Gamma, \Omega \vdash F_\theta : \texttt{Step}}{\Gamma, \Omega \vdash F}$$

Program

$$(\textbf{Prog})\ \frac{\forall \theta(Pars) \in Structs(D).\Gamma, \Omega; Structs(D) \vdash \theta(Pars) \quad \forall (F_\theta, \mathcal{S}) \in StepDef(D).\Gamma; Pars(D, \theta), \Omega; Structs(D) \vdash \mathcal{S} \quad \Gamma; Steps(D), \Omega \vdash Sc}{\Gamma, \Omega \ \vdash \ D\ Sc}$$

Table 1: The type system of AuDaLa, ordered by category.

'for all' notation, but that the predicate can be decomposed into a predicate of the form $\Gamma, \Omega \vdash T_1 \wedge \ldots \wedge \Gamma, \Omega \vdash T_n$. Alternatively, the rule can be decomposed in a set of rules which peel of the parameter list one by one.

The third category, *Expressions*, deals with checking whether expressions are well-typed, while the fourth category, *Statements*, deals with checking whether statements are admitted according to the type system. Note that expressions have types, while statements do not, which is exemplified by the two distinct rules for constructor expressions and constructor statements. Also note that variable assignment statements have their own sequence rule which adds the newly assigned variable to the environment, and that the if-then statements have their own sequence rule because they do not end with an ';' in the syntax. The sixth category, *Schedule*, deals with the schedule by checking whether all step names in the schedule have been assigned the type `Step`. We consider ε to be the empty schedule, which we allow.

The last category, *Program*, consists of only one rule, **Prog**. This is the rule which decomposes checking a program into multiple subtasks to check. In this rule, we define that a program consisting of definitions and a schedule is well-formed iff all struct types are defined well (top line), all steps are well defined (second line left) and the schedule is well defined (second line right). We denote the topmost syntactic *Defs* element as D and the schedule as Sc. We use the function $Structs(D)$ to extract struct type definition information from D, which consists of all struct types and their parameters formatted as $\theta(Pars)$. We overload $Pars$ with a function s.t. $Pars(D, \theta)$ returns the parameters of θ in D. We use the function $Steps(D)$ to extract step definition information from D, consisting of the step and its struct type. We consider these steps to have the type `Step`, so the output is formatted as a list consisting of elements with format F_θ : `Step`, where F is a step. We use the function $StepDef(D)$ to extract step definition information coupled with their statements, which outputs a list of elements of the format $(F_\theta, \mathcal{S})$.

Definition 4.1 (AuDaLa well-formedness). An AuDaLa program $\mathcal{P}$ is well-formed iff $\mathcal{P}$ is derivable using the rules in Table 1.

5. Semantics

In this section, we present the semantics of AuDaLa. These semantics take the form of a stack machine, as stacks are naturally suited for expressing the evaluation of expressions, and the stack machine also exposes the interleaving in the semantics very well, which we think suitable for a concurrent programming language.

To define the semantics, we start by defining some sets used in the semantics and which syntactical sets they mirror, if any. We then define *commands*, used as primitives for the behaviour encoded in the syntax, and a function from the syntax to commands. We define the states of an AuDaLa program, and then use the commands and the schedule to define operational behaviour of an AuDaLa program. We conclude with Definition 5.7.

In the semantics, we regularly use lists. List concatenation is denoted with a semicolon, and we identify a singleton list with its only element. The empty list is denoted ε.

If we work with lists of statements, we assume that in this list, the statements are also separated by a semicolon, even though if-statements do not have a semicolon at the end in the syntax. While the schedules are represented using their syntax, in the semantics, we do consider schedules to be lists separated by $<$. It follows that schedules can be empty (ε) and that $sc = sc < \varepsilon$. However, note that in the syntax, an empty schedule is not admitted in the type system, so every program starts with a non-empty schedule.

We define updates for functions as follows. Given a function $f : A \to B$ and $a \in A$ and $b \in B$, then $f[a \mapsto b](a) = b$ and $f[a \mapsto b](x) = f(x)$ for all $x \neq a$. We lift this operation to sets of updates: $f[\{a_1 \mapsto b_1, a_2 \mapsto b_2, \ldots\}] = f[a_1 \mapsto b_1][a_2 \mapsto b_2] \ldots$. Since the order of applying updates is relevant, this is only well-defined if the left-hand sides $a_1, a_2, \ldots$ are pairwise distinct. If B contains tuples, that is, $B = B_1 \times \ldots \times B_n$, we can also update a single element of a tuple: if $f(a) = \langle b_1, \ldots, b_n \rangle$, then we define $f[a, i \mapsto b](a) = \langle b_1, \ldots, b_{i-1}, b, b_{i+1}, \ldots, b_n \rangle$ and $f[a, i \mapsto b](x) = f(x)$ for all $x \neq a$.

We assume the existence of a parser for the concrete syntax. Henceforth, we work on an *abstract syntax tree* (AST) produced by running the parser on a well-formed AuDaLa program. We thus do not concern ourselves with operator precedence and parentheses, and we assume that polymorphic elements such as `null` and 42 are labelled with the right type for their context, *viz.*, $null_T$ is the expression *null* of type T.

We have a number of sets containing AST elements: ID is the set of all identifiers, LT is the set of all literals, SC is the set of all schedules, ST is the set of all statements, EX contains all expressions and O contains all syntactic binary operators. The set containing all syntactic types is $\mathcal{T} = \{\texttt{Nat}, \texttt{Int}, \texttt{Bool}, \texttt{String}\} \cup ID$, corresponding to the non-terminal $\langle Type \rangle$.

In our semantics, *labels* reference concrete instances of structs (as opposed to struct definitions). We assume some sufficiently large set $\mathcal{L}$ containing these labels. We also have the semantic types $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{B}$ and *String* corresponding to the natural numbers, the integers, the booleans and the set of all strings, respectively. We consider the natural numbers to be a subset of the integers containing all positive numbers and 0. All semantic values are collected in $\mathcal{V} = \mathcal{L} \sqcup \mathbb{Z} \sqcup \mathbb{B} \sqcup String$ (where $\sqcup$ denotes the disjoint union). The set of all semantic types is $\mathbb{T} = \{\mathcal{L}, \mathbb{N}, \mathbb{Z}, \mathbb{B}, String\}$. The semantic value of a literal $g \in LT$ is $val(g)$. We consider the semantic value of `this` to be the label $\ell \in \mathcal{L}$ of the struct instance executing the line of code in which `this` occurs.

In addition, we assume for every struct type θ the existence of a *null-label* $\ell^0_\theta \in \mathcal{L}$, so that we can provide a default value for each syntactical type with the function $defaultVal : \mathcal{T} \to \mathcal{V}$, defined as:

$$defaultVal(T) = \begin{cases} 0 & \text{if } T = \texttt{Nat} \text{ or } T = \texttt{Int} \\ false & \text{if } T = \texttt{Bool} \\ \varepsilon & \text{if } T = \texttt{String} \\ \ell^0_T & \text{if } T \in ID \end{cases}.$$

We define the set of all null-labels to be $\mathcal{L}^0$, with $\mathcal{L}^0 \subset \mathcal{L}$.

To facilitate conciseness in our operational semantics, we break down statements and expressions into *commands*, which can be viewed as atomic actions in the semantics.

Definition 5.1 (Commands). A *command* c is constructed according to the following grammar:

$$c ::= \mathbf{push}(v) \mid \mathbf{rd}(x) \mid \mathbf{wr}(x) \mid \mathbf{cons}(x) \mid \mathbf{if}(C) \mid \mathbf{not} \mid \mathbf{op}(o)$$

where $v \in \mathcal{V} \cup \{\mathbf{this}\}$ is a semantic value or **this**, a special value, $x \in ID$ is an identifier, C is a list of commands, and $o \in O$ is an operator. The set of all commands is $\mathcal{C}$.

Intuitively, **this** is the semantic equivalent to the syntactic `this`-expression. The precise effect of each command is discussed later in this section when the inference rules are given. Statements and expressions are compiled into a list of commands according to the following recursive interpretation function:

Definition 5.2 (Interpretation function). Let $x, x_1, \ldots, x_n \in ID$ be variables, $E, E_1, \ldots, E_m \in EX$ 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 $\texttt{op} \in O$ 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* $[\![\cdot]\!] : ST^* \cup EX \to \mathcal{C}^*$ transforming a list of statements or expressions into a list of commands:

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

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.

During the runtime of a program, multiple instances of a struct definition may exist simultaneously. We refer to these as *struct instances*.

Definition 5.3 (Struct instance). A *struct instance* is a tuple $\langle \theta, \gamma, \chi, \xi \rangle$ where:

- $\theta \in ID$ is the type of the struct,
- $\gamma \in \mathcal{C}^*$ is a list of commands that are to be executed,
- $\chi \in \mathcal{V}^*$ is a stack that stores values during the evaluation of an expression,
- $\xi : ID \to \mathcal{V}$ is an environment that stores the values of local variables as well as parameters.

We define $\mathcal{S}$ as the set of all possible struct instances.

A state of a program is the combination of a schedule that remains to be executed, a collection with all the struct instances that currently exist and a stack of Boolean values that are required to determine whether a fixpoint has been reached. Note that every label can refer to at most one distinct struct instance.

Definition 5.4 (State). A *state* is a tuple $\langle Sc, \sigma, s\chi \rangle$, where:

- $Sc \in SC$ is a schedule expressed as a list,
- $\sigma : \mathcal{L} \to \mathcal{S} \cup \{\bot\}$ is a *struct environment*,
- $s\chi \in \mathbb{B}^*$ is a *stability stack*.

The set of all states is defined as $S_{\mathcal{G}} = SC \times (\mathcal{L} \to \mathcal{S} \cup \{\bot\}) \times \mathbb{B}^*$.

Intuitively, the stability stack keeps track of whether a fixpoint should terminate. When starting execution of this fixpoint, a new variable is placed on the stability stack corresponding to this fixpoint. This variable is set to true every time the fixpoint iterates, and is set to false when a parameter is changed during the execution of the fixpoint. Consider for example the code for Prefix Sum (Listing 4). In it, the fixpoint for *read* $<$ *write* will place a single variable on the stability stack, which is reset to *true* at the start of an iteration and only remains *true* if no element updates its *prev* or *val* parameters to new values.

With a notion of states and struct instances, we define *null-instances*:

Definition 5.5 (Null-instances). Let $P = \langle Sc, \sigma, s\chi \rangle \in S_{\mathcal{G}}$ be a state. Then the set of *null-instances* in state P is defined as $\{\sigma(\ell) \mid \sigma(\ell) \neq \bot \wedge \ell \in \mathcal{L}^0\}$.

Thus, each struct instance that is labelled with a *null*-label is a *null*-instance.

Henceforth, we fix an AuDaLa program $\mathcal{P} \in \Pi$ s.t. $\mathcal{P} = D\ Sc_{\mathcal{P}}$, with D the definitions of the program and $Sc_{\mathcal{P}}$ the schedule of the program. We define $\Theta_{\mathcal{P}} \subseteq ID$ to be the set of all struct types defined in $\mathcal{P}$. The initial variable environment for a struct instance of type θ is ξ^0_θ, defined as $\xi^0_\theta(p) = \mathit{defaultVal}(T)$ for all $p \in \mathit{Pars}(\theta, \mathcal{P})$ where T is the type of p and $\mathit{Pars}(\theta, \mathcal{P})$ refers to the parameters of θ in $\mathcal{P}$ and is equal to $\mathit{Pars}(\theta, D)$ as used last section. For other variables $x \in ID$, $\xi^0_\theta(x)$ is left arbitrary.

The initial state of an AuDaLa program depends on which program is going to be executed (the state space does not depend on the program, *cf.* Def. 5.4):

Definition 5.6 (Initial state). 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.

Intuitively, this definition states that the initial state of a program $\mathcal{P}$ consists of the schedule as found in the program, a struct environment filled with *null*-instances for every struct type declared in $\mathcal{P}$ and an empty stack.

We proceed by defining the transition relation $\Rightarrow_{\mathcal{P}}$ by means of inference rules. In this relation, we consider every transition to be labelled by the inference rule that induced the transition. The dependence of $\Rightarrow_{\mathcal{P}}$ on $\mathcal{P}$ stems from the dependence on parameter information from $\mathcal{P}$. There are rules that define the execution of commands and rules for the execution of a schedule. We start with the former. Command $\mathbf{push}(v)$ pushes value v on the stack χ, and $\mathbf{push}(\mathbf{this})$ pushes the label of the structure instance on χ:

$$(\mathbf{ComPush})\frac{\sigma(\ell) = \langle \theta, \mathbf{push}(v); \gamma, \chi, \xi \rangle}{\langle Sc, \sigma, s\chi \rangle \Rightarrow_{\mathcal{P}} \langle Sc, \sigma[\ell \mapsto \langle \theta, \gamma, \chi; v, \xi \rangle], s\chi \rangle}$$

$$(\mathbf{ComPushThis})\frac{\sigma(\ell) = \langle \theta, \mathbf{push}(\mathbf{this}); \gamma, \chi, \xi \rangle}{\langle Sc, \sigma, s\chi \rangle \Rightarrow_{\mathcal{P}} \langle Sc, \sigma[\ell \mapsto \langle \theta, \gamma, \chi; \ell, \xi \rangle], s\chi \rangle}$$

The command $\mathbf{rd}(x)$ reads the value of variable x from environment ξ' of ℓ' and places it onto the stack:

$$(\mathbf{ComRd})\frac{\begin{array}{c}\sigma(\ell) = \langle \theta, \mathbf{rd}(x); \gamma, \chi; \ell', \xi\rangle \\ \sigma(\ell') = \langle \theta', \gamma', \chi', \xi'\rangle\end{array}}{\langle Sc, \sigma, s\chi\rangle \Rightarrow_{\mathcal{P}} \langle Sc, \sigma[\ell \mapsto \langle \theta, \gamma, \chi; \xi'(x), \xi\rangle], s\chi\rangle}$$

For normal struct instances, $\mathbf{wr}(x)$ takes a label ℓ' and a value v from the stack and writes this to $\xi'(x)$, the environment of the struct instance corresponding to ℓ'. If x is a parameter (as opposed to a local variable) and writing v changes its value, then any surrounding fixpoint in the schedule becomes unstable. In that case, we set the auxiliary value *su* (for *stability update*) to false and clear the stability stack by setting it to $s\chi_1 \wedge su; \ldots; s\chi_{|s\chi|} \wedge su$. Note that this leaves the stack unchanged if *su* is true. Below, in the update "$[\ell', 4 \mapsto \xi'[x \mapsto v]]$", recall that $f[a, i \mapsto b]$ denotes the update of a function that returns a tuple.

$$(\mathbf{ComWr})\frac{\begin{array}{c}\sigma(\ell) = \langle \theta, \mathbf{wr}(x); \gamma, \chi; v; \ell', \xi\rangle \\ \sigma(\ell') = \langle \theta', \gamma', \chi', \xi'\rangle \\ \ell' \notin \mathcal{L}^0 \vee x \notin \mathit{Pars}()(\theta', \mathcal{P}) \\ su = (x \notin \mathit{Pars}()(\theta', \mathcal{P}) \vee \xi'(x) = v)\end{array}}{\langle Sc, \sigma, s\chi\rangle \Rightarrow_{\mathcal{P}} \langle Sc, \sigma[\ell \mapsto \langle \theta, \gamma, \chi, \xi\rangle][\ell', 4 \mapsto \xi'[x \mapsto v]], s\chi_1 \wedge su; \ldots; s\chi_{|s\chi|} \wedge su\rangle}$$

The next rule skips the write if the target is a parameter of a *null*-instance, which ensures that the parameters of a *null*-instance cannot be changed. See Section 4 for the reasoning behind our use of *null*-instances. Note that we allow *null*-instances to use local variables, which is useful for the initialization of a system, for example.

$$(\mathbf{ComWrNSkip})\frac{\begin{array}{c}\sigma(\ell) = \langle \theta, \mathbf{wr}(x); \gamma, \chi; v; \ell', \xi\rangle \\ \sigma(\ell') = \langle \theta', \gamma', \chi', \xi'\rangle \\ \ell' \in \mathcal{L}^0 \wedge x \in \mathit{Pars}()(\theta', \mathcal{P})\end{array}}{\langle Sc, \sigma, s\chi\rangle \Rightarrow_{\mathcal{P}} \langle Sc, \sigma[\ell \mapsto \langle \theta, \gamma, \chi, \xi\rangle], s\chi\rangle}$$

Let b be a boolean value on the top of the stack χ. A **not** command negates b:

$$(\mathbf{ComNot})\frac{\sigma(\ell) = \langle \theta, \mathbf{not}; \gamma, \chi; b, \xi\rangle}{\langle Sc, \sigma, s\chi\rangle \Rightarrow_{\mathcal{P}} \langle Sc, \sigma[\ell \mapsto \langle \theta, \gamma, \chi; \neg b, \xi\rangle], s\chi\rangle}$$

An $\mathbf{op}(\circ)$ command applies the semantic equivalent $o \in \{=, \neq, \leq, \geq, <, >, *, /, \%, +, -, \hat{}, \wedge, \vee\}$ of the syntactic operator $\circ \in O$ to the two values at the top of χ (here taken to be the values a and b), of which the result is put on top of the stack:

$$(\mathbf{ComOp})\frac{\sigma(\ell) = \langle \theta, \mathbf{op}(\circ); \gamma, \chi; a; b, \xi\rangle}{\langle Sc, \sigma, s\chi\rangle \Rightarrow_{\mathcal{P}} \langle Sc, \sigma[\ell \mapsto \langle \theta, \gamma, \chi; (a \mathbin{o} b), \xi\rangle], s\chi\rangle}$$

Let $\theta' \in \Theta_{\mathcal{P}}$ be the type of a struct with n parameters. The command $\mathbf{cons}(\theta')$ creates a new struct instance of type θ' in the struct environment σ with a fresh label ℓ', and initializes the parameters to the top n values of the stack:

$$(\mathbf{ComCons})\frac{\begin{array}{c}\sigma(\ell) = \langle \theta, \mathbf{cons}(\theta'); \gamma, \chi; v_1; \ldots; v_n, \xi\rangle \\ \mathit{Pars}()(\theta', \mathcal{P}) = p_1 : T_1; \ldots; p_n : T_n \\ \sigma(\ell') = \bot\end{array}}{\begin{array}{l}\langle Sc, \sigma, s\chi\rangle \Rightarrow_{\mathcal{P}} \langle Sc, \sigma[\{\ell \mapsto \langle sL, \gamma, \chi; \ell', \xi\rangle, \\ \qquad \ell' \mapsto \langle sL', \varepsilon, \varepsilon, \xi^0_{\theta'}[\{p_1 \mapsto v_1, \ldots, p_n \mapsto v_n\}]\rangle\}], \mathit{false}^{|s\chi|}\rangle\end{array}}$$

The command $\mathbf{if}(C)$ with $C \in \mathcal{C}^*$ adds commands C to the start of γ if the top value of the stack is *true*. If the top value is *false*, the command does nothing:

$$(\mathbf{ComIfT})\frac{\sigma(\ell) = \langle \theta, \mathbf{if}(C); \gamma, \chi; \mathit{true}, \xi\rangle}{\langle Sc, \sigma, s\chi\rangle \Rightarrow_{\mathcal{P}} \langle Sc, \sigma[\ell \mapsto \langle \theta, C; \gamma, \chi, \xi\rangle], s\chi\rangle}$$

$$(\textbf{ComIfF})\frac{\sigma(\ell) = \langle \theta, \textbf{if}(C); \gamma, \chi; \mathit{false}, \xi\rangle}{\langle Sc, \sigma, s\chi\rangle \Rightarrow_{\mathcal{P}} \langle Sc, \sigma[\ell \mapsto \langle \theta, \gamma, \chi, \xi\rangle], s\chi\rangle}$$

In the remaining rules, let $\mathit{Done}(\sigma) = \forall \ell.(\sigma(\ell) = \bot \vee \exists \theta, \chi, \xi.\sigma(\ell) = \langle \theta, \varepsilon, \chi, \xi\rangle)$, let sc be a (possibly empty) schedule. The predicate $\mathit{Done}(\sigma)$ holds when all commands have been executed in all struct instances in σ.

We can initiate steps globally and locally. The global step initiation converts all statements in a step to commands for any structure instance that has that step and adds the commands to γ. Recall that $\mathcal{P} = D\ Sc$ and recall from last section that $\mathit{StepDef}(D)$ returns all pairs $(F_\theta, \mathcal{S})$ s.t. F is a step defined for $\theta \in \Theta_{\mathcal{P}}$ as the list of statements $\mathcal{S}$. Let $\mathcal{S}^F_\theta = \varepsilon$ if there is no list of statements $\mathcal{S}$ s.t. $(F_\theta, \mathcal{S}) \in \mathit{StepDef}(D)$, and let $\mathcal{S}^F_\theta$ be a list of statements s.t. $(F_\theta, \mathcal{S}^F_\theta) \in \mathit{StepDef}(D)$ otherwise.

$$(\textbf{InitG})\frac{\mathit{Done}(\sigma)}{\langle F < sc, \sigma, s\chi\rangle \Rightarrow_{\mathcal{P}} \langle sc, \sigma[\{\ell \mapsto \langle \theta, [\![\mathcal{S}^F_\theta]\!], \varepsilon, \xi\rangle \mid \sigma(\ell) = \langle \theta, \varepsilon, \chi, \xi\rangle\}], s\chi\rangle}$$

The local step initiation converts the step to commands and adds those commands to γ only for struct instances of a specified struct θ:

$$(\textbf{InitL})\frac{\mathit{Done}(\sigma)}{\langle \theta.F < sc, \sigma, s\chi\rangle \Rightarrow_{\mathcal{P}} \langle sc, \sigma[\{\ell \mapsto \langle \theta, [\![\mathcal{S}^F_\theta]\!], \varepsilon, \xi\rangle \mid \sigma(\ell) = \langle \theta, \varepsilon, \chi, \xi\rangle\}], s\chi\rangle}$$

Note that neither of the step initializations flush the previous local variables from the struct environments. Instead, the type system of AuDaLa as found in Section 4 limits the scope of local variables to the steps they are declared in. If required, it can also be enforced in the semantics by defining a function which takes the value of the struct instance environment if the given variable is a parameter, and $\bot$ otherwise.

Let sc_1 be a schedule. Fixpoints are initiated when first encountered:

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

The symbol *aFix* is a semantic symbol used to denote a fixpoint which has been initiated. When an initiated fixpoint is encountered again, the stability stack is used to determine whether the body should be executed again:

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

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

With these rules, we give an operational semantics for AuDaLa:

Definition 5.7 (Operational semantics). The semantics of $\mathcal{P}$ is the directed graph $\langle S_{\mathcal{G}}, \Rightarrow_{\mathcal{P}}, P^0_{\mathcal{P}}\rangle$, where $S_{\mathcal{G}}$ is the set of all states (Def. 5.4), $\Rightarrow_{\mathcal{P}}$ is the labelled transition relation for $\mathcal{P}$ as given above and $P^0_{\mathcal{P}}$ is the initial state of $\mathcal{P}$ (Def. 5.6).

The proofs of Section 7 and Section 8 are based upon the semantics as defined above. Generally, when we refer to the semantics of a program $\mathcal{P}$, we are talking about the semantics of $\mathcal{P}$ as defined in the above definition.

6. Example Algorithms

In this section, we provide more intuition on how AuDaLa works in practice by means of five example AuDaLa programs. The first creates a spanning tree, the second and third are a sorting programs and the fourth and fifth implement solutions for 3SUM. In AuDaLa, there is a struct instance for every data element. The amount of processors used is therefore assumed equal to the amount of data elements of relevance in the code. We give work (the total amount of operations), span (the length of the longest required chain of sequential operations) and running time estimates for the given programs. Other examples and implementations can be found in other AuDaLa papers [27, 29].

```
1  struct Node(dist: Int, in: Edge){}
2
3  struct Edge(s: Node, t: Node){
4    linkEdge{
5      if s.dist != −1 && t.dist = −1 then {
6        t.in := this;
7      }
8    }
9    handleEdge {
10     if t.in != null then {
11       if t.in = this then {
12         t.dist := s.dist + 1;
13       }
14       if t.in != this then {
15         s := null;
16         t := null;
17       }
18 }}}
19
20 Fix(linkEdge < handleEdge)
```

Listing 5: AuDaLa code for creating a spanning tree

6.1. Creating a spanning tree

Given a connected directed graph $G = (V, E)$ and a root node $u \in V$, we can create a spanning tree of G rooted in u using *breadth-first search*. In this tree, for every node v, the path from u to v is a shortest path in G. We do this by incrementally adding nodes from G with a higher distance to u to the spanning tree.

We first sketch our approach. In the ith BFS iteration, the algorithm adds all edges (s, t) to the tree such that the distance from u to s is $i - 1$ and the distance from u to t is still unknown. If multiple such edges lead to the same t, the algorithm uses a *race condition* to determine which edge is chosen. As any edge will suffice, this race condition is benign. The distance from u to t is then set to i and we continue with the next iteration. The program runs with $O(|V| + |E|)$ data elements. As every edge is considered only once, the amount of work performed by this algorithm is $O(|E|)$. The span of the algorithm is $O(d)$, with d the diameter of the graph, as in the worst case all edges in the path which determines the diameter of the graph need to be considered sequentially. It follows that the worst case running time for the algorithm is $O(d)$.

Contained in Listing 5 is an AuDaLa program that implements this approach. The program defines the struct *Node* (line 1) with parameters *dist*, to store the distance from root node u, and *in*, a reference to its incoming spanning tree edge. The struct *Edge* (line 3) has a source s and a target t.

During initialization, the input should be a directed graph, with a root *Node* u with *dist* 0 and with the *dist* parameter of the other *Nodes* set to -1. For every *Node*, the parameter *in* should be set to null.

Both steps in the program belong to *Edge*. The first step, *linkEdge*, first determines whether an *Edge* e from *Node* s to *Node* t is at the frontier of the tree in line 5. This is the case when s is in the tree, but t is not. If so, e nominates itself as the *Edge* connecting t to the tree, $t.in$ (line 6). This is a race condition won by only one edge for t, the edge which applies the semantic rule **ComWr** last. In the second step, *handleEdge*, if the nomination for t has finished (line 10) and e has won the nomination (line 11), e will update t's distance to the root. If e has lost, it will remove itself from the graph, here coded as setting the source and target parameters to *null* in lines 14 to 18.

To create a full spanning tree, this must be executed until all *Nodes* have a positive *dist* and all *Edges* are either $t.in$ for their target *Node* t or have *null* as their source and target. To this end, the schedule (line 20) contains a fixpoint, in which *Edges* first nominate themselves and then update the distances of new *Nodes*. This fixpoint terminates, as *Edges* in the spanning tree will continuously update their targets with

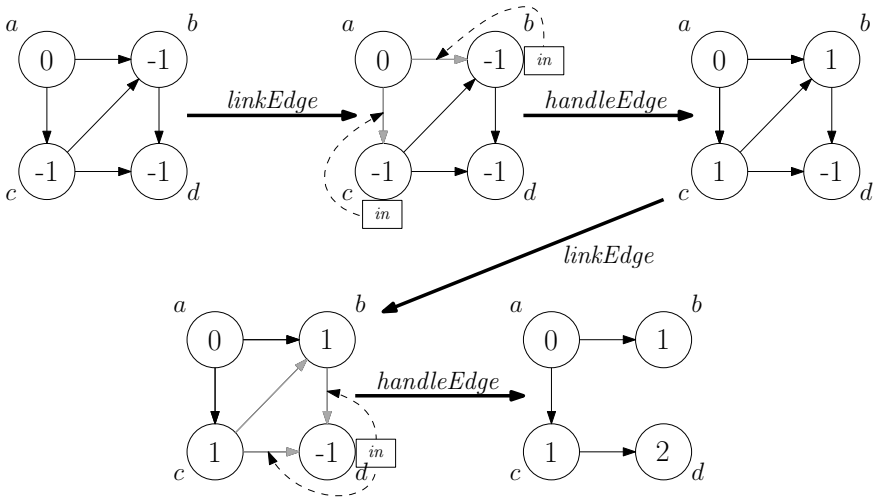

Figure 4: Execution of Listing 5 on a small graph. Every *Edge* newly considered in the current step is grey. Considered *Edges* stay considered, but stable. The dotted arrows denote the possible new values for $t.in$ of a target node t. Note that the *Edge* from c to d wins the race condition to the reference $d.in$.

the same information and *Edges* which lost their nomination will not get past the first conditions of the two steps, causing the data elements to stabilize after all *Nodes* have received a distance from u. Initialization steps should be placed at the start of the shown schedule. An execution of the program on a small graph is shown in Figure 4, where the edges and nodes of the graph are modelled by their respective structs.

6.2. Sorting

A concise example of a AuDaLa program for sorting a linked list of n elements can be found in Listing 6. In it, the elements of the list traverse the list together, during which each element e is looking for its successor in the sorted list. After the traversal, the successor element is saved and the link is updated to the saved element. This reorders the list to the sorted list. The program requires n data elements, one for every list element. It performs $O(n^2)$ work, as every element is compared to every other element, and its span is $O(n)$, as it walks through a list of size n sequentially. Therefore, the program runs in $O(n)$ time. We can achieve a time complexity of $O(\log n)$ by implementing Cole's algorithm [35] in AuDaLa, but that is outside the scope for this paper.

The program defines the struct *ListElem*, modeling the nodes of the list, with parameters *val*, *next*, a reference to the next *ListElem*, *newNext*, a reference to the *ListElem* that should come next in the sorted list, and *comp*, a reference to the current *ListElem newNext* is compared to. The initialization needs to make sure that every element has a distinct value, and that in every element, *comp* is set to the first element of the list and *newNext* is set to *null*.

To facilitate our strategy we give our *ListElem* two steps, one to check an element in the list called *compareElement* and one to reorder the list at the end called *reorder*. With the step *compareElement*, an element checks whether the element to which the *comp* reference leads is a better next element than the current element saved in *newNext* (line 4) and updates *newNext* if that is the case. Afterwards, the *comp* reference is updated to the next element in the list (line 7). With the *reorder* step, an element replaces their old *next* reference with *newNext* (line 11).

To have the program execute our strategy, we call a fixpoint on *compareElement*, such that every element checks all elements in the list. After that is done, the schedule tells the elements to *reorder* (line 15). An execution of the program on a small list is shown in Figure 5.

6.3. Linked List Copy Sort

A more complex example for sorting a linked list of n elements in AuDaLa can be found in Listing 7, which makes a sorted copy of the given list. At the start of execution, this copy consists of a single element,

```
struct ListElem(val: Int, next: ListElem, newNext: ListElem, comp: ListElem){
  compareElement {
    if comp != null then {
      if (comp.val > val && ((newNext = null) || comp.val < newNext.val)) then {
        newNext := comp;
      }
      comp := comp.next;
    }
  }
  reorder {
    next := newNext;
  }
}

Fix(compareElement) < reorder
```

Listing 6: AuDaLa code for sorting

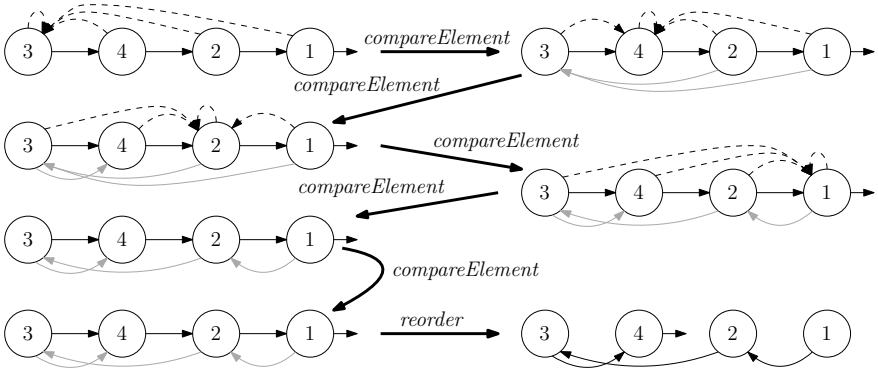

Figure 5: Execution of Listing 6 on a small list. The parameters *next*, *newNext* and *comp* are shown as black, grey and dashed unmarked arrows respectively, and the null-references for *newNext* and *comp* are not shown. The nodes corresponding to *ListElems* contain the value of parameter *val*.

which encompasses the full range of the old list. In every iteration, all elements of the new list split and divide their range (when necessary), and elements from the old list linked to an element of a new list redivide themselves over that element and its new split. This continues until every new element has exactly one old element linked to it. At that point, the new list is a sorted copy of the old list. The idea of the algorithm is to do tree-insert for every data element at the same time, using a tree of size $O(\log m)$, which is dynamically created and repurposed to use as few new data elements as possible. For a list with n data elements with maximum value m, the algorithm performs $O(n \log m)$ work, as all data elements compare themselves to at most $\log m$ other elements. The span of the algorithm is $O(\log m)$, for in the worst case, an element must go through $\log m$ comparisons before it is placed correctly in the new list. It follows that the program runs in $O(\log m)$ time with n data elements with maximum value m. As implemented, the program uses exactly $2n$ data elements. The program defines the struct *OldElem*, which represents an element of the old list, and *NewElem*, which represents an element of the new list. Struct *OldElem* has parameters *val*, which holds its value, *place*, which holds the *NewElem* it is currently linked to, and *move*, which saves whether the old element should move to the right neighbour of its current *place* during a split. Struct *NewElem* has the parameters *min*, *spl* (split) and *max*, which hold the current range of the new element and the value it will split the range on during the next split. Additionally, it has parameters *next*, which holds the next new element in the list, *p1*, which holds a representative old element which is in the first half of its range, *p2*, which holds a representative for the second half of its range, *hasSplit*, which signals to the old elements that

```
struct OldElem(val: Int, place: NewElem, move: Bool){
  checkStable {
    if (place.p2 != null || (val <= place.spl && place.p1 != this)) then {
      place.done := false;
    }
  }
  migrate {
    if (move) then {
      if (place.hasSplit) then {
        place := place.next;
      }
      move := false;
    }
    if (val <= place.spl) then {
      place.p1 := this;
    }
    if (val > place.spl) then {
      place.p2 := this;
      move := true;
    }
  }
}
struct NewElem(min: Int, spl: Int, max: Int, next: NewElem, p1: OldElem, p2: OldElem, hasSplit: Bool, done:
    Bool){
  split {
    if (!done) then {
      if (p1 != null && p2 != null) then {
        next := NewElem(spl+1,spl+1+(max−(spl+1))/2,max,next,null,null,false,true);
        max := spl;
        hasSplit := true;
      }
      if (p1 != null && p2 = null) then {
        max := spl;
        hasSplit := false;
      }
      if (p2 != null && p1 = null) then {
        min := spl + 1;
        hasSplit := false;
      }
      spl := min + (max − min)/2;
      p1 := null;
      p2 := null;
    }
    done := true;
  }
}

migrate < Fix(checkStable < split < migrate)
```

Listing 7: AuDaLa code for Linked List Copy Sort

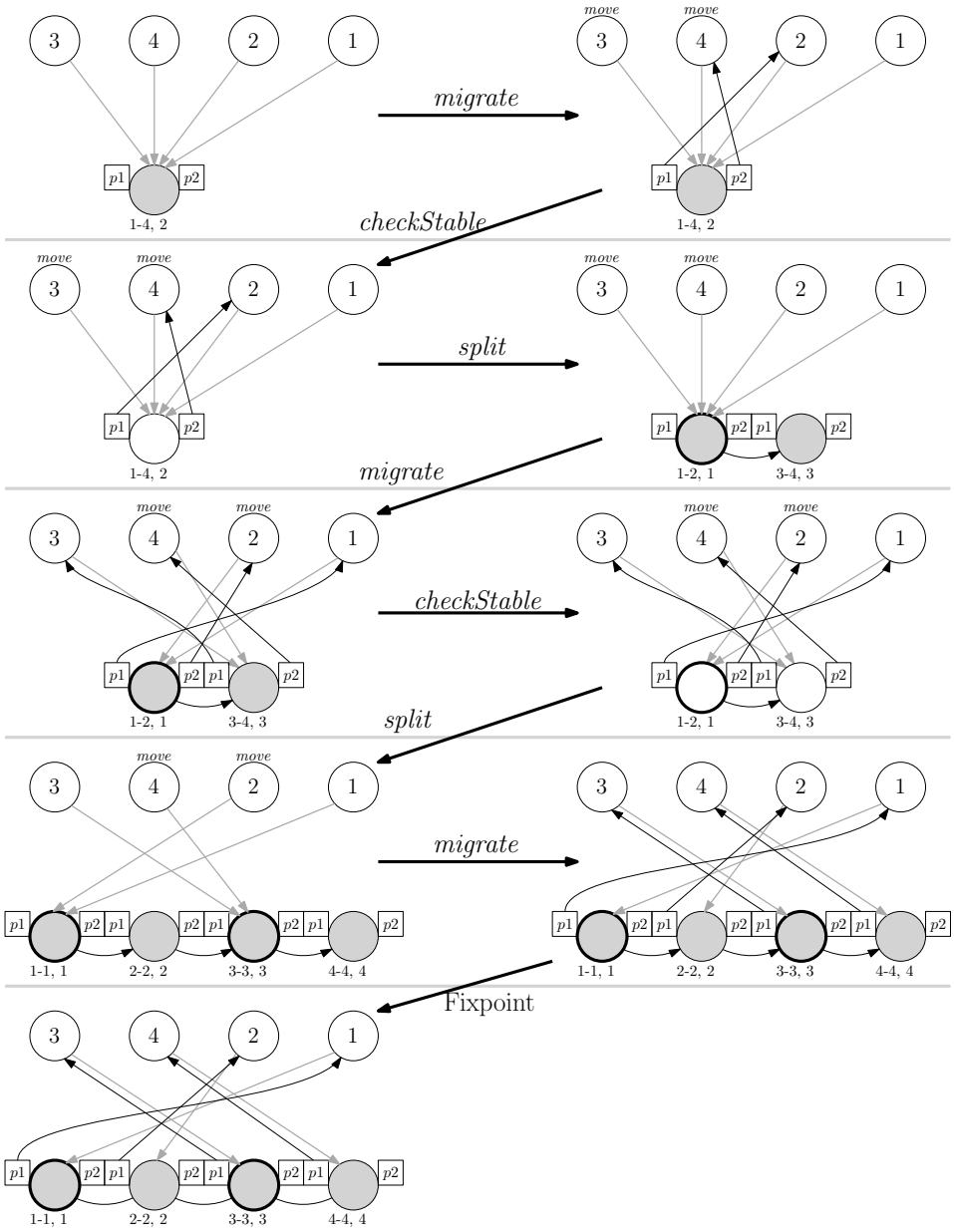

Figure 6: Execution of Listing 7 on a small list. The upper rows are the old elements, the lower rows the new elements. For the old elements, the parameter *place* is depicted by the grey arrows. For the new elements, the numbers beneath every element are formatted "[*min*]-[*max*], [*spl*]". New elements for which *hasSplit* is true have a bold border, and a new element for which *done* is true is grey.

```
struct Elem(val: Int, next: Elem, first: Elem, p1: Elem, p2: Elem, ans: Bool){
  next{
    if !ans then {
      p1 := p1.next;
      p2 := first;
    }
  }
  walk {
    if p1 != null && p2 != null  then {
      if val + p1.val + p2.val = 0) then {
        ans := true;
      }
    }
    if !ans then {
      p2 := p2.next
    }
  }
}
Fix(Fix(walk) < next)
```

Listing 8: AuDaLa implementation for naive 3SUM solution

the new element has split and *done*, which saves whether the element will have to split again.

During initialization, there should be a single new element b to which all old elements of the linked list are linked. For all old elements, *move* should be set to *false*. For b, *min* should be set to the minimum value of the linked list, *max* should be set to the maximum value of the linked list, *spl* should be set to $(max - min + 1)/2 + min$. The parameter *hasSplit* should be *false*, *done* should be *true* and *next*, *p1* and *p2* should be set to *null*. Finding *max* and *min* can be done with a method akin to Prefix Sum.

To facilitate the strategy of splitting the new elements and dividing the old elements after the split, we use three steps. The first step, *checkStable* is used to check whether a new element should split its range further. This is the case if there is still an element on the right side of the range of a new element or if there are multiple elements still on the left side of the range of a new element, which the step encodes. For the first part of the second step, *migrate*, let e be a new element which just split into e and e_2. In the first part of *migrate*, the old elements distribute themselves over e_1 and e_2 by either moving to e_2 or staying at e. In the second part of *migrate* the old elements prepare themselves for the next split and distribute themselves over the left and right sides of the range of their current element. The third step, *split*, regulates the splitting of new elements. Every new element halves the range, but based on the contents of the ranges, they either change their minimum or maximum if only one half is populated, or they make a new element if both halves are populated. They then reset their representatives of both ranges and the *done* parameter.

During execution, the method will execute through a fixpoint $migrate < Fix(checkStable < split < migrate)$. The first call to *migrate* divides the old elements over the range of the new elements. The fixpoint then keeps splitting until for every new element, there is only one old element (which is stored in *p1*). This is helped by the parameter *done*, which only allows for splits if the new element is unstable. The execution of the first iteration of the program on a small list is shown in Figure 6.

6.4. The 3-SUM Problem

As our last example, we give two algorithms for the 3SUM problem [36, 37]: Given a set S of n integers, are there three elements $a, b, c \in S$ s.t. $a + b + c = 0$? Note that these elements do not need to be distinct. The first program, in Listing 8, contains a naive solution, and uses a nested fixpoint. This serves as an example of the use of nested fixpoints. The second program, in Listing 8 solves this in linear time (but with $O(n^2)$ work). This program is based upon the program given by Hoffman [36]. We assume that the set has been given to us as a list. If not, we can convert the set to a list in linear time using a fixpoint in

```
struct Elem(val: Int, next: Elem, prev: Elem, p1: Elem, p2: Elem, ans: Int){
  iterate {
    if ans = 0 then {
      Int cur := val + p1.val + p2.val;
      if (p2 = prev || p1 = next) then {
        ans := −1;
      }
      if cur = 0 then {
        ans = 1;
      }
      if ans = 0 && cur > 0 then {
        p2 := p2.prev;
      }
      if ans = 0 && cur < 0 then {
        p1 := p1.next;
      }
    }
  }
}

Fix(iterate)
```

Listing 9: AuDaLa implementation for a more efficient 3SUM solution

which we nominate an element still in the set and then put it at the back of the list. What's more, our second program assumes that this list is sorted, which can happen through applying the example code from Section 6.2 or 6.3.

Both programs define the struct *Elem*, which represents an element of the list. In the first program, it has a value *val*, a reference to the next element *next*, a reference to the first element of the list *first* and a reference to two other mutable elements, *p1* and *p2*. They also have a parameter *ans*, which holds the answer to the question. After initialization, *val*, *next* and *first* are set as specified above. The parameter *ans* is set to *false* and *p1* and *p2* are set to *first*.

To solve the 3SUM problem, the first program will make every list element a walk through the list in a nested fashion to find elements b and c s.t a, b and c are not the null-*Element*. For every element b chosen by a, a will *walk* through the list to find c, if it exists. If not, it will replace b with its next element in the step *next*. This program performs $O(n^3)$ work (n elements check at most n^2 combinations), with a span of $O(n^2)$: in the worst case, where no triple exists, $O(n)$ elements in the list need to be traversed $O(n)$ times. The code therefore executes in $O(n^2)$ time.

In the second program (Listing 9), *Elem* has the parameters *next* and *prev* which refer to those respective elements in the list. It also has candidate references *p1* and *p2*, initialized as the first and last element in the list respectively, a value *val* and an integer *ans* to give the answer to the problem (initialized to 0).

The second program first computes the current value of the three saved elements. If the current value is 0, a solution is found and *ans* can be set to 1. Otherwise, it will either move *p1* to a node with a larger value or *p2* to a node with a smaller value, depending on the current value. If *p1* or *p2* passes the current node, then either there is a solution with the current node as one of the side nodes or there is no solution with the current node. Either way, the node may stop checking for a solution. This is reflected in *ans* being set to -1. At the end of the program, if there is a solution, then there will be at least one node per solution with *ans* set to 1, which is the middle node of the solution. We have depicted the fixpoint process for a single element in the list in the second program in Figure 7.

This program performs $O(n^2)$ work; every *Elem* does a single (possibly incomplete) walk through the list. The span is $O(n)$, constrained by a walk through the list of an *Elem*. The total running time is then $O(n)$.

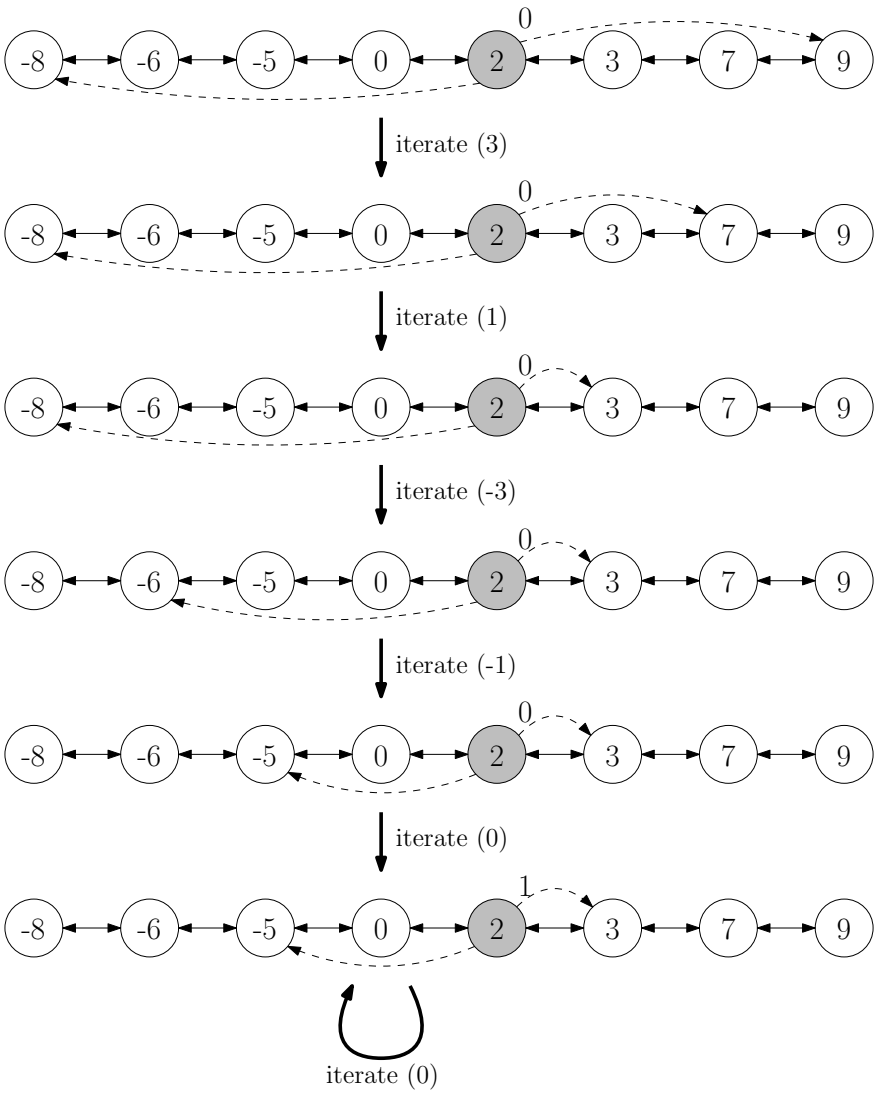

Figure 7: Execution of Listing 9 for a single element on a small list. The element considered is grey. The upper arrow points at *p2*, and the lower arrow points at *p1*. The value *ans* has been displayed above and right of the element, and the computed value cur for an iterate step has been displayed next to the iterate steps.

7. Properties of AuDaLa Programs

This section explores the behaviour of AuDaLa and AuDaLa programs by giving definitions and proving properties for AuDaLa. The goal of this is to get an insight in what generic facts we can establish for this behaviour. We also use the properties established in this section in Section 8 to prove two of our example algorithms correct.

We start out with some standard properties and definitions (Section 7.1). Then, we give a theorem on the properties of well-formed programs, which includes properties on the precise results of the execution of commands (Section 7.2). From these, we can state some corollaries that simplify reasoning about well-formed AuDaLa programs in proofs.

7.1. Standard Properties and Definitions

In this section, we give some standard definitions and properties of AuDaLa. First, to simplify reasoning about semantic values and variables, we define *reference notation* for semantic values, based on the dot notation as used in the syntax:

Definition 7.1 (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.

We then define the notions of *executions* and *accesses*:

Definition 7.2 (AuDaLa Execution). For AuDaLa, an *execution* from P refers to a possibly infinite chain of semantic rule transitions that start in a state P. A *finite execution* refers to a finite chain of semantic rule transitions that start in a state P and ends in a state P_1.

We extend this definition to fit our use of it: As steps are finite by the definition of steps (see Sections 2 to 5), an execution of a step F from P refers to a chain of rule transitions from P starting with an **Init**-transition for F and ending in a state that contains a struct environment σ for which $Done(\sigma)$ holds. A step F can be executed from P iff there exists an execution of F from P.

As expressions and statements are finite, an execution of an expression E, a statement S or a command list C by some struct instance s from P refers to a chain of rule transitions from P starting with the transition enabled by the first command of $[\![E]\!]$, $[\![S]\!]$ or C for s and ending with the transition enabled by the last command of, respectively, $[\![E]\!]$, $[\![S]\!]$ or C for s. Additionally, for every command in $[\![E]\!]$, $[\![S]\!]$ or C, there needs to be a corresponding transition induced by that command taken by s. Note that this allows for interleaving with other statements or expressions; not all transitions need to correspond to a command of $[\![E]\!]$, $[\![S]\!]$ or C in the command list of s. An expression E, a statement S or a command list C can be executed from P by a struct instance s iff there exists an execution of, respectively, E, S or C from P for s.

An execution of a program $\mathcal{P}$ refers to a possibly infinite chain of semantic rule transitions that start in the state $P^0_{\mathcal{P}}$. A finite execution of a program $\mathcal{P}$ starts in $P^0_{\mathcal{P}}$ and ends when no transition is possible. Lastly, we consider *bounded* executions, where an execution of size i from P starts at P and ends at some state P_1 reached after exactly i transitions.

Recall that parameters and local variables are both considered variables. We then define the following:

Definition 7.3 (AuDaLa Semantic Access). Let x be a variable. We define every application of the semantic rules **ComRd** and **ComWr** with respect to x to be an access of x. An application of **ComWrNSkip** for x does not constitute an access.

We use this to introduce some other definitions and lemmas, starting with a notion of determinism. For this, note that we non-deterministically assign new labels to newly created struct instances. As the exact labels we assign does not matter as long as all labels are unique, which is enforced by the semantics, we can ignore the values of newly assigned labels for a notion of determinism. We use the following notions:

Definition 7.4 (AuDaLa Determinism). Let $\mathcal{P}$ be an AuDaLa program containing step F. Let P be some state from which F can be executed. Then F is *deterministic* for P iff there exists exactly one state modulo newly assigned labels which is reached by executing F from P.

Additionally, a variable x is deterministic when executing F from P iff in all states reached by executing F from P the value for x is the same modulo newly assigned labels. Furthermore, F is (fully) deterministic iff for all states P that can execute F, F is deterministic for P, and x is (fully) deterministic if x is deterministic when executing F from all states that can execute F.

From here on, we will leave out the 'modulo newly assigned labels' for brevity; any mention made of something being deterministic from here on out means that it is deterministic modulo newly assigned labels.

Definition 7.5 (Race Condition). Let $\mathcal{P}$ be an AuDaLa program, let F be a step in $\mathcal{P}$ and let P be a state in $\mathcal{P}$ from which F can be executed. Then F executed from P contains a *race condition* for parameter x iff there exist distinct struct instances a and b that both access a parameter v in the same struct instance during an execution of F from P and at least one of the two instances writes to v. If both a and b write to x, this is a *write-write race condition*, otherwise this is a *read-write race condition*. If any step F in the execution of $\mathcal{P}$ has a race condition for some parameter, then $\mathcal{P}$ has a race condition.

Note that in the semantics, writes and reads are atomic. It follows that the value read from x in a read-write race condition must be either the initial value of x or a value written to x and the value contained in x after a write-write race condition must be one of the values written to x.

With the above definitions of race conditions and determinism, we are able to prove the following lemma:

Lemma 7.6 (AuDaLa Determinism)**.** *A step F from an AuDaLa program $\mathcal{P}$ is deterministic for some state P if F does not contain a race condition from P.*

Proof. If F is not deterministic for P, then there are multiple states which can result from executing F from P modulo newly assigned labels. W.l.o.g. we consider two executions e_1 and e_2 of F from P leading to states P_1 and P_2, which are different modulo newly assigned labels. If F can be executed from P, according to the semantics, this means that either **InitG** or **InitL** is available for step F from P. Let P' be the state after calling F. As from P', the schedule is not changed until after F has executed (due to the definition of the **Com** rules in Section 5), P_1 and P_2 must have the same schedule if they both executed F. Additionally, due to the definition of the rule **ComWr** and **ComCons** in Section 5, the stability stack of P_1 and P_2 differ iff the struct environment of P_1 and P_2 differ. We therefore conclude that the difference of P_1 and P_2 is either that (w.l.o.g.) P_1 has a struct instance that P_2 does not have modulo newly assigned labels, or there exists at least one variable that has a different value in P_1 (w.l.o.g.) compared to its value in P_2.

As F has only one definition in $\mathcal{P}$, it follows that any statement reachable in both executions is executed in the same manner by e_1 and e_2, and that any difference between the executions must stem from the values of variables read in the statements. Additionally, for any statement S reachable in e_1 but not in e_2, S must be in one or multiple embedded if-clauses which when executed lead to different results in e_1 and e_2, as there are no other statements possible in s which allow for the skipping of statements. From this, we conclude that any difference between P_1 and P_2 must be caused by values of variables during the execution of statements.

Let x be a variable which induces a difference between P_1 and P_2, and let that difference originate in some statement S, executed by struct instance p. If S is executed in e_1 but not in e_2, then this recursively depends on some other variable/statement pair (x', S'), with S' an if-statement, so we fix S to be a statement executed by both e_1 and e_2. During the execution of S by p, the only part of the statement S that can have multiple possible results after execution are the references to other variables. If we follow the chain of dependency back, it follows that for any local variable on which x is dependent, eventually we reach an initialization statement, which is dependent on earlier initialized local variables or parameters. It follows that the difference between P_1 and P_2 is dependent on parameter values.

As all parameters have only a single value in P, the difference must be dependent on a parameter x changed during F. Then at some point this parameter value must be updated with an update statement

with multiple possible results. As the sequentially consistent semantics do not allow p to reorder its writes or reads in any way (as the **Com** rules of Section 5 exclusively operate on the first command of the command list), these multiple possible results cannot originate from the actions of only a single structure instance. It follows that x must have been accessed by multiple struct instances during execution. As there is no method other than race conditions in the semantics which can make the interaction between these struct instances have multiple possible results, these struct instances must then have a race condition, which means F contains a race condition for x from P. □

We can adjust the focus from a whole step to a single parameter, leading to the following corollary:

Corollary 7.7 (AuDaLa Parameter Determinism)**.** *A parameter x is deterministic during the execution of a step F (of an AuDaLa program $\mathcal{P}$) from a state P iff F does not contain a race condition for x from P.*

Proof. This is true along the same lines as the proof for Lemma 7.6, except that we only consider differences in the value for x in P_1 and P_2 and consider x to be the variable used in the last two alineas of the proof. □

Finding race conditions can be difficult. To this end, we define *potential race conditions*, which can be found on a syntax level. To do so, we first define the following:

Definition 7.8 (Direct and indirect syntax access). Let A be a parameter reference in the syntax, following $\langle Var \rangle$. If A is of the form "$a_1.\ldots.a_n.x$", A is an *indirect* syntax access of x. If A is of the form "x", A is an *direct* syntax access of x. We consider a syntax access to be a *write* access if it occurs in an update or variable assignment statement to the left of the assignment operator.

Note that indirect syntax accesses can only be write syntax accesses in update statements. We use this to define *potential data races*:

Definition 7.9 (Potential Race Condition). Let $\mathcal{P}$ be an AuDaLa program and let F be a step in $\mathcal{P}$. Let a and b be a pair of struct types for which F is defined, with $a = b$ being allowed. Let there be a pair of statements S_1 in a and S_2 in b which include syntax accesses A and B of some parameter x. Let either A or B be indirect and let either A or B be a syntax write access to x. Then the pair (S_1, S_2) is a *potential race condition* on x.

These can be used to consider which parameters can possibly be in a data race:

Lemma 7.10. *Let $\mathcal{P}$ be an AuDaLa program and let F be a step in $\mathcal{P}$. If F does not have any potential race conditions on parameter x, F does not have any race conditions on parameter x.*

Proof. If F does not have any potential race conditions, there are no pairs of statements (S_1, S_2) that can be taken from any executions of F in any structs s.t. S_1 and S_2 are executed by different struct instances but access x in the same struct instance of which at least one access is a write access. Then it follows that F cannot have any race conditions on x. □

We also define a notion of *instability*:

Definition 7.11 (Instability). Let $\mathcal{P}$ be an AuDaLa program a step F. Let P be a state of $\mathcal{P}$ from which F can be executed, and let p be a struct instance in P. A parameter $p.x$ with value v before the execution of F is *unstable* after the execution of F from P, denoted as $p.x^*$, iff a write command is executed during t that sets $p.x$ to a value $a \neq v$.

In this paper, we are interested in proving programs that may contain write-write race conditions but do not contain read-write race conditions. Whether a race condition shows up during execution is, however, dependent on the initial state of an execution. As we discuss partial programs in this paper, which may start from states other than the initial state as defined in AuDaLa semantics, this should be made explicit:

Definition 7.12 (RW Race Condition Free from Premise State). We consider an AuDaLa program $\mathcal{P}$ to be without read-write race conditions from some state P_1 iff any execution of $\mathcal{P}$ starting in P_1 does not contain a read-write race condition. In the context of $\mathcal{P}$ being read-write race condition free from P_1, we call P_1 the *premise state*.

When our lemmas and theorems assume that a program is without read-write race conditions from some premise state, this is explicitly mentioned.

7.2. Properties of Well-Formed Programs

In this section, we analyze the properties of well-formed programs. We first introduce the notion of a well-formed AuDaLa state:

Definition 7.13 (Well-Formed AuDaLa State). Let $\mathcal{P}$ be any well-formed AuDaLa program and let P be an AuDaLa state reachable by some execution of $\mathcal{P}$. Then P is a well-formed AuDaLa state of $\mathcal{P}$.

The goal of this section is to establish that the properties in the following theorem hold for well-formed AuDaLa state. Note that a schedule Sc is well-formed modulo $aFix$ if replacing all occurrences of $aFix$ in Sc with Fix yields a well-formed schedule Sc. We denote well-formedness modulo $aFix$ as $\Gamma, \Omega \underset{aF}{\vdash} Sc$ for some environments Γ and Ω and schedule Sc.

Let $sem : \mathcal{T} \times \mathcal{P} \to \mathbb{T} \cup \{\bot\}$ be defined as:

$$
\begin{aligned}
&sem(\texttt{Nat}, \mathcal{P}) = \mathbb{N},\\
&sem(\texttt{Int}, \mathcal{P}) = \mathbb{Z},\\
&sem(\texttt{Bool}, \mathcal{P}) = \mathbb{B},\\
&sem(\texttt{String}, \mathcal{P}) = \mathit{String},\\
&sem(\theta, \mathcal{P}) = \theta \text{ if } \theta \in \Theta_{\mathcal{P}}\\
&sem(\theta, \mathcal{P}) = \bot \text{ if } \theta \notin \Theta_{\mathcal{P}}.
\end{aligned}
$$

Let a *schedule transition* be a transition induced by the transition rules **InitG**, **InitL**, **FixInit**, **FixIter** and **FixTerm**, and let a *command transition* be a transition induced by the other transition rules (the rules with prefix **Com**).

Theorem 7.14 (Properties of Well-Formed AuDaLa States)**.** *Let $P = \langle Sc, \sigma, s\chi\rangle$ be an AuDaLa state reached by some well-formed program $\mathcal{P} = D\ Sc_{\mathcal{P}}$ s.t. $\Gamma, \Omega \vdash D\ Sc_{\mathcal{P}}$ for some Γ, Ω. Then all of the following hold:*

1. *Either $Sc = \varepsilon$ or $\Gamma; Steps(D), \Omega \underset{aF}{\vdash} Sc$.*
2. *If $Done(\sigma)$ holds and $Sc \neq \varepsilon$, there exists at least one schedule transition that can be taken from P with definitions D.*
3. *If $Done(\sigma)$ holds and $Sc \neq \varepsilon$, then after a finite amount of transitions from P either the schedule is empty or there must be an **InitG** or **InitL** transition to some state $P' = \langle Sc', \sigma', s\chi'\rangle$ for which $Done(\sigma')$ does not hold.*
4. *If $\neg Done(\sigma)$, let $\ell \in \mathcal{L}$ s.t. $\sigma(\ell) = \langle \theta, \gamma, \chi, \xi\rangle$ and $\gamma \neq \varepsilon$. Let F be the step currently executed by $\sigma(\ell)$. Let $S_i; \ldots; S_n$ be the statements s.t. $[\![S_i]\!] = c_1; \cdots; c_m$ and $\gamma = c_j; \cdots; c_m; [\![S_{i+1}; \cdots; S_n]\!]$. Let $S_1; \ldots; S_{i-1}$ be the statements that have been executed by $\sigma(\ell)$ during the current execution of F. Then there exists a c_k with $k \leq j$ s.t. either $c_k; \ldots; c_j = [\![E]\!]$ for some expression E or $c_k; \ldots; c_j = [\![S_i]\!]$.*
5. *If $\neg Done(\sigma)$, let $\ell \in \mathcal{L}$ s.t. $\sigma(\ell) = \langle \theta, \gamma, \chi, \xi\rangle$ and $\gamma \neq \varepsilon$. Let F be the step currently executed by $\sigma(\ell)$. Let $S_i; \ldots; S_n$ be the statements s.t. $[\![S_i]\!] = c_1; \cdots; c_m$ and $\gamma = c_j; \cdots; c_m; [\![S_{i+1}; \cdots; S_n]\!]$. Let $S_1; \ldots; S_{i-1}$ be the statements that have been executed by $\sigma(\ell)$ during the current execution of F. Then, with c_k with $k \leq j$ s.t. $c_k; \ldots; c_j = [\![E]\!]$ or $c_k; \ldots; c_j = [\![S_i]\!]$, let $P_k = \langle Sc_k, \sigma_k, s\chi_k\rangle$ be the state right before the transition induced by c_k and let $\sigma_k(\ell) = \langle \theta, \gamma_k, \chi_k; \xi_k\rangle$. Then one of the following holds:*

(a) $c_j = \mathbf{push}(v)$ *for some* $v \in \mathcal{V}$ *or* $c_j = \mathbf{push}(\mathbf{this})$,

(b) $c_j = \mathbf{rd}(x)$ *and all of the following hold:*

i. $\chi = \chi_k; \ell'$ *and* $\sigma(\ell') = \langle \theta_{\ell'}, \gamma_{\ell'}, \chi_{\ell'}, \xi_{\ell'} \rangle$ *for some* $\theta_{\ell'}, \gamma_{\ell'}, \chi_{\ell'}, \xi_{\ell'}$ *and* $(x \notin Pars(\theta_{\ell'}, \mathcal{P}) \Rightarrow \ell' = \ell)$,

ii. *either* $S_q = (T\ x := E)$ *for some* $1 \leq q < i$, $T \in \mathcal{T}$ *and* $E \in EX$ *or* $(x : T) \in Pars(\theta_{\ell'}, \mathcal{P})$ *for some* $T \in \mathcal{T}$,

iii. *either* $T \in \Theta_{\mathcal{P}}$ *and* $\xi(x) \in \mathcal{L} \wedge \sigma(\xi_{\ell'}(x)) = \langle T, \gamma_x, \chi_x, \xi_x \rangle$ *(for some* γ_x, χ_x *and* ξ_x*) or* $T \in \mathcal{T} \setminus ID$ *and* $\xi'(x) \in sem(T)$.

(c) $c_j = \mathbf{wr}(x)$ *and all of the following hold:*

i. $\chi = \chi_k; v; \ell' \wedge \sigma(\ell') = \langle \theta_{\ell'}, \gamma_{\ell'}, \chi_{\ell'}, \xi_{\ell'} \rangle$ *for some* $\theta_{\ell'}, \gamma_{\ell'}, \chi_{\ell'}, \xi_{\ell'}$ *and* $(x \notin Pars(\theta, \mathcal{P}) \Rightarrow \ell' = \ell)$,

ii. *either* $S_q = (T\ x := E)$ *for some* $1 \leq q \leq i$, $T \in \mathcal{T}$ *and* $E \in EX$ *or* $(x : T) \in Pars(\theta', \mathcal{P})$ *for some* $T \in \mathcal{T}$,

iii. *either* $T \in \Theta_{\mathcal{P}}$ *and* $v \in \mathcal{L} \wedge \sigma(v) = \langle T, \gamma'', \chi'', \xi'' \rangle$ *(for some* γ'', χ'' *and* ξ''*) or* $T \in \mathcal{T} \setminus ID$ *and* $v \in sem(T)$.

(d) $c_j = \mathbf{cons}(\theta)$, *with* $\theta \in \Theta_{\mathcal{P}} \wedge Pars(\theta, \mathcal{P}) = p_1 : T_1; \cdots ; p_c : T_c$ *for some* c, *and* $\chi = \chi_k; v_1; \cdots ; v_c$ *and for all* $p_h : T_h$, *either* $T_h \in \{\texttt{Nat}, \texttt{Int}, \texttt{Bool}, \texttt{String}\}$ *and* $v_h \in sem(T_h)$ *or* $T_h \in \Theta_{\mathcal{P}}$ *and* $v_h \in \mathcal{L} \wedge \sigma(v_h) = \langle T_h, \gamma'', \chi'', \xi'' \rangle$.

(e) $c_j = \mathbf{if}(C)$ *and* $\chi = \chi_k; b$ *s.t.* $b \in \mathbb{B}$.

(f) $c_j = \mathbf{not}$ *and* $\chi = \chi_k; b$ *s.t.* $b \in \mathbb{B}$.

(g) $c_j = \mathbf{op}(\circ)$ *and* $\chi = \chi_k; a; b$ *and all of the following hold:*

i. *If* $\circ \in \{=, !=\}$ *then* $a, b \in \{\mathbb{N}, \mathbb{Z}\}$ *or there exists a* $T \in \{\mathbb{N}, \mathbb{Z}, \mathbb{B}, String\} \cup \Theta_{\mathcal{P}}$ *s.t. either* $T \notin \Theta_{\mathcal{P}}$ *and* $a, b \in T$ *or* $T \in \Theta_{\mathcal{P}}$ *and* $\sigma(a) = \langle T, \gamma_a, \chi_a, \xi_a \rangle$ *and* $\sigma(b) = \langle T, \gamma_b, \chi_b, \xi_b \rangle$,

ii. *If* $\circ \in \{<=, >=, <, >, =, !=, *, /, \%, +, \hat{}, -\}$ *then* $a, b \in \{\mathbb{N}, \mathbb{Z}\}$,

iii. *If* $\circ \in \{\&\&, ||\}$ *then* $a, b \in \mathbb{B}$.

Additionally, there exists a transition induced by c_j *from* P *to a state* $P' = \langle Sc', \sigma', s\chi' \rangle$ *s.t.* $\sigma'(\ell) = \langle \theta, \gamma', \chi', \xi' \rangle$ *and all of the following hold:*

(a) *If* $c_j = \mathbf{push}(v)$ *for some* $v \in \mathcal{V}$, *then* $\chi' = \chi_k; v$ *and* $\gamma' = c_{j+1}; \ldots ; c_m; [\![S_{i+1}; \cdots ; S_n]\!]$.

(b) *If* $c_j = \mathbf{push}(\mathbf{this})$, *then* $\chi' = \chi_k; \ell$ *and* $\gamma' = c_{j+1}; \ldots ; c_m; [\![S_{i+1}; \cdots ; S_n]\!]$.

(c) *If* $c_j = \mathbf{rd}(x)$, *with* $\chi = \chi_k; \ell'$ *and* $\sigma(\ell') = \langle \theta_{\ell'}, \gamma_{\ell'}, \chi_{\ell'}, \xi_{\ell'} \rangle$ *for some* $\theta_{\ell'}, \gamma_{\ell'}, \chi_{\ell'}, \xi_{\ell'}$, *then* $\chi' = \chi_k; \xi'(x)$ *and* $\gamma' = c_{j+1}; \ldots ; c_m; [\![S_{i+1}; \cdots ; S_n]\!]$.

(d) *If* $c_j = \mathbf{wr}(x)$, *then, with* $\chi = \chi_k; v; \ell'$, $\sigma'(\ell') = \langle \theta_{\ell'}, \gamma'_{\ell'}, \chi'_{\ell'}, \xi'_{\ell'} \rangle$ *s.t. if* $\ell' \notin \mathcal{L}_0$, $\xi'_{\ell'} = \xi_{\ell'}[x \mapsto v]$ *and* $x \notin Pars(\theta_{\ell'}, \mathcal{P}) \vee \xi_{\ell'}(x) \neq v$ *then* $s\chi' = false^{|s\chi|}$, *and if* $\ell' \in \mathcal{L}_0$, $\xi'_{\ell'} = \xi_{\ell'}$ *and* $s\chi' = s\chi$. *Additionally,* $\chi' = \chi_k$, $\gamma' = [\![S_{i+1}; \cdots ; S_n]\!]$.

(e) *If* $c_j = \mathbf{cons}(\theta)$, *then, with* $\chi = \chi_k; v_1; \cdots ; v_c$ *and* $Pars(\theta, \mathcal{P}) = p_1 : T_n; ...; p_c : T_c$, $\chi' = \chi_k; \ell'$ *for some* ℓ' *s.t.* $\sigma(\ell') = \bot$, $\sigma'(\ell') = \langle \theta, \varepsilon, \varepsilon, \xi^0_\theta[\{p_1 \mapsto v_1, \ldots , p_c \mapsto v_c\}] \rangle$, $\gamma' = c_{j+1}; \ldots ; c_m; [\![S_{i+1}; \cdots ; S_n]\!]$ *and* $s\chi' = false^{|s\chi|}$.

(f) *If* $c_j = \mathbf{if}(C)$, *with* $\chi = \chi_k; b$, $\chi' = \chi_k$ *and if* $b = true$, $\gamma' = C; [\![S_{i+1}; \cdots ; S_n]\!]$ *and if* $b = false$, $\gamma' = [\![S_{i+1}; \cdots ; S_n]\!]$.

(g) *If* $c_j = \mathbf{not}$, *with* $\chi = \chi_k; b$, $\chi' = \chi_k; \neg b$ *and* $\gamma' = c_{j+1}; \ldots ; c_m; [\![S_{i+1}; \cdots ; S_n]\!]$.

(h) *If* $c_j = \mathbf{op}(\circ)$, *with* $\chi = \chi_k; a; b$, *then* $\gamma' = c_{j+1}; \ldots ; c_m; [\![S_{i+1}; \cdots ; S_n]\!]$ *and, with* o *the semantic equivalent of* $\circ$, $\chi' = \chi_k; c$ *for some* c *s.t. all of the following hold:*

i. *If* $a \in T$ *and* $b \in T$ *with* $T \in \mathbb{T}$ *and* $\circ = \{=, !=\}$, *then* $c = a\, o\, b$ *and* $c \in \mathbb{B}$,

ii. *If* $a \in T_1$ *and* $b \in T_2$, *with* $T_1, T_2 \in \{\mathbb{N}, \mathbb{Z}\}$ *and* $\circ = \{<=, >=, <, >, =, !=\}$, *then* $c = a\, o\, b$ *and* $c \in \mathbb{B}$,

iii. *If* $a \in T_1$ *and* $b \in T_2$, *with* $T_1, T_2 \in \{\mathbb{N}, \mathbb{Z}\}$ *and* $\circ = \{*, /, \%, +, \hat{}, -\}$, *then* $c = a \circ b$ *and* $c \in \mathbb{Z}$,

iv. *If* $a, b \in \mathbb{N}$ *and* $\circ = \{*, /, \%, +, \hat{}\}$, *then* $c = a \circ b$ *and* $c \in \mathbb{N}$.

v. *If* $a, b \in \mathbb{B}$ *and* $\circ \in \{\&\&, ||\}$, *then* $c = a \circ b$ *and* $c \in \mathbb{B}$.

6. *If* $Done(\sigma)$ *does not hold in* P, *then after a finite amount of transitions there must be a transition to some state* $P' = \langle Sc', \sigma', s\chi' \rangle$ *for which* $Done(\sigma')$ *holds.*

We prove Properties 1, 2, 3, 4, 5 and 6 separately. Every proof is dependent on different parts of the semantics of $\mathcal{P}$ as defined in Section 5. All parts of the semantics of $\mathcal{P}$ are used in at least one of the proofs.

The first three properties are about the schedule, and state that the schedule is well-formed, a transition induced by the schedule can always be taken if there are no steps to be executed and the schedule is not empty, and that there are a finite amount of schedule transitions between steps or until the schedule is empty, respectively.

Proof of Property 1. We prove Property 1 by induction on the states traversed in any execution of $\mathcal{P}$. Note that the schedule of $P^0_{\mathcal{P}}$ is well-formed, as it is the schedule of $\mathcal{P}$, and $\mathcal{P}$ is well-formed. This proves the base case.

Then, let $P_i = \langle Sc, \sigma, s\chi \rangle$ be the ith state traversed in some execution of $\mathcal{P}$ with definitions D, from which some transition is taken to a state $P_{i+1} = \langle Sc', \sigma', s\chi' \rangle$. By induction hypothesis, we assume that $\Gamma; Steps(D), \Omega \underset{aF}{\vdash} Sc$ or that $Sc = \varepsilon$. Then, we do a case distinction:

- If the transition taken is a command transition, $Sc' = Sc$ so $\Gamma; Steps(D), \Omega \underset{aF}{\vdash} Sc'$ or $Sc' = \varepsilon$.
- If the transition is taken is a schedule transition, we know that $Sc \neq \varepsilon$, as every schedule transition requires something in the schedule.

  As Sc is well-formed, we know that it has a well-defined first element, s.t. $Sc = sc$ or $Sc = sc < Sched_1$. Then, if $Sc = sc$, we know that $\Gamma, \Omega \underset{aF}{\vdash} sc$, as Sc is well-formed modulo $aFix$. Then, we have a case distinction on the form of sc:

  1. If $sc = F$ for some step F or $sc = \theta.F$ for some struct type θ and some step F, then the schedule transition that is taken is *InitG* or *InitL* respectively. These will remove sc from the schedule.
  2. If $sc = Fix(sc_1)$ for some schedule sc_1, the transition must be *FixInit*, resulting in $sc' = sc_1 < aFix(sc_1)$ which replaces sc. As $\Gamma; Steps(D), \Omega \vdash Fix(sc_1)$, it holds that $\Gamma; Steps(D), \Omega \underset{aF}{\vdash} sc_1$. It follows that $\Gamma; Steps(D), \Omega \underset{aF}{\vdash} sc_1 < aFix(sc_1)$.
  3. If $sc = aFix(sc_1)$ for some schedule sc_1, the transition must be either *FixIter* or *FixTerm*. If the transition is *FixIter*, then $sc' = sc_1 < aFix(sc)$ replaces sc. As $\Gamma; Steps(D), \Omega \vdash aFix(sc_1)$, $\Gamma; Steps(D), \Omega \underset{aF}{\vdash} sc_1$. It follows that $\Gamma; Steps(D), \Omega \underset{aF}{\vdash} sc_1 < aFix(sc_1)$. If the transition is **FixTerm**, sc is removed from the schedule.

  From this, we can conclude that either $Sc' = \varepsilon$ or $\Gamma; Steps(D), \Omega \underset{aF}{\vdash} Sc'$, so the induction step holds.

  Then if $Sc = sc < Sc_1$, with sc being a single schedule element, note that $\Gamma; Steps(D), \Omega \underset{aF}{\vdash} Sc$ iff $\Gamma; Steps(D), \Omega \vdash Sc_1$ and $\Gamma; Steps(D), \Omega \underset{aF}{\vdash} sc$. Any schedule transition only affects the first schedule element, so after a single transition, Sc_1 will remain unchanged. This means that we can consider sc as a separate schedule. We then apply same case distinction as above, from which we can conclude that either the first element is removed after a transition or the transition results in a schedule Sc'' s.t. $\Gamma; Steps(D), \Omega \underset{aF}{\vdash} Sc''$. In the first case, $Sc' = Sc_1$ and $\Gamma, \Omega \underset{aF}{\vdash} Sc'$. In the second case, we know that $\Gamma; Steps(D), \Omega \underset{aF}{\vdash} Sc''$ and $\Gamma; Steps(D), \Omega \underset{aF}{\vdash} Sc_1$, so we know that $\Gamma; Steps(D), \Omega \underset{aF}{\vdash} Sc'$. Therefore, in this case, the induction step holds.

It follows that the induction step holds, so Property 1 also holds. □

Proof of Property 2. Let $P = \langle Sc, \sigma, s\chi \rangle$ be a state reached in an execution of $\mathcal{P}$ s.t. $Done(\sigma) = true$. Assume that $Sc \neq \varepsilon$. It follows from Property 1 that Sc is well-formed modulo $aFix$, so we know that Sc has a well-formed and well-defined first schedule element sc. Then, we do a case distinction on what this element is:

1. If $sc = F$ for some step F, then it follows that **InitG** is enabled: $Done(\sigma)$ holds, and beyond that, the only requirement is that $sc = F$. Same holds for $sc = \theta.F$ and **InitL**.

2. If $sc = Fix(sc_1)$ for some schedule sc_1, the transition **FixInit** can be taken, as $Done(\sigma)$ holds and sc fits the pattern in the schedule which is the other requirement.

3. If $sc = aFix(sc_1)$ for some schedule sc_1, then whether a transition can be taken depends on whether there is a value on top of the stability stack. This is, however, the case: The only rule which introduces a pattern which includes $aFix$ in the semantics is the rule **FixInit**, which also adds a value on top of the stability stack. The only rule which removes a value of from the stability stack is **FixTerm**, which also removes an $aFix$ pattern. It follows that the amount of values in the stability stack corresponds exactly to the amount of $aFix$ patterns. It follows that there is a value on the stability stack when a $aFix$ pattern is encountered. Then, either this value is *true* or *false*. If the value is *true*, note that $Done(\sigma)$ holds and that an $aFix$ pattern is the topmost schedule element, so **FixTerm** is enabled. Along the same lines, if the value is *false*, **FixIter** is enabled.

It follows that if the schedule is not empty and $Done(\sigma)$ holds, there must be a well-formed first schedule element and for all forms that element can take there is an enabled transition. □

Proof of Property 3. We prove Property 3 by first proving through induction that after any sequence of transitions from P either the schedule is empty or there must be an **InitG** or **InitL** transition. We then prove that after this **InitG** or **InitL** transition to, w.l.o.g., state P', $Done(\sigma')$ does not hold. Note then that if either **InitG** or **InitL** are enabled, by definition of the schedule and the transitions, no other transitions are enabled, so the **InitL** or **InitG** transition must also be taken. We therefore make no distinction between the an **InitG** or **InitL** transition being enabled and being taken.

We first do induction on the structure of the schedule. As base case, we take that $Sc = \varepsilon$, which directly satisfies our property. We then assume as induction hypothesis 1 that for all but the first element of the schedule, any sequence of schedule transitions eventually leads to either the schedule being empty or an **InitL** or **InitG** transition being taken.

Let sc be the first element of Sc, and assume that **InitG** and **InitL** are not enabled. It follows that $sc = aFix(sc_1)$ or $sc = Fix(sc_1)$. Let the next transition taken be t, from P to $P' = Sc', \sigma', s\chi'$. This transition exists, by Property 2. We then do induction on the amount of occurrences k of both Fix and $aFix$. If $k = 1$, It follows that sc_1 is either a step call, a typed step call or a sequence of step and typed step calls, as Sc is well-formed modulo $aFix$. Then if $t =$ **FixInit**, (so $sc = Fix(sc_1)$), Sc' will start with the sequence $sc_1 < aFix(sc_1)$, and as sc_1 must have a step call as first element, either **InitL** or **InitG** must be enabled. The same argument holds if $t =$ **FixIter**, but with $sc = aFix(sc_1)$. If $t =$ **FixTerm**, so $sc = aFix(sc_1)$, sc is removed. Then either $Sc' = \varepsilon$, or we use induction hypothesis 1 to conclude that any sequence of schedule transitions eventually leads to either the schedule being empty or an **InitL** or **InitG** transition being taken.

We then assume as induction hypothesis 2 that if $k = i-1$, any sequence of schedule transitions eventually leads to either the schedule being empty or an **InitL** or **InitG** transition being taken. Let $k = i$. Then if $t =$ **FixInit**, (so $sc = Fix(sc_1)$), Sc' will start with the sequence $sc_1 < aFix(sc_1)$, and as the amount of occurrences of both Fix and $aFix$ in sc_1 must then be $i - 1$, by induction hypothesis 2, any sequence of schedule transitions eventually leads to either the schedule being empty or an **InitL** or **InitG** transition being taken. The same argument holds if $t =$ **FixIter**, but with $sc = aFix(sc_1)$. If $t =$ **FixTerm**, so $sc = aFix(sc_1)$, sc is removed. Then again either $Sc' = \varepsilon$, or we use induction hypothesis 1 to conclude

that any sequence of schedule transitions eventually leads to either the schedule being empty or an **InitL** or **InitG** transition being taken.

It follows that after a finite amount of transitions from P either the schedule is empty or there must be an **InitG** or **InitL** transition to some state $P' = \langle Sc', \sigma', s\chi' \rangle$. By the definition of the schedule transitions, when w.l.o.g. transition **InitG** is taken for w.l.o.g. some step F, by definition of **InitG**, the commands of F are put into the command list of every struct instance of a struct type for which F is defined. As Sc is well-formed modulo $aFix$, there is at least one struct type for which F is defined, and as there always exists a null instance for every struct type, there is then at least one struct instance which will have commands in their command list. It follows that $\neg Done(\sigma')$. □

We now prove the more involved properties involving expression and statement execution. We first prove Property 4, which states that for any command c_j found inside the commands generated for a statement S_i, there exists a command c_k which signifies the start of the smallest subexpression of S_i of which the generated commands include the command c_j.

Proof of Property 4. To prove the lemma, we do a case distinction on which command c_j is and choose an appropriate c_k:

1. If $c_j = \mathbf{push}(val(g))$, it has been generated by the interpretation function in response to either the expression g, with $g \in LT$, or, if $g = defaultVal(T)$ for some T, to $null_T$. It follows that with $c_k = c_j$, the lemma holds. Analogously, if $c_j = \mathbf{push}(\mathbf{this})$, which is generated for the expression `this` or for a (sub)expression $x_1.\ldots.x_n$ for some $n \geq 0$, then we can also choose $c_k = c_j$.

2. If $c_j = \mathbf{rd}(x_i)$ for some variable x_i, then we know that c_j is generated by the interpretation function in response to some variable expression $x_1.\cdots.x_n$, with $x_i \in x_1.\cdots.x_n$. Let c_k be the first command of $[\![x_1.\cdots.x_n]\!]$. Then $c_k;\ldots;c_j = [\![x_1.\cdots.x_i]\!]$ and the lemma holds.

3. If c_j is a **wr**, **cons**, **if**, **not** or **op** command, it is automatically the last generated command for some expression or statement as per the definition of the interpretation function. The lemma will then hold for c_k being the first command generated for the same expression or statement.

As in all cases, a c_k can be chosen, the lemma holds. □

To prove Property 5, we first prove an auxiliary lemma, which proves that if we encounter a label, we know it has a defined struct instance. This lemma depends on the fact that writes are atomic in our semantics and that struct instances cannot be deleted.

Lemma 7.15. *Let $\mathcal{P}$ be an AuDaLa program. Let $P = \langle Sc, \sigma, s\chi \rangle$ be a state encountered during some execution of $\mathcal{P}$ s.t. there exists a label ℓ s.t. $\sigma(\ell) = \langle \theta, \gamma, \chi; \ell', \xi \rangle$ for some θ, γ, χ and ξ. Then $\sigma(\ell') \neq \bot$.*

Proof. We use the definitions of the lemma. If $\ell' \in \mathcal{L}_0$, then the lemma holds by the definition of $\mathcal{L}_0$ and null instances, as instances cannot be deleted in our semantics. Therefore, assume $\ell' \notin \mathcal{L}_0$. Then ℓ' must have been introduced to the pool of labels after the initial state, so ℓ' must have been introduced by some semantic transition. The only transition which can introduce new labels is **ComCons**, which also makes sure that there is a struct instance for the new label. It follows that in both cases, the lemma holds. □

We then prove Property 5 using induction. Intuitively, Property 5 states that when a command is encountered, the stack holds are prerequisite elements to be able to execute the command according to the transition induced by the command, and specifies the results of executing this command transition from the current state, including the types of those results where applicable. We prove this using induction on the distance d, which signifies the length of the sequence $c_k;\ldots;c_j$, with c_k being the first command of the smallest subexpression or statement of which the commands generated for a statement S_i contain c_j. If $d = 0$, then the command must be able to execute without context, and its results must be able to be established without the context. If $d > 0$, we apply the induction hypothesis that Property 5 holds for all command pairs c'_j, c'_k s.t. their distance is $d - 1$, and then prove that Property 5 holds for c_j as well.

Proof of Property 5. Let $P = \langle Sc, \sigma, s\chi\rangle$ be a well-formed AuDaLa state reached by some program $\mathcal{P} = D\ Sc_{\mathcal{P}}$ s.t. $\Gamma, \Omega \vdash D\ Sc_{\mathcal{P}}$ for some Γ, Ω. Then assume that $\neg Done(\sigma)$. Let $\ell \in \mathcal{L}$ be a label s.t. $\sigma(\ell) = \langle \theta, \gamma, \chi, \xi\rangle$ and $\gamma \neq \varepsilon$. Let F be the step currently executed by $\sigma(\ell)$, with statements $S_1; \cdots ; S_n$. Let S_i be the statement s.t. $[\![S_i]\!] = c_1; \cdots ; c_m$ and $\gamma = c_j; \cdots ; c_m; [\![S_{i+1}; \cdots ; S_n]\!]$. Then, let c_k with $k \leq j$ s.t. $c_k; \ldots ; c_j = [\![E]\!]$ or $c_k; \ldots ; c_j = [\![S_i]\!]$. Note that according to Property 4, this c_k is guaranteed to exist, so we assume c_k has the largest k possible for c_j. Let $P_k = \langle Sc_k, \sigma_k, s\chi_k\rangle$ be the state right before the transition induced by c_k and let $\sigma_k(\ell) = \langle \theta, \gamma_k, \chi_k; \xi_k\rangle$.

To prove Property 5, we do strong induction over the distance d between c_k and c_j, with $d = j - k$:

$d = 0$ Then it follows that $c_j = c_k$. This means that c_j is both the first and the last command of some expression or statement. By the definition of the interpretation function, this can only happen for the expressions g with $g \in LT$, **this**, $null_T$ and $x_1.\cdots .x_0$. In all of those cases, $c_j = \mathbf{push}(v)$ for some value $v \in \mathcal{V}$ or $c_j = \mathbf{push}(this)$. This satisfies the first part of Property 5. Then, if $c_j = \mathbf{push}(v)$ for some $v \in \mathcal{V}$, a transition **ComPush** will be enabled to a state $P' = \langle Sc', \sigma', s\chi'\rangle$ with $\sigma'(\ell) = \langle \theta, c_{j+1}; \ldots ; c_m; [\![S_{i+1}; \cdots ; S_n]\!]; \gamma', \chi; v, \xi'\rangle$. As $\chi = \chi_k$, this satisfies the property. If $c_j = \mathbf{push}(\mathbf{this})$, then a transition **ComPushThis** will be enabled to a state $P' = \langle Sc', \sigma', s\chi'\rangle$ with $\sigma'(\ell) = \langle \theta, c_{j+1}; \ldots ; c_m; [\![S_{i+1}; \cdots ; S_n]\!], \chi; \ell, \xi'\rangle$. As $\chi = \chi_k$, this satisfies the property.

$d > 0$ As induction hypothesis, we assume that the property holds for any command c'_j with a corresponding command c'_k s.t. the distance d' for c'_j and c'_k is smaller than d. As c_k has the largest possible k s.t. $c_k; \ldots ; c_j = [\![E]\!]$ or $c_k; \ldots ; c_j = [\![S_i]\!]$ and $d > 0$, it follows that c_j is not a **push** command; if c_j was a push command, it must have been introduced as the first command of an expression g with $g \in LT$, **this**, $null_T$ or $x_1.\cdots .x_n$ (by the definition of the interpretation function), so either $d = 0$ or there exists a larger k s.t. $c_k; \ldots ; c_j$ is the list of commands generated from an expression. We then do a case distinction on the commands c_j can be:

rd If $c_j = \mathbf{rd}(x)$ for some x, then $c_k = \mathbf{push}(\mathbf{this})$ and the expression E is a variable expression $x_1.\cdots .x$, as seen in the proof for Property 4. As $\mathbf{rd}(x)$ was generated by the interpretation function, it follows that there exists a variable x_h with $h \geq 0$ s.t. $E = x_1.\cdots .x_h.x$. It follows that $E = E'.x$ for $E' = x_1.\cdots .x_h$, and that we can apply the induction hypothesis on command c_{j-1}, as $c_k; \ldots ; c_{j-1} = [\![E']\!]$ and the distance between c_k and c_{j-1} is $d-1$. From Property 5 for c_{j-1} we then know that there was a transition from a state $P_h = \langle Sc_h, \sigma_h, s\chi_h\rangle$ with some ℓ_h s.t. $\sigma_h(\ell_h) = \langle \theta_h, \gamma_h, \chi_h, \xi_h\rangle$ and $\chi = \chi_k; \xi_h(x_h)$, for some x_h defined in S_i. We also know that $\xi_h(x_h) \in \mathcal{L}$, as otherwise we would have a variable expression $x_1.\cdots .x_h.x$ in $\mathcal{P}$ where x_h resolves to a value, not a reference, which is not provable in the type system of $\mathcal{P}$. As $\mathcal{P}$ is well-formed, this cannot happen. By Lemma 7.15, we know that $\sigma(\xi_h(x_h)) = \langle \theta_h, \gamma_h, \chi_h, \xi_h\rangle$ for some $\theta_h, \gamma_h, \chi_h, \xi_h$ and we also know that, with θ_h the struct type of $\sigma(\xi_h(x_h))$, $x \notin Pars(\theta_h, \mathcal{P}) \Rightarrow \xi_h(x_h) = \ell$, as otherwise $x_1.\cdots .x_h.x$ would not be admitted by the type rule **VarR**.

As $\mathcal{P}$ is well-formed, we know that there was either a statement $S_q = (T\ x := E_x)$ with $1 \leq q < i$ and $E_x \in EX$ or that $(x : T) \in Pars(\theta_h, \mathcal{P})$ for some $T \in \mathcal{T}$, as otherwise $x_1.\cdots .x_h.x$ would not be provable in the type system. Then, if $T \in \Theta_{\mathcal{P}}$, we know that $\xi(x) \in \mathcal{L}$. By Lemma 7.15 we then know that $\sigma(\xi_h(x)) = \langle \theta_x, \gamma_x, \chi_x, \xi_x\rangle$ for some $\theta_x, \gamma_x, \chi_x, \xi_x$ and we know that as $\mathcal{P}$ is well-formed, $\theta_x = T$, as otherwise $x_1.\cdots .x_h.x$ would not be provable in the type system.

Then there exists a transition **ComRd** to a state $P' = \langle Sc', \sigma', s\chi'\rangle$ with
$\sigma'(\ell) = \langle \theta, c_{j+1}; \ldots ; c_m; [\![S_{i+1}; \cdots ; S_n]\!], \chi_k; \xi_h(x), \xi'\rangle$, by definition of **ComRd**.

wr If $c_j = \mathbf{wr}(x)$ for some x, then $c_j = c_m$, by the definition of the interpretation function. It follows that $c_k = c_1$ and that $c_k; \ldots ; c_m = [\![S_i]\!]$. By the interpretation function, it follows that $[\![S_i]\!] = [\![E']\!]; [\![x_1.\cdots .x_h]\!]; \mathbf{wr}(x)$ for some expression E' and some variable expression $x_1.\cdots .x_h$. By the induction hypothesis on the commands $[\![E']\!] = c_1; \ldots ; c_a$, we know that after command c_a, we are in a state $P_a = \langle Sc_a, \sigma_a, s\chi_a\rangle$, with $\sigma_a(\ell) = \langle \theta, c_{a+1}; \ldots ; c_m; [\![S_{i+1}; \ldots ; S_n]\!], \chi_k; v, \xi_a\rangle$ for some value $v \in \mathcal{V}$. By the induction hypothesis on the commands $[\![x_1.\cdots .x_h]\!] = c_{a+1}; \ldots ; c_{m-1}$ and analogous reasoning as in the case above, we know that after command c_{m-1}, we are in a

state $P_b = \langle Sc_b, \sigma_b, s\chi_b\rangle$, with $\sigma_b(\ell) = \langle \theta, c_m; [\![S_{i+1}; \ldots; S_n]\!], \chi_k; v; \ell_h, \xi_a\rangle$ for some $\ell_h \in \mathcal{L}$, with $\sigma(\ell_h) = \langle \theta_h, \gamma_h, \chi_h, \xi_h\rangle$ for some $\theta_h, \gamma_h, \chi_h, \xi_h$ and with $x \notin Pars(\theta_h, \mathcal{P}) \Rightarrow \ell' = \ell$.

Again, as $\mathcal{P}$ is well-formed, we know that there was either a statement $S_q = (T\ x := E_x)$ with $1 \leq q < i$ and $E_x \in EX$ or that $(x : T) \in Pars(\theta_h, \mathcal{P})$ for some $T \in \mathcal{T}$, as otherwise $x_1.\cdots.x_h.x$ would not be provable in the type system. Then, if $T \in \Theta_{\mathcal{P}}$, we know that $v \in \mathcal{L}$. By Lemma 7.15 we then know that $\sigma(v) = \langle \theta_v, \gamma_v, \chi_v, \xi_v\rangle$ for some $\theta_v, \gamma_v, \chi_v, \xi_v$ and we know that as $\mathcal{P}$ is well-formed, $\theta_v = T$, as otherwise $x_1.\cdots.x_h.x := E'$ would not be provable in the type system.

Then if $\ell_h \notin \mathcal{L}_0$, there exists a transition **ComWr** to a state $P' = \langle Sc', \sigma', s\chi'\rangle$ with $\sigma'(\ell) = \langle \theta, [\![S_{i+1}; \cdots; S_n]\!], \chi_k, \xi'\rangle$ and $\sigma'(\ell') = \langle \theta_{\ell'}, \gamma_{\ell'}, \chi_{\ell'}, \xi'_{\ell'}\rangle$ s.t. $\xi'_{\ell'} = \xi_{\ell'}[x \mapsto v]$ and if $x \notin Pars(\theta_{\ell'}, \mathcal{P}) \vee \xi_{\ell'}(x) \neq v$ then $s\chi' = false^{|s\chi|}$ by definition of **ComWr**.

If $\ell_h \in \mathcal{L}_0$, there exists instead a transition **ComWrNSkip** to a state $P' = \langle Sc', \sigma', s\chi\rangle$ with $\sigma'(\ell) = \langle \theta, [\![S_{i+1}; \cdots; S_n]\!], \chi_k, \xi'\rangle$ and $\sigma'(\ell') = \langle \theta_{\ell'}, \gamma_{\ell'}, \chi_{\ell'}, \xi_{\ell'}\rangle$ by definition of **ComWrNSkip**.

cons If $c_j = \mathbf{cons}(\theta)$ for some struct type θ, then it must have been generated as the last command of an expression or statement $\theta(E_1, \ldots, E_h)$. It then follows that the largest possible c_k is the first command of $\theta(E_1, \ldots, E_h)$. Then let $Pars(\theta, \mathcal{P}) = (x_1 : T_1); \ldots; (x_h : T_h)$. It follows from repeated application of the induction hypothesis that every expression $E_i \in E_1, \ldots, E_h$ leaves a value $v \in \mathcal{V}$ on the stack, s.t. $\chi = \chi_k; v_1; \ldots; v_h$. We also know for every expression $E_i \in E_1, \ldots, E_h$ that either $T_i \in \{\texttt{Nat}, \texttt{Int}, \texttt{Bool}, \texttt{String}\}$ and $v_i \in sem(T_i)$ or $T_i \in \Theta_{\mathcal{P}}$ and $v_i \in \mathcal{L}$, as otherwise $\mathcal{P}$ would not be well-formed as the constructor statement or expression would not be provable in the type system. We then know that if $v_i \in \mathcal{L}$, by Lemma 7.15 $\sigma(v_i) = \langle \theta_i, \gamma_i, \chi_i, \xi_i\rangle$, and we know by the well-formedness of $\mathcal{P}$ that $\theta_i = T$.

Then by the definition of the transition **ComCons** with a fresh label ℓ', there exists a transition to a state $P' = \langle Sc', \sigma', s\chi'\rangle$ with $\sigma'(\ell) = \langle \theta, c_{j+1}; \ldots; c_m; [\![S_{i+1}; \cdots; S_n]\!], \chi_k; \ell', \xi'\rangle$ and $\sigma'(\ell') = \langle \theta, \varepsilon, \varepsilon, \xi^0_\theta[\{p_1 \mapsto v_1, \ldots, p_h \mapsto v_h\}]\rangle$ and $s\chi' = false^{|s\chi|}$.

if If $c_j = \mathbf{if}(C)$ for some command list C, then c_j must have been generated as part of an if-statement, s.t. $c_j = c_m$ and $[\![S_i]\!] = [\![E']\!]; \mathbf{if}([\![\mathcal{S}]\!])$ for some expression E' and some list of statements $\mathcal{S}$. It follows that $c_k = c_1$. Then by application of the induction hypothesis on $[\![E']\!] = c_1; \ldots; c_{m-1}$, we know that $\chi = \chi_k; b$, where $b \in \mathbb{B}$ because $\mathcal{P}$ would not be well-formed otherwise.

Then by the definition of the transitions **ComIfT** and **ComIfF**, there exists a transition to a state $P' = \langle Sc', \sigma', s\chi'\rangle$ with $\sigma'(\ell) = \langle \theta, \gamma', \chi_k, \xi'\rangle$ s.t. if $b = true$, $\gamma' = C; [\![S_{i+1}; \cdots; S_n]\!]$. and if $b = false$, $\gamma' = [\![S_{i+1}; \cdots; S_n]\!]$.

not If $c_j = \mathbf{not}$, it must have been generated as the last command of an expression $!E'$ for some expression E'. It follows that the largest possible c_k is the first command of $[\![!E]\!] = [\![E]\!]; \mathbf{not}$. By application of the induction hypothesis on $[\![E']\!]$, we then know that $\chi = \chi_k; b$, and we know that $b \in \mathbb{B}$ because otherwise $\mathcal{P}$ would not be well-formed.

Then by the definition of the transition **ComNot** there exists a transition to a state $P' = \langle Sc', \sigma', s\chi'\rangle$ with $\sigma'(\ell) = \langle \theta, [\![S_{i+1}; \cdots; S_n]\!], \chi_k; \neg b, \xi'\rangle$.

op If $c_j = \mathbf{op}(\circ)$ for some syntactic operator $\circ$, it must have been generated as the last command of an expression $E_1 \circ E_2$. It follows that the largest possible c_k is the first command of $[\![E_1 \circ E_2]\!] = [\![E_1]\!]; [\![E_2]\!]; \mathbf{op}(\circ)$. Then, by induction hypothesis on $[\![E_1]\!]$ and $[\![E_2]\!]$, and the relation between the interpretation function and the stack in the semantics, we know that $\chi = \chi_k; a; b$ for $a, b \in \mathcal{V}$. We then know that if $\circ \in \{=, !=\}$ then $a, b \in \{\mathbb{N}, \mathbb{Z}\}$ or there exists a $T \in \{\mathbb{N}, \mathbb{Z}, \mathbb{B}, String\} \cup \Theta_{\mathcal{P}}$ s.t. either $T \notin \Theta_{\mathcal{P}}$ and $a, b \in T$ or $T \in \Theta_{\mathcal{P}}$, by the type rules **Eq** and **Comp**. We also know that and $\sigma(a) = \langle T, \gamma_a, \chi_a, \xi_a\rangle$ and $\sigma(b) = \langle T, \gamma_b, \chi_b, \xi_b\rangle$ by Lemma 7.15 and the type rule **Eq**. By the type rules **Comp**, **NArith** and **IArith**, we know that if $\circ \in \{<=, >=, <, >, =, !=, *, /, \%, +, \hat{}, -\}$ then $a, b \in \{\mathbb{N}, \mathbb{Z}\}$, and by the type rule **BinLog** we know that if $\circ \in \{\&\&, ||\}$ then $a, b \in \mathbb{B}$.

Then by the definition of **ComOp**, there exists a transition to a state $P' = \langle Sc', \sigma', s\chi'\rangle$ with $\sigma'(\ell) = \langle \theta, c_{j+1}; \ldots; c_m [\![S_{i+1}; \cdots; S_n]\!], \chi_k; c\xi'\rangle$ s.t.:

(a) if $a \in T$ and $b \in T$ with $T \in \mathbb{T}$ and $\circ = \{=, !=\}$, then $c = a \mathbin{o} b$ and $c \in \mathbb{B}$,
(b) if $a \in T_1$ and $b \in T_2$, with $T_1, T_2 \in \{\mathbb{N}, \mathbb{Z}\}$ and $\circ = \{<=, >=, <, >, =, !=\}$, then $c = a \mathbin{o} b$ and $c \in \mathbb{B}$,
(c) if $a \in T_1$ and $b \in T_2$, with $T_1, T_2 \in \{\mathbb{N}, \mathbb{Z}\}$ and $\circ = \{*, /, \%, +, \hat{}, -\}$, then $c = a \mathbin{o} b$ and $c \in \mathbb{Z}$,
(d) if $a, b \in \mathbb{N}$ and $\circ = \{*, /, \%, +, \hat{}\}$, then $c = a \mathbin{o} b$ and $c \in \mathbb{N}$ and
(e) if $a, b \in \mathbb{B}$ and $\circ \in \{\&\&, ||\}$, then $c = a \mathbin{o} b$ and $c \in \mathbb{B}$.

It follows that the step holds for all possible cases, so the induction holds.

As the induction holds, Property 5 holds. □

Lastly, we prove Property 6, which states that step executions are finite.

Proof of Property 6. Let $P = \langle Sc, \sigma, s\chi \rangle$ be a well-formed AuDaLa state reached by some program $\mathcal{P} = D\ Sc_{\mathcal{P}}$ s.t. $\Gamma, \Omega \vdash D\ Sc_{\mathcal{P}}$ for some Γ, Ω. Then assume that $\neg Done(\sigma)$. Let k be the total amount of commands still present in some command list of some struct instance. $k > 0$, as $Done(\sigma) = false$. Then the amount of transitions enabled directly corresponds to the commands c s.t. for some $\ell' \in \mathcal{L}$, $\sigma(\ell') = \langle \theta', c; \gamma', \chi', \xi \rangle$. It follows that after a transition to a state P_1, there are $k - 1$ commands in total still present in some command list of some struct instance, and the transitions enabled are similarly bound to commands as in P. As k must be a finite number, as $\mathcal{P}$ is well-formed (with a finite syntax), it follows that after a finite amount of transitions, we arrive in a state $P' = \langle Sc', \sigma', s\chi' \rangle$ s.t. $Done(\sigma'') = true$. □

As all properties hold, we conclude that Theorem 7.14 holds.

To use this theorem, we can shift the focus from commands to expressions and statements. For expressions, we define *resulting values* for expressions:

Definition 7.16 (Expression Results). Let the result v of an execution of an expression E in a step F in a well-formed program $\mathcal{P}$ by a struct instance s be the value v resulting from the transition induced by the last command c_m of $[\![E]\!]$ in the command list of s as per Theorem 7.14.

Note that there always is at least one resulting value for every expression, by Theorem 7.14. However, this value does not have to be deterministic, because we allow for interleaving during the execution of an expression E by struct instance s. After all, if E contains a read to a variable which is not deterministic during F, then the result of E cannot be deterministic either. However, the following holds:

Lemma 7.17. *Let E be an expression in a step F in a well-formed program $\mathcal{P}$ executed by a struct instance s from some state P. Let $x_1, \ldots, x_n$ be the parameters referenced during E. If all parameters $x_i \in x_1, \ldots, x_n$ are deterministic during the execution of F from P, then the result of E executed by s is also deterministic during the execution of F from some state P.*

Proof. If all of the parameters of E are deterministic, it follows that any computations on them have a deterministic outcome, so the result is also deterministic. □

In this paper, we are interested in proving programs without read-write race conditions from some premise state P_1. Note that in program executions without read-write race conditions from P_1, all resulting values from all expressions are deterministic, as the fact that parameters are read in an expression E executed by a struct instance s during the execution of a step F from some state P means they cannot be written to during F, which leads to the parameters being deterministic. So in a program without read-write race conditions from P_1, expression results are by default deterministic.

On a more general level, we know from Theorem 7.14 that the execution of a statement S in a step F in a well-formed program $\mathcal{P}$ by a struct instance s leads to the effects in Property 5 depending on the last command in $[\![S]\!]$. Intuitively, this means the following:

Corollary 7.18 (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.

Proof. Follows directly from applications of Theorem 7.14 and Lemma 7.17. □

As in this paper we assume our programs do not have read-write race conditions from some premise state, this means that the direct effects of all statements are deterministic. However, we do not exclude write-write race conditions, so we need to weaken the above Corollary to find the permanent results of executing update statements in a step:

Corollary 7.19 (Permanent Update Results)**.** *Let $\mathcal{P}$ be a well-formed program without read-write race conditions from some premise state P_1, with a step F. Let F be executed from some state P and let s be a struct instance in P. Let S be the last update statement of some parameter $s'.x$ of some struct instance s' by s, which updates $s'.x$ to a value v. Let $\mathcal{S}$ be the last update statements of $s'.x$ across all definitions of F for any struct type θ. Then we can guarantee that after the execution of F from P:*

a. If s' is a null-instance, $s'.x = a = null$.

b. If s' is not a null-instance:

i. If $s'.x$ is not involved in a write-write race condition, $s'.x = b$.

ii. If $s'.x$ is involved in a write-write race condition in F, let N be the set of all possible values written to $s'.x$ during the execution of any statement from $\mathcal{S}$ by any struct instance during the execution of F from P. Then $s'.x \in N$.

Additionally, we know that if $s'.x$ has had its value changed during the execution of F, all values in the stability stack have been reset to false during the execution of F.

Implications of Theorem 7.14 and its corollaries. With the theorems, lemmas and corollaries proven in this section, one can prove the effects of an AuDaLa step F by hand on the level of the syntactic code of F in the program, without transforming that behaviour into commands, using the above corollaries to formally establish the behaviour of statements. For variables x deterministic during the execution of F, one only has to consider the final value of x relative to a single struct instance, which can then be generalised to all struct instances. This can be done by sequentially walking through the code of F. For variables x involved in write-write race conditions, one will need to construct the set N as defined in Corollary 7.19.

7.2.1. Type Soundness

With theorem 7.14, we can prove type soundness. We first give a small overview of what type soundness means for a type system, and then give the proof for AuDaLa's type system being sound.

Type soundness was first defined by Milner [38] as the property that well-typed programs do not go "wrong". This is further expanded upon by Wright and Felleisen [39], which identify this as *weak type soundness*, and more specifically define it as the absence of type errors during execution. They give a definition of *strong type soundness*, stating that if a program (or expression) is evaluated to be of some type by the type system, the value returned from evaluating that program also has that type. They then give the well-known syntactic approach for proving type soundness based on progress and preservation.

Note that weak type soundness is indeed weaker than strong type soundness. Weak type soundness will be satisfied if there are no type errors, even if there are type mismatches in the program (as long as they don't incur a type error). Strong type soundness does not allow for type mismatches between the program and its evaluation.

In our language, there are essentially two levels of the semantics. The first level works directly on the syntax and deals with the execution of the schedule. The second level works on commands generated by the syntax and deals with the execution of the statements of steps occurring in the schedule. To establish type soundness for AuDaLa, we then have to consider the type soundness of both levels.

Note that due to the fact that a good part of the semantics does not directly work on the syntax of the program, we cannot use progress and preservation. As defining a type system on the commands does not have a use beyond proving type soundness using the progress and preservation method (for commands only, not for the schedule), we have decided to instead establish type soundness using the semantics directly.

To prove the type soundness of our type system, we will need to prove that in every state of a well-formed program $\mathcal{P}$, we prove that weak and strong soundness hold for the execution of $\mathcal{P}$. We first establish weak soundness. To do this, we first have to define when a type error occurs in AuDaLa:

Definition 7.20 (Type Errors in AuDaLa). Let $P = \langle Sc, \sigma, s\chi \rangle$ be an AuDaLa state. We define a *type error* to occur in P iff one of the following holds:

1. The schedule Sc is neither empty nor well-formed modulo $aFix$.
2. The values on the stack of a struct instance in σ are not compatible with a command transition enabled for that struct instance.
3. A transition is enabled in some struct instance of σ that writes a value to a variable environment s.t. the variable written to and the value have different types.

Note that as the transitions on the schedule match with the first entry in the schedule, a schedule that is neither empty nor well-formed will terminate the program early, instead of incur a visible type error. We still count it though, as the error has been incurred by a mismatch in expected schedule content and actual schedule content.

Note then that AuDaLa's type system is weakly sound:

Corollary 7.21 (Weak type soundness for the type system of AuDaLa)**.** *There will be no type errors in any state encountered during any execution of a well-formed AuDaLa program $\mathcal{P}$.*

Proof. This follows from Theorem 7.14: the first property directly contradicts that an occurrence of the first possible type error can happen in $\mathcal{P}$, and the fifth property contradicts the occurrence of the second and third types of type error with its strong requirements on the types of the values on the stack and the values written to variable environments. □

Then, to prove that strong soundness holds, note that the only dynamic parts of an AuDaLa program that have types are the schedule and expressions used in steps. As we already established that the schedule is well-formed right until it is empty (which is expected, as if the schedule is empty the program terminates), we then only need to establish that the expressions evaluated during a program are evaluated to a value which matches type with the expression.

To do this, we define the following lemma:

Lemma 7.22 (Strong type soundness Addendum)**.** *Let $\mathcal{P}$ be a well-formed program, and let E be an expression defined in a step of a struct type with type θ. Let E have type T in $\mathcal{P}$ according to the type system. Let P be an AuDaLa state reached by $\mathcal{P}$ from which E can be executed. Then either $T \in \mathcal{T} \setminus ID$ and the result value of E executed from P will have the semantic type $sem(T)$, or $T \in \Theta_{\mathcal{P}}$ and the result value of E executed from P will be a label ℓ s.t. the struct type of $\sigma(\ell)$ is T.*

Proof. This follows from Property 5 of Theorem 7.14, but to properly establish it, we need to apply structural induction. As base cases, we have the following cases:

- If $E =$ `this`, then we know from the type system that the type of E must be θ, where θ is the struct type of the step E is defined in in $\mathcal{P}$. As $[\![E]\!] = \mathbf{push}(\mathbf{this})$ (by definition of the interpretation function), we know from Property 5 of Theorem 7.14 that if E is executed by a struct instance with struct type θ', the result value is a label of that struct instance. As E is defined as a part of a struct type θ, only struct instances with struct type θ can execute E, so the struct instance of the returned label ℓ will have type θ.

- If $E = null$, then let this occurrence of null be annotated with the type T in the syntax tree. We then know from the type system that E must have type T. From the interpretation function, we know that $E = \mathbf{push}(defaultVal(T))$. We then know from Property 5 of Theorem 7.14 that if E is executed by P it will push a value $v = defaultVal(T)$ to the stack. This value will then have type $sem(T)$ if $T \in \mathcal{T} \setminus ID$ and will reference a struct instance with struct type T if $T \in \Theta_{\mathcal{P}}$, by definition of $defaultVal$.

- If $E = g$, with $g \in LT$, then let $T \in \mathcal{T} \setminus ID$ be the syntactic type of g. We then know that E has type T according to the type system. From the interpretation function, we know that $E = \mathbf{push}(g)$. We then know from Property 5 of Theorem 7.14 that if E is executed by P it will push a value $val(g)$ to the stack. By definition of $val(g)$, this value has the type $sem(T)$, as it is the semantic equivalent of g.

- If $E = x$, with $x \in ID$, then we know that x must be a variable in $\mathcal{P}$. It follows that according to the type system it must have a type T, and that it is either a parameter or it is an already defined local variable (due to type system rules **SeqLVar** and **LVar**). It follows that with ξ the environment variable of the executing struct instance, $\xi(x) = v$ for some value v. From Property 5 of Theorem 7.14 we then know that v has type T if $T \in \Theta_{\mathcal{P}}$ and $sem(T)$ if $T \in \mathcal{T} \setminus ID$. From the type system, it then follows that E must also have type T. We then know from the interpretation function that $[\![E]\!] = \mathbf{push}(this); \mathbf{rd}(x)$, which means that the execution of E pushes first the label of the executing struct instance on the stack and then takes it to put v on the stack. It follows that v is the result value of E and that if E has type T, then if $T \in \mathcal{T} \setminus ID$, v has type $sem(T)$ and if $T \in \Theta_{\mathcal{P}}$, v has type T.

As the induction hypothesis, we assume that the lemma holds for all subexpressions of E. We then have the following step cases:

- If $E = X.x$, then we know from the type system that X has a type θ, and from the lemma we then know that the result value of X is a label ℓ for a struct instance with struct type θ. Then from the interpretation function we know that $E = [\![X]\!]; \mathbf{rd}(x)$, so with ξ_X the variable environment of the struct instance of the label ℓ, we know that E's return value is $\xi_X(x)$. Then we also know from the type system that x is a parameter with type T, containing a value v of type T if $T \in \Theta_{\mathcal{P}}$ or $sem(T)$ if $T \in \mathcal{T} \setminus ID$, by Theorem 7.14, property 5. From the type system, we then know that E must also have type T, and from Theorem 7.14, Property 5, we then know that the result value of E is v, due to which the lemma holds for this case.

- If $E = \theta(E_1, \ldots, E_n)$, then we know from the type system that E's type is θ. We then know from the interpretation function that $[\![E]\!] = [\![E_1]\!]; \ldots; [\![E_n]\!]; \mathbf{cons}(\theta)$ and we know from Theorem 7.14, Property 5, that $\mathbf{cons}(\theta)$ then returns the label of a struct instance with type θ. It then follows that the lemma holds in this case.

- If $E = !E'$, then we know from the type system that E has type `Bool`. From the interpretation function it then follows that $[\![E]\!] = [\![E']\!]; \swarrow$and as our type system requires that E' also has type `Bool`, we know that the value returned from $[\![E']\!]$ is a boolean. We then know by Theorem 7.14, Property 5, that the value returned from E is also a boolean, which satisfies the lemma in this case.
- If $E = (E')$, then let T be the type assigned by the type system to E. As $[\![E]\!] = [\![E']\!]$, and E' also has type T according to the type system, we can then apply the induction hypothesis to conclude that the lemma also holds for E.
- If $E = E_1 \circ E_2$, then by the interpretation function, we know that $[\![E]\!] = [\![E]\!]_1; [\![E]\!]_2; \mathbf{op}(\circ)$. Then we know by the induction hypothesis that the result values of E_1 and E_2 have the correct corresponding types to E_1 and E_2. By the interpretation function and the definition of the semantics, these must then be at the end of the stack. As $\mathcal{P}$ is well-formed, $\circ$ must be compatible with these values. Then, for all types T that E can have according to the type system and all possible $\circ$, it follows from Theorem 7.14 that the return value of E will have the type $sem(T)$.

From this it follows that the lemma holds. □

With this, we then conclude the following theorem:

Theorem 7.23. *The type system of AuDaLa is sound.*

Proof. By Corollary 7.21 and Lemma 7.22. □

8. Proving AuDaLa Code Correct

In this section, we prove the AuDaLa code of Listing 4 (Section 8.1) and Listing 7 (Section 8.2) correct, using the definitions, properties and lemmas of Section 7 and concepts from Sections 3, 4 and 5. We then conclude this section with some more general insights in the structure of proving AuDaLa programs correct (Section 8.3). During the proofs, we consider the program in a vacuum; we assume that if something is not changed by the program as a whole, it is not changed at all. With this, we rule out outside interference by hardware. As this paper is concerned with the theoretical foundations of AuDaLa and we intend to prove the correctness of AuDaLa programs, not AuDaLa program implementations we consider this to be a fair assumption. We also omit the type system proof derivation to prove both programs well-formed.

8.1. Proving Prefix Sum Correct.

In this subsection, we prove that the code as given in Listing 4 correctly executes the Prefix Sum function. We start by stating the problem formally:

Problem 8.1 (Prefix Sum). Given a sequence of elements $p_1, \ldots, p_n$, with values $x_1, \ldots, x_n$ for these elements, compute for each position $1 \leq i \leq n$ the sum $\Sigma_{k=1}^{i} x_k$. The sequence is given as a single, finite list without loops.

As the code does not describe initialization, we need to specify the requirements of a premise state from which this code can be executed with the intended effect. These requirements have already been stated in Section 6, but are formalized here:

Definition 8.1 (Premise State). We define a *premise state* of the Prefix Sum code as a state $P_1 = \langle Sc, \sigma, s\chi \rangle$ s.t. there are labels $\ell_1, \ldots, \ell_n$ for positions $p_1, \ldots, p_n$ where $\sigma(\ell_1) = \langle Position, \varepsilon, \varepsilon, \xi_1 \rangle$ is the first position p_1, with $p_1.val = x_1$ and $p_1.prev = null$, and for every p_i with $1 < i \leq n$, $p_i = \sigma(\ell_i) = \langle Position, \varepsilon, \varepsilon, \xi_i \rangle$, with $p_i.val = x_i$ and $p_i.prev = \ell_{i-1}$.

To prove the code in Listing 4 correct, we first consider the steps of this code, for which we prove the effects. The strategy is to first consider which parameters are in a race condition, if any, using information on the states from which the step is executed. Then, we can establish the effects of the step on the parameters using Corollary 7.19. For parameters not involved in race conditions, this consists of walking through the step sequentially and establishing the last value written to the parameter. After we have considered the effects of all the steps, we combine them into fixpoint inner contracts. These we then use to prove the effects of fixpoints, from which we establish the final result of executing the code. If this code corresponds to the requirements as stated in the problem, it is correct provided the state from which it is executed is a valid premise state.

For Listing 4, instead of considering variables separately we prove that all steps are deterministic as a whole:

Lemma 8.2. *All steps in Listing 4 are deterministic.*

Proof. This follows from the fact that no step in Listing 4 has potential race conditions:

- In *read*, a potential race condition would need to include either *auxval* or *auxprev*, as those are the parameters written to in *read*. However, these parameters are not accessed indirectly, so there can be no potential race conditions on them, following from Definition 7.9. Therefore, *read* has no potential race conditions.
- In *write*, no parameters used are accessed indirectly, which is required for a potential race condition to exist, according to Definition 7.9. Therefore, *write* has no potential race conditions.

Therefore, no step in Listing 4 has potential race conditions. It follows from Lemma 7.10 that no step in Listing 4 has race conditions. It follows by Lemma 7.6 that all steps in Listing 4 are deterministic. □

As all steps are deterministic, we can walk through a step sequentially to determine its effects. That way, we establish the following contracts for *read* and *write* in Listing 4:

Lemma 8.3 (*read* Contract)**.** *Let $P = \langle Sc, \sigma, s\chi\rangle$ be an AuDaLa state and let $\mathcal{L}_{Position}$ be the labels of Position structs in P. Executing* read *starting at state P results in a state $P' = \langle Sc, \sigma', s\chi\rangle$. In P', for every $\ell_p \in \mathcal{L}_{Position}$ s.t. $p = \sigma(\ell_p) = \langle Position, \gamma, \chi, \xi_p\rangle$, let p' be $\sigma'(\ell_p)$. Then $p'.auxval = p.prev.val$ and $p'.auxprev = p.prev.prev$.*

Proof. Let P, P', p and p' be as defined in the lemma. As *read* is deterministic, we can sequentially walk through the statements of the *read* step.

By applying Corollary 7.18 to the first statement of *read*, it follows that $p'.auxval = p.prev.val$. By applying it to the second statement of *read*, we know that $p'.auxprev$ is equal to $p.prev.prev$. □

Lemma 8.4. *Let $P = \langle Sc, \sigma, s\chi\rangle$ be an AuDaLa state and let $\mathcal{L}_{Position}$ be the labels of Position structs in P. Executing* write *starting at state P results in a state $P' = \langle Sc, \sigma', s\chi\rangle$. In P', for every ℓ_p s.t. $p = \sigma(\ell_p) = \langle Position, \gamma, \chi, \xi_p\rangle$, let p' be $\sigma'(\ell_p)$. Then $p'.val = p.val + p.auxval$ and $p'.prev = p.auxprev$.*

Proof. Let P, P', p and p' be defined in the lemma. As *write* is deterministic, we can sequentially walk through the statements of the *write* step. By applying Corollary 7.18 to all statements, we get that $p'.val = p.val + p.auxval$ and $p'.prev = p.auxprev$. □

We then consider the schedule. As the fixpoint in the schedule first executes *read* and then *write*, we define the following combined contract for *read* < *write*:

Lemma 8.5 (Fixpoint execution)**.** *Let $P = \langle Sc, \sigma, s\chi\rangle$ be an AuDaLa state and let $\mathcal{L}_{Position}$ be the labels of Position structs in P. Executing* read<write *starting at state P results in a state $P' = \langle Sc, \sigma', s\chi\rangle$. In P', for every ℓ_p s.t. $p = \sigma(\ell_p) = \langle Position, \gamma, \chi, \xi_p\rangle$, let p' be $\sigma'(\ell_p)$. Then $p'.val = p.val + p.prev.val$, $p'.prev = p.prev.prev$, $p'.auxval = p.prev.val$ and $p'.auxprev = p.prev.prev$.*

Proof. Follows from Lemmas 8.3 and 8.4. □

We now prove the following two lemmas. As the proof for these lemmas can be efficiently combined, we prove them together:

Lemma 8.6 (Termination)**.** *Executing* Fix(read < write) *terminates.*

Lemma 8.7 (Result)**.** *Executing* Fix(read < write) *starting at premise state* $P_1 = \langle Sc_1, \sigma_1, s\chi_1 \rangle$ *results in a state* $P' = \langle Sc', \sigma', s\chi' \rangle$*. In* P'*, for every* ℓ_i *s.t. position* $p_i = \sigma(\ell_i)$ *with* $1 \leq i \leq n$*, let* p'_i *be* $\sigma'(\ell_{p_i})$*. Then* $p'_i.val = \sum_{k=1}^{i} p_k.val$*.*

Proof of Termination and Result. For the first lemma, the fixpoint terminates when an iteration is reached which does not change any of the parameters. We need to prove that this iteration is eventually reached. The second lemma is more straightforward.

To prove both, we will first prove that after every iteration j of the fixpoint, $p'_i.val = \sum_{k=h}^{i} p_k.val$, with $h = \max(1, i - 2^j + 1)$ and $p'_i.prev = \ell_{i-2^j}$ or *null* if $i - 2^j \leq 0$ for any $1 \leq i \leq n$, by induction on j.

$j = 1$. For the first iteration, we start at premise state P_1 and end at a state P_2. In P_2, according to Lemma 8.5, $p'_i.val = p_i.val + p_i.prev.val = p_i.val + p_{i-1}.val$, with $p_{i-1} = \sigma_1(\ell_{i-1})$. If $i - 1 < 1$, then $p_{i-1} = null$, as $p_1.prev = null$, so $p_{i-1}.val = 0$ and then $p_i.val + p_{i-1}.val = p_1.val$. Therefore $p'_i.val = p_i.val + p_{i-1}.val = \sum_{k=h}^{i} p_k.val$, with $h = \max(1, i - 1) = \max(1, i - 2^j + 1)$, as $j = 1$.

We also know from Lemma 8.5 that $p'_i.prev = p_i.prev.prev$. If $i - 2 > 0$, then $i - 1 > 0$, so with $p_i.prev = \ell_{i-1}$ and $p_{i-1} = \sigma(\ell_{i-1})$, $p_{i-1}.prev = \ell_{i-2}$, so $p'_i.prev = \ell_{i-2} = \ell_{i-2^j}$. If $i - 2 \leq 0$, then either $i - 1 \leq 0$, in which case $p_i.prev.prev = null.prev = null$, or $i - 1 > 0$, in which case $p_{i-1}.prev = null$, so $p_i.prev.prev = null$ and $p'_i.prev = null$.

$j > 1$. By the induction hypothesis, when we start at state $P_{j-1} = \langle Sc, \sigma, s\chi \rangle$ for iteration j, for any $p_i = \sigma(\ell_i)$, $p_i.val = \sum_{k=h}^{i} p_k.val$ with $h = \max(1, i - 2^{j-1} + 1)$ and $p_i.prev = \ell_{i-2^{j-1}}$ or *null* if $i - 2^{j-1} \leq 0$. Due to Lemma 8.5, it follows that we end up in a state $P_j = \langle Sc', \sigma', s\chi' \rangle$ s.t. for $p'_i = \sigma(\ell_i)$, $p'_i.val = p_i.val + p_i.prev.val$ and $p'_i.prev = p_i.prev.prev$.

From the IH we know that $p_i.val = \sum_{k=h}^{i} p_k.val$ with $h = \max(1, i - 2^{j-1} + 1)$. Then if $i - 2^{j-1} \leq 0$ we know that $i - 2^j \leq 0$ and that $p_i.prev = null$, so $p'_i.prev = null.prev = null$. Additionally, $h = 1$, so $p_i.val = \sum_{1}^{i} p_k.val$, and $p'_i.val = p_i.val + p_i.prev.val = \sum_{1}^{i} p_k.val + 0 = \sum_{1}^{i} p_k.val$.

If $i - 2^{j-1} > 0$, then we know that $p_i.prev = \ell_{i-2^{j-1}}$ with $p_{i-2^{j-1}} = \sigma(\ell_{i-2^{j-1}})$. Then either $i - 2^j \leq 0$ or not.

If $i - 2^j \leq 0$, then $(i - 2^{j-1}) - 2^{j-1} \leq 0$, and then by the IH we know that $p_{i-2^{j-1}}.prev = null$, so $p'_i.prev = q_i.prev.prev = null$. Otherwise, $p_{i-2^{j-1}}.prev = \ell_{i-2^j}$, so $p'_i.prev = \ell_{i-2^j}$.

As $i - 2^{j-1} > 0$, we know that $p_i.val = \sum_{k=i-2^{j-1}+1}^{i} p_k.val$. We also know that $p_{i-2^{j-1}}.val = \sum_{k=g}^{i-2^{j-1}} p_k.val$ with $g = \max(1, i - 2^j + 1)$ by the IH and the facts established in the last paragraph. Then $p'_i.val = p_i.val + p_{i-2^{j-1}}.val = \sum_{k=i-2^{j-1}+1}^{i} p_k.val + \sum_{k=g}^{i-2^{j-1}} p_k.val = \sum_{k=g}^{i} p_k.val$.

As this proves all parts of the IH for iteration j, the induction step holds.

Note that $null.prev = null$ and $null.val = 0$. As per the induction, for any p_i and any iteration j such that $2^j \geq i$, $p_i.prev = null$. Therefore the *prev* parameter for p_i will be stable in iteration $\lceil \log_2(i) \rceil + 1$.

Also note that in iteration $\log_2(i) + 1$, $p_i.val$ will be stable, as per the induction. Therefore, Lemma 8.6 holds; the fixpoint will terminate after $\lceil \log_2 n \rceil + 1$ iterations, as the last of the n elements is the last to become stable.

After $\log_2(n) + 1$ iterations, $p'_i.val = \sum_{k=1}^{i} p_k.val$, as per the induction, so Lemma 8.7 holds. □

This results in our conclusion:

Theorem 8.8 (Correctness of Listing 4)**.** *When executing the AuDaLa code as given in Listing 4 from a premise state according to Definition 8.1 for a set of elements* $p_1, \ldots, p_n$ *with values* $x_1, \ldots, x_n$*, in the resulting state, every Position* q_i *corresponding to element* p_i *with* $1 \leq i \leq n$ *has its val parameter set to* $\sum_{k=1}^{i} x_k$*.*

Proof. Follows from Lemma 8.7 and the premise, as $\sum_{k=1}^{i} p_k.val = \sum_{k=1}^{i} x_k$. □

8.2. Proving Linked List Copy Sort Correct

In this subsection, we prove that the code as given in Listing 7 sorts correctly. We state the problem formally as:

Problem 8.2 (Sorting). Given a sequence S_1 of elements $p_1, \ldots, p_n$ with distinct positive values $x_1, \ldots, x_n$, rearrange these elements to a sequence S_2 $p'_1, \ldots, p'_n$ with values $x'_1, \ldots, x'_n$ s.t. for every element $p_i \in S_1$, a copy of that element p'_j exists s.t. $p'_j \in S_2$ and $x'_i < x'_{i+1}$ for every $1 \leq i < n$.

The premise state of Listing 7 is given as:

Definition 8.9 (Premise of Listing 7). We define a *premise state* of the Linked List Copy Sort code as a state $P_1 = \langle Sc, \sigma, s\chi \rangle$ s.t:

- For every element p_i in $p_1, \ldots, p_n$ there exists a struct instance a_i (in $a_1, \ldots a_n$) with label ℓ_i (in $\ell_1, \ldots, \ell_n$) s.t. $a_i = \langle \mathit{OldElem}, \varepsilon, \varepsilon, \xi_i \rangle$ and $\sigma(\ell_i) = a_i$.
- There exists a non-*null* struct instance b with label ℓ_b s.t. $b = \langle \mathit{NewElem}, \varepsilon, \varepsilon, \xi_b \rangle$.
- For all a_i, $a_i.val = x_i$ with $x_i \in x_1, \ldots x_n$, $a_i.place = b$ and $a_i.move = \mathit{false}$.
- $b.min = \min(x_1, \ldots, x_n)$, $b.max = \max(x_1, \ldots, x_n)$ (which can be done with a method akin to prefix-sum), $b.spl = \lfloor (b.max - b.min)/2 \rfloor + b.min$, $b.next = \ell^0_{\mathit{NewElem}}$, $b.p1 = \ell^0_{\mathit{OldElem}}$, $b.p2 = \ell^0_{\mathit{OldElem}}$, $b.hasSplit = \mathit{false}$ and $b.done = \mathit{true}$.

Considering the steps, note that to both *checkStable* and *migrate*, write-write race conditions are integral to its function. Therefore, the determinism of steps in Listing 7 can be encapsulated by the following lemma:

Lemma 8.10. *All of the following hold:*

1. *Only the parameter place.done can be involved in a race condition in the step checkStable, and if it is, it is involved in a write-write race condition.*
2. *Only the parameters place.p1 and place.p2 can be involved in a race condition in the step migrate, and if they are, they are involved in a write-write race condition.*
3. *The step split is deterministic.*

Proof. We prove all of the points separately:

1. From Listing 7, the only struct executing *checkStable* is the struct *OldElem*, and through the implementation of *checkStable* in *OldElem*, we know that *place.done* is the only parameter that is written to, so only *place.done* has potential race conditions. As *place.done* is not read in the code of *checkStable*, any race condition *place.done* is involved in must be a write-write race condition.
2. This point holds following the same reasoning as the first point, except with the parameters *place.p1* and *place.p2* and the step *migrate*.
3. In *split*, no parameters used are accessed indirectly, which is required for a potential race condition to exist, according to Definition 7.9. Therefore, *split* has no potential race conditions. It follows that *split* has no race conditions (Lemma 7.10) and therefore, *split* is deterministic (Lemma 7.6). □

With this lemma, we can establish contracts for the steps in Listing 7.

Lemma 8.11 (*migrate* Contract)**.** *Let $P = \langle Sc, \sigma, s\chi \rangle$ be an AuDaLa state and let $\mathcal{L}_{\mathit{OldElem}} \subseteq \mathcal{L}$ be the labels of OldElem structs in P. Executing migrate starting at state P results in a state $P' = \langle Sc, \sigma', s\chi \rangle$. In P', for every ℓ_p s.t. $p = \sigma(\ell_p) = \langle \mathit{OldElem}, \gamma, \chi, \xi_p \rangle$, $p' = \sigma'(\ell_p)$, all of the following hold:*

1. $p.move \wedge p.place.hasSplit \iff p'.place = p.place.next$,
2. $(p.val > p.place.spl) \iff p'.move = true$
3. $(p.val \leq p.place.spl) \implies \exists \ell \in \mathcal{L}_{OldElem}.(\sigma'(\ell) = q \wedge q.place = p'.place \wedge p'.place.p1 = \ell)$,
4. $(p.val > p.place.spl) \implies \exists \ell \in \mathcal{L}_{OldElem}.(\sigma'(\ell) = q \wedge q.place = p'.place \wedge p'.place.p2 = \ell)$.

Proof. Let P, P', p and p' be as defined in the lemma. As the only parameters accessed by p involved in a race condition are $p'.place.p1$ and $p'.place.p2$, which do not have any bearing on point 1 of the lemma, we can walk deterministically through the lines relevant for point 1, lines 7 to 13. Through application Corollary 7.18, point 1 follows, as *place* can only be updated when the if-statement resolves to true.

For point 2 and 3, note that the parameter *move* is not involved in any race conditions (as per Lemma 8.10), we then know through application Corollary 7.18 that *move* is set to *false* for the struct instance of ℓ_p after line 13 and that $p.val > p.place.spl \iff p'.move = true$.

Then to prove that $(p.val \leq p.place.spl) \implies \exists \ell \in \mathcal{L}_{OldElem}.(\sigma'(\ell) = q \wedge q.place = p'.place \wedge p'.place.p1 = \ell)$, let p_1 be the struct instance of the label ℓ_p at the moment of executing line 15 of the code. Then the update of $p_1.place.p1$ may be in a race condition with another struct instance executing line 15, q_1 with label ℓ_q, as long as $q_1.place = p_1.place$. Therefore, by Corollary 7.18 and Corollary 7.19, we know that after executing line 15, there exists a struct instance q_1 s.t. $q_1.place = p_1.place$ and $p_1.place.p1 = \ell_q$. As after line 13, the *place* parameter is not further updated in the step code, $p_1.place = p'.place$ and with $q = \sigma'(\ell_q)$, $q.place = q_1.place$, it follows by Corollary 7.18 and Corollary 7.19 that after executing line 15, $\exists \ell \in \mathcal{L}_{OldElem}.(\sigma'(\ell') = q \wedge q.place = p'.place \wedge p'.place.p1 = \ell)$.

When $p.val > p.place.split$, we know that $\ell_p \notin \mathcal{L}_0$, as then $p.val = 0$ and $p.place.split = 0$. Therefore, we know through Corollary 7.18 with line 19 that if $(p.val > p.place.split)$, $p'.move = true$. Proving that $(p.val > p.place.spl) \implies \exists \ell \in \mathcal{L}_{OldElem}.(\sigma'(\ell) = q \wedge q.place = p'.place \wedge p'.place.p2 = \ell)$ uses the same reasoning as proving that $(p.val \leq p.place.spl) \implies \exists \ell \in \mathcal{L}_{OldElem}.(\sigma'(\ell) = q \wedge q.place = p'.place \wedge p'.place.p1 = \ell)$, so point 4 holds. □

Lemma 8.12 (*checkStable* Contract)**.** *Let $P = \langle Sc, \sigma, s\chi \rangle$ be an AuDaLa state and let $\mathcal{L}_{OldElem} \subseteq \mathcal{L}$ be the labels of OldElem structs in P. Executing checkStable starting at state P results in a state $P' = \langle Sc, \sigma', s\chi \rangle$. In P', for every ℓ_p s.t. $p = \sigma(\ell_p) = \langle OldElem, \gamma, \chi, \xi_p \rangle$ and $p' = \sigma'(\ell_p)$, it holds that*

$$(p.place.p2 \neq \ell^0_{OldElem}) \vee (p.val \leq p.place.split \wedge p.place.p1 \neq \ell_p) \implies p'.place.done = false$$

Proof. Holds by application of Corollaries 7.18 and 7.19 Note that applying Corollary 7.19 is this simple because only *false* can be written to *place.done*. □

Corollary 8.13 (*checkStable* Reverse Contract)**.** *Let $P = \langle Sc, \sigma, s\chi \rangle$ be an AuDaLa state and let $\mathcal{L}_{NewElem} \subseteq \mathcal{L}$ be the labels of NewElem structs in P and let $\mathcal{L}_{OldElem} \subseteq \mathcal{L}$ be the labels of OldElem structs in P. Executing checkStable starting at state P results in a state $P' = \langle Sc, \sigma', s\chi \rangle$. In P', for every ℓ_q s.t. $q = \sigma(\ell_q) = \langle NewElem, \gamma, \chi, \xi_p \rangle$ and $q' = \sigma'(\ell_q)$, it holds that*

$$q'.done = false \iff (q.p2 \neq \ell^0_{OldElem}) \vee \exists \ell_r \in \mathcal{L}_{OldElem}(\sigma(\ell_r).val \leq q.split \wedge q.p1 \neq \ell_r)$$

Proof. Follows from applying Lemma 8.12 on all old elements from the perspective of the result, which allows us to make the implication a bi-implication. □

Lemma 8.14 (*split* Contract)**.** *Let $P = \langle Sc, \sigma, s\chi \rangle$ be an AuDaLa state and let $\mathcal{L}_{NewElem} \subseteq \mathcal{L} \setminus$ be the labels of NewElem structs in P. Let $\mathcal{L}_P$ be all labels used in P. Executing split starting at state P results in a state $P' = \langle Sc, \sigma', s\chi \rangle$. In P', for every ℓ_p s.t. $p = \sigma(\ell_p) = \langle OldElem, \gamma, \chi, \xi_p \rangle$ and $p' = \sigma'(\ell_p)$, all of the following hold:*

1. *If $\ell_p \in \mathcal{L}_0$, then $p = p'$ and no struct instance is created during the execution of split by the struct instance of ℓ_p.*

2. *Iff $\ell_p \notin \mathcal{L}_0$, all of the following hold:*

   (a) $p'.done = true$.

   (b) *Iff $p.done = false$:*

   i. $p.p1 \neq \ell^0_{OldElem} \wedge p.p2 \neq \ell^0_{OldElem} \iff \exists \ell \notin \mathcal{L}_P.(\sigma'(\ell) = \langle NewElem, \varepsilon, \varepsilon, \xi_\ell \rangle \wedge \xi_\ell = \xi^0_{NewElem}[\{min \mapsto p.spl{+}1, spl \mapsto p.spl{+}(p.max{-}p.spl)/2, max \mapsto p.max, next \mapsto p.next, done \mapsto true\}] \wedge p'.next = \ell) \wedge p'.max = p.spl \wedge p'.hasSplit = true,$

   ii. $p.p1 \neq \ell^0_{OldElem} \wedge p.p2 = \ell^0_{OldElem} \iff p'.max = p.spl \wedge p'.hasSplit = false,$

   iii. $p.p1 = \ell^0_{OldElem} \wedge p.p2 \neq \ell^0_{OldElem} \iff p'.min = p.spl + 1 \wedge p'.hasSplit = false,$

   iv. $p'.spl = p'.min + (p'.max - p'.min)/2 \wedge p'.p1 = \ell^0_{OldElem} \wedge p'.p2 = \ell^0_{OldElem}.$

Proof. As, by Lemma 8.10, *split* does not have any race conditions, point 2 follows from multiple applications of Corollary 7.18. For the first point, note that if $\ell_p \in \mathcal{L}_0$, $p.p1 = \ell^0_{OldElem}$ and $p.p2 = \ell^0_{OldElem}$. Therefore, the struct instance of ℓ_p will not create a new struct instance as per line 27. That $p = p'$ again follows from multiple applications of Corollary 7.18. Therefore, point 1 holds. □

We then consider the schedule. We first fix the state after the first execution of *migrate*:

Lemma 8.15 (Fixpoint premise)**.** *Let P_1 be the premise state and let P_2 be the state after the first execution of migrate. Then in P_2, all of the following hold:*

1. *For every element p_i in $p_1, \ldots, p_n$ there exists a struct instance a_i (in $A = a_1, \ldots a_n$) with label ℓ_i (in $L = \ell_1, \ldots, \ell_n$) s.t. $a_i = \langle OldElem, \varepsilon, \varepsilon, \xi_i \rangle$ and $\sigma(\ell_i) = a_i$.*

2. *There exists a non-null struct instance b with label ℓ_b s.t. $b = \langle newElem, \varepsilon, \varepsilon, \xi_b \rangle$.*

3. *For all a_i, $a_i.val = x_i$ with $x_i \in x_1, \ldots x_n$ and $a_i.place = b$.*

4. *$b.min = \min(x_1, \ldots, x_n)$, $b.max = \max(x_1, \ldots, x_n)$, $\_spl = (b.max - b.min)/2 + b.min$, $b.next = \ell^0_{NewElem}$, $b.hasSplit = false$ and $b.done = true$.*

5. *Let A_1 be the set of $a_i \in A$ s.t. $a_i.val \leq b.spl$, with labels L_1, and let $A_2 = A \setminus A_1$, with labels L_2. Then for all $a_i \in A_1$, $a_i.move = false$, for all $a_i \in A_2$, $a_i.move = true$, if $|A_1| > 0$, $\exists \ell_i \in L_1.(b.p1 = \ell_i)$, and if $|A_2| > 0$, $\exists \ell_i \in L_2.(b.p2 = \ell_i)$.*

Proof. The first three points are directly derived from Definition 8.9. The last point follows from Lemma 8.11, as when $|A_1| > 0$, there is at least one old element for which point 2 of Lemma 8.11 applies, and if $|A_2| > 0$, there is at least one old element for which point 3 of Lemma 8.11 applies. Note that point 1 applies to none of the elements from the premise. □

We then prove the following lemma specifying the effect of a single fixpoint execution:

Lemma 8.16 (Fixpoint execution contract)**.** *Let $P = \langle Sc, \sigma, s\chi \rangle$ be an AuDaLa state, let $\mathcal{L}_{OldElem} \subseteq \mathcal{L}$ be the labels of OldElem structs in P and let $\mathcal{L}_{NewElem} \subseteq \mathcal{L}$ be the labels of NewElem structs in P. Let $\mathcal{L}_P$ be all labels used in P. Executing an iteration of the fixpoint starting at state P results in a state $P' = \langle Sc, \sigma', s\chi \rangle$. In P', for every ℓ_p s.t. $p = \sigma(\ell_p) = \langle OldElem, \gamma, \chi, \xi_p \rangle$ and $p' = \sigma'(\ell_p)$ and ℓ_q s.t. $q = \sigma(\ell_q) = \langle NewElem, \gamma, \chi, \xi_q \rangle$, $q' = \sigma'(\ell_q)$ and $p.place = q$, all of the following hold:*

1. *If $\ell_p \in \mathcal{L}_0$, $p = p'$, and if $\ell_q \in \mathcal{L}_0$, $q = q'$ and no structure instance is created during the execution of split by the struct instance of ℓ_q.*

2. *Iff $\ell_p \notin \mathcal{L}_0$ and $\ell_q \notin \mathcal{L}_0$, all of the following hold:*

(a) $q'.done = true$

(b) *Iff* $(q.p2 \neq \ell^0_{OldElem}) \vee \exists \ell_r \in \mathcal{L}_{OldElem}(\sigma(\ell_r).val \leq q.split \wedge q.p1 \neq \ell_r)$ *all of the following hold:*

i. $q.p1 \neq \ell^0_{OldElem} \wedge q.p2 \neq \ell^0_{OldElem} \iff \exists \ell \notin \mathcal{L}_P.(\sigma'(\ell) = \langle NewElem, \varepsilon, \varepsilon, \xi_\ell \rangle \wedge \xi_\ell = \xi^0_{NewElem}[\{min \mapsto q.spl + 1, spl \mapsto q.spl + 1 + (q.max - (q.spl + 1))/2, max \mapsto q.max, next \mapsto q.next, done \mapsto true\}] \wedge q'.next = \ell) \wedge q'.max = q.spl \wedge q'.hasSplit = true$

ii. $q.p1 \neq \ell^0_{OldElem} \wedge q.p2 = \ell^0_{OldElem} \iff q'.max = q.spl \wedge q'.hasSplit = false,$

iii. $q.p1 = \ell^0_{OldElem} \wedge q.p2 \neq \ell^0_{OldElem} \iff q'.min = q.spl + 1 \wedge q'.hasSplit = false,$

iv. $q'.spl = q'.min + (q'.max - q'.min)/2 \wedge q'.p1 = \ell^0_{OldElem} \wedge q'.p2 = \ell^0_{OldElem}.$

v. $q'.done^*$

(c) $p.move \wedge q'.hasSplit \iff p'.place = q'.next,$

(d) $(p.val > p'.place.spl) \iff p'.move = true,$

(e) $(p.val \leq p'.place.spl) \implies \exists \ell \in \mathcal{L}_{OldElem}.(\sigma'(\ell) = r \wedge r.place = p'.place \wedge p'.place.p1 = \ell),$

(f) $(p.val > p'.place.spl) \implies \exists \ell \in \mathcal{L}_{OldElem}.(\sigma'(\ell) = r \wedge r.place = p'.place \wedge p'.place.p2 = \ell).$

Proof. This is a straightforward combination of the three contracts that make up the contracts for the fixpoint. If $\ell_p \in \mathcal{L}_0$, then all of its parameters are *null*, so none of the updates of ℓ_p will lead to changed parameters. If $\ell_q \in \mathcal{L}_0$, then point 2 holds by Lemma 8.14. Point 2*a* holds by Lemma 8.14. The condition of 2*b* is from Corollary 8.13; as *checkStable* determines whether *done* is *false* and the execution of *split* directly depends on that, the clauses *i-iv* are lifted from *split* and therefore hold due to Lemma 8.14. Clause *v* is a result of the interaction between *checkStable* and *split*; if the condition in 2*b* is true, then *checkStable* will change $q.done$ to *false*, which is later overwritten in *split* with *true*. If the condition is *false*, then $q.done$ will not be changed, but will stay constantly *true*. Clauses 2*c*, 2*d*, 2*e*, and 2*f* hold due to Lemma 8.11. □

We then use this lemma to prove the following lemma for executions of the fixpoint from the fixpoint premise:

Lemma 8.17 (Fixpoint execution properties)**.** *Let $F_{i-1} = \langle Sc, \sigma, s\chi \rangle$ be the state before the ith execution of the fixpoint from P_2 as defined in Lemma 8.15, with $F_0 = P_2$. Let $\mathcal{L}_{OldElem} \subseteq \mathcal{L}$ be the labels of OldElem structs in F_{i-1} and let $\mathcal{L}_{NewElem} \subseteq \mathcal{L}$ be the labels of NewElem structs in F_{i-1}. Let $\mathcal{L}_P$ be all labels used in P. Let ℓ_p be a label of an OldElem struct instance, where $\sigma(\ell_p) = p$. Then the following all hold:*

1. $p.val > p.place.spl \iff p.move = true.$
2. $(p.val \leq p.place.spl) \implies \exists \ell \in \mathcal{L}_{OldElem}.(\sigma(\ell) = r \wedge r.place = p.place \wedge p.place.p1 = \ell),$
3. $(p.val > p.place.spl) \implies \exists \ell \in \mathcal{L}_{OldElem}.(\sigma(\ell) = r \wedge r.place = p.place \wedge p.place.p2 = \ell).$
4. $p.place.spl = p.place.min + \lfloor (p.place.max - p.place.min)/2 \rfloor.$
5. *If* $p.place.next = \ell^0_{NewElem}$, *then* $\nexists \ell_t \in \mathcal{L}_{NewElem}.(\sigma(\ell_t) = t \wedge p.place \neq \ell_t \wedge t.next = \ell^0_{NewElem}).$
6. *If* $p.place.next \neq \ell^0_{NewElem}$, *then* $p.place.max < p.place.next.min.$
7. $p.place.min \leq p.val \leq p.place.max.$

Additionally, let let $F_i = \langle Sc', \sigma', s\chi' \rangle$ be the state after the ith execution of the fixpoint from P_2, with $\sigma'(\ell_p) = p'$. Then all of the the following properties, describing the relation between F_{i-1} and F_i, also hold:

C1 $p'.val = p.val$ *and* $p'.val$ *is stable.*

R1 *If* $\exists \ell_t \in \mathcal{L}_{OldElem} \setminus \{\ell_p\}.(\sigma(\ell_t) = t \wedge t.place = p.place) \vee p.val > p.place.spl$, *then* $p'.place.max - p'.place.min \leq \lceil p.place.max - p.place.min/2 \rceil.$

R2 *If* $\nexists \ell_t \in \mathcal{L}_{OldElem} \setminus \{\ell_p\}.(\sigma(\ell_t) = t \wedge t.place = p.place) \wedge p.val \leq p.place.spl$*, with* $p.place = \ell_q$*, then the struct instances of* ℓ_p *and* ℓ_q *do not make new struct instances in execution* i *and all parameters of* ℓ_p *and* ℓ_q *are stable after execution* i*,* $p'.place = \ell_q$ *and* $q'.p1 = \ell_p$*.*

Proof. First, **C1** directly follows from the fact that $p'.val$ is not assigned a value in the contract in Lemma 8.16. We use this to prove points 1-7 hold for all possible F_i by induction on i:

$i = 0$**.** Then $F_i = P_2$. Point 1, 2 and 3 hold by point 5 of the properties of P_2. Point 4 holds by point 4 of the properties of P_2 as given in Lemma 8.15. As there is only one *NewElem* b s.t. for all possible p, $p.place = b$ in P_2 (point 5), and $b.next = \ell^0_{NewElem}$ in P_2 (point 6), points 5 and 6 of the lemma hold. As $b.min$ is the minimum value of all old elements and $b.max$ is the maximum value of all old elements, point 7 of the lemma also holds.

$i > 0$**.** Then as induction hypothesis, we assume points 1-7 hold for F_{i-1}. Then we prove the points separately:

1. Assume that $p'.val > p'.place.spl$. Then as the *val* parameter never gets written to in Listing 7, $p'.val > p'.place.spl \iff p.val > p'.place.spl$. Then by point 2*d* of Lemma 8.16, $p'.val > p'.place.spl \iff p'.move = true$.
2. We assume that $(p'.val \leq p'.place.spl)$, and prove that $\exists \ell \in \mathcal{L}_{OldElem}.(\sigma'(\ell) = r \wedge r.place = p'.place \wedge p'.place.p1 = \ell)$. From Lemma 8.16, due to the fact that $p'.val = p.val$, we can use point 2*e* directly to establish this point.
3. Proven along the same lines as the above point, only using point 2*f* from Lemma 8.16 instead.
4. To prove that $p'.place.spl = p'.min + \lfloor (p'.max - p'.min)/2 \rfloor$, note that $p'.place$ is either equal to $p.place$ or not. We do a case distinction on these cases. If $p'.place = p.place$, according to point 2*b*-*iv* in Lemma 8.16 and the fact that AuDaLa only does integer calculations (and therefore the result is automatically rounded down), this point holds. If $p'.place \neq p.place$, let $p.place = \ell_q$ and let $q = \sigma(\ell_q)$ and $q' = \sigma'(\ell_q)$. Then as $p'.place \neq p.place$ and as the only possible value is then $p'.place = q'.next$ by point 2*c* of Lemma 8.16, we know from 2*c* that $p.move = true$ and $q'.hasSplit = true$. It then follows by point 2*b*-*i* of Lemma 8.16 that $q'.next.spl = q.spl + 1 + (q.max - (q.spl + 1))/2 = q'.next.min + (q'.next.max - q'.next.min)/2$. Then, as again AuDaLa only does integer calculations, the point also holds in this case.
5. By the induction hypothesis, if $p.place.next = \ell^0_{NewElem}$, then $\nexists \ell_t \in \mathcal{L}_{NewElem}.(\sigma(\ell_t) = t \wedge p.place \neq \ell_t \wedge t.next = \ell^0_{NewElem})$. Then as no *next* parameter is set to *null* in Listing 7, if there are two *NewElem*s in F_i with an empty *next* parameter, then both must be related to $p.place$. Let $p.place = \ell_q$, with $\sigma'(\ell_q) = q'$. Then as the struct instance of ℓ_q can create only one new struct instance in execution i, the *NewElem* struct instances with an empty *next* parameter must be ℓ_q and a struct instance with a label ℓ made by ℓ_q. But then by Lemma 8.16, this can only be $q'.next$, which means that ℓ_q does have a non-*null* value for *next*. Therefore, this point holds.
6. By the induction hypothesis, if $p.place.next \neq \ell^0_{NewElem}$, then $p.place.max < p.place.next.min$. Let $\ell_q = p.place.next$, with $\sigma(\ell_q) = q$ and $\sigma'(\ell_q) = q'$. Then $q.min \leq q'.min$ and $p'.max \leq p.max$, as seen in cases 2*b*-(*i*-*iv*) of Lemma 8.16. It follows that $p'.max < q'.min$ so this point holds.
7. By the induction hypothesis, $p.place.min \leq p.val \leq p.place.max$. Let $p.place = \ell_q$. Let $q = \sigma(\ell_q)$ and $q' = \sigma'(\ell_q)$. Then either $p'.place = \ell_q$ or not. We use a case distinction: If $p'.place = \ell_q$, then if the condition of 2*b* of Lemma 8.16 does not hold for q, $q' = q$, and as $p'.val = p.val$, this point holds. We therefore assume that the condition of 2*b* does hold for q.

   Then either $p.move = true$ or not. In the case where $p.move = true$, we know that $p'.val = p.val \geq q.spl + 1 \geq q'.min$. To prove that $p'.val \leq q'.max$, note that by the induction hypothesis, it follows from $p.val \geq q.spl + 1$ that $q.p2 \neq \ell^0_{OldElem}$ (point 3 of this lemma). As we know that $p.move = true$ but that $p'.place = \ell_q$, it follows that $q'.hasSplit = false$. From this it follows that 2*b*-*iii* holds and 2*b*-*i* and 2*b*-*ii* do not hold, from which we conclude that $q'.max = q.max$.

It follows that $p'.place.min \leq p'.val \leq p'.place.max$. If $p.move = false$, it follows that $p'.val = p.val \leq q.spl \leq q'.max$. We then also know that $q.p1 \neq \ell^0_{OldElem}$ (point 2 in this lemma), from which it follows that either 2*b-i* or 2*b-iii* of Lemma 8.16 can hold for ℓ_p and ℓ_q, but not 2*b-ii*. In both cases, $q'.min = q.min$, from whcih it follows that $p'.place.min \leq p'.val \leq p'.place.max$.

It rests to prove the point if $p'.place \neq \ell_q$. In this case, we know that $p'.place = q'.next$ (the only value other than ℓ_q it can have according to Lemma 8.16), from which it follows that $p.move = true$ and $q'.hasSplit = true$. From $p.move = true$, we get that $p.val \geq q.spl + 1$ (point 1 of this lemma). Then we know that $q.p2 \neq \ell^0_{OldElem}$ (point 3 of this lemma), from which it follows that the condition for 2*b* holds for ℓ_q. As $q'.hasSplit = true$, it follows that 2*b-i* holds for q, from which it follows that $p'.place.min = q.spl + 1$ and $p'.place.max = q.max$. As $p'.val = p.val$, it follows that $p'.place.min \leq p'.val \leq p'.place.max$. As then in all cases $p'.place.min \leq p'.val \leq p'.place.max$, this point holds.

As all points hold, the step case holds.

As the induction holds, points 1-7 hold. We use these points to prove points **R1** and **R2**, for F_{i-1} and F_i:

R1. If $\exists \ell_t \in \mathcal{L}_{OldElem}.(\sigma(\ell_t) = t \wedge t.place = p.place) \vee p.val > p.place.spl$, let $p.place = \ell_q$, with $\sigma(\ell_q) = q$ and $\sigma'(\ell_q) = q'$. Then if $p.val > p.place.spl$, we know from point 3 of this lemma that $q.p2 \neq \ell^0_{OldElem}$ and from point 1 of this lemma that $p.move = true$. Then either 2*b-i* or 2*b-iii* of Lemma 8.16 holds for p, p', q and q'. Regardless of the exact case, it follows that $p'.place.min = q.spl + 1$ and $p'.place.max = q.max$. From point 4 of this lemma, we know that $q.spl = q.min + \lfloor (q.max + q.min)/2 \rfloor$, so

$$\begin{aligned}
&p'.place.max - p'.place.min \\
&= q.max - (q.min + \lfloor (q.max - q.min)/2 \rfloor + 1) \\
&\leq (q.max - q.min) - (\lfloor (q.max - q.min)/2 \rfloor) \\
&= \lceil (q.max - q.min)/2 \rceil \\
&= \lceil (p.place.max - p.place.min)/2 \rceil.
\end{aligned}$$

Then if $p.val \leq p.place.spl$, it holds that $\exists \ell_t \in \mathcal{L}_{OldElem}.(\sigma(\ell_t) = t \wedge t.place = p.place)$, so we know that there are multiple *OldElems* which qualify for $q.p1$. Then the condition of 2*b* holds, and either 2*b-i* holds or 2*b-ii* holds. In both cases it follows that $p'.place.max = q.spl$ and $p'.place.min = q.min$. From point 4 of this lemma, we know that $q.spl = q.min + \lfloor (q.max + q.min)/2 \rfloor$, so

$$\begin{aligned}
&p'.place.max - p'.place.min \\
&= q.spl - q.min \\
&= q.min - q.min + \lfloor (q.max + q.min)/2 \rfloor \\
&\leq \lceil (q.max + q.min)/2 \rceil \\
&= \lceil p.place.max - p.place.min/2 \rceil.
\end{aligned}$$

It follows that **R1** holds.

R2. Assume $\nexists \ell_t \in \mathcal{L}_{OldElem} \setminus \{\ell_p\}.(\sigma(\ell_t) = t \wedge t.place = p.place) \wedge p.val \leq p.place.spl$. The condition holds straightforwardly when $\ell_p \in \mathcal{L}_0$ or $\ell_q \in \mathcal{L}_0$, by point 1 of Lemma 8.16 and the fact that no struct instances are ever created by p during the execution of the fixpoint as per the same lemma. Then if both ℓ_p and ℓ_q are not *null*-labels, as p does not create struct instances and the condition for 2*b* does not hold for q due to the assumption, no struct instances are created.

To prove that the parameters are stable, we also consider Lemma 8.16. As the condition for 2*b* does not hold, we know that only $q.p1$ and $q.p2$ can have changed, through 2*e* and 2*f*. $q'.done$ cannot have changed, as only when the condition of 2*b* holds $q'.done$ can be unstable.

Due to our assumption, we know that $2f$ cannot apply and that $q'.p1$ can only be ℓ_q. Additionally, from point 2 of this lemma and our assumption, we also know that $q.p1 = \ell_p$. Therefore, all parameters of q are stable and $q'.p1 = q.p1 = \ell_p$.

For the parameters of ℓ_p, note that $p.place = \ell_q$. Due to point 1 of this lemma, we know due to the assumption that $p.move = false$, from which it follows by $2c$ of Lemma 8.16 that $p'.place = p.place = \ell_q$. Additionally, from the assumption and $2d$ of Lemma 8.16 it follows that $p'.move = false$ as well. From point $\boldsymbol{C}$ of this lemma, $p.val$ has not changed in the iteration and is stable. Therefore, all of the parameters of ℓ_p are stable in p' and $p'.place = \ell_q$.

Then if $\nexists \ell_t \in \mathcal{L}_{OldElem} \setminus \{\ell_p\}.(\sigma(\ell_t) = t \wedge t.place = p.place) \wedge p.val \leq p.place.spl$, all parameters of ℓ_p and ℓ_q are stable after the execution of the fixpoint iteration, $p'.place = \ell_q$ and $q.p1 = \ell_p$.

With all points proven, the lemma holds. □

We use this lemma to prove the following lemma:

Lemma 8.18 (Fixpoint termination and result)**.** *The execution of the fixpoint starting at P_2 as specified in Lemma 8.15 terminates. Let $P' = \langle Sc, \sigma', s\chi \rangle$ be the state after the execution of the fixpoint. Then the following all hold:*

1. *For every NewElem element $q \in \sigma'$ with label ℓ_q, there is exactly one OldElem element $p \in \sigma'$ with label ℓ_p s.t. $p.place = \ell_q$ and $p.place.min \leq p.val \leq p.place.max$. This element can be referenced by $q.p1$.*
2. *For every NewElem element $q \in \sigma'$ s.t. $q.next \neq \ell^0_{NewElem}$, $q.max < q.next.min$.*
3. *There is exactly one NewElem element $q \in \sigma'$ s.t. $q.next = \ell^0_{NewElem}$.*

Proof. First of all, in any state P with struct environment σ during or after the execution of the fixpoint, there is always at least one *OldElem* $p \in \sigma$ s.t. $p.place = \ell_q$ for every *NewElem* $q \in \sigma$ with label ℓ_q. This is due to new elements only being made if there are *OldElem*s in both $q.p1$ and $q.p2$ as representatives of both halves of the range of q and the range of q being dynamically updated otherwise (Lemma 8.16).

Due to **R1** of Lemma 8.17 We know that if there are multiple elements with ℓ_q as value of their *place* parameter before an iteration of the fixpoint, the range of ℓ_q will halve during the iteration of the fixpoint. As we deal with distinct numbers, after enough halvings of the range, there will always eventually be exactly one *OldElem* that has ℓ_q as its *place*, and due to the dynamic range halving, this element is eventually referenced by the *p1* parameter of ℓ_q. Due to **R2** of Lemma 8.17 it follows that that element and ℓ_q are then stable. As this holds for any ℓ_q, all *OldElems* and *NewElems* will eventually be stable and the fixpoint will terminate.

From this and from point 7 of Lemma 8.17, it also follows that point 1 of this lemma holds. Point 2 follows from point 6 of Lemma 8.17, and point 3 holds from point 5 of Lemma 8.17. Therefore, the entire lemma holds. □

It then follows that:

Theorem 8.19 (Correctness of Listing 7)**.** *When executing the AuDaLa code as given in Listing 7 from a premise state according to Definition 8.9 for a set of elements $p_1, \ldots, p_n$ with distinct positive values $x_1, \ldots, x_n$, it results in a state in which the elements are arranged into a sequence $S_2 = p'_1, \ldots, p'_n$ with values $x'_1, \ldots, x'_n$ s.t. for every $p_i \in S_1$, $p_i \in S_2$ and $x'_i < x'_{i+1}$ for every $1 \leq i \leq n$.*

Proof. The sequence is the sequence of *NewElems*, with the corresponding *OldElem* referenced through parameter *p1*. In this sequence, every old element is represented due to point 1 of Lemma 8.18, the sequence is complete due to point 2 and the sequence is sorted due to point 3 and point 1. □

8.3. Conclusions on Proving AuDaLa programs

In this section, we identify some similarities of the proofs we have presented. The first thing we establish is that we always maintain roughly the following structure, which we expect can be applied to most AuDaLa proofs:

1. Establish the overall premise.
2. Establish the degree of determinism and find out which parameters are not directly deterministic.
3. Establish a contract for each step:
    - Each step is considered separately.
    - The syntax directly corresponds to the form of the step contract.
    - Contracts can be combined by merging the relevant predicates; for example, a contract of a sequence of two steps can be formed by first simplifying the precondition predicate for the second step with the postcondition predicate for the first step and then adding the remaining predicate to the precondition predicate of the first step.
    - Contracts are *relative*: they require a basis element, which is standardly an abstract element executing a step, but which can also be another involved element (as in the case of Corollary 8.13). It is therefore often possible to abstract away from which specific struct instance is executing a step.
4. For every fixpoint sequentially:
    - Establish the premise state before the fixpoint.
    - Make a combined contract for the steps inside the fixpoint.
    - Prove fixpoint termination (if possible).
    - Prove the fixpoint result (which can be combined with the previous step).
5. Relate the result of executing the last step/fixpoint to the overall result, and relate that back to the original premise state.

This gives us some insight in how we can further develop verification systems for AuDaLa. First of all, as step contracts are relative to a basis and do not necessarily need to make assumptions about the context other than the available structs and their parameters, they can be reasoned about locally. They are also *modular*, for as the step contracts do not require context, they can be made separately and combined where necessary. As steps are finite by definition, generating step contracts can be automated. Similarly, as determinism is dependent on the syntax, this can possibly be automated as well. Then, for fixpoints, as they are always dependent on stability of the entire system for termination, we estimate that the termination proof often comes down to proving that there is a monotonic function over all the data which eventually stabilizes. Finding this function also gives insight in the result of the fixpoint.

9. Related Work

Conceptually, our work is related to the Parallel Pointer Machine (PPM) [26, 40], which models memory as a graph that is traversed by processors. AuDaLa differs from this in that processors are implicit and data is the main focus.

The concept of cooperating data elements is present in the Chemical Abstract Machine[41], based on the Γ-language [42, 43]. In the data-autonomous paradigm these components are coordinated by a schedule as opposed to the Chemical Abstract Machine, where the data elements float around freely. By extension, AuDaLa is related to the Γ-Calculus Parallel Programming Framework [7].

The data-autonomous paradigm shares the same focus on data as *message passing* languages like Active Pebbles [8], ParCel-2 [19] and AL-1 [9], but differs in using shared variables instead of synchronisation and messages. It also does not allow the use of data as passive elements, like in the messages of MPI [20].

The *specialist-parallel* approach [44] models a problem as a network of relatively autonomous nodes which perform one specified task. In comparison, the data-autonomous paradigm defines their specialists around data instead of tasks and data elements perform multiple or no tasks depending on their steps.

In AuDaLa, the relations between data elements can be viewed as a graph, which is also the case for *graph based* languages, such as DDG [23], a scheduling language, and GraphGrind [22], a graph partitioning language. The Connection Machine [45] uses a graph-based hardware architecture for parallel computation. Similarly, the way data is expressed in Legion [21] and OP2 [16] is similar to AuDaLa. However, these two languages work top down from a main process that calls functions on data, which is unlike the data-autonomous paradigm.

Since the data-autonomous paradigm extends data-parallelism (see Figure 1), AuDaLa shares concepts with other data-parallel languages like CUDA [17, 24] and OpenCL [18]. It has the most in common with object-oriented approaches to data-parallelism, like the POOL family of languages [6], languages in which small elements do parallel computations based on their neighbours, like ReLaCS [13], PPC [14], Chestnut [12] and the ParCel languages [11, 19], and *actor languages*.

Actor languages, like Ly [10], ParCel-1 [11], POb$C{+}{+}$ [46] and A-NETL [47], treat objects as independent, collaborating actors, in a similar way as how the data-autonomous paradigm treats data. Often, these languages use the *message passing* model to cooperate, which AuDaLa does not. Of those who do not, OpenABL [48] uses agents similar to data elements, but gives the agents to functions instead of functions to agents. *Active Object* languages [49, 50] do give their objects functions, which is very closely related to data elements. The execution of functions in these objects however, is fully asynchronous: objects can activate other objects by calling methods in them for them to execute. This is less structured than in AuDaLa, in which the functions to be executed are defined in the schedule. As a result, AuDaLa does not use futures, unlike most active object languages.

The use of a schedule in the data-autonomous paradigm relates AuDaLa to some more functional data-parallel languages as well, like Halide [15], which uses a schedule as well, and even Futhark [51], in which the manipulation of an array has some similarity to calling a step in AuDaLa. The schedule can be considered as a coordination language [52] for the paradigm and AuDaLa, but is fully integrated and required for both to function. It also does not need to create channels between components, like for example Reo [53].

Similar to our motivation, ICE [54], which is a framework for implementing PRAM algorithms, sets the goal of bridging the gap between algorithms and implementation. However, as ICE is based on a PRAM, it is not data autonomous.

In our approach for proving AuDaLa programs, we take cues from *Hoare logic* [55, 56], which has previously been used and expanded upon to prove programs correct or find correct programs [57, 58, 59, 60, 61, 62, 63] and of which some principles seem to suit AuDaLa particularly well.

Our proofs consist of two parts. The first part, in which we generate step contracts, is akin to *symbolic execution* [64, 65], albeit a variant which does not need to consider loops and needs to include parallelism. Our approach, like *separation logic* [66, 67], has a local focus: contracts are generated relative to their executing elements, consider only the parameters accessed by their executing elements and work with pointers. However, the way this locality is achieved is orthogonal; where separation logic partitions the memory to achieve locality for processes, our approach partitions the process into the steps and achieves the locality by only considering one step at a time.

The second part of the proofs aligns more with the standard practices of the Hoare logic family, where there is a precondition (defined by the initialization), an envisioned postcondition and the proof needs to bridge the gap between them. The difference is that instead of predefined rules (as used by Hoare logic family methods), the rules used here are the step contracts generated in the first part.

In our approach, we make a distinction between write-write race conditions and read-write race conditions, where we consider write-write race conditions benign enough to include and read-write race conditions as harmful. Distinctions between race conditions, whether it is good to have them and methods to find the harmful ones have been discussed before [68, 69].

10. Conclusion

In this paper, we presented the data-autonomous paradigm and introduced it by means of the Autonomous Data Language, by giving examples of standard algorithms and discussing the syntax, type system and semantics. We have given the basics for proving algorithms correct in AuDaLa, and demonstrated their use by proving some examples correct. While we do not prove the safety of our type system in this paper, we suspect that it can be derived from Theorem 7.14. We leave that to future work.

Our contributions contribute to the research on the data-autonomous paradigm. With this paper, we have shown that the data-autonomous paradigm can be simple and verifiable. This combines with earlier work [27], where we show that AuDaLa is Turing Complete, to show that the paradigm is not less expressive than other paradigms. We have also defined a weak memory semantics for AuDaLa in [28], which is used to create a compiler from AuDaLa to CUDA in [29], which allows for feasibly fast parallel execution of AuDaLa programs, demonstrating that the data-autonomous paradigm can be used practically.

One extension which we think is quite important for the practical use of AuDaLa is an extension to allow inheritance and packages in AuDaLa. Currently AuDaLa does not support these, which can lead to quite large struct definitions which have implementations of multiple different algorithms and functions in them. These algorithms also need to be reimplemented for every new struct type. With inheritance, it would be easier to make multiple subtypes of structs, which would ease this problem, while packages would allow structs to refer to different files for implementations. Note that these packages would need to also define a set of parameters required to be able to use the implementation in them.

For another practical extension, note that currently AuDaLa is focused on design of programs and not on the debugging of programs. It does not give error messages or halt when the given program does unexpected things which may not have been the intention of the programmer (such as have data races or write to *null*). The current view is that the programmer should understand what the program does from its design, not its effects. However, an extension of AuDaLa could be the inclusion of error messages and possible interruptions when a possibly unintended behaviour happens. Note that this will have an impact on Theorem 7.14.

With the strong modularity of the steps and the structure of the proofs given in Section 8, we were strongly reminded of Hoare Logic [55] and Concurrent Separation Logic [66, 67]. In the future, we intend to formalize a proof method for AuDaLa using, or based on, a suitable Hoare Logic or CSL, using the insights gained from the lemmas of Section 7 and the proofs of Section 8.

In other future work, we can extend on AuDaLa by finding and implementing algorithms to reduce the manual work it takes to generate AuDaLa step contracts. We can also utilize AuDaLa's commands to create data-race finding methods or extend AuDaLa. It would also be interesting to mechanize some of the proofs in a proving framework. Lastly, we can also look further into which domains of programs are particularly well expressed in AuDaLa, and compare AuDaLa to languages of that domain.

Acknowledgements. We, the authors, would like to give thanks to the reviewers, and one reviewer in particular, whose engaged and in-depth commentary throughout the revision process has spurred us to reach new heights in this paper. We greatly enjoyed the correspondence and your comments were greatly appreciated. Thank you very much.

Funding. Thomas Neele is supported by NWO grant VI.Veni.232.224.

References

[1] C. E. Leiserson, et al., There's plenty of room at the Top: What will drive computer performance after Moore's law?, Science 368 (6495) (2020) eaam9744. doi:10.1126/science.aam9744.

[2] F. Ciccozzi, et al., A Comprehensive Exploration of Languages for Parallel Computing, ACM Comput. Surv. 55 (2) (2022) 24:1–24:39. doi:10.1145/3485008.

[3] M. B. Giles, I. Reguly, Trends in high-performance computing for engineering calculations, Phil. Trans. R. Soc. A. 372 (2022) (2014). doi:10.1098/rsta.2013.0319.

[4] A. Geist, D. A. Reed, A survey of high-performance computing scaling challenges, Int. J. High Perform. Comput. Appl. 31 (1) (2017) 104–113. doi:10.1177/1094342015597083.

[5] T. T. P. Franken, T. Neele, J. F. Groote, An Autonomous Data Language, in: ICTAC 2023 Proc., Vol. 14446 of LNCS, Springer Nature Switzerland, 2023, pp. 158–177. doi:10.1007/978-3-031-47963-2_11.
[6] P. America, F. van der Linden, A parallel object-oriented language with inheritance and subtyping, SIGPLAN Not. 25 (10) (1990) 161–168. doi:10.1145/97946.97966.
[7] S. Gannouni, A Gamma-calculus GPU-based parallel programming framework, in: WSWAN Proc., IEEE, 2015, pp. 1–4. doi:10.1109/WSWAN.2015.7210299.
[8] J. J. Willcock, T. Hoefler, N. G. Edmonds, A. Lumsdaine, Active pebbles: parallel programming for data-driven applications, in: ICS Proc., ACM, 2011, p. 235. doi:10.1145/1995896.1995934.
[9] A. Marcoux, C. Maurel, P. Salle, AL 1: a language for distributed applications, in: FTDCS1990 Workshop Proc., IEEE, 1988, pp. 270–276. doi:10.1109/FTDCS.1988.26707.
[10] D. Ungar, S. S. Adams, Harnessing emergence for manycore programming: early experience integrating ensembles, adverbs, and object-based inheritance, in: OOPSLA Proc., ACM, 2010, pp. 19–26. doi:10.1145/1869542.1869546.
[11] S. Vialle, T. Cornu, Y. Lallement, ParCeL-1: a parallel programming language based on autonomous and synchronous actors, SIGPLAN Not. 31 (8) (1996) 43–51. doi:10.1145/242903.242945.
[12] A. Stromme, R. Carlson, T. Newhall, Chestnut: a GPU programming language for non-experts, in: PMAM Proc., ACM, 2012, pp. 156–167. doi:10.1145/2141702.2141720.
[13] F. Raimbault, D. Lavenier, RELACS for systolic programming, in: ASAP Proc., IEEE, 1993, pp. 132–135. doi:10.1109/ASAP.1993.397128.
[14] M. Maresca, P. Baglietto, A programming model for reconfigurable mesh based parallel computers, in: PMMPC Workshop Proc., IEEE, 1993, pp. 124–133. doi:10.1109/PMMP.1993.315547.
[15] J. Ragan-Kelley, et al., Halide: decoupling algorithms from schedules for high-performance image processing, Commun. ACM 61 (2017) 106–115. doi:10.1145/3150211.
[16] G. Mudalige, M. Giles, I. Reguly, C. Bertolli, P. Kelly, OP2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures, in: InPar 2012, 2012, pp. 1–12. doi:10.1109/InPar.2012.6339594.
[17] M. Garland, et al., Parallel Computing Experiences with CUDA, IEEE Micro 28 (4) (2008) 13–27. doi:10.1109/MM.2008.57.
[18] N. Chong, A. F. Donaldson, J. Ketema, A sound and complete abstraction for reasoning about parallel prefix sums, SIGPLAN Not. 49 (1) (2014) 397–409. doi:10.1145/2578855.2535882.
[19] P.-J. Cagnard, The ParCeL-2 Programming Language, in: Euro-Par 2000, Vol. 1900 of LNCS, Springer, 2000, pp. 767–770. doi:10.1007/3-540-44520-X_106.
[20] L. Clarke, I. Glendinning, R. Hempel, The MPI Message Passing Interface Standard, in: Programming Environments for Massively Parallel Distributed Systems, Monte Verità, Birkhäuser, 1994, pp. 213–218. doi:10.1007/978-3-0348-8534-8_21.
[21] M. Bauer, S. Treichler, E. Slaughter, A. Aiken, Legion: Expressing locality and independence with logical regions, in: SC '12, 2012, pp. 1–11. doi:10.1109/SC.2012.71.
[22] J. Sun, H. Vandierendonck, D. S. Nikolopoulos, GraphGrind: addressing load imbalance of graph partitioning, in: ICS Proc., ACM, 2017, pp. 1–10. doi:10.1145/3079079.3079097.
[23] V. Tran, L. Hluchy, G. Nguyen, Parallel programming with data driven model, in: EMPDP Proc., IEEE, 2000, pp. 205–211. doi:10.1109/EMPDP.2000.823413.
[24] M. Harris, S. Sengupta, J. D. Owens, Parallel Prefix Sum (Scan) with CUDA, GPU gems 3 (39) (2007) 851–876.
[25] S. Fortune, J. Wyllie, Parallelism in random access machines, in: Proceedings of the tenth annual ACM symposium on Theory of computing, STOC '78, Association for Computing Machinery, 1978, pp. 114–118. doi:10.1145/800133.804339.
[26] S. A. Cook, P. W. Dymond, Parallel pointer machines, computational complexity 3 (1) (1993) 19–30. doi:10.1007/BF01200405.
[27] T. T. P. Franken, T. Neele, AuDaLa is Turing Complete, in: FORTE 2024 Proc., Vol. 14678 of LNCS, Springer Nature Switzerland, 2024, pp. 221–229. doi:10.1007/978-3-031-62645-6_12.
[28] G. P. Leemrijse, T. T. P. Franken, T. Neele, Formalisation of a new weak Semantics for AuDaLa, in: ATVA 2024 Proc., LNCS, Springer International Publishing, 2024, (Accepted paper, proceedings to be published).
[29] G. Leemrijse, Towards relaxed memory semantics for the Autonomous Data Language, MSc. thesis, Eindhoven University of Technology (2023).
[30] W. D. Hillis, G. L. Steele, Data parallel algorithms, Commun. ACM 29 (12) (1986) 1170–1183. doi:10.1145/7902.7903.
[31] B. C. Pierce, Types and programming languages, MIT press, 2002.
[32] L. Cardelli, Type systems, ACM Comput. Surv. 28 (1) (1996) 263–264. doi:10.1145/234313.234418.
[33] L. Cardelli, Type systems, in: A. B. Tucker (Ed.), Computer Science Handbook, Second Edition, Chapman & Hall/CRC, 2004, Ch. 97, accessed through http://lucacardelli.name/indexPapers.html.
[34] J. C. Mitchell, CHAPTER 8 - Type Systems for Programming Languages, in: J. Van leeuwen (Ed.), Formal Models and Semantics, Handbook of Theoretical Computer Science, Elsevier, Amsterdam, 1990, pp. 365–458. doi:10.1016/B978-0-444-88074-1.50013-5.
[35] R. Cole, Parallel Merge Sort, SIAM J. Comput. 17 (1988) 770–785. doi:10.1137/0217049.
[36] M. Hoffman, Visibility graphs and 3-sum, https://ti.inf.ethz.ch/ew/courses/CG09/materials/v12.pdf, accessed: 2025-03-11.
[37] A. Gajentaan, M. H. Overmars, On a class of $O(n2)$ problems in computational geometry, Computational Geometry 5 (3) (1995) 165–185. doi:10.1016/0925-7721(95)00022-2.
[38] R. Milner, A theory of type polymorphism in programming, Journal of Computer and System Sciences 17 (3) (1978)

348–375. doi:10.1016/0022-0000(78)90014-4.
URL https://www.sciencedirect.com/science/article/pii/0022000078900144
[39] A. K. Wright, M. Felleisen, A Syntactic Approach to Type Soundness, Information and Computation 115 (1) (1994) 38–94. doi:10.1006/inco.1994.1093.
URL https://www.sciencedirect.com/science/article/pii/S0890540184710935
[40] M. T. Goodrich, S. R. Kosaraju, Sorting on a parallel pointer machine with applications to set expression evaluation, J. ACM 43 (2) (1996) 331–361. doi:10.1145/226643.226670.
[41] G. Berry, G. Boudol, The chemical abstract machine, Theor. Comput. Sci. 96 (1) (1992) 217–248. doi:10.1016/0304-3975(92)90185-I.
[42] J.-P. Banâtre, D. Le Métayer, The gamma model and its discipline of programming, Science of Computer Programming 15 (1) (1990) 55–77. doi:10.1016/0167-6423(90)90044-E.
[43] J.-P. Banâtre, A. Coutant, D. Le Metayer, A parallel machine for multiset transformation and its programming style, Future Generation Computer Systems 4 (2) (1988) 133–144. doi:10.1016/0167-739X(88)90012-X.
[44] N. Carriero, D. Gelernter, How to write parallel programs: a guide to the perplexed, ACM Comput. Surv. 21 (3) (1989) 323–357. doi:10.1145/72551.72553.
[45] W. D. Hillis, The connection machine, MIT Press, Cambridge, Mass, 1989.
[46] E. G. Pinho, F. H. de Carvalho, An object-oriented parallel programming language for distributed-memory parallel computing platforms, Science of Computer Programming 80 (2014) 65–90. doi:10.1016/j.scico.2013.03.014.
[47] T. Baba, T. Yoshinaga, A-NETL: a language for massively parallel object-oriented computing, in: PMMPC Proc., IEEE, 1995, pp. 98–105. doi:10.1109/PMMPC.1995.504346.
[48] B. Cosenza, et al., OpenABL: A Domain-Specific Language for Parallel and Distributed Agent-Based Simulations, in: Euro-Par 2018, Vol. 11014 of LNCS, Springer, 2018, pp. 505–518. doi:10.1007/978-3-319-96983-1_36.
[49] F. S. de Boer, D. Clarke, E. B. Johnsen, A Complete Guide to the Future, in: Programming Languages and Systems, Vol. 4421 of LNCS, Springer, 2007, pp. 316–330. doi:10.1007/978-3-540-71316-6_22.
[50] F. de Boer, et al., A Survey of Active Object Languages, ACM Comput. Surv. 50 (5) (2017) 76:1–76:39. doi:10.1145/3122848.
[51] T. Henriksen, et al., Futhark: purely functional GPU-programming with nested parallelism and in-place array updates, in: PLDI 2017, PLDI 2017, ACM, 2017, pp. 556–571. doi:10.1145/3062341.3062354.
[52] F. Arbab, P. Ciancarini, C. Hankin, Coordination languages for parallel programming, Parallel Computing 24 (7) (1998) 989–1004. doi:10.1016/S0167-8191(98)00039-8.
[53] F. Arbab, Reo: a channel-based coordination model for component composition, Mathematical Structures in Computer Science 14 (3) (2004) 329–366. doi:10.1017/S0960129504004153.
[54] F. Ghanim, U. Vishkin, R. Barua, Easy PRAM-Based High-Performance Parallel Programming with ICE, IEEE Trans. Parallel Distrib. Syst. 29 (2) (2018) 377–390. doi:10.1109/TPDS.2017.2754376.
[55] C. A. R. Hoare, An axiomatic basis for computer programming, Communications of the ACM 12 (10) (1969) 576–580. doi:10.1145/363235.363259.
[56] K. R. Apt, E.-R. Olderog, Assessing the Success and Impact of Hoare's Logic, in: Theories of Programming: The Life and Works of Tony Hoare, Association for Computing Machinery, 2021, pp. 41–76. doi:10.1145/3477355.3477359.
[57] S. Owicki, D. Gries, Verifying properties of parallel programs: an axiomatic approach, Communications of the ACM (1976) 279–285doi:10.1145/360051.360224.
[58] S. Owicki, D. Gries, An axiomatic proof technique for parallel programs I, Acta Informatica (1976) 319–340doi:10.1007/BF00268134.
[59] W. H. J. Feijen, A. J. M. v. Gasteren, On a Method of Multiprogramming, Springer Science & Business Media, 2013.
[60] E. de Vries, V. Koutavas, Reverse Hoare Logic, in: Software Engineering and Formal Methods, Vol. 7041 of LNPSE, Springer, 2011, pp. 155–171. doi:10.1007/978-3-642-24690-6_12.
[61] M. O. Myreen, A. C. J. Fox, M. J. C. Gordon, Hoare Logic for ARM Machine Code, in: International Symposium on Fundamentals of Software Engineering, Vol. 4767 of LNPSE, Springer, 2007, pp. 272–286. doi:10.1007/978-3-540-75698-9_18.
[62] C. Stirling, A generalization of Owicki-Gries's Hoare logic for a concurrent while language, Theoretical Computer Science (1988) 347–359doi:10.1016/0304-3975(88)90033-3.
[63] L. Lamport, The 'Hoare logic' of concurrent programs, Acta Informatica (1980) 21–37doi:10.1007/BF00289062.
[64] J. C. King, Symbolic execution and program testing, Communications of the ACM (1976). doi:10.1145/360248.360252.
[65] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, I. Finocchi, A Survey of Symbolic Execution Techniques, ACM Computing Surveys (2018) 50:1–50:39doi:10.1145/3182657.
[66] S. Brookes, P. W. O'Hearn, Concurrent separation logic, ACM SIGLOG News (2016) 47–65doi:10.1145/2984450.2984457.
URL https://dl.acm.org/doi/10.1145/2984450.2984457
[67] P. O'Hearn, Separation logic, Communications of the ACM (2019) 86–95doi:10.1145/3211968.
[68] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, B. Calder, Automatically classifying benign and harmful data races using replay analysis, in: PLDI 2007 Proc., Association for Computing Machinery, 2007, pp. 22–31. doi:10.1145/1250734.1250738.
[69] H.-J. Boehm, How to miscompile programs with "benign" data races, in: USENIX 2011 Proc., USENIX Association, 2011.