Radboud University Nijmegen

Faculty of Science

Term Rewriting with Shared Memory

Research Internship Project

Author:
Mikhail Ushakov
s1080976

Supervisor:
Dr. Cynthia Kop

June 2026

Contents

Chapter 1

Introduction

Concurrent data structures play a fundamental role in modern computer systems and software development. They form a foundation of performance-critical applications such as database management systems, web servers, operating systems, parallel sorting algorithms, and large-scale graph processing frameworks. As hardware architectures increasingly rely on multicore processors, the ability to safely and efficiently access shared data in parallel has become a central concern in both academic research and industry. While concurrency enables significant performance improvements, it also introduces extra complexity to the system. Shared-memory access and modification can lead to subtle errors such as data races, deadlocks, and inconsistent states. These issues are especially hard to reason about using only traditional testing techniques: most of the concurrency bugs arise irregularly, only under specific execution interleavings. Because of this, concurrent software can be considered one of the most error-prone categories of software, requiring the development of formal models and verification techniques.

Term Rewriting Systems (TRS) provide a prominent mathematical framework for describing computation through rule-based transformations. This framework has been successfully applied to a wide range of problems, including program termination analysis, confluence, equivalence checking, and even formal verification of imperative programs. Extensions such as Logically Constrained Term Rewriting Systems (LCTRS) [21] further enhance the expressive power of term rewriting by integrating logical constraints and interpreted theories, enabling more realistic and effective modelling of programs. Despite the discussed advantages, traditional term rewriting frameworks offer limited support for modelling shared mutable state. While prior work has explored parallel rewriting and the representation of global system states in structured terms, a unified approach that combines parallel rewriting semantics with explicit shared-memory operations remains largely unexplored. This knowledge gap limits the applicability of term rewriting to the analysis and modelling of modern concurrent algorithms and data structures.

The goal of this project is to address the limitations of traditional rewriting by introducing Memory Term Rewriting Systems (MemTRS) - an extension of LCTRS that incorporates a global mutable memory state and explicit read, write, and "compare-and-swap" operations. MemTRS is designed to capture the essential aspects of shared-memory concurrency and to preserve the formal structure of term rewriting and the capability for formal analysis. By defining the parallel reduction relation over terms paired with memory states, MemTRS can provide a framework for reasoning about concurrent computations, memory effects, and nondeterminism arising from parallel execution. This research project investigates both the theoretical foundations and the practical applicability of MemTRS. On the theoretical side, we formalize the syntax and semantics of rewriting with shared memory, define the parallel reduction relations and strategies, and analyze the potential sources of nondeterminism such as data races. And on the practical side, we demonstrate how common data structures such as arrays, matrices, and concurrent queues can be modelled in MemTRS. Finally, we present a practical implementation of MemTRS based on a *Cora* [22] fork, a standard library of MemTRS encodings, and a REPL for experiments. We evaluate the behavior of this implementation in multiple case studies. In this work, we aim to assess whether Memory Term Rewriting Systems can serve as a formal model for concurrent computation with shared memory and whether this new kind of system offers advantages over current term rewriting approaches.

Chapter 2

Preliminaries

In the preliminaries' section we will introduce the Logically Constrained Term Rewriting Systems (LCTRSs). We will use these systems as a base framework for the new Term Rewriting Systems with shared memory we want to present in this paper. We base the definitions for LCTRSs on the work by Kop and Nishida [21].

A Logically Constrained TRS is a Term Rewriting System, where certain terms carry an implicit meaning, and each rule is equipped with a logical constraint. Each term in LCTRS has a sort restriction.

2.1 Alphabet

The alphabet $\Sigma \cup \mathcal{V}$ of a Logically Constrained TRS consists of the following:

1. A countably infinite set of pairs $\mathcal{V}$, where each pair $x : \iota$ is a variable equipped with a sort.
2. A non-empty set of function symbols coupled with type declarations $\Sigma = \Sigma_{\text{terms}} \cup \Sigma_{\text{theory}}$. We refer to a function symbol with type declaration stored in Σ as: $f : [\iota_1 \times \cdots \times \iota_n] \Rightarrow \kappa$ (here $\iota_1, \ldots, \iota_n$ are types of function arguments, and κ is the output type of the function f). This division of the function symbol set also allows us to specify the special subset of sorts: $\mathcal{S}_{\text{theory}} \subseteq \mathcal{S}$. This subset consists of all the sorts occurring in the function symbol signatures in the set Σ_{theory}. Also, there exists a special function $\mathcal{I}$ that assigns a set to every sort $\iota \in \mathcal{S}_{\text{theory}}$.

   Additionally, there exists a special function $\mathcal{J}$ that maps each $f : [\iota_1 \times \cdots \times \iota_n] \Rightarrow \kappa \in \Sigma_{\text{theory}}$ to a function $\mathcal{J}_f : \mathcal{I}_{\iota_1} \times \cdots \times \mathcal{I}_{\iota_n} \to \mathcal{I}_\kappa$.

   For each sort $\iota \in \mathcal{S}_{\text{theory}}$ we fix a special subset $\mathcal{V}al_\iota \subseteq \Sigma_{\text{theory}}$. These subsets contain values associated with each sort available for the LCTRS. The set of all values can be described as:

$$\mathcal{V}al = \bigcup_{\iota \in \mathcal{S}_{\text{theory}}} \mathcal{V}al_\iota$$

   For each element $(a : [\,] \Rightarrow \iota) \in \mathcal{V}al_\iota$, the function $\mathcal{J}$ gives one-to-one mapping from $\mathcal{V}al_\iota$ to $\mathcal{I}_\iota$.

2.2 Terms

The set of terms $Terms(\Sigma \cup \mathcal{V})$, where $\Sigma = \Sigma_{\text{terms}} \cup \Sigma_{\text{theory}}$ consists of all possible terms in the Logically Constrained TRS. In general, a term is an expression s that is a combination of the function symbols (and constants) from the set Σ, and variables from the set $\mathcal{V}$. This term s is built such that $\Gamma \vdash s : \iota$ can be derived for some environment Γ, and sort ι, using the inference rules:

$$\frac{}{\Gamma \cup x : \iota \vdash x : \iota}$$

$$\frac{\Gamma \vdash s_1 : \iota_1 \ldots \Gamma \vdash s_n : \iota_n \quad f : [\iota_1 \times \cdots \times \iota_n] \Rightarrow \kappa \in \Sigma}{\Gamma \vdash f(s_1, \ldots, s_n) : \kappa}$$

Additionally for the term s we define a set $Var(s)$. This set consists of all variables occurring in the term s. We call a term s ground if $Var(s) = \emptyset$.

Similarly to the distinction between the different types of function symbols (Σ_{terms} and Σ_{theory}), there exists a similar distinction between the terms:

- We call a term in $Terms(\Sigma_{\text{theory}} \cup \mathcal{V})$ a logical term. These are the terms that are used to define a function or constraint in the model.
- And we call a term in $Terms(\Sigma_{\text{terms}} \cup \mathcal{V})$ a proper term. These are the terms that are used to describe objects that we want to rewrite.

As we discussed in the Alphabet section 2.1, for every function symbol from the theory set Σ_{theory} we define the interpretation mapping $\mathcal{J}$. This mapping can be extended to an interpretation on the ground terms in $\mathcal{T}erms(\Sigma \cup \mathcal{V})$ in the following way:

$$[\![f(s_1, \ldots, s_n)]\!]_{\mathcal{J}} = \mathcal{J}_f([\![s_1]\!]_{\mathcal{J}}, \ldots, [\![s_n]\!]_{\mathcal{J}})$$

Additionally in the alphabet section we defined the set of values $\mathcal{V}al \subseteq \Sigma_{\text{theory}}$. We identify a value c with the logical term $c()$.

A logical ground term s has a value t if it is a value such that $[\![s]\!] = [\![t]\!]$. Because ground terms by definition do not introduce any variables, we can always "evaluate" the logical ground term s and obtain the value t using appropriate interpretations. Every ground logical term has a unique value.

Finally, we introduce the notion of the logical constraint: it is a logical term of some sort *bool*, such that $\mathcal{I}_{bool} = \mathbb{B}$. And usually $\mathcal{V}al_{bool} = \{\texttt{true}, \texttt{false}\}$.

There exists a notion of the validity of the logical constraint:

- We say that the ground logical constraint s is valid if $[\![s]\!]_{\mathcal{J}} = \top$.
- We say that the non-ground logical constraint s is valid if $\forall\gamma$, where γ is a substitution that maps variables in $\mathcal{V}ar(s)$ to a value: $s\gamma$ is valid.

2.3 Substitution

A substitution σ is a sort-preserving mapping: $\mathcal{V} \to \mathcal{T}erms(\Sigma \cup \mathcal{V})$ (mapping from variables to terms).

There exists an operation of applying the substitution to the term. It can be defined inductively:

- For variables: $x\sigma = \sigma(x)$
- For function symbols: $f(t_1, \ldots, t_n)\sigma = f(t_1\sigma, \ldots, t_n\sigma)$

The result of applying the substitution $\gamma = \{x_1 : \iota_1 \mapsto t_1, x_2 : \iota_2 \mapsto t_2, \ldots\}$ to the term t is the term t with all occurrences of any x_i replaced by the corresponding t_i (described in the substitution).

2.4 Contexts

The context is a special term C with zero or more special variables $\Box_1, \ldots \Box_n$ each occurring once. If this context term C can be derived for the environment $\Gamma \cup \{\Box_1 : \iota_1, \ldots, \Box_n : \iota_n\}$ and the sort κ, and if it holds that $\forall i \in \{1, \ldots, n\} : \Gamma \vdash s_i : \iota_i$, then we define term $C[s_1, \ldots, s_n]$ as a term C with each $\Box_i$ replaced by the corresponding s_i.

2.5 Reduction Rules

A rule in the Logically Constrained TRS is a tuple $l \to r\ [\varphi]$, where l and r are terms and φ is a logical constraint. We have a restriction for the term l: it must have the form $f(l_1, \ldots, l_n)$, where $f \in \Sigma_{\text{terms}} \setminus \Sigma_{\text{theory}}$. Also, the terms l and r must be of the same sort. And for every $x : \iota \in Var(r) \setminus Var(l)$, it holds that $\iota \in \mathcal{S}_{\text{theory}}$.

A substitution γ respects the rule $l \to r\ [\varphi]$ if:

1. $Dom(\gamma) = Var(l) \cup Var(r) \cup Var(\varphi)$.
2. $\gamma(x)$ is a value $\forall x \in Var(\varphi) \cup (Var(r) \setminus Var(l))$.
3. $\varphi\gamma$ is valid (the constraint is satisfied).

Notion of the rewrite relation $\to_{\mathcal{R}}$ for the set of rules $\mathcal{R}$: it is a relation on terms, that is defined as the union of two other relations $\to_{\texttt{rule}}$ and $\to_{\texttt{calc}}$, where:

- $l\gamma \to_{\texttt{rule}} r\gamma$ if $l \to r\ [\varphi] \in \mathcal{R}$, and γ respects $l \to r\ [\varphi]$.
- $f(s_1, \ldots, s_n) \to_{\texttt{calc}} v$ if $f \in \Sigma_{\text{theory}} \setminus \Sigma_{\text{terms}}$, all s_i are values, and v is the value of $f(s_1, \ldots, s_n)$.
- If $\exists i : (t_i \to_{\texttt{rule}} u_i) \wedge (\forall j \neq i : t_j = u_j)$, then: $f(t_1, \ldots, t_n) \to_{\texttt{rule}} f(u_1, \ldots, u_n)$.
- If $\exists i : (t_i \to_{\texttt{calc}} u_i) \wedge (\forall j \neq i : t_j = u_j)$, then: $f(t_1, \ldots, t_n) \to_{\texttt{calc}} f(u_1, \ldots, u_n)$.

A logically constrained term rewriting system (LCTRS) is defined as the pair:

$$(\mathcal{T}erms(\Sigma \cup \mathcal{V}), \to_{\mathcal{R}})$$

To improve readability, we will omit trivial constraints in the rewriting systems presented later in the report. That is, a rule written as $l \to r$ should be read as shorthand for the constrained rule $l \to r\ [\texttt{true}]$. This convention is purely notational and is not intended to change the semantics of the rewriting relation. Non-trivial constraints will always be written explicitly.

Chapter 3

Related work

3.1 Parallel Term Rewriting

Different research groups have previously studied the idea of parallelism in the context of Term Rewriting. Alouini and Kirchner [1] studied the possibility of implementing the conditional concurrent rewriting on distributed memory machines. The researchers developed a transformation from conditional term rewriting to unconditional term rewriting. The resulting system is a terminating rewrite system that simulates the original conditional system in a correct and complete way. Also, the transformation function is designed to preserve the inherent concurrency of the initial system as much as possible. The authors believe that the introduced technologies and ideas can be used to improve the efficiency of term normalization procedures in concurrent deduction.

Example 1. Consider the following Conditional Term Rewriting System (CTRS) describing a sorting algorithm on lists:

$$\mathcal{R} = \begin{cases} l_{1,1} : \mathbf{ins}(x, \mathbf{c}(y,z)) \to \mathbf{c}(x, \mathbf{c}(y,z)) & \text{if } x \leq y \downarrow^{!} \mathtt{tt} \\ l_{2,1} : \mathbf{ins}(x, \mathbf{c}(y,z)) \to \mathbf{c}(y, \mathbf{ins}(x,z)) & \text{if } x \leq y \downarrow^{!} \mathtt{ff} \\ l_{3,1} : \mathbf{ins}(x, \mathtt{nil}) \to \mathbf{c}(x, \mathtt{nil}) \\ l_{1,2} : \mathbf{isort}(\mathtt{nil}) \to \mathtt{nil} \\ l_{2,2} : \mathbf{isort}(\mathbf{c}(x,y)) \to \mathbf{ins}(x, \mathbf{isort}(y)) \\ l_{1,3} : \mathtt{0} \leq x \to \mathtt{tt} \\ l_{2,3} : \mathbf{s}(x) \leq \mathtt{0} \to \mathtt{ff} \\ l_{3,3} : \mathbf{s}(x) \leq \mathbf{s}(y) \to x \leq y \end{cases}$$

The transformation of the CTRS $\mathcal{R}$ will result in the following system:

$$\mathcal{R}'' = \begin{cases} l^1_{1,1} : \mathbf{d^0_1}(\mathbf{ins}(\mathbf{d^1_{y'_2}}(\mathbf{c}(\mathbf{d^1_{y'_4}}(z))))) & \to \mathbf{d^0_1}(\mathbf{ins_1}(\mathbf{d^1_{y'_1}}(x)(\mathbf{d^1_{y'_2}}(\mathbf{c}(\mathbf{d^1_{y'_3}}(y), \mathbf{d^1_{y'_4}}, \mathbf{d^1_{y'_5}}(z))))), \\ & \quad \mathbf{d^0_1}(\downarrow(\mathbf{d^1_{y'_1}}(x) \leq \mathbf{d^1_{y'_3}}(y), \mathbf{d^0_1}(\mathtt{tt})), \mathbf{c}(\mathbf{d^1_{y'_3}}(y), \mathbf{d^1_{y'_5}}(z)))) \\ l^2_{1,1} : \mathbf{d^0_x}(\mathbf{ins}(\mathbf{d^{y_1}_{y'_1}}(x)(\mathbf{d^{y_3}_{y'_3}}(y), \mathbf{d^{y_4}_{y'_4}}(z)))) & \to \mathbf{d^0_x}(\mathbf{ins_1}(\mathbf{d^{y_1}_{y'_1}}(x)(\mathbf{d^{y_2}_{y'_2}}(\mathbf{c}(\mathbf{d^{y_3}_{y'_3}}(y), \mathbf{d^{y_4}_{y'_4}}, \mathbf{d^{y_5}_{y'_5}}(z))))), \\ & \quad \mathbf{d^0_1}(\downarrow(\mathbf{d^{y_1}_{y'_1}}(x) \leq \mathbf{d^{y_3}_{y'_3}}(y), \mathbf{d^0_1}(\mathtt{tt}))), \mathbf{c}(\mathbf{d^{y_1}_{y'_1}}(x), \mathbf{c}(\mathbf{d^{y_3}_{y'_3}}(y), \mathbf{d^{y_5}_{y'_5}}(z)))) \\ l^3_{1,1} : \mathbf{d^0_x}(\mathbf{ins_1}(\mathbf{d^{y_1}_{y'_1}}(x), \mathtt{True}, y)) & \to \mathbf{\Phi_{intro}}(y) \\ l^4_{1,1} : \mathbf{d^0_x}(\mathbf{ins_1}(\mathbf{d^1_{y'_1}}(x), \mathtt{False}, y)) & \to \mathbf{d^0_{x+1}}(\mathbf{ins}(\mathbf{d^1_{y'_1}}(x))) \\ l^5_{1,1} : \mathbf{d^0_x}(\mathbf{ins_1}(\mathbf{d^{y_1}_{y'_1}}(x), \mathtt{False}, y)) & \to \mathbf{d^0_x}(\mathbf{ins}(\mathbf{d^{y_1}_{y'_1}}(x))) \\ l^6_{3,1} : \mathbf{d^0_x}(\mathbf{ins_1}(\mathbf{d^1_{y'_1}}(x))) & \to \mathbf{d^1_x}(\mathbf{ins}(\mathbf{d^1_{y'_1}}(x))) \\ l^7_{3,1} : \mathbf{\Phi_{intro}}(\mathbf{ins}(x)) & \to \mathbf{d^0_1}(\mathbf{ins}(\mathbf{\Phi_{intro}}(x))) \\ l^8_{3,1} : \mathbf{\Phi_{intro}}(\mathbf{d^y_{y'}}(x)) & \to \mathbf{d^y_{y'}}(x) \\ l^1_{1,10} : \mathbf{d^0_1}(\downarrow(\mathbf{d^1_z}(x), \mathbf{d^1_{z'}})) & \to \mathtt{True} \\ l^2_{1,10} : \mathbf{d^0_1}(\downarrow(\mathbf{d^1_z}(x), \mathbf{d^1_{z'}}(y))) & \to \mathtt{False} \\ & \vdots \end{cases}$$

The generated Unconditional Term Rewriting System allows for the parallel reduction expressed in the original system.

Kirchner and Viry [19] presented an implementation technique of rewriting on any existing loosely-coupled parallel architectures. The presented approach allows the parallel execution of programs directly from their specification, without having to adapt them to support parallelism, providing a simple and precise operational semantics. Finally, the solution proposed by the authors involves only local computations (at least for left-linear rules). The authors claim that the introduced concurrent rewriting can provide a programming paradigm allowing efficient parallel execution of programs without any explicit parallel directive. It can also be used as the kernel for simplification-based theorem provers.

Eerd, Groote, Hijma, Martens, Osama, and Wijs [14] presented a way to implement a many-sorted term rewriting on a GPU. The described method is based on the idea of letting the GPU repeatedly perform massively parallel evaluation of all subterms. The results show that when the many-sorted term rewriting system has sufficient internal parallelism, GPU rewriting substantially outperforms CPU rewriting (up to a factor of 10). The researchers also considered such optimizations as a relaxed innermost many-sorted term rewriting strategy and garbage collection, and concluded that they could further improve the system's performance.

3.2 Modelling Memory in Term Rewriting

The idea of modelling computer memory in Term Rewriting has been explored in several research projects. Arvind and Shen [2] presented the application of Term Rewriting Systems to describe microarchitectures. In their approach, the state of a system is represented as a TRS term, and the rewriting rules specify state transitions. To illustrate this idea, the authors used a minimalistic RISC instruction set and defined its operational semantics using a single-cycle, non-pipelined, in-order execution processor model. The processor consists of a program counter (pc), register file (rf), and instruction memory (im). Together with the data memory (dm), the processor constitutes the whole system, which can be represented as a TRS term: *Sys(Proc(pc, rf, im), dm)*. The authors also extended their analysis to a more sophisticated processor architecture supporting register renaming and speculative execution. While the memory system is modelled as operating asynchronously with respect to the processor, its concrete organization is not discussed in detail.

In another work, Hoe and Arvind [17] presented a Term-Rewriting-Based framework for the specification analysis and synthesis of processor microarchitectures. In the presented approach, the complete system state is represented as a structured term, and the behavior is modelled through the atomic rewrite rules. Memory structures are modelled as abstract functional arrays embedded within the global system term. The memory representation avoids side effects and allows precise reasoning about state transitions and concurrency. The main emphasis of the paper is to show that Term Rewriting can be used to model complex processor behavior, and subsequently can be compiled into the correct-by-construction hardware (specifically using a subset of the Verilog hardware description language).

Chapter 4

Term Rewriting with Global Memory (MemTRS)

A Memory Term Rewriting System is an extension of Logically Constrained Term Rewriting Systems, designed to support concurrent memory operations.

4.1 Alphabet of a Memory TRS

Since MemTRS uses integers to represent memory addresses and booleans to represent the outcomes of memory operations, its set of sorts contains the integer and boolean sorts: $\{int, bool\} \subseteq \mathcal{S}$.

In comparison to LCTRS, MemTRS has a special set of function symbols dedicated to memory operations:

$$\left.\begin{cases} \textbf{GET} : [int] \Rightarrow int, \\ \textbf{SET} : [int \times int] \Rightarrow bool, \\ \textbf{CAS} : [int \times int \times int] \Rightarrow bool \end{cases}\right\} \subseteq \Sigma_{mem}$$

So, the set of function symbols in MemTRS is defined as: $\Sigma = \Sigma_{\text{terms}} \cup \Sigma_{\text{theory}} \cup \Sigma_{\text{mem}}$

4.2 Memory in MemTRS

To support concurrent data structures, MemTRS introduces the concept of memory in Term Rewriting Systems. The memory unit is represented by a partial function:

$$mem : [int] \rightharpoonup int$$

An address x is called valid in mem, if $x \in \text{dom}(mem)$ (or, equivalently, if $mem(x) \downarrow$). If $x \notin \text{dom}(mem)$, we write $mem(x) \uparrow$ and call x invalid.

Since memory is mutable, there should exist a memory updating mechanism. For a valid address x and a value v, the updated memory function $mem[x \mapsto v]$ is defined on the same domain as mem, with:

$$mem[x \mapsto v](x) = v$$

and:

$$mem[x \mapsto v](y) = mem(y) \quad \text{for all } y \neq x$$

Example 2. Consider the memory function $mem : [int] \rightharpoonup int$, defined as follows: $\text{dom}(mem) = \mathbb{N}, \forall x \in \mathbb{N} \,.\, mem(x) = 0$. After updating the memory, the memory function holds the value 1 at the address 0. It can be represented in the following way:
$mem' = mem[0 \mapsto 1]$, and $mem'(x) = \begin{cases} 0, & x \neq 0 \\ 1, & x = 0 \end{cases}$

4.3 Memory Functions

As mentioned before, MemTRS introduces three special function symbols to support memory-related operations:

- **GET** : $[int] \Rightarrow int$ - this function symbol represents the memory read operation. After the reduction step, if the address x is valid, the $\mathbf{GET}(x)$ term should be reduced to the value stored in the memory at the address x. Otherwise, no memory reduction is applicable.
- **SET** : $[int \times int] \Rightarrow bool$ - this function symbol represents the memory update operation. After the reduction step, if the address x is valid, the $\mathbf{SET}(x, v)$ is reduced to `true`, expressing the update of the memory function with the substitution $[x \mapsto v]$. Otherwise, the term is reduced to `false`, and the memory function remains unchanged.
- **CAS** : $[int \times int \times int] \Rightarrow bool$ - this function symbol represents the atomic "Compare-And-Swap" operation. After the reduction step, the term $\mathbf{CAS}(x, e, v)$ reduces to `true` if the address x is valid and the value currently stored there is equal to e; in that case, the memory function is updated by the substitution $[x \mapsto v]$. If the address x is invalid, or if the value stored at x is different from e, then the term reduces to `false` and the memory function remains unchanged.

4.4 Reduction Rules of the Memory TRS

The rewrite rules of Memory Term Rewriting Systems don't differ significantly from those in LCTRS. The only difference is that both sides of the rewrite rule may contain memory-related operations. We use the notational convention introduced in Section 2.5: rules without an explicit constraint are implicitly constrained by [`true`].

However, the rewrite relation $\to_{\mathcal{R}}$ for the set of rewrite rules $\mathcal{R}$ has different semantics than in the case of LCTRS. This relation is defined as a relation on configurations consisting of the term and the memory function: $\langle t, mem \rangle$. To properly define the rewrite relation $\to_{\mathcal{R}}$, we should first introduce the notion of positions in terms. This notion will also play a crucial role in the future definition of the parallel reduction.

4.4.1 Positions

Definition 4.4.1. The set of positions in the term t is a set of strings: $\mathcal{P}os(t)$, inductively defined as follows:

- If $t = x \in \mathcal{V}ar$, then $\mathcal{P}os(t) = \{\varepsilon\}$ (where ε denotes an empty string).
- If $t = f(s_1, \ldots, s_n)$, then $\mathcal{P}os(t) = \{\varepsilon\} \cup \bigcup_{i=1}^{n} \{ip \mid p \in \mathcal{P}os(s_i)\}$

Definition 4.4.2. The position ε is called a root position of the term t and the function symbol or variable on that position is called the root symbol of t.

Definition 4.4.3. The prefix order is defined as follows: $p \leq q$ if and only if there exists another position p', such that: $pp' = q$ - this is a partial order on positions.

Definition 4.4.4. The positions p and q are considered parallel $p \mid\mid q$ if and only if p and q are incomparable with respect to $\leq$.

Definition 4.4.5. For the position $p \in \mathcal{P}os(t)$, the subterm of t at the position p is denoted as: $t|_p$, is defined by an induction on the length of the position p:

- $t|_\varepsilon = t$,
- $f(t_1, \ldots, t_n)|_{iq} = t_i|_q$

Using these definitions, we can define the set of parallel positions:

Definition 4.4.6. A set of positions P is parallel if $p \mid\mid q$ or $p = q$ holds for all positions $p, q \in P$.

4.4.2 Rewriting Relation on Terms

The one-step rewrite relation in MemTRS is defined on configurations consisting of a term and a memory function: $\langle t, mem \rangle$. Since rewriting in MemTRS is position-sensitive, for every position p we define a relation $\to^p_{\mathcal{R}}$. This relation is the union of three position-sensitive relations: $\to^p_{\texttt{rule}}$, $\to^p_{\texttt{calc}}$, and $\to^p_{\texttt{mem}}$:

Definition 4.4.7. For every term $s \in \mathcal{T}erms$, for every position $p \in \mathcal{P}os(s)$, we define the reduction relations

$$\to^p_{\texttt{rule}}, \qquad \to^p_{\texttt{calc}}, \qquad \to^p_{\texttt{mem}}$$

on configurations inductively as follows:

- $\langle l\gamma, mem\rangle \to^{\varepsilon}_{\texttt{rule}} \langle r\gamma, mem\rangle$, if $l \to r\ [\varphi] \in \mathcal{R}$, and γ respects $l \to r\ [\varphi]$.
- $\langle f(s_1, \ldots, s_n), mem\rangle \to^{\varepsilon}_{\texttt{calc}} \langle v, mem\rangle$, if $f \in \Sigma_{theory} \setminus (\Sigma_{terms} \cup \Sigma_{mem})$, all s_i are values, and v is the result of evaluation of $\mathcal{J}(f(s_1, \ldots, s_n))$.
- $\langle \mathbf{GET}(x), mem\rangle \to^{\varepsilon}_{\texttt{mem}} \langle mem(\mathcal{I}_{\text{int}}(x)), mem\rangle$, if $x \in \mathcal{V}al_{\text{int}}$, and $mem(\mathcal{I}_{\text{int}}(x)) \downarrow$.
- $\langle \mathbf{SET}(x, v), mem\rangle \to^{\varepsilon}_{\texttt{mem}} \langle \texttt{true}, mem[\mathcal{I}_{\text{int}}(x) \mapsto \mathcal{I}_{\text{int}}(v)]\rangle$, if $x, v \in \mathcal{V}al_{\text{int}}$, and $mem(\mathcal{I}_{\text{int}}(x)) \downarrow$.
- $\langle \mathbf{SET}(x, v), mem\rangle \to^{\varepsilon}_{\texttt{mem}} \langle \texttt{false}, mem\rangle$, if $x, v \in \mathcal{V}al_{\text{int}}$, and $mem(\mathcal{I}_{\text{int}}(x)) \uparrow$.
- $\langle \mathbf{CAS}(x, e, v), mem\rangle \to^{\varepsilon}_{\texttt{mem}} \langle \texttt{true}, mem[\mathcal{I}_{\text{int}}(x) \mapsto \mathcal{I}_{\text{int}}(v)]\rangle$, if $x, e, v \in \mathcal{V}al_{\text{int}}$, and $mem(\mathcal{I}_{\text{int}}(x)) = \mathcal{I}_{\text{int}}(e)$.
- $\langle \mathbf{CAS}(x, e, v), mem\rangle \to^{\varepsilon}_{\texttt{mem}} \langle \texttt{false}, mem\rangle$, if $x, e, v \in \mathcal{V}al_{\text{int}}$, $mem(\mathcal{I}_{\text{int}}(x)) \downarrow$, and $mem(\mathcal{I}_{\text{int}}(x)) \neq \mathcal{I}_{\text{int}}(e)$.
- $\langle \mathbf{CAS}(x, e, v), mem\rangle \to^{\varepsilon}_{\texttt{mem}} \langle \texttt{false}, mem\rangle$, if $x, e, v \in \mathcal{V}al_{\text{int}}$, and $mem(\mathcal{I}_{\text{int}}(x)) \uparrow$.
- If
$$\langle t_i, mem\rangle \to^{q}_{x} \langle u_i, mem'\rangle$$
for some $q \in \mathcal{P}os(t_i)$, and $x \in \{\texttt{rule}, \texttt{calc}, \texttt{mem}\}$, then
$$\langle f(t_1, \ldots, t_i, \ldots, t_n), mem\rangle \to^{iq}_{x} \langle f(t_1, \ldots, u_i, \ldots, t_n), mem'\rangle$$

Finally, for every position p, we can define the one-step reduction relation:

$$\to^{p}_{\mathcal{R}} := \to^{p}_{\texttt{rule}} \cup \to^{p}_{\texttt{calc}} \cup \to^{p}_{\texttt{mem}}$$

For simplicity, if the rewriting position can be inferred from the context, we can use the shorthand notation:

$$\langle s, mem\rangle \to_{\mathcal{R}} \langle t, mem'\rangle \quad \text{iff} \quad \exists p \in \mathcal{P}os(s).\ \langle s, mem\rangle \to^{p}_{\mathcal{R}} \langle t, mem'\rangle$$

Example 3. Consider the MemTRS with the following set of rules $\mathcal{R} = \{\mathbf{f}(x, y) \to \mathbf{g}(y, x)\}$. The single-step reduction of the starting term $\mathbf{f}(\mathtt{1}, \mathbf{GET}(\mathtt{0}))$ at the position 2ε, using the starting memory function $mem : \text{dom}(mem) = \mathbb{N}, \forall x \in \mathbb{N}\ .\ mem(x) = 0$, can be performed the following way:

$$\langle \mathbf{f}(\mathtt{1}, \mathbf{GET}(\mathtt{0})), mem\rangle \to^{2\varepsilon}_{\texttt{mem}} \langle \mathbf{f}(\mathtt{1}, \mathtt{0}), mem\rangle$$

4.5 Parallel Reduction

To express the notion of concurrent execution in MemTRS, another relation on configurations of terms and memory functions should be defined. This is a notion of parallel reduction.

4.5.1 Parallel Reduction Relation

Finally, we can define a notion of parallel reduction which will express the parallel computation in MemTRS:

Definition 4.5.1. Let $P = \{p_1, \ldots, p_n\} \subseteq \mathcal{P}os(s)$ be a set of parallel positions. We write

$$\langle s, mem\rangle \rightrightarrows^{P}_{\mathcal{R}} \langle t, mem'\rangle$$

if there exists a permutation $q_1, \ldots q_n$ of the positions in P and configurations:

$$\langle s_0, mem_0\rangle, \langle s_1, mem_1\rangle, \ldots, \langle s_n, mem_n\rangle,$$

such that:

$$\langle s_0, mem_0\rangle = \langle s, mem\rangle, \qquad \langle s_n, mem_n\rangle = \langle t, mem'\rangle,$$

and for every $i \in \{1, \ldots, n\}$:

$$\langle s_{i-1}, mem_{i-1}\rangle \to^{q_i}_{\mathcal{R}} \langle s_i, mem_i\rangle.$$

The term t is computed by applying reduction rules of the MemTRS $\mathcal{R}$, calculation, or memory steps to the subterms at the positions from the set P. The memory function mem' represents the resulting memory state. The reductions at the parallel positions can be performed in arbitrary order. This captures the nondeterminism of parallel execution.

Note that the set of positions P does not uniquely determine the performed parallel rewriting step: for each position $p \in P$, any applicable local reduction may be chosen. Hence, the relation $\rightrightarrows_{\mathcal{R}}^{P}$ is, in general, nondeterministic.

The single-step parallel reduction relation can be generalized to a multistep parallel reduction relation.

Definition 4.5.2. Let $L = [P_1, \ldots, P_n]$ be a list of parallel position sets. We write

$$\langle s, mem \rangle \rightrightarrows_{\mathcal{R}}^{*L} \langle t, mem' \rangle$$

if there exist configurations:

$$\langle s_0, mem_0 \rangle, \langle s_1, mem_1 \rangle, \ldots, \langle s_n, mem_n \rangle$$

such that:

$$\langle s_0, mem_0 \rangle = \langle s, mem \rangle, \qquad \langle s_n, mem_n \rangle = \langle t, mem' \rangle,$$

and for every $i \in \{1, \ldots, n\}$:

$$\langle s_{i-1}, mem_{i-1} \rangle \rightrightarrows_{\mathcal{R}}^{P_i} \langle s_i, mem_i \rangle.$$

In particular, when $n = 0$, we have $L = [\,]$, which gives us:

$$\langle s, mem \rangle \rightrightarrows_{\mathcal{R}}^{*[\,]} \langle s, mem \rangle.$$

The multi-step parallel reduction relation allows us to specify the set of disjoint positions P_i, where the reduction rules can be applied, for every reduction step i. This is very similar to the concept of performing a computation using multiple isolated threads that communicate via shared memory.

Example 4. Consider the MemTRS with the following set of rules, which expresses the computation of the sum of two squares using memory:

$$\mathcal{R} = \begin{cases} \mathbf{start}(x, y) \to \mathbf{s}(\mathbf{SET}(\mathtt{0}, x), \mathbf{SET}(\mathtt{1}, y)) \\ \mathbf{s}(\mathtt{true}, \mathtt{true}) \to \mathbf{t}(\mathbf{pow}(\mathtt{0}, \mathtt{2}), \mathbf{pow}(\mathtt{1}, \mathtt{2})) \\ \mathbf{t}(\mathtt{true}, \mathtt{true}) \to \mathbf{GET}(\mathtt{0}) + \mathbf{GET}(\mathtt{1}) \\ \\ \mathbf{pow}(a, x) \to \mathbf{pow_1}(a, \mathbf{GET}(a), x) \\ \mathbf{pow_1}(a, v, x) \to \mathbf{pow_2}(a, v, \mathbf{SET}(a, \mathtt{1}), x) \\ \mathbf{pow_2}(a, v, \mathtt{true}, x) \to \mathbf{pow_3}(a, v, \mathbf{SET}(a, v * \mathbf{GET}(a)), x - \mathtt{1}) \; [x > \mathtt{0}] \\ \mathbf{pow_2}(a, v, \mathtt{true}, x) \to \mathtt{true} \; [x \leq \mathtt{0}] \\ \mathbf{pow_3}(a, v, \mathtt{true}, x) \to \mathbf{pow_2}(a, v, \mathtt{true}, x) \end{cases}$$

The reduction of the starting term $start(3, 4)$ using the starting memory function $mem : \forall x \in \mathbb{N} \,.\, mem(x) = 0$, can be performed the following way:

$\langle \mathbf{start}(\mathtt{3}, \mathtt{4}), mem \rangle$

$$\begin{aligned}
&\rightrightarrows_{\mathcal{R}}^{\{\varepsilon\}} && \langle \mathbf{s}(\mathbf{SET}(\mathtt{0}, \mathtt{3}), \mathbf{SET}(\mathtt{1}, \mathtt{4})), mem \rangle \\
&\rightrightarrows_{\mathcal{R}}^{\{1\varepsilon, 2\varepsilon\}} && \langle \mathbf{s}(\mathtt{true}, \mathtt{true}), mem[0 \mapsto 3][1 \mapsto 4] \rangle \\
&\rightrightarrows_{\mathcal{R}}^{\{\varepsilon\}} && \langle \mathbf{t}(\mathbf{pow}(\mathtt{0}, \mathtt{2}), \mathbf{pow}(\mathtt{1}, \mathtt{2})), mem[0 \mapsto 3][1 \mapsto 4] \rangle \\
&\rightrightarrows_{\mathcal{R}}^{\{1\varepsilon, 2\varepsilon\}} && \langle \mathbf{t}(\mathbf{pow_1}(\mathtt{0}, \mathbf{GET}(\mathtt{0}), \mathtt{2}), \mathbf{pow_1}(\mathtt{1}, \mathbf{GET}(\mathtt{1}), \mathtt{2})), mem[0 \mapsto 3][1 \mapsto 4] \rangle \\
&\rightrightarrows_{\mathcal{R}}^{\{12\varepsilon, 22\varepsilon\}} && \langle \mathbf{t}(\mathbf{pow_1}(\mathtt{0}, \mathtt{3}, \mathtt{2}), \mathbf{pow_1}(\mathtt{1}, \mathtt{4}, \mathtt{2})), mem[0 \mapsto 3][1 \mapsto 4] \rangle \\
&\rightrightarrows_{\mathcal{R}}^{\{1\varepsilon, 2\varepsilon\}} && \langle \mathbf{t}(\mathbf{pow_2}(\mathtt{0}, \mathtt{3}, \mathbf{SET}(\mathtt{0}, \mathtt{1}), \mathtt{2}), \mathbf{pow_2}(\mathtt{1}, \mathtt{4}, \mathbf{SET}(\mathtt{1}, \mathtt{1}), \mathtt{2})), mem[0 \mapsto 3][1 \mapsto 4] \rangle \\
&\rightrightarrows_{\mathcal{R}}^{\{13\varepsilon, 23\varepsilon\}} && \langle \mathbf{t}(\mathbf{pow_2}(\mathtt{0}, \mathtt{3}, \mathtt{true}, \mathtt{2}), \mathbf{pow_2}(\mathtt{1}, \mathtt{4}, \mathtt{true}, \mathtt{2})), mem[0 \mapsto 1][1 \mapsto 1] \rangle \\
&\rightrightarrows_{\mathcal{R}}^{\{1\varepsilon, 2\varepsilon\}} && \langle \mathbf{t}(\mathbf{pow_3}(\mathtt{0}, \mathtt{3}, \mathbf{SET}(\mathtt{0}, \mathtt{3} * \mathbf{GET}(\mathtt{0})), \mathtt{2} - \mathtt{1}), \\
& && \quad \mathbf{pow_3}(\mathtt{1}, \mathtt{4}, \mathbf{SET}(\mathtt{1}, \mathtt{4} * \mathbf{GET}(\mathtt{1})), \mathtt{2} - \mathtt{1})), \\
& && \quad mem[0 \mapsto 1][1 \mapsto 1] \rangle \\
&\rightrightarrows_{\mathcal{R}}^{\{1322\varepsilon, 14\varepsilon, 2322\varepsilon, 24\varepsilon\}} && \langle \mathbf{t}(\mathbf{pow_3}(\mathtt{0}, \mathtt{3}, \mathbf{SET}(\mathtt{0}, \mathtt{3} * \mathtt{1}), \mathtt{1}), \\
& && \quad \mathbf{pow_3}(\mathtt{1}, \mathtt{4}, \mathbf{SET}(\mathtt{1}, \mathtt{4} * \mathtt{1}), \mathtt{1})), mem[0 \mapsto 1][1 \mapsto 1] \rangle \\
&\rightrightarrows_{\mathcal{R}}^{\{132\varepsilon, 232\varepsilon\}} && \langle \mathbf{t}(\mathbf{pow_3}(\mathtt{0}, \mathtt{3}, \mathbf{SET}(\mathtt{0}, \mathtt{3}), \mathtt{1}), \mathbf{pow_3}(\mathtt{1}, \mathtt{4}, \mathbf{SET}(\mathtt{1}, \mathtt{4}), \mathtt{1})), mem[0 \mapsto 1][1 \mapsto 1] \rangle
\end{aligned}$$

$$
\begin{aligned}
&\rightrightarrows_{\mathcal{R}}^{\{13\varepsilon,23\varepsilon\}} && \langle \mathbf{t}(\mathbf{pow_3}(0,3,\mathtt{true},1),\mathbf{pow_3}(1,4,\mathtt{true},1)), mem[0\mapsto 3][1\mapsto 4]\rangle\\
&\rightrightarrows_{\mathcal{R}}^{\{1\varepsilon,2\varepsilon\}} && \langle \mathbf{t}(\mathbf{pow_2}(0,3,\mathtt{true},1),\mathbf{pow_2}(1,4,\mathtt{true},1)), mem[0\mapsto 3][1\mapsto 4]\rangle\\
&\rightrightarrows_{\mathcal{R}}^{\{1\varepsilon,2\varepsilon\}} && \langle \mathbf{t}(\mathbf{pow_3}(0,3,\mathbf{SET}(0,3*\mathbf{GET}(0)),1-1),\\
& && \quad \mathbf{pow_3}(1,4,\mathbf{SET}(1,4*\mathbf{GET}(1)),1-1)),\\
& && \quad mem[0\mapsto 3][1\mapsto 4]\rangle\\
&\rightrightarrows_{\mathcal{R}}^{\{1322\varepsilon,14\varepsilon,2322\varepsilon,24\varepsilon\}} && \langle \mathbf{t}(\mathbf{pow_3}(0,3,\mathbf{SET}(0,3*3),0),\\
& && \quad \mathbf{pow_3}(1,4,\mathbf{SET}(1,4*4),0)), mem[0\mapsto 3][1\mapsto 4]\rangle\\
&\rightrightarrows_{\mathcal{R}}^{\{132\varepsilon,232\varepsilon\}} && \langle \mathbf{t}(\mathbf{pow_3}(0,3,\mathbf{SET}(0,9),0),\mathbf{pow_3}(1,4,\mathbf{SET}(1,16),0)), mem[0\mapsto 3][1\mapsto 4]\rangle\\
&\rightrightarrows_{\mathcal{R}}^{\{13\varepsilon,23\varepsilon\}} && \langle \mathbf{t}(\mathbf{pow_3}(0,3,\mathtt{true},0),\mathbf{pow_3}(1,4,\mathtt{true},0)), mem[0\mapsto 9][1\mapsto 16]\rangle\\
&\rightrightarrows_{\mathcal{R}}^{\{1\varepsilon,2\varepsilon\}} && \langle \mathbf{t}(\mathbf{pow_2}(0,3,\mathtt{true},0),\mathbf{pow_2}(1,4,\mathtt{true},0)), mem[0\mapsto 9][1\mapsto 16]\rangle\\
&\rightrightarrows_{\mathcal{R}}^{\{1\varepsilon,2\varepsilon\}} && \langle \mathbf{t}(\mathtt{true},\mathtt{true}), mem[0\mapsto 9][1\mapsto 16]\rangle\\
&\rightrightarrows_{\mathcal{R}}^{\{\varepsilon\}} && \langle \mathbf{GET}(0)+\mathbf{GET}(1), mem[0\mapsto 9][1\mapsto 16]\rangle\\
&\rightrightarrows_{\mathcal{R}}^{\{1\varepsilon,2\varepsilon\}} && \langle 9+16, mem[0\mapsto 9][1\mapsto 16]\rangle\\
&\rightrightarrows_{\mathcal{R}}^{\{\varepsilon\}} && \langle 25, mem[0\mapsto 9][1\mapsto 16]\rangle
\end{aligned}
$$

The example shows that parallel multi-step reduction can express how the parallel computation was performed.

4.5.2 Parallel Reduction Strategies

Similar to regular term rewriting, parallel rewriting uses different strategies for term reduction. In literature, authors mainly consider two rewriting strategies: *innermost* rewriting and *outermost* rewriting. Each of the two strategies can be generalized for use in a parallel context.

In this section we consider only deterministic rewriting strategies that at each reduction step capture all parallel innermost or outermost positions. However, following the definition of the parallel reduction relation, one can define another strategy that does not necessarily involve full sets of innermost/outermost parallel positions.

Innermost Parallel Reduction

Definition 4.5.3 (Redex. [13, Adapted from definition on p. 1])**.** A *redex* is a subterm at a position which matches the left-hand-side (LHS) of a rewrite rule, or if it can be reduced with a calculation or memory step.

Definition 4.5.4 (Innermost redex. [5, Adapted from p. 5])**.** A subterm of a term is an *innermost redex* if it can be reduced using either one of the rules of the Memory Term Rewriting System, calculation step, or a memory step; however, it does not have any proper subterm that is also a *redex*.

Definition 4.5.5 (Innermost parallel reduction. [20, Definition 2.2.4 on p. 78])**.** Innermost parallel reduction is a reduction strategy that, at each step, parallelly reduces all innermost redexes.

Example 5. Consider a starting term $\mathbf{s}(\mathbf{t}(1),\mathbf{SET}(0,1))$, starting memory function $mem : \mathrm{dom}(mem) = \mathbb{N}, \forall x \in \mathbb{N}\,.\, mem(x) = 0$, and the Memory Term Rewriting system

$$\mathcal{R} = \begin{cases} \mathbf{s}(x,y) \to x \\ \mathbf{t}(x) \to x \end{cases}$$

The starting tuple can be reduced to the normal form using the parallel innermost reduction in the following way:

$\langle \mathbf{s}(\mathbf{t}(1),\mathbf{SET}(0,1)), mem\rangle$

$$
\begin{aligned}
&\rightrightarrows_{\mathcal{R}}^{\{1\varepsilon,2\varepsilon\}} && \langle \mathbf{s}(1,\mathtt{true}), mem[0\mapsto 1]\rangle\\
&\rightrightarrows_{\mathcal{R}}^{\{\varepsilon\}} && \langle 1, mem[0\mapsto 1]\rangle
\end{aligned}
$$

Outermost Parallel Reduction

Definition 4.5.6 (Outermost redex. [18, Adapted from])**.** A subterm of a term is an *outermost* (or maximal) redex if it is not contained in any other redexes.

Definition 4.5.7. Outermost parallel reduction is a reduction strategy that parallelly reduces all maximal redexes.

Example 6. Consider the starting term, starting memory function, and the MemTRS from the Example 5.

The starting tuple can be reduced to the normal form using the parallel outermost reduction in the following way:

$\langle \mathbf{s}(\mathbf{t}(\mathtt{1}), \mathbf{SET}(\mathtt{0}, \mathtt{1})), mem \rangle$

$\rightrightarrows_{\mathcal{R}}^{\{\varepsilon\}} \quad \langle \mathbf{t}(\mathtt{1}), mem \rangle$

$\rightrightarrows_{\mathcal{R}}^{\{\varepsilon\}} \quad \langle \mathtt{1}, mem \rangle$

Parallel Reduction Relation with Strategy

Using the parallel reduction strategies described above, it is possible to define deterministic reduction relations. These relations do not require a specific set of reduction positions to specify a relation on terms. The set of positions is inferred from the starting term using a specific reduction strategy.

Definition 4.5.8. We say that if the starting term s and the initial memory function mem are parallelly rewritten to another term t and memory function mem' using the MemTRS $\mathcal{R}$ with a certain parallel reduction strategy $\mathbb{S} \in \{\mathbb{I}, \mathbb{O}\}$, we can express it with the following relation:

$$\langle s, mem \rangle \rightrightarrows_{\mathcal{R}}^{\mathbb{S}} \langle t, mem' \rangle$$

Here, $\mathbb{I}$ represents the innermost parallel reduction strategy, and $\mathbb{O}$ represents the outermost parallel reduction strategy.

4.6 Dealing with Non-determinism

Before diving into the non-determinism of Memory Term Rewriting, we should take a step back and acknowledge that Term Rewriting generally has a lot of inherent non-determinism. Term Rewriting Systems may contain overlapping rules, so choosing different reduction order may produce different results. Similar effect is present in parallel reduction as well: choosing different sets of parallel positions $P, P' : P \neq P'$, and performing a parallel reduction of some starting configuration $\langle s, mem \rangle$ may also yield a different outcome:

$$\langle s, mem \rangle \rightrightarrows_{\mathcal{R}}^{P} \langle t, mem_1 \rangle \qquad \langle s, mem \rangle \rightrightarrows_{\mathcal{R}}^{P'} \langle t', mem_2 \rangle.$$

The parallel reduction relation on terms with memory introduces nondeterminism in the rewriting relation. Since the parallel reduction relation imposes no restrictions on the positions used in rewriting, some combinations of positions can lead to a widespread parallel programming issue: data races.

This section will cover potential race conditions with memory that can occur in MemTRS systems including descriptions of the problems and possible resolution strategies.

4.6.1 Parallel read and write

The first problem occurs when the term contains two positions that read and write to the same address in memory.

Example 7. Consider the term $\mathbf{t}(\mathbf{GET}(\mathtt{0}), \mathbf{SET}(\mathtt{0}, \mathtt{1}))$, the starting memory function $mem : \mathrm{dom}(mem) = \mathbb{N}, \forall x \in \mathbb{N} \,.\, mem(x) = 0$, and some MemTRS $\mathcal{R}$. If the set of parallel positions $P = \{1\varepsilon, 2\varepsilon\}$ will be used in the parallel reduction of this term, then the starting tuple can be nondeterministically rewritten to two different configurations:

- $\langle \mathbf{t}(\mathbf{GET}(\mathtt{0}), \mathbf{SET}(\mathtt{0}, \mathtt{1})), mem \rangle \rightrightarrows_{\mathcal{R}}^{P} \langle \mathbf{t}(\mathtt{0}, \mathtt{true}), mem[0 \mapsto 1] \rangle$,

- $\langle \mathbf{t}(\mathbf{GET}(0), \mathbf{SET}(0,1)), mem\rangle \rightrightarrows_{\mathcal{R}}^{P} \langle \mathbf{t}(1, \mathtt{true}), mem[0 \mapsto 1]\rangle$

To avoid this kind of nondeterminism, it is advisable to design the Memory Rewriting System so that read and write operations associated with the same memory cell cannot appear in a single term. For instance, this can be achieved by chaining the read and write operations by adding special "blocking" rules to the rewriting system that ensure the read operation can be performed only after the write operation.

Example 8. Consider the starting term $\mathbf{t}(\mathbf{GET}(0), \mathbf{SET}(0,1))$. To resolve the data race that can occur in the parallel reduction of the term, one can perform the following:

1. Replace the starting term with the term $\mathbf{t_1}(\mathbf{SET}(0,1))$
2. Expand the MemTRS $\mathcal{R}$ with the rule: $\mathbf{t_1}(\mathtt{true}) \rightarrow \mathbf{t_2}(\mathbf{GET}(0))$

These additions will ensure that the reduction of the starting term will never have a non-deterministic behavior:

$\langle \mathbf{t_1}(\mathbf{SET}(0,1)), mem\rangle$

$\rightrightarrows_{\mathcal{R}}^{\{1\varepsilon\}} \quad \langle \mathbf{t_1}(\mathtt{true}), mem[0 \mapsto 1]\rangle$
$\rightrightarrows_{\mathcal{R}}^{\{\varepsilon\}} \quad \langle \mathbf{t_2}(\mathbf{GET}(0)), mem[0 \mapsto 1]\rangle$
$\rightrightarrows_{\mathcal{R}}^{\{1\varepsilon\}} \quad \langle \mathbf{t_2}(1), mem[0 \mapsto 1]\rangle$

Another approach to resolving this data race is to *restrict* the parallel positions to include only read or write operations during the parallel reduction step.

4.6.2 Multiple parallel writes

Another data race problem in Memory Rewriting Systems arises when multiple write statements to the exact memory location appear in a single term. If at least two of the positions of these write operations are included in the set of parallel positions, it is undefined which value will be stored in memory after the reduction.

Example 9. Consider the starting term: $s = \mathbf{t}(\mathbf{SET}(0,1), \mathbf{SET}(0,2), \mathbf{SET}(0,3))$, the starting memory function:

$$mem : \mathrm{dom}(mem) = \mathbb{N}, \forall x \in \mathbb{N} \,.\, mem(x) = 0$$

and some MemTRS $\mathcal{R}$. If the set of parallel positions $P = \{1\varepsilon, 2\varepsilon, 3\varepsilon\}$ will be used in the parallel reduction of this term, then the starting tuple $\langle s, mem\rangle$ can be nondeterministically rewritten to three different configurations:

- $\langle s, mem\rangle \rightrightarrows_{\mathcal{R}}^{P} \langle \mathbf{t}(\mathtt{true}, \mathtt{true}, \mathtt{true}), mem[0 \mapsto 1]\rangle$
- $\langle s, mem\rangle \rightrightarrows_{\mathcal{R}}^{P} \langle \mathbf{t}(\mathtt{true}, \mathtt{true}, \mathtt{true}), mem[0 \mapsto 2]\rangle$
- $\langle s, mem\rangle \rightrightarrows_{\mathcal{R}}^{P} \langle \mathbf{t}(\mathtt{true}, \mathtt{true}, \mathtt{true}), mem[0 \mapsto 3]\rangle$

To prevent this data race during the parallel rewriting, the set of parallel positions should not include positions of multiple write operations associated with the same memory address.

4.6.3 Side-effects of the Outermost Reduction Strategy

As discussed in Section 4.5.2, there exist different strategies for parallel reductions in MemTRS. The examples 5 and 6 clearly show that reducing the terms using these strategies can yield different normal forms, including distinct memory configurations.

This is a crucial detail to keep in mind when modelling a parallel algorithm in MemTRS. Using the outermost reduction strategy can result in unexpected effects: for instance, a memory operation in a term may be eliminated or delayed, which can lead to undesirable side effects.

So, the innermost (call-by-value) parallel rewriting strategy is much more predictable while working with memory operations. The reduction of read and, more importantly, write operations will be prioritized over reducing terms that may contain them. This will allow sequential memory operations to be performed and generally give more control over when they occur. This is crucial for modeling many parallel algorithms that use shared memory.

4.6.4 Simulating Memory Locks with CAS Operation

One of the basic memory operations in MemTRS is the atomic compare-and-set operation **CAS**. This construct allows us to simulate locks and mutexes - key building blocks of various concurrent algorithms and data structures.

Example 10. Consider a starting term $\mathbf{s}(\mathbf{inc}(\mathtt{0}), \mathbf{inc}(\mathtt{0}))$, starting memory function:

$$mem : \mathrm{dom}(mem) = \mathbb{N}, \forall x \in \mathbb{N} \,.\, mem(x) = 0$$

and the Memory Term Rewriting system:

$$\mathcal{R} = \begin{cases} \mathbf{inc}(addr) \to \mathbf{incTryLock}(addr, \mathbf{CAS}(addr, \mathtt{0}, \mathtt{1})) & (1) \\ \mathbf{incTryLock}(addr, \mathtt{true}) \to \mathbf{incRelease}(addr, \mathbf{SET}(addr + \mathtt{1}, \mathbf{GET}(addr + \mathtt{1}) + \mathtt{1})) & (2) \\ \mathbf{incTryLock}(addr, \mathtt{false}) \to \mathbf{inc}(addr) & (3) \\ \mathbf{incRelease}(addr, \mathtt{true}) \to \mathbf{SET}(addr, \mathtt{0}) & (4) \end{cases}$$

This system implements the atomic integer counter. The memory location `0` contains the lock value that allows the execution of the increment operation for only one "computation thread" at the time. The memory location `1` holds the counter value. After performing the increment operation, the lock is released. Since we are using the parallel reduction with shared memory, we have non-determinism: any of the "threads" can acquire the lock and perform the increment operation. The reduction of the starting term can be performed in the following way (one of the possible runs):

$\langle \mathbf{s}(\mathbf{inc}(\mathtt{0}), \mathbf{inc}(\mathtt{0})), mem \rangle$

$\Rightarrow_{\mathcal{R}}^{\{1\varepsilon,2\varepsilon\}}$ $\langle \mathbf{s}(\mathbf{incTryLock}(\mathtt{0}, \mathbf{CAS}(\mathtt{0},\mathtt{0},\mathtt{1})), \mathbf{incTryLock}(\mathtt{0}, \mathbf{CAS}(\mathtt{0},\mathtt{0},\mathtt{1}))), mem \rangle$

$\Rightarrow_{\mathcal{R}}^{\{12\varepsilon,22\varepsilon\}}$ $\langle \mathbf{s}(\mathbf{incTryLock}(\mathtt{0}, \mathtt{false}), \mathbf{incTryLock}(\mathtt{0}, \mathtt{true})), mem[0 \mapsto 1] \rangle$

$\Rightarrow_{\mathcal{R}}^{\{1\varepsilon,2\varepsilon\}}$ $\langle \mathbf{s}(\mathbf{inc}(\mathtt{0}), \mathbf{incRelease}(\mathtt{0}, \mathbf{SET}(\mathtt{0}+\mathtt{1}, \mathbf{GET}(\mathtt{0}+\mathtt{1})+\mathtt{1}))), mem[0 \mapsto 1] \rangle$

$\Rightarrow_{\mathcal{R}}^{\{1\varepsilon,221\varepsilon,22211\varepsilon\}}$ $\langle \mathbf{s}(\mathbf{incTryLock}(\mathtt{0}, \mathbf{CAS}(\mathtt{0},\mathtt{0},\mathtt{1})), \mathbf{incRelease}(\mathtt{0}, \mathbf{SET}(\mathtt{1}, \mathbf{GET}(\mathtt{1})+\mathtt{1}))), mem[0 \mapsto 1] \rangle$

$\Rightarrow_{\mathcal{R}}^{\{12\varepsilon,2221\varepsilon\}}$ $\langle \mathbf{s}(\mathbf{incTryLock}(\mathtt{0}, \mathtt{false}), \mathbf{incRelease}(\mathtt{0}, \mathbf{SET}(\mathtt{1}, \mathtt{0}+\mathtt{1}))), mem[0 \mapsto 1] \rangle$

$\Rightarrow_{\mathcal{R}}^{\{1\varepsilon,222\varepsilon\}}$ $\langle \mathbf{s}(\mathbf{inc}(\mathtt{0}), \mathbf{incRelease}(\mathtt{0}, \mathbf{SET}(\mathtt{1}, \mathtt{1}))), mem[0 \mapsto 1] \rangle$

$\Rightarrow_{\mathcal{R}}^{\{1\varepsilon,22\varepsilon\}}$ $\langle \mathbf{s}(\mathbf{incTryLock}(\mathtt{0}, \mathbf{CAS}(\mathtt{0},\mathtt{0},\mathtt{1})), \mathbf{incRelease}(\mathtt{0}, \mathtt{true})), mem[0 \mapsto 1][1 \mapsto 1] \rangle$

$\Rightarrow_{\mathcal{R}}^{\{12\varepsilon,2\varepsilon\}}$ $\langle \mathbf{s}(\mathbf{incTryLock}(\mathtt{0}, \mathtt{false}), \mathbf{SET}(\mathtt{0}, \mathtt{0})), mem[0 \mapsto 1][1 \mapsto 1] \rangle$

$\Rightarrow_{\mathcal{R}}^{\{1\varepsilon,2\varepsilon\}}$ $\langle \mathbf{s}(\mathbf{inc}(\mathtt{0}), \mathtt{true}), mem[0 \mapsto 0][1 \mapsto 1] \rangle$

$\Rightarrow_{\mathcal{R}}^{\{1\varepsilon\}}$ $\langle \mathbf{s}(\mathbf{incTryLock}(\mathtt{0}, \mathbf{CAS}(\mathtt{0},\mathtt{0},\mathtt{1})), \mathtt{true}), mem[0 \mapsto 0][1 \mapsto 1] \rangle$

$\Rightarrow_{\mathcal{R}}^{\{12\varepsilon\}}$ $\langle \mathbf{s}(\mathbf{incTryLock}(\mathtt{0}, \mathtt{true}), \mathtt{true}), mem[0 \mapsto 1][1 \mapsto 1] \rangle$

$\Rightarrow_{\mathcal{R}}^{\{1\varepsilon\}}$ $\langle \mathbf{s}(\mathbf{incRelease}(\mathtt{0}, \mathbf{SET}(\mathtt{0}+\mathtt{1}, \mathbf{GET}(\mathtt{0}+\mathtt{1})+\mathtt{1})), \mathtt{true}), mem[0 \mapsto 1][1 \mapsto 1] \rangle$

$\Rightarrow_{\mathcal{R}}^{\{121\varepsilon,12211\varepsilon\}}$ $\langle \mathbf{s}(\mathbf{incRelease}(\mathtt{0}, \mathbf{SET}(\mathtt{1}, \mathbf{GET}(\mathtt{1})+\mathtt{1})), \mathtt{true}), mem[0 \mapsto 1][1 \mapsto 1] \rangle$

$\Rightarrow_{\mathcal{R}}^{\{1221\varepsilon\}}$ $\langle \mathbf{s}(\mathbf{incRelease}(\mathtt{0}, \mathbf{SET}(\mathtt{1}, \mathtt{1}+\mathtt{1})), \mathtt{true}), mem[0 \mapsto 1][1 \mapsto 1] \rangle$

$\Rightarrow_{\mathcal{R}}^{\{122\varepsilon\}}$ $\langle \mathbf{s}(\mathbf{incRelease}(\mathtt{0}, \mathbf{SET}(\mathtt{1}, \mathtt{2})), \mathtt{true}), mem[0 \mapsto 1][1 \mapsto 1] \rangle$

$\Rightarrow_{\mathcal{R}}^{\{12\varepsilon\}}$ $\langle \mathbf{s}(\mathbf{incRelease}(\mathtt{0}, \mathtt{true}), \mathtt{true}), mem[0 \mapsto 1][1 \mapsto 2] \rangle$

$\Rightarrow_{\mathcal{R}}^{\{1\varepsilon\}}$ $\langle \mathbf{s}(\mathbf{SET}(\mathtt{0}, \mathtt{0}), \mathtt{true}), mem[0 \mapsto 1][1 \mapsto 2] \rangle$

$\Rightarrow_{\mathcal{R}}^{\{1\varepsilon\}}$ $\langle \mathbf{s}(\mathtt{true}, \mathtt{true}), mem[0 \mapsto 0][1 \mapsto 2] \rangle$

Chapter 5

Data structures in MemTRS

In the previous chapter, we've seen that Memory Term Rewriting systems provide basic building blocks for constructing algorithms with shared-memory operations. However, the existing memory functions **GET**, **SET**, and **CAS** are very minimal and require extra effort to model algorithms that utilize such data structures as arrays or matrices.

To address this issue, as a part of this research project, we designed several utility rewrite systems for common shared data structures, such as arrays, matrices and concurrent queues. These rules provide higher-level abstractions over the basic memory operations of MemTRS. In comparison to the core memory functions, not all the high-level operations are atomic in the sense of being performed in a single reduction step. Many of them are implemented as the multistep procedures, and their intermediate steps may therefore be exposed during a parallel computation. However, operations that require synchronization, such as memory allocation and concurrent queue updates, are protected using the atomic **CAS** operation. This makes the intended critical sections safe with respect to the shared-memory semantics introduced in the Chapter 4.5. This is very similar to the idea of thread-safe functions in multithreaded computing.

5.1 Memory Model

Since the shared data structures in MemTRS are stored entirely in the global memory, it's essential to consider the concept of *memory allocation*. Automatic memory management will significantly optimize work with shared data structures, removing the user's burden of managing memory addresses and searching for free locations.

In this research project, we decided to implement a simple memory model without deallocation and garbage collection. The memory cell with the address `0` is reserved to store the address of the next "free" memory cell. The memory cell `1` is reserved for the allocation lock: to perform parallel memory allocations safely, we will use the lock mechanism implemented with the **CAS** operation. This serializes the allocation procedure - only one computation "thread" will be able to perform this operation at a time. The system uses the following rules to manage memory addresses:

$$\mathcal{R}_{\text{alloc}} = \begin{cases} \textbf{alloc}(\mathit{size}) \to \textbf{allocLock}(\mathit{size}, \textbf{CAS}(\texttt{1}, \texttt{0}, \texttt{1}))\ [\mathit{size} > 0] & (1) \\ \textbf{alloc}(\mathit{size}) \to \texttt{-1}\ [\mathit{size} \leq \texttt{0}] & (2) \\ \textbf{allocLock}(\mathit{size}, \texttt{false}) \to \textbf{alloc}(\mathit{size}) & (3) \\ \textbf{allocLock}(\mathit{size}, \texttt{true}) \to \textbf{allocRead}(\mathit{size}, \textbf{GET}(\texttt{0})) & (4) \\ \textbf{allocRead}(\mathit{size}, \mathit{fresh}) \to \textbf{allocReserve}(\mathit{fresh}, \textbf{SET}(\texttt{0}, \mathit{fresh} + \mathit{size})) & (5) \\ \textbf{allocReserve}(\mathit{addr}, \texttt{true}) \to \textbf{allocUnlock}(\mathit{addr}, \textbf{SET}(\texttt{1}, \texttt{0})) & (6) \\ \textbf{allocUnlock}(\mathit{addr}, \texttt{true}) \to \mathit{addr} & (7) \end{cases}$$

The function **alloc** takes as a parameter the size of the memory block that, for example, is planned to be used for some data structure. If the requested size is non-positive, the operation returns the error code `-1` without modifying the memory. Since valid allocated addresses are non-negative, this value cannot be confused with a valid memory address. For a positive size, the allocation mechanism first attempts to acquire the lock using CAS. If this step fails, the allocator retries in a busy loop until it acquires the lock. After gaining control of the resource, the operation reads the next free address, advances the free pointer by the requested block size, releases the lock, and returns the starting address of this new block.

5.2 Arrays

The most basic data structure used in various sorting and searching algorithms is an array of integers. Since MemTRS provides direct access to the shared memory through the operations **GET** and **SET**, the array lookup and update operations can be represented naturally by address arithmetic. In our encoding, arrays are stored in memory as sequences of values, where the first value in the sequence denotes the array's size.

> **Example 11.** Consider the sequence of values: $[1, 2, 3]$. After being stored in a memory $mem : \mathrm{dom}(mem) = \mathbb{N}, \forall x \in \mathbb{N} \,.\, mem(x) = 0$, the resulting memory layout will look like this:
>
> $$\begin{aligned} &\mathrm{mem}(0) = 6 \\ &\mathrm{mem}(1) = 0 \\ &mem(2) = 3 \\ &mem(3) = 1 \\ &mem(4) = 2 \\ &mem(5) = 3 \\ &\ldots \end{aligned}$$
>
> The address `0` stores the location of the next "free" memory cell - address `6`. The address `1` is reserved for the memory allocation lock. The address `2` stores the array's size. And the memory locations `3-5` store the sequence of the array values.

To model different algorithms using arrays, we developed a set of rewrite rules for common array operations.

5.2.1 Array Creation

The following set of rules creates an array. This operation allocates a block of memory, writes the array size to the first address of that block, and returns the address of the created array. If the user supplies the invalid size for the array (negative or 0), then the operation does not perform any allocation and just returns the error status code `-1`. We assume that when called with a positive size, **alloc** guarantees a valid fresh memory block.

$\mathcal{R}_{\mathrm{createArray}} =$

$$\begin{cases} \textbf{createArray}(size) \to \textbf{createArraySetSize}(\textbf{alloc}(size + \texttt{1}), size)\ [size > 0] & (1) \\ \textbf{createArray}(size) \to \texttt{-1}\ [size \leq 0] & (2) \\ \textbf{createArraySetSize}(addr, size) \to & \\ \quad \textbf{createArraySetSizeDone}(addr, size, \textbf{SET}(addr, size))\ [addr \geq \texttt{0}] & (3) \\ \textbf{createArraySetSize}(addr, size) \to \texttt{-1}\ [addr < \texttt{0}] & (4) \\ \textbf{createArraySetSizeDone}(addr, size, \texttt{true}) \to \textbf{createArrayFillZeros}(addr, addr + \texttt{1}, size) & (5) \\ \textbf{createArrayFillZeros}(aAddr, addr, rem) \to & \\ \quad \textbf{createArrayFillZerosDone}(aAddr, addr + \texttt{1}, rem - \texttt{1}, \textbf{SET}(addr, \texttt{0}))\ [rem > \texttt{0}] & (6) \\ \textbf{createArrayFillZeros}(aAddr, addr, rem) \to aAddr\ [rem \leq \texttt{0}] & (7) \\ \textbf{createArrayFillZerosDone}(aAddr, addr, rem, \texttt{true}) \to \textbf{createArrayFillZeros}(aAddr, addr, rem) & (8) \end{cases}$$

5.2.2 Array Lookup

The following set of rules implement the bounds-checked access to array elements. If the given element index is within the array bounds, the term reduces to the corresponding array value. Otherwise, the term reduces to the error status code `-1`, without performing the **GET** operation outside the array.

$$\mathcal{R}_{\mathrm{getArr}} = \begin{cases} \textbf{getArr}(addr, i) \to \textbf{getArrBounds}(addr, i, \textbf{GET}(addr)) & (1) \\ \textbf{getArrBounds}(addr, i, size) \to \textbf{GET}(addr + i + \texttt{1})\ [\texttt{0} \leq i \wedge i < size] & (2) \\ \textbf{getArrBounds}(addr, i, size) \to \texttt{-1}\ [i < \texttt{0} \vee i \geq size] & (3) \end{cases}$$

5.2.3 Array Update

The following set of rules implements the bounds-checked array update procedure. If the given index is within the array bounds, the memory is updated with a new value and the term reduces to `true`. Otherwise, the term

reduces to `false`, without performing the **SET** operation outside the array.

$$\mathcal{R}_{\text{setArr}} = \begin{cases} \textbf{setArr}(addr, i, value) \to \textbf{setArrBounds}(addr, i, value, \textbf{GET}(addr)) & (1) \\ \textbf{setArrBounds}(addr, i, value, size) \to \textbf{SET}(addr + i + \texttt{1}, value)\ [\texttt{0} \leq i \wedge i < size] & (2) \\ \textbf{setArrBounds}(addr, i, value, size) \to \texttt{false}\ [i < \texttt{0} \vee i \geq size] & (3) \end{cases}$$

5.2.4 Swap Elements

The following set of rules implements the bounds-checked procedure for swapping two elements in the array. If both indices are within the array bounds, the corresponding values are exchanged in memory and the term reduces to `true`. Otherwise, the term reduces to `false` without performing any **GET** or **SET** operations outside the array.

$\mathcal{R}_{\text{swapArr}} =$

$$\begin{cases} \textbf{swapArr}(addr, i_1, i_2) \to \textbf{swapArrBounds}(addr, i_1, i_2, \textbf{GET}(addr)) & (1) \\ \textbf{swapArrBounds}(addr, i_1, i_2, size) \to \texttt{false}\ [(i_1 \geq size) \vee (i_2 \geq size) \vee (i_1 < \texttt{0}) \vee (i_2 < \texttt{0})] & (2) \\ \textbf{swapArrBounds}(addr, i_1, i_2, size) \to & \\ \quad \textbf{swapArrLoad}(addr, i_1, i_2, \textbf{GET}(addr + i_1 + \texttt{1}), \textbf{GET}(addr + i_2 + \texttt{1})) & \\ \quad\quad [(\texttt{0} \leq i_1) \wedge (i_1 < size) \wedge (\texttt{0} \leq i_2) \wedge (i_2 < size)] & (3) \\ \textbf{swapArrLoad}(addr, i_1, i_2, val_1, val_2) \to & \\ \quad \textbf{swapArrWrite}(\textbf{SET}(addr + i_2 + \texttt{1}, val_1), \textbf{SET}(addr + i_1 + \texttt{1}, val_2)) & (4) \\ \textbf{swapArrWrite}(status_1, status_2) \to \texttt{true}\ [status_1 \wedge status_2] & (5) \\ \textbf{swapArrWrite}(status_1, status_2) \to \texttt{false}\ [\neg(status_1 \wedge status_2)] & (6) \end{cases}$$

5.2.5 Fill Array

The following set of rules allows us to fill the given array with some default value. If the array filling operation is successful, the reduction of the term will result in a constant `true`. If any of the array update operations fails, the reduction will result in a constant `false`.

$\mathcal{R}_{\text{fillArray}} =$

$$\begin{cases} \textbf{fillArray}(addr, val) \to \textbf{fillArrayLoop}(addr, val, \texttt{0}, \textbf{getArrSize}(addr)) & (1) \\ \textbf{fillArrayLoop}(addr, val, i, size) \to \textbf{fillArrayStep}(addr, val, i, size, \textbf{setArr}(addr, i, val))\ [i < size] & (2) \\ \textbf{fillArrayLoop}(addr, val, i, size) \to \texttt{true}\ [i \geq size] & (3) \\ \textbf{fillArrayStep}(addr, val, i, size, \texttt{true}) \to \textbf{fillArrayLoop}(addr, val, i + \texttt{1}, size) & (4) \\ \textbf{fillArrayStep}(addr, val, i, size, \texttt{false}) \to \texttt{false} & (5) \end{cases}$$

We use the following helper rule for determining the size of the array:

$\mathcal{R}_{\text{getArrSize}} = \{\textbf{getArrSize}(addr) \to \textbf{GET}(addr)\}$

5.3 Matrices

Arrays can be used as building blocks for more complex shared data structures, such as matrices. In our MemTRS encoding, a matrix is represented in memory as an array of row addresses, where each row is stored as an array. Thus, the top-level array stores pointers to the arrays representing individual matrix rows.

Example 12. Consider the following matrix:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

After being stored in the memory $mem : \text{dom}(mem) = \mathbb{N}, \forall x \in \mathbb{N}\ .\ mem(x) = 0$, the resulting memory

layout will look like this:

$$\begin{array}{ll} \text{mem}(0) = 11 & mem(6) = 1 \\ \text{mem}(1) = 0 & mem(7) = 2 \\ mem(2) = 2 & mem(8) = 2 \\ mem(3) = 5 & mem(9) = 3 \\ mem(4) = 8 & mem(10) = 4 \\ mem(5) = 2 & \ldots \end{array}$$

So, memory locations 2 - 4 store the indexing array for matrix rows, and groups of locations 5 - 7 and 8 - 10 store the arrays representing the rows in the matrix. And memory locations 0 - 1 are reserved for the next free block pointer, and a lock for the memory allocation.

To model different algorithms with matrices, we developed sets of rewriting rules for common matrix operations.

5.3.1 Matrix Creation

The following set of rules creates a matrix in memory. Assuming the matrix is of dimension $w \times h$, the construction allocates $h + 1$ arrays: h row arrays of size w, and one top-level array of size h that stores the addresses of row arrays. If either dimension is non-positive, the operation returns the error code `-1`.

$\mathcal{R}_{\text{createMatrix}} =$

$$\begin{cases} \textbf{createMatrix}(w, h) \to \textbf{createMatrixInit}(\textbf{createArray}(h), w, h)\ [w > 0 \wedge h > 0] & (1) \\ \textbf{createMatrix}(w, h) \to \texttt{-1}\ [w \leq 0 \vee h \leq 0] & (2) \\ \textbf{createMatrixInit}(arrsAddr, w, h) \to \textbf{createMatrixLoop}(arrsAddr, w, h, 0) & (3) \\ \textbf{createMatrixLoop}(arrsAddr, w, h, i) \to & \\ \quad \textbf{createMatrixCreateRow}(\textbf{createArray}(w), arrsAddr, w, h, i)\ [i < h] & (4) \\ \textbf{createMatrixCreateRow}(addr, arrsAddr, w, h, i) \to & \\ \quad \textbf{createMatrixStoreRow}(\textbf{setArr}(arrsAddr, i, addr), addr, arrsAddr, w, h, i) & (5) \\ \textbf{createMatrixStoreRow}(\texttt{true}, addr, arrsAddr, w, h, i) \to \textbf{createMatrixLoop}(arrsAddr, w, h, i + 1) & (6) \\ \textbf{createMatrixLoop}(arrsAddr, w, h, i) \to arrsAddr\ [i \geq h] & (7) \end{cases}$$

5.3.2 Matrix Lookup

The following set of rules implements bounds-checked access to the matrix elements. If the specified element indices are outside the matrix bounds, the reduction of the term will result in an error status code `-1`, without calling the **GET** function outside the matrix memory space.

$\mathcal{R}_{\text{getMatrix}} =$

$$\begin{cases} \textbf{getMatrix}(addr, x, y) \to & \\ \quad \textbf{getMatrixBounds}(addr, \textbf{getMatrixWidth}(addr), \textbf{getMatrixHeight}(addr), x, y) & (1) \\ \textbf{getMatrixBounds}(addr, width, height, x, y) \to & \\ \quad \textbf{getArr}(\textbf{getArr}(addr, y), x)\ [0 \leq x \wedge x < width \wedge 0 \leq y \wedge y < height] & (2) \\ \textbf{getMatrixBounds}(addr, width, height, x, y) \to \texttt{-1}\ [0 > x \vee x \geq width \vee 0 > y \vee y \geq height] & (3) \end{cases}$$

The matrix lookup operation relies on two helper functions that allow access to the matrix dimensions:

$\mathcal{R}_{\text{getMatrixWidth}} = \{\textbf{getMatrixWidth}(addr) \to \textbf{getArrSize}(\textbf{getArr}(addr, 0))\}$

$\mathcal{R}_{\text{getMatrixHeight}} = \{\textbf{getMatrixHeight}(addr) \to \textbf{getArrSize}(addr)\}$

5.3.3 Matrix Update

The following set of rules implements a bounds-checked update of a matrix element. If the specified element index is within the matrix bounds, after the reduction, the memory will be updated with the new value, and the term will be reduced to the constant `true`. Otherwise, if the specified indices are invalid, the term reduces

to `false` and the **SET** operation won't be called outside the matrix memory space.

$\mathcal{R}_{\text{setMatrix}} =$

$$\begin{cases} \textbf{setMatrix}(addr, x, y, val) \rightarrow & \\ \quad \textbf{setMatrixBounds}(addr, \textbf{getMatrixWidth}(addr), \textbf{getMatrixHeight}(addr), x, y, val) & (1) \\ \textbf{setMatrixBounds}(addr, width, height, x, y, val) \rightarrow & \\ \quad \textbf{setArr}(\textbf{getArr}(addr, y), x, val)\ [0 \leq x \wedge x < width \wedge 0 \leq y \wedge y < height] & (2) \\ \textbf{setMatrixBounds}(addr, width, height, x, y, val) \rightarrow \texttt{false}\ [0 > x \vee x \geq width \vee 0 > y \vee y \geq height] & (3) \end{cases}$$

5.3.4 Filling the Matrix

The following set of rules fills the matrix stored at address *addr* with the specified value *val*. If the given memory location stores a valid matrix, after reduction, all the matrix elements will be replaced with the value *val*, and the term will be reduced to the constant `true`.

$\mathcal{R}_{\text{fillMatrix}} =$

$$\begin{cases} \textbf{fillMatrix}(addr, val) \rightarrow \textbf{fillMatrixLoop}(addr, val, \textbf{getMatrixHeight}(addr), 0) & (1) \\ \textbf{fillMatrixLoop}(addr, val, height, i) \rightarrow & \\ \quad \textbf{fillMatrixStep}(addr, val, height, i, \textbf{fillArray}(\textbf{getArr}(addr, i), val))\ [i < height] & (2) \\ \textbf{fillMatrixStep}(addr, val, height, i, \texttt{true}) \rightarrow \textbf{fillMatrixLoop}(addr, val, height, i + 1) & (3) \\ \textbf{fillMatrixStep}(addr, val, height, i, \texttt{false}) \rightarrow \texttt{false} & (4) \\ \textbf{fillMatrixLoop}(addr, val, height, i) \rightarrow \texttt{true}\ [i \geq height] & (5) \end{cases}$$

5.3.5 Filling the Matrix's Main Diagonal

The following set of rules fills the main diagonal of the matrix stored at address *addr* with the value *val*. For a matrix of dimensions $width \times height$, the main diagonal consists of the entries (i, i) for all indices:

$$0 \leq i < \min(width, height)$$

So, this procedure updates diagonal elements only up to the smaller of the two dimensions. If all updates succeed, the term reduces to `true`; otherwise it reduces to `false`.

$\mathcal{R}_{\text{fillMatrixDiag}} =$

$$\begin{cases} \textbf{fillMatrixDiag}(addr, val) \rightarrow & \\ \quad \textbf{fillMatrixDiagLoop}(addr, val, \textbf{min}(\textbf{getMatrixWidth}(addr), \textbf{getMatrixHeight}(addr)), 0) & (1) \\ \textbf{fillMatrixDiagLoop}(addr, val, side, i) \rightarrow & \\ \quad \textbf{fillMatrixDiagStep}(addr, val, side, i, \textbf{setMatrix}(addr, i, i, val))\ [i < side] & (2) \\ \textbf{fillMatrixDiagStep}(addr, val, side, i, \texttt{true}) \rightarrow \textbf{fillMatrixDiagLoop}(addr, val, side, i + 1) & (3) \\ \textbf{fillMatrixDiagStep}(addr, val, side, i, \texttt{false}) \rightarrow \texttt{false} & (4) \\ \textbf{fillMatrixDiagLoop}(addr, val, side, i) \rightarrow \texttt{true}\ [i \geq side] & (5) \end{cases}$$

The operation relies on the following helper function:

$$\mathcal{R}_{\text{min}} = \begin{cases} \textbf{min}(x, y) \rightarrow x\ [x \leq y] \\ \textbf{min}(x, y) \rightarrow y\ [y < x] \end{cases}$$

5.3.6 Converting from Memory to Term Representation

We use the following constructors to represent matrices as terms:

$$\begin{cases} \texttt{rowNil} :: \text{matrix} & (1) \\ \textbf{row} :: \text{list} \rightarrow \text{matrix} \rightarrow \text{matrix} & (2) \end{cases}$$

Matrix rows are represented as lists built from the constructors:

$$\begin{cases} \texttt{nil} :: \text{list} & (1) \\ \textbf{cons} :: \text{int} \to \text{list} \to \text{list} & (2) \end{cases}$$

The following set of rules converts the matrix stored at address *addr* into this term representation.

$\mathcal{R}_{\text{matrixToTerm}} =$

$$\begin{cases} \textbf{matrixToTerm}(addr) \to \textbf{matrixToTermLoop}(addr, \textbf{getMatrixHeight}(addr), 0, \texttt{rowNil}) & (1) \\ \textbf{matrixToTermLoop}(addr, size, i, m) \to & \\ \quad \textbf{row}(\textbf{arrayToList}(\textbf{getArr}(addr, i)), \textbf{matrixToTermNext}(addr, size, i, m))\ [i < size] & (2) \\ \textbf{matrixToTermNext}(addr, size, i, m) \to \textbf{matrixToTermLoop}(addr, size, i + 1, m) & (3) \\ \textbf{matrixToTermLoop}(addr, size, i, m) \to m\ [i \geq size] & (4) \end{cases}$$

The conversion relies on the following helper rules:

$$\mathcal{R}_{\text{readRange}} = \begin{cases} \textbf{readRange}(start, end) \to \textbf{readRangeLoop}(start, end, \texttt{nil}) & (1) \\ \textbf{readRangeLoop}(start, end, l) \to & \\ \quad \textbf{cons}(\textbf{GET}(start), \textbf{readRangeLoop}(start + 1, end, l))\ [start \leq end] & (2) \\ \textbf{readRangeLoop}(start, end, l) \to l\ [start > end] & (3) \end{cases}$$

$$\mathcal{R}_{\text{arrayToList}} = \begin{cases} \textbf{arrayToList}(addr) \to \textbf{arrayToListRead}(addr, \textbf{GET}(addr)) & (1) \\ \textbf{arrayToListRead}(addr, size) \to \textbf{readRange}(addr + 1, addr + size) & (2) \end{cases}$$

5.3.7 Converting Graph Data Structure into the Matrix

Consider the following constructors that can be used to represent directed graphs as terms using the adjacency list approach:

$$\begin{cases} \texttt{nilG} :: \text{graph} & (1) \\ \textbf{consG} :: \text{list} \to \text{graph} \to \text{graph} & (2) \end{cases}$$

Also, one can use the following helper rules to count the number of nodes in the graph or get a list of adjacent nodes with respect to some node:

$$\mathcal{R}_{\text{numNodes}} = \begin{cases} \textbf{numNodes}(\texttt{nilG}) \to 0 & (1) \\ \textbf{numNodes}(\textbf{consG}(l, g)) \to 1 + \textbf{numNodes}(g) & (2) \end{cases}$$

$$\mathcal{R}_{\text{getAdj}} = \begin{cases} \textbf{getAdj}(n, \textbf{consG}(l, g)) \to l\ [n = 0] & (1) \\ \textbf{getAdj}(n, \textbf{consG}(l, g)) \to \textbf{getAdj}(n - 1, g)\ [n > 0] & (2) \end{cases}$$

The following set of rules converts a graph represented by adjacency lists into a matrix stored in memory. The resulting matrix is an adjacency matrix, where the entry at position (x, y) indicates whether there is an edge from node y to node x. If the graph is empty, the operation returns the error code `-1`.

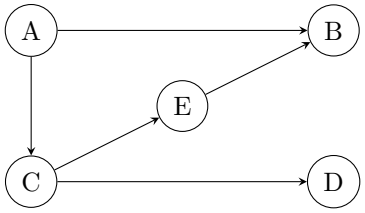
Since we need first to allocate the matrix in memory to later use the correct matrix address in the graph conversion procedure, we need to enforce the evaluation of the subterm $\textbf{createMatrix}(size, size)$, before reducing to the term $\textbf{graphToMatrixInit}(graph, default, addr, i)$. This is not a problem if we use the left-most innermost Call-By-Value reduction strategy, but it might be problematic if we were to use other reduction strategies. In *Cora*, this can be enforced by adding a simple equality constraint such as $addr = addr$, which forces the corresponding subterm to be reduced to a value before the rule is applied.

We assume that the adjacency lists contain valid zero-based indices.

$\mathcal{R}_{\text{graphToMatrix}} =$

$$
\begin{cases}
\textbf{graphToMatrix}(\mathit{graph}, \mathit{default}) \to \textbf{graphToMatrixSize}(\mathit{graph}, \mathit{default}, \textbf{numNodes}(\mathit{graph})) & (1)\\
\textbf{graphToMatrixSize}(\mathit{graph}, \mathit{default}, \mathit{size}) \to & \\
\quad \textbf{graphToMatrixInit}(\mathit{graph}, \mathit{default}, \textbf{createMatrix}(\mathit{size}, \mathit{size}), \texttt{0})\ [\mathit{size} > \texttt{0}] & (2)\\
\textbf{graphToMatrixSize}(\mathit{graph}, \mathit{default}, \mathit{size}) \to \texttt{-1}\ [\mathit{size} \leq \texttt{0}] & (3)\\
\textbf{graphToMatrixInit}(\mathit{graph}, \mathit{default}, \mathit{addr}, i) \to & \\
\quad \textbf{graphToMatrixLoop}(\mathit{graph}, \mathit{addr}, i, \textbf{fillMatrix}(\mathit{addr}, \mathit{default}))\ [\mathit{addr} = \mathit{addr}] & (4)\\
\textbf{graphToMatrixLoop}(\textbf{consG}(l, \mathit{graph}), \mathit{addr}, i, \texttt{true}) \to & \\
\quad \textbf{graphToMatrixEdges}(\mathit{graph}, l, \mathit{addr}, i) & (5)\\
\textbf{graphToMatrixLoop}(\texttt{nilG}, \mathit{addr}, i, \texttt{true}) \to \mathit{addr} & (6)\\
\textbf{graphToMatrixEdges}(\mathit{graph}, \textbf{cons}(x, l), \mathit{addr}, i) \to & \\
\quad \textbf{graphToMatrixStoreEdge}(\mathit{graph}, l, \mathit{addr}, i, \textbf{setMatrix}(\mathit{addr}, x, i, \texttt{1})) & (7)\\
\textbf{graphToMatrixEdges}(\mathit{graph}, \texttt{nil}, \mathit{addr}, i) \to \textbf{graphToMatrixLoop}(\mathit{graph}, \mathit{addr}, i + \texttt{1}, \texttt{true}) & (8)\\
\textbf{graphToMatrixStoreEdge}(\mathit{graph}, l, \mathit{addr}, i, \texttt{true}) \to \textbf{graphToMatrixEdges}(\mathit{graph}, l, \mathit{addr}, i) & (9)
\end{cases}
$$

Example 13. Consider the following directed graph:



This graph is represented by the following term. In the term encoding, node labels from the figure are translated to zero-based internal indices; thus, node A is represented by `0`, node B by `1`, and so on.

$$
\begin{aligned}
&\textbf{consG}(\textbf{cons}(\texttt{1}, \textbf{cons}(\texttt{2}, \texttt{nil})),\\
&\textbf{consG}(\texttt{nil},\\
&\textbf{consG}(\textbf{cons}(\texttt{3}, \textbf{cons}(\texttt{4}, \texttt{nil})),\\
&\textbf{consG}(\texttt{nil},\\
&\textbf{consG}(\textbf{cons}(\texttt{1}, \texttt{nil}),\\
&\texttt{nilG})))))
\end{aligned}
$$

After converting the term representation of this graph, using the default value `0`, we will obtain the following matrix in the memory:

$$
\begin{pmatrix}
0 & 1 & 1 & 0 & 0\\
0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 1 & 1\\
0 & 0 & 0 & 0 & 0\\
0 & 1 & 0 & 0 & 0
\end{pmatrix}
$$

The raw memory in this case looks the following way:

[38, 0 , 5, 8, 14, 20, 26, 32 , 5, 0, 1, 1, 0, 0 , 5, 0, 0, 0, 0, 0 , 5, 0, 0, 0, 1, 1 , 5, 0, 0, 0, 0, 0 , 5, 0, 1, 0, 0, 0]

5.4 Concurrent Queues

We can also consider a more complex data structure, such as a concurrent queue. A queue is a data structure that serves as an ordered collection of objects. It supports two main operations:

- **push**, which adds one element to the back of the queue.
- **pop**, which removes one element from the front of the queue.

This data structure is useful for modelling parallel graph algorithms, such as Breadth-First Search. In MemTRS, we implement the concurrent queue using a circular array. To ensure mutually exclusive access to the queue

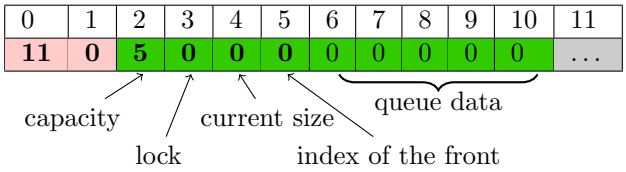
during updates, the operations are protected by a lock implemented with the atomic **CAS** operation. Thus, although the queue operations are encoded by several rewrite steps, their critical sections are serialized. Every queue stores its administrative fields in the first four memory cells.

For a queue stored at address *addr*:

- the capacity c of the queue is stored at address *addr*,
- the special lock value is stored at the address $addr + 1$,
- the current size of the queue is stored at the address $addr + 2$,
- and finally, the index of the front element is located at the address $addr + 3$.

The next c addresses starting at $addr + 4$ are reserved for the queue elements.

Example 14. Consider the empty queue stored at the address $addr = 2$ with the capacity 5. After being constructed in memory, the resulting memory layout will look like this (the memory cells occupied by the queue data structure are marked with the **green** color):



The queue occupies the addresses $2, \ldots, 10$. Addresses 0 and 1 store the administrative information about the memory allocation system. After successive insertions of the elements $5, 4, 3, 2, 1$, we obtain the following memory configurations:

Command	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
push(2, 5)	**11**	**0**	**5**	**0**	**1**	**0**	5	0	0	0	0	...
push(2, 4)	**11**	**0**	**5**	**0**	**2**	**0**	5	4	0	0	0	...
push(2, 3)	**11**	**0**	**5**	**0**	**3**	**0**	5	4	3	0	0	...
push(2, 2)	**11**	**0**	**5**	**0**	**4**	**0**	5	4	3	2	0	...
push(2, 1)	**11**	**0**	**5**	**0**	**5**	**0**	5	4	3	2	1	...

We can see that while inserting new elements to the queue, the size field of the queue is iteratively incremented (memory address $addr + 2 = 4$). Also, in this example the lock field (memory address $addr + 1 = 3$) stays equal to 0, because in the end of each queue operation the lock is released (e.g. $= 0$ in this context). If one decides to remove one element from the queue and then insert the element 6 back, the memory configuration will be updated as follows:

Command	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
pop(2) = 5	**11**	**0**	**5**	**0**	**4**	**1**	5	4	3	2	1	...
push(2, 6)	**11**	**0**	**5**	**0**	**5**	**1**	6	4	3	2	1	...

Because the queue is implemented using the circular array, the element deletion just shifted the index of the front element (memory address $addr + 3 = 5$), meaning we have $front = 1$, $size = 4$, and $cap = 5$. So after inserting the element `6` into the queue, the element is written at index $(1 + 4) \ \% \ 5 = 0$, i.e. at address $addr + 4 = 6$. In our queue implementation, popped elements are not erased from the memory immediately. The queue is determined by the pair $(front, size)$.

After all these memory manipulations, we got the following queue:

$$\overset{\text{front}}{\longleftarrow} [4, 3, 2, 1, 6] \overset{\text{back}}{\longleftarrow}$$

5.4.1 Concurrent Queue Construction

The following set of rules allows for the construction of a concurrent queue data structure. We assume that the queue can hold up to *cap* number of elements at a time. Also, similar to arrays, we assume that when called

with a positive size, the **alloc** operation guarantees a valid fresh memory block.

$\mathcal{R}_{\text{createQueue}} =$

$$\begin{cases}
\textbf{createQueue}(cap) \to \textbf{createQueueStoreCap}(\textbf{alloc}(cap + 4), cap)\ [cap > 0] & (1)\\
\textbf{createQueue}(cap) \to \texttt{-1}\ [cap \leq 0] & (2)\\
\textbf{createQueueStoreCap}(addr, cap) \to \textbf{createQueueInitRest}(addr, cap, \textbf{SET}(addr, cap))\ [addr \geq 0] & (3)\\
\textbf{createQueueStoreCap}(addr, cap) \to \texttt{-1}\ [addr < 0] & (4)\\
\textbf{createQueueInitRest}(addr, cap, \texttt{true}) \to \textbf{createQueueInitLoop}(addr, addr + 1, cap + 3) & (5)\\
\textbf{createQueueInitLoop}(queueAddr, addr, remaining) \to & \\
\quad \textbf{createQueueInitStep}(queueAddr, addr + 1, remaining - 1, \textbf{SET}(addr, 0))\ [remaining > 0] & (6)\\
\textbf{createQueueInitLoop}(queueAddr, addr, remaining) \to queueAddr\ [remaining \leq 0] & (7)\\
\textbf{createQueueInitStep}(queueAddr, addr, remaining, \texttt{true}) \to & \\
\quad \textbf{createQueueInitLoop}(queueAddr, addr, remaining) & (8)
\end{cases}$$

The rules first allocate a fresh memory block of size $cap + 4$ and store the queue capacity in its first cell. They then initialize the remaining $cap + 3$ cells to 0, thereby setting the lock value, the current size, the front index, and all queue entries to zero. After that, the address of the queue is returned.

5.4.2 Concurrent Queue Element Insertion

The following set of rules inserts the new element into the queue. Let's assume the queue has the capacity *cap*, current number of elements *size*, and the front index *front*. Then, if the queue has enough space for one more element, the element will be placed at the location $(front + size)\ \%\ cap$. Also, after executing this operation, the size of the queue should be incremented, and finally, the boolean constant `true` is returned. If there is not enough space for one more element, the operation returns the boolean constant `false`. It's important to mention that because the queue data structure is expected to be used in the parallel setting, The **push** operation blocks other accesses to the queue by using the atomic **CAS**-operation.

We assume that the address *addr* refers to a valid queue created by **createQueue**. Hence, the memory block starting at *addr* stores a positive capacity, a lock value, a current size satisfying $0 \leq size \leq cap$, and a front index satisfying $0 \leq front < cap$. Under this assumption, the position $(front + size)\ \%\ cap$ is well-defined and points to a valid queue cell.

$\mathcal{R}_{\text{push}} =$

$$\begin{cases}
\textbf{push}(addr, element) \to \textbf{pushTryLock}(addr, element, \textbf{CAS}(addr + 1, 0, 1)) & (1)\\
\textbf{pushTryLock}(addr, element, \texttt{false}) \to \textbf{push}(addr, element) & (2)\\
\textbf{pushTryLock}(addr, element, \texttt{true}) \to & \\
\quad \textbf{pushCheckSpace}(addr, element, & \\
\qquad (\textbf{GET}(addr + 3) + \textbf{GET}(addr + 2))\ \%\ \textbf{GET}(addr), \textbf{GET}(addr + 2) < \textbf{GET}(addr)) & (3)\\
\textbf{pushCheckSpace}(addr, element, rear, \texttt{true}) \to & \\
\quad \textbf{pushCommit}(addr, \textbf{SET}(addr + 4 + rear, element), \textbf{SET}(addr + 2, \textbf{GET}(addr + 2) + 1)) & \\
\qquad [rear = rear] & (4)\\
\textbf{pushCheckSpace}(addr, element, rear, \texttt{false}) \to \textbf{pushUnlockFail}(addr) & (5)\\
\textbf{pushUnlockFail}(addr) \to \textbf{SET}(addr + 1, 0) \wedge \texttt{false} & (6)\\
\textbf{pushCommit}(addr, \texttt{true}, \texttt{true}) \to \textbf{pushUnlockSuccess}(addr) & (7)\\
\textbf{pushUnlockSuccess}(addr) \to \textbf{SET}(addr + 1, 0) & (8)
\end{cases}$$

5.4.3 Concurrent Queue Element Extraction

The following set of rules extracts an element from the concurrent queue. Assume the queue has the capacity *cap*, current size *size*, and front index *front*. If the queue is non-empty, the operation reads the element stored at address $addr + 4 + front$. After that, the front index is updated to $(front + 1)\ \%\ cap$, the size is decremented, and the extracted element is returned. If the queue is empty, the operation returns the error code `-1`. As in the implementation of **push**, the critical section is protected by a lock implemented using **CAS**.

We assume that $\textbf{pop}(addr)$ is applied only to a valid queue created by **createQueue**. In particular, the memory block starting at *addr* stores a positive capacity, a current size satisfying $0 \leq size \leq cap$, and a front

index satisfying $0 \leq \mathit{front} < \mathit{cap}$. We also assume that the queue stores only non-negative values, so `-1` can be safely used as the error code for an empty queue.

$\mathcal{R}_{\text{pop}} =$

$$
\begin{cases}
\mathbf{pop}(\mathit{addr}) \to \mathbf{popTryLock}(\mathit{addr}, \mathbf{CAS}(\mathit{addr} + \mathtt{1}, \mathtt{0}, \mathtt{1})) & (1) \\
\mathbf{popTryLock}(\mathit{addr}, \mathtt{false}) \to \mathbf{pop}(\mathit{addr}) & (2) \\
\mathbf{popTryLock}(\mathit{addr}, \mathtt{true}) \to \mathbf{popCheckNonEmpty}(\mathit{addr}, \mathbf{GET}(\mathit{addr} + \mathtt{2}) > \mathtt{0}) & (3) \\
\mathbf{popCheckNonEmpty}(\mathit{addr}, \mathtt{false}) \to \mathbf{popCommit}(\mathit{addr}, \mathtt{-1}, \mathtt{true}, \mathtt{true}) & (4) \\
\mathbf{popCheckNonEmpty}(\mathit{addr}, \mathtt{true}) \to \mathbf{popReadFront}(\mathit{addr}, \mathbf{GET}(\mathit{addr} + \mathtt{4} + \mathbf{GET}(\mathit{addr} + \mathtt{3}))) & (5) \\
\mathbf{popReadFront}(\mathit{addr}, \mathit{element}) \to & \\
\quad \mathbf{popCommit}(\mathit{addr}, \mathit{element}, & \\
\quad\quad \mathbf{SET}(\mathit{addr} + \mathtt{2}, \mathbf{GET}(\mathit{addr} + \mathtt{2}) - \mathtt{1}), & \\
\quad\quad\quad \mathbf{SET}(\mathit{addr} + \mathtt{3}, (\mathbf{GET}(\mathit{addr} + \mathtt{3}) + \mathtt{1}) \,\%\, \mathbf{GET}(\mathit{addr}))) & (6) \\
\mathbf{popCommit}(\mathit{addr}, \mathit{element}, \mathtt{true}, \mathtt{true}) \to \mathbf{popUnlock}(\mathit{addr}, \mathit{element}, \mathbf{SET}(\mathit{addr} + \mathtt{1}, \mathtt{0})) & (7) \\
\mathbf{popUnlock}(\mathit{addr}, \mathit{element}, \mathtt{true}) \to \mathit{element} & (8)
\end{cases}
$$

Chapter 6

Practical Implementation of MemTRS

As a practical outcome of this project, we developed execution support for Memory Term Rewriting Systems. The first version of the implementation was organized as a single extension of the *Cora* constrained rewriting analyzer tool [22]. Later, the implementation was refactored into three separate repositories: `mikhirurg/cora` [27], `MemTRS-STDLIB` [29], and `MemTRS-REPL` [28]. The `mikhirurg/cora` repository contains the fork of *Cora* with parallel reduction support, execution support for memory term rewriting, and tests for the implemented algorithms and data structures. The `MemTRS-STDLIB` repository contains reusable encodings of algorithms and data structures in MemTRS, and `MemTRS-REPL` provides an interface for running experiments with these systems.

In the final organization of the project, the *Cora* fork acts as the execution component of the practical implementation, while the MemTRS programs and experiment interface are kept in separate repositories. The main changes in the *Cora* fork are the addition of a dedicated *MemReducer* component for the memory operations **GET**, **SET**, and **CAS**, and the extension of the *Reducer* module with a parallel reduction mode based on Java virtual threads.

6.1 Cora Fork as the Execution Component

The term rewriting engine of *Cora* is implemented by the class *Reducer*. A *TRS* object in *Cora* stores the currently enabled rule schemes together with explicit rewriting rules of the given system. In our MemTRS implementation, we included the additional rule scheme *Mem*, corresponding to the memory reduction steps of MemTRS. When a *Reducer* object is constructed from a TRS, it creates reduction components for the enabled rule schemes and then adds one *RuleReducer* for each user-defined rewrite rule. For MemTRS the most relevant components are *RuleReducer*, which executes the primitive memory operations, *CalcReducer*, which evaluates theory calculations, and *MemReducer*, which executes kernel memory operations.

The central public method *reduce(Term s)* performs one rewrite step on the input term according to the currently selected rewriting strategy and reduction mode. With the strategy *Full*, any redex may be selected. With *Innermost*, the reducer uses the existing leftmost-innermost traversal order. With *CallByValue*, only redexes whose strict subterms satisfy the Call-By-Value condition are considered. After enumerating the candidate subterms, the reducer tries the available reduction components and returns the term obtained after one successful step. If no step is possible, it returns `null`.

The companion method *normalize(Term s)* repeatedly applies the *reduce* until no further reduction is possible and records the resulting sequence of terms. Together, these methods form the operational core used by MemTRS experiments.

6.2 Shared Memory Implementation

Formally, MemTRS uses the following kind of partial memory functions:

$$mem : [int] \rightharpoonup int$$

In our extension this abstraction is implemented by a bounded global *AtomicIntegerArray* object *MEMORY*, stored in the *MemReducer*. The valid memory addresses are defined by the valid indices of the underlying array object (from the range $0 \leq addr < Settings.getMemMaxSize()$). The *AtomicIntegerArray* is a wrapper object around the generic array that ensures atomic access to the memory cells. This is essential for implementing the semantics of **CAS** in a multithreaded setting.

The *MemReducer* contains a specialized helper method *resetMemory()* that initializes the bounded memory and reserves the first two cells for the MemTRS standard library: address `0` stores the next free memory

location, and address `1` stores the allocation lock. The current solution can be safely used in a multithreaded environment, which perfectly aligns with the idea of parallel rewriting with memory functions. This Java-level memory object should be understood as a bounded implementation model used for experiments, rather than a perfect translation of the abstract partial-memory semantics.

6.3 Parallel Rewriting Implementation

Another key feature of the MemTRS is the notion of parallel reduction, introduced in Section 4.5. To support this operationally, we extended the *Reducer* with three reduction modes: *FirstMatch*, *Random*, and *Parallel*. The first two modes compute an ordinary one-step reduction, while *Parallel* computes candidate reductions for multiple positions concurrently.

In the *Parallel* mode, before performing the reduction step, the reducer first considers all the subterms of the given term and selects the candidate subterms according to the selected rewriting *strategy*. Under the *Full* reduction strategy, the reducer considers all the subterms as candidates, and processes them in an unordered way. Under *CallByValue* and *Innermost* reduction strategies the reducer processes the subterms in leftmost-innermost traversal order. However, in case of *CallByValue* reduction strategy the subterms are filtered by the method *cbvReductionOK*, and under *Innermost* strategy the reducer considers all the subterms. For each candidate position, the reducer spawns one Java virtual thread. This thread tries the available reduction components and computes at most one candidate reduct for its position.

After all tasks have been started, the main thread collects their results and keeps only a prefix-free set of positions using the data structure *PrefixFreeTrie*. Here, a set of positions is *prefix-free* if no selected position is a prefix of another one. Equivalently, the selected positions are pairwise parallel. The prefix-free *trie* is a prefix tree used to maintain exactly such a set [26]. Therefore, the accepted candidates can be rewritten simultaneously, and their reducts can be safely inserted back into the original term.

Thus, one call to *reduce* implements one parallel reduction step by greedily selecting a prefix-free set of parallel positions from concurrently computed candidates. In the *Innermost* and *CallByValue* settings, candidate positions are processed in leftmost-innermost traversal order, so this greedy construction keeps the deepest compatible candidates on each branch. Hence, in these settings, we get one of the largest-possible prefix-free subsets of the computed candidate positions. In general, however, the implementation does not claim to rewrite all redexes simultaneously. Instead, it computes many candidate steps concurrently and commits a compatible parallel subset of them.

Although the parallel mode can be combined with several rewriting strategies, the MemTRS experiments use it primarily in the Call-By-Value setting. This gives a more predictable behavior for memory programs, because a redex is reduced only after its strict subterms have already been evaluated.

We specifically chose Java virtual threads because parallel rewriting produces a large number of short-lived tasks. Platform threads are comparably expensive in this setting and are limited by the small number of available operating system threads. In contrast, virtual threads allow the reducer to explore many candidate positions at once with much lower scheduling overhead. For example, during the sorting of 128 elements in the List QuickSort system from Section 7.1.2, the *Reducer* constructs more than 62,485,326 short-lived virtual threads. Platform threads would be less suitable for this workload.

6.4 Standard Library and REPL

The reusable MemTRS encodings presented in this report are separated from the *Cora* fork and stored in the `MemTRS-STDLIB` repository. This repository contains the encodings of the data structures and algorithms used in this report, including arrays, matrices, concurrent queues, sorting algorithms, and graph algorithms. This separation makes the MemTRS systems independent from the Java implementation of the rewriting engine: new systems can be added to the standard library without modifying the *Cora* fork.

The `MemTRS-REPL` repository provides the experimentation layer. It depends on the *Cora* fork and on `MemTRS-STDLIB`, and allows the user to load MemTRS systems, execute reductions, and run experiments interactively. In this structure, the *Cora* fork implements the operational semantics, the standard library provides the MemTRS programs, and the REPL provides the user-facing experiment interface.

6.5 MemTRS Analysis

The *Cora* term rewriting analyzer contains techniques such as Dependency Pairs and Reduction Pairs for proving termination of Logically Constrained Term Rewriting Systems by encoding these tasks as SMT problems. While for ordinary LCTRSs this procedure is relatively straightforward, in the MemTRS case we have an additional challenge, as memory operations are stateful.

Initially, we tried to treat **GET**, **SET**, and **CAS** as ordinary theory symbols. However, that design allowed them to occur inside logical constraints, which introduced undesirable semantic effects. Constraint evaluation is expected to be pure, while memory operations depend on the mutable global state, and memory-modifying operations such as **SET** and **CAS** may change that state. As a consequence, checking whether a rule is applicable could already access or modify memory, even if the constraint layer evaluates to `false` and the rule itself is not applied. For this reason we decided to handle memory operations separately in the dedicated *MemReducer*, making the terms for kernel memory operations disjoint from the theory symbols.

We attempted to translate the MemTRS theory terms into the SMT; however, we encountered some challenges. The global memory representation using the SMT-LIB arrays is not straightforward: arrays are represented as functions with operators **select** and **store**. This makes it complicated to recover the array values from the generated concrete valuations. This problem can be resolved by introducing separate variables for each array element; however, this approach is also problematic: modelling large memory blocks in this way leads to a substantial increase in variables and constraints. As a result, memory usage rapidly grows, and the solver's search space explodes, which can significantly degrade performance and scalability.

Parallel rewriting introduces another difficulty. In MemTRS, one parallel step may reduce several pairwise parallel redexes that interact through the shared memory, especially in case of the operations **SET** and **CAS**. A proper SMT encoding would have to model both memory-transforming state updates and nondeterministic choice of parallel redex sets. This leads to numerous case distinctions and quickly increases the size and complexity of the resulting SMT problem encoding.

Despite the analysis challenges, the operational part of the implementation was tested on numerous examples including operations with arrays, atomic counters, concurrent queues, and matrices. We also tested the reduction engine with implementations of different algorithms from the standard library, such as QuickSort on the array, and graph algorithms such as Breadth-First Search and Floyd-Warshall. This shows that the current implementation already provides a working execution platform for MemTRS experiments, while the full integration into the automated termination analysis still remains future work.

Chapter 7

Algorithms in MemTRS

7.1 Sorting Algorithms

7.1.1 QuickSort with Memory

QuickSort (or QSort) is an efficient general-purpose sorting algorithm developed in 1959 by Tony Hoare [16]. The algorithm uses the divide-and-conquer approach based on the partitioning procedure.

Partition Procedure

The partitioning routine runs on the specified subsequence of the array. This procedure splits the array sequence into two parts where all elements of the first part are less than or equal to the special "pivot" element, and all elements of the second part are greater than that "pivot". Depending on the implementation, the "pivot" selection procedure can be done differently. Some algorithms randomly select the pivot element, while others use predefined positions. We will adopt a straightforward approach to element selection: select the last element of the sequence as the "pivot" element. So, in the end, the procedure **partitionArray** updates the given range in the array by placing all elements that are less than or equal to the "pivot" before the "pivot" element, and all elements that are greater than it after it. The **partitionArray** function returns the new position of the pivot element in the array.

The MemTRS $\mathcal{R}_{\text{partitionArray}}$ consists of rules that simulate the Lomuto-style partitioning [3]. The rule (1) prepares the partition procedure by fetching the value of the "pivot" element from the array and setting the temporary pivot index. The rules (2) – (6) define the procedure that iterates within the range $[l, r]$. The rule (2) fetches the value of the array element at the index j and ensures that the index j does not exceed the end of the range. The rule (3) checks whether the current element is smaller than or equal to the pivot. If that is not the case, it moves to the next loop iteration (rule (4)) If that is the case, it swaps the current element with the element at the temporary pivot index. And then the rule (5) increments the temporary pivot index i and moves to the next loop iteration. In case the index j exceeds the array bound r, rule (6) swaps the pivot with the last element of the range in the array. Finally, if the array swap procedure is successful, the rule (7) returns the pivot index i within the range $[l, r]$.

$\mathcal{R}_{\text{partitionArray}} =$

$$
\begin{cases}
\textbf{partitionArray}(addr, l, r) \to \textbf{partitionLoop}(addr, l, r, \textbf{getArr}(addr, r), l, l) & (1)\\
\textbf{partitionLoop}(addr, l, r, pivot, i, j) \to \textbf{partitionCheck}(addr, l, r, pivot, i, j, \textbf{getArr}(addr, j))\ [j < r] & (2)\\
\textbf{partitionCheck}(addr, l, r, pivot, i, j, jVal) \to & \\
\quad \textbf{partitionSwapStep}(addr, l, r, pivot, i, j, \textbf{swapArr}(addr, i, j))\ [jVal \leq pivot] & (3)\\
\textbf{partitionCheck}(addr, l, r, pivot, i, j, jVal) \to \textbf{partitionLoop}(addr, l, r, pivot, i, j + \texttt{1})\ [jVal > pivot] & (4)\\
\textbf{partitionSwapStep}(addr, l, r, pivot, i, j, \texttt{true}) \to \textbf{partitionLoop}(addr, l, r, pivot, i + \texttt{1}, j + \texttt{1}) & (5)\\
\textbf{partitionLoop}(addr, l, r, pivot, i, j) \to \textbf{partitionReturn}(i, \textbf{swapArr}(addr, i, r))\ [j \geq r] & (6)\\
\textbf{partitionReturn}(i, \texttt{true}) \to i & (7)
\end{cases}
$$

QuickSort Algorithm

The sorting procedure also runs on the specified subsequence of the array $[l, r]$ located at address $addr$. After sorting, the procedure returns a boolean indicating whether the algorithm completed successfully.

The MemTRS $\mathcal{R}_{\text{qsortArray}}$ consists of the rules that simulate the QuickSort algorithm. It relies on the MemTRS $\mathcal{R}_{\text{partitionArray}}$, which describes the Lomuto-style partition procedure. Rules (1) and (2) check whether the left bound is less than the right one. If so, the partition procedure is executed, yielding the pivot position q. Otherwise, the QuickSort procedure finishes and returns a boolean `true`. The rule (3) uses the pivot position q to execute the QuickSort algorithm in parallel on two subsequences $[l, q-1]$ and $[q+1, r]$. Finally, the rule (4) combines the results of sorting two subsequences (returns `true` if both of the sorting calls successfully finished).

We assume that $\mathbf{qsortArray}(addr, l, r)$ is called only for a valid array stored at the address $addr$ with bounds satisfying $0 \leq l$, $r < \mathbf{getArrSize}(addr)$, and $l \leq r + 1$.

$\mathcal{R}_{\text{qsort}} =$

$$\begin{cases} \mathbf{qsortArray}(addr, l, r) \to \texttt{true}\ [r - l \leq 0] & (1) \\ \mathbf{qsortArray}(addr, l, r) \to \mathbf{qsortSplit}(addr, l, r, \mathbf{partition}(addr, l, r))\ [r - l > 0] & (2) \\ \mathbf{qsortSplit}(addr, l, r, q) \to \mathbf{qsortMerge}(\mathbf{qsortArray}(addr, l, q - 1), \mathbf{qsortArray}(addr, q + 1, r)) & (3) \\ \mathbf{qsortMerge}(left, right) \to left \wedge right & (4) \end{cases}$$

Discussion

The presented version of the QuickSort algorithm is an in-place sorting procedure with average-case time complexity $\mathcal{O}(n \log n)$ and $\mathcal{O}(n)$ space complexity. This algorithm uses memory to store the input array, perform constant-time element lookup, and swap two elements. If one would attempt to replace arrays with term-based lists naively, the time complexity of the algorithm would significantly worsen: all arbitrary element operations would become linear instead of constant, which would make the overall time complexity $\mathcal{O}(n^2 \log n)$. So, the memory is crucial for this version of the algorithm.

7.1.2 QuickSort without Memory

We will also look at another version of the QuickSort algorithm that does not require shared memory to sort arrays of integers. This version of the algorithm is based on the "QuickSort for Linked Lists" proposed by Tom Verhoeff [30]. Here, the integer array is represented as a linked list. We select the first element of the list as the "pivot" element, and then perform the partition procedure, resulting in two new lists: the elements less than the pivot are added to the left sub-list, and the other elements are added to the right sub-list. And finally, the QuickSort procedure is called on both sub-lists.

Partition Procedure

$\mathcal{R}_{\text{partitionList}} =$

$$\begin{cases} \mathbf{partitionList}(pivot, \mathbf{cons}(x, xs), ys, zs, rest) \to \mathbf{partitionList}(pivot, xs, \mathbf{cons}(x, ys), zs, rest)\ [x < pivot] & (1) \\ \mathbf{partitionList}(pivot, \mathbf{cons}(x, xs), ys, zs, rest) \to \mathbf{partitionList}(pivot, xs, ys, \mathbf{cons}(x, zs), rest)\ [x \geq pivot] & (2) \\ \mathbf{partitionList}(pivot, \texttt{nil}, ys, zs, rest) \to \mathbf{qsortList}(ys, \mathbf{cons}(pivot, \mathbf{qsortList}(zs, rest))) & (3) \end{cases}$$

QuickSort Algorithm

$$\mathcal{R}_{\text{qsortList}} = \begin{cases} \mathbf{quicksortList}(lst) \to \mathbf{qsortList}(lst, \texttt{nil}) & (1) \\ \mathbf{qsortList}(\texttt{nil}, rest) \to rest & (2) \\ \mathbf{qsortList}(\mathbf{cons}(x, xs), rest) \to \mathbf{partitionList}(x, xs, \texttt{nil}, \texttt{nil}, rest) & (3) \end{cases}$$

Discussion

The presented version of the QuickSort algorithm is an array-free sorting procedure with average-case time complexity $\mathcal{O}(n \log n)$ and $\mathcal{O}(n)$ space complexity. This version does not require the usage of shared global memory, as it performs the sorting operations by rebuilding term-based lists.

7.1.3 Case Study: Performance Comparison of Two QuickSort Implementations

We compared the time performance of two implementations of the QuickSort algorithm. Both of the algorithms were executed on the same randomly generated lists of integers. The following input sizes were used in the experiment: $1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024$. The experiments were conducted using the *Cora* [22] term rewriting analyzer tool, specifically in the parallelized term reduction mode supporting MemTRS, which was developed for this research project.

During the experiment, we measured the overall runtime performance of the algorithms. We used the *Java Flight Recorder (JFR)* [15] tool to assess the algorithm's footprint, including the number of object allocation samples and the garbage collector's performance. Additionally, we performed *CPU-time profiling* [4] of the rewriting engine to identify the so-called "hot" methods to determine which parts of the engine are primarily used when executing sorting algorithms. Finally, we used CPU time-profiling information to identify which threads carried most of the computation, providing insight into how well different algorithms utilize parallelization. We used the *jfr-query-experiments* [6] tool to extract information from the JFR recordings.

A complete set of raw experimental results is available in the Appendix A.

Time Complexity

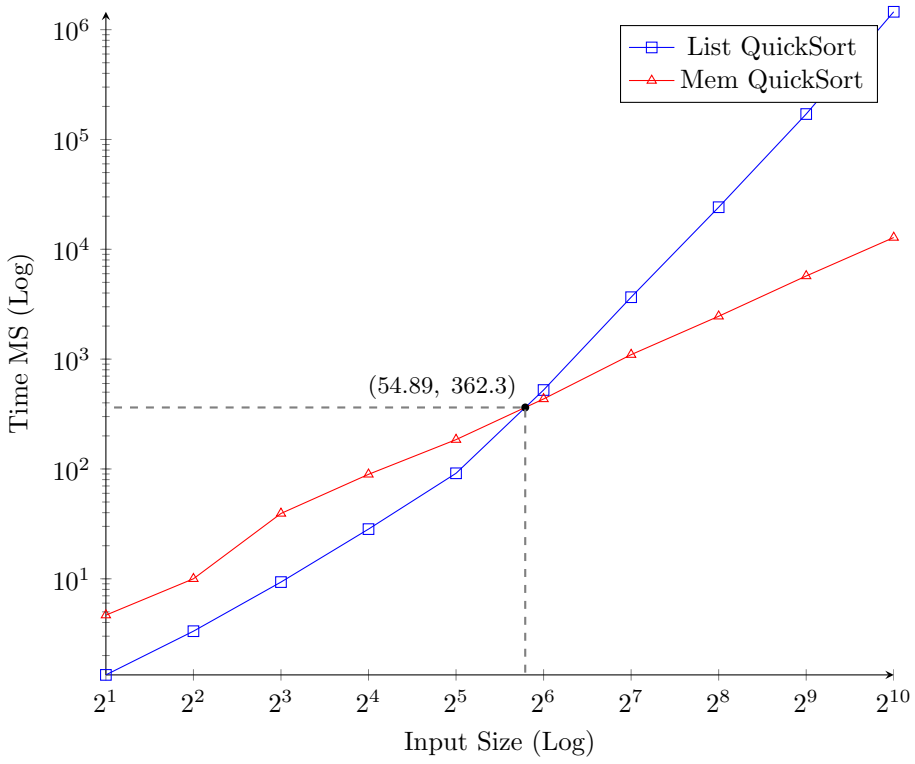


Figure 7.1: QuickSort implementations time performance comparison

The experiment shows that the list version of the QuickSort algorithm outperforms the memory version in runtime performance for input sizes less than approximately 55 elements. However, for larger input arrays, the memory-based QuickSort performs significantly better. This sudden performance drop prompted us to investigate the potential causes of this algorithmic behavior. One potential cause of performance degradation may be the architecture of the underlying term rewriting engine.

The parallel reduction procedure is organized as follows: first, the engine queries all subterms of the current term; then, depending on the rewriting strategy, it selects them. For the *"CallByValue"* rewrite strategy, the engine filters the subterms that can be safely used in Call-By-Value reduction.

Proposition 1. *A term can be safely reduced using the Call-By-Value strategy if:*

- *Its strict subterms are either:*
  - *variables,*
  - *values, or*

– *they have a form* $f(s_1, \ldots, s_n)$*, where* f *is a constructor or* $n < arity(f)$

During this procedure, the verification method constructs a linked list and processes all the nested subterms of the given term. This is quite a heavy operation: it requires creating a linked list for every subterm of the term under rewriting. Additionally, the linked list itself creates many new residual objects during common operations, such as adding a new element. Specifically, linked lists use special "Node" wrappers around every element object that are added to the list, and each of the wrappers is allocated on the heap. As a result, we get a substantial number of additional objects constructed during subterm processing.

In practice, it means that the larger the term expressed in the rewriting engine is, the more operations it will require to verify the Call-By-Value properties of the term (we have more subterms to process), and the more residual objects for that term will be created.

Garbage Collector Performance

We used the JFR tool to estimate the allocation pressure during the execution of the sorting procedure for each input size. Since our query uses the event `ObjectAllocationSample`, the reported values should be interpreted as sampled allocation events rather than the exact total number of created objects.

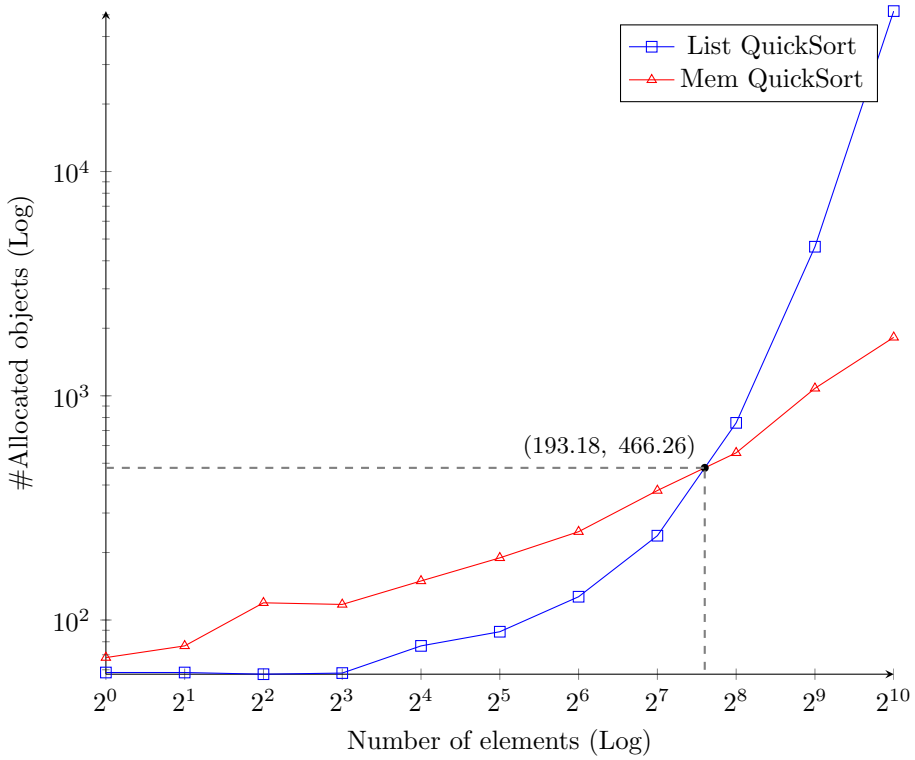


Figure 7.2: QuickSort implementations allocation sample comparison

Request for the *jfr-query-experiments* tool:

```
COLUMN 'Allocation Samples'
FORMAT none
SELECT COUNT(*) FROM ObjectAllocationSample
```

The experiment shows that for input sizes greater than approximately 194 elements, the list-based QuickSort algorithm produces significantly more allocation samples during execution. This can be explained by the fact that all array elements are represented as subterms, and during sorting, the same number of elements is preserved in the term. Which means that, in every reduction step, we have to run the Call-By-Value subterm verification procedure on all subterms of a term that store the input array data. Since we represent the input arrays as nested terms (using the **cons**-constructor), each subterm of the list construction is essentially a sublist of an input list.

Consider we have to verify all the subterms of a term that stores the list of size n: that means we need to consider $\mathcal{O}(n)$ subterms (each of size $n, n-1, n-2..., 1$). Each Call-By-Value procedure executed for the term of size n also has the complexity of $\mathcal{O}(n)$, which gives us an overall complexity of $\mathcal{O}(n^2)$ during each reduction step. This procedure also produces a significant number of short-lived objects (which scales as $\mathcal{O}(n^2)$).

In the memory version of the QuickSort algorithm, the array elements are stored in a special global memory array. This means the size of the term at each reduction step is independent of the input size. Essentially, we do not store the input array in the term. This explains why the memory version of QuickSort produces a much smaller number of allocation samples than the list-based version, and why this number grows significantly more slowly with the input size. Also, this means that at every reduction step, the overhead of verifying the Call-By-Value condition for subterms is notably lower than in the list-based version.

This information about the observed allocation pressure during the algorithm runtime inspired us to investigate how the large number of shortly living objects can influence the performance of algorithms when executed in the Java Virtual Machine environment. It can be measured by recording the total garbage collector pause, a temporary suspension of the application execution while the garbage collector reclaims the unused memory.

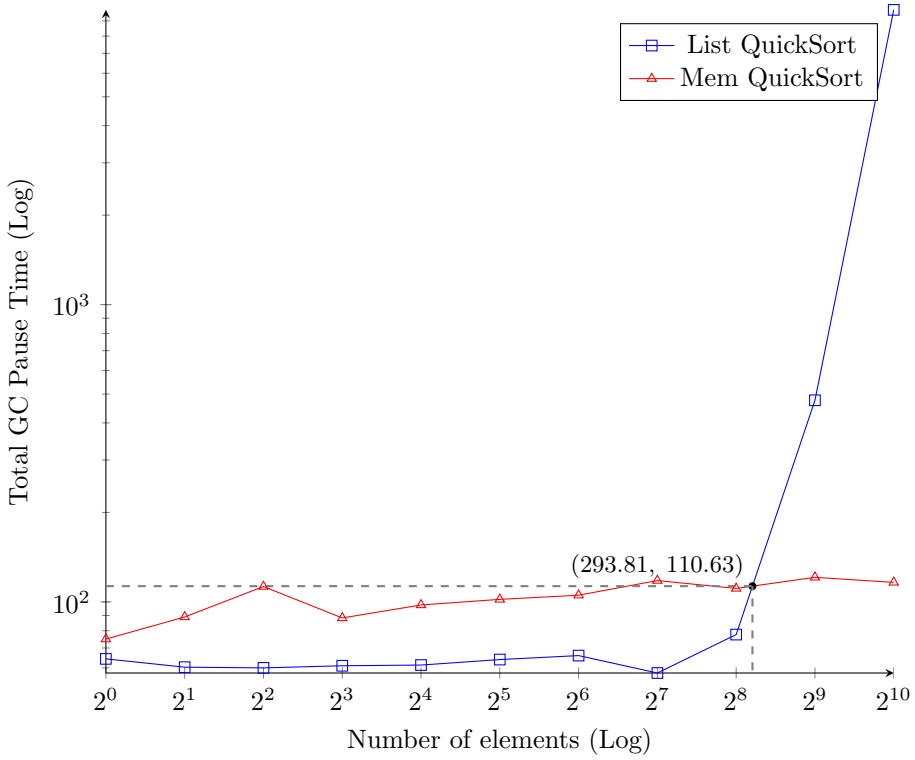


Figure 7.3: QuickSort implementation total GC pause time comparison

Request for the *jfr-query-experiments* tool:

```
COLUMN 'Total GC Pause'
FORMAT none
SELECT SUM(duration) FROM GCPhasePause
```

The experiments showed that after approximately 294 elements, the Total GC pause time of the list version of the QuickSort algorithm became significantly larger than in the case of the memory version. This shows that the larger number of created objects indeed influences the algorithm's performance.

CPU-Time Profiling

We decided to perform CPU-Time Profiling to confirm our hypothesis about the implementation specifics of the rewriting engine. This profiling technique samples every computation thread every n milliseconds of CPU time, so we essentially know which function was executed at every timestamp.

We ran the entire test suite, which consisted of sorting input arrays of sizes [1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024], 3 times each, under the Java Flight Recorder. The test suite was executed for both QuickSort implementations. Using this information, we collected 100 (method, thread) pairs with the largest number of recorded samples. This way, we can clearly see the methods and threads responsible for the highest CPU consumption during the suite execution. And after that, we collected the execution thread statistics to see which threads are carrying most of the execution.

The full tables are available in the Appendix A section. Here, we will only consider the methods directly related to the rewriting engine.

CPU-Time Profiling: List QuickSort

Table 7.1: CPU-Time Profiling: List QuickSort (Only Rewriting Engine Methods)

#	Method	Thread	Samples	Percent
1	**cora.reduction.Reducer.cbvReductionOK(Term)**	main	12,839	16.51%
2	**charlie.terms.Application.querySubterms()**	main	9,108	11.71%
4	**charlie.terms.Application.queryArgument(int)**	main	5,535	7.12%
7	**charlie.util.Pair.<init>(Object, Object)**	main	2,258	2.90%
9	**charlie.terms.TermInherit.calculateFreeReplaceablesFo ReplaceableList)**	main	1,423	1.83%
10	**charlie.terms.Application.construct(Term, List)**	main	1,153	1.48%
14	**charlie.terms.ValueInherit.isVariable()**	main	988	1.27%
17	**charlie.terms.Application.numberArguments()**	main	937	1.20%
21	**charlie.terms.Application.match(Term, Substitution)**	*virtual*	767	0.99%
22	**charlie.terms.TermInherit .calculateBoundVariable-sAndRefreshSubs(List, ReplaceableList, Replace-ableList, ImmutableList$Builder)**	*virtual*	746	0.96%
23	**charlie.terms.VariableEnvironment.iterator()**	main	674	0.87%
24	**charlie.terms.TermInherit.isValue()**	main	622	0.80%
25	**charlie.terms.TermInherit.isVariable()**	main	615	0.79%
28	**cora.reduction.RuleReducer.applicable(Term)**	*virtual*	542	0.70%
30	**charlie.terms.ValueInherit.isValue()**	main	481	0.62%
33	**charlie.util.Pair.fst()**	main	435	0.56%
34	**cora.reduction.Reducer.lambda$reduce$2(Term, Po-sition)**	*virtual*	413	0.53%
35	**charlie.util.Pair.snd()**	main	368	0.47%
37	**charlie.terms.position.ArgumentPos.<init>(int, Po-sition)**	main	341	0.44%
40	**charlie.terms.Application.<init>(Term, List)**	*virtual*	292	0.38%
41	**charlie.terms.TermInherit.vars()**	main	287	0.37%
44	**charlie.terms.TermPrinter.print(Term, Renaming, StringBuilder)**	main	267	0.34%
45	**charlie.terms.ReplaceableList.iterator()**	main	249	0.32%
56	**charlie.terms.Application.queryRoot()**	main	171	0.22%
59	**cora.reduction.RuleReducer.findHeadAdditions(Term)**	*virtual*	152	0.20%
62	**charlie.terms.Application.queryType()**	*virtual*	142	0.18%
64	**cora.reduction.BetaReducer.applicable(Term)**	*virtual*	135	0.17%
71	**charlie.terms.Var.match(Term, Substitution)**	*virtual*	115	0.15%
75	**cora.reduction.CalcReducer.applicable(Term)**	*virtual*	101	0.13%
79	**charlie.terms.ValueInherit.queryType()**	*virtual*	96	0.12%
81	**charlie.terms.Application .queryImmediateHeadSub-term(int)**	*virtual*	94	0.12%
82	**charlie.terms.Constant.alphaEquals(Term, Map, Map, int)**	*virtual*	93	0.12%
84	**cora.reduction.Reducer.lambda$reduce$1(Pair)**	main	91	0.12%
88	**cora.reduction.Reducer.reduce(Term)**	main	86	0.11%
90	**charlie.trs.Rule.queryType()**	*virtual*	81	0.10%
91	**charlie.terms.Application.setupReplaceables(List)**	*virtual*	81	0.10%
92	**charlie.terms.TermInherit.hashCode()**	*virtual*	78	0.10%
93	**charlie.terms.Subst.extend(Replaceable, Term)**	*virtual*	76	0.10%
99	**cora.reduction.MemReducer.applicable(Term)**	*virtual*	64	0.08%
100	**cora.reduction.MemReducer.checkIfGET(Term)**	*virtual*	64	0.08%

The CPU-Profiling shows that a substantial number of samples were recorded in the methods `cbvReductionOK(Term)`, `querySubterms()`, and `queryArgument(int)`. This means that, when running the list version of the QuickSort algorithm, we indeed spend a lot of CPU time filtering subterms to determine whether they are Call-By-Value reducible. Also, we can see that the majority of the computation is happening on the "main" thread, which means we don't have a good parallelization of the rewriting engine in this case. We can confirm this by grouping

the recorded samples by threads, which gives us the following picture:

Table 7.2: Thread Statistics: List QuickSort

#	Thread	Samples	Percent
1	main	51,010	65.58%
2	*virtual*	13,992	17.99%
3	ForkJoinPool-1-worker-9	1,137	1.46%
4	ForkJoinPool-1-worker-11	1,108	1.42%
5	ForkJoinPool-1-worker-12	1,097	1.41%
6	ForkJoinPool-1-worker-2	1,068	1.37%
7	ForkJoinPool-1-worker-10	1,065	1.37%
8	ForkJoinPool-1-worker-1	1,064	1.37%
9	ForkJoinPool-1-worker-7	1,056	1.36%
10	ForkJoinPool-1-worker-3	1,045	1.34%
11	ForkJoinPool-1-worker-8	1,031	1.33%
12	ForkJoinPool-1-worker-4	1,025	1.32%
13	ForkJoinPool-1-worker-5	1,020	1.31%
14	ForkJoinPool-1-worker-6	1,000	1.29%
15	RMI TCP Connection(idle)	46	0.06%
16	JFR Shutdown Hook	9	0.01%
17	JFR Periodic Tasks	7	0.01%
18	Attach Listener	5	0.01%
19	JMX server connection timeout 21049	1	0.00%

Requests for the *jfr-query-experiments* tool:

CPU-Time Profiling:

```
COLUMN 'Method', 'Thread', 'Samples', 'Percent'
FORMAT none, missing:virtual, none, normalized
SELECT stackTrace.topFrame AS T, eventThread.osName, COUNT(*), COUNT(*)
FROM CPUTimeSample GROUP BY T
```

Thread Statistics:

```
COLUMN 'Thread', 'Samples', 'Percent'
FORMAT missing:virtual, none, normalized
SELECT eventThread.osName as T, COUNT(*), COUNT(*)
FROM CPUTimeSample GROUP BY T
```

As we can see, over than 65% of CPU time samples come from the "main" thread. Moreover, only 17.99% of samples come from the virtual threads where the engine performs the parallel reduction; the remaining samples account for the administrative methods of the "ForkJoinPool" that help schedule and synchronize the virtual threads.

CPU-Time Profiling: Memory QuickSort

Table 7.3: CPU-Time Profiling: Memory QuickSort (Only Rewriting Engine Methods)

#	Method	Thread	Samples	Percent
2	**cora.reduction.RuleReducer.applicable(Term)**	*virtual*	806	5.89%
4	**charlie.terms.TermInherit .calculateFreeReplaceablesForSubterms(List, ReplaceableList)**	main	571	4.17%
5	**charlie.trs.Rule.queryType()**	*virtual*	558	4.08%
6	**charlie.terms.Application.construct(Term, List)**	main	545	3.98%
7	**charlie.terms.Application.match(Term, Substitution)**	*virtual*	484	3.54%
9	**cora.reduction.Reducer.lambda$reduce$2(Term, Position)**	*virtual*	474	3.47%

#	Method	Thread	Samples	Percent
11	**charlie.terms.TermInherit .calculateBoundVariablesAndRefreshSubs (List, ReplaceableList, ReplaceableList, ImmutableList$Builder)**	main	311	2.27%
14	**cora.reduction.RuleReducer .findHeadAdditions(Term)**	*virtual*	261	1.91%
16	**charlie.terms.ValueInherit.numberArguments()**	*virtual*	186	1.36%
18	**charlie.terms.Application.queryType()**	main	146	1.07%
19	**charlie.terms.Application.queryArgument(int)**	main	135	0.99%
24	**charlie.terms.TermPrinter.print(Term, Renaming, StringBuilder)**	*virtual*	123	0.90%
30	**charlie.terms.Application.<init>(Term, List)**	main	93	0.68%
31	**charlie.terms.ValueInherit.queryType()**	main	92	0.67%
32	**charlie.terms.Application.querySubterms()**	main	91	0.67%
35	**cora.reduction.CalcReducer.applicable(Term)**	*virtual*	88	0.64%
36	**charlie.terms.ValueInherit.isApplication()**	*virtual*	86	0.63%
37	**charlie.terms.Subst.<init>()**	*virtual*	81	0.59%
40	**charlie.terms.FunctionSymbol .compareTo(FunctionSymbol)**	main	64	0.47%
41	**charlie.terms.Application.numberArguments()**	main	62	0.45%
44	**cora.reduction.Reducer.cbvReductionOK(Term)**	main	57	0.42%
45	**charlie.terms.LeafTermInherit.queryType()**	main	53	0.39%
46	**charlie.terms.ValueInherit .queryImmediateHeadSubterm(int)**	*virtual*	53	0.39%
49	**charlie.terms.Var.match(Term, Substitution)**	*virtual*	49	0.36%
54	**charlie.terms.ValueInherit.queryType()**	main	40	0.29%
55	**charlie.terms.Application .queryImmediateHeadSubterm(int)**	*virtual*	40	0.29%
57	**charlie.terms.TermInherit.match(Term)**	*virtual*	38	0.28%
58	**charlie.terms.ReplaceableList .combine(ReplaceableList)**	main	37	0.27%
59	**cora.reduction.BetaReducer.applicable(Term)**	*virtual*	35	0.26%
61	**charlie.types.TypePrinter.print(Type, StringBuilder)**	main	34	0.25%
65	**charlie.terms.ReplaceableList.size()**	main	31	0.23%
77	**cora.reduction.MemReducer.checkIfGET(Term)**	*virtual*	27	0.20%
80	**charlie.terms.Subst.extend(Replaceable, Term)**	*virtual*	25	0.18%
82	**charlie.terms.position.ArgumentPos.toString()**	main	24	0.18%
83	**charlie.util.Pair.<init>(Object, Object)**	main	24	0.18%
84	**charlie.terms.TermInherit .setVariables(ReplaceableList, ReplaceableList)**	main	23	0.17%
87	**charlie.terms.TermPrinter.printInfixHelper(Term, Renaming, StringBuilder, int)**	*virtual*	23	0.17%
88	**charlie.terms.Application.setupReplaceables(List)**	*virtual*	22	0.16%
89	**charlie.types.Base$$TypeSwitch .0x000000006f0e4000.typeSwitch(Object, int)**	*virtual*	22	0.16%
90	**charlie.terms.TermPrinter .generateUniqueNaming(List)**	*virtual*	22	0.16%
92	**cora.reduction.MemReducer.applicable(Term)**	*virtual*	21	0.15%
94	**charlie.trs.Rule.queryLeftSide()**	*virtual*	21	0.15%
97	**charlie.terms.ValueInherit.queryRoot()**	*virtual*	20	0.15%
100	**charlie.terms.TermInherit.equals(Term)**	*virtual*	20	0.15%

The CPU-Time profiling shows that the sample distribution is much more balanced than in the List QuickSort case. Relatively high number of samples come from the methods `RuleReducer.applicable(Term)`, `TermInherit.calculateFreeReplaceablesForSubterms(List, ReplaceableList)`, `Rule.queryType()`, and `Application.construct(Term, List)`. The `TermInherit.calculateFreeReplaceablesForSubterms(...)` and `Application.construct(...)` methods run on the "main" thread and related to the procedure of collecting reduction results from the parallel threads and applying them to the original term. During this procedure,

the rewriting engine replaces the reduced subterms on their positions in the original term. Because the terms in *Cora* are immutable, each "replacement" of the subterm of a term triggers the construction of the new term. In our case, we see many "Application" objects being created. This also explains the high number of CPU samples from the method `TermInherit.calculateFreeReplaceablesForSubterms(...)`, as it participates in "Application" objects creation.

However, most other methods come from the virtual threads, meaning they are directly connected to the parallel reduction step. If we look at the Call-By-Value reduction subterm verification, the method `cbvReductionOK(Term)` is the 44th most sampled method (no other threads are executing this method), accounting for only 0.42% of all samples. Which means the subterm Call-By-Value reduction filtering consumes far fewer resources than in the List QuickSort case.

Also, we can see that most computations occur on the virtual threads. This suggests that we have a good parallelization of the rewriting procedure. We can confirm this by grouping the recorded samples by threads, which gives us the following picture:

Table 7.4: Thread Statistics: Memory QuickSort

#	**Thread**	**Samples**	**Percent**
1	*virtual*	8,692	63.55%
2	main	2,394	17.50%
3	ForkJoinPool-1-worker-2	238	1.74%
4	ForkJoinPool-1-worker-11	235	1.72%
5	ForkJoinPool-1-worker-8	233	1.70%
6	ForkJoinPool-1-worker-7	231	1.69%
7	ForkJoinPool-1-worker-12	216	1.58%
8	ForkJoinPool-1-worker-4	215	1.57%
9	ForkJoinPool-1-worker-5	213	1.56%
10	ForkJoinPool-1-worker-3	212	1.55%
11	ForkJoinPool-1-worker-10	211	1.54%
12	ForkJoinPool-1-worker-9	206	1.51%
13	ForkJoinPool-1-worker-6	184	1.35%
14	ForkJoinPool-1-worker-1	184	1.35%
15	RMI TCP Connection(1)-192.168.50.145	12	0.09%
16	JFR Shutdown Hook	1	0.01%

As we see, 63.55% of CPU time samples come from the virtual threads. Only 17.50% of samples come from the "main" thread, indicating decent parallelization.

Total Parallel Reduction Steps Comparison

Finally, we investigated whether the two QuickSort implementations exhibit the same average-case asymptotic behaviour. To this end, we repeated the runtime performance experiment, but instead of measuring wall-clock time, we recorded the total number of parallel reduction steps performed during sorting.

We implemented this measurement by adding a counter to the rewriting engine and incrementing it after each call to the `reduce(Term s)` method.

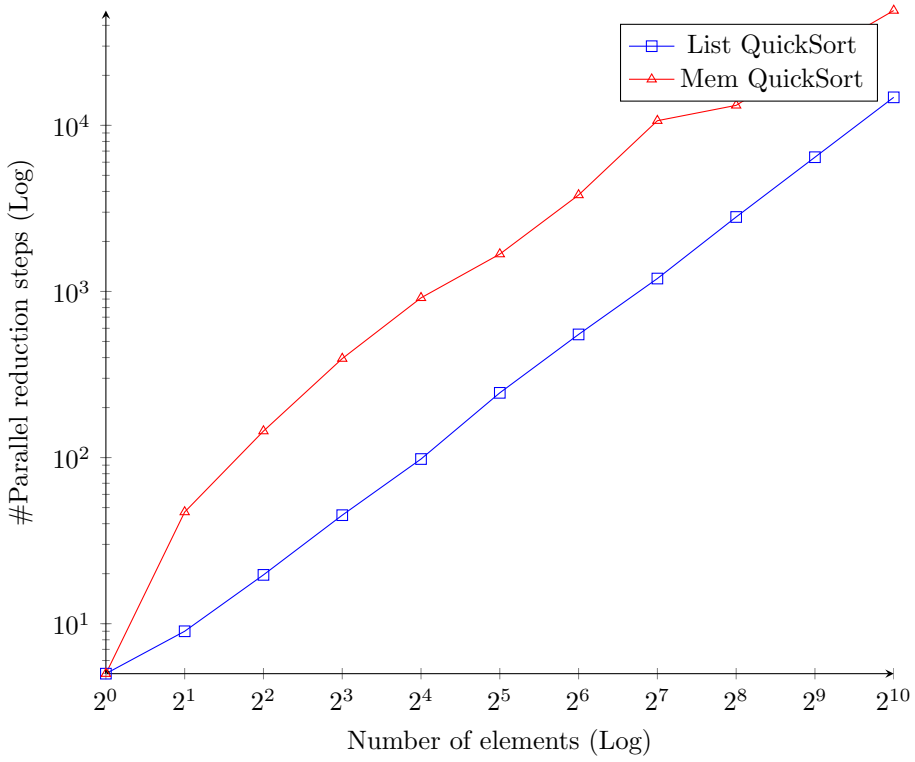


Figure 7.4: QuickSort implementations total parallel reduction steps comparison

The results suggest that both implementations have the same asymptotic growth rate with respect to the number of parallel reduction steps. However, the list-based QuickSort implementation has a smaller constant factor. This can be explained by the fact that array operations in the memory-based implementation require several reduction steps, whereas the corresponding operations in the list-based implementation are expressed more directly.

Conclusion

In conclusion, we can see that while both versions of the QuickSort algorithm have the same average-case asymptotic complexity (which we confirmed in a separate experiment), the specifics of the term rewriting engine implementation give a difference in the algorithm's practical performance. The experiment shows that for larger input sizes, the memory-based QuickSort algorithm performs better in the *Cora* term rewriting engine. The CPU-Time profiling technique allowed us to confirm our hypothesis about the inefficiencies in the rewriting engine. The Call-By-Value subterm filtering procedure consumes significant resources. Because in the case of the List QuickSort, we are operating with much bigger terms, we can see a direct influence of the rewriting engine performance.

As a result, we can conclude that with the current implementation of the term rewriting engine, the memory version of the QuickSort algorithm performs better than the list version. However, we cannot say anything about the inherent difficulty of the List QuickSort versus the memory-based QuickSort, since based on our experiment, we cannot conclude whether the different implementation of the underlying term rewriting engine would avoid this problem entirely (the investigation of this claim is beyond the scope of this research project).

7.2 Graph Algorithms in MemTRS

In the previous section, we showed that MemTRS can be used to create effective implementations of array sorting algorithms. However, we have not yet explored shared memory in practice. While the QuickSort algorithm's memory implementation uses global memory to access and update array elements, the parallel sorting procedure is applied to disjoint parts of the array. So, we cannot really see the benefit of shared memory being modified in one computation "thread" and then accessed in another. Some well-parallelizable graph algorithms, like Breadth-First Search (BFS), or All-Pair-Shortest-Paths (APSP) algorithms, seem to give us an opportunity to explore the benefits of shared memory.

7.2.1 Breadth-First Search (BFS)

Breadth-First Search [9] is an algorithm for searching a graph data structure for a node satisfying some property. There are multiple modifications of the original search algorithm allowing for extracting specific information about the given graph structure. For example, one can use the BFS algorithm to discover all nodes reachable from a given starting node. Or, in the case of the non-weighted graph, one can calculate the distances from the starting node to all other nodes in the graph.

The general idea of the algorithm is the following: while running, we maintain a queue q of nodes to be processed and an array *used* that stores information about already discovered nodes. We start from some initial node s and place it in the queue. Then, at each iteration of the algorithm, we poll the node v from the queue q, and inspect all outgoing edges (v, u). Whenever we encounter a node u that has not been discovered yet, we mark it in *used* and place it in the queue q. The algorithm terminates when the queue q is empty.

Parallel BFS with Memory

First, we consider the parallel implementation of the BFS algorithm that uses shared memory. The given version of the algorithm allows for discovering all the nodes in the graph accessible from some starting node *start*. The presented implementation is using the graph structure from the subsection 5.3.7, array helper methods from the section 5.2, and concurrent queue functions from the section 5.4.

Our implementation of the parallel BFS algorithm is similar to other implementations presented in [7], [24], and [8]. The MemTRS algorithm processes nodes at a single layer in parallel, accounting for level synchronization. While executing the algorithm, we maintain two queues: one holds nodes to be processed on the current layer, and the other collects nodes for the next layer. We process the nodes of the current layer in parallel, using special **bfsWorker**-terms. Current implementation allows scheduling up to 10 **bfsWorker**-"threads" in parallel.

Generally, the parallel process scheduling can be implemented in various ways. Another design choice is to use iterative scheduling for an unlimited number of parallel processes. In some way, that design choice is more flexible, allowing for higher parallelism. However, the downside of this approach is that parallel task execution becomes less uniform (since the start time of each new task is delayed). This makes the complexity analysis of the algorithms less predictable. To improve uniformity and the predictability of experimental results, we decided to use a straightforward approach with static parallelism (limited to 10 **bfsWorker**-threads).

During scheduling (rule 4), we extract graph nodes from the current layer queue. After that, **bfsWorker**s traverse the graph structure in parallel by exploring the outgoing edges, marking the visited nodes in the shared array, and updating the shared queue associated with the next layer (rules $6-10$). After finishing the work, the **bfsWorker** can either return `1` if the operation was successful, or it returns `0` - for example, when the current layer queue does not have any nodes, and the operation cannot be scheduled (this comes from the behavior of the **pop** operation on the concurrent queue - if the queue is empty, the operation returns the status code `-1`).

If all the workers finished execution with status code `1`, we attempt to process the current layer again (in case 10 **bfsWorker**s weren't enough) (rule 13). Otherwise, if any worker finished with status code `0`, we have finished processing the current layer and can process the next layer (rule 14). And finally, if none of the **bfsWorker**-"threads" returned the status code `1`, then we have finished processing the whole graph (rule 15).

In the end, the algorithm returns the address of the array that stores the information about the visited nodes. In the parallel MemTRS encoding below, we use a mark-on-discovery variant of the BFS algorithm, where a node is marked as soon as it is successfully discovered and added to the queue.

$\mathcal{R}_{\text{parBFS}} =$

$$
\begin{cases}
\textbf{parBFS}(\mathit{graph}, \mathit{start}) \to \textbf{parBFSAlloc}(\mathit{graph}, \mathit{start}, \textbf{numNodes}(\mathit{graph})) & (1) \\
\textbf{parBFSAlloc}(\mathit{graph}, \mathit{start}, \mathit{size}) \to & \\
\quad \textbf{parBFSSeed}(\mathit{graph}, \mathit{start}, \textbf{createArray}(\mathit{size}), & \\
\qquad \textbf{createQueue}(\mathit{size} * \mathit{size} + \mathtt{1}), \textbf{createQueue}(\mathit{size} * \mathit{size} + \mathtt{1}))\ [\mathit{size} \geq \mathtt{0}] & (2) \\
\textbf{parBFSAlloc}(\mathit{graph}, \mathit{start}, \mathit{size}) \to \mathtt{-1}\ [\mathit{size} < \mathtt{0}] & (3) \\
\textbf{parBFSSeed}(\mathit{graph}, \mathit{start}, \mathit{array}, \mathit{queue_1}, \mathit{queue_2}) \to & \\
\quad \textbf{parBFSLayer}(\mathit{graph}, \mathit{array}, \mathit{queue_1}, \mathit{queue_2}, & \\
\qquad \textbf{push}(\mathit{queue_1}, \mathit{start}), \textbf{SET}(\mathit{array} + \mathtt{1} + \mathit{start}, \mathtt{1})) & (4) \\
\textbf{parBFSLayer}(\mathit{graph}, \mathit{array}, \mathit{queue_1}, \mathit{queue_2}, \mathtt{true}, \mathtt{true}) \to & \\
\quad \textbf{parBFSSync}(\mathit{graph}, \mathit{array}, \mathit{queue_1}, \mathit{queue_2}, & \\
\qquad \textbf{bfsWorker}(\mathit{graph}, \mathit{array}, \mathit{queue_2}, \textbf{pop}(\mathit{queue_1})), & \\
\qquad \textbf{bfsWorker}(\mathit{graph}, \mathit{array}, \mathit{queue_2}, \textbf{pop}(\mathit{queue_1})), & \\
\qquad \textbf{bfsWorker}(\mathit{graph}, \mathit{array}, \mathit{queue_2}, \textbf{pop}(\mathit{queue_1})), & \\
\qquad \textbf{bfsWorker}(\mathit{graph}, \mathit{array}, \mathit{queue_2}, \textbf{pop}(\mathit{queue_1})), & \\
\qquad \textbf{bfsWorker}(\mathit{graph}, \mathit{array}, \mathit{queue_2}, \textbf{pop}(\mathit{queue_1})), & \\
\qquad \textbf{bfsWorker}(\mathit{graph}, \mathit{array}, \mathit{queue_2}, \textbf{pop}(\mathit{queue_1})), & \\
\qquad \textbf{bfsWorker}(\mathit{graph}, \mathit{array}, \mathit{queue_2}, \textbf{pop}(\mathit{queue_1})), & \\
\qquad \textbf{bfsWorker}(\mathit{graph}, \mathit{array}, \mathit{queue_2}, \textbf{pop}(\mathit{queue_1})), & \\
\qquad \textbf{bfsWorker}(\mathit{graph}, \mathit{array}, \mathit{queue_2}, \textbf{pop}(\mathit{queue_1})), & \\
\qquad \textbf{bfsWorker}(\mathit{graph}, \mathit{array}, \mathit{queue_2}, \textbf{pop}(\mathit{queue_1}))) & (5) \\
\textbf{bfsWorker}(\mathit{graph}, \mathit{array}, \mathit{queue}, x) \to \mathtt{0}\ [x < \mathtt{0}] & (6) \\
\textbf{bfsWorker}(\mathit{graph}, \mathit{array}, \mathit{queue}, x) \to \textbf{bfsWorkerLoadAdj}(\mathit{graph}, \mathit{array}, \mathit{queue}, x)\ [x \geq \mathtt{0}] & (7) \\
\textbf{bfsWorkerLoadAdj}(\mathit{graph}, \mathit{array}, \mathit{queue}, x) \to & \\
\quad \textbf{bfsWorkerScanAdj}(\mathit{graph}, \mathit{array}, \mathit{queue}, x, \textbf{getAdj}(x, \mathit{graph})) & (8) \\
\textbf{bfsWorkerScanAdj}(\mathit{graph}, \mathit{array}, \mathit{queue}, x, \textbf{cons}(u, \mathit{newNodes})) \to & \\
\quad \textbf{bfsWorkerClaim}(\mathit{graph}, \mathit{array}, \mathit{queue}, x, \textbf{cons}(u, \mathit{newNodes}), \textbf{CAS}(\mathit{array} + \mathtt{1} + u, \mathtt{0}, \mathtt{1})) & (9) \\
\textbf{bfsWorkerClaim}(\mathit{graph}, \mathit{array}, \mathit{queue}, x, \textbf{cons}(u, \mathit{newNodes}), \mathtt{true}) \to & \\
\quad \textbf{bfsWorkerEnqueue}(\mathit{graph}, \mathit{array}, \mathit{queue}, x, \mathit{newNodes}, \textbf{push}(\mathit{queue}, u)) & (10) \\
\textbf{bfsWorkerClaim}(\mathit{graph}, \mathit{array}, \mathit{queue}, x, \textbf{cons}(u, \mathit{newNodes}), \mathtt{false}) \to & \\
\quad \textbf{bfsWorkerScanAdj}(\mathit{graph}, \mathit{array}, \mathit{queue}, x, \mathit{newNodes}) & (11) \\
\textbf{bfsWorkerEnqueue}(\mathit{graph}, \mathit{array}, \mathit{queue}, x, \mathit{newNodes}, \mathtt{true}) \to & \\
\quad \textbf{bfsWorkerScanAdj}(\mathit{graph}, \mathit{array}, \mathit{queue}, x, \mathit{newNodes}) & (12) \\
\textbf{bfsWorkerScanAdj}(\mathit{graph}, \mathit{array}, \mathit{queue}, x, \mathtt{nil}) \to \mathtt{1} & (13) \\
\textbf{parBFSSync}(\mathit{graph}, \mathit{array}, \mathit{queue_1}, \mathit{queue_2}, c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8, c_9, c_{10}) \to & \\
\quad \textbf{parBFSLayer}(\mathit{graph}, \mathit{array}, \mathit{queue_1}, \mathit{queue_2}, \mathtt{true}, \mathtt{true}) & \\
\qquad [c_1 + c_2 + c_3 + c_4 + c_5 + c_6 + c_7 + c_8 + c_9 + c_{10} = \mathtt{10}] & (14) \\
\textbf{parBFSSync}(\mathit{graph}, \mathit{array}, \mathit{queue_1}, \mathit{queue_2}, c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8, c_9, c_{10}) \to & \\
\quad \textbf{parBFSLayer}(\mathit{graph}, \mathit{array}, \mathit{queue_2}, \mathit{queue_1}, \mathtt{true}, \mathtt{true}) & \\
\qquad [(c_1 + c_2 + c_3 + c_4 + c_5 + c_6 + c_7 + c_8 + c_9 + c_{10} < \mathtt{10}) \wedge & \\
\qquad (c_1 + c_2 + c_3 + c_4 + c_5 + c_6 + c_7 + c_8 + c_9 + c_{10} > \mathtt{0})] & (15) \\
\textbf{parBFSSync}(\mathit{graph}, \mathit{array}, \mathit{queue_1}, \mathit{queue_2}, c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8, c_9, c_{10}) \to \mathit{array} & \\
\quad [c_1 + c_2 + c_3 + c_4 + c_5 + c_6 + c_7 + c_8 + c_9 + c_{10} = \mathtt{0}] & (16)
\end{cases}
$$

Sequential BFS without Memory

We will also look at another version of the BFS algorithm for searching the reachable nodes. This version does not use the shared memory to keep track of the visited nodes. This design choice also has implications for the algorithm's parallelization. Since we are not using global memory, there is no trivial way to synchronize information about visited nodes across parallel instances of the BFS algorithm. Because of this, we decided to consider the sequential version of the BFS algorithm.

The following set of rewriting rules follows the same idea as the parallel algorithm. The only difference is

that we replace the array of visited nodes in the global memory with the linked list *visited*. Also, the nodes in the queue q are processed iteratively rather than in parallel.

$\mathcal{R}_{\text{seqBFS}} =$

$$\begin{cases}
\textbf{seqBFS}(\mathit{graph}, v) \to \textbf{seqBFSQueue}(\mathit{graph}, \textbf{cons}(v, \texttt{nil}), \textbf{genList}(\textbf{numNodes}(\mathit{graph}))) & (1)\\
\textbf{seqBFSQueue}(\mathit{graph}, \texttt{nil}, \mathit{visited}) \to \mathit{visited} & (2)\\
\textbf{seqBFSQueue}(\mathit{graph}, \textbf{cons}(x, q), \mathit{visited}) \to \textbf{seqBFSVisit}(\mathit{graph}, x, q, \mathit{visited}) & (3)\\
\textbf{seqBFSVisit}(\mathit{graph}, x, q, \mathit{visited}) \to \textbf{seqBFSScanAdj}(\mathit{graph}, x, q, \textbf{setl}(x, \texttt{1}, \mathit{visited}), \textbf{getAdj}(x, \mathit{graph})) & (4)\\
\textbf{seqBFSScanAdj}(\mathit{graph}, x, q, \mathit{visited}, \texttt{nil}) \to \textbf{seqBFSQueue}(\mathit{graph}, q, \mathit{visited}) & (5)\\
\textbf{seqBFSScanAdj}(\mathit{graph}, x, q, \mathit{visited}, \textbf{cons}(u, l)) \to & \\
\quad \textbf{seqBFSCheckNeighbor}(\mathit{graph}, x, q, \mathit{visited}, \textbf{cons}(u, l), \textbf{getl}(u, \mathit{visited})) & (6)\\
\textbf{seqBFSCheckNeighbor}(\mathit{graph}, x, q, \mathit{visited}, \textbf{cons}(u, l), \mathit{isVisited}) \to & \\
\quad \textbf{seqBFSScanAdj}(\mathit{graph}, x, \textbf{pushBack}(u, q), \mathit{visited}, l)\ [\mathit{isVisited} = \texttt{0}] & (7)\\
\textbf{seqBFSCheckNeighbor}(\mathit{graph}, x, q, \mathit{visited}, \textbf{cons}(u, l), \mathit{isVisited}) \to & \\
\quad \textbf{seqBFSScanAdj}(\mathit{graph}, x, q, \mathit{visited}, l)\ [\mathit{isVisited} = \texttt{1}] & (8)
\end{cases}$$

The following set of rewriting rules defines the operation of adding the element to the end of the queue:

$$\mathcal{R}_{\text{pushBack}} = \begin{cases}
\textbf{pushBack}(\mathit{element}, \textbf{cons}(u, l)) \to \textbf{cons}(u, \textbf{pushBack}(\mathit{element}, l)) & (1)\\
\textbf{pushBack}(\mathit{element}, \texttt{nil}) \to \textbf{cons}(\mathit{element}, \texttt{nil}) & (2)
\end{cases}$$

The following set of rewriting rules defines the operation of accessing the n-th element in the linked list:

$$\mathcal{R}_{\text{getl}} = \begin{cases}
\textbf{getl}(n, \textbf{cons}(m, l)) \to m\ [n = \texttt{0}] & (1)\\
\textbf{getl}(n, \textbf{cons}(m, l)) \to \textbf{getl}(n - \texttt{1}, l)\ [n > \texttt{0}] & (2)
\end{cases}$$

Finally, the following set of rewrite rules defines the operation of setting the n-th element of the linked list to a new value.

$$\mathcal{R}_{\text{setl}} = \begin{cases}
\textbf{setl}(n, v, \textbf{cons}(m, l)) \to \textbf{cons}(v, l)\ [n = \texttt{0}] & (1)\\
\textbf{setl}(n, v, \textbf{cons}(m, l)) \to \textbf{cons}(m, \textbf{setl}(n - \texttt{1}, v, l))\ [n > \texttt{0}] & (2)
\end{cases}$$

7.2.2 Case Study: Parallel-Rewrite Complexity of Two BFS Implementations

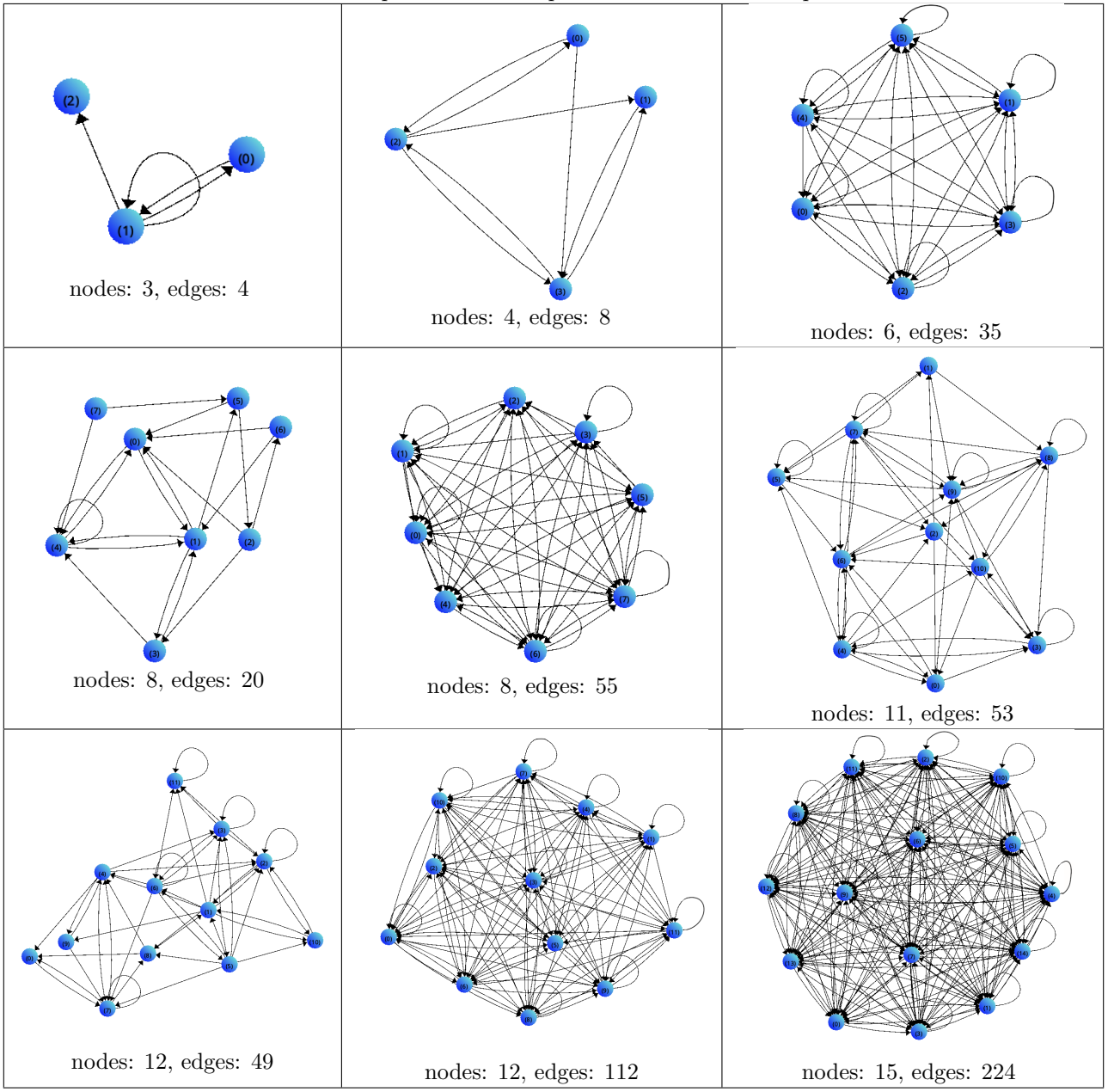
We compared two BFS encodings in the rewriting engine: a parallel MemTRS implementation with global memory and a sequential list-based implementation without memory. Since the previous QuickSort experiment 7.1.3 showed that engine-level performance is strongly influenced by the size and shape of rewritten terms, we use the total number of parallel reduction steps as our main benchmarking metric. This metric does not directly measure the execution time, but it gives a stable view of how much rewriting work is performed by each encoding.

We ran both BFS algorithms on randomly generated directed graphs, with numbers of nodes ranging from 1 to 15, always starting the search from the node `0`. For each graph size n, we considered all possible numbers of edges from 1 up to $n^2 - 1$. For every combination of node count and edge count, we repeated the experiment three times and averaged the resulting numbers of parallel reduction steps.

It is important to note that this comparison does not isolate a single factor: the two encodings differ in their use of global memory, in their queue representation, and in whether they process nodes sequentially or in parallel. Our goal is not to evaluate one isolated optimization in abstraction, but to compare the strongest practical BFS encodings we obtained in the two settings.

Here are some example graphs generated during the experiments:

Table 7.5: Example Random Graphs Generated for the Experiment



The following graph shows how the number of nodes and edges in the graph influences the total number of parallel rewriting steps while running the BFS algorithms from the node 0.

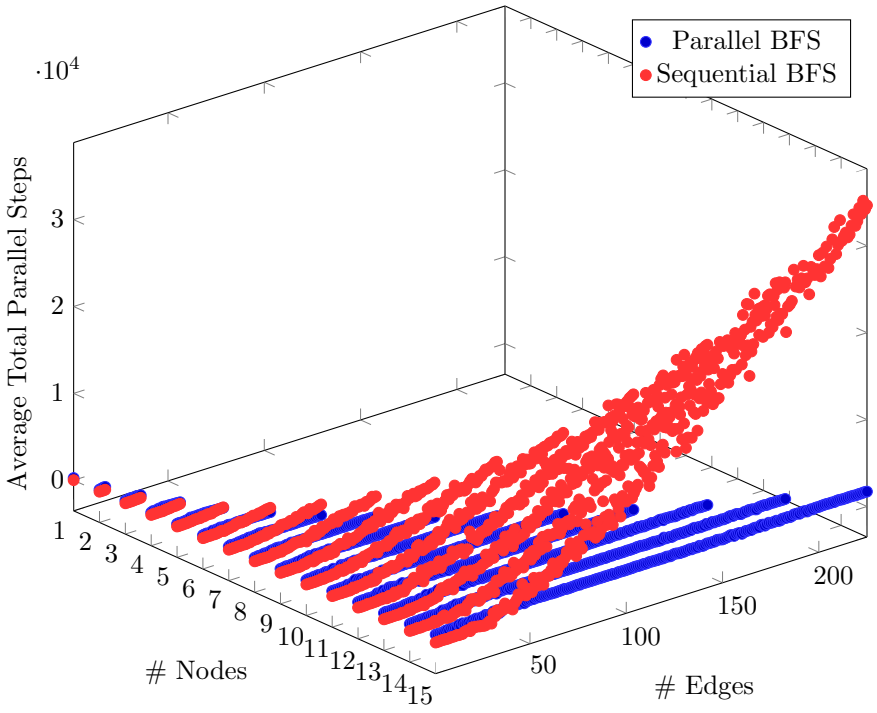


Figure 7.5: BFS implementation parallel rewrite steps comparison (3d graph)

We can clearly see that the number of edges in the graph influences the total number of parallel rewrite steps for both parallel and sequential BFS. However, we cannot see a significant influence of the number of nodes on the number of parallel rewrite steps. It can be explained by the fact that we only consider graphs with up to 15 nodes. For a larger number of nodes and the same number of edges, we will see a difference: we will have to generate/allocate a larger list/array of visited nodes.

The following graph is the projection of the original dataset onto the two-dimensional plane obtained by omitting the number of nodes dimension.

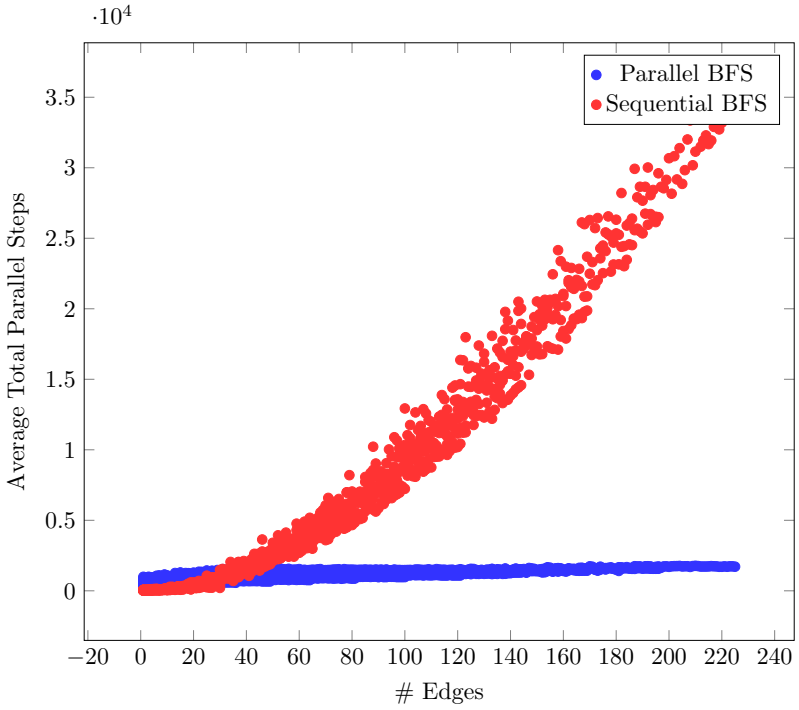


Figure 7.6: BFS implementation parallel rewrite steps comparison (number of edges only)

We can clearly see that the parallel version of the algorithm requires far fewer parallel reduction steps than the serial algorithm. However, it is important to note that for graphs with small number of edges, the parallel BFS shows slightly worse performance due to additional overhead from scheduling parallel **bfsWorker**s.

Also, we can see that, for the serial algorithm, the number of parallel reduction steps grows faster as the graph becomes denser. This can be explained by the fact that, in the serial algorithm, we use a linked list to represent the array of visited nodes. Each operation on this linked list (to read or update a value at the given position), in the worst-case scenario, has complexity of $\mathcal{O}(n)$. In addition, the list-queue operation **pushBack** is also linear in the list-queue length. We execute this operation for every encountered edge - to check whether the node it's pointing to was already visited, and for every encountered node in the graph - to mark the node as visited. That means, the original complexity of the BFS algorithm in best-case scenario changes from $\mathcal{O}(V+E)$ to $\mathcal{O}(V^2 + E \cdot V)$ (where V is the number of nodes in the graph, and E is the number of edges).

On the contrary, the parallel BFS implementation shows a linear correspondence between the number of edges and the total number of parallel rewrite steps. This is a significant improvement compared to the sequential implementation. Every new node at one layer is processed in parallel (in graphs with $V \leq 10$), and for each node, we have to explore every outgoing edge. There is one bottleneck: since we still use a linked list to store the graphs, the graph access operation **getAdj** has complexity $\mathcal{O}(V)$. Another limitation of this algorithm is that we use a fixed number of 10 parallel **bfsWorker**s. Although it is not a problem for a few nodes, with a higher number of vertices in the graph, it can worsen the performance of one-layer processing.

So, in the idealized setting, the complexity is $\mathcal{O}(D \cdot (V + \Delta))$, where D is the graph diameter (the farthest distance between any two vertices in the graph), and Δ denotes the largest degree of a node in the graph. Since $\Delta \leq V$, this bound can be simplified to $\mathcal{O}(D \cdot V)$. If we additionally assume that the graph diameter is bounded by a constant, as in complete graphs or sufficiently well-connected dense graphs, the bound becomes $\mathcal{O}(V)$.

We also note that the parallel variant uses mark-on-discovery through the atomic **CAS** operation, whereas the sequential list-based variant marks a node then it is processes from the queue. This difference may also influence the measured number of rewrite steps.

Conclusion

We can say that the parallel BFS algorithm with global memory provides a significant performance boost in the number of parallel rewrite steps. The explicit level-synchronization and usage of the concurrent shared queue data structure guarantees the correct traversal of the graph. A slight modification of this algorithm can allow us not only to discover accessible nodes, but also to compute the distances from the starting node to any other node in the graph. It's important to mention that the shared memory in this case enables parallel execution in general. We were unable to find a sustainable way to implement information sharing about visited nodes between parallel BFS instances without using global memory.

For some readers it might seem unfair that we were comparing single-threaded term-based BFS and parallel memory-based BFS. Indeed, there exists an intermediate step: the single-threaded memory-based BFS algorithm. However, this additional comparison does not fully align with the objectives of our research project. In this case study, we examine two types of term-rewriting systems as complete computation environments and aim to exploit all available features of each system to achieve the best possible BFS performance within that system.

7.2.3 Floyd-Warshall Algorithm

The Floyd-Warshall algorithm [10] is an algorithm for finding the shortest paths in a directed weighted graph with positive or negative edge weights.

First, we will introduce some definitions regarding the paths in the weighted graph.

Definition 7.2.1 (Path in a graph. [11, Defined on p. 1165–1166])**.** A path of length k from vertex u to vertex u' in a graph $G = (V, E)$ is a sequence $\langle v_0, v_1, \ldots, v_k \rangle$ of vertices such that $u = v_0$, $u' = v_k$, and $(v_{i-1}, v_i) \in E$ for every $i = 1, \ldots, k$. There is always a path with no edges from a vertex to itself. A path is simple if all vertices in the path are distinct.

Definition 7.2.2 (Weight of a path in the graph. [12, Defined on p. 605])**.** The weight $w(p)$ of a path $p = \langle v_0, v_1, \ldots, v_k \rangle$ is the sum of weights of its constituent edges:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

Definition 7.2.3 (Shortest path between two vertices in the weighted graph. [12, Defined on p. 605])**.** The shortest path from vertex u to v is defined as any path with the weight $w(p) = \delta(u, v)$. Where the shortest-path weight $\delta(u, v)$ from u to v is defined as:

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \to^p v\} & \text{if there is a path from } u \text{ to } v, \\ \infty & \text{otherwise} \end{cases}$$

Following Cormen et al. [12, p. 606–608], we assume that the input graph contains no negative-weight cycles. Under this assumption, shortest-path weights are well-defined, and shortest paths can be assumed to be simple.

Let's consider the arbitrary directed weighted graph and enumerate the graph vertices as follows: $v_1, v_2, \ldots, v_n$. For $k = 0, 1, \ldots, n$ we can define $d_{x,y}^{(k)}$ as the length of the shortest path from v_x to v_y that uses only vertices from the set $\{v_1, \ldots, v_k\}$ as intermediate vertices (all vertices of the path other than v_x and v_y).

At each step, for the given pair of vertices v_x and v_y, the shortest path p between them using intermediate vertices from the set $\{v_1, \ldots, v_k\}$, either does not include v_k or includes v_k:

- If v_k is not used in that path p, then the shortest path between v_x and v_y is the same as the shortest path that uses intermediate vertices from the set $\{v_1, \ldots, v_{k-1}\}$.
- If v_k is used, then the path can be split into two subpaths: $v_x \to^{p_1} v_k$ and $v_k \to^{p_2} v_y$. Both subpaths p_1 and p_2 only use intermediate vertices from the set $\{v_1, \ldots, v_{k-1}\}$.

This observation leads to the following recursive definition for shortest-path estimates:

$$d_{x,y}^{(k)} = \begin{cases} w_{x,y} & \text{if } k = 0, \\ \min\{d_{x,y}^{(k-1)}, d_{x,k}^{(k-1)} + d_{k,y}^{(k-1)}\} & \text{if } k \geq 1, \end{cases}$$

where $w_{x,y}$ is the corresponding entry of the initial weight matrix:

$$w_{x,y} = \begin{cases} 0 & \text{if } x = y, \\ w(v_x, v_y) & \text{if } x \neq y \text{ and there is an edge from } v_x \text{ to } v_y, \\ \infty & \text{otherwise.} \end{cases}$$

Finally, the matrix $D^{(n)} = (d_{x,y}^{(n)})$ gives the shortest distances between all pairs of vertices: $d_{x,y}^{(n)} = \delta(x, y)$.

For this case study, we use the unweighted directed graph structure from Section 5.3.7. We interpret such a graph as a weighted one by assigning weight 1 to every existing edge. This way, the implementation computes all-pairs shortest paths in unweighted directed graphs, expressed as a special case of the Floyd-Warshall algorithm.

The update rules operate on the initialized distance matrix and have the same form for arbitrary edge weights. Supporting weighted input graphs would require extending the graph representation and the **graphToMatrix** conversion procedure, so that edge weights are stored in the initial matrix.

Sequential Version

The sequential version of the algorithm is based on the recurrent relation described above. The rule (1) defines the preparation stage of the algorithm, in which we convert the input graph into a distance matrix in memory. We fill this matrix with the default value $V \cdot 100$, which serves as a finite representation of the "infinite" distance

between two nodes in the graph with no edges in between. The rule (2) defines the second step of the algorithm, in which we set the main diagonal of the matrix to 0; this shows that the distance between a node and itself is 0. Finally, the rules (3) – (10) define the three nested loops bottom-up procedure that computes the distance matrix using the recurrent relation for distances between vertices.

We intentionally decided to use memory for the sequential algorithm version. If we were to use the term data structures to represent matrices, all matrix access and update operations would degrade to linear operations, which would obviously make the algorithm's complexity significantly worse than in the case of the memory implementation. So, assuming the complexity of the sequential version is $\mathcal{O}(V^3)$, the complexity of the same algorithm using only terms would be $\mathcal{O}(V^4)$ (V is a number of nodes in the graph). By using memory in the sequential version, we can focus on how parallel workers can improve the algorithm's performance.

$\mathcal{R}_{\text{floydWarshall}} =$

$$
\begin{cases}
\textbf{floydWarshall}(g) \to \textbf{floydWarshallInit}(\textbf{graphToMatrix}(g, \textbf{numNodes}(g) * \texttt{100})) & (1)\\
\textbf{floydWarshallInit}(addr) \to & \\
\quad \textbf{floydWarshallZeroDiag}(addr, \textbf{getMatrixWidth}(addr), \textbf{fillMatrixDiag}(addr, \texttt{0}))\ [addr \geq \texttt{0}] & (2)\\
\textbf{floydWarshallInit}(addr) \to \texttt{-1}\ [addr < \texttt{0}] & (3)\\
\textbf{floydWarshallZeroDiag}(addr, size, \texttt{true}) \to \textbf{floydWarshallLoopK}(addr, size, \texttt{0}, \texttt{0}, \texttt{0}) & (4)\\
\textbf{floydWarshallLoopK}(addr, size, i, j, k) \to \textbf{floydWarshallLoopI}(addr, size, i, j, k)\ [k < size] & (5)\\
\textbf{floydWarshallLoopI}(addr, size, i, j, k) \to \textbf{floydWarshallLoopJ}(addr, size, i, j, k)\ [i < size] & (6)\\
\textbf{floydWarshallLoopJ}(addr, size, i, j, k) \to & \\
\quad \textbf{floydWarshallStore}(addr, size, i, j, k, & \\
\quad\quad \textbf{setMatrix}(addr, i, j, & \\
\quad\quad\quad \textbf{min}(\textbf{getMatrix}(addr, i, j), \textbf{getMatrix}(addr, i, k) + \textbf{getMatrix}(addr, k, j))))\ [j < size] & (7)\\
\textbf{floydWarshallStore}(addr, size, i, j, k, \texttt{true}) \to \textbf{floydWarshallLoopJ}(addr, size, i, j + \texttt{1}, k) & (8)\\
\textbf{floydWarshallLoopJ}(addr, size, i, j, k) \to \textbf{floydWarshallLoopI}(addr, size, i + \texttt{1}, \texttt{0}, k)\ [j \geq size] & (9)\\
\textbf{floydWarshallLoopI}(addr, size, i, j, k) \to \textbf{floydWarshallLoopK}(addr, size, \texttt{0}, j, k + \texttt{1})\ [i \geq size] & (10)\\
\textbf{floydWarshallLoopK}(addr, size, i, j, k) \to addr\ [k \geq size] & (11)
\end{cases}
$$

Parallel Version

The parallel version of the algorithm is similar in nature to the serial one. The difference here is that we use the 2-D block-mapping approach when processing each matrix. Our implementation of the parallel Floyd-Warshall algorithm is based on the Checkerboard Version of the Parallel Floyd Algorithm presented by Kumar and Singh [23]. In the current implementation, we split the matrix into four blocks and perform calculations in parallel across them.

Unlike the sequential version, the parallel version uses two matrices. For each value of k, all workers should compute parts of $D^{(k)}$ from the same previous matrix $D^{(k-1)}$. Because of this, we use *curr* as a reference matrix for current iteration, and write the computation results to the matrix *next*. After all block workers finish, rule (14) swaps two matrices and proceeds with the next value of k. In other words, the double-buffering avoids read/write interference between workers and makes parallel update correspond to the recursive definition above.

$\mathcal{R}_{\text{parFloydWarshall}} =$

$$
\begin{cases}
\textbf{parFloydWarshall}(g) \to \textbf{parFloydWarshallAlloc}(g, \textbf{numNodes}(g)) & (1)\\
\textbf{parFloydWarshallAlloc}(g, size) \to & \\
\quad \textbf{parFloydWarshallInitDiag}(size, \textbf{graphToMatrix}(g, size * \texttt{100}), & \\
\quad\quad \textbf{graphToMatrix}(g, size * \texttt{100}))\ [size \geq \texttt{0}] & (2)\\
\textbf{parFloydWarshallAlloc}(g, size) \to \texttt{-1}\ [size < \texttt{0}] & (3)\\
\textbf{parFloydWarshallInitDiag}(size, curr, next) \to & \\
\quad \textbf{parFloydWarshallStart}(size, curr, next, & \\
\quad\quad \textbf{fillMatrixDiag}(curr, \texttt{0}), \textbf{fillMatrixDiag}(next, \texttt{0}))\ [curr \geq \texttt{0} \wedge next \geq \texttt{0}] & (4)\\
\textbf{parFloydWarshallInitDiag}(size, curr, next) \to \texttt{-1}\ [curr < \texttt{0} \vee next < \texttt{0}] & (5)\\
\textbf{parFloydWarshallStart}(size, curr, next, \texttt{true}, \texttt{true}) \to & \\
\quad \textbf{parFloydWarshallLoopK}(size, curr, next, \texttt{0}) & (6)
\end{cases}
$$

$$
\begin{cases}
\textbf{parFloydWarshallLoopK}(size, curr, next, k) \rightarrow \\
\quad \textbf{parFloydWarshallSync}(size, curr, next, k, \\
\qquad \textbf{fwBlockWorker}(size, curr, next, k, size \ / \ 2, size \ / \ 2, 0, 0), \\
\qquad \textbf{fwBlockWorker}(size, curr, next, k, size, size \ / \ 2, size \ / \ 2, 0), \\
\qquad \textbf{fwBlockWorker}(size, curr, next, k, size \ / \ 2, size, 0, size \ / \ 2), \\
\qquad \textbf{fwBlockWorker}(size, curr, next, k, size, size, size \ / \ 2, size \ / \ 2)) \ [k < size] & (7) \\
\textbf{parFloydWarshallLoopK}(size, curr, next, k) \rightarrow curr \ [k \geq size] & (8) \\
\textbf{fwBlockWorker}(size, curr, next, k, rowEnd, colEnd, row, col) \rightarrow \\
\quad \textbf{fwBlockContinue}(size, curr, next, k, rowEnd, colEnd, row + 1, col, \\
\qquad \textbf{fwUpdateCell}(curr, next, row, col, k)) \ [row < rowEnd \wedge col < colEnd] & (9) \\
\textbf{fwBlockWorker}(size, curr, next, k, rowEnd, colEnd, row, col) \rightarrow \\
\quad \textbf{fwBlockWorker}(size, curr, next, k, rowEnd, colEnd, 0, col + 1) \ [row \geq rowEnd \wedge col < colEnd] & (10) \\
\textbf{fwBlockWorker}(size, curr, next, k, rowEnd, colEnd, row, col) \rightarrow 1 \ [col \geq colEnd] & (11) \\
\textbf{fwBlockContinue}(size, curr, next, k, rowEnd, colEnd, row, col, \texttt{true}) \rightarrow \\
\quad \textbf{fwBlockWorker}(size, curr, next, k, rowEnd, colEnd, row, col) & (12) \\
\textbf{fwUpdateCell}(curr, next, row, col, k) \rightarrow \\
\quad \textbf{setMatrix}(next, row, col, \\
\qquad \textbf{min}(\textbf{getMatrix}(curr, row, col), \textbf{getMatrix}(curr, row, k) + \textbf{getMatrix}(curr, k, col))) & (13) \\
\textbf{parFloydWarshallSync}(size, curr, next, k, 1, 1, 1, 1) \rightarrow \\
\quad \textbf{parFloydWarshallLoopK}(size, next, curr, k + 1) & (14)
\end{cases}
$$

7.2.4 Case Study: Comparing Parallel Rewrite Steps of Two Floyd-Warshall Implementations

In this experiment, we use the same setting as in the BFS implementations comparison. We ran both Floyd-Warshall algorithms on the random directed graphs with up to 15 nodes. For each of the graphs, we considered all possible numbers of edges.

The following graph shows how the number of nodes and edges in the graph influences the total number of parallel rewriting steps:

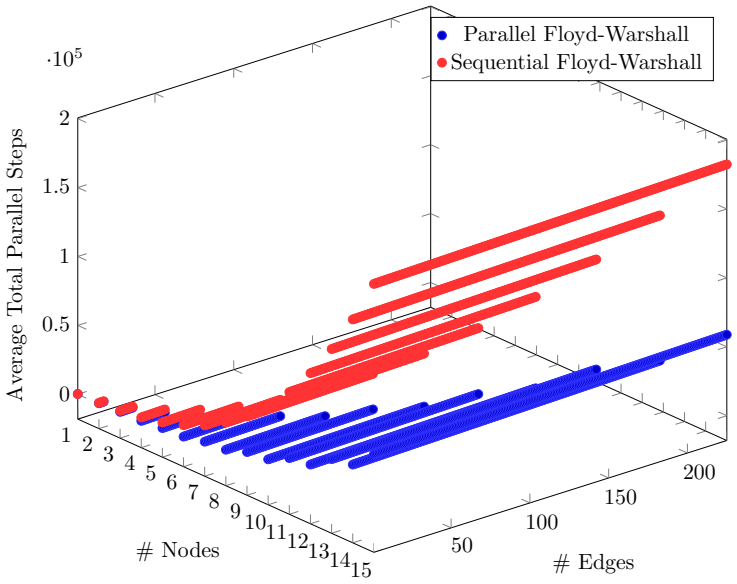


Figure 7.7: Floyd-Warshall algorithm implementation parallel rewrite steps comparison (3d graph)

We observe that the number of nodes has a much stronger influence on the total number of parallel rewrite steps than the number of edges. The effect of the number of edges appears to be comparatively small and roughly linear, which is consistent with the initial conversion of the input graph into a matrix.

The following graph is a projection of the original dataset onto the two-dimensional plane, obtained by omitting the edge dimension.

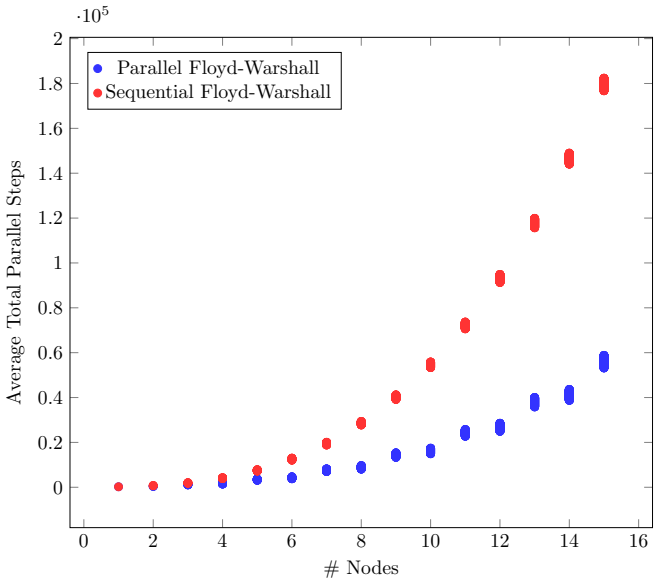


Figure 7.8: Floyd-Warshall algorithm implementation parallel rewrite steps comparison (number of nodes only)

This graph shows that the parallel version of the algorithm requires noticeably fewer parallel reduction steps than the sequential version. This is consistent with the parallel algorithm described in [23]. While the complexity of the sequential implementation is $\mathcal{O}(V^3)$, according to the paper, the complexity of the parallel implementation should be $\mathcal{O}(\frac{V^3}{p})$ (where p is the number of parallel processors). This is also consistent with our implementation: we process distance matrices V times, and 4 workers process the $\frac{V}{2} \times \frac{V}{2}$ elements of each matrix in parallel. This gives us the total complexity of $\mathcal{O}(V \times \frac{V}{2} \times \frac{V}{2}) = \mathcal{O}(V \times \frac{V^2}{4}) = \mathcal{O}(\frac{V^3}{4})$ (and we use $p = 4$ "parallel processors").

Conclusion

We can say that the parallel Floyd-Warshall algorithm shows a noticeable performance boost compared to the sequential version. The usage of shared memory allows for constant matrix access and update operations, which, in combination with parallel rewriting allows recreating existing parallel algorithms. What's remarkable is that the parallel complexity of the implemented algorithm matches the theoretical estimates in the literature.

Chapter 8

Summary

In this research project, we introduced Memory Term Rewriting Systems (MemTRS) as an extension of Logically Constrained Term Rewriting Systems designed to support shared-memory concurrency. The primary objective of this project was to narrow the gap between classical term-rewriting frameworks and the requirements of modern parallel computation, where multiple execution threads can interact through mutable shared state.

We formally defined the syntax and semantics of MemTRS, including the atomic read, write, and CAS operations and parallel reduction relations. By extending the rewriting relation to operate over pairs of terms and memory states, MemTRS captures both functional reductions and memory side effects within a single formal model. The introduction of parallel reduction and strategies further enables reasoning about concurrent execution and nondeterminism. One of the key contributions of this work is the analysis of nondeterministic behaviors arising from parallel rewriting with shared memory, such as data races caused by concurrent reads and writes. We discussed several techniques to mitigate these issues, including special rule designing, and restriction of parallel positions. These mechanisms allow MemTRS to model common concurrency control patterns such as locks and mutexes.

To demonstrate the practical expressiveness of MemTRS, we developed term-rewriting representations for fundamental data structures, including arrays, matrices, and concurrent queues. To simplify working with these data structures, we even developed a simple memory allocation scheme. Building on these abstractions, we modelled and analyzed several algorithms, including array sorting algorithms and graph algorithms such as Breadth-First Search and Floyd-Warshall. Finally, we presented an implementation of MemTRS within the *Cora* term rewriting engine, showing that the proposed framework is not only well-defined on a theoretical level, but also practically implementable. The experimental results indicate that MemTRS can effectively model realistic concurrent algorithms while preserving clear operational semantics. The case study involving the QuickSort algorithms also revealed some limitations of the underlying rewriting engine, particularly a significant performance drop during term reduction with a large number of potential redexes. These problems were investigated by running experiments with the Java Flight Recorder profiling tool and by studying the source code of the rewriting engine.

Overall, this work demonstrates that Memory Term Rewriting Systems provide a promising foundation for modelling and reasoning about shared-memory parallel computation. By extending term rewriting with concurrent shared memory operations, MemTRS opens new possibilities for the formal analysis, verification, and experimentation with concurrent data structures and algorithms.

Chapter 9

Reflection

In this final chapter, I will reflect on the results of the internship project. I will consider the weaknesses of the presented approach, look into the strong parts of the research project, emphasize the topics I learned throughout working on this project, and discuss the improvements and future work that should be done to better answer the research question.

9.1 Personal Learning Outcomes and Achievements

While working on this project, I learned a lot of skills and concepts related to both practical software engineering and theoretical research. On the practical side, I gained experience with the internal construction of a *Cora* term rewriting engine, Java parallel programming, profiling tools such as Java Flight Recorder, and the implementation of thread-safe shared-memory structures. In particular, the implementation of the parallel rewriting relation of MemTRS made me better understand how reductions are selected, how candidate redexes are computed, and how parallel reductions can be executed safely in the presence of shared mutable state.

Beyond these implementation skills, I also learned how to express imperative multithreaded algorithms in a functional term-rewriting style. Algorithms that are straightforward in an imperative language often require a careful encoding of control flow, memory allocation, synchronization, and evaluation order when written as rewrite rules. This is especially visible in the encoding of arrays, matrices, concurrent queues, sorting algorithms, and graph algorithms in the simplified memory model used in this project. As a result, I gained a better understanding of the gap between the algorithm that uses shared memory and a rewriting system that models this algorithm.

From the theoretical perspective, I learned how to extend the Logically Constrained Term Rewriting Systems with a new shared memory component and selected primitive memory operations. Defining MemTRS required more than adding **GET**, **SET**, **CAS** as extra symbols: it required redefining the rewrite relation over the configurations consisting of a term and a memory state, and defining what it means to perform reductions in parallel. This showed me that adding shared memory to rewriting is not a simple implementation extension. It affects the semantics of rule application, because reductions may now be influenced by the memory state. It also raises additional questions about nondeterminism and the influence of different rewriting strategies, since the order in which memory operations are performed can change the resulting memory configuration.

I also briefly explored the encoding of termination problems into SAT/SMT, in particular using Z3. Although, this didn't influence the final project, it made me understand that termination analysis for MemTRS is not a straightforward adaptation of existing LCTRS techniques. This point is discussed further in subsection 9.3.4.

9.2 Lessons Learned from Mistakes

During this project I encountered several mistakes and false starts, which were important for understanding the problem better. One of the most significant theoretical mistakes was the initial attempt to model memory operations as LCTRS theory terms. At first, this seemed natural, because operations such as **GET**, **SET**, and **CAS** have fixed intended meanings. However, treating them as theory terms meant that, by design of the LCTRS framework, they could also occur inside rule constraints. This made the semantics difficult to formalize, because checking whether a rule is applicable could itself depend on reading memory or even modifying it. This showed me that memory operations should not be treated as ordinary pure theory function symbols. As a result, I moved towards a design where memory operations are handled explicitly as memory reduction steps, and where synchronization can be implemented using the atomic **CAS** operation.

Another important mistake was in the implementation of the parallel reduction. In earlier version of the rewriting engine, one parallel step could rewrite redexes that were not truly disjoint. This meant that the implementation was closer to the modeling of the *development* operation over a term (as for instance studied in [25]). Fixing this required ensuring that during one parallel step only redexes at disjoint positions are rewritten. This taught me that that parallel reduction in the implementation must follow the formal definition carefully; it's not enough to simply perform several reductions at once.

One of the false starts in the project was related to the memory model. In the first version, the system had a very limited one-cell memory representation, which made it difficult to encode realistic algorithms. This led to more general memory-function/array model which I ended up using in the project. I also had an unsuccessful attempt of integrating MemTRS memory into the *Cora*'s SMT-based termination analysis. The idea was to represent the global memory through an array-like structure with a dedicated SMT encoding. However, this turned out to be difficult: SMT-LIB arrays are represented through the **select** and **store** functions, which made it hard to recover concrete memory contents from solver models. Encoding each memory cell separately would avoid this problem, but would quickly introduce too many variables and constraints. This made me realize that termination analysis for MemTRS needs a more dedicated treatment of memory, rather than a direct extension of the existing SMT encoding.

The memory implementation also went through several revisions. I initially tried to use ordinary non-volatile structures, but they were not safe for parallel access and led to difficult concurrency bugs. After that I used a *ConcurrentMap* with a mechanism that generated memory values on demand, aiming to simulate raw memory more realistically and avoid allocating large data structures upfront. This new approach was thread safe, but it introduced too much overhead and affected the performance experiments. I therefore moved to a simpler fixed-size memory model based on *AtomicIntegerArray*, which is less flexible but safer, more predictable, and more efficient at runtime.

Finally, some choices were not incorrect, but turned out to be suboptimal. For example, the initial implementation used native Java threads with a fixed thread pool. Later, I replaced this by *virtual threads*, which led to a noticeable speedup in the experiments. This experience taught me that even when the formal MemTRS model abstracts away from scheduling and execution details, the implementation of the rewriting engine still strongly influences the measured performance. Therefore, the benchmarks in this project should be interpreted as results for this particular *Cora*-based implementation of MemTRS, not as direct measurements of abstract algorithms alone.

9.3 Hard Questions about the Current Approach

9.3.1 Cora as an Implementation Platform

A major limitation of this project is that *Cora* turned out not to be a suitable execution engine for efficient rewriting with memory. Although, it was very useful as a prototyping environment, most of the runtime cost in the experiments does not seem to come from the intended algorithmic work itself, but from administrative work inside the rewriting engine, such as finding candidate redexes, filtering subterms, constructing reducts, and managing immutable term objects.

This is especially problematic for recursive data structures such as lists, where many rewrite steps operate on large terms. In such cases, the cost of rewriting in *Cora* can dominate the actual cost of the algorithm. Therefore, the experiments should not be interpreted as evidence that MemTRS algorithms are particularly efficient in their current implementation. More precisely, the experiments show that *Cora* can be used to prototype the semantics, but probably not as the basis for an efficient shared-memory rewriting engine.

A more efficient implementation would likely require a notably different representation of terms using mutable data structures, rather than only small optimizations to the existing *Cora* engine. For example, improving the subterm filtering procedure may help, but it probably does not address the deeper problem that the engine was not designed for this style of execution-heavy parallel rewriting.

9.3.2 The Benefit of Term Rewriting

Another important question is whether term rewriting adds a real benefit for the algorithms studied in this project. Memory clearly adds expressive power to LCTRSs: it makes it possible to directly model arrays, matrices, queues in one shared mutable state. However, many of the resulting translations of the algorithms look very similar to imperative C-style implementations, only encoded with rewrite rules.

This raises a fair question: by adding memory, the encodings may lose some natural advantages of term rewriting. In ordinary term rewriting, algorithms can often be written using recursive structure and pattern matching. The pure LCTRS-based implementation of the QuickSort algorithm on lists is a good example of this style. In contrast, the memory-based algorithms often become more iterative with explicit loops, counters, array indices and special worker-"processes".

This suggests that added shared memory naturally encourages a more imperative programming style inside the rewriting framework. This is useful for representing shared-memory algorithms, because many of them rely on explicit array positions, loop counters, and memory updates. However, it also weakens some usual advantages of term rewriting: computations become less structurally recursive, and in some cases parallelism has to be introduced more explicitly. Therefore, an important question is whether MemTRS can combine shared memory with the declarative strengths of rewriting, or whether it mainly becomes a rewriting-based encoding of imperative programs.

9.3.3 Usability Concerns

During this project we mostly focused on whether the term rewriting with shared memory is possible, and not on whether it is pleasant to use. The definite answer I got after working on this project is that parallel rewriting with memory is possible, but the user has to think about many low-level details.

When expressing algorithms in MemTRS, the user must explicitly consider memory layout, synchronization, effects of the chosen reduction strategy, possible race conditions between **GET** and **SET**, and the use of **CAS** for locks. In several places, correctness may depend on carefully forcing arguments to be evaluated. This makes the system fragile: a careless change in rewrite rules can introduce an undesirable nondeterministic behavior.

A useful direction would be to design high-level abstractions on top of MemTRS. In the current approach, the user still has to reason manually about the evaluation order, synchronization and possible interactions between parallel reductions. Abstractions for common patterns such as parallel loops, worker scheduling, and lock-based updates, could make it easier to express algorithms in MemTRS without relying on manual scheduling decisions.

9.3.4 Termination Analysis Challenges

Termination analysis for MemTRS is likely much harder than simply adapting existing techniques for LCTRS. Initially, it seemed that methods such as dependency pairs could be extended to the new setting. However, the presence of memory and consequent synchronization burden significantly changes the problem.

For example, many algorithms use locks implemented with the **CAS** operation. A process may repeatedly fail to acquire a lock and stay in the busy waiting loop. So, the termination of the whole system may depend on whether another "parallel process" eventually releases the lock. Therefore, termination is no longer a purely syntactic property of rewrite rules. It also depends on memory states and parallel reduction scheduling.

This means that before developing a termination method for MemTRS, we first have to define what termination should mean in this setting. Should every possible reduction sequence terminate? Should unfair schedules be ignored? Should a system be considered terminating if it terminates under fair scheduling? These questions are essential, because standard termination notions may be too strict or may classify reasonable concurrent algorithms as non-terminating.

For this reason we should not consider termination analysis for MemTRS as an easy extension of LCTRS termination. It would require dedicated techniques that account for memory reachability, lock acquisition and release, and order of parallel execution.

9.4 Conclusion

Overall, this project showed me that MemTRS is a useful formal experiment, but not yet a practical or comfortable framework for effortless expression of concurrent algorithms. The work demonstrates that shared memory can be integrated into term rewriting, however it also shows that doing so introduces noticeable semantic and practical complications. Memory operations affect rule applicability, reduction order, and parallelism. They introduce extra layer of non-determinism, influence implementation efficiency, and raise extra questions about termination. In that sense, the most important outcome of this project is not only the proposed formalism and concluded experiments, but also the identification of the problems that arise when one tries to combine term rewriting with a shared mutable state.

Bibliography

[1] Iliès Alouini and Claude Kirchner. *Conditional Concurrent Rewriting*. Research Report RR-2777. INRIA, 1996, p. 24. URL: https://inria.hal.science/inria-00073915.

[2] Arvind and X. Shen. "Using term rewriting systems to design and verify processors". In: *IEEE Micro* 19.3 (1999), pp. 36–46. DOI: 10.1109/40.768501.

[3] Martin Aumüller and Nikolaj Hass. "Simple and Fast BlockQuicksort using Lomuto's Partitioning Scheme". In: *2019 Proceedings of the Meeting on Algorithm Engineering and Experiments (ALENEX)*, pp. 15–26. DOI: 10.1137/1.9781611975499.2. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611975499.2. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611975499.2.

[4] Jaroslav Bachorík, Johannes Bechberger, and Ron Pressler. *JEP 509: JFR CPU-Time Profiling (Experimental)*. 2024. URL: https://openjdk.org/jeps/509 (visited on 12/09/2025).

[5] Thaïs Baudon, Carsten Fuhs, and Laure Gonnord. "Analysing Parallel Complexity of Term Rewriting". In: *Logic-Based Program Synthesis and Transformation*. Ed. by Alicia Villanueva. Cham: Springer International Publishing, 2022, pp. 3–23. ISBN: 978-3-031-16767-6.

[6] Johannes Bechberger. *JFR Query Experiments*. 2025. URL: https://github.com/parttimenerd/jfr-query-experiments (visited on 12/09/2025).

[7] Aydin Buluç and Kamesh Madduri. "Parallel breadth-first search on distributed memory systems". In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '11. Seattle, Washington: Association for Computing Machinery, 2011. ISBN: 9781450307710. DOI: 10.1145/2063384.2063471. URL: https://doi.org/10.1145/2063384.2063471.

[8] Pao-I Chen and Tsung-Wei Huang. "An Efficient Implementation of Parallel Breadth-first Search". In: *Proceedings of the 1st FastCode Programming Challenge*. FCPC '25. The Westin Las Vegas Hotel & Spa, Las Vegas, NV, USA: Association for Computing Machinery, 2025, pp. 8–13. ISBN: 9798400714467. DOI: 10.1145/3711708.3723443. URL: https://doi.org/10.1145/3711708.3723443.

[9] T.H. Cormen et al. "Introduction to Algorithms, fourth edition". In: MIT Press, 2022. Chap. Chapter 20. Elementary Graph Algorithms, pp. 554–562. ISBN: 9780262046305. URL: https://books.google.nl/books?id=HOJyzgEACAAJ.

[10] T.H. Cormen et al. "Introduction to Algorithms, fourth edition". In: MIT Press, 2022. Chap. Chapter 23. All-Pairs Shortest Paths. Pp. 655–661. ISBN: 9780262046305. URL: https://books.google.nl/books?id=HOJyzgEACAAJ.

[11] T.H. Cormen et al. "Introduction to Algorithms, fourth edition". In: MIT Press, 2022. Chap. Appendix B. Sets, Etc. Pp. 1165–1166. ISBN: 9780262046305.

[12] T.H. Cormen et al. "Introduction to Algorithms, fourth edition". In: MIT Press, 2022. Chap. Chapter 22. Single-Source Shortest Paths, pp. 604–605. ISBN: 9780262046305. URL: https://books.google.nl/books?id=HOJyzgEACAAJ.

[13] M. C. J. D. van Eekelen and M. J. Plasmeijer. "Specification of reduction strategies in term rewriting systems". In: *Graph Reduction*. Ed. by Joseph H. Fasel and Robert M. Keller. Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, pp. 215–239. ISBN: 978-3-540-47963-5.

[14] Johri van Eerd et al. "Innermost many-sorted term rewriting on GPUs". In: *Sci. Comput. Program.* 225.C (Jan. 2023). ISSN: 0167-6423. DOI: 10.1016/j.scico.2022.102910. URL: https://doi.org/10.1016/j.scico.2022.102910.

[15] Markus Grönlund and Erik Gahlin. *JEP 328: Flight Recorder*. 2017. URL: https://openjdk.org/jeps/328 (visited on 12/09/2025).

[16] C. A. Hoare. "Algorithm 64: Quicksort". In: *Commun. ACM* 4 (1961), p. 321. URL: https://doi.org/10.1145/366622.366644.

[17] James C. Hoe and Arvind. “Hardware Synthesis from Term Rewriting Systems”. In: *VLSI: Systems on a Chip: IFIP TC10 WG10.5 Tenth International Conference on Very Large Scale Integration (VLSI'99) December 1–4, 1999, Lisboa, Portugal*. Ed. by Luis Miguel Silveira, Srinivas Devadas, and Ricardo Reis. Boston, MA: Springer US, 2000, pp. 595–619. ISBN: 978-0-387-35498-9. DOI: 10.1007/978-0-387-35498-9_52. URL: https://doi.org/10.1007/978-0-387-35498-9_52.

[18] Susan B. Horwitz. *Lambda Calculus (CS704 Notes)*. Definition: An outermost redex is a redex that is not contained inside another one. University of Wisconsin–Madison. URL: https://pages.cs.wisc.edu/~horwitz/CS704-NOTES/1.LAMBDA-CALCULUS.html (visited on 01/05/2026).

[19] Claude Kirchner and Patrick Viry. “Implementing parallel rewriting”. In: *Parallelization in Inference Systems*. Ed. by B. Fronhöfer and G. Wrightson. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 121–138. ISBN: 978-3-540-47066-3.

[20] J. W. Klop. “Chapter 1: Term Rewriting Systems”. In: *Handbook of Logic in Computer Science*. Ed. by S. Abramsky, D. Gabbay, and T. Maibaurn. Oxford University Press, 1992, pp. 1–116.

[21] Cynthia Kop and Naoki Nishida. “Term Rewriting with Logical Constraints”. In: *Frontiers of Combining Systems*. Ed. by Pascal Fontaine, Christophe Ringeissen, and Renate A. Schmidt. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 343–358. ISBN: 978-3-642-40885-4.

[22] Cynthia Kop and Deivid Vale. *COnstrained Rewriting Analyser (CORA)*. 2019. URL: https://github.com/hezzel/cora/ (visited on 12/09/2025).

[23] Vipin Kumar and Vineet Singh. “Scalability of parallel algorithms for the all-pairs shortest-path problem”. In: *Journal of Parallel and Distributed Computing* 13.2 (1991), pp. 124–138. ISSN: 0743-7315. DOI: https://doi.org/10.1016/0743-7315(91)90083-L. URL: https://www.sciencedirect.com/science/article/pii/074373159190083L.

[24] Charles E. Leiserson and Tao B. Schardl. “A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers)”. In: *Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '10. Thira, Santorini, Greece: Association for Computing Machinery, 2010, pp. 303–314. ISBN: 9781450300797. DOI: 10.1145/1810479.1810534. URL: https://doi.org/10.1145/1810479.1810534.

[25] Vincent van Oostrom. “Developing developments”. In: *Theor. Comput. Sci.* 175.1 (Mar. 1997), pp. 159–181. ISSN: 0304-3975. DOI: 10.1016/S0304-3975(96)00173-9. URL: https://doi.org/10.1016/S0304-3975(96)00173-9.

[26] Takuya Takagi et al. “Packed Compact Tries: A Fast and Efficient Data Structure for Online String Processing”. In: *Combinatorial Algorithms*. Ed. by Veli Mäkinen, Simon J. Puglisi, and Leena Salmela. Cham: Springer International Publishing, 2016, pp. 213–225. ISBN: 978-3-319-44543-4.

[27] Mikhail Ushakov. *Cora fork with Memory Term Rewriting execution support*. 2026. URL: https://github.com/mikhirurg/cora (visited on 06/07/2026).

[28] Mikhail Ushakov. *MemTRS-REPL*. 2026. URL: https://github.com/mikhirurg/MemTRS-REPL (visited on 06/07/2026).

[29] Mikhail Ushakov. *MemTRS-STDLIB*. 2026. URL: https://github.com/mikhirurg/MemTRS-STDLIB (visited on 06/07/2026).

[30] Tom Verhoeff. “Quicksort for linked lists”. In: 1993. URL: https://alexandria.tue.nl/extra1/wskrap/publichtml/9310415.pdf.

Appendix A

Experiments Data

A.1 QuickSort Benchmarking

A.1.1 Time Complexity

Number of elements	Time: list QuickSort (ms)	Time: mem QuickSort (ms)
1 (1)	1	1
1 (2)	1	1
1 (3)	1	1
1.000 (avg)	**1.000**	**1.000**
2 (1)	1	5
2 (2)	2	4
2 (3)	1	5
2.000 (avg)	**1.333**	**4.667**
4 (1)	4	10
4 (2)	3	11
4 (3)	3	9
4.000 (avg)	**3.333**	**10.000**
8 (1)	9	45
8 (2)	9	25
8 (3)	10	48
8.000 (avg)	**9.333**	**39.333**
16 (1)	32	79
16 (2)	26	112
16 (3)	27	77
16.000 (avg)	**28.333**	**89.333**
32 (1)	92	217
32 (2)	92	193
32 (3)	90	145
32.000 (avg)	**91.333**	**185.000**
64 (1)	488	458
64 (2)	576	402
64 (3)	506	442
64.000 (avg)	**523.333**	**434.000**
128 (1)	4278	1357
128 (2)	3034	884
128 (3)	3679	1055
128.000 (avg)	**3663.667**	**1098.667**
256 (1)	21927	1910
256 (2)	27365	2691
256 (3)	23228	2769
256.000 (avg)	**24173.333**	**2456.667**
512 (1)	174721	5317
512 (2)	169003	5695
512 (3)	167893	6136
512.000 (avg)	**170539.000**	**5716.000**

Number of elements	Time: list QuickSort (ms)	Time: mem QuickSort (ms)
1024 (1)	1255143	12846
1024 (2)	1516209	11158
1024 (3)	1593761	14409
1024.000 (avg)	**1455037.667**	**12804.333**

A.1.2 Number of Objects Statistics

Number of elements	#Allocated objects (list QSort)	#Allocated objects (mem QSort)
1 (1)	58	62
1 (2)	59	70
1 (3)	58	72
1.000 (avg)	**58.333**	**68.000**
2 (1)	57	77
2 (2)	59	76
2 (3)	59	77
2.000 (avg)	**58.333**	**76.667**
4 (1)	55	146
4 (2)	58	117
4 (3)	59	95
4.000 (avg)	**57.333**	**119.333**
8 (1)	58	118
8 (2)	58	116
8 (3)	58	118
8.000 (avg)	**58.000**	**117.333**
16 (1)	84	150
16 (2)	88	149
16 (3)	58	149
16.000 (avg)	**76.667**	**149.333**
32 (1)	88	178
32 (2)	89	181
32 (3)	89	209
32.000 (avg)	**88.667**	**189.333**
64 (1)	146	269
64 (2)	117	236
64 (3)	118	238
64.000 (avg)	**127.000**	**247.667**
128 (1)	238	357
128 (2)	238	388
128 (3)	237	389
128.000 (avg)	**237.667**	**378.000**
256 (1)	748	536
256 (2)	776	537
256 (3)	748	597
256.000 (avg)	**757.333**	**556.667**
512 (1)	4391	1027
512 (2)	4510	1201
512 (3)	4959	1010
512.000 (avg)	**4620.000**	**1079.333**
1024 (1)	52055	2055
1024 (2)	51984	1737
1024 (3)	51783	1674
1024.000 (avg)	**51940.667**	**1822.000**

A.1.3 GC Pause Statistics

Number of elements	Total GC Pause Time (list QSort)	Total GC Pause Time (mem QSort)
1 (1)	68	63
1 (2)	63	69
1 (3)	62	93
1.000 (avg)	**64.333**	**75.000**
2 (1)	62	86
2 (2)	58	91
2 (3)	61	90
2.000 (avg)	**60.333**	**89.000**
4 (1)	63	101
4 (2)	59	145
4 (3)	58	92
4.000 (avg)	**60.000**	**112.667**
8 (1)	61	87
8 (2)	61	91
8 (3)	61	87
8.000 (avg)	**61.000**	**88.333**
16 (1)	62	96
16 (2)	63	108
16 (3)	59	89
16.000 (avg)	**61.333**	**97.667**
32 (1)	73	103
32 (2)	60	98
32 (3)	59	105
32.000 (avg)	**64.000**	**102.000**
64 (1)	74	117
64 (2)	63	100
64 (3)	61	99
64.000 (avg)	**66.000**	**105.333**
128 (1)	60	110
128 (2)	58	133
128 (3)	55	111
128.000 (avg)	**57.667**	**118.000**
256 (1)	74	108
256 (2)	78	110
256 (3)	81	115
256.000 (avg)	**77.667**	**111.000**
512 (1)	437	111
512 (2)	478	147
512 (3)	515	105
512.000 (avg)	**476.667**	**121.000**
1024 (1)	9958	136
1024 (2)	10087	108
1024 (3)	9363	105
1024.000 (avg)	**9802.667**	**116.333**

A.1.4 Total Parallel Reduction Steps

Number of elements	Parallel steps: list QuickSort	Parallel steps: mem QuickSort
1 (1)	5	5
1 (2)	5	5
1 (3)	5	5
1.000 (avg)	**5.000**	**5.000**
2 (1)	9	51
2 (2)	9	51
2 (3)	9	39
2.000 (avg)	**9.000**	**47.000**
4 (1)	20	182
4 (2)	19	103
4 (3)	20	148
4.000 (avg)	**19.667**	**144.333**
8 (1)	50	415
8 (2)	45	496
8 (3)	40	272
8.000 (avg)	**45.000**	**394.333**
16 (1)	101	592
16 (2)	99	1053
16 (3)	94	1101
16.000 (avg)	**98.000**	**915.333**
32 (1)	240	1681
32 (2)	238	1294
32 (3)	258	2062
32.000 (avg)	**245.333**	**1679.000**
64 (1)	532	4385
64 (2)	571	4432
64 (3)	551	2611
64.000 (avg)	**551.333**	**3809.333**
128 (1)	1210	12957
128 (2)	1180	7990
128 (3)	1199	11062
128.000 (avg)	**1196.333**	**10669.667**
256 (1)	2819	15456
256 (2)	2787	10986
256 (3)	2819	13155
256.000 (avg)	**2808.333**	**13199.000**
512 (1)	6185	22432
512 (2)	6338	29021
512 (3)	6776	26825
512.000 (avg)	**6433.000**	**26092.667**
1024 (1)	15071	47288
1024 (2)	14638	54890
1024 (3)	14547	45247
1024.000 (avg)	**14752.000**	**49141.667**

A.1.5 Memory QuickSort CPU-Time Profiling

#	Method	Thread	Samples	Percent
1	java.util.concurrent.ForkJoinPool.signalWork()	ForkJoinPool-1-worker-4	902	6.60%
2	cora.reduction.RuleReducer.applicable(Term)	*virtual*	806	5.89%
3	java.util.concurrent.ForkJoinPool .runWorker(ForkJoinPool$WorkQueue)	ForkJoinPool-1-worker-2	672	4.91%
4	charlie.terms.TermInherit .calculateFreeReplaceablesForSubterms(List, ReplaceableList)	main	571	4.17%
5	charlie.trs.Rule.queryType()	*virtual*	558	4.08%
6	charlie.terms.Application.construct(Term, List)	main	545	3.98%
7	charlie.terms.Application.match(Term, Substitution)	*virtual*	484	3.54%
8	java.util.concurrent.ForkJoinPool .deactivate(ForkJoinPool$WorkQueue, int)	ForkJoinPool-1-worker-10	482	3.52%
9	cora.reduction.Reducer.lambda$reduce$2(Term, Position)	*virtual*	474	3.47%
10	java.lang.String.equals(Object)	main	364	2.66%
11	charlie.terms.TermInherit .calculateBoundVariablesAndRefreshSubs(List, ReplaceableList, ReplaceableList, ImmutableList$Builder)	main	311	2.27%
12	java.lang.AbstractStringBuilder.append(String)	main	286	2.09%
13	java.lang.StringBuilder.append(String)	main	272	1.99%
14	cora.reduction.RuleReducer .findHeadAdditions(Term)	*virtual*	261	1.91%
15	java.lang.String.getBytes(byte[], int, byte)	main	247	1.81%
16	charlie.terms.ValueInherit.numberArguments()	*virtual*	186	1.36%
17	com.google.common.collect.RegularImmutableList .size()	main	159	1.16%
18	charlie.terms.Application.queryType()	main	146	1.07%
19	charlie.terms.Application.queryArgument(int)	main	135	0.99%
20	java.lang.AbstractStringBuilder.<init>(int)	main	135	0.99%
21	java.lang.AbstractStringBuilder .needsNewBuffer(byte[], byte, int, byte)	main	134	0.98%
22	java.util.concurrent.ForkJoinPool .awaitWork(ForkJoinPool$WorkQueue, int)	ForkJoinPool-1-worker-9	129	0.94%
23	com.google.common.collect.RegularImmutableList .get(int)	main	123	0.90%
24	charlie.terms.TermPrinter.print(Term, Renaming, StringBuilder)	*virtual*	123	0.90%
25	java.util.TreeSet.size()	main	122	0.89%
26	java.util.TreeMap.getFirstEntry()	main	112	0.82%
27	java.lang.StringConcatHelper.prepend(int, byte, byte[], String, String)	main	112	0.82%
28	java.util.concurrent.ConcurrentHashMap .tabAt(ConcurrentHashMap$Node[], int)	main	98	0.72%
29	java.util.ArrayList.elementData(int)	main	98	0.72%
30	charlie.terms.Application.<init>(Term, List)	main	93	0.68%
31	charlie.terms.ValueInherit.queryType()	main	92	0.67%
32	charlie.terms.Application.querySubterms()	main	91	0.67%
33	java.lang.String.length()	main	91	0.67%
34	java.util.HashMap.getNode(Object)	main	90	0.66%
35	cora.reduction.CalcReducer.applicable(Term)	*virtual*	88	0.64%
36	charlie.terms.ValueInherit.isApplication()	*virtual*	86	0.63%
37	charlie.terms.Subst.<init>()	*virtual*	81	0.59%
38	java.util.concurrent.ConcurrentHashMap .replaceNode(Object, Object, Object)	main	76	0.56%
39	java.util.TreeMap$KeySet.iterator()	main	69	0.50%
40	charlie.terms.FunctionSymbol .compareTo(FunctionSymbol)	main	64	0.47%

#	Method	Thread	Samples	Percent
41	**charlie.terms.Application.numberArguments()**	main	62	0.45%
42	**java.util.concurrent.ConcurrentHashMap .addCount(long, int)**	main	58	0.42%
43	**java.util.Arrays.copyOf(byte[], int)**	main	57	0.42%
44	**cora.reduction.Reducer.cbvReductionOK(Term)**	main	57	0.42%
45	**charlie.terms.LeafTermInherit.queryType()**	main	53	0.39%
46	**charlie.terms.ValueInherit .queryImmediateHeadSubterm(int)**	*virtual*	53	0.39%
47	**java.lang.VirtualThread.unmount()**	ForkJoinPool-1-worker-5	51	0.37%
48	**java.lang.VirtualThread.compareAndSetState(int, int)**	ForkJoinPool-1-worker-11	50	0.37%
49	**charlie.terms.Var.match(Term, Substitution)**	*virtual*	49	0.36%
50	**java.lang.VirtualThread$VThreadContinuation$1 .run()**	*virtual*	44	0.32%
51	**java.util.ArrayList.get(int)**	main	43	0.31%
52	**java.lang.AbstractStringBuilder .ensureCapacityNewCoder(byte[], byte, int, int, byte)**	main	41	0.30%
53	**java.util.Arrays.copyOfRange(byte[], int, int)**	main	41	0.30%
54	**charlie.terms.ValueInherit.queryType()**	main	40	0.29%
55	**charlie.terms.Application .queryImmediateHeadSubterm(int)**	*virtual*	40	0.29%
56	**jdk.internal.misc.Unsafe.getAndBitwiseOrInt(Object, long, int)**	ForkJoinPool-1-worker-5	38	0.28%
57	**charlie.terms.TermInherit.match(Term)**	*virtual*	38	0.28%
58	**charlie.terms.ReplaceableList .combine(ReplaceableList)**	main	37	0.27%
59	**cora.reduction.BetaReducer.applicable(Term)**	*virtual*	35	0.26%
60	**java.util.Arrays.copyOf(Object[], int)**	main	35	0.26%
61	**charlie.types.TypePrinter.print(Type, StringBuilder)**	main	34	0.25%
62	**java.util.TreeMap.<init>()**	main	34	0.25%
63	**java.util.HashMap.resize()**	main	32	0.23%
64	**java.lang.Thread.<init>(String, int, boolean)**	main	31	0.23%
65	**charlie.terms.ReplaceableList.size()**	main	31	0.23%
66	**java.lang.invoke.VarHandleInts $FieldInstanceReadWrite .compareAndSet(VarHandle, Object, int, int)**	*virtual*	31	0.23%
67	**jdk.internal.util.DecimalDigits.stringSize(int)**	*virtual*	31	0.23%
68	**java.util.ArrayList.size()**	main	30	0.22%
69	**java.util.concurrent.ForkJoinPool .compareAndSetCtl(long, long)**	ForkJoinPool-1-worker-11	30	0.22%
70	**java.util.concurrent.ConcurrentHashMap .casTabAt(ConcurrentHashMap$Node[], int, ConcurrentHashMap$Node, ConcurrentHashMap$Node)**	main	30	0.22%
71	**java.lang.Integer.toString(int)**	*virtual*	30	0.22%
72	**java.lang.System$1.setScopedValueCache(Object[])**	ForkJoinPool-1-worker-7	29	0.21%
73	**java.lang.invoke.VarHandleReferences $FieldInstanceReadWrite .compareAndSet(VarHandle, Object, Object, Object)**	*virtual*	29	0.21%
74	**jdk.internal.misc.Unsafe .compareAndSetBoolean(Object, long, boolean, boolean)**	ForkJoinPool-1-worker-2	29	0.21%
75	**com.google.common.collect.SingletonImmutableList .size()**	*virtual*	28	0.20%
76	**java.util.ArrayList.grow(int)**	main	27	0.20%
77	**cora.reduction.MemReducer.checkIfGET(Term)**	*virtual*	27	0.20%
78	**java.util.concurrent.ForkJoinTask$InterruptibleTask .exec()**	ForkJoinPool-1-worker-1	27	0.20%

#	Method	Thread	Samples	Percent
79	**java.util.HashMap.hash(Object)**	*virtual*	26	0.19%
80	**charlie.terms.Subst.extend(Replaceable, Term)**	*virtual*	25	0.18%
81	**java.util.HashMap.putVal(int, Object, Object, boolean, boolean)**	main	24	0.18%
82	**charlie.terms.position.ArgumentPos.toString()**	main	24	0.18%
83	**charlie.util.Pair.<init>(Object, Object)**	main	24	0.18%
84	**charlie.terms.TermInherit .setVariables(ReplaceableList, ReplaceableList)**	main	23	0.17%
85	**com.google.common.collect.ImmutableList .asImmutableList(Object[], int)**	main	23	0.17%
86	**java.lang.StringBuilder.toString()**	main	23	0.17%
87	**charlie.terms.TermPrinter.printInfixHelper(Term, Renaming, StringBuilder, int)**	*virtual*	23	0.17%
88	**charlie.terms.Application.setupReplaceables(List)**	*virtual*	22	0.16%
89	**charlie.types.Base$$TypeSwitch.0x000000006f0e4000 .typeSwitch(Object, int)**	*virtual*	22	0.16%
90	**charlie.terms.TermPrinter .generateUniqueNaming(List)**	*virtual*	22	0.16%
91	**jdk.internal.util.DecimalDigits .uncheckedGetCharsLatin1(int, int, byte[])**	*virtual*	21	0.15%
92	**cora.reduction.MemReducer.applicable(Term)**	*virtual*	21	0.15%
93	**java.lang.String.<init>(AbstractStringBuilder, Void)**	main	21	0.15%
94	**charlie.trs.Rule.queryLeftSide()**	*virtual*	21	0.15%
95	**java.util.concurrent.FutureTask.run()**	*virtual*	20	0.15%
96	**java.lang.String.compareTo(String)**	main	20	0.15%
97	**charlie.terms.ValueInherit.queryRoot()**	*virtual*	20	0.15%
98	**java.util.concurrent.ConcurrentHashMap .putVal(Object, Object, boolean)**	main	20	0.15%
99	**java.lang.VirtualThread .externalSubmitRunContinuationOrThrow()**	main	20	0.15%
100	**charlie.terms.TermInherit.equals(Term)**	*virtual*	20	0.15%

A.1.6 List QuickSort CPU-Time Profiling

#	Method	Thread	Samples	Percent
1	**cora.reduction.Reducer.cbvReductionOK(Term)**	main	12,839	16.51%
2	**charlie.terms.Application.querySubterms()**	main	9,108	11.71%
3	**java.util.concurrent.ForkJoinPool .deactivate(ForkJoinPool$WorkQueue, int)**	ForkJoinPool-1-worker-6	6,054	7.78%
4	**charlie.terms.Application.queryArgument(int)**	main	5,535	7.12%
5	**java.util.concurrent.ForkJoinPool .runWorker(ForkJoinPool$WorkQueue)**	ForkJoinPool-1-worker-9	3,671	4.72%
6	**java.util.LinkedList.unlinkFirst(LinkedList$Node)**	main	3,046	3.92%
7	**charlie.util.Pair.<init>(Object, Object)**	main	2,258	2.90%
8	**java.util.LinkedList.linkLast(Object)**	main	1,960	2.52%
9	**charlie.terms.TermInherit .calculateFreeReplaceablesForSubterms(List, ReplaceableList)**	main	1,423	1.83%
10	**charlie.terms.Application.construct(Term, List)**	main	1,153	1.48%
11	**com.google.common.collect .RegularImmutableList.size()**	main	1,096	1.41%
12	**java.util.concurrent.ForkJoinPool.signalWork()**	ForkJoinPool-1-worker-3	1,086	1.40%
13	**java.util.ArrayList.add(Object, Object[], int)**	main	1,039	1.34%
14	**charlie.terms.ValueInherit.isVariable()**	main	988	1.27%
15	**java.util.ArrayList$Itr.next()**	main	976	1.25%
16	**java.util.HashMap.getNode(Object)**	main	953	1.23%
17	**charlie.terms.Application.numberArguments()**	main	937	1.20%
18	**java.util.LinkedList$Node.¡init¿(LinkedList$Node, Object, LinkedList$Node)**	main	885	1.14%
19	**java.util.TreeMap.getFirstEntry()**	main	820	1.05%
20	**java.util.TreeSet.iterator()**	main	773	0.99%
21	**charlie.terms.Application.match(Term, Substitution)**	*virtual*	767	0.99%
22	**charlie.terms.TermInherit .calculateBoundVariablesAndRefreshSubs(List, ReplaceableList, ReplaceableList, ImmutableList$Builder)**	*virtual*	746	0.96%
23	**charlie.terms.VariableEnvironment.iterator()**	main	674	0.87%
24	**charlie.terms.TermInherit.isValue()**	main	622	0.80%
25	**charlie.terms.TermInherit.isVariable()**	main	615	0.79%
26	**java.util.ArrayList.grow(int)**	main	575	0.74%
27	**java.util.Arrays.copyOf(Object[], int)**	main	570	0.73%
28	**cora.reduction.RuleReducer.applicable(Term)**	*virtual*	542	0.70%
29	**com.google.common.collect.RegularImmutableList .get(int)**	main	501	0.64%
30	**charlie.terms.ValueInherit.isValue()**	main	481	0.62%
31	**java.lang.String.equals(Object)**	RMI TCP Connection(idle)	476	0.61%
32	**java.util.TreeMap$KeySet.iterator()**	main	461	0.59%
33	**charlie.util.Pair.fst()**	main	435	0.56%
34	**cora.reduction.Reducer.lambda$reduce$2(Term, Position)**	*virtual*	413	0.53%
35	**charlie.util.Pair.snd()**	main	368	0.47%
36	**java.util.ArrayList.add(Object)**	main	362	0.47%
37	**charlie.terms.position.ArgumentPos.<init>(int, Position)**	main	341	0.44%
38	**java.util.concurrent.ForkJoinPool .awaitWork(ForkJoinPool$WorkQueue, int)**	ForkJoinPool-1-worker-1	298	0.38%
39	**java.util.concurrent.ConcurrentHashMap .tabAt(ConcurrentHashMap$Node[], int)**	main	297	0.38%
40	**charlie.terms.Application.<init>(Term, List)**	*virtual*	292	0.38%
41	**charlie.terms.TermInherit.vars()**	main	287	0.37%
42	**java.util.concurrent.ConcurrentHashMap .replaceNode(Object, Object, Object)**	*virtual*	285	0.37%
43	**java.util.ArrayList.grow()**	main	283	0.36%

#	Method	Thread	Samples	Percent
44	**charlie.terms.TermPrinter.print(Term, Renaming, StringBuilder)**	main	267	0.34%
45	**charlie.terms.ReplaceableList.iterator()**	main	249	0.32%
46	**java.util.TreeSet.size()**	*virtual*	237	0.30%
47	**java.util.TreeMap.navigableKeySet()**	main	233	0.30%
48	**java.util.concurrent.ConcurrentHashMap .addCount(long, int)**	main	223	0.29%
49	**jdk.internal.util.DecimalDigits.stringSize(int)**	*virtual*	211	0.27%
50	**java.lang.VirtualThread.compareAndSetState(int, int)**	main	209	0.27%
51	**java.lang.Thread.<init>(String, int, boolean)**	main	201	0.26%
52	**java.lang.VirtualThread.unmount()**	ForkJoinPool-1-worker-7	201	0.26%
53	**java.util.HashMap.hash(Object)**	main	194	0.25%
54	**java.util.concurrent.ForkJoinPool .compareAndSetCtl(long, long)**	ForkJoinPool-1-worker-6	185	0.24%
55	**java.util.concurrent.ForkJoinTask$InterruptibleTask .exec()**	ForkJoinPool-1-worker-12	180	0.23%
56	**charlie.terms.Application.queryRoot()**	main	171	0.22%
57	**java.util.concurrent.ConcurrentHashMap .casTabAt(ConcurrentHashMap$Node[], int, ConcurrentHashMap$Node, ConcurrentHashMap$Node)**	main	168	0.22%
58	**java.lang.String.hashCode()**	main	153	0.20%
59	**cora.reduction.RuleReducer .findHeadAdditions(Term)**	*virtual*	152	0.20%
60	**java.lang.invoke.VarHandleReferences $FieldInstanceReadWrite.compareAndSet(VarHandle, Object, Object, Object)**	*virtual*	152	0.20%
61	**jdk.internal.util.DecimalDigits .uncheckedGetCharsLatin1(int, int, byte[])**	*virtual*	145	0.19%
62	**charlie.terms.Application.queryType()**	*virtual*	142	0.18%
63	**java.util.HashSet.contains(Object)**	main	140	0.18%
64	**cora.reduction.BetaReducer.applicable(Term)**	*virtual*	135	0.17%
65	**jdk.internal.misc.Unsafe.getAndBitwiseOrInt(Object, long, int)**	ForkJoinPool-1-worker-10	132	0.17%
66	**java.lang.String.getBytes(byte[], int, byte)**	main	132	0.17%
67	**java.lang.VirtualThread.runContinuation()**	ForkJoinPool-1-worker-9	123	0.16%
68	**jdk.internal.misc.Unsafe .compareAndSetBoolean(Object, long, boolean, boolean)**	ForkJoinPool-1-worker-1	120	0.15%
69	**jdk.internal.util.DecimalDigits .uncheckedPutPairLatin1(byte[], int, int)**	*virtual*	118	0.15%
70	**java.lang.Integer.toString(int)**	*virtual*	118	0.15%
71	**charlie.terms.Var.match(Term, Substitution)**	*virtual*	115	0.15%
72	**java.util.HashMap.resize()**	main	114	0.15%
73	**java.lang.VirtualThread$VThreadContinuation$1 .run()**	*virtual*	110	0.14%
74	**java.util.concurrent.ConcurrentHashMap .putVal(Object, Object, boolean)**	main	103	0.13%
75	**cora.reduction.CalcReducer.applicable(Term)**	*virtual*	101	0.13%
76	**java.util.LinkedList.removeFirst()**	main	99	0.13%
77	**java.lang.System$1.setScopedValueCache(Object[])**	*ForkJoinPool-1-worker-12*	97	0.12%
78	**jdk.internal.vm.Continuation.run()**	ForkJoinPool-1-worker-4	96	0.12%
79	**charlie.terms.ValueInherit.queryType()**	*virtual*	96	0.12%
80	**java.util.concurrent.ForkJoinPool$WorkQueue .push(ForkJoinTask, ForkJoinPool, boolean)**	main	94	0.12%
81	**charlie.terms.Application .queryImmediateHeadSubterm(int)**	*virtual*	94	0.12%
82	**charlie.terms.Constant.alphaEquals(Term, Map, Map, int)**	*virtual*	93	0.12%

#	Method	Thread	Samples	Percent
83	**java.util.concurrent.ForkJoinPool.slotOffset(int)**	ForkJoinPool-1-worker-11	91	0.12%
84	**cora.reduction.Reducer.lambda$reduce$1(Pair)**	main	91	0.12%
85	**java.lang.Thread$ThreadIdentifiers.next()**	main	91	0.12%
86	**java.lang.VirtualThread.setCarrierThread(Thread)**	ForkJoinPool-1-worker-4	90	0.12%
87	**java.util.HashMap.putVal(int, Object, Object, boolean, boolean)**	*virtual*	89	0.11%
88	**cora.reduction.Reducer.reduce(Term)**	main	86	0.11%
89	**java.lang.invoke.VarHandleInts $FieldInstanceReadWrite .compareAndSet(VarHandle, Object, int, int)**	*virtual*	86	0.11%
90	**charlie.trs.Rule.queryType()**	*virtual*	81	0.10%
91	**charlie.terms.Application.setupReplaceables(List)**	*virtual*	81	0.10%
92	**charlie.terms.TermInherit.hashCode()**	*virtual*	78	0.10%
93	**charlie.terms.Subst.extend(Replaceable, Term)**	*virtual*	76	0.10%
94	**java.util.concurrent.ForkJoinPool$WorkQueue .tryLockPhase()**	main	74	0.10%
95	**java.util.concurrent.FutureTask.run()**	*virtual*	72	0.09%
96	**java.util.TreeMap.<init>()**	*virtual*	72	0.09%
97	**java.util.TreeMap$KeyIterator .<init>(TreeMap, TreeMap$Entry)**	*virtual*	68	0.09%
98	**java.util.concurrent.FutureTask.<init>(Callable)**	main	67	0.09%
99	**cora.reduction.MemReducer.applicable(Term)**	*virtual*	64	0.08%
100	**cora.reduction.MemReducer.checkIfGET(Term)**	*virtual*	64	0.08%

A.1.7 Memory QuickSort Thread Statistics

Table A.1: Thread Statistics: Memory QuickSort

#	Thread	Samples	Percent
1	*virtual*	8,692	63.55%
2	main	2,394	17.50%
3	ForkJoinPool-1-worker-2	238	1.74%
4	ForkJoinPool-1-worker-11	235	1.72%
5	ForkJoinPool-1-worker-8	233	1.70%
6	ForkJoinPool-1-worker-7	231	1.69%
7	ForkJoinPool-1-worker-12	216	1.58%
8	ForkJoinPool-1-worker-4	215	1.57%
9	ForkJoinPool-1-worker-5	213	1.56%
10	ForkJoinPool-1-worker-3	212	1.55%
11	ForkJoinPool-1-worker-10	211	1.54%
12	ForkJoinPool-1-worker-9	206	1.51%
13	ForkJoinPool-1-worker-6	184	1.35%
14	ForkJoinPool-1-worker-1	184	1.35%
15	RMI TCP Connection(1)-192.168.50.145	12	0.09%
16	JFR Shutdown Hook	1	0.01%

A.1.8 List QuickSort Thread Statistics

Table A.2: Thread Statistics: List QuickSort

#	Thread	Samples	Percent
1	main	51,010	65.58%
2	*virtual*	13,992	17.99%
3	ForkJoinPool-1-worker-9	1,137	1.46%
4	ForkJoinPool-1-worker-11	1,108	1.42%
5	ForkJoinPool-1-worker-12	1,097	1.41%
6	ForkJoinPool-1-worker-2	1,068	1.37%
7	ForkJoinPool-1-worker-10	1,065	1.37%
8	ForkJoinPool-1-worker-1	1,064	1.37%
9	ForkJoinPool-1-worker-7	1,056	1.36%
10	ForkJoinPool-1-worker-3	1,045	1.34%
11	ForkJoinPool-1-worker-8	1,031	1.33%
12	ForkJoinPool-1-worker-4	1,025	1.32%
13	ForkJoinPool-1-worker-5	1,020	1.31%
14	ForkJoinPool-1-worker-6	1,000	1.29%
15	RMI TCP Connection(idle)	46	0.06%
16	JFR Shutdown Hook	9	0.01%
17	JFR Periodic Tasks	7	0.01%
18	Attach Listener	5	0.01%
19	JMX server connection timeout 21049	1	0.00%