



## MASTER RESEARCH INTERNSHIP



## INTERNSHIP REPORT

---

# Scheduling data structures with term rewriting techniques

---

**Domain: Formal Languages and Automata Theory - Programming Languages**

*Author:*  
Thais BAUDON

*Supervisors:*  
Laure GONNORD (CASH, Lyon)  
Carsten FUHS (Birkbeck, Univ.  
of London)

**Abstract:** In this internship, we study the problem of scheduling programs with complex data structures. As related work, we show that although scheduling operations on efficient data structures apart from arrays should have a great impact on performance, there is nearly no work on this topic. However, some related work from the rewriting community gives us great insights about the link between termination and scheduling, that we propose to study further. Our contributions as a first step toward full efficient compilation of programs with inductive data structures are 1. first algorithms for this parallel evaluation, 2. a code generation algorithm to generate efficient parallel evaluators, 3. a prototype implementation and first experimental results.

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Instruction and data structures scheduling</b>	<b>2</b>
<b>3</b>	<b>Term rewriting systems and termination</b>	<b>6</b>
<b>4</b>	<b>Termination and scheduling</b>	<b>10</b>
<b>5</b>	<b>Parallel evaluation of term rewriting systems</b>	<b>15</b>
<b>6</b>	<b>A code generator for a parallel evaluator based on partial evaluation</b>	<b>22</b>
<b>7</b>	<b>Prototype and first experimental results</b>	<b>29</b>
<b>8</b>	<b>Conclusion and future work</b>	<b>32</b>
<b>A</b>	<b>Full final generated code for tree size</b>	<b>33</b>

# 1 Introduction

**General context: CODAS project** This internship takes place in the context of the ANR CODAS project, in which Laure Gonnord and Carsten Fuhs are involved.

The long term objective of the ANR CODAS project is to give a general way to reason about and manipulate programs with general control flow and complex data structures, for HPC applications.

In the context of the project, Paul Iannetta is pursuing his PhD studies on the definition of a general compilation framework that includes trees (in the *polyhedral model* spirit, briefly discussed in [Section 2](#)), but has not explored the relationship between inductive data structures and term rewriting.

The goal of this internship was to explore this idea. Indeed, the problem of scheduling is closely linked to the well-studied topic of termination, and we investigated the benefits we could encounter from the rewriting community that has come to nice results on termination in the last twenty years. Term rewriting systems are a core rule-based programming language that captures inductive data structures via pattern matching and can express concepts from functional and imperative programming.

**Context** More precisely, the focus of this internship is *static* (i.e., compile-time) scheduling of individual programs containing inductive data structures (such as lists and trees), as opposed to dynamic/run-time scheduling approaches or coarse-grain task scheduling. The approach we finally propose is *hybrid* in the sense that we were able to statically generate a partial evaluator that delegates some dynamic scheduling choices to the (OCaml) runtime.

**Outline** [Section 2](#) reports on related work and states the problem of scheduling programs operating on inductive data structures. After recalling useful definitions for term rewriting systems (in [Section 3](#)), [Section 4](#) gives first insights into the link between scheduling and termination proofs.

During the internship, we explored how term rewriting systems (TRS) algorithms are capable of handling programs on inductive data structures.

We use term rewriting systems as *intermediate* representations of programs operating on inductive data structures. From this choice, we propose to exploit the notion of parallel evaluation of terms to infer parallelism/absence of dependency. This report describes the following contributions:

- two first algorithms for *parallel evaluation* of term rewriting systems ([Section 5](#));
- a code generation algorithm that exploits further the notion of *structural dependency* ([Section 6](#));
- the implementation of the code generation algorithm and a preliminary experimental evaluation ([Section 7](#)).

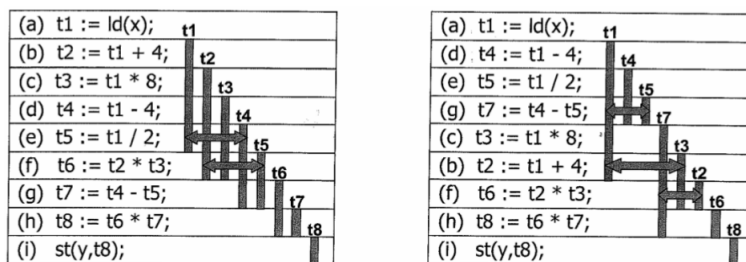
## 2 Instruction and data structures scheduling

In this section, we recall the problem of instruction scheduling, and the global objective of scheduling programs on inductive data structures.

This section reproduces in great part the bibliography document produced before the internship <sup>1</sup>, but focuses more on the parts directly related to the obtained results. In particular, we do not cover the details of polyhedral-based imperative scheduling.

**Overview** Instruction scheduling consists in reordering instructions so as to minimise execution time and resource usage. It is traditionally done in the compiler's *middle end*. One of its main objectives is to decrease register pressure.

**Example 1 (Better schedule means better register allocation: taken from the Dragon Book [Aho et al., 1986])** *The two following versions of the 3-address code of a given block are semantically equivalent, however the version on the right will generate a better register allocation since the maximum number of simultaneously alive registers is 3 (comparing to 4 on the left):*



We also note that (d) has been moved backward without breaking the semantics since it does not use any value provided or invalidated by (b) or (c).

In the last example, we can see two crucial concepts: the notion of *scheduling*, that assigns a (logical) date to each statement, and the notion of memory *dependencies*, that should not be broken by a rescheduling. These notions will be defined later.

### 2.1 Scheduling arrays in the imperative world

One of the most well-known algorithms used for scheduling imperative programs with arrays, such as the one depicted in Example 2 is the one from Karp/Miller/Winograd [Karp et al., 1967], which proposes a way to schedule a set of uniform recurrence

<sup>1</sup>This document can be found at the following URL: <http://perso.eleves.ens-rennes.fr/people/thais.baudon/biblio.pdf>

equations. More recent and more expressive algorithms of course exist, but we think that the program model and the associated schedule provides more evidence of the similarities with term rewriting systems.

---

**Example 2 (An imperative regular loop with arrays)**

---

```
for (p = 7; p < N; p++) {
  A1[p] = A2[p-1] + A3[p-3];
  A2[p] = A1[p-2] * A2[p-6] - A3[p-4] * A3[p-7];
  A3[p] = A2[p-5] + 1;
}
```

---

The notion of (read/write) dependency is straightforward (thus we do not give a formal definition of it): in order to safely compute  $A_1[p]$ , the compilation of  $A_3[p-3]$  should have been performed before.

Sets of uniform recurrence equations are proposed to model and reason on these imperative programs. Example 3 depicts these recurrence equations ( $f_i$  denote scalar operations, the enclosing loop is implicit). The equations are called uniform since each computation (for instance  $a_1(p)$ ), depends on other ones with a constant offset (for instance  $a_3$  with offset 3).

We give the formal notion of *uniform* dependency, which will later show some similarities with the notion of structural dependency. With this definition, we are able to say that  $(1, p)$  depends directly on  $(3, p-3)$ .

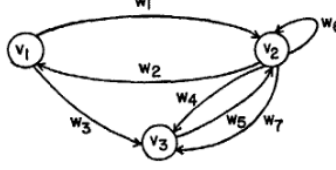
**Definition 1 (Uniform dependency)** *The notion of uniform dependency is defined by the existence of a set of  $m$  vectors  $w_1, \dots, w_m \in \mathbb{Z}^n$  such that for every  $i \in \{1, \dots, m\}$  and all  $p \in R$ :*

- *(additional constraint, that we do not describe here)*
- *the computation of  $a_i(p)$  is uniformly dependent on  $k$  other computations indexed by  $i_1, \dots, i_k$ , that is:*  
 $a_i(p) = f_i(a_{i_1}(p - w_{i_1}), \dots, a_{i_k}(p - w_{i_k}))$ , where the function  $f_i$  is not constant w.r.t. any of its  $k$  variables.  
*Then for each  $j \in \{i_1, \dots, i_k\}$ ,  $(i, p)$  is said to depend directly on  $(j, p - w_j)$ .*

From these recurrence equations and the notion of dependency, a dependency graph is constructed, which is also depicted in Example 3. Defining such a graph is here possible since the “distance” between reads and writes are constant. This dependency distance is depicted on the arrows of the graph.

**Example 3 (System of uniform recurrence equations and its dependency graph, taken from [Karp et al., 1967], that would come from the program depicted in Example 2)**

$$\begin{aligned}
a_1(p) &= f_1(a_2(p-1), a_3(p-3)) \\
a_2(p) &= f_2(a_1(p-2), a_2(p-6), \\
&\quad a_3(p-4), a_3(p-7)) \\
a_3(p) &= f_3(a_2(p-5))
\end{aligned}$$



**Definition 2 (Scheduling [Karp et al., 1967])** A schedule  $S$  is a function from  $\{1, 2, \dots, m\} \times R$  into the positive integers such that if  $(k, p) \rightarrow (l, q)$  then  $S(k, p) > S(l, q)$ . The quantity  $S(k, p)$  may be interpreted as the time at which  $a_k(p)$  is computed (assuming each such computation requires exactly one unit of time).

Intuitively, assuming each computation requires exactly one unit of time, a schedule  $S$  assigns a positive timestamp  $S(k, p)$  to every computation  $a_k(p)$  such that if  $(k, p) \rightarrow (k', p')$ , then  $S(k, p) > S(k', p')$ . A schedule may expose potential parallelism by assigning the same timestamp to two or more independent computations.

A schedule can be seen as a concretisation of the dependencies, and computing a schedule can be seen as some kind of topologic sort on the dependency graph.

The paper [Karp et al., 1967] proposes to compute a *free schedule* that maximises parallelism (i.e., it maximises the number of “common dates of computations”). More precisely, if it exists (which is shown to be equivalent to termination), the free schedule is derived from a system of constraints expressing that:

- The schedule is valid: it assigns a positive date to every computation and does not break any dependency (i.e.,  $(k, p) \rightarrow (l, q) \Rightarrow S(k, p) > S(l, q)$ ).
- There is maximal parallelism: for any  $k, p$ , the timestamp assigned by the free schedule is lower than or equal to the timestamp assigned by any other schedule.

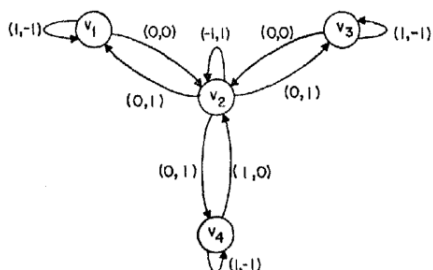
Moreover, if the free schedule exists, it can be expressed with the following formulas:

$$\begin{aligned}
T(k, p) &= \begin{cases} 1 + \max \{T(l, q) \mid q \in R \wedge (k, p) \xrightarrow{1} (l, q)\} \\ 1 & \text{if no such } (l, q) \text{ exists} \end{cases} \\
&= 1 + \max \{t \geq 0 \mid \exists (l, q), (k, p) \xrightarrow{t} (l, q)\}
\end{aligned}$$

that imply that such a schedule can be statically computed greedily on the dependency graph, as we show in Example 4.

**Example 4 (Example taken from [Karp et al., 1967]: computation of a new schedule)** Given the dependency graph on the left, the table on the right details the associated free schedule  $T$ . Given the computation dependencies on  $a_2$ , the computation  $a_2(1, 1)$  should be done strictly after  $a_1(1, 1)$ ,  $a_3(1, 1)$ ,  $a_4(1, 1)$ , that it depends on (for

instance  $a_3(1, 1) \xrightarrow{(0,0)} a_2(1, 1)$ ). The free schedule is greedily obtained by beginning from the set of “initial” computations and walking through dependences (and subtracting the weight of the edges to the current “iteration vector”).



$\tau$	$T$		
1	$a_1(1, 1)$	$a_3(1, 1)$	$a_4(1, 1)$
2		$a_2(1, 1)$	
3	$a_1(1, 2)$	$a_3(1, 2)$	$a_4(1, 2)$
4	$a_1(2, 1)$	$a_3(2, 1)$	$a_4(2, 1)$
5		$a_2(2, 1)$	
6		$a_2(1, 2)$	
7	$a_1(1, 3)$	$a_3(1, 3)$	$a_4(1, 3)$
8	$a_1(2, 2)$	$a_3(2, 2)$	$a_4(2, 2)$
9	$a_1(3, 1)$	$a_3(3, 1)$	$a_4(3, 1)$
10		$a_2(3, 1)$	

The paper also provides an upper bound on the timestamp assigned to each computation by the free schedule, which is in our knowledge the first mention and usage of the relationship between termination and scheduling. In [Alias et al., 2010], this relationship and an adaptation of the algorithm are used to automatically derive ranking functions, with the aim to automatically prove program termination and bounds on its maximal runtime (see Section 4.2).

**Remark 1** *This section was devoted to the study of scheduling recurrence equations that are classically obtained from (a particular set of) imperative programs, for which the proposed scheduling strategy achieves the fastest possible schedule. However, such a set of recurrence equations can also be viewed as rewriting rules; or could be obtained from recursive programs, as we will show later on.*

## 2.2 Trees and other complex/inductive data structures

The literature of scheduling barely explores the optimised scheduling of programs with non-affine control flow or complex data structures. [Cohen et al., 1996] proposes a data flow analysis of imperative programs containing recursive control flow and data structures, however the approach is considered as too costly.

As [Kannan and Hamilton, 2018, Example 5] and the corresponding discussion show, such a parallelisation (as a form of scheduling) can be done by hand. This internship shows how to obtain such a parallelisation automatically. The paper proposes an automatic program transformation to parallel programs which introduces new data types, but according to [Kannan and Hamilton, 2018, Section 4.2] does not lead to noticeable speedups over the earlier hand-parallelised version. Thus, a benefit of our approach presented in Sections 6 and 7 is that the result is as readable and understandable as the hand-optimised version, while achieving similar performance to the (significantly less

readable) output of the transformation in [Kannan and Hamilton, 2018].<sup>2</sup>

Most previous work on program optimisation in the context of the polyhedral model<sup>3</sup> focuses exclusively on programs that perform affine accesses to arrays. Because trees and other inductive data structures lack this regularity, the framework does not apply to programs that use these structures. Therefore, few to no efficient scheduling techniques apply to these programs.

### 3 Term rewriting systems and termination

*Term Rewriting Systems* (TRS) are particularly apt to represent programs with complex data structures, such as lists or trees. As such, they provide a very suitable intermediate representation for parallelising such programs, in particular with recursion. In this section we introduce the basic ideas behind term rewriting systems and their termination. Proof techniques for termination proofs of TRSs give us first insights into what schedules for programs on inductive data structures may look like.

#### 3.1 Term rewriting systems as intermediate representation

From papers of the literature we borrow the following classic definitions (the research topic is older [Baader and Nipkow, 1998], however we may decide to cite more recent papers in which definitions have been stabilised).

**Definition 3 (Term, position [Baader and Nipkow, 1998])** *A signature  $\Sigma$  is a set of function symbols with associated arities.  $\mathcal{T}(\Sigma, \mathcal{V})$  denotes the set of terms over a finite signature  $\Sigma$  and the set of variables  $\mathcal{V}$ .*

*For a term  $t$ , the set  $\mathcal{Pos}(t)$  of its positions is defined inductively as a set of strings of positive integers:*

- *if  $t \in \mathcal{V}$ , then  $\mathcal{Pos}(t) = \{\varepsilon\}$ ;*
- *if  $t = f(t_1, \dots, t_n)$ , then  $\mathcal{Pos}(t) = \{\varepsilon\} \cup \bigcup_{1 \leq i \leq n} \{i\pi \mid \pi \in \mathcal{Pos}(t_i)\}$ .*

*The position  $\varepsilon$  is called the root position of term  $t$ . If  $t$  has the form  $f(t_1, \dots, t_n)$ , we write  $\text{root}(t) = f$  for the root of  $t$ .*

*The prefix order  $\leq$  on positions is the partial order given by:  $\pi \leq \tau$  iff. there exists  $\pi'$  such that  $\pi.\pi' = \tau$ .*

*For  $\pi \in \mathcal{Pos}(t)$ , we write  $t|_\pi$  for the subterm of  $t$  at position  $\pi$  and  $t[s]_\pi$  for the term that results from replacing the subterm at position  $\pi$  in  $t$  with the term  $s$ .*

<sup>2</sup>Since the original focus of the internship was on exploring the links between termination and scheduling, we did not discuss this paper in the initial bibliography.

<sup>3</sup>An algebraic framework that is capable of efficiently expressing and computing massive parallelism of computation kernels. The associated bibliographic report details this framework a bit more.



**Definition 4 (Term Rewriting System (TRS), innermost rewriting, defined symbols, constructor symbols [Baader and Nipkow, 1998])** A TRS  $\mathcal{R}$  is a set of rules  $\ell \rightarrow r$  where  $\ell$  and  $r$  are terms. To avoid non-termination,  $\ell$  must not be a variable and all variables that appear in  $r$  must also appear in  $\ell$ .

A rule  $\ell \rightarrow r$  can be applied to a term  $t$  if  $\ell$  matches a subterm  $u$  of  $t$  with some substitution  $\sigma$  (namely,  $u = \sigma(\ell)$ ). The rule is applied by replacing the subterm  $u$  with  $\sigma(r)$ , resulting in a new term  $v$  (a so-called rewrite step, denoted  $t \rightarrow_{\mathcal{R}} v$ ).

Here,  $\sigma$  is called the matcher and the term  $\ell\sigma$  is called the redex of the rewrite step. If  $\ell\sigma$  does not have a proper subterm that is also a possible redex,  $\ell\sigma$  is an innermost redex, and the rewrite step is an innermost rewrite step denoted by  $s \dot{\rightarrow}_{\mathcal{R}} t$ . A reduction is a sequence of rewrite steps. A TRS is terminating if all its reductions are finite.

$\Sigma_d^{\mathcal{R}} = \{f \mid \exists \ell \rightarrow r \in \mathcal{R}, \text{root}(\ell) = f\}$  and  $\Sigma_c^{\mathcal{R}} = \Sigma \setminus \Sigma_d^{\mathcal{R}}$  are the defined and constructor symbols of  $\mathcal{R}$ .<sup>4</sup> Finally, let  $\text{Pos}_d(t) = \{\pi \in \text{Pos}(t), \text{root}(t|_{\pi}) \in \Sigma_d\}$ .

Term rewriting systems are thus able to model programs of various paradigms and their termination:

**Example 5 (Tree size)** The link between recursive programs with pattern matching and TRS is immediate, for instance the following “Rust like” program operating on trees:

```
fn size(&self) -> int {
  match self {
    &Tree::Tree { v, ref l, ref r }
      => l.size() + r.size() + 1,
    &Tree::Nil => 0 , } }
```

can be modeled by the TRS  $\mathcal{R}$  given by the following rules:

$\text{plus}(\text{Zero}, y) \rightarrow y$	$\text{size}(\text{Nil}) \rightarrow \text{Zero}$
$\text{plus}(\text{S}(x), y) \rightarrow \text{S}(\text{plus}(x, y))$	$\text{size}(\text{Tree}(v, l, r)) \rightarrow \text{S}(\text{plus}(\text{size}(l), \text{size}(r)))$

We can easily see that we have

$$\begin{aligned} \text{size}(\text{Tree}(\text{Zero}, \text{Nil}, \text{Nil})) &\dot{\rightarrow}_{\mathcal{R}} \text{S}(\text{plus}(\text{size}(\text{Nil}), \text{size}(\text{Nil}))) \\ &\dot{\rightarrow}_{\mathcal{R}} \text{S}(\text{plus}(\text{Zero}, \text{size}(\text{Nil}))) \\ &\dot{\rightarrow}_{\mathcal{R}} \text{S}(\text{plus}(\text{Zero}, \text{Zero})) \\ &\dot{\rightarrow}_{\mathcal{R}} \text{S}(\text{Zero}) \end{aligned}$$

Morally, “the computation of  $\text{size}(\text{Tree}(0, \text{Nil}, \text{Nil}))$  terminates and returns 1”.

<sup>4</sup>We may omit the superscript and just write  $\Sigma_d$  and  $\Sigma_c$  if  $\mathcal{R}$  is not of importance or clear from the context.

Imperative programs can be represented via TRSs as well. One example are object-oriented programs with tree-shaped data structures where different node types (leaves vs inner nodes) are represented by different (sub)classes, such as in Java. Here the translation from Java Bytecode to TRSs by Otto et al. [Otto et al., 2010] introduces constructor symbols for different classes and uses pattern matching on the class of the object at hand to perform dynamic method dispatching via TRSs.

### 3.2 Proving termination with dependency pairs and polynomial interpretations

One of the most powerful termination methods is the *dependency pair* (DP) technique [Arts and Giesl, 2000], implemented in virtually all current termination tools for TRS. The intuition is to capture the dependencies between function calls, which can then be used to prove termination via the following reasoning: Every infinite reduction must have infinitely many function calls. So, if we can show that there cannot be infinitely many function calls in a reduction, we know that the TRS is terminating.

**Definition 5 (Dependency pair, from [Arts and Giesl, 2000])** For a TRS  $\mathcal{R}$ , the defined symbols are the root symbols of the left-hand sides of rules. For every defined symbol  $f$ , we extend the TRS with a fresh tuple symbol  $f^\sharp$  with the same arity as  $f$ . If  $t = f(t_1, \dots, t_n)$  and  $f$  is a defined symbol, we write  $t^\sharp$  for  $f^\sharp(t_1, \dots, t_n)$ . If  $l \rightarrow r \in \mathcal{R}$  and  $t$  is a subterm of  $r$  with defined root symbol, then the rule  $l^\sharp \rightarrow t^\sharp$  is a dependency pair of  $\mathcal{R}$ . The set of all dependency pairs of  $\mathcal{R}$  is denoted  $DP(\mathcal{R})$ .

**Example 6 (Dependency pairs for Ex. 5)** In our example, `size` and `plus` are defined symbols, and we have  $DP(\mathcal{R}) = \{(1), (2), (3), (4)\}$ :

$$\text{size}^\sharp(\text{Tree}(v, l, r)) \rightarrow \text{plus}^\sharp(\text{size}(l), \text{size}(r)) \quad (1)$$

$$\text{size}^\sharp(\text{Tree}(v, l, r)) \rightarrow \text{size}^\sharp(l) \quad (2)$$

$$\text{size}^\sharp(\text{Tree}(v, l, r)) \rightarrow \text{size}^\sharp(r) \quad (3)$$

$$\text{plus}^\sharp(x, \text{S}(y)) \rightarrow \text{plus}^\sharp(x, y) \quad (4)$$

Intuitively, dependency pairs define a “happens before” relation between function calls. This relation is equivalent to the one defined by dependency graphs, as introduced in Section 2.1.

To prove termination, we have to show that there cannot be infinitely many function calls in any reduction. More precisely, one has to prove that there is no infinite chain

$$\sigma_1(u_1) \rightarrow_{DP(\mathcal{R})} \sigma_1(v_1) \xrightarrow{\mathcal{R}}^* \sigma_2(u_2) \rightarrow_{DP(\mathcal{R})} \sigma_2(v_2) \xrightarrow{\mathcal{R}}^* \sigma_3(u_3) \rightarrow_{DP(\mathcal{R})} \sigma_3(v_3) \dots$$

where  $u_i \rightarrow v_i \in DP(\mathcal{R})$  and  $\sigma_i$  are substitutions. To this end, the DP method consists in finding two relations  $\succ$  and  $\lesssim$  such that:

$$\bigwedge_{u \rightarrow v \in DP(\mathcal{R})} u \succ v \wedge \bigwedge_{l \rightarrow r \in \mathcal{R}} l \lesssim r \quad (5)$$

A popular method to search for relations  $\succ$  and  $\succeq$  automatically are *polynomial interpretations* [Lankford, 1979]. A polynomial interpretation  $Pol$  maps each  $n$ -ary function symbol  $f$  to a polynomial  $f_{Pol}$  over  $n$  variables  $x_1, \dots, x_n$  with coefficients from  $\mathbb{N}$ . This mapping is extended to terms by defining  $[x]_{Pol} = x$  for all variables  $x$  and  $[f(t_1, \dots, t_n)]_{Pol} = f_{Pol}([t_1]_{Pol}, \dots, [t_n]_{Pol})$ .<sup>5</sup>

**Example 7** Consider  $Pol_1$  with  $Zero_{Pol_1} = 0$ ,  $Nil_{Pol_1} = 1$ ,  $size_{Pol_1}(x_1) = x_1$ ,  $S_{Pol_1}(x_1) = size_{Pol_1}^\#(x_1) = x_1 + 1$ ,  $plus_{Pol_1}(x_1, x_2) = plus_{Pol_1}^\#(x_1, x_2) = x_1 + x_2$ ,  $Tree_{Pol_1}(x_1, x_2, x_3) = x_2 + x_3 + 1$ . Then  $[size^\#(Tree(v, l, r))] = l + r + 2$  and  $[plus^\#(size(l), size(r))] = l + r$ .

Let  $\succ$  (resp.  $\succeq$ ) be the relation such that for all terms  $u$  and  $v$ ,  $u \succ v$  (resp.  $u \succeq v$ ) iff.  $[u] > [v]$  (resp.  $[u] \geq [v]$ ) holds for all instantiations of the variables with natural numbers. So with  $Pol_1$  we obtain  $size^\#(Tree(v, l, r)) \succ plus^\#(size(l), size(r))$ . In fact, all DPs (1)–(4) are strictly decreasing and the rules are at least weakly decreasing, i.e., the requirement (5) holds. Thus, termination of the TRS is proven.

More generally, polynomial interpretations are similar to ranking functions for proving termination of imperative programs (explained in Section 4), in that they expose a strictly decreasing yet bounded quantity, with the same variables as the considered terms. An algorithm to compute polynomial interpretations for TRS through *parametric* polynomial interpretations is depicted in [Fuhs et al., 2007].

### 3.3 Using term rewriting termination to prove full program termination

Progress in automated termination analysis for TRSs has given rise to several powerful and fully automatic tools over the last years (e.g., AProVE [Giesl et al., 2017], MuTerm [Alarcón et al., 2011], TTT2 [Korp et al., 2009]). Two-stage approaches to termination analysis, harnessing the power of termination tools for TRSs also for programming languages, have been proposed [Giesl et al., 2011, Otto et al., 2010, Giesl et al., 2012]. In the first stage, symbolic execution and abstraction on programming language level are used to over-approximate all possible program executions. From this program analysis, one then extracts a TRS whose termination implies the termination of the original program. In the second stage, termination of this TRS is analysed with existing tools. This approach is currently limited mainly by scalability of the program analysis front-ends.

More recently Fuhs et al. [Fuhs et al., 2009] proposed an extension of term rewriting by *built-in predefined integers*, based on the observation that termination, in particular for imperative programs, often depends on integer variables. This allows existing techniques for proving termination on integers to be combined with techniques for term rewriting. Tools for analysis of term rewriting are now successfully applied for ter-

<sup>5</sup>If the interpretation  $Pol$  is clear from the context, we also write  $[t]$  instead of  $[t]_{Pol}$ .

mination analysis (e.g., for Java [Otto et al., 2010, Brockschmidt et al., 2012] and C programs [Ströder et al., 2017]).

These techniques are implemented in AProVE [Giesl et al., 2017], a tool that automatically generates termination proofs for TRSs as well as imperative and functional programs, using built-in predefined integers to convert programs to TRSs and techniques such as dependency pairs to prove termination of TRSs.

## 4 Termination and scheduling

In this section we explore how termination and scheduling are linked; this opens doors to use and adapt termination/complexity bounds synthesis techniques to schedule programs.

### 4.1 Ranking functions for termination

Classically, imperative programs are proven to be terminating via the synthesis of a *ranking function*, whose concept comes from the seminal paper of Floyd [Floyd, 1967].

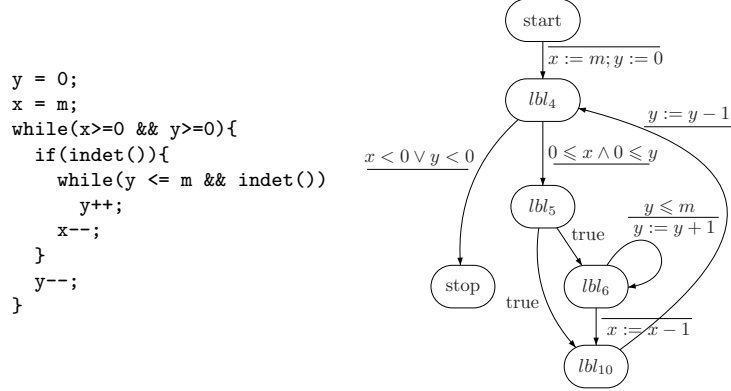
To model such programs, we can use a variant of control flow graphs (commonly used as intermediate representations inside compilers) known as *affine integer interpreted automata*.

Intuitively, a program state is a pair  $(k, \mathbf{x})$  where  $k \in \mathcal{K}$  is a *control point* and  $\mathbf{x} \in \mathbb{Z}^n$  is a *valuation* that assigns an integer value to each of the  $n$  variables. For any two control points  $k$  and  $k'$ , the possible transitions from  $k$  to  $k'$  are determined by a *guard*  $g$  (logical formula expressed with affine inequalities) and an *action*  $a$  (affine function from  $\mathbb{Z}^n$  to  $\mathbb{Z}^n$ ). Then for any valuation  $\mathbf{x}$  such that  $g(\mathbf{x}) = \text{true}$ , there is a transition from  $(k, \mathbf{x})$  to  $(k', a(\mathbf{x}))$ .

**Example 8** *Program and corresponding automaton (taken from [Alias et al., 2010])*  
Each control point of the automaton depicts a line in the program, and transitions of the form  $\frac{\text{condition}}{\text{action}}$  perform guarded transitions on the variables of the program.

Termination of a program may then be proven by finding a function that strictly decreases along every program transition, yet has a “lower bound” (thus preventing program transitions from occurring/happening indefinitely). Such a function is called a *ranking function*. We provide a formal definition to show the similarity with the definition of schedule constraints in Definition 2.

**Definition 6 (Ranking function for a CFG (from [Alias et al., 2010]))** *Let  $\mathcal{R} \subset \mathcal{K} \times \mathbb{Z}^n$  be the set of reachable states and for each control point  $k$ , let  $\mathcal{R}_k = \{\mathbf{x} \in \mathbb{Z}^n \mid (k, \mathbf{x}) \in \mathcal{R}\}$  be the set of the possible variable valuations for the control point  $k$ .*



A ranking function is a function  $\rho : \mathcal{K} \times \mathbb{Z}^n \rightarrow \mathcal{W}$  from the automaton states to a well-founded set  $(\mathcal{W}, \preceq)$ , whose values decrease at each transition  $(k, k', g, a)$ :

$$\mathbf{x} \in \mathcal{R}_k \wedge g(\mathbf{x}) = \text{true} \wedge \mathbf{x}' = a(\mathbf{x}) \implies \rho(k', \mathbf{x}') \prec \rho(k, \mathbf{x})$$

It is affine if it is affine in the second parameter (the variables).

In practice, it is sufficient to find such a ranking function for the head of loops, because potentially non-terminating imperative constructs are basically always loops. It is also convenient to compute in the well founded set  $(\mathbb{N}^d, \leq_d)$  (vectors of expressions with the lexicographic order).

**Example 9** Two-dimensional ranking function for the previous example A (two dimensional) ranking function for the program depicted in Example 8 is for instance defined by the following (affine) expressions:

<i>start</i>	$(2m + 4, -)$
<i>lbl<sub>4</sub></i>	$(2x + 3, 3y + 3)$
<i>lbl<sub>5</sub></i>	$(2x + 3, 3y + 2)$
<i>lbl<sub>6</sub></i>	$(2x + 2, m - y + 1)$
<i>lbl<sub>10</sub></i>	$(2x + 3, 3y + 1)$

For instance, the quantity  $2x + 3$  in *lbl<sub>4</sub>* is equal to  $2m + 3$  after the first step of the program execution; which is strictly less than  $2m + 4$  in control point *start*.

## 4.2 From scheduling to termination

From the crucial observation that a ranking function should be positive, strictly decreasing on the edges of the CFG, the proposition of the paper [Alias et al., 2010] is to adapt the Karp/Miller/Winograd algorithm (see Section 2.1) to affine interpreted automata.

It computes a multidimensional vector for each control point of the program. It has no syntactic restriction on the shape of the program (any affine transition function is ok), however it can fail to find a ranking function.

The result is an algorithm that tries to find a multidimensional ranking function, dimension by dimension. The algorithm relies strongly on the precision of the invariants precomputed beforehand. If the invariants were precise enough, then the algorithm would find the best (in terms of dimension) ranking function.

The  $n$ -dimensional ranking function  $\rho$  built by the algorithm has co-domain  $(\mathbb{N}^n, \preceq_n)$ , where  $\preceq_n$  is the lexicographic order on  $k$ -vectors. Each iteration determines a one-dimensional function  $\sigma$  which will become the next dimension of  $\rho$ . The function  $\sigma$  is determined through linear programming techniques, with the following constraints on each transition  $t = (k, g, a, k')$  that has not been satisfied by a previous dimension, where  $\mathcal{Q}_t$  is the dependency polyhedron associated with  $t$  and  $\mathcal{P}_k$  is the over-approximation of  $\mathcal{R}_k$  provided by invariants:

- $\forall x \in \mathcal{P}_k, \sigma(k, x) \geq 0$
- $\forall (x, x') \in \mathcal{Q}_t, \sigma(k, x) - \sigma(k', x') \geq \epsilon_t$ , with  $\epsilon_t \in \{0, 1\}$
- $\sigma$  maximises  $\sum_t \epsilon_t$ , i.e., satisfies as many transitions as possible

The algorithm ends when all transitions have been satisfied or when  $\sum_t \epsilon_t = 0$ , in which case no ranking function has been found.

### 4.3 From termination to scheduling

The preliminary paper [Alias et al., 2016] further explores this relationship between schedules and termination proofs and derives an estimation on the *parallel complexity* of the given programs. This paper can be viewed as the first attempt at using TRS termination to schedule program with trees.

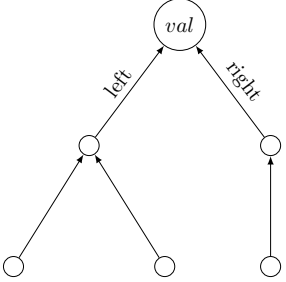
**Example 10 (Maximum element of a tree (taken from [Alias et al., 2016]))** *The following program implements the computation of the maximum elements of a tree. On the right, we depict its call graph that resembles a lot of a “dependency graph”.*

```

public class Tree {
  private int val;
  private Tree left;
  private Tree right;

  public int treeMax() {
    int leftMax = Integer.MIN_VALUE;
    int rightMax = Integer.MIN_VALUE;
    if (this.left != null) {
      leftMax = this.left.treeMax(); // S1
    }
    if (this.right != null) {
      rightMax = this.right.treeMax(); // S2
    }
    return Math.max(this.val, Math.max(leftMax, rightMax));
  }
}

```



**Example 11** (Example of [Alias et al., 2016], cont’) *The following dependency pairs can be derived from the example, and the polynomial interpretation (with max function) can be used to prove termination and provide a complexity bound:*

$$\begin{array}{ll}
 \text{Pol}(\text{dep}(x_1)) = x_1 & \text{dep}(\text{Tree}(\text{val}, \text{left}, \text{right})) \rightarrow \text{dep}(\text{left}) \\
 \text{Pol}(\text{Tree}(x_1, x_2, x_3)) = \max(x_2, x_3) + 1 & \text{dep}(\text{Tree}(\text{val}, \text{left}, \text{right})) \rightarrow \text{dep}(\text{right})
 \end{array}$$

*Pol* essentially maps a tree to its *height*, recursively defined as the maximum of its two children’s heights plus one. When interpreted as a bound on parallel complexity, this means that the two recursive calls on the left and right subtrees can be executed in parallel, resulting in a runtime<sup>6</sup> bounded by the height of the initial tree. In contrast, the overall runtime of the original sequential program is bounded only by the *size* of the input tree, which may be exponentially larger than its depth.

Thus the paper [Alias et al., 2016] suggests a novel view on the analysis of runtime (parallel) complexity [Hirokawa and Moser, 2008, Noschinski et al., 2013] for term rewriting. This paper can be thought of as the first attempt in exploring the “less trivial” usage of term rewriting definitions and results into the world of static compilation and scheduling.

#### 4.4 Synthesis

At that point, we are able to provide a synthesis of the previous bibliographic analysis in Figure 1.

<sup>6</sup>on a machine with unbounded parallelism

Imperative programs	Functional programs	Term rewriting systems
program		TRS + input term gen + output term parser
main	main	defsyms of the input term/input term gen + output term parser
statement	function	defined symbol
conditional branch (not always)	pattern matching branch	rule
(inductive) data structure definition		constructors
input data (structure)		input term without defined symbols
statement in an imperative control structure	recursive function call	defined symbol in a term
loop nest	nested function calls	term with nested defined symbols
?	function arguments	defined symbol “arguments”/immediate subterms
iteration vector ( $i, j$ )	?	position in the input structure (input term without defined symbols)
iteration domain of a statement	?	set of the <i>positions</i> of all subterms that appear at least once in the input term as arguments a defined symbol
array dimensions	?	input term <i>shape</i> or possibly its height if it is “regular enough”
?	recursive function call	defsymb that appears in the rhs of a rule (DP)
program termination		TRS termination
ranking function	ranking function?	polynomial interpretation
sequential/parallel complexity bounds		sequential/parallel TRS complexity bounds
<b>sequential/parallel execution</b>		<b>sequential/parallel rewriting</b>

Figure 1: Connections between scheduling and termination for TRS and different programming paradigms

One limitation of termination and complexity analysis for TRS in general is that the parametrisation is relatively coarse. In classic settings for scheduling such as the polyhedral model, the *shape* of the underlying data structure is completely defined by  $n$  parameters for the sizes of the array dimensions. Thus, one can completely describe



the schedule in terms of these parameters. In contrast, complexity analysis tools for term rewriting tend to provide complexity bounds (e.g., via polynomial interpretations) parametrised only in a single parameter, the *size* of the input data. However, there are many differently shaped trees of the same size, so the input parameter does *not* uniquely describe the shape of the data. Having an exact description of the shape of the input would be a prerequisite for deriving a schedule.

The approach by [Alias et al., 2016] alleviates the issue somewhat by suggesting the *height* of the input tree instead of its size as parameter for the analysis. However, also in that setting, many differently shaped trees have the same height, so also like this, we do not get an unambiguous schedule. Still, we might over-approximate and derive a schedule for, e.g., a *complete* binary tree for a given height and at runtime terminate computations early if the actual parameter tree has “missing branches”.

However, since going down this path did not seem like the most fruitful approach, we decided to consider other approaches for scheduling (and parallelising) TRS.

## 5 Parallel evaluation of term rewriting systems

Following the idea of Section 4.3, we propose to use parallel evaluation of TRS as a model of parallel execution of programs, and provide first algorithms to rewrite terms in parallel following/according to an innermost strategy.

Indeed, given a program and a TRS that is equivalent to this program, executing the program on some input data is equivalent to *evaluating* the corresponding input term, that is, rewriting it to a normal form by applying the rules of the TRS.

Furthermore, a rewriting strategy that rewrites *independent* terms in parallel translates to a schedule that exposes potential parallelism of the program.

Therefore, given a frontend that is able to produce an equivalent TRS from any program (along with “conversion” functions from input data to terms and from output terms to output data), any term rewriting engine that performs some rewrite steps in parallel functions as a parallel program interpreter. Figure 2 summarises this approach.

The algorithm(s) of this section describe such a parallel term rewriting engine.

### 5.1 Parallel innermost rewriting

The notion of parallel-innermost rewriting dates back at least to [Vuillemin, 1974]. Informally, in a parallel-innermost rewrite step, all innermost redexes are rewritten simultaneously. This is equivalent to executing all function calls in parallel on a machine with unbounded parallelism.

**Definition 7 (Parallel-innermost rewriting [Fernández et al., 2005])** *A term  $s$  rewrites innermost in parallel to  $t$  with a TRS  $\mathcal{R}$ , written  $s \Downarrow_{\mathcal{R}}^i t$ , iff  $s \xrightarrow{i}_{\mathcal{R}}^+ t$ , and*

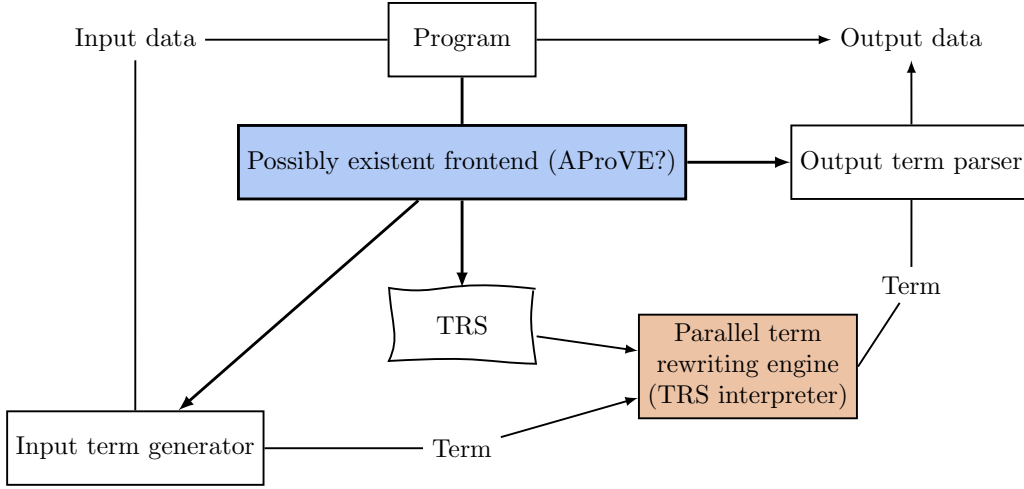


Figure 2: Both the TRS and the input term are processed at execution time.

either (a)  $s \xrightarrow{i} \mathcal{R} t$  with  $s$  an innermost redex, or (b)  $s = f(s_1, \dots, s_n)$ ,  $t = f(t_1, \dots, t_n)$ , and for all  $1 \leq k \leq n$  either  $s_k \xrightarrow{i} \mathcal{R} t_k$  or  $s_k = t_k$  is a normal form.

**Example 12 (size)** Consider again the TRS  $\mathcal{R}$  from Example 5. Here  $\Sigma_d^{\mathcal{R}} = \{\text{plus}, \text{size}\}$  and  $\Sigma_c^{\mathcal{R}} = \{\text{Zero}, \text{S}, \text{Nil}, \text{Tree}\}$ . We have the following parallel innermost rewrite sequence, where innermost redexes are underlined:

$$\begin{aligned}
 & \text{size}(\text{Tree}(\text{Zero}, \text{Nil}, \text{Tree}(\text{Zero}, \text{Nil}, \text{Nil}))) \\
 \xrightarrow{i} \mathcal{R} & \text{S}(\text{plus}(\text{size}(\text{Nil}), \text{size}(\text{Tree}(\text{Zero}, \text{Nil}, \text{Nil})))) \\
 \xrightarrow{i} \mathcal{R} & \text{S}(\text{plus}(\text{Zero}, \text{S}(\text{plus}(\text{size}(\text{Nil}), \text{size}(\text{Nil})))))) \\
 \xrightarrow{i} \mathcal{R} & \text{S}(\text{plus}(\text{Zero}, \text{S}(\text{plus}(\text{Zero}, \text{Zero})))) \\
 \xrightarrow{i} \mathcal{R} & \text{S}(\text{plus}(\text{Zero}, \text{S}(\text{Zero}))) \\
 \xrightarrow{i} \mathcal{R} & \text{S}(\text{S}(\text{Zero}))
 \end{aligned}$$

Note that in the second and in the third step, two innermost steps each are happening in parallel. An equivalent regular innermost rewrite sequence without parallel evaluation of redexes would have needed two more steps.

## 5.2 Implementing parallel-innermost rewriting

In this section, we show how to implement a simple parallel-innermost rewriting engine.

**Computation of a single “parallel innermost” rewriting step** Let  $\mathcal{R}$  be a TRS and let  $t$  be a term that we wish to rewrite. Our first task is to find all innermost redexes, which we will rewrite simultaneously. Potential redexes must have a defined symbol at their root. Thus, they can occur only at positions in  $\text{Pos}_d(t)$ . Candidates

for innermost redexes are those subterms that occur at positions  $\pi \in \mathcal{P}os_d(t)$  that are *maximal* in  $\mathcal{P}os_d(t)$  (i.e., there is no  $\pi' \in \mathcal{P}os_d(t)$  with  $\pi < \pi'$  in the prefix order). However,  $t|_\pi$  might be a normal form (in a TRS, a function need not be defined for all possible arguments). We obtain the set of all positions of innermost redexes  $\mathcal{I}\mathcal{R}$  using [Algorithm 1](#).

[Algorithm 1](#) iteratively removes innermost normal forms from the set of innermost redex candidates  $\mathcal{I}\mathcal{R}$ . Because  $\mathcal{I}\mathcal{R}$  is initialised as  $\mathcal{P}os_d(t)$  and only positions of normal forms are removed from it during any iteration,  $\mathcal{I}\mathcal{R}$  always contains all (innermost or otherwise) redex positions. After exiting the while loop,  $\mathcal{I}\mathcal{R}$  only contains redex positions, and the set of all innermost redex positions is the set of all maximal positions in  $\mathcal{I}\mathcal{R}$  (a redex is innermost iff. it has no other redex as a strict subterm, iff. its position is not a strict prefix of any other redex position).

---

**Algorithm 1:** Computation of positions of innermost redexes

---

```

Function ComputePos( $\mathcal{R}$ ,  $t$ )
  Data:  $\mathcal{R}$  a set of Rewriting Rules
  Data:  $t$  a term
  Result: The set of all positions of innermost redexes
   $\mathcal{I}\mathcal{R} = \mathcal{P}os_d(t)$ 
   $changed = \mathbf{True}$ 
  while  $changed$  do
     $changed = \mathbf{False}$ 
    for  $\pi \in \mathcal{I}\mathcal{R}$  such that  $\pi$  is maximal in  $\mathcal{I}\mathcal{R}$  do
      if  $t|_\pi$  does not match any rule in  $\mathcal{R}$  then // Normal Form
         $\mathcal{I}\mathcal{R} = \mathcal{I}\mathcal{R} \setminus \{\pi\}$ 
         $changed = \mathbf{True}$ 
  // Delete the non-innermost positions
   $\mathcal{I}\mathcal{R} = \{\pi \mid \pi \text{ is maximal in } \mathcal{I}\mathcal{R}\}$ 
  return  $\mathcal{I}\mathcal{R}$ 

```

---

**Example 13 (Ex. 5 & 12 continued)** For the term

$$t = S(\text{plus}(\text{size}(\text{Nil}), \text{size}(\text{Tree}(\text{Zero}, \text{Nil}, \text{Nil}))))$$

we have  $\mathcal{P}os_d(t) = \{1, 11, 12\}$ . From this set, [Algorithm 1](#) removes the position 1 since 1 is not an innermost position.

As, starting from the root, all positions need to be visited and checked if they contain a defined symbol, we immediately obtain:

**Proposition 1** For a given term  $t$ , the cost of computing  $\mathcal{P}os_d(t)$  is linear in the size  $|t|$  of  $t$ , that is, the number of positions in  $t$ .

Now that we have all the information, we can perform a parallel innermost rewrite step, thanks to [Algorithm 2](#), which performs the following reduction involving all innermost redex positions  $\pi_i$  precomputed in  $\mathcal{IR}$ :

$$t = t[\ell_1\sigma_1]_{\pi_1} \dots [\ell_n\sigma_n]_{\pi_n} \dashv\dashv^i_{\mathcal{R}} t[r_1\sigma_1]_{\pi_1} \dots [r_n\sigma_n]_{\pi_n} = t'$$

Here  $\ell_i \rightarrow r_i$  is the rule used to rewrite the innermost redex  $t|_{\pi_i}$  with matcher  $\sigma_i$ .

**Remark 2** From now on, we assume that the considered TRS is deterministic, i.e., exactly one rule can be used to rewrite a given innermost redex. This is true for TRS derived from programs consisting of pattern matching and function calls.

---

**Algorithm 2:** Computation of one step of parallel reduction

---

**Function** *ComputeOneParallelStep*( $\mathcal{R}$ ,  $t$ ,  $\mathcal{IR}$ )  
**Data:**  $\mathcal{R}$ ,  $t$   
**Data:**  $\mathcal{IR}$  the set of relevant positions  
**Result:**  $t$  the term after one parallel innermost reduction  
**for**  $\pi_i \in \mathcal{IR}$  **in parallel do**  
    Let  $\ell_i \rightarrow r_i \in \mathcal{R}$  s.t.  $t|_{\pi_i} = \ell_i\sigma_i$   
    Change  $t|_{\pi_i}$  into  $r_i\sigma_i$   
**return**  $t$

---

**Example 14** The parallel-innermost rewrite step

$t = \text{S}(\text{plus}(\text{size}(\text{Nil}), \text{size}(\text{Tree}(\text{Zero}, \text{Nil}, \text{Nil})))) \dashv\dashv^i_{\mathcal{R}} \text{S}(\text{plus}(\text{Zero}, \text{S}(\text{plus}(\text{size}(\text{Nil}), \text{size}(\text{Nil}))))$   
from the previous example can be implemented with:

- $\mathcal{P}os_d(t) = \{1, 11, 12\}$ ,  $\mathcal{IR} = \{11, 12\}$ ,
- $t = t[\text{size}(\text{Nil})]_{11}[\text{size}(\text{Tree}(\text{Zero}, \text{Nil}, \text{Nil}))]_{12}$ ,
- $\ell_1 \rightarrow r_1 = \text{size}(\text{Nil}) \rightarrow \text{Zero}$ ,  $\sigma_1 = []$  (identity),
- $\ell_2 \rightarrow r_2 = \text{size}(\text{Tree}(v, l, r)) \rightarrow \text{S}(\text{plus}(\text{size}(l), \text{size}(r)))$ ,  $\sigma_2 = [v := \text{Zero}, l := \text{Nil}, r := \text{Nil}]$ .

---

**Algorithm 3:** Computation of all reductions from initial term

---

**Function**  $ComputeAllSteps(\mathcal{R}, t_0)$   
**Data:**  $\mathcal{R}, t_0$   
**Result:**  $t$  the (reduced) term after all rewriting steps  
 $t = t_0$   
 $\mathcal{IR} = ComputePos(t)$   
**while**  $\mathcal{IR} \neq \emptyset$  **do**  
   $t = ComputeOneParallelStep(\mathcal{IR}, t)$   
   $\mathcal{IR} = ComputePos(t)$   
**return**  $t$

---

**Computation of the whole parallel reduction of a term** From the previous algorithm we finally obtain the non-optimised algorithm depicted in [Algorithm 3](#) in which we recompute all positions of “relevant redexes” each time we perform a parallel innermost step.

However, the new positions to be reduced can be pre-computed from one step to the next, thanks to the following observations. Let  $\pi'$  be the position of a redex that appears in  $t' = ComputeOneParallelStep(\mathcal{IR}, t)$ .

- If there is  $\pi_i \in \mathcal{IR}$  such that  $\pi' \geq \pi_i$ , then the redex  $t'|_{\pi'}$  occurs in  $r_i\sigma_i$ . Because  $l_i\sigma_i$  was an innermost redex, no redex appears in its variables, i.e., no term assigned to a variable by  $\sigma_i$  contains any redex. Therefore, any redex in  $r_i\sigma_i$  was introduced by  $r_i$  and the defined symbol at  $t'|_{\pi'}$  also appears in  $r_i$  at some position  $\tau \in Pos_d(r_i)$ , with  $\pi' = \pi_i\tau$ . (Since it depends only on the considered TRS and rule,  $Pos_d(r_i)$  is known statically for each rule.)
- Otherwise, the (defined) symbol at this position was not affected by the parallel innermost rewrite step (and was already present in  $t$ ):  $\pi' \in Pos_d(t)$ . More precisely,  $t'|_{\pi'}$  was a non-innermost redex in  $t$ :  $\pi' \in \mathcal{NIR}$ .

### 5.3 An optimised version of the parallel evaluation

This idea is implemented in [Algorithm 4](#), which performs the whole parallel reduction of a term, and illustrated in [Example 15](#).

[Algorithm 4](#) rewrites each innermost redex in parallel to produce the rewritten term  $t'$  (which is possible because all innermost redexes are independent). It also computes the sets of innermost redexes  $\mathcal{IR}'$  and non-innermost redex candidates  $\mathcal{NIR}'$  of  $t'$  from the candidates in  $\mathcal{NIR}$  and the redexes introduced by the rewritten subterms.

For each innermost redex position  $\pi_i$ , the redex at this position is rewritten using the applicable rewrite rule. The rewritten subterm  $t'|_{\pi_i} = r_i\sigma_i$  may contain new redexes;

however (see previous paragraph), any new redex was introduced by a defined symbol in  $r_i$ . Therefore, it is sufficient to check the positions in  $\mathcal{Pos}_d(r_i)$  to find innermost redexes and non-innermost redex candidates in the new subterm (line 6).

Innermost redexes (resp. non-innermost redex candidates) of the new subterm are also innermost redexes (resp. non-innermost redex candidates) of the rewritten term  $t'$  and can therefore be moved from the set of redex candidates  $\mathcal{NIR}$  to the set of innermost redexes  $\mathcal{IR}'$  (line 11) (resp. non-innermost redex candidates  $\mathcal{NIR}'$  (line 12)).

If at least one redex was found in the rewritten subterm  $t'|_{\pi_i}$ , then no term whose position is a prefix of  $\pi_i$  can be an innermost redex: all prefixes of  $\pi_i$  in the set of redex candidates  $\mathcal{NIR}$  can be moved to the set of non-innermost redex candidates  $\mathcal{NIR}'$  (line 10).

Otherwise, the rewritten subterm is a normal form and the prefixes of  $\pi_i$  in  $\mathcal{NIR}$  may be positions of innermost redexes of  $t'$ . Because the term is not completely rewritten at this point, it is not possible yet to determine whether subterms at these positions are innermost redexes, non-innermost redex candidates or normal forms. These remaining redex candidates are examined after rewriting the whole term (line 14).

Compared with [Algorithm 3](#), this approach removes the need to compute and analyse the set  $\mathcal{Pos}_d(t)$  at every step. In particular, normal forms are only seen once before being removed from the set of redex candidates, whereas the previous approach re-processed them at every rewrite step.

**Example 15** *The first and second parallel-innermost rewrite steps from the previous example:*

$$\begin{aligned} t &= \text{S}(\text{plus}(\text{size}(\text{Nil}), \text{size}(\text{Tree}(\text{Zero}, \text{Nil}, \text{Nil})))) \\ \Downarrow_{\mathcal{R}}^i t' &= \text{S}(\text{plus}(\text{Zero}, \text{S}(\text{plus}(\text{size}(\text{Nil}), \text{size}(\text{Nil})))) \\ \Downarrow_{\mathcal{R}}^i t'' &= \text{S}(\text{plus}(\text{Zero}, \text{S}(\text{plus}(\text{Zero}, \text{Zero})))) \end{aligned}$$

can be implemented with:

- *Initialisation:*  $\text{FindRedexes}(\mathcal{R}, t, \mathcal{Pos}_d(t)); \mathcal{IR} = \{11, 12\}; \mathcal{NIR} = \{1\}$
- *First iteration/rewrite step:*
  - $t|_{11} = \text{size}(\text{Nil})$  rewrites to  $\text{Zero}$ , which is a normal form.
  - $t|_{12} = \text{size}(\text{Tree}(\text{Zero}, \text{Nil}, \text{Nil}))$  rewrites to  $\text{S}(\text{plus}(\text{size}(\text{Nil}), \text{size}(\text{Nil})))$ . Redexes may appear at the following positions in the rewritten term:

$$\mathcal{Pos}_d(\text{S}(\text{plus}(\text{size}(l), \text{size}(r)))) = \{1, 11, 12\}$$

The rewritten term contains both innermost redexes and non-innermost redex candidates:  $\mathcal{IR}_{12} = \{11, 12\}; \mathcal{NIR}_{12} = \{1\}$ , which are added to the global sets of innermost redexes and non-innermost candidates:  $\mathcal{IR}' = \{1211, 1212\}; \mathcal{NIR}' = \{121\}$ .

---

**Algorithm 4:** Computation of all reductions from initial term with incremental computation of  $\mathcal{IR}$

---

```

Function FindRedexes( $\mathcal{R}, t, \mathcal{C}$ )
  Data:  $\mathcal{R}$  a set of Rewriting Rules
  Data:  $t$  a term
  Data:  $\mathcal{C}$  a set of redex candidates
  Result:  $\mathcal{IR}$  the set of all positions of innermost redexes
  Result:  $\mathcal{NIR}$  the set of all positions of non-innermost redex candidates
1   $Rdx = \mathcal{C}$ 
2   $changed = \mathbf{True}$ 
3  while  $changed$  do
4     $changed = \mathbf{False}$ 
5    for  $\pi \in Rdx$  such that  $\pi$  is maximal in  $Rdx$  do
6      if  $t|_{\pi}$  does not match any rule in  $\mathcal{R}$  then // Normal Form
7         $Rdx = Rdx \setminus \{\pi\}$ 
8         $changed = \mathbf{True}$ 
9   $\mathcal{IR} = \{\pi \mid \pi \text{ is maximal in } Rdx\}$ 
10  $\mathcal{NIR} = Rdx \setminus \mathcal{IR}$ 
11 return  $\mathcal{IR}, \mathcal{NIR}$ 

Function ParallelInnermostReduction( $\mathcal{R}, t$ )
  Data:  $\mathcal{R}$ 
  Data:  $t$ 
  Result:  $t'$ 
1   $\mathcal{IR}, \mathcal{NIR} = \text{FindRedexes}(\mathcal{R}, t, \text{Pos}_d(t))$ 
2  while  $\mathcal{IR} \neq \emptyset$  do
3    for  $\pi_i \in \mathcal{IR}$  in parallel do
4      Let  $\ell_i \rightarrow r_i \in \mathcal{R}$  s.t.  $t|_{\pi_i} = \ell_i \sigma_i$ 
5      Change  $t|_{\pi_i}$  into  $r_i \sigma_i$ 
6       $\mathcal{IR}_i, \mathcal{NIR}_i = \text{FindRedexes}(\mathcal{R}, r_i \sigma_i, \text{Pos}_d(r_i))$ 
7       $\mathcal{IR}_i = \{\pi_i \tau \mid \tau \in \mathcal{IR}_i\}$ 
8       $\mathcal{NIR}_i = \{\pi_i \tau \mid \tau \in \mathcal{NIR}_i\}$ 
9      if  $\mathcal{IR}_i \neq \emptyset$  then
10       // Do not consider prefixes of new redexes
11       // as innermost redex candidates
12        $\mathcal{NIR}_i = \mathcal{NIR}_i \cup \{\pi \in \mathcal{NIR} \mid \pi \text{ is a prefix of } \pi_i\}$ 
13     // Merge redex candidates that were processed locally
14     // into the global sets
15      $\mathcal{IR}' = \bigcup_i \mathcal{IR}_i$ 
16      $\mathcal{NIR}' = \bigcup_i \mathcal{NIR}_i$ 
17      $\mathcal{NIR} = \mathcal{NIR} \setminus \mathcal{NIR}'$ 
18     // Scan previously non-innermost redex candidates
19     // for innermost redexes
20      $\mathcal{IR}, \mathcal{NIR} = \text{FindRedexes}(\mathcal{R}, t, \mathcal{NIR})$ 
21      $\mathcal{IR} = \mathcal{IR} \cup \mathcal{IR}'$ 
22      $\mathcal{NIR} = \mathcal{NIR} \cup \mathcal{NIR}'$ 
23 return  $t$ 

```

---

Any strict prefixes of the new redex positions are moved from innermost redex candidates to the set of non-innermost candidates:  $\mathcal{NIR} = \emptyset$ ;  $\mathcal{NIR}' = \{121, 1\}$ .

- Innermost redexes and redex candidates for the next step:  $\mathcal{IR} = \{1211, 1212\}$ ;  $\mathcal{NIR} = \{1, 121\}$ .

Rewritten term:  $t' = \text{S}(\text{plus}(\text{Zero}, \text{S}(\text{plus}(\text{size}(\text{Nil}), \text{size}(\text{Nil}))))$

- Second iteration/rewrite step:

- $t'|_{1211} = \text{size}(\text{Nil})$  rewrites to **Zero**, which is a normal form.
- $t'|_{1212} = \text{size}(\text{Nil})$  rewrites to **Zero**, which is a normal form.
- Rewritten term:  $t'' = \text{S}(\text{plus}(\text{Zero}, \text{S}(\text{plus}(\text{Zero}, \text{Zero}))))$ .
- Innermost redexes and redex candidates for the next step:  $\text{FindRedexes}(\mathcal{R}, t, \{1, 121\})$ ;  $\mathcal{IR} = \{121\}$ ;  $\mathcal{NIR} = \{1\}$ .

- ...

**Conclusion** In this section, we have provided a proper definition for the notion of parallel execution that was informally proposed in [Alias et al., 2016]. Based on the TRS literature, we were able to provide a definition of it as a rewriting strategy. We propose, as a first contribution, a parallel evaluator that mimics this rewriting strategy on a given input term. We also show that we can optimise this algorithm by propagating additional information (“redex candidates”) from one reduction step to another.

The next section aims to achieve the initial goal of statically scheduling programs through static generation of a parallel term evaluator.

## 6 A code generator for a parallel evaluator based on partial evaluation

The algorithm described in the previous section processes the input term at the same time as the TRS itself. It is similar to an interpreter, in that fixed parts of the input program are processed at execution time, rather than compiled away entirely.

In this section, we propose to perform as much of this TRS analysis as possible statically. The result is a “term interpreter”/parallel term rewriting engine (based on a kind of *partial evaluation*, which from a TRS generates a code that is able to process any input term. In Figure 3 we depict the whole process, which has strong similarities with “parser generation”).

The generated code, which is then compiled, performs an hybrid schedule during execution. In other words, some parallelism is exposed in the generated code, however the computation that is done at runtime is still not fully predicted at compile-time.



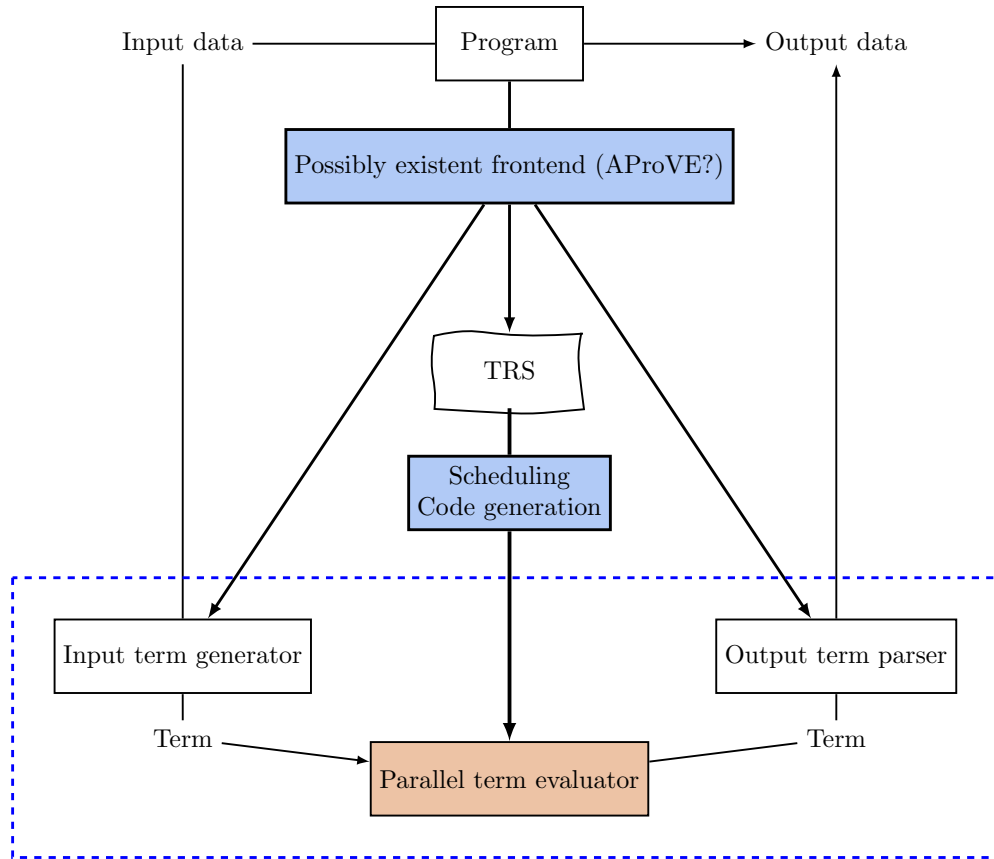


Figure 3: Flow of the approach. The blue dotted box encompasses the code that would be generated by the combination of an appropriate frontend with the proposed approach.

Concretely, the code that is generated by our engine is an OCaml interpreter using pattern-matching and parallel constructors of Multicore OCaml <sup>7</sup>.

In this section, we will depict the full process on Example 16 for which we will generate a parallel evaluator.

**Example 16 (Running example)** *We recall the computation of the size of a given tree as a TRS (cf Example 5):*

$$\begin{array}{l|l}
 \text{plus}(\text{Zero}, y) \rightarrow y & \text{size}(\text{Nil}) \rightarrow \text{Zero} \\
 \text{plus}(\text{S}(x), y) \rightarrow \text{S}(\text{plus}(x, y)) & \text{size}(\text{Tree}(v, l, r)) \rightarrow \text{S}(\text{plus}(\text{size}(l), \text{size}(r)))
 \end{array}$$

<sup>7</sup>An ocaml library and compiler for parallelism <https://github.com/ocaml-multicore/ocaml-multicore>

## 6.1 Parallel term evaluation through pattern matching

In this section we also assume a *deterministic* TRS: for any innermost-terminating term, there is a unique parallel-innermost rewriting sequence from this term to a normal form. We are looking to generate a program that *evaluates* its input term, that is, computes and returns the normal form obtained after applying this parallel-innermost rewriting sequence on the input term.

For any term, the next rewriting step (if any) depends on whether or not the term is *argument-normalised* (we will often just write *normalised*), that is, whether or not it is an innermost redex or a normal form. Therefore, the generated term evaluator features two distinct evaluation functions: the evaluation function for non-normalised terms recursively evaluates any nested redexes, while the one for normalised terms rewrites its input term according the applicable rule, if any. The input term is initially assumed to be a non-innermost redex: it is statically unknown and may contain several, possibly nested redexes.

**Example 17** *On Example 16,  $\text{size}(\text{Nil})$  is a normalised term and  $\text{plus}(\text{size}(\text{Nil}), \text{size}(\text{Nil}))$  is not.*

**Normalised terms** Rewriting an innermost redex can be seen as a form of pattern matching on terms where each pattern is the left-hand side of a rule, and the associated branch including the function calls resulting from the application of the rule that need to be considered first for the next (parallel-)innermost step is the right-hand side of the same rule.

The evaluation function for normalised terms follows this principle to select the rule (if any) to apply on the input term. Furthermore, unlike for the input term itself, each pattern matching branch is known statically, making it possible to define a general way to evaluate any term that matches this pattern.

More precisely, let  $\ell \rightarrow r$  be a rule. Let  $s = \ell\sigma$  be an innermost redex that matches  $\ell$  with some matcher  $\sigma$ . The term  $s$  rewrites innermost in parallel to  $t = r\sigma$  (first step of the evaluation of  $s$ ). The term  $t$  then needs to be fully rewritten to complete the evaluation of  $s$ . The positions of potential redexes of  $t$  are in  $\text{Pos}_d(r)$  (as explained in the previous section), which is known statically.

**Example 18** *The normalised term  $\text{size}(\text{Tree}(\text{Zero}, \text{Nil}, \text{Nil}))$  matches the left-hand side of the rule  $\text{size}(\text{Tree}(x, l, r)) \rightarrow \text{S}(\text{plus}(\text{size}(l), \text{size}(r)))$ . The positions of all potential redexes of any term that matches the right-hand side pattern are statically known. It rewrites innermost in parallel to  $\text{S}(\text{plus}(\text{size}(\text{Nil}), \text{size}(\text{Nil})))$ . From the statically known potential redex positions, we immediately know that the potential redexes of this new term are the underlined subterms.*

The next subsection further explains how to statically schedule the evaluation of any such term  $s$ .

**Non-normalised terms** In a similar way, the evaluation function for non-normalised terms (such as potentially the first term in a (parallel-)innermost reduction) uses pattern matching to select and apply statically scheduled/pre-computed branches. The root symbol of the input term is matched against every constructor and defined symbol of the TRS.

More precisely, let  $f$  be a symbol of arity  $n$ . Let  $t = f(t_1, \dots, t_n)$  be a non-normalised term whose root symbol is  $f$ . Each subterm  $t_i$  may contain redexes. The evaluation of  $t$  then consists of two steps:

1. evaluate each subterm  $t_i$  to a normal form  $t'_i$ . Since the  $t_i$  are *structurally independent*, they may be evaluated in parallel.
2. if  $s$  is a defined symbol, then the newly normalised term  $s(t'_1, \dots, t'_n)$  may match a rule and should be evaluated as an innermost redex. Otherwise,  $f$  is a constructor and  $f(t_1, \dots, t_n)$  is a normal form.

If  $n = 0$ , then the term  $t = f$  is in fact already normalised: it is either a constant (constructor) or an innermost redex (defined symbol).

**Example 19** *The symbol `plus` is a defined symbol of arity 2. Any non-normalised term with `plus` as its root symbol matches the following pattern: `plus(t1, t2)`, where  $t_1$  and  $t_2$  are “unknown”, potentially non-normalised terms.*

*By definition,  $t_1$  and  $t_2$  ultimately reduce to normal forms  $t'_1$  and  $t'_2$  after an unknown number of parallel-innermost rewrite steps. The terms  $t_1$  and  $t_2$  are structurally independent subterms of `plus(t1, t2)` and may be evaluated simultaneously. Therefore, `plus(t1, t2)` rewrites innermost in parallel to the normalised term `plus(t1, t2)`.*

*Since `plus` is a defined symbol, `plus(t1, t2)` may match the left-hand side of a rule: the next step in the evaluation of the initial term is to evaluate this potential innermost redex as described earlier.*

*With  $t_1 = t_2 = \text{size}(\text{Nil})$ ,  $t_1$  and  $t_2$  both rewrite to `Zero`. The resulting normalised term is `plus(Zero, Zero)`, which rewrites to `Zero`.*

## 6.2 Static local scheduling of rules

When generating evaluation code for terms that match a certain pattern, the pattern provides some information about the shape of the term that is generally statically unknown. This makes it possible to generate a “local” schedule for each pattern matching branch. This schedule includes

1. the set of the positions redexes that may appear in the rewritten term, and
2. the order in which these potential redexes should be evaluated.

Evaluating a term that matches a pattern then consists in recursively evaluating each of these redexes according to the schedule.

As explained in the previous section, the positions of the potential redexes of a normalised term that matches some pattern  $r$  are in  $\mathcal{Pos}_d(r)$ . The evaluation order of these potential redexes was already (informally) defined in the previous section: the innermost potential redexes should be evaluated first, then this procedure should be repeated until no more potential redexes are left and the term is normalised.

More formally, the prefix order on the positions of potential redexes also defines a partial order on their evaluation: for  $\pi, \pi' \in \mathcal{Pos}_d(r)$ ,  $r|_\pi$  must be evaluated before  $r|_{\pi'}$  iff.  $\pi'$  is a prefix of  $\pi$ . If neither position is a prefix of the other, then  $r|_\pi$  and  $r|_{\pi'}$  are *structurally independent* and may be evaluated in parallel.

It is then possible to assign a timestamp  $T(\pi)$  to each potential redex position  $\pi \in \mathcal{Pos}_d(r)$ <sup>8</sup>:

1. if  $r|_\pi$  is an innermost potential redex/ $\pi$  is maximal in  $\mathcal{Pos}_d(r)$ , then  $T(\pi) = 0$ ;
2. otherwise,  $T(\pi) = 1 + \max\{T(\pi.x) \mid \pi.x \in \mathcal{Pos}_d(r)\}$ .

### 6.3 Final code generation algorithm

The remainder of this section describes a code generation algorithm that generates a pattern matching-based term evaluator from a TRS.

The generated term evaluator consists of the two functions `evalNormalised` and `evalNonNormalised`, which perform pattern matching on their input term to evaluate normalised and non-normalised terms, respectively.

The right-hand sides of the pattern matching branches consist of the following pseudo-code expressions:

- term expressions that may include identifiers of newly computed values in addition to TRS symbols
- assignments of the form `id := expr`
- `evalNormalised(t)`, `evalNonNormalised(t)` denote the recursive evaluation of the normalised/non-normalised term expression `t`
- parallel expressions of the form `Par(e, e')` where the expressions `e` and `e'` may be computed simultaneously
- sequential expressions of the form `Seq(e, e')` where `e` must be computed before `e'`

The “local” schedules described earlier directly translate to these constructs, as shown in Algorithm 5.

**Example 20** *The full evaluator generated from the TRS of our running example is given below:*

---

<sup>8</sup>This resembles a lot the free schedule definition of [Karp et al., 1967] described in Section 2.1!

---

**Algorithm 5:** Scheduling and code generation
 

---

**Function** *GenNormalisedSchedule*( $\ell \rightarrow r$ )

**Data:**  $\ell \rightarrow r$  a rule

**Result:** the (pseudo-)code of the pattern matching/evaluation branch for normalised terms that match  $\ell$ 
 $s$  = empty expression;

**for**  $0 \leq i \leq \max |\pi|$ ,  $\pi \in \mathcal{Pos}_d(r)$  **do**
 $s_i$  = empty expression;

**for**  $\pi \in \mathcal{Pos}_d(r)$  s.t.  $|\pi| = i$  **do**
 $r' = r|_{\pi}$ ;

**for**  $k$  s.t.  $\pi.k \in \mathcal{Pos}_d(r)$  **do**
 $\lfloor$  Replace  $r'|_k$  with a fresh identifier  $x_{\pi.k}$  in  $r'$ ;

 $\lfloor$   $s_i = \text{Par}(s_i, x_{\pi} := \text{evalInnermost}(s))$ ;

 $\lfloor$   $s = \text{Seq}(s, s_i)$ ;

 $\lfloor$  **return**  $s$ 
**Function** *GenNonNormalisedSchedule*( $f, a, \text{isDefSym}$ )

**Data:**  $f$  a constructor or defined symbol of arity  $a$ 
**Data:**  $\text{isDefSym}$ , true iff.  $f$  is a defined symbol

**Result:** the (pseudo-)code of the pattern matching/evaluation branch for non-normalised terms whose root symbol is  $f$ 
 $x_1, \dots, x_a, y_1, \dots, y_a$  = fresh identifiers;

 $\text{lhs} = f(x_1, \dots, x_a)$ ;

 $\text{res} = f(y_1, \dots, y_a)$ ;

**if**  $\text{isDefSym}$  **then**
 $\lfloor$   $\text{res} = \text{evalNormalised}(\text{res})$ ;

**if**  $a = 0$  **then**
 $\lfloor$  // Constant or innermost redex;

 $\lfloor$  **return**  $\text{lhs} \rightarrow \text{res}$ ;

**else**
 $\lfloor$   $\text{rhs} = \text{empty expression}$ ;

 $\lfloor$  **for**  $1 \leq i \leq a$  **do**
 $\lfloor$   $\text{rhs} = \text{Par}(\text{rhs}, y_i := \text{evalNonNormalised}(x_i))$ ;

 $\lfloor$  **return**  $\text{lhs} \rightarrow \text{Seq}(\text{rhs}, \text{res})$ ;

**Function** *GenEvaluator*( $\mathcal{R}, \mathcal{C}, \mathcal{D}$ )

**Data:**  $\mathcal{R}$  a set of rules

**Data:**  $\mathcal{C}$  the set of all constructor symbols of the TRS

**Data:**  $\mathcal{D}$  the set of all defined symbols of the TRS

**Result:** The (pseudo-)code of a parallel term rewriting engine

 $\text{normEvalFun}$ ,  $\text{nonNormEvalFun}$  = empty pattern matching expressions;

**for**  $\ell \rightarrow r \in \mathcal{R}$  **do**
 $\lfloor$   $s = \text{GenNormalisedSchedule}(r)$ ;

 $\lfloor$  Add the pattern matching branch  $\ell \rightarrow s$  to  $\text{normEvalFun}$ ;

**for**  $f \in \mathcal{C}$  **do**
 $\lfloor$   $a = \text{arity of } f$ ;

 $\lfloor$  Add  $\text{GenNonNormalisedSchedule}(f, a, \text{false})$  to  $\text{nonNormEvalFun}$ ;

**for**  $f \in \mathcal{D}$  **do**
 $\lfloor$   $a = \text{arity of } f$ ;

 $\lfloor$  Add  $\text{GenNonNormalisedSchedule}(f, a, \text{true})$  to  $\text{nonNormEvalFun}$ ;

**return**  $\text{normEvalFun}$ ,  $\text{nonNormEvalFun}$ ;

---

```

let evalNormalised = function
| size(nil) -> Zero
| size(Tree(x, l, r)) ->
  (* size(l) and size(r) are structurally independent
  subterms of the rhs S(plus(size(l), size(r)))
  and are thus evaluated in parallel *)
  Seq(Par(l' := evalNormalised size(l),
          r' := evalNormalised size(r)),
      (* plus(size(l), size(r)) must be evaluated
      after its structural dependencies size(l) and size(r) *)
      s := evalNormalised plus(l', r'),
      S(s))
| plus(x, Zero) -> x
| plus(x, S(y)) -> evalNormalised plus(S(x), y)
| t -> t (* normal form *)

let evalNonNormalised = function
(* defined symbols *)
| size(t) ->
  Seq(t' := evalNonNormalised t, evalNormalised size(t'))
| plus(x, y) ->
  Seq(Par(x' := evalNonNormalised x,
          y' := evalNonNormalised y),
      evalNormalised plus(x', y'))
(* constructors *)
| Tree(x, l, r) ->
  Seq(Par(x' := evalNonNormalised x,
          l' := evalNonNormalised l,
          r' := evalNonNormalised r),
      Tree(x', l', r'))
| Nil -> Nil
| S(x) ->
  Seq(x' := evalNonNormalised x, S(x'))
| Zero -> Zero

```

**Example 21** *The input term  $\text{size}(\text{Tree}(\text{Zero}, \text{Nil}, \text{Nil}))$  goes through the following evaluation steps:*

$\text{evalNonNormalised}(\text{input\_term})$ : *the non-normalised term  $\text{size}(\text{Tree}(\text{Zero}, \text{Nil}, \text{Nil}))$  matches the pattern  $\text{size}(t)$  with  $t = \text{Tree}(\text{Zero}, \text{Nil}, \text{Nil})$ . Its evaluation consists of two sequential steps:*

1.  $t' := \text{evalNonNormalised}(t)$ : *the non-normalised term  $t = \text{Tree}(\text{Zero}, \text{Nil}, \text{Nil})$  matches the pattern  $\text{Tree}(x, l, r)$  with  $x = \text{Zero}$ ,  $l = \text{Nil}$  and  $r = \text{Nil}$ . Its evaluation consists of two sequential steps:*

(a) *parallel evaluation of the following non-normalised terms:*

$x = \text{Zero}$  *matches the pattern*  $\text{Zero}$ , *rewrites to*  $x' = \text{Zero}$

$l = \text{Nil}$  *matches the pattern*  $\text{Nil}$ , *rewrites to*  $l' = \text{Nil}$

$r = \text{Nil}$  *matches the pattern*  $\text{Nil}$ , *rewrites to*  $r' = \text{Nil}$

(b) *return*  $t' = \text{Tree}(x', l', r') = \text{Tree}(\text{Zero}, \text{Nil}, \text{Nil})$

2. `evalNormalised(size(t'))`: the normalised term `size(Tree(Zero, Nil, Nil))` matches the pattern `size(Tree(x, l, r))` with  $x = \text{Zero}$ ,  $l = \text{Nil}$  and  $r = \text{Nil}$ . Its evaluation consists of three sequential steps:

(a) parallel evaluation of the following normalised terms:

`size(l) = size(Nil)` matches the pattern `size(Nil)`, rewrites to  $l' = \text{Zero}$

`size(r) = size(Nil)` matches the pattern `size(Nil)`, rewrites to  $r' = \text{Zero}$

(b) `s := evalNormalised(plus(l', r'))`: the normalised term `plus(Zero, Zero)` matches the pattern `plus(x, Zero)` and rewrites to  $s = x = \text{Zero}$

(c) return the normal form  $S(s) = S(\text{Zero})$

## 7 Prototype and first experimental results

### 7.1 Prototype implementation

We implemented Algorithm 5 as a first prototype in OCaml based on a publicly available TRS parser<sup>9</sup>:

- The “assignments” of values to fresh identifiers, recursive calls to *eval(Non)Normalised* and sequential expressions directly translate to OCaml expressions.
- The *Par* expressions from the previous section, denoting the parallel evaluation of several independent expressions, were implemented as *task parallelism*: each expression is evaluated in a separate thread.

For parallel execution, we rely on the use of `Domainslib`<sup>10</sup>, a parallel programming library based on multicore OCaml<sup>11</sup>, an extension of the OCaml compiler that enables concurrency and parallelism. `Domainslib` provides *async* and *await* functions, used to respectively launch the evaluation of an expression in a separate thread and wait for the thread to finish in order to get the result of the computation (evaluated term).

The parallel evaluation functions (both normalised and non-normalised) are parametrised by a threshold on the depth of the parallel call stack: upon reaching this threshold, the sequential evaluation functions (in which each *Par* expression has been replaced with a *Seq* expression) are used instead in order to avoid stack overflow.

The input term generator (see Figure 3) is currently hand-written and generates a term consisting of a defined symbol (equivalent to the main function call of the initial program) applied to a normal form (which corresponds to an input data structure). It is parametrised by the height of the generated data structure.

<sup>9</sup><https://www.lri.fr/~marche/tpdb/format.html>

<sup>10</sup><https://github.com/ocaml-multicore/domainslib>

<sup>11</sup><https://github.com/ocaml-multicore/ocaml-multicore>

After generating an input term, the evaluator spawns a pool of  $n$  threads and launches parallel, then sequential evaluation of this term, while measuring both the total CPU time and real elapsed time (a.k.a. wallclock time) taken by both computations.

**Example 22** *The full generated code for `tree_size` can be found in Appendix A. However we provide here an excerpt to provide additional information on it and comparison to Example 20.*

```
let threshold = ref 5
open Domainslib.Task

(*[...]*)

let rec par_eval_norm pool depth t =
  let rec_eval_fun =
    if depth < !threshold then par_eval_norm pool (depth+1) else seq_eval_norm in
  let rec_eval_norm_fun = rec_eval_fun in
  match t with
  | Size(Nil) -> Zero
  | Size(Tree(v, l, r)) -> (* parallel eval of required terms *)
    let x28 = async pool (fun _ -> (rec_eval_norm_fun (Size(r))))
    in let x27 = async pool (fun _ -> (rec_eval_norm_fun (Size(l))))
    in (* combine after recovering x27,x28 results *)
      let x29 = (rec_eval_norm_fun (Plus(await pool x27, await pool x28)))
      in S(x29) (*result*)
  | Plus(x, Zero) -> x
  | Plus(x, S(y)) -> let x30 = (rec_eval_norm_fun (Plus(S(x), y)))
  in x30

(*[...]*)

;;
```

## 7.2 First results

**Experimental setting** We measured the speedup achieved by the parallel evaluation function over its sequential version, defined as  $\frac{t_{\text{seq}}}{t_{\text{par}}}$ .

For each TRS, we compute the speedups obtained for input terms with different sizes. We report the average speedup over five evaluations after removing the highest and the lowest speedup values. The evaluations were run on an Intel i5-2415M CPU clocked at 2.3GHz with four cores (two physical cores with two threads each); the size of the thread pool was therefore set to  $n = 4$ .

For this evaluation we have written the input term generators by hand, because we do not have any input front-end at the moment. Similarly, the output terms are post-processed by hand.

**Preliminary results** We tested our implementation on our running example (size of a binary tree) and four other simple examples such as:

- maximal element of a binary tree;



- element lookup in a binary tree;
- quicksort of a list;
- identity map on a list.

Some of these examples feature *conditional* rules, such as  $\text{if}(\text{True}, t, f) \rightarrow t$ , which are treated similarly to non-conditional rules: we propagate the tests “as-is” in the pattern matching guards.

The current parallel evaluation strategy may lead to the undesired, potentially time-expensive evaluation of the non-taken conditional branch. As future work, it may be possible to generate code that aborts the evaluation of non-taken conditional branches as soon as the condition is fully evaluated.

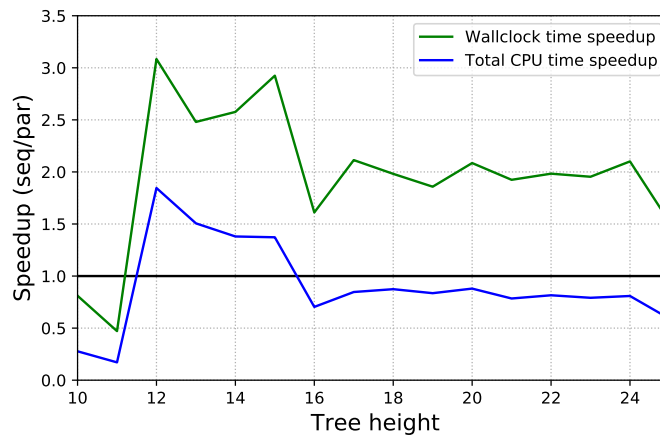


Figure 4: Experimental results for tree-size. speed-ups with respect to sequential evaluation, as a function of the size of the input tree.

From the generated files we were able to make the following first analyses:

- Initially, as expected, we get poor speedup (for instance, for tree size, in Figure 4), because of the intrinsic low parallelism of our terms. However these results show that we successfully captured this low amount of parallelism. Performance decreases for large trees, our hypothesis being that cache issues fuzzes a bit the performance of the OCaml runtime.
- The “optimal” threshold on the parallel call stack height seems highly dependent on the specific TRS and may be a function of the input term height (e.g. *idMap* TRS example).

## 8 Conclusion and future work

The initial goal of the internship was to use term rewriting systems as an intermediate representation to statically schedule imperative programs on inductive data structures.

Indeed, terms are particularly apt to represent inductive structures, and representing programs as TRS allows to use the numerous existing techniques to prove certain properties of TRS (such as termination or complexity bounds, which are closely linked to scheduling) for programs.

However, unlike arrays, which are the main focus of most existing scheduling algorithms, the shape of a term is not always fully determined by its size or height (whereas knowing the dimensions of an array allows to iterate over each of its elements).

As an alternative to a fully static parallel scheduling algorithm, the proposed approach can be seen as the partial evaluation of a parallel term rewriting engine, where the aspects of a term's evaluation schedule that only depend on the considered TRS are computed statically, and the remaining components that depend on the shape of the input term are determined during execution.

This approach applies to TRS, independently of whether they were generated from an imperative or functional program.

Future work could include an experimental evaluation of the proposed approach on different TRS, and a comparison to existing scheduling approaches.

## A Full final generated code for tree size

Listing 1: 'tree size'

```

let threshold = ref 5
open Domainslib.Task

type term = Plus of term * term
| Size of term
| S of term
| Tree of term * term * term
| Zero
| Nil ;;

let rec seq_eval_norm t =
  match t with
  | Size(Nil) -> Zero
  | Size(Tree(v, l, r)) -> let x28 = (seq_eval_norm (Size(r)))
    in let x27 = (seq_eval_norm (Size(l)))
    in let x29 = (seq_eval_norm (Plus(x27, x28)))
    in S(x29)
  | Plus(x, Zero) -> x
  | Plus(x, S(y)) -> let x30 = (seq_eval_norm (Plus(S(x), y)))
    in x30
  | Plus(x31, x32) -> Plus(x31, x32)
  | Size(x33) -> Size(x33)
  | S(x34) -> S(x34)
  | Tree(x35, x36, x37) -> Tree(x35, x36, x37)
  | Zero -> Zero
  | Nil -> Nil
;;

let rec seq_eval t =
  match t with
  | Size(Nil) -> let x1 = (seq_eval_norm (Size(Nil)))
    in x1
  | Size(Tree(v, l, r)) -> let x4 = (seq_eval (r))
    in let x3 = (seq_eval (l))
    in let x2 = (seq_eval (v))
    in let x5 = (seq_eval_norm (Size(Tree(x2, x3, x4))))
    in x5
  | Plus(x, Zero) -> let x6 = (seq_eval (x))
    in let x7 = (seq_eval_norm (Plus(x6, Zero)))
    in x7
  | Plus(x, S(y)) -> let x9 = (seq_eval (y))
    in let x8 = (seq_eval (x))
    in let x10 = (seq_eval_norm (Plus(x8, S(x9))))
    in x10
  | Plus(x11, x12) -> let x14 = (seq_eval (x12))
    in let x13 = (seq_eval (x11))
    in let x15 = (seq_eval_norm (Plus(x13, x14)))
    in x15
  | Size(x16) -> let x17 = (seq_eval (x16))
    in let x18 = (seq_eval_norm (Size(x17)))
    in x18
  | S(x19) -> let x20 = (seq_eval (x19))
    in S(x20)

```

```

| Tree(x21, x22, x23) -> let x26 = (seq_eval (x23))
  in let x25 = (seq_eval (x22))
    in let x24 = (seq_eval (x21))
      in Tree(x24, x25, x26)
| Zero -> Zero
| Nil -> Nil
;;

let rec par_eval_norm pool depth t =
let rec_eval_fun =
  if depth < !threshold then par_eval_norm pool (depth+1) else seq_eval_norm in
let rec_eval_norm_fun = rec_eval_fun in
match t with
| Size(Nil) -> Zero
| Size(Tree(v, l, r)) -> let x28 = async pool (fun _ -> (rec_eval_norm_fun (Size(r))))
  in let x27 = async pool (fun _ -> (rec_eval_norm_fun (Size(l))))
    in let x29 = (rec_eval_norm_fun (Plus(await pool x27, await pool x28)))
      in S(x29)
| Plus(x, Zero) -> x
| Plus(x, S(y)) -> let x30 = (rec_eval_norm_fun (Plus(S(x), y)))
  in x30
| Plus(x31, x32) -> Plus(x31, x32)
| Size(x33) -> Size(x33)
| S(x34) -> S(x34)
| Tree(x35, x36, x37) -> Tree(x35, x36, x37)
| Zero -> Zero
| Nil -> Nil
;;

let rec par_eval pool depth t =
let rec_eval_fun =
  if depth < !threshold then par_eval pool (depth+1) else seq_eval
and rec_eval_norm_fun =
  if depth < !threshold then par_eval_norm pool (depth+1) else seq_eval_norm in
match t with
| Size(Nil) -> let x1 = (rec_eval_norm_fun (Size(Nil)))
  in x1
| Size(Tree(v, l, r)) -> let x4 = async pool (fun _ -> (rec_eval_fun (r)))
  in let x3 = async pool (fun _ -> (rec_eval_fun (l)))
    in let x2 = async pool (fun _ -> (rec_eval_fun (v)))
      in let x5 = (rec_eval_norm_fun (Size(Tree(await pool x2, await pool x3, await pool x4))))
        in x5
| Plus(x, Zero) -> let x6 = (rec_eval_fun (x))
  in let x7 = (rec_eval_norm_fun (Plus(x6, Zero)))
    in x7
| Plus(x, S(y)) -> let x9 = async pool (fun _ -> (rec_eval_fun (y)))
  in let x8 = async pool (fun _ -> (rec_eval_fun (x)))
    in let x10 = (rec_eval_norm_fun (Plus(await pool x8, S(await pool x9))))
      in x10
| Plus(x11, x12) -> let x14 = async pool (fun _ -> (rec_eval_fun (x12)))
  in let x13 = async pool (fun _ -> (rec_eval_fun (x11)))
    in let x15 = (rec_eval_norm_fun (Plus(await pool x13, await pool x14)))
      in x15
| Size(x16) -> let x17 = (rec_eval_fun (x16))
  in let x18 = (rec_eval_norm_fun (Size(x17)))
    in x18
| S(x19) -> let x20 = (rec_eval_fun (x19))
  in S(x20)

```

```

| Tree(x21, x22, x23) -> let x26 = async pool (fun _ -> (rec_eval_fun (x23)))
  in let x25 = async pool (fun _ -> (rec_eval_fun (x22)))
    in let x24 = async pool (fun _ -> (rec_eval_fun (x21)))
      in Tree(await pool x24, await pool x25, await pool x26)
| Zero -> Zero
| Nil -> Nil
;;

```

## References

- [Aho et al., 1986] Aho, A. V., Sethi, R., and Ullman, J. D. (1986). *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., USA.
- [Alarcón et al., 2011] Alarcón, B., Gutiérrez, R., Lucas, S., and Navarro-Marset, R. (2011). Proving termination properties with mu-term. In *Proc. AMAST '10*, pages 201–208.
- [Alias et al., 2010] Alias, C., Darte, A., Feautrier, P., and Gonnord, L. (2010). Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In *Proc. SAS '10*, pages 117–133.
- [Alias et al., 2016] Alias, C., Fuhs, C., and Gonnord, L. (2016). Estimation of Parallel Complexity with Rewriting Techniques. In *Proc. WST '16*, pages 2:1–2:5.
- [Arts and Giesl, 2000] Arts, T. and Giesl, J. (2000). Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178.
- [Baader and Nipkow, 1998] Baader, F. and Nipkow, T. (1998). *Term rewriting and all that*. Cambridge University Press.
- [Brockschmidt et al., 2012] Brockschmidt, M., Musiol, R., Otto, C., and Giesl, J. (2012). Automated termination proofs for Java programs with cyclic data. In *Proc. CAV '12*, pages 105–122.
- [Cohen et al., 1996] Cohen, A., Collard, J.-F., and Griehl, M. (1996). Data Flow Analysis of Recursive Structures. In *Workshop on Compilers for Parallel Computers*, Aachen, Germany.
- [Fernández et al., 2005] Fernández, M., Godoy, G., and Rubio, A. (2005). Orderings for innermost termination. In *Proc. RTA '05*, pages 17–31.
- [Floyd, 1967] Floyd, R. (1967). Assigning meaning to programs. In *Proc. MACS '67*, pages 19–32.
- [Fuhs et al., 2007] Fuhs, C., Giesl, J., Middeldorp, A., Schneider-Kamp, P., Thiemann, R., and Zankl, H. (2007). SAT solving for termination analysis with polynomial interpretations. In *Proc. SAT '07*, pages 340–354.

- [Fuhs et al., 2009] Fuhs, C., Giesl, J., Plücker, M., Schneider-Kamp, P., and Falke, S. (2009). Proving termination of integer term rewriting. In *Proc. RTA '09*, pages 32–47.
- [Giesl et al., 2017] Giesl, J., Aschermann, C., Brockschmidt, M., Emmes, F., Frohn, F., Fuhs, C., Hensel, J., Otto, C., Plücker, M., Schneider-Kamp, P., Ströder, T., Swiderski, S., and Thiemann, R. (2017). Analyzing program termination and complexity automatically with AProVE. *Journal of Automated Reasoning*, 58(1):3–31.
- [Giesl et al., 2011] Giesl, J., Raffelsieper, M., Schneider-Kamp, P., Swiderski, S., and Thiemann, R. (2011). Automated termination proofs for Haskell by term rewriting. *ACM Transactions on Programming Languages and Systems*, 33(2):7:1–7:39.
- [Giesl et al., 2012] Giesl, J., Ströder, T., Schneider-Kamp, P., Emmes, F., and Fuhs, C. (2012). Symbolic evaluation graphs and term rewriting: A general methodology for analyzing logic programs. In *Proc. PPDP '12*, pages 1–12.
- [Hirokawa and Moser, 2008] Hirokawa, N. and Moser, G. (2008). Automated complexity analysis based on the dependency pair method. In Armando, A., Baumgartner, P., and Dowek, G., editors, *Proc. IJCAR '08*, pages 364–379. Springer.
- [Kannan and Hamilton, 2018] Kannan, V. and Hamilton, G. W. (2018). Functional program transformation for parallelisation using skeletons. *International Journal of Parallel Programming*, 46(1):152–172.
- [Karp et al., 1967] Karp, R. M., Miller, R. E., and Winograd, S. (1967). The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14(3):563–590.
- [Korp et al., 2009] Korp, M., Sternagel, C., Zankl, H., and Middeldorp, A. (2009). Tyrolean Termination Tool 2. In *Proc. RTA '09*, pages 295–304.
- [Lankford, 1979] Lankford, D. S. (1979). On proving term rewriting systems are Noetherian. Technical Report MTP-3, Louisiana Technical University, Ruston, LA, USA.
- [Noschinski et al., 2013] Noschinski, L., Emmes, F., and Giesl, J. (2013). Analyzing innermost runtime complexity of term rewriting by dependency pairs. *Journal of Automated Reasoning*, 51(1):27–56.
- [Otto et al., 2010] Otto, C., Brockschmidt, M., von Essen, C., and Giesl, J. (2010). Automated termination analysis of Java Bytecode by term rewriting. In *Proc. RTA '10*, pages 259–276.
- [Ströder et al., 2017] Ströder, T., Giesl, J., Brockschmidt, M., Frohn, F., Fuhs, C., Hensel, J., Schneider-Kamp, P., and Aschermann, C. (2017). Automatically proving termination and memory safety for programs with pointer arithmetic. *Journal of Automated Reasoning*, 58(1):33–65.

[Vuillemin, 1974] Vuillemin, J. (1974). Correct and optimal implementations of recursion in a simple programming language. *Journal of Computer and System Sciences*, 9(3):332–354.