An Improved Algorithm for E-Generalization

Jochen Burghardt

An Improved Algorithm for E-Generalization

Jochen Burghardt

jochen.burghardt@alumni.tu-berlin.de

Sep 2017

Abstract

E-generalization computes common generalizations of given ground terms w.r.t. a given equational background theory E. In [Bur05], we had presented a computation approach based on standard regular tree grammar algorithms, and a PROLOG prototype implementation. In this report, we present algorithmic improvements, prove them correct and complete, and give some details of an efficiency-oriented implementation in `C` that allows us to handle problems larger by several orders of magnitude.

Key words: E-Anti-unification; Equational theory; Generalization

Contents

1 Introduction

E-generalization computes common abstractions of given objects, considering given background knowledge to obtain most suitable results.

More formally, assuming familiarity with term rewriting theory (e.g. [DJ90]), the problem of E-generalization consists in computing the set of common generalizations of given ground terms w.r.t. a given equational background theory E. In [Bur05, Def.1,2], we gave a formal definition of the notion of a *complete set of E-generalizations of given ground terms*, and presented [Thm.7] an approach to compute such a set by means of standard algorithms on regular tree grammars (e.g. [TW68,CDG+01]). Its most fruitful application turned out to be the synthesis of non-recursive function definitions from input-output examples: given some variables $x_1, \ldots, x_n$, some ground substitutions $\sigma_1, \ldots, \sigma_m$, and some ground terms $t_1, \ldots, t_m$, we construct a term $t[x_1, \ldots, x_n]$ such that

$$\begin{aligned} t[x_1\sigma_1, \ldots, x_n\sigma_1] &=_E t_1 \\ \vdots \qquad\qquad & \qquad \vdots \\ t[x_1\sigma_m, \ldots, x_n\sigma_m] &=_E t_m \end{aligned}$$

As a main example application of this approach, we could obtain construction law terms for sequences. For example, given the sequence $0, 1; 4, 9, 16$, we could find a term $t[v_p, v_2, v_1]$ such that the equations in Fig. 1 (right) hold, where E consisted of the usual equations defining $(+)$ and $(*)$ on natural numbers in 0-s notation (see Fig. 1 left), writing e.g. 9 for $s^9(0)$ for convenience. Each such term $t[v_p, v_2, v_1]$ computes each supplied sequence member 4, 9, and 16 correctly from its 1st and 2nd predecessor v_1 and v_2, respectively, and its position v_p in the sequence (counting from 0). The supplied values 4, 9, 16 are said to be *explained* by the term $t[v_p, v_2, v_1]$; in the sequence, we write a *semicolon* to separate an initial part that doesn't need to be explained from an adjacent part that does.

For the given example, we obtained e.g. the term $t[v_p, v_2, v_1] = v_p * v_p$; when the background theory also included $(-)$, we additionally obtained e.g. $t[v_p, v_2, v_1] = v_1 + 2 * v_p - 1$. When instantiated by the substitution $\{v_p \mapsto 5, v_2 \mapsto 9, v_1 \mapsto 16\}$, both terms yield 25 as sequence continuation, which is usually accepted as a "correct" sequence extrapolation. This way, E-generalization, or nonrecursive-function synthesis, can be used to solve a certain class of intelligence tests.

In order to avoid solutions like $t[v_p, v_2, v_1] = (v_p + 0 * v_2) * v_p$, we defined a notion of *term weight* and focused on terms of minimal weight for t. As we computed a regular tree grammar describing all possible solution terms in a compact form, we could use a version of Knuth's

$$\begin{aligned} x + 0 &= x \\ x + s(y) &= s(x + y) \\ x * 0 &= 0 \\ x * s(y) &= x * y + x \end{aligned}$$

Pos	0	1	2	3	4	5
Val	0	1	4	9	16	25

$$\begin{aligned} & \overline{v_p\ v_2\ v_1} \\ t[2,0,1] &=_E 4 \\ t[3,1,4] &=_E 9 \\ t[4,4,9] &=_E 16 \end{aligned}$$

Figure 1. Background theory (l), sequence example (m), sequence law synthesis (r)

algorithm from [Knu77] to actually enumerate particular law terms in order of increasing weight.

*

The current report is in some sense the continuation of [Bur05]. After giving some definitions in Sect. 2, it reports in Sect. 3 algorithmic improvements on the problem of computing E-generalizations. Based on them, an efficiency-oriented reimplementation in `C` allows us to handle problems larger by several orders of magnitude, compared to the PROLOG implementation described in [Bur05, Sect. 5.4]. Section 4 discusses some details of that implementation. Section 5 shows some run time statistics.

We used our implementation to investigate the intelligence test *IST'70-ZR* [Amt73]; a report on our findings is forthcoming.

2 Definitions

2.1 Terms

Definition 1 *(Signature)*
Let Σ be a finite set of function symbols, together with their arities. We abbreviate the set of all n-ary functions by Σ_n. Let $\mathcal{V}$ be a finite set of variables. □

Definition 2 *(Term)*
For $V \subseteq \mathcal{V}$, let $\mathcal{T}_V$ denote the set of terms over Σ with variables from V, defined in the usual way. In particular, $\mathcal{T}_{\{\}}$ denotes the set of ground terms. We use $(=)$ to denote the syntactic equality of terms.

Definition 3 *(Equality, normal form)*
Let E be a congruence relation on $\mathcal{T}_{\{\}}$, i.e. an equivalence relation that is compatible with each $f \in \Sigma$. We require that E is computable in the following sense: a computable function $\mathsf{nf} : \mathcal{T}_{\{\}} \longrightarrow \mathcal{T}_{\{\}}$ shall exist such that for all $t, t_1, t_2 \in \mathcal{T}_{\{\}}$ we have

(1) Idempotency: $\mathsf{nf}(t) = \mathsf{nf}(\mathsf{nf}(t))$,
(2) Decisiveness: $t_1 =_E t_2 \;\Leftrightarrow\; \mathsf{nf}(t_1) = \mathsf{nf}(t_2)$, *and*
(3) Representativeness: $t =_E \mathsf{nf}(t)$.

In property 2, syntactic term equality is used on the right-hand side of "$\Leftrightarrow$". Property 3 is redundant, it follows by applying 2 to 1. We call $\mathsf{nf}(t)$ the normal form *of t, and say that t* evaluates to *$\mathsf{nf}(t)$. Let $\mathsf{NF} = \{\mathsf{nf}(t) \mid t \in \mathcal{T}_{\{\}}\}$ be the set of all normal forms, also called* values. □

Most often, nf is given by a confluent and terminating term rewriting system (see e.g. [DJ90, p.245,267]); the latter is obtained in turn most often from a conservative extension of an initial algebra of ground constructor terms (see e.g. [Duf91, Sect.7.3.2, p.160], [Pad89]). As an example, let $\mathsf{NF} = \{s^i(0) \mid i \in \mathbb{N}\}$ and let nf be given by the usual rewriting rules defining addition and multiplication on that set, like those shown in Fig. 1 (left); e.g. $\mathsf{nf}(s^2(0) * s^3(0)) = s^6(0) = \mathsf{nf}(s(0) + s^5(0))$.

2.2 Weights

In this subsection, we give the necessary definitions and properties about term weights. We associate to each function symbol f a weight function $\overline{f}$ of the same arity (Def. 6) which operates on some well-ordered domain $\mathcal{W}$ (Def. 4). Based on these functions, we inductively define the weight of a term (Def. 7).

Definition 4 *(Weight domain, minimal weight)*
Let $\mathcal{W}$ be a set and $(<)$ an irreflexive total well-founded ordering on W, such that $\mathcal{W}$ has a maximal element ∞. We use $(\leqslant)$ to denote the reflexive closure of $(<)$. Since $(<)$ is well-founded, each non-empty subset S of $\mathcal{W}$ contains a minimal element $\min S \in S$; we additionally define $\min\{\} := \infty$. □

For our algorithm in Sect. 3, we need a maximal element ∞ as initial value in minimum-computation algorithms. For the algorithm's completeness, we additionally need the absence of limit ordinals (e.g. [Hal68, Sect.19]) below ∞ (see Lem. 15). Altogether, for Alg. 13 we have to chose $\mathcal{W}$ order-isomorphic to $\mathbb{N} \cup \{\infty\}$. However, the slightly more general setting from Def. 4 is more convenient in theoretical discussions and counter-examples. We need $(<)$ to be total and well-founded in any case, in order for a set of terms to contain a minimal-weight term.

Definition 5 *(Function properties wrt. order)*
A function $\overline{f} : \mathcal{W}^n \longrightarrow \mathcal{W}$ is called

- *increasing if $\bigwedge_{i=1}^n x_i \leqslant \overline{f}(x_1, \ldots, x_n)$,*
- *strictly increasing if $\bigwedge_{i=1}^n (x_i < \infty \Rightarrow x_i < \overline{f}(x_1, \ldots, x_n))$,*
- *monotonic if $(\bigwedge_{i=1}^n x_i \leqslant y_i) \Rightarrow \overline{f}(x_1, \ldots, x_n) \leqslant \overline{f}(y_1, \ldots, y_n)$,*
- *strictly monotonic if it is monotonic and $(\bigwedge_{i=1}^n x_i < \infty) \wedge (\bigwedge_{i=1}^n x_i \leqslant y_i) \wedge (\bigvee_{i=1}^n x_i < y_i) \Rightarrow \overline{f}(x_1, \ldots, x_n) < \overline{f}(y_1, \ldots, y_n)$,*
- *strict if $(\bigwedge_{i=1}^n w_i < \infty) \Rightarrow \overline{f}(w_1, \ldots, w_n) < \infty$, in particular, a 0-ary weight function $\overline{f}$ is strict if $\overline{f} < \infty$.* □

Knuth's algorithm — which will play an important role below — requires weight functions to be monotonic and increasing [Knu77, p.1l].

Definition 6 *(Weight function associated to a function symbol)*
Let $\mathcal{W}$ and $(<)$ as in Def. 4. Let a signature Σ be given. For each $n \in \mathbb{N}$ and $f \in \Sigma_n$, let a monotonic and strictly increasing weight function $\overline{f} : \mathcal{W}^n \longrightarrow \mathcal{W}$ be given; we call $\overline{f}$ the weight function associated with f. We assume that $\overline{f}(x_1, \ldots, x_n)$ can always be computed in time $\mathcal{O}(n)$. □

Definition 7 *(Term weight, term set weight)*
Given the weight functions, define the weight $\mathsf{wg}(t)$ of a ground term t inductively by

$$\mathsf{wg}(f(t_1, \ldots, t_n)) := \overline{f}(\mathsf{wg}(t_1), \ldots, \mathsf{wg}(t_n)).$$

For a set of ground terms $T \subseteq \mathcal{T}_{\{\}}$, define

$$\mathsf{wg}(T) := \min\{\mathsf{wg}(t) \mid t \in T\}$$

to be the minimal weight of all terms in T. Note that $\mathsf{wg}(T) \in \mathcal{W}$ is always well-defined, and for nonempty T we have $\mathsf{wg}(T) = \mathsf{wg}(t)$ for some $t \in T$, since $\mathcal{W}$ is well-ordered. □

Example 8 *(Weight function examples)*
The most familiar examples of weight measures are the size $\mathsf{sz}(t)$, and the height $\mathsf{hg}(t)$ of a term t, i.e. the total number of nodes, and the length of the longest path from the root to any leaf, respectively. If $\mathcal{W} = \mathbb{N} \cup \{\infty\}$ and $\overline{f}(x_1, \ldots, x_n) = 1 + x_1 + \ldots + x_n$ for each $f \in \Sigma_n$, we get $\mathsf{wg}(t) = \mathsf{sz}(t)$; the definitions $\overline{f}(x_1, \ldots, x_n) = 1 + \max\{x_1, \ldots, x_n\}$ for $f \in \Sigma_n$ yield $\mathsf{wg}(t) = \mathsf{hg}(t)$. □

Example 9 *(Weight function counter-examples)*
We give two counter-examples to show what our weight functions cannot achieve.

First, it would be desirable in some contexts to prefer terms with minimal sets of distinct variables. Choosing $\mathcal{W} = \wp(\mathcal{V})$, $\overline{v} = \{v\}$ *for* $v \in \mathcal{V}$, *and*[1] $\overline{f}(x_1, \ldots, x_n) = x_1 \cup \ldots \cup x_n$ *for* f *an* n*-ary function symbol (including constants), we obtain as* $\mathsf{wg}(t)$ *the set of distinct variables occurring in the term* t. *However, we cannot use* $\subsetneq$ *as irreflexive well-ordering on* $\wp(\mathcal{V})$ *in the sense of Def. 4, since it is not total. Even if in a generalized setting* $<$ *on* $\mathcal{W}$ *was allowed to be a partial ordering, and we were interested only in the* number *of distinct variables, Knuth's algorithm from [Knu77] to compute minimal terms cannot be generalized accordingly to this setting, unless* $P = NP$, *as shown in [Bur03, Sect.5, Lem.29].*

Similarly, it can be desirable to have as $\mathsf{wg}(t)$ *the number of distinct subterms of* t. *Following similar proof ideas as in [Bur03, Sect.5], it can be shown that this is impossible unless the monotonicity requirement is dropped for weight functions, and that Knuth's algorithm cannot be adapted to that setting, again unless* $P = NP$. □

The following property will be used in the completeness and correctness proof of Alg. 13 below.

Lemma 10 *(Strict weight-functions)*
If all weight functions are strict, then $\forall t \in \mathcal{T}_{\mathcal{V}} : \mathsf{wg}(t) < \infty$.

PROOF. Induction on the height of t. □

[1] relaxing $\overline{f}$'s increasingness from strict to nonstrict, for sake of the example

3 An improved algorithm to compute E-generalizations

In this section, we discuss some algorithmic improvements on the problem of computing E-generalizations. We first give the improved algorithm in Sect. 3.1, and prove its completeness and correctness in Sect. 3.2 (see Thm. 19). In Sect. 3.3, we relate it to the grammar-based algorithm from [Bur05, Sect. 3.1], indicating that the former is an improvement of the latter. Implementation details are discussed in Sect. 4.

3.1 The algorithm

If f is an n-ary function symbol, and $w, w_1, \ldots, w_n \in \mathcal{W}$ are weights such that $\overline{f}(w_1, \ldots, w_n) = w$, we call the $n+1$-tuple $\langle f, w_1, \ldots, w_n \rangle$ a weight decomposition list evaluating to w.

Our algorithm (Fig. 2 shows an example state) can be decomposed in two layers. The lower one (Alg. 11, see left and red part in Fig. 2) generates, in order of increasing evaluation weight, a sequence of all possible weight decomposition lists. The latter is fed into the higher layer (Alg. 12, see right and blue part in Fig. 2), where each decomposition list is used to generate a corresponding set of terms.

This pipeline architecture is similar to that of a common compiler front-end, where a scanner generates a sequence of tokens which is processed by a parser.

Algorithm 11 *(Weight decomposition list generation)*

Input:

- *a signature* Σ,
- *a computable weight function* $\overline{f}$*, for each function symbol* $f \in \Sigma$
- *a finite set* $V \subset \mathcal{V}$ *of variables to be used in terms*

Output:

- *a potentially infinite stream of weight decomposition lists, with their evaluated weights being a non-decreasing sequence*

Algorithm:

(1) Maintain a set F *of weight decomposition lists to be considered next.*
(2) Maintain a set $\mathsf{F}_{\mathsf{hist}}$ *of evaluating weights of all decomposition lists that have ever been drawn from* F.
(3) Initially, let $\mathsf{F} = \{\langle v \rangle \mid v \in V\} \cup \{\langle c \rangle \mid c \in \Sigma_0\}$ *be obtained from all variables and signature constants.*
(4) Initially, let $\mathsf{F}_{\mathsf{hist}} = \{\}$.
(5) While F *is non-empty, loop:*
  (a) Remove from F *a weight decomposition list* $\langle f, w_1, \ldots, w_n \rangle$ *evaluating to the least weight among all lists in* F*; let* w *denote that weight.*
  (b) Output $\langle f, w_1, \ldots, w_n \rangle$ *to the stream.*

(c) *Insert w into $\mathsf{F}_{\mathsf{hist}}$.*
(d) *If in step 5a the least evaluating weight in F had increased since the previous visit, enter each possible weight decomposition list into F that can be built from w and some weights from $\mathsf{F}_{\mathsf{hist}}$.*
More formally: for every (non-constant) function symbol f from the signature, enter into F each weight decomposition list $\langle f, x_1, \ldots, x_n \rangle$ such that $w \in \{x_1, \ldots, x_n\} \subseteq \mathsf{F}_{\mathsf{hist}}$. □

Algorithm 12 *(Term generation from decomposition lists)*

Input:

- *a pair of goal values $\langle g_1, g_2 \rangle \in \mathsf{NF} \times \mathsf{NF}$,*
- *ground substitutions σ_1, σ_2,*
- *a finite, or potentially infinite, stream of weight decomposition lists, ordered by ascending evaluation weight.*

Output (if the algorithm terminates):

- *a term t of minimal weight such that $t\sigma_i =_E g_i$ for $i = 1, 2$.*

Algorithm:

(1) *Maintain a partial mapping $\phi : \mathsf{NF} \times \mathsf{NF} \hookrightarrow \mathcal{T}_{\mathcal{V}}$ that yields the term of least weight considered so far (if any) for each value pair v_1, v_2 such that $\phi(v_1, v_2)\sigma_1 =_E v_1$ and $\phi(v_1, v_2)\sigma_2 =_E v_2$, if $\phi(v_1, v_2)$ is defined.*
(2) *Maintain a set of minimal terms generated so far. Terms will be added to it in order of increasing weight; so we can easily maintain a weight layer structure within it. More formally: for each weight $w \in \mathcal{W}$, let D_w be the set of all minimal terms generated so far that have weight w.*
(3) *Initially, let ϕ be the empty mapping.*
(4) *Initially, let $\mathsf{D}_w = \{\}$, for each $w \in \mathcal{W}$.*
(5) *While the input stream of weight decomposition lists is non-empty, loop:*
  (a) *Let $\langle f, w_1, \ldots, w_n \rangle$ be the next list from the stream, let w denote the weight it evaluates to.*
  (b) *For all $t_1 \in \mathsf{D}_{w_1}, \ldots, t_n \in \mathsf{D}_{w_n}$, loop:*
    (i) *Build the term $t = f(t_1, \ldots, t_n)$. This term has weight w by construction.*
    (ii) *Let $v_1 = \mathsf{nf}(t\sigma_1)$, and $v_2 = \mathsf{nf}(t\sigma_2)$.*
    (iii) *If $\phi(v_1, v_2)$ is undefined, then*
      (A) *Add $\langle v_1, v_2 \rangle \mapsto t$ to ϕ.*
      (B) *Add t to D_w.*
      (C) *If $\langle v_1, v_2 \rangle = \langle g_1, g_2 \rangle$, then stop with success:*
        t is a term of minimal weight such that $t\sigma_i =_E g_i$.
(6) *Stop with failure: no term t with $t\sigma_i =_E g_i$ exists.* □

Note that the loop in step 5 may continue forever, if the input stream is infinite but no solution exists. Next, we compose Alg. 11 and Alg. 12:

Algorithm 13 *(Value-pair cached term generation)*

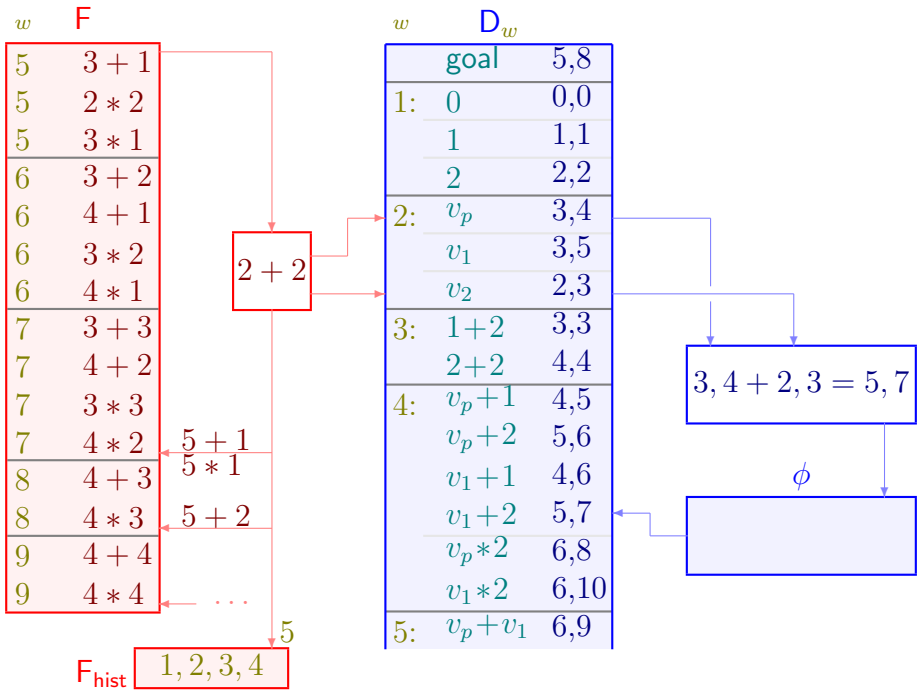

Figure 2. Example state in Alg. 11 (red) and 12 (blue)

Input:

- *a signature* Σ,
- *a computable weight function* $\overline{f}$, *for each function symbol* $f \in \Sigma$
- *a pair of goal values* $\langle g_1, g_2 \rangle \in \mathsf{NF} \times \mathsf{NF}$,
- *substitutions* σ_1, σ_2.

Output:

- *a term* t *of minimal weight such that* $t\sigma_i =_E g_i$ *for* $i = 1, 2$.

Algorithm:

- *Let* $V = \operatorname{dom}\sigma_1 \cup \operatorname{dom}\sigma_2$.
- *Feed* Σ, *all* $\overline{f}$, *and* V *into Alg. 11.*
- *Feed* $\langle g_1, g_2 \rangle$, σ_1, σ_2, *and Alg. 11's output stream into Alg. 12.*
- *Run Alg. 11 in parallel to Alg. 12 until either algorithm terminates.*

Implementation issues:

- *The set* F *in Alg. 11 is best realized by a heap (e.g. [AHU74, Sect. 3.4]).*
- *Since we have* $w \in \mathsf{F}_{\mathsf{hist}}$ *in Alg. 11 iff* $\mathsf{D}_w \neq \{\}$ *in Alg. 12, the former test can be implemented by the latter one, thus avoiding the need for an implementation of* $\mathsf{F}_{\mathsf{hist}}$.
- *The mapping* ϕ *in Alg. 12 is best implemented by a hash table (e.g. [AHU74, Sect. 4.2]) of balanced trees (e.g. [AHU74, Sect. 4.9]).*
- *The sets* D_w *in Alg. 12 are just segments of one global list of terms, ordered by non-decreasing weight* w.
- *The test* $\langle v_1, v_2 \rangle = \langle g_1, g_2 \rangle$ *in step 5b.iii.C of Alg. 12 can be speeded-up by initializing* $\phi(g_1, g_2)$ *to a special non-term entry "*goal*".* □

Example 14 *(Fibonacci sequence)*
Figure 2 shows an example state of Alg. 11 and 12, employed to obtain a law for the sequence $1, 2, 3; 5, 8$. *It assumes the* sz *weight from Exm. 8, slightly modified by assigning weight* 2, *rather than* 1, *to variable symbols.*

In the left part, in red, the Alg. 11 state is shown, with weight decomposition lists given in infix notation. The list $\langle +, 2, 2\rangle$ *was just drawn from* F. *It evaluates to weight* 5, *which occurs for the first time, so all lists buildable from* 5 *and some member of* $\mathsf{F}_{\mathsf{hist}}$ *are entered into* F, *of which* $\langle +, 5, 1\rangle$, $\langle *, 5, 1\rangle$, *and* $\langle +, 5, 2\rangle$ *are shown as examples.*

In the right part, in blue, the Alg. 12 state is shown. To the right of each term t *in some* D_w, *its evaluation value under* σ_1 *and* σ_2, *i.e.* $\mathsf{nf}(t\sigma_1), \mathsf{nf}(t\sigma_2)$ *is given; note that both values agree for ground terms. Alg. 12 is just building all terms corresponding to the input weight decomposition list* $\langle +, 2, 2\rangle$, *i.e. all sums of two generated minimal terms from* D_2. *After the term* $v_p + v_1$ *has been built and entered into* D_5, *the term* $v_p + v_2$ *is currently under consideration. It evaluates to the normal form* 5 *and* 7 *under* σ_1 *and* σ_2, *respectively. Looking up the pair* $\langle 5, 7\rangle$ *with* ϕ *reveals that there was already a term of less weight with these values, viz.* $v_1 + 2$; *so the term* $v_p + v_2$ *is discarded. Next, the term* $v_1 + v_2$ *will be built, evaluating to* $\langle 5, 8\rangle$; *lookup via* ϕ *will show that this is the goal pair, and the algorithm will terminate successfully, with* $v_1 + v_2$ *as a law term for the given sequence. We tacitly ignored commutative variants of buildable terms, such as* $v_1 + v_p$, $v_2 + v_p$, *and* $v_2 + v_1$; *see Sect. 4.3 below for a formal treatment.*

Figure 3 shows the detailed run of Alg. 11 in this example, up to the state shown in Fig. 2 (left). Figures 4 and 5 show the full run of Alg. 12, until the solution $v_1 + v_2$ *is found. For sake of brevity we again ignored commutative variants and some other trivial computations (marked "skipped" in the step field).* □

3	$\mathsf{F} = \{v_p, v_1, v_2, 0, 1, 2\}$
4	$\mathsf{F}_{\mathsf{hist}} = \{\}$
5	$\mathsf{F} = \{0, 1, 2, v_p, v_1, v_2\}$
5a	draw 0 from F, let $w = \overline{0} = 1$
5b	output 0 to Alg.12
5c	$\mathsf{F}_{\mathsf{hist}} = \{1\}$
5d	enter $1{+}1$, $1{*}1$ into F
5	$\mathsf{F} = \{1, 2, v_p, v_1, v_2, 1{+}1, 1{*}1\}$
5a	draw 1 from F, let $w = \overline{1} = 1$
5b	output 1
5	$\mathsf{F} = \{2, v_p, v_1, v_2, 1{+}1, 1{*}1\}$
5a	draw 2 from F, let $w = \overline{2} = 1$
5b	output 2
5	$\mathsf{F} = \{v_p, v_1, v_2, 1{+}1, 1{*}1\}$
5a	draw v_p from F, let $w = \overline{v_p} = 2$
5b	output v_p
5c	$\mathsf{F}_{\mathsf{hist}} = \{1, 2\}$
5d	enter $2{+}1, 2{*}1, 2{+}2, 2{*}2$ into F
5	$\mathsf{F} = \{v_1, v_2, 1{+}1, 1{*}1, 2{+}1, 2{*}1, 2{+}2, 2{*}2\}$
5a	draw v_1 from F, let $w = \overline{v_1} = 2$
5b	output v_1
5	$\mathsf{F} = \{v_2, 1{+}1, 1{*}1, 2{+}1, 2{*}1, 2{+}2, 2{*}2\}$
5a	draw v_2 from F, let $w = \overline{v_2} = 2$
5b	output v_2
5	$\mathsf{F} = \{1{+}1, 1{*}1, 2{+}1, 2{*}1, 2{+}2, 2{*}2\}$
5a	draw $1{+}1$ from F, let $w = 1 \overline{+} 1 = 3$
5b	output $1{+}1$
5c	$\mathsf{F}_{\mathsf{hist}} = \{1, 2, 3\}$
5d	enter $3{+}1, 3{*}1, 3{+}2, 3{*}2, 3{+}3, 3{*}3$ into F
5	$\mathsf{F} = \{1{*}1, 2{+}1, 2{*}1, 2{+}2, 3{+}1, 2{*}2, 3{*}1, 3{+}2, 3{*}2, 3{+}3, 3{*}3\}$
5a	draw $1{*}1$ from F, let $w = 1 \overline{*} 1 = 3$
5b	output $1{*}1$
5	$\mathsf{F} = \{2{+}1, 2{*}1, 2{+}2, 3{+}1, 2{*}2, 3{*}1, 3{+}2, 3{*}2, 3{+}3, 3{*}3\}$
5a	draw $2{+}1$ from F, let $w = 2 \overline{+} 1 = 4$
5b	output $2{+}1$
5c	$\mathsf{F}_{\mathsf{hist}} = \{1, 2, 3, 4\}$
5d	enter $4{+}1, 4{*}1, 4{+}2, 4{*}2, 4{+}3, 4{*}3, 4{+}4, 4{*}4$ into F
5	$\mathsf{F} = \{2{*}1, 2{+}2, \ldots, 3{+}2, 4{+}1, 3{*}2, 4{*}1, 3{+}3, 4{+}2, 3{*}3, 4{*}2, 4{+}3, 4{*}3, 4{+}4, 4{*}4\}$
5a	draw $2{*}1$ from F, let $w = 2 \overline{*} 1 = 4$
5b	output $2{*}1$
5	$\mathsf{F} = \{2{+}2, \ldots, 3{+}2, 4{+}1, 3{*}2, 4{*}1, 3{+}3, 4{+}2, 3{*}3, 4{*}2, 4{+}3, 4{*}3, 4{+}4, 4{*}4\}$
5a	draw $2{+}2$ from F, let $w = 2 \overline{+} 2 = 5$
5b	output $2{+}2$
	——state shown in Fig. 2 (left)——
5c	$\mathsf{F}_{\mathsf{hist}} = \{1, 2, 3, 4, 5\}$
5d	enter $5{+}1, 5{*}1, 5{+}2, 5{*}2, 5{+}3, 5{*}3, 5{+}4, 5{*}4, 5{+}5, 5{*}5$ into F

Figure 3. Run of Alg. 11 in Exm. 14

3	$\phi = \{\}$
4	$\mathsf{D}_1 = \mathsf{D}_2 = \ldots = \{\}$
5a	read 0 from Alg.11, let $w = \overline{0} = 1$
5bi,ii	build the term 0, it evaluates to 0, 0
5biiiA,B	add $\langle 0,0\rangle \mapsto 0$ to ϕ, add 0 to D_1
5a	read 1, let $w = \overline{1} = 1$
5bi,ii	build the term 1, it evaluates to 1, 1
5biiiA,B	add $\langle 1,1\rangle \mapsto 1$ to ϕ, add 1 to D_1
5a	read 2, let $w = \overline{2} = 1$
5bi,ii	build the term 2, it evaluates to 2, 2
5biiiA,B	add $\langle 2,2\rangle \mapsto 2$ to ϕ, add 2 to D_1
5a	read v_p, let $w = \overline{v_p} = 2$
5bi,ii	build the term v_p, it evaluates to 3, 4
5biiiA,B	add $\langle 3,4\rangle \mapsto v_p$ to ϕ, add v_p to D_2
5a	read v_1, let $w = \overline{v_1} = 2$
5bi,ii	build the term v_1, it evaluates to 3, 5
5biiiA,B	add $\langle 3,5\rangle \mapsto v_1$ to ϕ, add v_1 to D_2
5a	read v_2, let $w = \overline{v_2} = 2$
5bi,ii	build the term v_2, it evaluates to 2, 3
5biiiA,B	add $\langle 2,3\rangle \mapsto v_2$ to ϕ, add v_2 to D_2
5a	read 1+1, let $w = 1 \,\overline{+}\, 1 = 3$
5b	combine with "+" all terms from $\mathsf{D}_1 = \{0,1,2\}$ and D_1:
5bi,ii	build the term 0+0, it evaluates to 0, 0
5biii	$\phi(0,0) = 0$ is already defined
5bi,ii	build the term 0+1, it evaluates to 1, 1
5biii	$\phi(1,1) = 1$ is already defined
5bi,ii	build the term 0+2, it evaluates to 2, 2
5biii	$\phi(2,2) = 2$ is already defined
5bi,ii	build the term 1+0, it evaluates to 1, 1
5biii	$\phi(1,1) = 1$ is already defined
5bi,ii	build the term 1+1, it evaluates to 2, 2
5biii	$\phi(2,2) = 2$ is already defined
5bi,ii	build the term 1+2, it evaluates to 3, 3
5biiiA,B	add $\langle 3,3\rangle \mapsto 1{+}2$ to ϕ, add 1+2 to D_3
5bi,ii	build the term 2+0, it evaluates to 2, 2
5biii	$\phi(2,2) = 2$ is already defined
5bi,ii	build the term 2+1, it evaluates to 3, 3
5biii	$\phi(3,3) = 1{+}2$ is already defined
5bi,ii	build the term 2+2, it evaluates to 4, 4
5biiiA,B	add $\langle 4,4\rangle \mapsto 2{+}2$ to ϕ, add 2+2 to D_3
5a	read 1∗1, let $w = 1 \,\overline{*}\, 1 = 3$
5bi	build the terms $0{*}0, 0{*}1, 0{*}2, 1{*}0, 1{*}1, 1{*}2, 2{*}0, 2{*}1, 2{*}2$
skipped	none of them evaluate to a new value vector, so no change results

Figure 4. Run of Alg. 12 in Exm. 14 (part 1)

5a	read $2{+}1$, let $w = 2 \overline{+} 1 = 4$
5b	combine with "+" all terms from $\mathsf{D}_2 = \{v_p, v_1, v_2\}$ and $\mathsf{D}_1 = \{0, 1, 2\}$:
skipped	terms $v_p{+}0, v_1{+}0, v_2{+}0$ cause no change
5bi,ii	build the term $v_p{+}1$, it evaluates to 4, 5
5biiiA,B	add $\langle 4, 5\rangle \mapsto v_p{+}1$ to ϕ, add $v_p{+}1$ to D_4
5bi,ii	build the term $v_p{+}2$, it evaluates to 5, 6
5biiiA,B	add $\langle 5, 6\rangle \mapsto v_p{+}2$ to ϕ, add $v_p{+}2$ to D_4
5bi,ii	build the term $v_1{+}1$, it evaluates to 4, 6
5biiiA,B	add $\langle 4, 6\rangle \mapsto v_1{+}1$ to ϕ, add $v_1{+}1$ to D_4
5bi,ii	build the term $v_1{+}2$, it evaluates to 5, 7
5biiiA,B	add $\langle 5, 7\rangle \mapsto v_1{+}2$ to ϕ, add $v_1{+}2$ to D_4
5bi,ii	build the term $v_2{+}1$, it evaluates to 3, 4
5biii	$\phi(3, 4) = v_p$ is already defined
5bi,ii	build the term $v_2{+}2$, it evaluates to 4, 5
5biii	$\phi(4, 5) = v_p{+}1$ is already defined
5a	read $2{*}1$, let $w = 2 \overline{*} 1 = 4$
5b	combine with "$*$" all terms from $\mathsf{D}_2 = \{v_p, v_1, v_2\}$ and $\mathsf{D}_1 = \{0, 1, 2\}$:
skipped	terms $v_p{*}0, v_p{*}1, v_1{*}0, v_1{*}1, v_2{*}0, v_2{*}1$ cause no change
5bi,ii	build the term $v_p{*}2$, it evaluates to $6, 8$
5biiiA,B	add $\langle 6, 8\rangle \mapsto v_p{*}2$ to ϕ, add $v_p{*}2$ to D_4
5bi,ii	build the term $v_1{*}2$, it evaluates to $6, 10$
5biiiA,B	add $\langle 6, 10\rangle \mapsto v_1{*}2$ to ϕ, add $v_1{*}2$ to D_4
5bi,ii	build the term $v_2{*}2$, it evaluates to $4, 6$
5biii	$\phi(4, 6) = v_1{+}1$
5a	read $2{+}2$, let $w = 2 \overline{+} 2 = 5$
5b	combine with "+" all terms from $\mathsf{D}_2 = \{v_p, v_1, v_2\}$ and D_2:
skipped	terms $v_p{+}v_p, v_1{+}v_1, v_2{+}v_2$ cause no change
5bi,ii	build the term $v_p{+}v_1$, it evaluates to 6, 9
5biiiA,B	add $\langle 6, 9\rangle \mapsto v_p{+}v_1$ to ϕ, add $v_p{+}v_1$ to D_5
5bi,ii	build the term $v_p{+}v_2$, it evaluates to 5, 7
5biii	$\phi(5, 7) = v_1{+}2$ is already defined
	——state shown in Fig. 2 (right)——
5bi,ii	build the term $v_1{+}v_2$, it evaluates to 5, 8
5biiiA,B	add $\langle 5, 8\rangle \mapsto v_1{+}v_2$ to ϕ, add $v_1{+}v_2$ to D_5
5biiiC	found $\langle 5, 8\rangle$, so **stop** with success

Figure 5. Run of Alg. 12 in Exm. 14 (part 2)

Lemma 15 *(Properties of Alg. 11)*
Algorithm 11 has the following properties.

(1) No Weight decomposition list appears twice in the output of Alg. 11.
(2) The sequence of evaluating weights of output decomposition lists is non-decreasing.
(3) This sequence cannot get stationary infinitely long.
(4) Each weight decomposition list evaluating to a weight $< \infty$ that is inserted into F (in step 3 or 5d) will eventually be drawn out of it (in step 5a).

PROOF.

(1) Each weight decomposition list entered into F in step 5d contains a new weight, viz. w from step 5a, among its arguments.
(2) In step 5d each weight decomposition list inserted into F evaluates to a weight greater or equal to the current minimum, due to the monotonicity of weight functions.
(3) It suffices to show that there are only finitely many decomposition lists evaluating to the same weight. Given some $\langle f, x_1, \ldots, x_n\rangle$, let $x = \overline{f}(x_1, \ldots, x_n)$ be the weight it evaluates to. Since $(<)$ is well-founded, there are only finitely many weights $\leqslant x$. However, each decomposition list $\langle g, y_1, \ldots, y_n\rangle$ with $y_i > x$ for some i evaluates to a weight $> x$ since $\overline{g}$ is increasing. (Totality of $(<)$ is needed for this argument, but no strict increasingness of $\overline{g}$.)
(4) Assume the weight decomposition list $\langle f, w_1, \ldots, w_n\rangle$ has been inserted into F. We show that it will eventually be drawn from F. Consider again the the sequence of evaluating weights of output decomposition lists. If it is a finite sequence, F must eventually get empty, and hence $\langle f, w_1, \ldots, w_n\rangle$ must have been drawn from F. If it is an infinite one, it will eventually get larger than the result weight of $\langle f, w_1, \ldots, w_n\rangle$, by 2 and 3; before this can happen, $\langle f, w_1, \ldots, w_n\rangle$ must have been drawn from F, by 2. □

Lemma 16 *(Completeness of Alg. 11)*
Let V be the set of variables given to Alg. 11. Let $t \in \mathcal{T}_V$ be an arbitrary term in these variables, let $t = f(t_1, \ldots, t_n)$. Then the output stream of Alg. 11 will eventually contain the list $\langle f, \mathsf{wg}(t_1), \ldots, \mathsf{wg}(t_n)\rangle$; this list evaluates to weight $\mathsf{wg}(t)$. Observe that for $n = 0$ also constants and variables are admitted as t.

PROOF. By Lem. 15.4 it is sufficient to show that $\langle f, \mathsf{wg}(t_1), \ldots, \mathsf{wg}(t_n)\rangle$ is eventually entered into F. We do this by induction on t:

- If $n = 0$, i.e. if t is a constant or variable, it is entered into F in step 3 of Alg. 11.
- If $n > 0$, then by induction hypothesis (I.H.) some list evaluating to $\mathsf{wg}(t_i)$ will eventually be drawn from F, for $i = 1, \ldots, n$. When the last, i.e. the largest of them is drawn for the first time in step 5a, we have $\{\mathsf{wg}(t_1), \ldots, \mathsf{wg}(t_n)\} \subseteq \mathsf{F}_{\mathsf{hist}}$. Therefore, the weight decomposition list $\langle f, \mathsf{wg}(t_1), \ldots, \mathsf{wg}(t_n)\rangle$ is among those that are entered into F in step 5d. □

Lemma 17 *(Value-pair cache property)*
Let t be the term built in step 5b.i of Alg. 12. After completion of step 5b.iii, $\phi(\mathsf{nf}(t\sigma_1), \mathsf{nf}(t\sigma_2))$

is defined, and yields a term t' such that $t'\sigma_1 =_E t\sigma_1 \wedge t'\sigma_2 =_E t\sigma_2$ and $\mathsf{wg}(t') \leqslant \mathsf{wg}(t)$.

PROOF. In step 5b.ii, Alg. 12 computes $v_1 = \mathsf{nf}(t\sigma_1)$ and $v_2 = \mathsf{nf}(t\sigma_2)$. When $\phi(v_1, v_2)$ is yet undefined, it is set to t and there is nothing to show.

When $\phi(v_1, v_2)$ was already defined, say to be t', we have $\mathsf{wg}(t') \leqslant \mathsf{wg}(t)$, since the terms in step 5b.i are built in order of non-decreasing weight (a property of Alg. 11). Moreover, we have $t'\sigma_1 = \phi(v_1, v_2)\sigma_1 =_E v_1 =_E t\sigma_1$ using Def. 3, and similar for σ_2. □

Lemma 18 *In the setting of Alg. 13, let $V = \operatorname{dom}\sigma_1 \cup \operatorname{dom}\sigma_2$. If the test in step 5b.iii.C of Alg. 12 is omitted such that the loop of step 5 processes all weight decomposition lists, then the thus modified Alg. 13 has the following property: For each term $t \in \mathcal{T}_V$, a term t' with $t'\sigma_1 =_E t\sigma_1$ and $t'\sigma_2 =_E t\sigma_2$ is eventually entered into $\mathsf{D}_{\mathsf{wg}(t')}$ such that $\mathsf{wg}(t') \leqslant \mathsf{wg}(t)$.*

PROOF. Induction on t; let $t = f(t_1, \ldots, t_n)$, where n may also be zero. By I.H., the algorithm will eventually find terms t'_i such that $t'_i\sigma_1 =_E t_i\sigma_1 \wedge t'_i\sigma_2 =_E t_i\sigma_2$ and $\mathsf{wg}(t'_i) \leqslant \mathsf{wg}(t_i)$, and put them into $\mathsf{D}_{\mathsf{wg}(t'_i)}$, for $i = 1, \ldots, n$.

By Lem. 16, eventually a weight decomposition list $\langle f, \mathsf{wg}(t'_1), \ldots, \mathsf{wg}(t'_n)\rangle$ will appear in the stream in step 5a of Alg. 12. The algorithm will then generate $f(t''_1, \ldots, t''_n)$ for all $t''_i \in \mathsf{D}_{\mathsf{wg}(t'_i)}$, for $i = 1, \ldots, n$. In particular, it will build the term $f(t'_1, \ldots, t'_n)$ in step 5b.i.

By Lem. 17, after completion of algorithm step 5b.iii the entry $\phi(f(t'_1, \ldots, t'_n)\sigma_1, f(t'_1, \ldots, t'_n)\sigma_2)$ is defined, say to be t', with the property $t'\sigma_1 =_E f(t'_1, \ldots, t'_n)\sigma_1 =_E f(t_1, \ldots, t_n)\sigma_1 = t\sigma_1$. Similarly, we have $t'\sigma_2 =_E t\sigma_2$. Moreover, Lem. 17 yields that $\mathsf{wg}(t') \leqslant \mathsf{wg}(f(t'_1, \ldots, t'_n)) \leqslant \mathsf{wg}(f(t_1, \ldots, t_n)) = \mathsf{wg}(t)$, using $\overline{f}$'s monotonicity. □

Theorem 19 *(Correctness and completeness of Alg. 13)*
Let $g_1, g_2 \in \mathsf{NF}$ be the goal values given to Alg. 12. If some term t exists such that

$$t\sigma_1 =_E g_1 \wedge t\sigma_2 =_E g_2, \qquad (*)$$

then Alg. 12 will eventually stop successfully in step 5b.iii.C, with a result term that is of minimal weight with property $(*)$. Note that the result term needn't be t itself. If no such term t exists, the algorithm will fail in step 6, or will loop forever.

PROOF. W.l.o.g., let t be a term of minimal weight with property $(*)$. By Lem 18, we have that the modified (test step 5b.iii.C omitted) Alg. 12 will eventually generate a term t' with property $(*)$ such that $\mathsf{wg}(t') \leqslant \mathsf{wg}(t)$. Since t was minimal, we get $\mathsf{wg}(t') = \mathsf{wg}(t)$ also minimal.

Hence, in the unmodified algorithm, test step 5b.iii.C will apply to t', unless it applied to an earlier-generated term. In both cases, we are done, since the test ensures property $(*)$, and the first term passing the test is of minimal weight (again, a property of Alg. 11). Note that Lem. 18 and the existence of t ensures that Alg. 12 doesn't stop with failure in step 6.

If no term t exists with property $(*)$, the test in step 5b.iii.C cannot succeed. Hence, the only way to terminate the algorithm is in step 6. □

Example 20 *(Incompleteness for proper weight limit ordinal)*
If $\mathcal{W}$ and $(<)$ were such that a limit ordinal different from ∞ existed in $\mathcal{W}$, then Alg. 13 may be incomplete. Modifying Def. 4, let $\mathcal{W} = (\mathbb{N} \times \mathbb{N}) \cup \{\infty\}$, ordered by the lexicographic combination of the usual order on natural numbers, with each proper pair being less than ∞. Consider the signature $\Sigma = \{3, (+), (*)\}$. Define weight functions by

$$\begin{aligned}
\overline{3} &= \langle 0,\, 1\rangle,\\
\langle w_1, w_2\rangle \mathbin{\overline{+}} \langle w_3, w_4\rangle &= \langle 1 + w_1 + w_3,\, 1 + w_2 + w_4\rangle,\\
\langle w_1, w_2\rangle \mathbin{\overline{*}} \langle w_3, w_4\rangle &= \langle w_1 + w_3,\, 1 + w_2 + w_4\rangle, \quad \textit{and}\\
\infty \mathbin{\overline{+}} w = w \mathbin{\overline{+}} \infty = \infty \mathbin{\overline{*}} w = w \mathbin{\overline{*}} \infty &= \infty.
\end{aligned}$$

Intuitively, we have $\mathsf{wg}(t)$ being a pair consisting of the number of $(+)$-occurrences in t and the total number of symbols in t. All these weight functions are strictly monotonic, strictly increasing, and strict.

- *$\overline{*}$ is strictly monotonic:*
  - *Let $\langle w_1, w_2\rangle < \langle w_1', w_2'\rangle$.*
    If $w_1 < w_1'$, then $w_1 + w_3 < w_1' + w_3$.
    If $w_1 = w_1'$ and $w_2 < w_2'$, then $1 + w_2 + w_4 < 1 + w_2' + w_4$.
    From any case, we get $\langle w_1, w_2\rangle \overline{*} \langle w_3, w_4\rangle = \langle w_1 + w_3, 1 + w_2 + w_4\rangle < \langle w_1' + w_3, 1 + w_2' + w_4\rangle = \langle w_1', w_2'\rangle \overline{*} \langle w_3, w_4\rangle$.
  - *If $w_{12} < \infty$, then $w_{12} \overline{*} w_{34} \leqslant \infty = \infty \overline{*} w_{34}$.*
- *$\overline{*}$ is strictly increasing:*
  - *If $w_3 = 0$, then $w_2 < 1 + w_2 + w_4$.*
    If $w_3 > 0$, then $w_1 < w_1 + w_3$.
    From any case, we get
    $\langle w_1, w_2\rangle < \langle w_1 + w_3, 1 + w_2 + w_4\rangle = \langle w_1, w_2\rangle \overline{*} \langle w_3, w_4\rangle$.
  - *$\infty \leqslant \infty = \infty \overline{*} w_{34}$ is trivial.*
- *$\overline{+}$ similar.*

There is an infinite ascending chain of weights $\langle 0, 1\rangle < \ldots < \langle 0, 2n{+}1\rangle < \ldots < \langle 1, 3\rangle$, corresponding to a sequence of terms $3,\ 3{*}3,\ (3{*}3){*}3,\ \ldots,\ 3{+}3$. We don't need variables and substitutions in this example; if desired, one may think both σ_1 and σ_2 being the empty substitution. There is a term evaluating to 6, viz. $t = 3{+}3$, however all infinitely many $*$-terms of the chain are of less weight and will be generated before t. Since each of them denotes a power of 3, none may evaluate to 6. □

3.3 Relation to the grammar-based algorithm

In this section, we give an argument to show that Alg. 13 is more efficient than the *Constrained E-generalization algorithm* from [Bur05, Sect.3.1, p.6]. The latter algorithm used regular tree grammars to represent equivalence classes (mod. E) of values as well as sets of generalization terms. We will assume here that a more concise representation is possible, based on *grammar schemes* (as suggested in [Bur05, p.11]).

We will make certain unjustified, but optimistic, assumptions about grammar schemes, their

formal definability, and their implementability. We will show that even under these assumptions an appropriately improved Constrained E-generalization algorithm is not better than Alg. 13, at least for equational theories of realistic complexity.

A (nondeterministic) regular tree grammar [TW68,CDG$^+$01] is a triple $\mathcal{G} = \langle \Sigma, \mathcal{N}, \mathcal{R} \rangle$. Σ is a signature, $\mathcal{N}$ is a finite set of nonterminal symbols and $\mathcal{R}$ is a finite set of rules of the form $N ::= f_1(N_{11}, ..., N_{1n_1}) \mid \ldots \mid f_m(N_{m1}, ..., N_{mn_m})$ or, abbreviated, $N ::= \left|_{i=1}^{m}\right. f_i(N_{i1}, ..., N_{in_i})$. Given a grammar $\mathcal{G}$ and a nonterminal $N \in \mathcal{N}$, the language $\mathcal{L}_\mathcal{G}(N)$ produced by N is defined in the usual way as the set of all ground terms derivable from N as the start symbol. For our purposes, it is sufficient to define a *grammar scheme* somewhat informal to be an algorithm that generates a grammar from some input.

We will present our argument along the following E-generalization example. Consider the equational theory E, defining $(+)$, from Fig. 1 (left, topmost two equations). Its equivalence classes can be described by a grammar scheme $\mathcal{G}$ consisting of the rules

$$\begin{aligned} N_0 &::= \quad 0 \quad \mid \quad N_0 + N_0 \\ N_n &::= s(N_{n-1}) \mid \left|_{i=0}^{i}\right. N_i + N_{n-i} \qquad \text{for } n \geqslant 1 \end{aligned}$$

Due to the aggregation of the rules for $n \geqslant 1$, this is a grammar scheme rather than just a grammar. We have that $\mathcal{L}_\mathcal{G}(N_n) = [s^n(0)]$ is the equivalence class of all terms that evaluate to $s^n(0)$ wrt. $=_E$. Abbreviating again $s^n(0)$ by n for convenience, consider the example substitutions

$$\sigma_1 = \{\ v_1 \mapsto 3,\ v_2 \mapsto 2\ \} \text{ and}$$
$$\sigma_2 = \{\ v_1 \mapsto 5,\ v_2 \mapsto 3\ \}\ .$$

If in $\mathcal{G}$ we add

- to the rule of N_3 an alternative "$\ldots \mid v_1$",
- to the rule of N_2 an alternative "$\ldots \mid v_2$", and

we get a grammar scheme $\mathcal{G}_1$ such that $\mathcal{L}_{\mathcal{G}_1}(N_n)$ is the set of all terms t such that $t\sigma_1$ evaluates to n, by [CDG$^+$99, Theorem 7 in Sect. 1.4]. Similarly, if in $\mathcal{G}$ we add

- to the rule of N_5 an alternative "$\ldots \mid v_1$",
- to the rule of N_3 an alternative "$\ldots \mid v_2$", and

we get a grammar scheme $\mathcal{G}_2$ such that $\mathcal{L}_{\mathcal{G}_2}(N_n)$ is the set of terms evaluating to n under σ_2.

Next, we will compute a grammar scheme $\mathcal{G}_{1,2}$ containing, for each $n, n' \in \mathbb{N}$, a nonterminal $N_{n,n'}$ with $\mathcal{L}_{\mathcal{G}_{1,2}}(N_{n,n'}) = \mathcal{L}_{G_1}(N_n) \cap \mathcal{L}_{G_2}(N_{n'})$ being the set of all terms evaluating to n and n' under σ_1 and σ_2, respectively. For example, we will get $v_1 + v_2 \in \mathcal{L}_{\mathcal{G}_{1,2}}(N_{5,8})$, indicating that the term $v_1 + v_2$ is a common generalization of 5 and 8 w.r.t. $(=_E)$ and the given substitutions. This will allow us later on to establish that term as a construction law of the Fibonacci sequence, as explained in [Bur05, Sect. 5.2].

Following the usual product-automaton construction (e.g. [CDG$^+$99, Sect. 1.3]), we obtain a

grammar scheme $\mathcal{G}_{1,2}$:

$$
\begin{array}{lrcl}
N_{0,0} & ::= & 0 \mid & N_{0,0}+N_{0,0} \\
N_{0,n'} & ::= & \Big|_{i'=0}^{n'} & N_{0,i'}+N_{0,n'-i'} \qquad \text{for } n' \geqslant 1 \\
N_{n,0} & ::= & \Big|_{i=0}^{n} & N_{i,0}+N_{n-i,0} \qquad \text{for } n \geqslant 1 \\
N_{2,3} & ::= & s(N_{1,2}) \mid v_2 \mid \Big|_{i=0}^{2} \Big|_{i'=0}^{3} & N_{i,i'}+N_{2-i,3-i'} \\
N_{3,5} & ::= & s(N_{2,4}) \mid v_1 \mid \Big|_{i=0}^{3} \Big|_{i'=0}^{5} & N_{i,i'}+N_{3-i,5-i'} \\
N_{n,n'} & ::= & s(N_{n-1,n'-1}) \mid \Big|_{i=0}^{n} \Big|_{i'=0}^{n'} & N_{i,i'}+N_{n-i,n'-i'} \qquad \text{for all other } n,n'
\end{array}
$$

Since $N_{5,8} ::= \ldots N_{3,5} + N_{2,3} \ldots$, we will get in fact $v_1 + v_2 \in \mathcal{L}_{\mathcal{G}_{1,2}}(N_{5,8})$.

We use Knuth's algorithm [Knu77] to compute a term $t_{n,n'} \in \mathcal{L}_{\mathcal{G}_{1,2}}(N_{n,n'})$ of minimal weight. When applied to a grammar scheme like that of $\mathcal{G}_{1,2}$, and when terms are maintained along with their weights, it amounts to the following method:

Algorithm 21 *(Knuth's algorithm for grammar schemes)*
Input:

- *a regular tree grammar scheme $\mathcal{G}$, with the signature Σ being its set of terminal symbols, and its nonterminals having the form $N_{n,n'}$ with $n, n' \in \mathbb{N}$*
- *a computable weight function $\overline{f}$, for each function symbol $f \in \Sigma$*
- *a pair $\langle g_1, g_2 \rangle \in \mathbb{N} \times \mathbb{N}$ of goal values*

Output:

- *a partial mapping $\phi : \mathcal{N} \hookrightarrow \mathcal{T}_{\{\}}$ from nonterminals to weight-minimal terms of their generated language, such that $\phi(N_{n,n'})$, if defined, is a weight-minimal term in $\mathcal{L}_{\mathcal{G}}(N_{n,n'})$, and such that $\phi(N_{g_1,g_2})$ is defined if $\mathcal{L}_{\mathcal{G}}(N_{g_1,g_2})$ isn't empty*

Algorithm:

(1) Initially, let ϕ be the empty mapping.
(2) While $N_{g_1,g_2} \notin \operatorname{dom}\phi$, perform steps 2a to 2d:
  (a) For each grammar rule alternative $N_{n,n'} ::= \ldots \mid f(N_{i_1,i'_1}, \ldots, N_{i_k,i'_k})$, such that $N_{n,n'}$ is not in $\operatorname{dom}\phi$, but all $N_{i_1,i'_1}, \ldots, N_{i_k,i'_k}$ are, consider the term $f(\phi(N_{i_1,i'_1}), \ldots, \phi(N_{i_k,i'_k}))$.
  (b) Among these considered terms, choose some t of minimal weight.
  (c) Let $N_{n,n'} ::= \ldots$ be the grammar rule in step 2a where t originated from.
  (d) Add $\langle N_{n,n'}, t \rangle$ to ϕ. □

We make the optimistic assumptions that Alg. 21 can be implemented to handle even

- grammar schemes representing rules for infinitely many nonterminals (such as $N_{n,n'}$ for all $n, n' \in \mathbb{N}$ in our current example), and
- grammar scheme rules with infinitely many alternatives (such as $N_{0,0} ::= \Big|_{i=0}^{\infty} \Big|_{i'=0}^{\infty} N_{i,i'} - N_{i,i'}$ for an equational theory that includes subtraction).

However, we require that there are only finitely many function symbols, and each of them has

a fixed finite arity.

Next, if sufficiently many or sufficiently sophisticated operators are handled by the grammars, then for every combination j, j' a corresponding nonterminal $N_{j,j'}$ will appear in $\mathcal{G}_{1,2}$, as Exm. 22 demonstrates.

Example 22 *(Fully connected grammar)*
Let $\mathcal{G}$ be a tree grammar such that $\mathcal{L}_G(N_i) = [s^i(0)]_E$ wrt. some equational theory describing at least $(+)$ and $(-)$; assume $N_0, \ldots, N_n$ belong to $\mathcal{G}$. Then for each $0 \leqslant i, j \leqslant n$, we have a derivation

$$N_i \longrightarrow N_n - N_{n-i} \longrightarrow (N_j + N_{n-j}) - N_{n-i}.$$

Hence, such derivations also exist in the lifted grammars $\mathcal{G}_1$ and $\mathcal{G}_2$. In the intersection grammar $G_{1,2}$, we therefore have a derivation

$$N_{i,i'} \longrightarrow N_{n,n} - N_{n-i,n-i'} \longrightarrow (N_{j,j'} + N_{n-j,n-j'}) - N_{n-i,n-i'}$$

for each $0 \leqslant i, i', j, j' \leqslant n$. That is, each nonterminal $N_{j,j'}$ occurs in a derivation from a start symbol $N_{i,i'}$. □

In these cases, in order to find a minimal weight term for the goal nonterminal N_{g_1,g_2}, we have to consider $N_{j,j'}$ for every possible combination j, j', anyway. That is, considering only nonterminals that are reachable from N_{g_1,g_2} in $\mathcal{G}_{1,2}$ doesn't restrict the search space. Hence, we do not need to consider a grammar at all; we merely have to apply the given operations f and build term normal forms. More precisely, step 2a of Alg. 21 can be replaced by just building $f(\phi(N_{i_1,i'_1}), \ldots, \phi(N_{i_k,i'_k}))$ as soon as all $\phi(N_{i_1,i'_1}), \ldots, \phi(N_{i_k,i'_k})$ are defined, and considering it as a possibly minimal term for $N_{\mathsf{nf}(f(i_1,\ldots,i_k)),\mathsf{nf}(f(i'_1,\ldots,i'_k))}$. Since we don't need grammars or grammar schemes any longer, we don't have to bother about justifying our above optimistic assumptions about implementability of the latter.

Knuth original algorithm is just about minimal weights, not minimal weight terms; it uses a heap to find a minimal weight among the considered weights. When dealing with weight terms instead, collecting all terms corresponding to a *weight decomposition list* is an optimization. When we implement it, we arrive at Alg. 13. Thus, the latter has been obtained as an improvement of the Constrained E-generalization algorithm from [Bur05].

From an implementation point of view, the main advantage of Alg. 13 is that it doesn't need to store huge grammars in memory. Starting e.g. from a grammar $\mathcal{G}$ for the equivalence classes of $0, \ldots, 120$ w.r.t. just $+$ and $-$, the constrained E-generalization algorithm from [Bur05] arrives at a grammar $\mathcal{G}_{1,2}$ with $121^2 = 14641$ rules and a total of more than $\sum_{n,n'=0}^{120}(n+1)(n'+1) + (121-n)(121-n') = 108\,958\,322$ alternatives. $\mathcal{G}_{1,2,3}$ and $\mathcal{G}_{1,2,3,4}$ have already more than $8 \cdot 10^{11}$ and $5 \cdot 10^{15}$ alternatives, respectively. Figure 6 summarizes these relations. In this example, the alternative count is of magnitude of the nonterminal count's square. The Constrained E-generalization algorithm from [Bur05] needs memory proportional to the alternatives count, while both D and ϕ in Alg. 12 need memory proportional to the nonterminal count. The size of F in Alg. 11 is bounded by w^a where w is the number of distinct weights and a the maximal arity of operators; this bound can't get larger than the alternatives count, and usually is much smaller.

Tuple length	1	2	3	4
Nonterminals	121	14 641	1 771 561	214 358 881
Alternatives	14 762	108 958 322	804 221 374 682	5 935 957 966 527 842

Figure 6. Grammar size vs. tuple length for $0, \ldots, 120$ and $+,-$

The IST'70-ZR intelligence test could not be tackled with an algorithm that needs to store grammars corresponding to product automata, even with today's main memory technology. (It can be tackled, however, with the implementation of Alg. 13, a report is forthcoming.)

*

It is instructive to compare the best algorithm we could come up with, i.e. Alg. 13, with the most naive algorithm for the same task. The latter simply generates all terms in order of increasing weight, and stops as soon as $t\sigma_1 =_E g_1$ and $t\sigma_2 =_E g_2$ for the current term t. The only difference to Alg. 13 is that it doesn't cache the minimal weight terms for each value combination.

Arriving, after proceeding on a long and sophisticated path, that close to the starting point, is rather discouraging. It may indicate that there is no better way to solve the task, at least not on a Turing / Von Neuman architecture. A possible remedy might be to devise a highly parallel architecture to synthesize nonrecursive function definition terms from examples. This would comply with the neuronal structure of a human brain.

Example 23 *(Interleaved sequence)*
As a closing example for this section, consider the sequence $0, 1, 2, 1, 4, 1, 6, 1, 8, 1, \ldots$. *Its most obvious construction law term is* $if(v_p\%2 = 0, v_p, 1)$, *where* $if(x, y, z)$ *and* $x\%y$ *denotes a case distinction and the integer remainder, written in the* `C` *programming language as "*`(x?y:z)`*" and "*`x%y`*", respectively. The Lagrange interpolation term for this sequence is*

$$\frac{-1}{405}v_p^9 + \frac{31}{315}v_p^8 - \frac{1556}{945}v_p^7 + \frac{676}{45}v_p^6 - \frac{2192}{27}v_p^5 + \frac{11872}{45}v_p^4 - \frac{201536}{405}v_p^3 + \frac{17152}{35}v_p^2 - \frac{59077}{315}v_p,$$

it has the size 116 *even if each constant is considered to be only of size* 1. *Due to the huge number of value-tuples resulting from terms of smaller size, our implementation is unable to find the Lagrange term.*

However, if the set of available operators in law terms is varied, surprisingly unexpected law terms are found. Some of them are shown in Fig. 7, where a column below an operator shows its evaluation result on the sequence, and a column corresponding to a law term's root is shown in boldface. We denote by `u`, `f`, *and* `t` *an undefined value, the boolean falsity, and truth, respectively. We use* / *and* // *to denote the ordinary and the truncating integer division, for example,* $7/2$ *and* $7 // 2$ *yields* `u` *and* 3, *respectively.*

We could find neither a law term built from only $\%, v_p, v_1$ *nor one built from only* $+, v_p, v_1$*; see Fig. 8 for details.* □

$if(v_p\%2{=}0,v_p,1)$	$v_p-(v_p\%2)*(v_p-1)$	$v_p-(v_1//2)*2$	$2*(v_p-v_1)-v_2$	$v_p//v_1$
0 0 0 t 0	0 **0** 0 0 0 0 u			
1 1 1 f 1	1 **1** 1 1 0 1 0	1 **1** 0 0 0		
2 2 0 t 2	2 **2** 2 0 0 2 1	2 **2** 1 0 0	2 2 1 1 **2** 0	2 **2** 1
1 3 1 f 3	3 **1** 3 1 2 3 2	3 **1** 2 1 2	2 3 1 2 **1** 1	3 **1** 2
4 4 0 t 4	4 **4** 4 0 0 4 3	4 **4** 1 0 0	6 4 3 1 **4** 2	4 **4** 1
1 5 1 f 5	5 **1** 5 1 4 5 4	5 **1** 4 2 4	2 5 1 4 **1** 1	5 **1** 4
6 6 0 t 6	6 **6** 6 0 0 6 5	6 **6** 1 0 0	10 6 5 1 **6** 4	6 **6** 1
1 7 1 f 7	7 **1** 7 1 6 7 6	7 **1** 6 3 6	2 7 1 6 **1** 1	7 **1** 6
8 8 0 t 8	8 **8** 8 0 0 8 7	8 **8** 1 0 0	14 8 7 1 **8** 6	8 **8** 1
1 9 1 f 9	9 **1** 9 1 8 9 8	9 **1** 8 4 8	2 9 1 8 **1** 1	9 **1** 8

Figure 7. Law terms in Exm. 23

Optr set	Sz	Terms	Computations	Memory (bytes)	Time (sec)
$\%, v_p, v_1$	36	9688	51979260	134990875	11
$+, v_p, v_1$	726	243035	19408297875	159683896	3218

Figure 8. Failed attempts in Exm. 23

4 Implementation overview

In this section, we sketch some features of our `C` implementation. Section 4.1 gives a brief overview of all kernel modules, while Sect. 4.2 sketches the modules that hold user-definable code.

The subsequent sections elaborate on certain approaches to optimize the term-generation process. We discuss

- in Sect. 4.3 how to avoid building redundant commutative and associative variants of terms, like $(t_2 + t_1) + t_3$ when $t_1 + (t_2 + t_3)$ already was built;
- in Sect. 4.4 how to avoid building other redundant terms, like $t - t$;
- in Sect. 4.5 how to avoid building nonsensical terms, like $(t_1 \leqslant t_2) * t_3$;
- in Sect. 4.6 how to efficiently build terms involving $if(\cdot, \cdot, \cdot)$ and similar operators.
- in Sect. 4.7, how to implement the mapping ϕ from Alg. 12 efficiently.

4.1 Kernel modules overview

We give a short description of each module file of the `C`-implementation:

(1) Identifiers and syntax
   (a) Module `stringTab.c` — Administration of strings.
   (b) Module `idTab.c` — Administration of operator and variable identifiers. This module implements the scanner underlying the parser in module 1c.
   (c) Module `parser.c` — Parsing routines: `parseValues`, `parseOpDefs`, `parseRedices`, `parseVariables`, `parseGoals`
   (d) Module `wgfTab.c` — Administration of weight functions (like `size`, `height`); each of those functions is made are accessible by a name. The actual code of weight functions is contained in module `UserWgf.c` (module 6 in Sect. 4.2).

(2) Values, sorts, and redices
   (a) Module `valDefTab.c` — Administration of normal forms ("values"). While from a theoretical viewpoint it is convenient to consider values to be terms in normal form (Def. 3, our implementation represents each value by an integer. Depending on the sort (see module 2b) it belongs to, it may be interpreted as an index into into an identifier table `valTab[]` (like `"true"`, `"false"` for `t` and `f`, respectively), or handled by user-defined scan and print routines. This module provides access functions to `valTab[]`.
   (b) Module `opTab.c` — Administrates user-defined sorts. Allows for building a new sort from a given set of values. Checks user-claimed algebraic properties (associativity, commutativity, idempotency) of operator functions; this is feasible since all sorts are finite; to save computation time, only border-cases may be checked for associativity. Assigns an inhabited[2]- and needed[3]-flag to each sort.
   (c) Module `opDefTab.c` — Administrates user-defined operators and functors, including variables. The execution routine of such operators and functors simply does table-

[2] at least one term of this sort can be built
[3] at least one term of the sort of a goal contains a subterm of this sort

"val"	obtain the set of values
	"parse" read from file
"def"	obtain operator functionality
	"parse" read from file
"red"	obtain redices (Sect. 4.4)
	"parse" read from file
"var"	obtain variables and their value vectors
	"parse" read from file
	"all" one variable for each combination of values
	"seq" generate from given value sequence
	"univ" generate universal substitution variables
"goals"	obtain goals and their value vectors
	"parse" read from file
	"seq" generate from given value sequence
"upper"	obtain settings for projection-like operators (Sect. 4.6)
	"none" do not use any
	"idx" use array indexing
	"condChoice" use kind of $if(\cdot,\cdot,\cdot)$

Figure 9. Execution phases / contexts in module 4a

lookup in `opDefTab[]`. Operator and functor result values are read by `parseOpDefs` and stored consecutively, least argument running fastest, first one running slowest.

(d) Module `redex.c` — Administration of user-provided redices. See Sect. 4.4.

(e) Module `sorts.c` — Operator selection from argument or result sort. See Sect. 4.5.

(3) Goals and solution terms

(a) Module `term.c` — Generic term traversal routines; term enumeration; term normal-form computation (i.e. evaluation using the user-defined operator implementation routines)

(b) Module `contTab.c` — Administrates the set D of terms of minimal weight for their result vector. Those terms are ordered by weight and main operator. The table `contChainStart[]` allows one to access a list of all terms of a given weight. The function `idIT NextInhbArgOp(maIT *maList,cnIT *cnChain);` allows one to find the next operator common to `maList` (see module 2e) and `cnChain`.

(c) Module `goals.c` — Administrates given goal vectors. Allows for multiple goals. Allows the `compute` algorithm (module 5f) to enter a preliminary (non-minimal) as well as a final (minimal) solution term for a goal. Implements a timeout-mechanism to stop search and return the best (i.e. least-weight) solution terms found so far.

(4) Overall control

(a) Module `param.c` — Administration of command line arguments and global variables. As a poor-man's substitute for object oriented programming [4], we separated the program execution into phases [5], and implemented the execution of each phase by an indirect call only. This allowed us to change the behavior of each phase by command line options. Figure 9 shows a list of the execution phases. See also module 4b.

(b) Module `main.c` — As described in module 4a, `main()` just calls for each phase the corresponding routine, as set by the command line arguments.

[4] In fact, all but the core algorithm should better have been implemented in `C++`.

[5] called "contexts" in the C source code, in order to avoid confusion with a lower-level notion of "phase" used there for debugging purposes only

(5) Core algorithm
   (a) Modules `bbtTab.c` and `bbt.c` — Implements a balanced binary tree; keys are indices into `contTab` (module 3b). To save memory, we don't use nodes without child pointers; the data field of such a node is instead stored directly in the pointer field of its parent node.
   (b) Module `assoc.c` — Implements a lookup mechanism for indices into `contTab` on top of module 5a. Discussed in detail in Sect. 4.7.
   (c) Module `heap.c` — Implements a double-ended priority queue of weight-term indices, ordered by result weight; it is possible to extract the set of all weight-terms with least weight simultaneously. In case of imminent memory overflow, the largest weights may be efficiently found and discarded.
   (d) Module `weightTerm.c` — Administration of weight decomposition lists (called "weight-terms" in the implementation); each one can be accessed by an index of type `wtIT` that points to the `wtTab[]` table defined in this module.
   (e) Module `computeWt.c` — Implements Alg. 11. Provides optimized routines for commutative operators, see Sect. 4.3.
   (f) Module `compute.c` — Provides the `compute` routine, the implementation of Alg. 12 and 13. It includes optimized routines for associative, commutative, or idempotent operators, see Sect. 4.3.
(6) Module `upper.c` — Administration of projection-like operators, see Sect. 4.6 ("upper" refers to their position at the terms' roots).
(7) Equation generation — Used to generate equations from a given finite algebra, as described in [Bur02]. Not completely implemented, and not described here.
(8) Module `User.c` — Interface to user-definable code (see Sect. 4.2); contains initialization and finalization routines.

4.2 User modules overview

As a naming convention, file names starting with "`User`" indicate user-definable code. Whenever nontrivial parametrizations are to be user-provided we followed the approach to include them into the `C` source core, instead of providing an own parametrization language, this way saving the effort of implementing an according interpreter. Therefore, the user is required to re-compile the whole software e.g. after adding another weight function routine to module `UserWgf.c`. We briefly sketch each user module in the following.

(1) Module `UserOp.c` — Interface to operator-defining user modules: For each operator, we have to provide a computation routine, an initialization routine (called whenever the operator of the weight decomposition list changes in Alg.12's input), property flags, and a brief textual description to be shown by the command-line option `-help-ops`. Operators are grouped by application domain:
   (a) Module `UserOpArith.c` — Provides user-defined arithmetic operators, like $+$, $-$, $*$, $/$, $//$, $\%$, min, max, $<$, $\leqslant$, $=$, $\geqslant$, $>$, $\neg$, $\wedge$, $\vee$, $\Rightarrow$
   (b) Module `UserOpBit.c` — Provides bitwise operators on unsigned integers, like `&`, `|`, `^`, `~`, `<<`, `>>`
   (c) Module `UserOpString.c` — String operations (experimental, only a tiny character set and short maximal length can be supported), like concatenation, reversal, head, tail, character replacement, rotation, interleaving, length.

(2) Module `UserTerm.c` (doesn't work yet) — symbolic terms as values.
   (a) Module `UserTermArith0s.c` (doesn't work yet) — example application to symbolic terms built from `0` and `succ`, shall also eventually provide `+` and `*`.
   (b) Module `UserTermLambda.c` (doesn't work yet) — symbolic typed lambda terms.
   (c) Module `UserTermList.c` (doesn't work yet) — symbolic list built from `nil` and `cons`.
(3) Module `UserUp.c` — Interface to projection-like operator modules, see Sect. 4.6 for details. The following operator kinds are provided:
   (a) Module `UserUpCondChoice.c` — defines $if(\cdot,\cdot,\mathsf{u})$ and an angelic nondeterministic choice.
   (b) Module `UserUpIdx.c` — defines operators to select from a given array.
   (c) Module `UserUpNone.c` — trivial implementations to be used by the kernel code in absence of any projection-like operator.
(4) Module `UserVal.c` — defines routines to scan and print used-defined values, including (experimental) symbolic terms.
(5) Variable definitions modules:
   (a) Module `UserVarAll.c` — code to generate one variable for every possible combination of values, or for random combination of values.
   (b) Module `UserVarUniv.c` — code to generate variables corresponding to "universal substitutions", as described in [Bur05, Lem.5, p.7].
   (c) Module `UserSeq.c` — Application of E-anti-unification to sequence law guessing, as described in [Bur05, Sect.5.2, p.28–29].
(6) Module `UserWgf.c` — define code for weight functions, like `size`, and `height`.

Next, we elaborate on some selected details of our implementation.

4.3 Pruning for binary operators

When a binary operator $f \in \Sigma$ is associative, commutative, or idempotent, it is reasonable to expect that its weight function $\overline{f} : \mathcal{W} \times \mathcal{W} \longrightarrow \mathcal{W}$ shares the same properties. While we don't require this correspondence, we provide optimizations of Alg. 13 for those operators that obey it.

For this purpose, the following total well-ordering ($\succeq$) on terms has been approved useful: define $f(t_1,\ldots,t_n) \succeq f'(t'_1,\ldots,t'_{n'})$ iff

$$\mathsf{wg}(f(t_1,\ldots,t_n)) > \mathsf{wg}(f'(t'_1,\ldots,t'_n)), \quad \text{or}$$
$$\mathsf{wg}(f(t_1,\ldots,t_n)) = \mathsf{wg}(f'(t'_1,\ldots,t'_n)) \ \wedge \mathsf{ts}(f) \geqslant \mathsf{ts}(f'),$$

where $\mathsf{ts} : \Sigma \longrightarrow I\!N$ is an arbitrary ranking function[6] on the set of operator symbols. In particular, $t_1 \succeq t_2$ implies $\mathsf{wg}(t_1) \geqslant \mathsf{wg}(t_2)$ for arbitrary terms t_1, t_2, so the former ordering is a refinement of the latter.

In Alg. 11, we order the weight decomposition lists in the F heap in fact by $\succeq$, rather than just

[6] Its name derives from "time stamp", as we use just the order in which function symbols are entered into the symbol table in module 1b in Sect. 4.1.

by weight. More precisely, we compare such lists by

$$\begin{aligned} & \langle f, w_1, \ldots, w_n \rangle \succeq \langle f', w'_1, \ldots, w'_n \rangle \\ \text{iff} \quad & \overline{f}(w_1, \ldots, w_n) > \overline{f}'(w'_1, \ldots, w'_n) \\ \text{or} \quad & \overline{f}(w_1, \ldots, w_n) = \overline{f}'(w'_1, \ldots, w'_n) \quad \wedge \mathsf{ts}(f) \geqslant \mathsf{ts}(f'). \end{aligned}$$

As a consequence in Alg. 12, the set D holding the minimal terms generated so far is segmented not just by increasing weight, but moreover by increasing $\mathsf{ts}(\cdot)$ rank of the main operator. That is, for each $f \in \Sigma$, the set

$$\mathsf{D}_{w,f} = \{f(t_1, \ldots, t_n) \in \mathsf{D}_w \mid t_1, \ldots, t_n \in \mathcal{T}\}$$

corresponds to a continuous segment of the list implementing D_w. The finer segmentation is indicated by light grey bars in Fig. 2, assuming $\mathsf{ts}(0) < \mathsf{ts}(1) < \mathsf{ts}(2) < \mathsf{ts}(v_p) < \mathsf{ts}(v_1) < \mathsf{ts}(v_2) < \mathsf{ts}(+) < \mathsf{ts}(*)$.

If some operator $\oplus$ as well as its weight function $\overline{\oplus}$ is known to be commutative, we need to build the term $t_1 \oplus t_2$ in step 5b.i of Alg. 12 only if $t_1 \succeq t_2$. The term $t_2 \oplus t_1$ needn't be built in that case since it evaluates to the same value pair and has the same weight as $t_1 \oplus t_2$. Since building the former term is about as fast as testing the latter condition, we can't save time just by guarding the former by the latter. Instead, we need to restrict the loop 5b in Alg. 12 such that it ranges only over $t_1 \in \mathsf{D}_{w_1}, t_2 \in D_{w_2}$ with $t_1 \succeq t_2$.

To this end, we insert in step 5d of Alg. 11 a weight decomposition list $\langle \oplus, x_1, x_2 \rangle$ into F only if $x_1 \geqslant x_2$. Then, in step 5a of Alg. 12, only these lists will be read for $\oplus$, and hence the loop 5b will range only over terms t_1, t_2 of weight w_1, w_2 with $w_1 \geqslant w_2$. Since this covers all t_1, t_2 with $t_1 \succeq t_2$, the algorithm remains complete. It saves execution time since we have, per weight decomposition list $\langle \oplus, x_1, x_2 \rangle$, one extra test $x_1 \geqslant x_2$ in Alg. 11, but avoid building in Alg. 12 all terms from the usually large set $\{t_1 \oplus t_2 \mid t_1 \in D_{x_1}, t_2 \in D_{x_2}\}$ when the test fails.

Similarly, if $\oplus$ and $\overline{\oplus}$ is known to be both associative and commutative, we don't need to build terms of the form $t'_1 \oplus (t'_2 \oplus t'_3)$ at all, and need to build $(t'_1 \oplus t'_2) \oplus t'_3$ only if $t'_1 \succeq t'_2 \succeq t'_3$. Again, due to $\oplus$'s commutativity, it is sufficient in step 5d of Alg. 11 to insert a weight decomposition list $\langle \oplus, x_1, x_2 \rangle$ into F only if $x_1 \geqslant x_2$. Moreover, weight decomposition lists of the form $\langle \oplus, x_1, x_2 \rangle$ are handled differently in the steps below 5 of Alg. 12. If, in step 5b, t_1 is currently from $\mathsf{D}_{w_1,\oplus}$, i.e. if has the form $t_1 = t'_1 \oplus t'_2$, we choose t_2 such that $t'_2 \succeq t_2$, i.e. we choose t_2 from

$$\bigcup \{\mathsf{D}_{w_2,f} \mid w_2 \leqslant \mathsf{wg}(t'_2) \wedge \mathsf{ts}(f) \leqslant \mathsf{ts}(\oplus)\}.$$

This is achieved by function `selectArgOp1_2AC` in file `compute.c`.

Figure 10 shows the effect of this kind of pruning for associative, commutative, or/and idempotent operators on a practical example. [7] It shows the growth of D vs. the computation time

[7] Attempt to compute, from the operators listed in the caption, a law term for the integer sequence $55, 57; 60, 20, 10, 12, 15$ (task "A1 103" of the intelligence test IST'70-ZR).

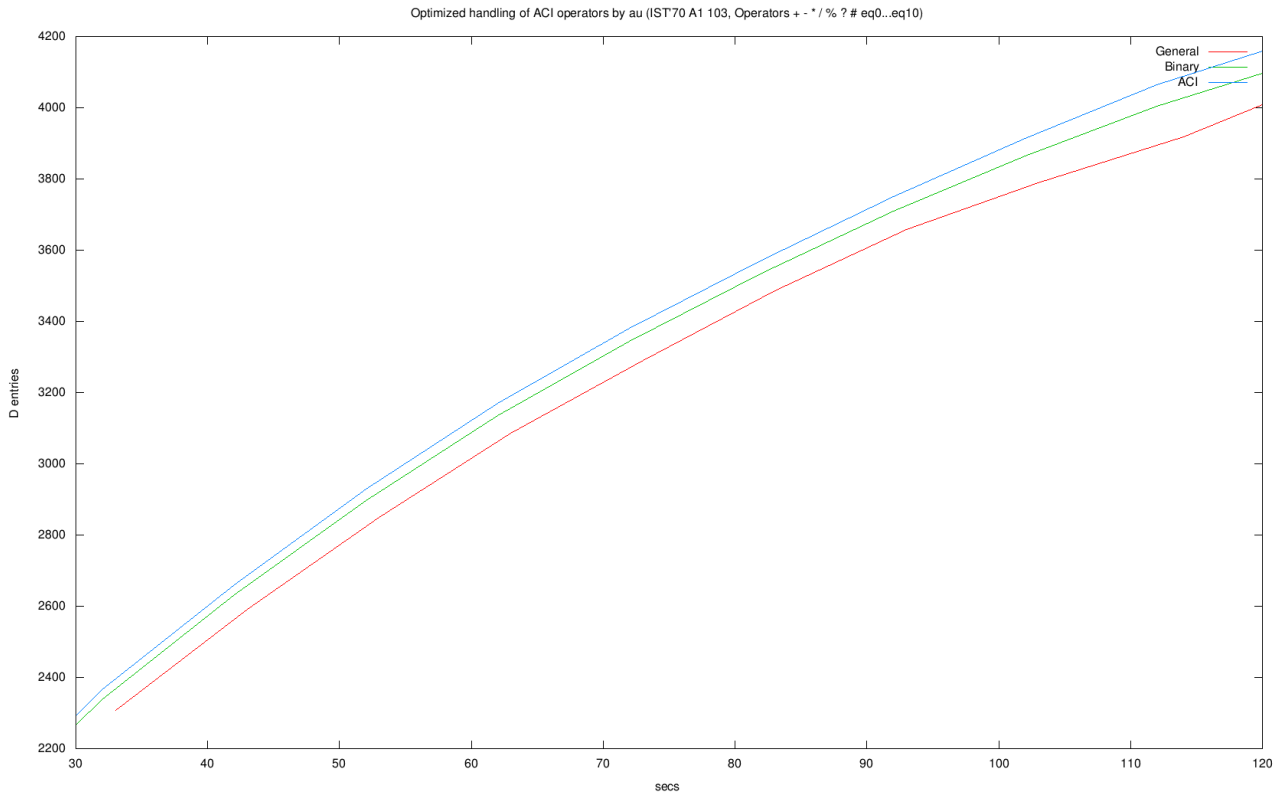

Figure 10. Acceleration by ACI pruning

for three different implementations of Alg. 13:

- "*General*" — without optimization
- "*Binary*" — uses particularized subroutines in Alg. 12 whenever a weight decomposition list with a binary operator is handled
- "*ACI*" — additionally implements pruning for associative, commutative, and/or idempotent operators as described above

The data shows that the improvement by the latter is rather small; after 120 seconds of computation we have a ratio $ACI{:}Binary = 101.5\%$ and $ACI{:}General = 103.8\%$.

4.4 Module `redex.c`

This module handles user-provided redices. For example, terms of the form $x * (y + z)$ need not be generated in addition to terms of the form $(x * y) + (x * z)$ if $*$ is known to distribute over $+$.

Only linear terms[8] of depth 2 can be given as redices; other terms, like e.g. $x - x$ and $\sqrt{x^2}$, cannot be handled. Redices are given in the form `MainOp ArgOp1 ... ArgOpN`, where `N` is the arity of `MainOp`, meaning that term of the form $MainOp(ArgOp1(\ldots), \ldots, ArgOpN(\ldots))$ need not be built. A single dot "`.`" denotes a fresh variable. In the above distributivity example, the redex "`* . +`" would be provided, denoting the redex $x_1 * (x_{21} + x_{22})$.

[8] i.e. terms without multiple occurrences of a variable

0:	$\longrightarrow$ int	
v:	$\longrightarrow$ int	
$+$: int, int	$\longrightarrow$ int	c,a
$-$: int, int	$\longrightarrow$ int	
$*$: int, int	$\longrightarrow$ int	c,a
$<$: int, int	$\longrightarrow$ bool	
$\vee$: bool, bool	$\longrightarrow$ bool	c,a,i
$\wedge$: bool, bool	$\longrightarrow$ bool	c,a,i
$\neg$: bool	$\longrightarrow$ bool	

$+$	0 .	not useful for pruning
$+$	. 0	not useful for pruning
$*$	$+$.	$(a+b)*c = a*c+b*c$
$*$	. $+$	$a*(b+c) = a*b+a*c$
$\wedge$	$\vee$.	$(a \vee b) \wedge c = a \wedge c \vee b \wedge c$
$\wedge$	. $\vee$	$a \wedge (b \vee c) = a \wedge b \vee a \wedge c$
$\neg$	$\vee$	$\neg(a \vee b) = \neg a \wedge \neg b$
$\neg$	$\wedge$	$\neg(a \wedge b) = \neg a \vee \neg b$
$\neg$	$\neg$	$\neg\neg a = a$

Figure 11. Example user-defined signatures (left) and redices (right)

	1	2
0		
v		
$+$	1	7
$-$	1	1
$*$	7	12
$<$	1	1
$\vee$	16	21
$\wedge$	25	29
$\neg$	32	

	0	1	2	3	4	5
0:	NIL					
1:	0	v	$+$	$-$	$*$	NIL
7:	0	v	$-$	$*$	NIL	
12:	0	v	$-$	NIL		
16:	$<$	$\vee$	$\wedge$	$\neg$	NIL	
21:	$<$	$\wedge$	$\neg$	NIL		
25:	$<$	$\wedge$	$\neg$	NIL		
29:	$<$	$\neg$	NIL			
32:	$<$	NIL				

Figure 12. `mainIndexTab` (left) and `mainArgTab` (right) resulting from Fig. 11

Simple redices, i.e. redices where `ArgOpI` is "." for all but one value of `I`, are considered in module `sorts.c` (see Sect. 4.5); they need not be kept in the redex table. All remaining redices are kept in that table, indexed by their main operator, and are used in the routine `bool isRedex(idIT mainOp,const idIT argOp[])`.

Filtering based on `isRedex` is done after drawing a weight term, and after choosing `argOp`s, but before choosing terms from D, in the routines `selectDnArgOpsV`, `selectArgOp1_2`, `selectArgOp1_2C`, `selectArgOp1_2AC`, `selectWt1`, all from file `compute.c`.

4.5 Many-sorted signatures for operators

In order to exclude nonsensical terms like e.g. $3 + (2 > 1)$ from the build process in Alg. 13, we implemented many-sorted signatures for operators. The user can define arbitrarily many sorts, each by either listing its member values (cf. module 2a in Sect. 4.1) or by naming its `print` and `scan` routines (defined in `UserVal.c`, cf. module 4 in Sect.4.2). Each operator is declared having a fixed signature of input sorts and a result sort (module 2b in Sect. 4.1).

In file `sorts.c`, we maintain tables `mainIndexTab` and `mainArgTab` about which operators are allowed at a given argument position of a given main operator. This way, in step 5b.i of Alg. 12 we need to consider only terms t_i starting with an operator allowed at position i of the main operator f. Given w, f, and i, the function `NextInhabitedArgOp` in file `contTab.c` is used to enumerate all operators f_i such that $\mathsf{D}_{w,f_i} \neq \{\}$ and f_i is allowed at position i of f.

In addition to sort information, simple redex information can be considered in the tables `mainIndexTab` and `mainArgTab`. For the example redex $x_1 * (x_{21} + x_{22})$ from Sect. 4.4, we

can remove "+" from the list of allowed operators at position 2 of "∗". Similarly, we can remove "∗" from the same list if we know that "∗" is associative. Moreover, an uninhabited or unneeded sort (see module 2b in Sect. 4.1) can be removed from all such lists.

Figure 11 (left) shows a user-defined example set of operator signatures and properties, with "c", "a", and "i" indicating commutativity, associativity, and idempotency, respectively. Figure 11 (right) show a user-defined list of simple redices, with comments or justifying equations right of the box. Figure 12 (left) and (right) shows the layout of the resulting `mainIndexTab` and `mainArgTab`, respectively. The former is indexed (vertically) with an operator and (horizontally) with an argument position, yielding in turn the index where the list of allowed operators starts in `mainArgTab`. For example, `mainIndexTab[*,1] = 7` and `mainArgTab[7]` represents the `NIL`-terminated list `0`, `v`, `-`, `*`.

4.6 Projection-like operators

In many applications of E-generalization, or just of function synthesis, considering $if(\cdot,\cdot,\cdot)$ or a similar operator is indispensable at least from a practical point of view (cf. the discussion in Exm. 23). In this Section, we formally define the general notion of a *projection-like operator*, and describe implementation optimizations for Alg.13 related to that operator class.

We call $f : \mathsf{NF}^{m+n} \to \mathsf{NF}$ a projection-like operator, if there is a partition of its arguments into (w.l.o.g.) $\mathsf{NF}^m \times \mathsf{NF}^n$ such that always

$$f(c_1, \ldots, c_m, d_1, \ldots, d_n) \in \{\vec{d_1}, \ldots, \vec{d_n}\} \cup \{\mathsf{u}\}$$

and it depends only on $c_1, \ldots, c_m$ which argument position is "switched through" to the output, that is, if

$$\forall c_1, \ldots, c_m \; \exists i \; \forall d_1, \ldots, d_n : \; f(c_1, \ldots, c_m, d_1, \ldots, d_n) = d_i$$

holds. We call $c_1, \ldots, c_m \in \mathsf{NF}^m$ the control and $d_1, \ldots, d_n \in \mathsf{NF}^n$ the data input to f.

Examples of projection-like operators include: [9]

- Projection $\pi_i^n : \mathsf{NF}^0 \times \mathsf{NF}^n \to \mathsf{NF}$, with $\pi_i^n(d_1, \ldots, d_n) = d_i$;
- Index $\mathsf{idx} : \mathsf{NF} \times \mathsf{NF}^n \to \mathsf{NF}$, with $\mathsf{idx}(c, d_1, \ldots, d_n) = d_c$, for $1 \leqslant c \leqslant n$;
- If-then-else $if : \mathsf{NF} \times \mathsf{NF}^2 \to \mathsf{NF}$, with $if(\mathsf{t}, d_1, d_2) = d_1$ and $if(\mathsf{f}, d_1, d_2) = d_2$;
- Ifdef $? : \mathsf{NF} \times \mathsf{NF} \to \mathsf{NF}$, with $?(\mathsf{t}, d) = d$;
- Choice $\#^n : \mathsf{NF}^n \times \mathsf{NF}^n \to \mathsf{NF}$, with $\#^n(c_1, \ldots, c_n, d_1, \ldots, d_n) = d_i$ if i is minimal with $\{c_1, \ldots, c_n\} \subseteq \{c_i, \mathsf{u}\}$.
  The classical angelic choice operator returns u if its welldefined input values disagree, and the unique welldefined input value, else. It can be modeled by $\#^n(x_1, \ldots, x_n, x_1, \ldots, x_n)$, taking the same input as both control and data input.

Provided all operators are *strict*, i.e. return u whenever one of their inputs is u, it possible to write every term such that never a projection-like operator occurs below a non-projection-like one. Formally, let $f_i : \mathsf{NF}^{m_i} \times \mathsf{NF}^{n_i} \to \mathsf{NF}$ be projection-like for $i = 1, \ldots, l$, and $g : \mathsf{NF}^l \to \mathsf{NF}$

[9] All operator results are u for all input value combinations not explicitly shown.

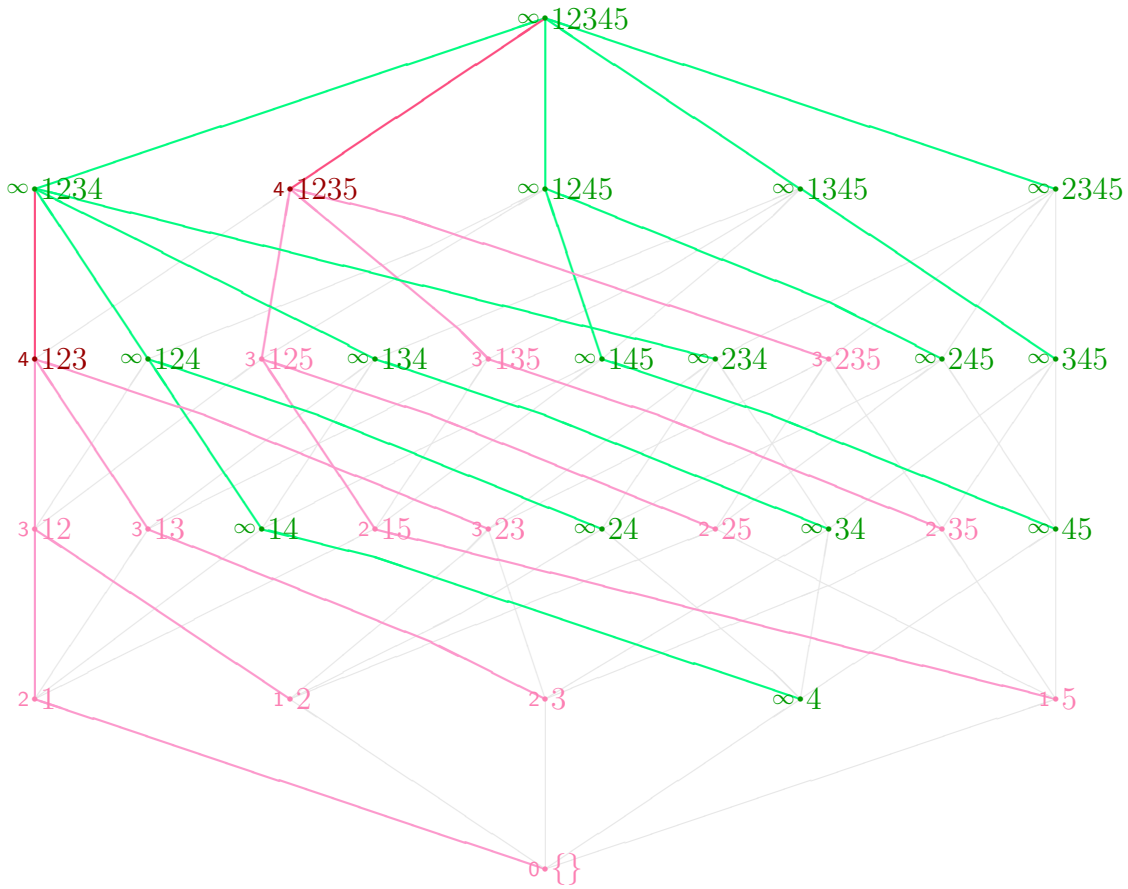

Figure 13. Example update of data term lattice

arbitrary. Each term

$$g(f_1(c_{11}, ..., c_{1m_1}, d_{11}, ..., d_{1n_1}), \ldots, f_l(c_{l1}, ..., c_{lm_l}, d_{l1}, ..., d_{ln_l}))$$

equals a term

$$f(c_{11}, ..., c_{1m_1}, \ldots, c_{l1}, ..., c_{lm_l}, g_1(...), \ldots, g_{...}(...))$$

for a suitable projection-like operator $f : \mathsf{NF}^{m_1+...+m_l} \times \mathsf{NF}^{\cdots} \to \mathsf{NF}$. Therefore, we don't lose completeness of Alg. 13 if we modify it such that projection-like operators appear only at the top of built terms. However, moving projection-like operators to the top may result in exponential term growth.

Since our implementation optimizations don't make much sense when the E-generalization of only 2 terms is computed, we assume in the rest of this section that an arbitrary number K of terms $t_1, \ldots, t_\mathsf{K}$ is to be generalized simultaneously. The algorithms from Sect. 3 can be extended in a straight-forward way. We call a subset of $\{1, \ldots, \mathsf{K}\}$ an index set.

4.6.1 The data term lattice

We maintain a lattice of all subsets of $\{1, \ldots, \mathsf{K}\}$, i.e. all index sets. For every such subset, the lattice holds a "data" term of minimal weight found so far that matches the goal tuple at least at that subset. Whenever a new term is entered into D, its index positions matching the goal tuple are determined, and the data term lattice is updated accordingly. The lattice satisfies the invariant that the weight of a node is larger or equal than that of each of its subset nodes. In particular, its bottom node always has the least weight at all in $\mathcal{W}$. When no term has yet been found for some index set, its weight is set to ∞.

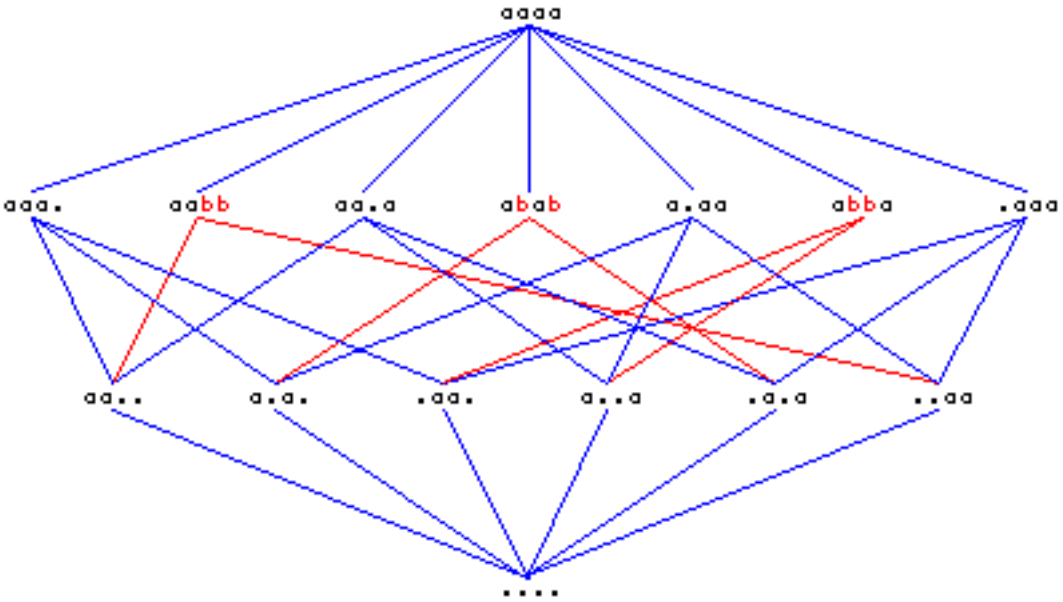

Figure 14. Lattice of equivalence relations for $\mathsf{K} = 4$

Figure 13 shows an example update operation of the data term lattice for $\mathsf{K} = 5$, abbreviating e.g. $\{1, 2, 3, 4, 5\}$ as 12345. To the right and left of each node, the set of indices matching the goal tuple and the minimal weight found so far is shown, respectively. Figure 13 assumes a term of weight 5 matching the goal tuple at position $\{1, 2, 3, 4, 5\}$ has been found recently, and is about to be entered into the lattice. To this end, all green nodes are to be updated to weight 5, while red nodes are left unchanged. Updates are attempted along the spanning tree of the sublattice below $\{1, 2, 3, 4, 5\}$; it is obtained by connecting each index set (e.g. $\{1, 2, 4\}$) to its leftmost, i.e. lexicographically least, immediate superset (e.g. $\{1, 2, 3, 4\}$). Observe that the structure of the spanning tree depends on its topmost node. For example, the lattice for $\{1, 2, 3, 4\}$ is a sublattice of that for $\{1, 2, 3, 4, 5\}$, while the spanning tree for $\{1, 2, 3, 4\}$ is not a subtree of that for $\{1, 2, 3, 4, 5\}$.

While the data term lattice is independent of the particular kind of projection-like operators in use, the control term data structure does depend on the latter. We describe the structures and algorithms for the `C switch` like idx operator family [10] in the following; the usual ternary $\mathit{if}(\cdot, \cdot, \cdot)$ and some variants can be handled in a similar way.

4.6.2 The idx *control term lattice*

We maintain a lattice of all equivalence relations on the set of indices of value tuples. As an example, Fig. 14 shows the lattice for $\mathsf{K} = 4$, with "a" and "b" in an equivalence relation's node denoting the partition an element is mapped to by that relation, and "." indicating mapping to a singleton partition; e.g. "aa.." denotes the partition $\{a = \{1, 2\}, \{3\}, \{4\}\}$, corresponding to the equivalence relation $\{\langle 1, 1\rangle, \langle 1, 2\rangle, \langle 2, 1\rangle, \langle 2, 2\rangle, \langle 3, 3\rangle, \langle 4, 4\rangle\}$.

For every such equivalence relation R, its lattice node holds a "control" term t of minimal weight found so far such that $\mathsf{nf}(t\sigma_i) = \mathsf{nf}(t\sigma_j)$ iff $i \; R \; j$. For each R, we also maintain the

[10] The $n+1$-ary operation $\mathsf{idx}_n(c, d_0, \ldots, d_{n-1})$ is defined to yield d_c if $0 \leqslant c < n$, and u, else, similar to an array indexing expression `d[c]` in imperative programming languages.

count of partitions of R for which data terms exist in the data lattice. When a data term exists for each partition, and a control term exists, we can build a solution term by composing them appropriately with an idx operator.

For example, from a control term t_c evaluating to 08101, and data terms t_0, t_1, and t_8 matching the goal vector at index $1, 4$, at index $3, 5$, and at index 2, respectively, we can build the term $\mathsf{idx}_9(t_c, t_0, t_1, 0, 0, 0, 0, 0, 0, t_8)$, which matches the goal vector at index $1, 2, 3, 4, 5$ by construction. The zero arguments of idx_9 are used for padding purposes only, i.e. to get t_8 into the right place.

Whenever a new control[11] term is entered into D, and its node in the control lattice is still empty, we store it there, and initialize its partition count. An earlier term need never be replaced by a later one, since terms appear in order of increasing weight.

Whenever a new data term is entered into D, we update the data term sublattice below it as described above in Sect. 4.6.1. For each updated sublattice node, corresponding to a set S of indices, we increment the partition count in the control term lattice for all equivalences having S as one of its partitions. These equivalences are found as refinements of the relation $i\ R\ j \Leftrightarrow (i \in S \Leftrightarrow j \in S)$.

4.7 Module `assoc.c`

This module implements a lookup mechanism for indices into `contTab` on top of `bbt.c` (module 5a in Sect. 4.1). For each user-sort, a hash table is allocated, the size of which is always some power c^n of the sort's cardinality c. Each hash entry then points to a balanced binary tree.

Figure 15 shows an example,[12] assuming a sort `bool` with values `f` and `t`, a sort `int` with values `0` to `9`, operators $\mathtt{0}, \mathtt{1}, \mathtt{2}, \mathtt{3} :\rightarrow \mathtt{int}$, $+, * : \mathtt{int}, \mathtt{int} \rightarrow \mathtt{int}$, $< : \mathtt{int}, \mathtt{int} \rightarrow \mathtt{bool}$, and variables $\mathtt{x}, \mathtt{y} :\rightarrow \mathtt{int}$ with $\mathtt{x} \simeq \langle \mathtt{5}, \mathtt{2}, \mathtt{1} \rangle$ and $\mathtt{y} \simeq \langle \mathtt{5}, \mathtt{9}, \mathtt{2} \rangle$.[13] We have $c = 2$, $n = 3$ for `bool`, and $c = 10$, $n = 1$ for `int`. The balanced binary tree for the `int` entry `5` represents the term set $\{2 + 3, x, y\}$ of all terms evaluating to some $\langle 5, \cdot, \cdot \rangle$ that have been found so far.

The lexicographic comparison of value vectors within a binary tree need not consider the first n vector components, as they always agree. Therefore, we store the value of n in to field `cmpStart` of a `struct _userSort` (module 2b in Sect. 4.1).

If, for some sort, c^n is sufficiently small, all possible value vectors fit into the hash table, like for sort `bool` in the above example. In this case, only trivial balanced binary trees occur as hash entries. Moreover, when each hash entry is filled, we know that a minimal term has already been generated for each value vector; and we need not build any more terms of the sort. For this purpose, we use the field `hashEmptyCnt` of a `struct _userSort` and the flag `usfSaturated`. The latter is checked by `computeHeapSequence` in `compute.c`.

---

[11] A term t is a potential control term if it has an appropriate sort (e.g. `int`) and $0 \leqslant t\sigma_i < A - 1$ for all i, where A denotes the maximal arity of admitted idx operators.

[12] Our implementation includes an own undefined value u_s into every sort s. For sake of simplicity we ignore them in this example.

[13] i.e. with $\mathsf{K} = 3$, $\sigma_1 = \{x \mapsto 5, y \mapsto 5\}$, $\sigma_2 = \{x \mapsto 2, y \mapsto 9\}$, and $\sigma_3 = \{x \mapsto 1, y \mapsto 2\}$

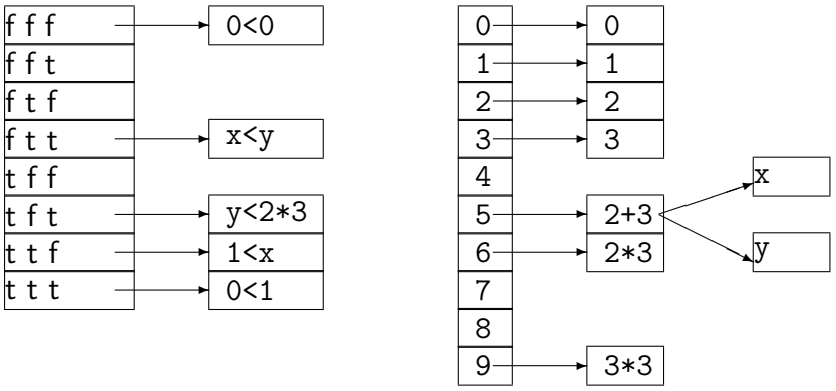

Figure 15. Example hash tables in `assoc.c` (Sect. 4.7)

5 Run time statistics

In this section, we give some statistical figures about a typical run of our implementation. In order to compute a law term for the IST'70-ZR test "A1 102",[14] we set $\mathsf{K} = 6$ and search a term for the value vector $\langle 3, 6, 8, 16, 18, 36\rangle$, given the variables $v_1 = \langle 1, 3, 6, 8, 16, 18\rangle$ and $v_p = \langle 0, 1, 2, 3, 4, 5\rangle$; admitted operators were $+, -, *, /, //, \%, \mathsf{idx}_2, \mathsf{idx}_3, \ldots$, all weight functions were such that weight and size[15] of a term agreed. Before the first second elapsed, during the computation of terms of weight 5, the terms $v_p\%2$, $v_1 * 2$, and $v_1 + 2$ became available[16] and the term $\mathsf{idx}_2(v_p\%2, v_1 * 2, v_1 + 2)$ was built and found to be a solution. However, the search continued since this solution was not known to be of minimal weight, while a minimal solution was required by a command-line option. After 210 seconds of user time, the memory was exhausted during build of weight level 12, and the implementation aborted, returning the above term as the best available solution.

Figure 16 shows the number of terms for which the value vectors were computed and turned out to be new ("computeSolved"), already known ("computeAgain"), undefined ("computeUndef"), and for which ϕ was looked up ("bbtEnter", cf. Fig. 2); the latter equals the sum of "computeSolved" and "computeAgain".

Figure 17 shows the memory consumption

- for the balanced binary tree implementing ϕ ("bbt1" and "bbt02", corresponding to nodes with 1 and 2 children, respectively),
- for D ("cont"),
- for F ("heap"), and
- for the table holding weight terms ("wt", pointed-to by "heap" entries).

Figure 18 shows the number of computed terms for each second of elapsed time. Finally, Fig. 19 shows for each weight the number of new, undefined, and old terms of that weight, and the total time spent with their computation.

[14] This test asks for a law of $1; 3, 6, 8, 16, 18, 36$; e.g. the term $if(v_p\%2 = 0, v_1 + v_1, v_1 + 2)$ is a "correct" solution.
[15] See Exm. 8.
[16] Due to sort inclusion functions that are omitted here for simplicity, the terms have a size of 4, 5, and 5, respectively.

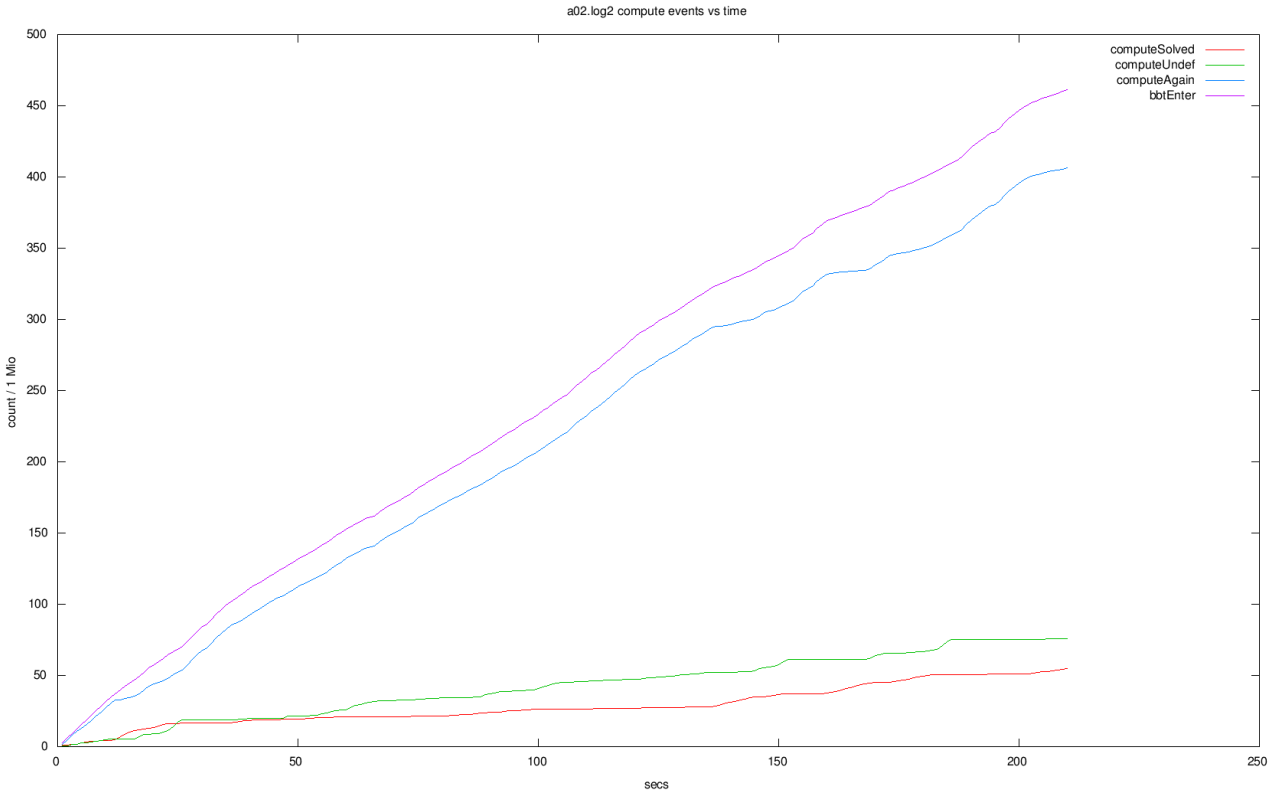

Figure 16. Compute event statistics

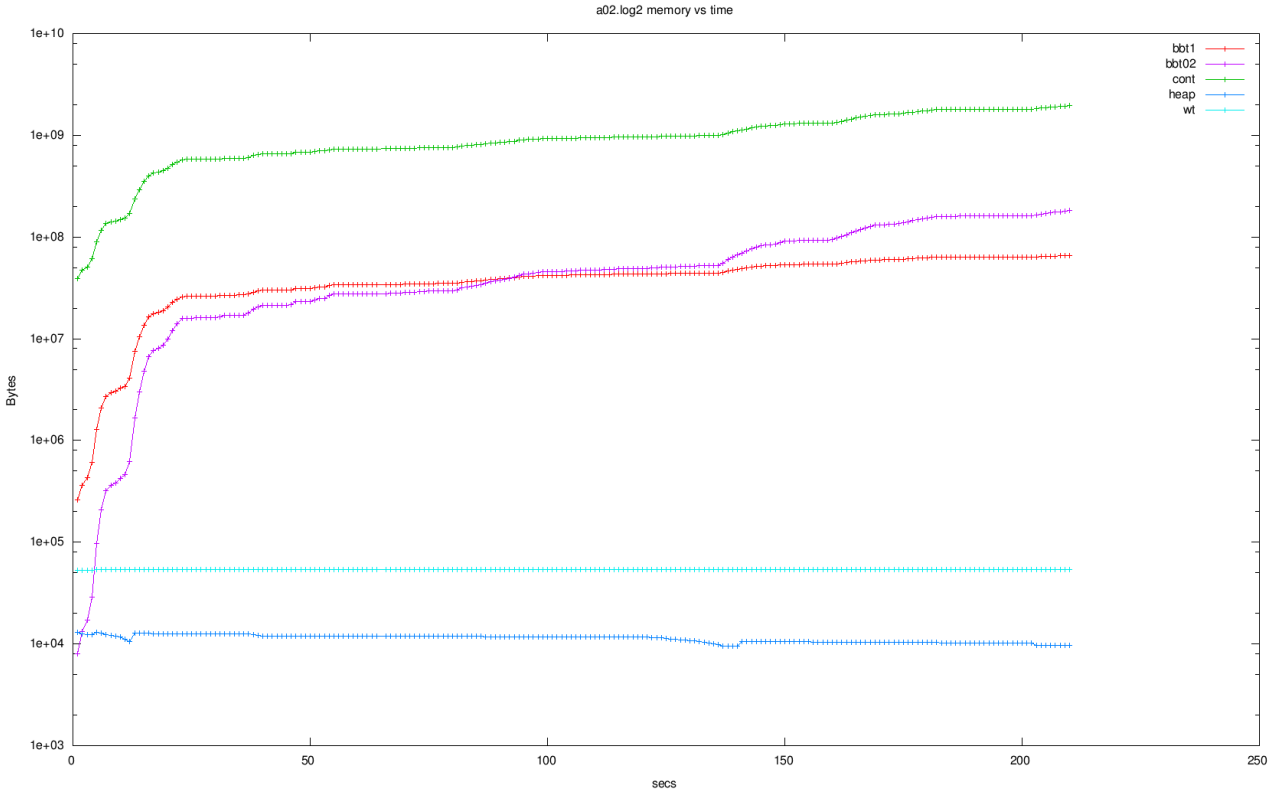

Figure 17. Memory statistics

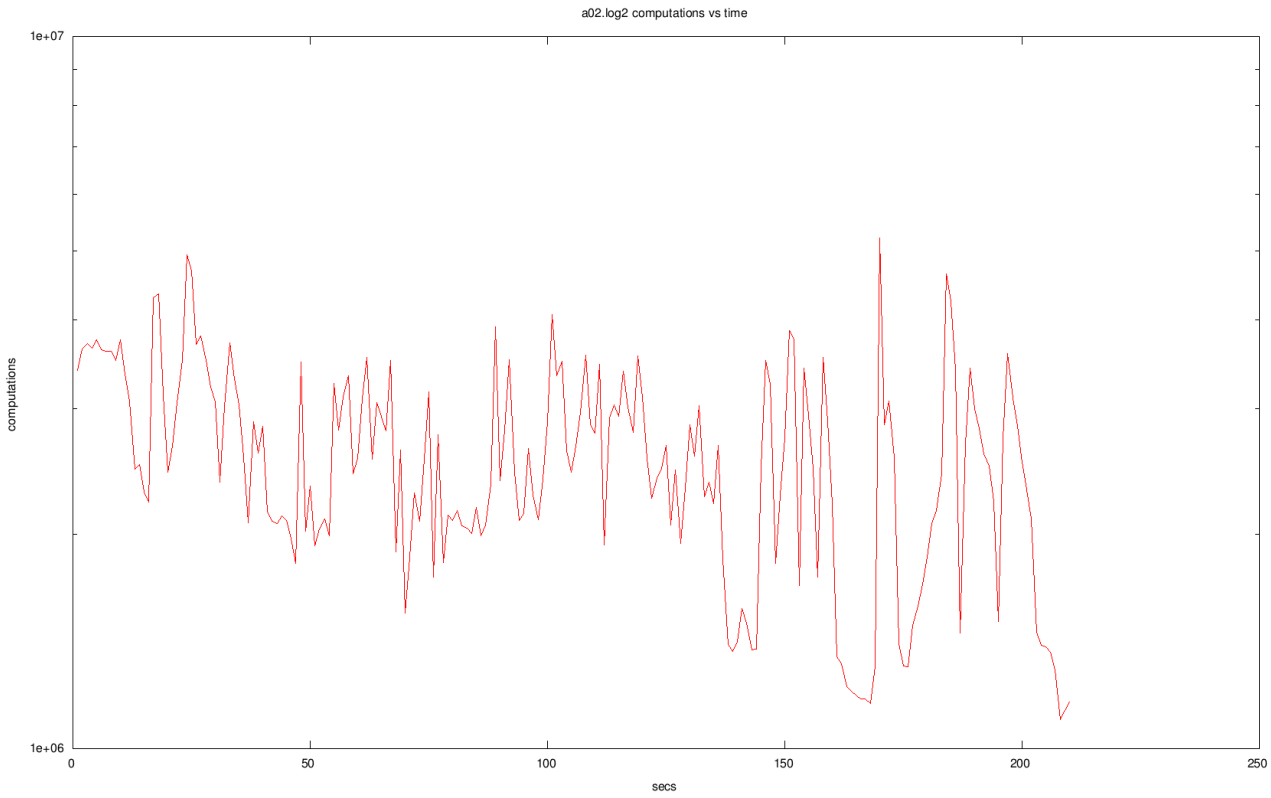

Figure 18. Value vectors statistics

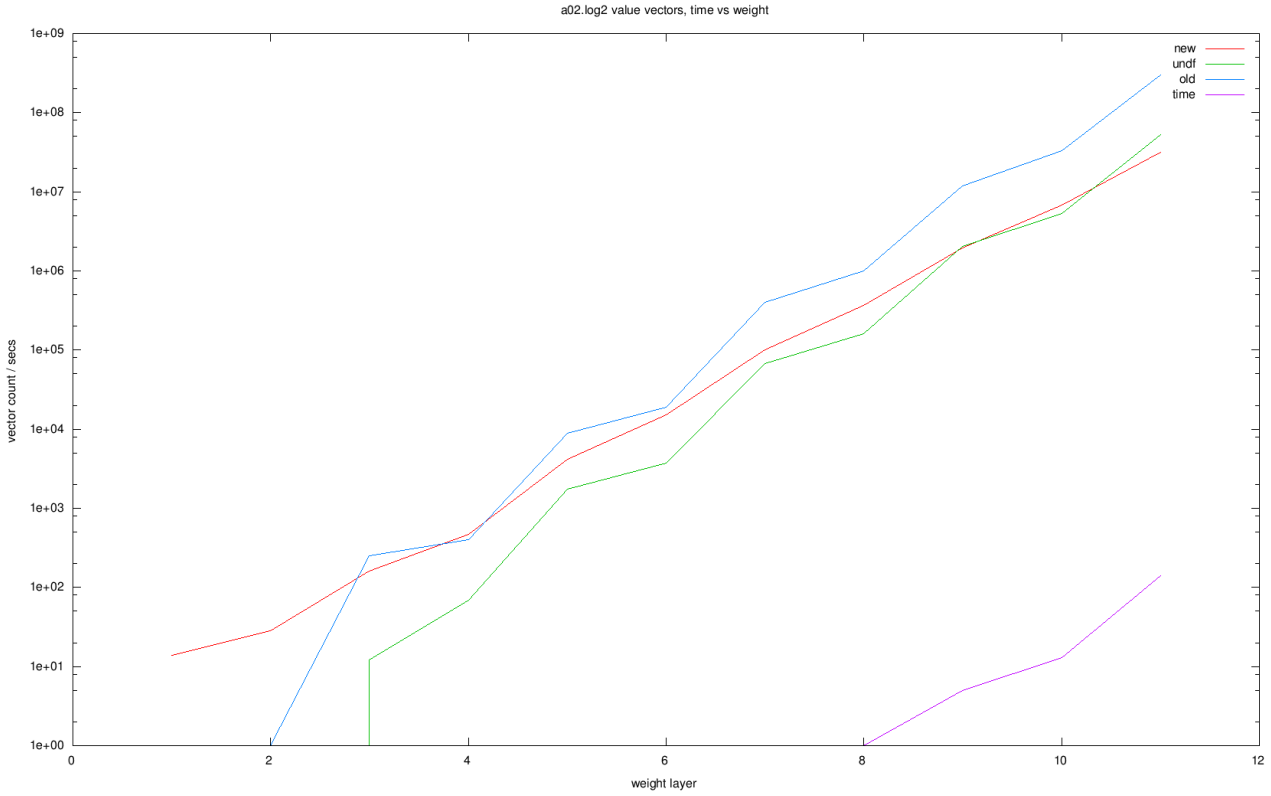

Figure 19. Weight statistics

References

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading/MA, 1974.

[Amt73] Rudolf Amthauer. *IST 70 Intelligenz-Struktur-Test — Handanweisung für die Durchführung und Auswertung*. Verlag für Psychologie Hogrefe, Göttingen, 4th edition, 1973.

[Bur02] Jochen Burghardt. Axiomatization of finite algebras. In *Proc. KI 2002*, number 2479 in LNAI, pages 222–234. Springer, 2002.

[Bur03] Jochen Burghardt. Weight computation of regular tree languages. Technical Report 001, FIRST, Dec 2003.

[Bur05] Jochen Burghardt. *E*-generalization using grammars. *Artificial Intelligence Journal*, 165(1):1–35, 2005.

[CDG+99] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications*. Available from `www.grappa.univ-lille3.fr/tata`, Oct 1999.

[CDG+01] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications*. Oct 2001.

[DJ90] N. Dershowitz and J.-P. Jouannaud. *Rewrite Systems*, volume B of *Handbook of Theoretical Computer Science*, pages 243–320. Elsevier, 1990.

[Duf91] David A. Duffy. *Principles of Automated Theorem Proving*. Wiley, New York, 1991.

[Hal68] Paul R. Halmos. *Naive Set Theory*. Nostrand, Princeton, 1968.

[Knu77] D.E. Knuth. A generalization of Dijkstra's algorithm. *Information Processing Letters*, 6(1):1–5, 1977.

[Pad89] Peter Padawitz. Inductive proofs by resolution and paramodulation. Internal report, Univ. Passau, 1989.

[TW68] J.W. Thatcher and J.B. Wright. Generalized finite automata theory with an application to a decision problem of second-order logic. *Mathematical Systems Theory*, 2(1), 1968.