Termination by Abstraction

Nachum Dershowitz*

School of Computer Science, Tel-Aviv University,
Ramat Aviv, Tel-Aviv 69978, Israel
nachum.dershowitz@cs.tau.ac.il

Abstract. Abstraction can be used very effectively to decompose and simplify termination arguments. If a symbolic computation is nonterminating, then there is an infinite computation with a top redex, such that all redexes are immortal, but all children of redexes are mortal. This suggests applying weakly-monotonic well-founded relations in abstraction-based termination methods, expressed here within an abstract framework for term-based proofs. Lexicographic combinations of orderings may be used to match up with multiple levels of abstraction.

A small number of firms
have decided to terminate
their independent abstraction schemes.

– Netherlands Ministry of Spatial Planning,
Housing and the Environment (2003)

1 Introduction

For as long as there have been algorithms, the question of their termination – though undecidable, in general – has had to be addressed. Not surprisingly, one of the earliest proofs of termination of a computer program was by Turing himself [43], mapping the program state to the ordinal numbers.

Floyd [22] suggested using arbitrary well-founded (partial) orderings; this direction was developed further by Manna [34]. Such a termination proof typically involves several steps:

1. Choose an appropriate well-founded set.
2. Choose a set of points in each potentially infinite computation at which to measure progress towards termination.
3. Establish invariant properties that always hold at those points.
4. Choose a mapping from states to the well-founded set by which to measure progress.
5. Show a necessary decrease in this measure with each transition from point to point.

* Research supported in part by the Israel Science Foundation (grant no. 254/01).

B. Demoen and V. Lifschitz (Eds.): ICLP 2004, LNCS 3132, pp. 1–18, 2004.

For a survey of termination methods for ordinary programs, see [30][1].

Showing termination of symbolic computations often requires special tools, since state transitions involve symbolic expressions that may grow bigger and bigger, while progress is being made towards a final result. Therefore, one often resorts to powerful term-based orderings, such as have been developed for rewrite systems [13]. We are mainly interested here in relatively simple symbolic termination functions, mapping symbolic states to terms, and in sophisticated methods of showing that they decrease. More complicated symbolic transformations have been considered, for example in [3, 4].

We use rewriting [15, 20, 42] as a prototypical symbolic computation paradigm (and employ terminology and notation from [20]). A rewrite system is *(uniformly) terminating* if there is no term to which rules in can be applied over-and-over-again forever; see [13]. Narrowing (a unification-based version of rewriting) has been proposed as a basis for functional-logic programming; see [19, 27]. Termination of narrowing has been considered in several works [28, 21, 6]. Much effort has also been devoted to devising methods for establishing termination of logic programs. For a survey, see [10]; a recent dissertation on the subject is [39]; interfaces to several automated tools (cTI, Hasta-La-Vista, TALP, TermiLog, and TerminWeb) are available over the web. Methods have been suggested for converting well-moded logic programs into rewrite systems with identical termination behavior [2, 36].

In the next section, we sketch how abstraction is used to decompose termination proofs. Section 3 introduces notation and monotonicity properties, and is followed by a section containing some termination methods for rewriting based on those properties. In Section 5, we look at constricting derivations, which are used in the following section to design dependency-based approaches, in which the symbolic state is a "critical" immortal subterm. Correctness of the various methods and their interrelatedness are the subjects of Section 7. We conclude with an example.

2 Abstraction

A *transition system* is a graph in which vertices are states (S) of a computation and edges ($\rightsquigarrow$) are state-transitions, as defined by some program. A *computation*

[1] It is misleading to suggest (cf. [26]) that – for deterministic (or bounded-nondeterministic) programs – it suffices to use the natural numbers as the well-founded set (Step 1), claiming that – after all – the (maximum) number of iterations of any terminating loop is fixed and depends only on the values of the inputs. This fixation on the naturals begs the real issue, since the proof (Step 5) may require transfinite induction over ordinals much larger than ω. For example, one can easily program the deterministic Battle of Hercules and Hydra (or Goodstein sequences) [31]. Though there exists an integer-valued function that counts how many steps it takes Hercules to eliminate any Hydra, *proving* that it is well-defined, and that it decreases with each round of the battle, provably *requires* a stronger principle of induction (viz. ε_0) than that provided by the Peano Axioms of arithmetic.

is a sequence $s_0 \rightsquigarrow s_1 \rightsquigarrow \cdots$ of states, where the arrows represent transitions. We say that a binary relation $\gg$ (over some S) is *terminating* if there are no infinite descending sequences $s_1 \gg s_2 \gg \cdots$ (of elements $s_i \in S$). This property will be denoted SN($\gg$) for "strongly normalizing". Thus, we aim to show SN($\rightsquigarrow$), that is, that the transition relation $\rightsquigarrow$ is terminating, for given transition systems $\rightsquigarrow$. To show that no infinite computation is possible, one can make use of any other terminating relation $\gg$, and show that transitions $\rightsquigarrow$ are decreasing in $\gg$. That is, we need $s \rightsquigarrow s'$ to imply $s \gg s'$, or $\rightsquigarrow \subseteq \gg$, for short.

Abstraction and dataflow analysis can be used to restrict the cases for which a reduction needs to be confirmed. The underlying idea is that of abstract interpretation, as introduced by Sintzoff [40], Wegbreit [45], and others, and formalized by Cousot and Cousot [9]. The property we are concerned with here is termination. For use of abstraction in termination of logic programs, see [44].

A partial ordering $>$ is *well-founded* if it is terminating. If $\geq$ is a quasi-ordering (i.e. a reflexive-transitive binary relation) and $\leq$ its inverse, then we can use $\simeq$ to denote the associated equivalence ($\geq \cap \leq$, viewing orderings as sets of ordered pairs) and $>$ to denote the associated partial ordering ($\geq \setminus \leq$). We will say that a quasi-ordering $\geq$ is well-founded whenever its strict part $>$ is. We often use well-founded partial and quasi-orderings in proofs, since they are transitive. Specifically, we know that $s \geq t > u$ and $s > t \geq u$ each imply $s > u$.

As is customary, for any binary relation $\looparrowright$, we use $\looparrowright^+$ for its transitive closure, $\looparrowright^*$ for its reflexive-transitive closure, and $\looparrowright^-$ (or $\hookleftarrow$, typography permitting) for its inverse. If a relation $\succ$ is terminating, then both its transitive closure $\succ^+$ and reflexive-transitive closure $\succ^*$ are well-founded. *In what follows, we will dedicate the symbol $\succ$ for terminating relations, $\succ$ for well-founded partial orderings, and $\succsim$ for well-founded quasi-orderings.* The intersection of a terminating relation with any other binary relation is terminating:

$$\mathrm{SN}(\succ) \quad \Rightarrow \quad \mathrm{SN}(\succ \cap \gg)\,. \tag{1}$$

It is often convenient to introduce an intermediate notion in proofs of termination, namely, a "termination function" τ, mapping states to some set W, and show that state transition $s \rightsquigarrow s'$ implies $\tau(s) \succ \tau(s')$, for some terminating relation $\succ$. Accordingly, one can view $\tau(s)$ as an "abstraction" of state s for the purposes of a termination proof. Instead of proving that $\rightsquigarrow$ is terminating, one considers the abstracted states $\tau(S) = \{\tau(s) \mid s \in S\} \subseteq W$ and supplies a proof of termination for the abstract transition relation $\tau(\rightsquigarrow)$, defined as $\{\tau(s) \rightsquigarrow \tau(s') \mid s \rightsquigarrow s'\}$.

Suppose the only loops in the abstracted transition graph $\tau(S)$ are self-loops. That is, $\tau(s) \rightsquigarrow^* \tau(s') \rightsquigarrow^* \tau(s)$ implies $\tau(s) = \tau(s')$. Then termination can be decomposed into subproofs for each of the loops and for the remainder of the graph, *sans* loops. For the latter, one needs to check that $\tau(\rightsquigarrow)$ has no infinite chains, which is trivially true when the abstract graph is finite. For each of the self-loops, one needs to reason on the concrete level, but under the assumption that τ remains invariant (its value is some constant).

Oftentimes [34], one maps states to a lexicographically-ordered tuple of elements, a pair $\langle \tau_1(s), \tau_2(s) \rangle$, say. Then one needs to show, separately (if one wishes), that every transition $s \rightsquigarrow s'$ implies $\tau_1(s) \succsim \tau_1(s')$, for some well-founded quasi-ordering $\succsim$, and that $s \rightsquigarrow s'$ and $\tau_1(s) \simeq \tau_1(s')$ imply $\tau_2(s) \succ \tau_2(s')$, for some terminating relation $\succ$.

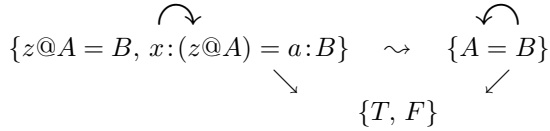
In the symbolic case, the set of ground terms in a computation can be divided according to some set of patterns of which they are instances. If there are only a finite number of different patterns, and computations do not cycle among the patterns, then one only needs to show termination of computations involving a single pattern. In logic programming, these can be predicate names and argument modes. For rewriting, syntactic path orderings [12, 13], based on a precedence of function symbols, are used, but one must consider subterms, as well as the top-level term. Abstraction is also the essence of the "operator derivability" method of [21] for pruning unsatisfiable narrowing goals (as used for functional-logic programming), where terms $f(\cdots)$ are represented by their outermost symbol f. A more sophisticated use of patterns to prune narrowing-based goal solving was developed in [6].

Example 1. The rewrite system

$$\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl} \varepsilon @ z & \to & z & \varepsilon = \varepsilon & \to & T & \varepsilon = x:y & \to & F \\ (x:y) @ z & \to & x:(y @ z) & x:y = x:z & \to & y = z & x:y = \varepsilon & \to & F\,, \end{array}$$

for appending and comparing lists, can be used to compute directly by rewriting (using pattern matching), or can be used to solve goals by narrowing (using unification), or by their pleasant combination [19]: eager simplification interspersed between outermost narrowing steps. The goal $z @ (b:\varepsilon) = a:b:\varepsilon$, for example, where z is the existential "logic" variable being solved for, narrows to the subgoal $b:\varepsilon = a:b:\varepsilon$ (applying the first rule and assigning $z \mapsto \varepsilon$), which dies (for lack of applicable rule). The original goal also narrows to $x:(z' @ (b:\varepsilon)) = a:b:\varepsilon$ via the bottom-left rule (with $z \mapsto x:z'$), which narrows to $z' @ (b:\varepsilon) = b:\varepsilon$ ($x \mapsto a$), which narrows to $b:\varepsilon = b:\varepsilon$ ($z' \mapsto \varepsilon$), which simplifies to T.

Suppose we are interested in solving goals of that form, $z @ A = B$, where A and B are (fully instantiated) lists. As abstract states, we can take the goal patterns $\{z @ A = B,\ x:(z @ A) = a:B\}$, $\{A = B\}$, and $\{T, F\}$, where a can be any atom. As we just saw, a goal of the form $z @ A = B$ can narrow to one of the forms $A = B$ and $x:(z' @ A) = B$. In the second event, if $B = \varepsilon$, then the goal simplifies to F; otherwise, it is of the form $x:(z @ A) = a:B$. For the latter goal, we have $x:(z @ A) = a:B \rightsquigarrow z @ A = B$. All told, the possible abstract transitions are



Since there are self-loops for the top two pattern sets, a proof of termination only requires showing that with each trip around these loops some measure

decreases. For the right loop, we measure a goal $A = B$ by the list B under the sublist (or list length) ordering. For the left loop, we first look at its pattern [with $z@A = B \ll x:(z@A) = a:B$], and, for like patterns, take B. □

In general, consider an infinite computation $s_0 \rightsquigarrow s_1 \rightsquigarrow \cdots$, and let τ assign one of finitely many colors to each state. By the Pigeonhole Principle, infinitely many states must share the same color. Hence, there must be a subcomputation $s_i \rightsquigarrow^+ s_j$ $(i < j)$ for which $\tau(s_i) = \tau(s_j)$. So if we show the impossibility of this happening infinitely often for any color, then we have precluded having infinite computations altogether.

Rather than just coloring vertices (states) of the transition graph, it is even better to also color its edges and paths: each subcomputation $s_i \rightsquigarrow^+ s_j$ $(i < j)$ is assigned one of a finite palette A of colors. Then, by (a simple case of) the infinite version of Ramsey's Theorem, there is an infinite (noncontiguous) subsequence of states $s_{i_1} \rightsquigarrow^+ s_{i_2} \rightsquigarrow^+ \cdots$, such that every one of its subcomputations $s_{i_j} \rightsquigarrow^+ s_{i_k}$ has the identical color $a \in A$ [and also $\tau(s_{i_j}) = \tau(s_{i_k})$]. So, to preclude nontermination, we need only show every such cyclical subcomputation decreases in some well-founded sense.

As shown in [16, Lemma 3.1], this fact can be applied to the call tree of a logic program. (See also the discussion in [7].) This leads to the *query-mapping method* of Sagiv [38, 16] and to similar techniques [8, 33].

3 Formalism

Let F be some vocabulary of (ranked) function symbols, and T the set of (ground) terms built from them. A *flat context* ℓ is a term of the form $f(t_1, \ldots, t_{i-1}, \Box, t_{i+1}, \ldots, t_n)$, where $f \in F$ is a function symbol of arity $n > 0$, the t_i are any terms, and $\Box$ is a special symbol denoting a "hole". If ℓ is such a flat context and t a term (or context), then by $\ell[t]$ we denote the term (or context) $f(t_1, \ldots, t_{i-1}, t, t_{i+1}, \ldots, t_n)$. We will view ℓ also as the binary relation $\{\langle t, \ell[t]\rangle \mid t \in T\}$, mapping a term t to the term $\ell[t]$, containing t as its immediate subterm. The inverse of flat ℓ, with its hole at position i, is the projection π_i. Let L be the set of all flat contexts (for some vocabulary), and $\Pi = L^-$, the set of all projections.

A *context* c is just an element of L^*, that is, the relation between any term t and some particular superterm $c[t]$ containing t where c's hole was. It has the shape of a "teepee", a term minus a subterm, so may be represented by a term $c[\Box]$ with one hole. Let $C = L^* \subseteq T \times T$ denote all contexts; put another way, C is just the subterm relation $\trianglelefteq$. Its inverse $\trianglerighteq$ is the superterm relation and its strict part $\triangleright$ is proper superterm.

A *rewrite system* is a set of rewrite rules, each of the form $l \to r$, where l and r are (first-order) terms. Rules are used to compute by replacing a subterm of a term t that matches the left-side pattern l with the corresponding instance of the right side r. For a rewrite system R, viewed as a binary relation (set of pairs of terms), we will use the notation $\propto_R$ to signify all its ground instances (a set

of pairs of ground terms), and $\to_R$ for the associated rewrite relation (also on ground terms). The latter is the relevant transition relation.

Composition of relations will be indicated by juxtaposition. If S and R are binary relations on terms, then by $S[R]$ we denote the composite relation:

$$S[R] \quad = \quad \bigcup_{x \in S} x^- R x \,,$$

which takes a backwards S-step before R, and then undoes that S-step. Let Γ be the set of all ground instantiations, where a ground instantiation γ is the relation $\langle t, t\gamma \rangle$, where $t\gamma$ is the term t with its variables instantiated as per γ. The inverse operation γ^{-1} is "generalization", which replaces subterms by variables. With this machinery in place, the *top-rewrite relation* (rule application) and *rewrite steps* (applying a rule at a subterm) are definable as follows:

$$\begin{aligned} \propto_R \quad &= \quad \Gamma[R] \\ \to_R \quad &= \quad C[\propto_R] \,. \end{aligned}$$

Thus,

$$\begin{aligned} \propto_R \;&=\; \{l\gamma \propto r\gamma \mid l \to r \in R,\ \gamma \in \Gamma\} \\ \to_R \;&=\; \{c[l\gamma] \to c[r\gamma] \mid l \to r \in R,\ \gamma \in \Gamma,\ c \in C\} \,. \end{aligned}$$

Of course,

$$\propto_R \quad \subseteq \quad \to_R \ . \tag{2}$$

Since we will rarely talk about more than one system at a time, we will often forget subscripts.

Two properties of relations are central to the termination tools we describe:

Mono($\sqsupset$):	$\sqsupset \ell \ \subseteq \ \ell \sqsupset$
Harmony($\sqsupset$,$\gg$):	$\sqsupset \ell \ \subseteq \ \ell \gg$

where $\sqsupset$ and $\gg$ are arbitrary binary relations over terms and ℓ is an arbitrary flat context. See the diagrams in Fig. 1. Mono is "monotonicity", a.k.a. the "replacement property" (relations are inherited by superterms). Rewriting is monotonic:

$$\text{Mono}(\to) \,. \tag{3}$$

Harmony means that

$$s \sqsupset t \quad \Rightarrow \quad \ell[s] \gg \ell[t]$$

for all $\ell \in L$ and $s, t \in T$ [2]. So, monotonicity of a relation is self-harmony:

$$\text{Harmony}(\gg, \gg) \Leftrightarrow \text{Mono}(\gg) \,. \tag{4}$$

[2] Harmony is called "quasi-monotonicity of $\gg$ with respect to $\sqsupset$" in [5].

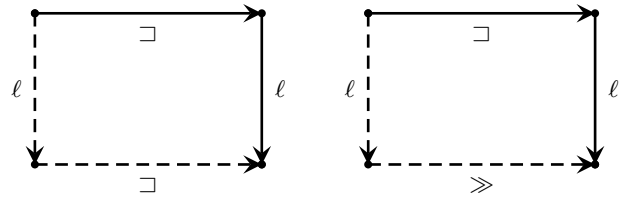


Fig. 1. Monotonicity and harmony.

Clearly:

$$\text{Mono}(\gg) \wedge \text{Mono}(\sqsupset) \Rightarrow \text{Mono}(\gg \cap \sqsupset) \tag{5}$$
$$\text{Mono}(\gg) \Rightarrow \text{Mono}(\rightarrow \cap \gg) \tag{6}$$
$$\text{Mono}(\gg \cap \sqsupset) \Rightarrow \text{Harmony}(\gg \cap \sqsupset, \gg) \tag{7}$$
$$\text{Harmony}(\not\mapsto, \gg) \wedge \text{Harmony}(\not\mapsto, \sqsupset) \Leftrightarrow \text{Harmony}(\not\mapsto, \gg \cap \sqsupset)\,, \tag{8}$$

for any relations $\gg, \sqsupset, \not\mapsto$.

All such relations refer to ground terms. They may be lifted to free terms in the standard manner: Demanding that $u \gg v$, for terms u and v with free variables, means that $u\gamma \gg v\gamma$ for all substitutions γ of ground terms for those variables.

Let $\rightarrow$ be a rewrite relation, the termination of which is in question and $\propto$, its rule application relation. To prove $\text{SN}(\rightarrow)$, we make use of various combinations of conditions involving two basic properties:

$$\begin{array}{rcl} \text{Rule}(\sqsupset): & \propto & \subseteq \quad \sqsupset \\ \text{Reduce}(\sqsupset): & \rightarrow & \subseteq \quad \sqsupset \end{array}$$

The following relations are all easy:

$$\text{Rule}(\propto) \tag{9}$$
$$\text{Rule}(\rightarrow) \tag{10}$$
$$\text{Reduce}(\rightarrow) \tag{11}$$
$$\text{Rule}(\gg) \wedge \text{Rule}(\sqsupset) \Leftrightarrow \text{Rule}(\gg \cap \sqsupset) \tag{12}$$
$$\text{Reduce}(\gg) \wedge \text{Reduce}(\sqsupset) \Leftrightarrow \text{Reduce}(\gg \cap \sqsupset) \tag{13}$$
$$\text{Rule}(\gg) \wedge \text{Mono}(\rightarrow \cap \gg) \Leftrightarrow \text{Reduce}(\gg) \tag{14}$$
$$\text{Rule}(\gg) \wedge \text{Harmony}(\rightarrow \cap \gg, \gg) \Leftrightarrow \text{Reduce}(\gg) \tag{15}$$
$$\text{Reduce}(\gg) \wedge \text{Harmony}(\gg, \sqsupset) \Rightarrow \text{Mono}(\rightarrow \cap \sqsupset) \tag{16}$$

Statements (14,15) are by induction on term structure.

As described in Section 2, one can (always) prove termination by showing that $\rightarrow$ is contained in some terminating relation $\succ$. Accordingly, the first, and most general, method employed in termination arguments is[3]:

[3] These two methods are usually phrased in terms of well-founded orderings, rather than terminating binary relations.

Obvious ($\not\succ$)**:** Reduce($\not\succ$).

More precisely, this means

$$\mathrm{SN}(\not\succ) \wedge \mathrm{Reduce}(\not\succ) \quad \Rightarrow \quad \mathrm{SN}(\rightarrow)\,,$$

where SN($\not\succ$) makes explicit the assumption that $\not\succ$ is terminating.

Since Reduce refers to the "global" rewriting of any term at any redex, it is easier to deal with Rule, which is a "local" condition on rules only, and impose monotonicity on the relation:[3]

Standard ($\not\succ$) [32]**:** Rule($\not\succ$), Mono($\not\succ$).

4 Harmonious Methods

The following properties can be used to show that the union of two relations is terminating:

Commute($\sqsupset$,$\gg$):	$\sqsupset \gg \;\subseteq\; \gg^+ \sqsupset^*$
Compat($\sqsupset$,$\gg$):	$\sqsupset \gg \;\subseteq\; \sqsupset$

For example [3]:

$$\mathrm{Commute}(\not\succ, \not\succ') \wedge \mathrm{SN}(\not\succ) \wedge \mathrm{SN}(\not\succ') \quad \Rightarrow \quad \mathrm{SN}(\not\succ \cup \not\succ')\,.$$

Obviously, if $>$ is the strict part of a quasi-order $\geq$, then:

$$\mathrm{Compat}(>, \geq)\,. \tag{17}$$

Requiring that the relation $\not\succ$ be monotonic, as in the **Standard** method of the previous section, may be too restrictive; all that is actually needed is that it be harmonious with rewriting:

Kamin & Lévy ($\not\succ$) [29]**:** Rule($\not\succ$), Harmony($\rightarrow \cap \not\succ, \not\succ$).

When terms are larger than their subterms, monotonicity can be weakened to refer instead to a non-strict quasi-ordering $\precsim$ ($\prec$ is its strict part)[4]:

Quasi-Simplification Ordering ($\precsim$) [12]**:** Rule($\prec$), Sub($\precsim$), Mono($\precsim$).

where a binary relation has the *subterm property* (Sub) if it contains the superterm relation ($\trianglerighteq$):

Sub($\sqsupset$):	$\trianglerighteq \;\subseteq\; \sqsupset$

[4] We are ignoring the fact that the subterm and monotonicity conditions for quasi-simplification orderings obviate the separate need to show that the ordering is well-founded [12].

By definition:

$$\mathrm{Sub}(\trianglerighteq)\,. \tag{18}$$

As illustrated in [1], the fact that the **Quasi-Simplification Ordering** method, as well as the ones developed in Section 6 below, do not require $\mathrm{Mono}(\succ)$ means that selected function symbols and argument positions can be ignored completely (cf. the use of weak monotonicity in [14]). See the example in Section 8.

As before, what is actually needed is that the relation $\succsim$ be monotonic when restricted to pairs that are related by rewriting:

Subterm $(\succsim)$ [13]**:** $\mathrm{Rule}(\succ)$, $\mathrm{Sub}(\succsim)$, $\mathrm{Harmony}(\rightarrow \cap \succsim, \succsim)$.

Furthermore, the proof of this method in [13] is based on the requirements:

Right $(\succsim)$**:** $\mathrm{Right}(\succ)$, $\mathrm{Harmony}(\rightarrow \cap \succsim, \succsim)$.

Here, we are using the property

$$\mathrm{Right}(\sqsupset)\colon \quad \propto \trianglerighteq \;\subseteq\; \sqsupset$$

meaning that left-hand sides are bigger than all right-side subterms. The composite relation $\propto \trianglerighteq$ comprises the "dependency pairs" of [1]. Trivially:

$$\mathrm{Right}(\gg) \Rightarrow \mathrm{Rule}(\gg)\,, \tag{19}$$

In the following formulations, $\not\succ$ is terminating, but $\gg$ need not be. If one relation is monotonic, then the other should live harmoniously with it:

Harmony $(\not\succ, \gg)$**:** $\mathrm{Rule}(\not\succ \cap \gg)$, $\mathrm{Mono}(\gg)$, $\mathrm{Harmony}(\not\succ \cap \gg, \not\succ)$.

The *semantic path ordering* of [29] (see [13]) is a special case, using $\rightarrow$ for $\gg$, for which only the conditions of the **Kamin & Lévy** method need be shown (see Lemma 5 below).

Monotonicity $(\not\succ, \gg)$**:** $\mathrm{Rule}(\not\succ \cap \gg)$, $\mathrm{Mono}(\gg)$, $\mathrm{Harmony}(\gg, \not\succ)$.

The *monotonic semantic path ordering* of [5] uses a semantic path ordering for $\not\succ$, demanding $\mathrm{Rule}(\gg^* \cap \succ)$ and $\mathrm{Harmony}(\gg, \gg^* \cap \succsim)$, in the final analysis.

The correctness of these methods is proved in Section 7 below. A more complicated alternative is

Weak $(\not\succ, \gg)$**:** $\mathrm{Right}(\not\succ)$, $\mathrm{Harmony}(\propto \cap \not\succ, \gg)$, $\mathrm{Harmony}(\rightarrow \cap \gg, \gg)$, $\mathrm{Commute}(\gg, \not\succ)$.

5 Constrictions

The goal we haven been pursuing is to establish finiteness of sequences of transitions, beginning in any valid state. It will be convenient to define the set (monadic

predicate) $\rightsquigarrow^\infty$ of elements that can initiate infinite chains in a relation $\rightsquigarrow$, as follows:

$$\rightsquigarrow^\infty \quad = \quad \{s_0 \,|\, \exists\, s_1, s_2, \ldots . \forall j.\, s_j \rightsquigarrow s_{j+1}\}\,.$$

Thus, $\rightsquigarrow^\infty$ is the set of "immortal" initial states. With this notation in mind, termination of a transition system, $\mathrm{SN}(\rightsquigarrow)$, is emptiness of $\rightsquigarrow^\infty$ (that is, denial of immortality):

$$\mathrm{SN}(\gg) \quad \Leftrightarrow \quad \gg^\infty = \emptyset\,.$$

For rewriting, since contexts and rewrites commute ($\rhd \rightarrow \subseteq \rightarrow \rhd$), meaning that if a subterm can be rewritten, so can the whole term, we have [15]:

$$\rightarrow^\infty \quad = \quad (\rightarrow \cup \rhd)^\infty\,. \tag{20}$$

Two important observations on nontermination of rewriting can be made:

- If a system is nonterminating, then there is an infinite derivation with at least one redex at the top of a term. In fact, any immortal term has a subterm initiating such a derivation:

$$\rightarrow^\infty \quad \subseteq \quad \trianglerighteq \rightarrow^* \propto \rightarrow^\infty\,. \tag{21}$$

  See, for example, [11],[12, p. 287].
- If a system is nonterminating, then there is an infinite derivation in which all proper subterms of every redex are mortal. By *mortal*, we mean that it initiates finite derivations only. Let's call such redexes *critical*. Rewriting at critical redexes yields a "constricting" derivation in the sense of Plaisted [37].

For given rewrite relation $\rightarrow$, let T_∞ be its immortal terms ($T_\infty = \rightarrow^\infty$), $T_{<\infty}$ the mortal ones ($T \setminus T_\infty$), and $T_\circ = T_\infty \cap L[T_{<\infty}]$ the critical terms (immortal terms all of whose subterms are mortal). To facilitate composition, it will be convenient to associate a binary relation $P?$ with monadic predicates P:

$$P? \quad = \quad \{\langle x, x\rangle \,|\, x \in P\},$$

the identity relation restricted to the domain of P. Let $\propto_\circ = T_\circ? \propto$ be a constricting rewrite step (at a critical redex) and $\rightarrow_\circ = C[\propto_\circ]$ be the corresponding rewrite relation. The following facts hold:

$$\rightarrow T_\infty? \subseteq T_\infty? \tag{22}$$

$$\trianglerighteq T_\infty? \subseteq T_\infty? \tag{23}$$

$$\rightarrow_\circ T_{<\infty}? \subseteq T_{<\infty}?\,. \tag{24}$$

In words: mortals remain mortal after rewriting; mortals beget mortal subterms; immortals remain immortal after constriction.

Let a non-top constriction be denoted $\rightarrow_B = \Pi[\rightarrow_\circ]$. Let $\rightarrow_D$ be a top constriction, followed by a sequence of projections, followed by a sequence of non-top

constrictions: $\to_D = \propto_\circ \trianglerighteq \to_B^*$. Considering constrictions suffices for termination [37][5]:

$$\to^\infty \quad = \quad \to_\circ^\infty \quad = \quad \trianglerighteq \to_B^* \to_D^\infty \ . \tag{25}$$

Thus, we aim to show only that $\to_D$ is terminating. To prove this one can use compatible well-founded orderings $\succ$ and $\succsim'$ such that $\propto_\circ \trianglerighteq \subseteq \succ$ and $\to_B \subseteq \succsim'$. This is the basis for the various dependency-pair methods[6]. Since constricting redexes don't have immortal children, termination follows even if the condition $\propto_\circ \trianglerighteq \subseteq \succ$ is weakened to $\propto_\circ \trianglerighteq \subseteq (\succ \cup \rhd)$ [7].

Therefore, we can restrict the following two properties of rewrite sequences to refer only to constrictions:

Subrule($\sqsupset$):	$\propto_\circ \subseteq \sqsupset \cup \rhd$
Depend($\sqsupset$):	$\propto_\circ \trianglerighteq \subseteq \sqsupset \cup \rhd$

where:

$$\text{Depend}(\gg) \quad \Rightarrow \quad \text{Subrule}(\gg) \tag{26}$$

$$\text{Rule}(\sqsupset) \quad \Rightarrow \quad \text{Subrule}(\sqsupset) \tag{27}$$

$$\text{Subrule}(\gg) \wedge \text{Sub}(\sqsupset) \wedge \text{Compat}(\gg, \sqsupset) \quad \Rightarrow \quad \text{Depend}(\gg) \tag{28}$$

$$\text{Subrule}(\succ) \wedge \text{Sub}(\succsim) \quad \Rightarrow \quad \text{Depend}(\succ)\,. \tag{29}$$

Statement (29) follows from (17,28).

All this establishes the correctness of the following method:

Basic ($\succsim$)**:** Depend($\succ$), Reduce($\succsim$).

6 Dependency Methods

In what follows, let $\succsim$ and $\succsim'$ be arbitrary well-founded quasi-orderings, and $\succ$ and $\succ'$ their associated strict well-founded partial orderings.

The dependency-pair method [1] of proving termination of rewriting takes into account the possible transitions from one critical term to the next ($\to_D$) in an infinite rewriting derivation. Using the notations of the previous sections, we have two additional variations on this theme:

[5] The idea is reminiscent of Tait's reducibility predicate [41]. Constricting derivations were also used by [24] to argue about the sufficiency of "forward closures" for proving termination of "overlaying" systems (see [13]).

[6] Another way to understand the dependency method is to transform ordinary rewrite rules into equational Horn clauses (i.e. conditional rewrite rules; see [42, Sect. 3.5]). A rule $l \to f(r_1, \ldots, r_n)$ has the same operational behavior as the flat, conditional rule $r_1 \to^* y_1, \ldots, r_n \to^* y_n : \ l \to f(y_1, \ldots, y_n)$. Using the *decreasing method* [18] for operational termination requires that $l \succ r_1, \ldots, r_n, f(y_1, \ldots, y_n)$ for all y_i such that $r_i \to^* y_i$.

[7] For proving "call-by-value" termination, or for (locally-confluent overlay) rewrite systems for which innermost derivations are the least likely to terminate, the conditions can be simplified, since $\to_B$ steps cannot preceded a $\to_D$ step. See [1].

Main ($\succsim$) [1]**:** Depend($\succ$), Mono($\to \cap \succsim$).

Intermediate ($\succsim, \succ'$)**:** Depend($\succ'$), Reduce($\succsim$), Compat($\succ'$,$\succsim$).

More specific techniques may be derived from these. For example, the dependency method of [20] may be expressed as follows[8]:

Harmonious Dependency ($\succsim, \succsim'$) [20]**:** Depend($\succ'$), Rule($\succsim$), Mono($\succsim$), Harmony($\succsim$,$\succsim'$).

Let $\widehat{F}$ be a mirror image of F: $\widehat{F} = \{\widehat{g} | \, g \in F\}$. Denote by $\widehat{s}$ the term $s = f(u_1, \ldots, u_n)$ with root symbol $f \in F$ replaced by its mirror image $\widehat{f} \in \widehat{F}$, that is, $\widehat{s} = \widehat{f}(u_1, \ldots, u_n)$. Let $\widehat{T}$ be T's image under $\widehat{\cdot}$. If $\succ$ is a partial ordering of $\widehat{T}$, define another partial ordering $\widehat{\succ}$ as $u \mathrel{\widehat{\succ}} v$, for terms $u, v \in T$, when $\widehat{u} \succ \widehat{v}$. The original dependency-pair method is approximately[8]:

Dependency Pairs ($\succsim$) [1]**:** Depend($\widehat{\succ}$), Rule($\succsim$), Mono($\succsim$).

Here, Mono applies to both hatted ($f \in \widehat{F}$) and bareheaded ($f \in F$) terms, hence implies Harmony.

A more recent version of the dependency-pair method is essentially:

Variant ($\succsim, \succ'$) [25]**:** Depend($\succ'$), Rule($\succsim$), Mono($\succsim$), Compat($\succ', \succsim$).

7 Method Dependencies

Entailments between the methods are depicted in Fig. 2. The following series of lemmata justify the figure, by establishing dependencies between the different methods and their correctness. As a starter, take:

Lemma 1 ([32]). Obvious ($\succcurlyeq$) $\Rightarrow$ **Standard** ($\succcurlyeq$) .

In general, such an implication $\mathbf{M} \Rightarrow \mathbf{M}'$ means that method $\mathbf{M}'$ is a special case of method $\mathbf{M}$. To prove the implication, viz. that correctness of the antecedent method $\mathbf{M}$ implies correctness of the consequent $\mathbf{M}'$, one shows that the *requirements* for $\mathbf{M}'$ imply the requirements for $\mathbf{M}$. This includes the requirement that any terminating relation(s) or well-founded ordering(s) used by $\mathbf{M}$ should be a derivative of those used by $\mathbf{M}'$.

Suppose method $\mathbf{M}(\succcurlyeq)$ has requirements C and $\mathbf{M}'(\succcurlyeq')$ requires C'. Then, to claim $\mathbf{M} \Rightarrow \mathbf{M}'$ one needs to establish

$$C' \wedge \mathrm{SN}(\succcurlyeq') \wedge \neg\mathrm{SN}(\to) \quad \Rightarrow \quad C \wedge \mathrm{SN}(\succcurlyeq) \, .$$

In particular, to prove Lemma 1, we show that the conditions for the latter imply the conditions for the former:

[8] The dependency-pair methods of [1, 20, 25] exclude only variables u in r, rather than all left-side proper subterms, from the requirement that $l \succ' u$ of Depend($\succ'$) or $l \mathrel{\widehat{\succ}} u$ of Depend($\widehat{\succ}$). This can make a practical difference [35].

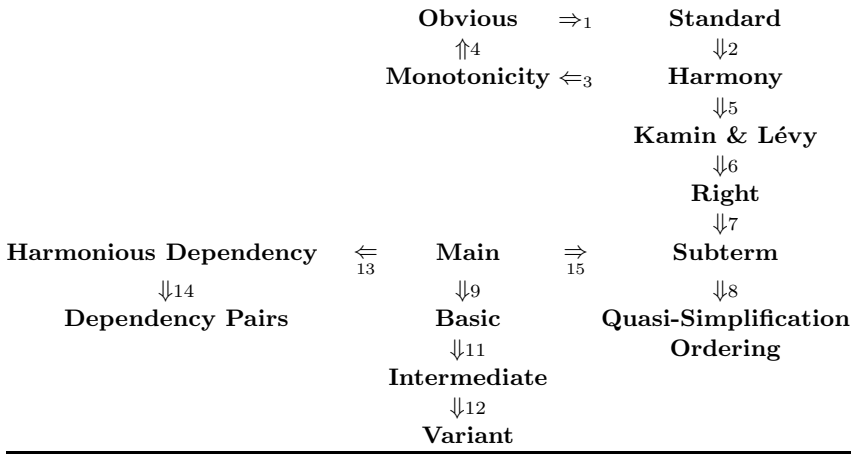


Fig. 2. Dependencies of methods. (Numbers refer to lemmata.)

Proof. By (6,14),

$$\mathrm{SN}(\succ) \wedge \mathrm{Rule}(\succ) \wedge \mathrm{Mono}(\succ) \quad \Rightarrow \quad \mathrm{SN}(\succ) \wedge \mathrm{Reduce}(\succ)\,. \qquad \square$$

Lemma 2. Standard $(\succ \cap \gg) \Rightarrow$ **Harmony** $(\succ,\gg)$.

Here, the implication means that correctness of **Harmony**, using the terminating relation $\succ$, follows from the **Standard** method, using the restricted relation $\rightarrow \cap \succ$ (which is also terminating, by Eq. 1).

Lemma 3. Harmony $(\succ,\gg) \Rightarrow$ **Monotonicity** $(\succ,\gg)$.

This circle of dependencies can be closed:

Lemma 4. Monotonicity $(\succ,\rightarrow) \Rightarrow$ **Obvious** $(\succ)$.

The correctness of **Obvious** – using the ordering $\succ$, follows from the **Monotonicity** method – using the monotonic rewrite relation $\rightarrow$ for $\gg$.

Lemma 5. Harmony $(\succ,\rightarrow) \Rightarrow$ **Kamin & Lévy** $(\succ)$.

Proof. We need

$$\begin{aligned} &\mathrm{Rule}(\succ) \wedge \mathrm{Harmony}(\rightarrow \cap \succ, \succ) \\ \Rightarrow \quad &\mathrm{Rule}(\rightarrow \cap \succ) \wedge \mathrm{Harmony}(\rightarrow \cap \succ, \succ) \wedge \mathrm{Mono}(\rightarrow)\,, \end{aligned}$$

which follows from (10,3,12). $\square$

We have split the argument of [12] for the **Quasi-Simplification Ordering** method into three parts, with the **Right** and **Subterm** methods as intermediate stages:

Lemma 6. Kamin & Lévy ($\succ^\omega$) ⇒ **Right** ($\succsim$), where $s \succ^\omega t$ is the well-founded multiset extension [17] of $\succ$ to the bags of all the subterms of s and t.

Proof. We need to show that

$$\mathrm{Right}(\succ) \wedge \mathrm{Harmony}(\to \cap \succsim, \succsim) \quad \Rightarrow \quad \mathrm{Rule}(\succ^\omega) \wedge \mathrm{Mono}(\to \cap \succ^\omega)\,.$$

For $u \propto v$, Right($\succ$) means that a bag containing u alone is strictly greater than a bag with all of v's subterms. So, by the nature of the bag ordering (adding to a bag makes it bigger), Rule($\succ^\omega$) follows. If $u \succ^\omega w$, one only needs to know that $\ell[u] \succsim \ell[v]$ for $\ell[u] \succ^\omega \ell[v]$ to hold, which we have thanks to Harmony (and Eqs. 15, 19), as long as $u \to v$. □

Lemma 7. Right ($\succsim$) ⇒ **Subterm** ($\succsim$).

Proof. To see that

$$\mathrm{Rule}(\succ) \wedge \mathrm{Sub}(\succsim) \wedge \mathrm{Harmony}(\to \cap \succsim, \succsim) \Rightarrow \mathrm{Right}(\succ) \wedge \mathrm{Harmony}(\to \cap \succsim, \succsim)\,,$$

note that Right($\succ$) follows from Rule($\succ$), Sub($\succsim$), and the compatibility of a quasi-ordering with its strict part (17). □

Lemma 8. Subterm ($\succsim$) ⇒ **Quasi-Simplification Ordering** ($\succsim$).

Proof. Harmony($\to \cap \succsim, \succsim$) follows from Mono($\succsim$) and (3,5,7). □

Turning to the dependency methods:

Lemma 9 ([1]). Main ($\succsim$) ⇒ **Basic** ($\succsim$).

Proof. Follows from (14). □

Lemma 10. Basic ($\succsim$) ⇒ **Main** ($\succsim$), for constrictions $\to_\circ$.

Proof. From Subrule($\succsim$) and Mono($\to_\circ \cap \succsim$), one can show Reduce($\succsim$) for constrictions. □

Lemma 11. Basic ($\succsim^*$) ⇒ **Intermediate** ($\succsim$,$\succ'$), where $\succsim^*$ symbolizes the transitive closure of $\succsim \cup \succ'$.

Proof. We need

$$\mathrm{Depend}(\succ') \wedge \mathrm{Reduce}(\succsim) \wedge \mathrm{Compat}(\succ', \succsim) \quad \Rightarrow \quad \mathrm{Depend}(\succ^*) \wedge \mathrm{Reduce}(\succsim^*)\,.$$

Note that $\succsim^*$ is well-founded on account of Compat($\succ', \succsim$). The rest is straightforward. □

Lemma 12. Intermediate ($\succsim$,$\succ'$) ⇒ **Variant** ($\succsim$,$\succ'$).

Proof. By (6,14). □

Lemma 13. Main ($\succsim'$) ⇒ **Harmonious Dependency** ($\succsim$,$\succsim'$).

Proof. By (14,16). □

Lemma 14. Harmonious Dependency $(\succsim,\widehat{\succsim})$ ⇒ **Dependency Pairs** $(\succsim)$.

Proof. Harmony$(\succsim,\widehat{\succsim})$ holds trivially. □

The linchpin step is:

Lemma 15. Main $(\succsim)$ ⇒ **Subterm** $(\succsim)$.

Proof. This follows from (27,29). □

8 Illustrations

We conclude with an example.

Example 2. Consider the seven rules:

$$\begin{array}{lcl@{\qquad}rcl} 0+x & \to & x & s(x)+y & \to & s(x+y) \\ 0-x & \to & 0 & s(x)-s(y) & \to & x-y \\ 0\times x & \to & 0 & s(x)\times y & \to & y+(x\times y) \\ & & & s(x)\times(y+z) & \to & x\times(y+z)+(y+z) \end{array}$$

To prove termination by the **Harmonious Dependency** method, we can use the style of the general path ordering [14, 23], which allows one to compare terms by comparing a mix of precedences, interpretations, and *selected* arguments. Take a "natural" interpretation $[\![\cdot]\!]$ to show that $s \to t$ preserves the value of the interpretation (this natural equivalence of value will be $\succsim$), and for $\succ'$ use a termination function based on an interpretation $\{\!\{\cdot\}\!\}$, where:

$$\begin{array}{rcl@{\qquad}rcl} [\![0]\!] & = & 0 & & & \\ [\![s(x)]\!] & = & [\![x]\!]+1 & \{\!\{s(x)\}\!\} & = & \langle s,0,0\rangle \\ [\![x+y]\!] & = & [\![x]\!]+[\![y]\!] & \{\!\{x+y\}\!\} & = & \langle +,[\![y]\!],[\![x]\!]\rangle \\ [\![x-y]\!] & = & [\![x]\!]-\min([\![x]\!],[\![y]\!]) & \{\!\{x-y\}\!\} & = & \langle -,[\![x]\!],0\rangle \\ [\![x\times y]\!] & = & [\![x]\!]\cdot[\![y]\!] & \{\!\{x\times y\}\!\} & = & \langle \times,[\![x]\!],0\rangle\,, \end{array}$$

with triples ordered lexicographically. The precedence $\times > + > - > s$ is the abstraction[9]. Terms in the same abstract class are compared by the remaining components, which express the recursion scheme. Harmony follows from the use of $[\![\cdot]\!]$ in $\{\!\{\cdot\}\!\}$. Now, one shows the following inequalities for constricting transitions:

$$\begin{array}{rcl} \{\!\{s(x)+y\}\!\} & > & \{\!\{s(x+y)\}\!\},\ \{\!\{x+y\}\!\} \\ \{\!\{s(x)-s(y)\}\!\} & > & \{\!\{x-y\}\!\} \\ \{\!\{s(x)\times y\}\!\} & > & \{\!\{y+(x\times y)\}\!\},\ \{\!\{x\times y\}\!\} \\ \{\!\{s(x)\times(y+z)\}\!\} & > & \{\!\{x\times(y+z)+(y+z)\}\!\}\,,\{\!\{x\times(y+z)\}\!\}\,. \end{array}$$ □

[9] There is no need to explicitly exclude the case that u is headed by a constructor (as done in [1]). One simply makes terms headed by constructors smaller than terms headed by defined symbols, which is why s is minimal. The constructor 0 need not be interpreted, since it appears on the left of every rule in which it appears at all.

Rather than a simple precedence, one can devise a "pattern-based" ordering. Patterns that can never have a top redex are made minimal in the surface ordering, and safely ignored. Symbolic inductive techniques may be used to discover patterns that generalize terms in computations.

References

1. Thomas Arts and Jürgen Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.
2. Thomas Arts and Hans Zantema. Termination of logic programs using semantic unification. In M. Proietti, editor, *Proceedings of the Fifth International Workshop on Logic Program Synthesis and Transformation*, volume 1048 of *Lecture Notes in Computer Science*, pages 219–233, Berlin, 1996. Springer-Verlag.
3. Leo Bachmair and Nachum Dershowitz. Commutation, transformation, and termination. In J. H. Siekmann, editor, *Proceedings of the Eighth International Conference on Automated Deduction (Oxford, England)*, volume 230 of *Lecture Notes in Computer Science*, pages 5–20, Berlin, July 1986. Springer-Verlag.
4. Françoise Bellegarde and Pierre Lescanne. Termination by completion. *Applied Algebra on Engineering, Communication and Computer Science*, 1(2):79–96, 1990.
5. Cristina Borralleras, Maria Ferreira, and Albert Rubio. Complete monotonic semantic path orderings. In *Proceedings of the Seventeenth International Conference on Automated Deduction (Pittsburgh, PA)*, volume 1831 of *Lecture Notes in Artificial Intelligence*, pages 346–364. Springer-Verlag, June 2000.
6. Jacques Chabin and Pierre Réty. Narrowing directed by a graph of terms. In R. V. Book, editor, *Rewriting Techniques and Applications: Proceedings of the Fourth International Conference (RTA-91)*, pages 112–123. Springer, Berlin, 1991.
7. Michael Codish, Samir Genaim, Maurice Bruynooghe, John Gallagher, and Wim Vanhoof. One loop at a time. In *6th International Workshop on Termination (WST 2003)*, June 2003.
8. Michael Codish and Cohavit Taboch. A semantic basis for the termination analysis of logic programs. *J. Logic Programming*, 41(1):103–123, 1999.
9. Patrick M. Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.
10. Danny De Schreye and Stefaan Decorte. Termination of logic programs: The never-ending story. *J. Logic Programming*, 19/20:199–260, 1993.
11. Nachum Dershowitz. Termination of linear rewriting systems. In *Proceedings of the Eighth International Colloquium on Automata, Languages and Programming (Acre, Israel)*, volume 115 of *Lecture Notes in Computer Science*, pages 448–458, Berlin, July 1981. European Association of Theoretical Computer Science, Springer-Verlag.
12. Nachum Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17(3):279–301, March 1982.
13. Nachum Dershowitz. Termination of rewriting. *J. Symbolic Computation*, 3(1&2): 69–115, February/April 1987.
14. Nachum Dershowitz and Charles Hoot. Natural termination. *Theoretical Computer Science*, 142(2):179–207, May 1995.

15. Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Methods and Semantics, chapter 6, pages 243–320. North-Holland, Amsterdam, 1990.
16. Nachum Dershowitz, Naomi Lindenstrauss, Yehoshua Sagiv, and Alexander Serebrenik. A general framework for automatic termination analysis of logic programs. *Applicable Algebra in Engineering, Communication and Computing*, 12(1/2):117–156, 2001.
17. Nachum Dershowitz and Zohar Manna. Proving termination with multiset orderings. *Communications of the ACM*, 22(8):465–476, August 1979.
18. Nachum Dershowitz, Mitsuhiro Okada, and G. Sivakumar. Confluence of conditional rewrite systems. In S. Kaplan and J.-P. Jouannaud, editors, *Proceedings of the First International Workshop on Conditional Term Rewriting Systems (Orsay, France)*, volume 308 of *Lecture Notes in Computer Science*, pages 31–44, Berlin, July 1987. Springer-Verlag.
19. Nachum Dershowitz and David A. Plaisted. Equational programming. In J. E. Hayes, D. Michie, and J. Richards, editors, *Machine Intelligence 11: The logic and acquisition of knowledge*, chapter 2, pages 21–56. Oxford Press, Oxford, 1988.
20. Nachum Dershowitz and David A. Plaisted. Rewriting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 9, pages 535–610. Elsevier Science, 2001.
21. Nachum Dershowitz and G. Sivakumar. Goal-directed equation solving. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 166–170, St. Paul, MN, August 1988. AAAI.
22. Robert W. Floyd. Assigning meanings to programs. In *Proceedings of Symposia in Applied Mathematics, XIX: Mathematical Aspects of Computer Science*, pages 19–32, Providence, RI, 1967. American Mathematical Society.
23. Alfons Geser. An improved general path order. *Applicable Algebra in Engineering, Communication, and Computing*, 7(6):469–511, 1996.
24. Oliver Geupel. Overlap closures and termination of term rewriting systems. Report MIP-8922, Universität Passau, Passau, West Germany, July 1989.
25. Jürgen Giesl and Deepak Kapur. Dependency pairs for equational rewriting. In *Proceedings of the Twelfth International Conference on Rewriting Techniques and Applications (Utrecht, The Netherlands)*, volume 2051 of *Lecture Notes in Computer Science*, pages 93–107. Springer-Verlag, 2001.
26. David Gries. Is sometime ever better than alway? *ACM Transactions on Programming Languages and Systems*, 1(2):258–265, October 1979.
27. Michael Hanus. The integration of functions into logic programming: From theory to practice. *J. Logic Programming*, 19&20:583–628, 1994.
28. Jean-Marie Hullot. Canonical forms and unification. In R. Kowalski, editor, *Proceedings of the Fifth International Conference on Automated Deduction (Les Arcs, France)*, volume 87 of *Lecture Notes in Computer Science*, pages 318–334, Berlin, July 1980. Springer-Verlag.
29. Sam Kamin and Jean-Jacques Lévy. Two generalizations of the recursive path ordering, February 1980. Unpublished note, Department of Computer Science, University of Illinois, Urbana, IL.
Available at http://www.ens-lyon.fr / LIP / REWRITING / OLD_PUBLICATIONS_ON_TERMINATION / KAMIN_LEVY (viewed June 2004).
30. Shmuel M. Katz and Zohar Manna. A closer look at termination. *Acta Informatica*, 5(4):333–352, December 1975.

31. Laurie Kirby and Jeff Paris. Accessible independence results for Peano arithmetic. *Bulletin London Mathematical Society*, 14:285–293, 1982.
32. Dallas S. Lankford. On proving term rewriting systems are Noetherian. Memo MTP-3, Mathematics Department, Louisiana Tech. University, Ruston, LA, May 1979. Revised October 1979.
33. Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *Conference Record of the Twenty-Eighth Symposium on Principles of Programming Languages*, volume 36 (3), pages 81–92, London, UK, January 2001. ACM SIGPLAN Notices.
34. Zohar Manna. *Mathematical Theory of Computation*. McGraw-Hill, New York, 1974.
35. Aart Middeldorp, June 2003. Personal communication.
36. Enno Ohlebusch, Claus Claves, and Claude Marché. TALP: A tool for the termination analysis of logic programs. In P. Narendran and M. Rusinowitch, editors, *Proceedings of the Tenth International Conference on Rewriting Techniques and Applications (Trento, Italy)*, volume 1631 of *Lecture Notes in Computer Science*, pages 270–273, Berlin, July 1999. Springer-Verlag.
37. David A. Plaisted. Semantic confluence tests and completion methods. *Information and Control*, 65(2/3):182–215, 1985.
38. Yehoshua Sagiv. A termination test for logic programs. In *Logic Programming: Proceedings of the 1991 International Symposium*, pages 518–532, San Diego, CA, October 1991. MIT Press.
39. Alexander Serebrenik. *Termination Analysis of Logic Programs*. PhD thesis, Katholieke Universiteit Leuven, Leuven, Belgium, July 2003.
40. Michele Sintzoff. Calculating properties of programs by valuations on specific models. *Proceedings of the ACM Conference on Proving Assertions About Programs*, 7(1):203–207, January 1972.
41. William W. Tait. Intensional interpretations of functionals of finite type I. *Journalof Symbolic Logic*, 32(2):198–212, 1967.
42. "Terese" (M. Bezem, J. W. Klop and R. de Vrijer, eds.). *Term Rewriting Systems*. Cambridge University Press, 2002.
43. Alan M. Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69, Cambridge, England, 1949. University Mathematics Laboratory.
44. Kristof Vershaetse and Danny De Schreye. Deriving termination proofs for logic programs, using abstract procedures. In K. Furukawa, editor, *Proceeedings of the Eighth International Conference on Logic Programming*, pages 301–315, Cambridge, MA, 1991. The MIT Press.
45. Ben Wegbreit. Property extraction in well-founded property sets. *IEEE Transactions on Software Engineering*, SE-1(3):270–285, September 1975.