Term Rewriting with Logical Constraints*

Cynthia Kop[1] and Naoki Nishida[2]

[1] Department of Computer Science, University of Innsbruck
Technikerstraße 21a, 6020 Innsbruck, Austria
Cynthia.Kop@uibk.ac.at
[2] Graduate School of Information Science, Nagoya University
Furo-cho, Chikusa-ku, 4648603 Nagoya, Japan
nishida@is.nagoya-u.ac.jp

Abstract. In recent works on program analysis, transformations of various programming languages to term rewriting are used. In this setting, *constraints* appear naturally. Several definitions which combine rewriting with logical constraints, or with separate rules for integer functions, have been proposed. This paper seeks to unify and generalise these proposals.

Keywords: Term rewriting, Constraints, Integer rewriting.

1 Introduction

Given the prevalence of computer programs in modern society, an important role is reserved for program analysis. Such analysis could take the form of for instance *termination* ("will this program end eventually, regardless of user input?"), *productivity* ("will this program stay responsive during its run?") and *equivalence* ("will this optimised code return the same result as the original?").

In recent years, there have been several results which transform a real-world program analysis problem into a query on term rewriting systems (TRSs). Such transformations are used to analyse termination of small, constructed languages (e.g. [3]), but also real code, like Java Bytecode [13], Haskell [10], LLVM [5], or Prolog [15]. Similar transformations are used to analyse code equivalence in [4,9].

In these works, *constraints* arise naturally. Where traditional TRSs generally consider well-founded sets like the natural number, more dedicated techniques are necessary when dealing with for instance integers or floating point numbers. Unfortunately, standard techniques for analysing TRSs are not equipped to also handle constraints. While integers and constraints *can* be encoded in TRSs, the results are either hairy or infinite, and generally hard to handle.

For this reason, rewriting with native support for logical constraints over a model was proposed [9]. While the results from normal term rewriting do not immediately apply in this setting, the *ideas* extend easily, so dedicated results are derived without much effort. Thus, constrained TRSs give a useful abstraction layer for program analysis. Several alternative definitions of constrained rewriting, focused on *integer constraints*, have also been given, see e.g. [4,5,8].

* The research in this paper is supported by the Austrian Science Fund (FWF) international project I963 and the Japan Society for the Promotion of Science.

P. Fontaine, C. Ringeissen, and R.A. Schmidt (Eds.): FroCoS 2013, LNAI 8152, pp. 343–358, 2013.

Unfortunately, the various formalisms are incompatible; results from one style of constrained rewriting do not necessarily transfer to another. This is a shame, as e.g. the lemma generation method in [12] (there used to prove equivalence of C-functions) might otherwise be reused in termination proofs. Also, dependency pairs and graphs are introduced in each of [3,8,14]. Thus, a lot of time is spent on redoing the same work for slightly different settings. Moreover, there are things we cannot do easily with any of them, such as overflow-conscious analysis.

In this paper, we propose a new formalism which unifies existing definitions of constrained rewriting. This formalism seeks to be *general*: unlike most of its predecessors, we do not limit interest to the integers, since in the future we will likely want to analyse programs which involve for instance real numbers or bitvectors. Moreover, we do not restrict attention to one kind of analysis (e.g. only termination or function equivalence). This way, we may for instance use the same dependency pair framework both to analyse termination of Haskell, and as part of a proof that two Java programs produce the same result (as termination is an essential property in inductive equivalence proofs, see e.g. [4,12]).

Paper Setup. This paper is structured as follows. In Section 2 we consider some preliminaries: both mathematical notions and a definition of many-sorted term rewriting. In Sections 3 and 4 we introduce the *LCTRS* formalism, which is the main contribution of this work. In Section 5 we will study how LCTRSs relate to existing definitions. Finally, to demonstrate how existing analysis techniques extend, we will consider basic confluence and termination results in Section 6.

2 Preliminaries

2.1 Sets and Functions

We assume that the mathematical notion of a *set* is well-understood.

The *function space* from a set A to a set B, denoted $A \Longrightarrow B$, consists of all sets f of pairs $\langle a, b\rangle$ with $a \in A$ and $b \in B$, such that for all $a \in A$ there is a unique $b \in B$ with $\langle a, b\rangle \in f$. The $\Longrightarrow$ is considered right-associative, so $A \Longrightarrow B \Longrightarrow C$ is the function space $A \Longrightarrow (B \Longrightarrow C)$. We use functional notation: for $f \in A_1 \Longrightarrow \cdots \Longrightarrow A_n \Longrightarrow B$, $f(a_1, \ldots, a_n)$ denotes the unique b with $\langle a_1, \langle a_2, \ldots \langle a_n, b\rangle \ldots\rangle \in f$. When dealing with constraints, we need a notion of truth. To this end, we will often use the set $\mathbb{B} = \{\top, \bot\}$ of *booleans*.

Example 1. We consider the set $\mathbb{N}$ of natural numbers and the set $\mathbb{R}$ of real numbers. An example element of the function space $\mathbb{R} \Longrightarrow \mathbb{R} \Longrightarrow \mathbb{N}$ is the function $\lambda x \in \mathbb{R}, y \in \mathbb{R}.\ abs(\lceil x + y\rceil - 9)$. Here, the λ notation denotes function construction. The comparison relation $>$ on natural numbers is an element of $\mathbb{N} \Longrightarrow \mathbb{N} \Longrightarrow \mathbb{B}$, and can also be denoted in extended form, $\lambda x \in \mathbb{N}, y \in \mathbb{N}.\ x > y$.

2.2 Many-Sorted Term Rewriting Systems

Next, we consider term rewriting. In constrained rewriting, *types* like integers and booleans appear naturally. Since we have, at present, little reason to introduce function types, let us consider *many-sorted TRSs*.

Sorts and Signature. We assume given a set $\mathcal{S}$ of *sorts* and a set $\mathcal{V}$ of *variables*. A *signature* Σ is a set of *function symbols* f, each equipped with a *sort declaration* of the form $[\iota_1 \times \ldots \times \iota_n] \Rightarrow \kappa$ with all ι_i and κ sorts. A *variable environment* is a set Γ of variable : sort pairs.

Terms. Fixing a signature Σ, a *term* is any expression s built from function symbols in Σ, variables, commas and parentheses, such that $\Gamma \vdash s : \iota$ can be derived for some environment Γ and sort ι, using the following inference rules:

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

For any non-variable term s, there is a unique sort ι such that $\Gamma \vdash s : \iota$ for some Γ; we say that ι is the sort of s. The set of terms over Σ and $\mathcal{V}$ is denoted $\mathcal{T}erms(\Sigma, \mathcal{V})$. $Var(s)$ is the set of variables in s. A term s is *ground* if $Var(s) = \emptyset$.

Contexts and Substitution. Fixing an environment Γ, a *substitution* is a mapping $[x_1 := s_1, \ldots, x_k := s_k]$ from variables to terms, with $\{x_1 : \iota_1, \ldots, x_k : \iota_k\} \subseteq \Gamma$ and where $\Gamma \vdash s_1 : \iota_1, \ldots, \Gamma \vdash s_k : \iota_k$. The result $s\gamma$ of applying a substitution γ on a term s, is s with all occurrences of any x_i replaced by s_i. A *context* C is a term with zero or more special variables: $\Box_1, \ldots, \Box_n$, each occurring once. If $\Gamma \cup \{\Box_1 : \iota_1, \ldots, \Box_n : \iota_n\} \vdash C : \kappa$, and also $\Gamma \vdash s_i : \iota_i$ for all i, then we define the term $C[s_1, \ldots, s_n]$ as C with each $\Box_i$ replaced by the corresponding s_i.

Rules and Rewriting. In a many-sorted TRS (without constraints!) rules are pairs $l \to r$ where l and r are terms, l is not a variable and $Var(r) \subseteq Var(l)$; moreover, l and r must have the same sort. A (finite or infinite) set of rules $\mathcal{R}$ induces a rewrite relation $\to_{\mathcal{R}}$ on the set of terms by the following inference rule:

$$C[l\gamma] \to_{\mathcal{R}} C[r\gamma] \text{ for all rules } l \to r, \text{ contexts } C \text{ and substitutions } \gamma.$$

$\to_{\mathcal{R}}^{+}$ denotes the transitive closure of $\to_{\mathcal{R}}$, and $\to_{\mathcal{R}}^{*}$ the reflexive-transitive one.

Example 2. We consider a many-sorted TRS, with signature and rules as follows:

$$\begin{array}{lll} \mathsf{0} : \mathsf{int} & \mathsf{plus} : [\mathsf{int} \times \mathsf{int}] \Rightarrow \mathsf{int} & \mathsf{geq2} : [\mathsf{int} \times \mathsf{int} \times \mathsf{int} \times \mathsf{int}] \Rightarrow \mathsf{bool} \\ \mathsf{s} : [\mathsf{int}] \Rightarrow \mathsf{int} & \mathsf{sum} : [\mathsf{int}] \Rightarrow \mathsf{int} & \mathsf{sum2} : [\mathsf{bool} \times \mathsf{int}] \Rightarrow \mathsf{int} \\ \mathsf{p} : [\mathsf{int}] \Rightarrow \mathsf{int} & \mathsf{geq} : [\mathsf{int} \times \mathsf{int}] \Rightarrow \mathsf{bool} & \end{array}$$

$$\begin{array}{rlrl} \mathsf{sum}(x) & \to \mathsf{sum2}(\mathsf{geq}(\mathsf{0}, x), x) & \mathsf{geq}(x, y) & \to \mathsf{geq2}(x, y, \mathsf{0}, \mathsf{0}) \\ \mathsf{sum2}(\mathsf{true}, x) & \to \mathsf{0} & \mathsf{geq2}(\mathsf{s}(x), y, z, u) & \to \mathsf{geq2}(x, y, \mathsf{s}(z), u) \\ \mathsf{sum2}(\mathsf{false}, \mathsf{s}(x)) & \to \mathsf{plus}(\mathsf{s}(x), \mathsf{sum}(x)) & \mathsf{geq2}(\mathsf{p}(x), y, z, u) & \to \mathsf{geq2}(x, y, z, \mathsf{s}(u)) \\ \mathsf{plus}(\mathsf{s}(x), y) & \to \mathsf{s}(\mathsf{plus}(x, y)) & \mathsf{geq2}(\mathsf{0}, \mathsf{s}(x), y, z) & \to \mathsf{geq2}(\mathsf{0}, x, y, \mathsf{s}(z)) \\ \mathsf{plus}(\mathsf{p}(x), y) & \to \mathsf{p}(\mathsf{plus}(x, y)) & \mathsf{geq2}(\mathsf{0}, \mathsf{p}(x), y, z) & \to \mathsf{geq2}(\mathsf{0}, x, \mathsf{s}(y), z) \\ \mathsf{plus}(\mathsf{0}, y) & \to y & \mathsf{geq2}(\mathsf{0}, \mathsf{0}, \mathsf{s}(x), \mathsf{s}(y)) & \to \mathsf{geq2}(\mathsf{0}, \mathsf{0}, x, y) \\ \mathsf{s}(\mathsf{p}(x)) & \to x & \mathsf{geq2}(\mathsf{0}, \mathsf{0}, x, \mathsf{0}) & \to \mathsf{true} \\ \mathsf{p}(\mathsf{s}(x)) & \to x & \mathsf{geq2}(\mathsf{0}, \mathsf{0}, \mathsf{0}, \mathsf{s}(x)) & \to \mathsf{false} \end{array}$$

Here, $\mathsf{sum}(n)$ calculates $\Sigma_{i=1}^{n} i$. Because we consider integers, rather than the (well-founded) natural numbers, this is a somewhat tricky system for common analysis methods. A term $\mathsf{sum}(\mathsf{s}(\mathsf{0}))$ is reduced to $\mathsf{s}(\mathsf{0})$ in 11 steps.

Example 3. We might simplify Example 2 by considering an infinite signature, which contains all integers, and an infinite set of rules, as is (roughly) done in [8]:

$$\begin{aligned}
\mathsf{sum}(x) &\to \mathsf{sum2}(\mathsf{geq}(0,x),x) \\
\mathsf{sum2}(\mathsf{true},x) &\to 0 \\
\mathsf{sum2}(\mathsf{false},x) &\to \mathsf{plus}(x,\mathsf{sum}(\mathsf{plus}(x,-1))) \\
\mathsf{plus}(\mathsf{n},\mathsf{m}) &\to \mathsf{k} && \forall n,m,k \in \mathbb{Z} \text{ such that } n+m=k \\
\mathsf{geq}(\mathsf{n},\mathsf{m}) &\to \mathsf{true} && \forall n,m \in \mathbb{Z} \text{ such that } n \geq m \\
\mathsf{geq}(\mathsf{n},\mathsf{m}) &\to \mathsf{false} && \forall n,m \in \mathbb{Z} \text{ such that } n < m
\end{aligned}$$

This system is more pleasant, but has infinitely many rules, which makes it awkward to deal with except for dedicated techniques. Also, we still have to encode the constraints in the rules (and add rules to evaluate them), which makes analysis tricky. For example, termination of $\mathsf{sum}(x)$ relies on x getting closer to 0 in every step; to prove this, we must track the implications of $\mathsf{geq}(0,x) \to^*_{\mathcal{R}} \mathsf{false}$.

Note: term rewriting is usually defined without sorts. Then, function symbols have an *arity* (number of arguments) rather than a sort declaration. Such a TRS can be seen as a many-sorted TRS by assigning to symbols with arity n a sort declaration $[\mathsf{term} \times \ldots \times \mathsf{term}] \Rightarrow \mathsf{term}$, with n occurrences of term before the $\Rightarrow$.

3 Term Rewriting with Logical Constraints

Examples 2 and 3 illustrate why rewriting with native support for integer operations and constraints is a good idea. Normal rewriting simply does not seem adequate when handling data types which are not usually defined inductively.

We could add integers and integer constraints to rewriting, as in [3]. But with equal effort, we may be more general. Rather than focusing on $\mathbb{Z}$, we follow the ideas of [9] and take the underlying domain, and operations on it, as parameters.

Terms. We assume given a signature $\Sigma = \Sigma_{terms} \cup \Sigma_{theory}$. *Terms* are elements of $\mathcal{T}erms(\Sigma, \mathcal{V})$ as in Section 2.2. Moreover, we assume given a mapping $\mathcal{I}$ which assigns to each sort occurring in Σ_{theory} a set, and an *interpretation mapping* $\mathcal{J}$ which maps each $f : [\iota_1 \times \ldots \times \iota_n] \Rightarrow \kappa \in \Sigma_{theory}$ to a function $\mathcal{J}_f$ in $\mathcal{I}_{\iota_1} \Longrightarrow \ldots \Longrightarrow \mathcal{I}_{\iota_n} \Longrightarrow \mathcal{I}_\kappa$. For every sort ι occurring in Σ_{theory} we also fix a set $\mathcal{V}al_\iota \subseteq \Sigma_{theory}$ of *values*: function symbols $a : [] \Rightarrow \iota$, where $\mathcal{J}$ gives a one-to-one mapping from $\mathcal{V}al_\iota$ to $\mathcal{I}_\iota$. Let $\mathcal{V}al$ be the set of all values. We generally identify a value c with the logical term $c()$. An interpretation mapping can be extended to an interpretation on ground terms in $\mathcal{T}erms(\Sigma_{theory}, \mathcal{V})$ in the obvious way:

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

The elements of Σ_{theory} and Σ_{terms} may overlap only on values ($\Sigma_{theory} \cap \Sigma_{terms} \subseteq \mathcal{V}al$). We call a term in $\mathcal{T}erms(\Sigma_{theory}, \mathcal{V})$ a *logical term*, and a term in $\mathcal{T}erms(\Sigma_{terms}, \mathcal{V})$ a *proper term*. Intuitively, logical terms define a function or constraint in the model, while proper terms are the objects we want to rewrite. Mixed terms typically occur as intermediate steps in a reduction.

A ground logical term s *has value* t if t is a value such that $[\![s]\!] = [\![t]\!]$. Every ground logical term has a unique value. A *logical constraint* is a logical term of

some sort bool with $\mathcal{I}_{\mathsf{bool}} = \mathbb{B}$. We generally let $\mathcal{V}al_{\mathsf{bool}} = \{\mathsf{true}, \mathsf{false}\}$. A ground logical constraint s is *valid* if $[\![s]\!]_{\mathcal{J}} = \top$. A non-ground constraint s is valid if $s\gamma$ is valid for all substitutions γ which map the variables in $Var(s)$ to a value.

Example 4. Choosing $\mathcal{I}_{\mathsf{int}} = \mathbb{Z}$ and $\mathcal{I}_{\mathsf{bool}} = \mathbb{B}$, we might let Σ_{theory} be the set below, with interpretations as given in e.g. SMTLIB [1]:

$$\begin{array}{lll} \mathsf{true} : \mathsf{bool} & + : [\mathsf{int} \times \mathsf{int}] \Rightarrow \mathsf{int} & \wedge : [\mathsf{bool} \times \mathsf{bool}] \Rightarrow \mathsf{bool} \\ \mathsf{false} : \mathsf{bool} & \geq : [\mathsf{int} \times \mathsf{int}] \Rightarrow \mathsf{bool} & \neg : [\mathsf{bool}] \Rightarrow \mathsf{bool} \\ \mathsf{n} : \mathsf{int}\ (n \in \mathbb{Z}) & = : [\mathsf{int} \times \mathsf{int}] \Rightarrow \mathsf{bool} & \end{array}$$

Moreover, we let $\Sigma_{terms} = \{\mathsf{sum} : [\mathsf{int}] \Rightarrow \mathsf{int}\} \cup \{\mathsf{n} : \mathsf{int} \mid n \in \mathbb{Z}\}$.

The values in Σ are true, false, and n for all $n \in \mathbb{Z}$. Examples of logical terms, considering $\geq$, $+$ and $=$ as infix symbols, are $\mathsf{0} = \mathsf{0} + -\mathsf{1}$ and $x + \mathsf{3} \geq y + -\mathsf{42}$. Both are constraints. $\mathsf{5} + \mathsf{9}$ is also a ground logical term, but is not a constraint. $\mathsf{sum}(x)$ and $\mathsf{sum}(\mathsf{sum}(\mathsf{42})$ are proper terms. The value $\mathsf{0}$ is both a proper and a logical term. $\mathsf{sum}(\mathsf{37} + \mathsf{5})$ is neither, but is still a term (also called *mixed term*).

In Example 4 we restricted interest to functions in SMTLIB for Σ_{theory}, but this is not fundamental; we might also for instance have a symbol $\mathsf{p} : [\mathsf{int}] \Rightarrow \mathsf{int}$ with $\mathcal{J}_{\mathsf{p}} = \lambda x.x - 1$, or $\mathsf{pi} : [\mathsf{int}] \Rightarrow \mathsf{int}$ with $\mathcal{J}_{\mathsf{pi}} = \lambda n.$"the n^{th} decimal of π". It is in general a good idea, however, to limit interest to computable functions.

Rules and Rewriting. A *rule* is a triple $l \to r\ [\varphi]$ where l and r are terms, and φ is a logical constraint. l must have the form $f(l_1, \ldots, l_n)$ with $f \in \Sigma_{terms} \backslash \Sigma_{theory}$, and l and r must have the same sort. If $\varphi = \mathsf{true}$ with $\mathcal{J}(\mathsf{true}) = \top$, the rule is usually just denoted $l \to r$. A rule is *regular* if $Var(\varphi) \subseteq Var(l)$ and *standard* if l is a proper term. We define $LVar(l \to r\ [\varphi]))$ as $Var(\varphi) \cup (Var(r) \setminus Var(l))$.

A substitution γ *respects* a rule $l \to r\ [\varphi]$ if $Dom(\gamma) = Var(l) \cup Var(r) \cup Var(\varphi)$, $\gamma(x)$ is a value for all $x \in LVar(l \to r\ [\varphi])$ and $\varphi\gamma$ is valid.

We assume given a set of rules $\mathcal{R}$. The *rewrite relation* $\to_{\mathcal{R}}$ is a relation on terms, defined as the union of $\to_{\mathtt{rule}}$ and $\to_{\mathtt{calc}}$, where:

$$\begin{array}{rl} C[l\gamma] \to_{\mathtt{rule}} C[r\gamma] & \text{if } l \to r\ [\varphi] \in \mathcal{R} \text{ and } \gamma \text{ respects } l \to r\ [\varphi] \\ C[f(s_1, \ldots, s_n)] \to_{\mathtt{calc}} C[v] & \text{if } f \in \Sigma_{theory} \backslash \Sigma_{terms}, \text{ all } s_i \text{ values,} \\ & \quad \text{and } v \text{ is the value of } f(s_1, \ldots, s_n) \end{array}$$

A reduction step with $\to_{\mathtt{calc}}$ is called a *calculation*. A term is in *normal form* if (and only if) it cannot be reduced with $\to_{\mathcal{R}}$. Sometimes we consider *innermost reduction*: $C[f(\boldsymbol{s})] \to_{\mathcal{R},in} C[t]$ if $f(\boldsymbol{s}) \to_{\mathcal{R}} t$, and all s_i are in normal form.

A *logical constrained term rewriting system* (LCTRS) is defined as the pair $(\mathcal{T}erms(\Sigma, \mathcal{V}), \to_{\mathcal{R}})$. An LCTRS is typically given by providing $\mathcal{I}$ and $\mathcal{J}$ and the sets $\Sigma_{terms}, \Sigma_{theory}$ and $\mathcal{R}$. When clear from context, the signatures and mappings may be omitted. An *innermost LCTRS* is the pair $(\mathcal{T}erms(\Sigma, \mathcal{V}), \to_{\mathcal{R},in})$.

A (normal or innermost) LCTRS is *standard* or *regular* if all its rules are standard or regular respectively. In a regular LCTRS, $\to_{\mathcal{R}}$ is computable, provided $\mathcal{J}_f$ is computable for all $f \in \Sigma_{theory}$. Even in a regular LCTRS, the right-hand sides of rules may contain fresh variables. This can for example be used to simulate user input. Think for example of a rule $\mathsf{Start} \to \mathsf{Handle}(\mathit{input})$ where *input* is a variable; by definition of $\to_{\mathcal{R}}$, *input* can only be instantiated by a value.

Example 5. It is time to see how these definitions work in practice. Let us modify Example 2 to use constraints and calculations. We have defined Σ_{theory} and Σ_{terms} in Example 4. The rules are replaced by the following set:

$$\mathsf{sum}(x) \to 0\ [0 \geq x] \qquad \mathsf{sum}(x) \to x + \mathsf{sum}(x + -1) \quad [\neg(0 \geq x)]$$

Note that the sum rules may only be applied to a term $\mathsf{sum}(n)$ whose immediate argument n is a value, so this subterm itself cannot contain the symbol sum.

For an example derivation, let us calculate $\Sigma_{n=1}^{2} n$. We have: $\mathsf{sum}(2) \to_{\mathtt{rule}} 2 + \mathsf{sum}(2 + -1) \to_{\mathtt{calc}} 2 + \mathsf{sum}(1) \to_{\mathtt{rule}} 2 + (1 + \mathsf{sum}(1 + -1)) \to_{\mathtt{calc}} 2 + (1 + \mathsf{sum}(0)) \to_{\mathtt{rule}} 2 + (1 + 0) \to_{\mathtt{calc}} 2 + 1 \to_{\mathtt{calc}} 3$. In each step, it so happens that we have exactly one choice of what rule to apply, and where. For example in the first step, $\neg(0 \geq 2)$ holds and $0 \geq 2$ does not, so only the second rule is applicable. Neither rule can be applied on $\mathsf{sum}(2 + -1)$, as $2 + -1$ is no value; $\to_{\mathtt{calc}}$ *is* applicable. We cannot use a calculation step on $2 + (1 + \mathsf{sum}(0))$, as there is no subterm with the right form; the system does not know about associativity.

One might wonder why we insist that all variables in *LVar* are instantiated with values, rather than just logical terms. Having a rule $f(x, y) \to y\ [x \geq y]$, could we not reduce $f(x + 1, x)$ without this instantiation?

The reason to require that $\gamma(x)$ is ground for $x \in Var(\varphi)$ is simplicity of the rewrite relation: by posing this restriction, validity of $\varphi\gamma$ is mostly easy to test. Without it, validity might not be computable.[1] Moreover, without this restriction the reduction relation is not preserved under substitution: for a symbol $\mathsf{a} \in \Sigma_{terms} \setminus \Sigma_{theory}$, we cannot have $f(\mathsf{a} + 1, \mathsf{a}) \to \mathsf{a}$, as $\mathsf{a} + 1$ is not a logical term. It does make sense to study whether a term $f(x + 1, y)$, or even a term $f(x, y)$ with $x > y$ reduces. In Section 4 we will see how to rewrite *constrained terms*.

By requiring that $\gamma(x)$ is even a value, we avoid complicating notions like complexity. If we could $\to_{\mathtt{rule}}$-reduce terms like $\mathsf{sum}(3 + -1 + -1 + -1)$, then these additions would have to be calculated in every step when testing the constraint. There does not seem to be any advantage to allowing these hidden calculations; we can simply $\to_{\mathtt{calc}}$-normalise ground logical terms before applying a rule step. For the same motivation of complexity, we only allow $\to_{\mathtt{calc}}$ to take single steps, rather than allowing $C[s] \to_{\mathtt{calc}} C[v]$ for any logical term s with value v.

Note that $\to_{\mathtt{calc}}$ is not special; using $\to_{\mathtt{calc}}$ is functionally equivalent to extending $\mathcal{R}$ with all rules $f(x_1, \ldots, x_n) \to y\ [f(x_1, \ldots, x_n) = y]$ where $f \in \Sigma_{theory}$.

Regularity and Standardness. Regularity is a useful condition. An irregular LCTRS is not in general deterministic, polynomially solvable, or even computable. Consider for example a rule $\mathsf{f}(x) \to \mathsf{g}(\mathsf{f}(y), \mathsf{f}(z))\ [x = y * z \wedge y > 1 \wedge z > 1]$, which quickly decomposes a natural number into its prime factors.

Still, there is some advantage in allowing irregular systems. For example, in termination analysis a transformation might chain the two regular rules $f(x) \to g(x - 1, \mathit{input})\ [x > 0]$ and $g(x, y) \to f(y)\ [x \geq y]$ into a single irregular rule: $f(x) \to f(y)\ [x > 0 \wedge x - 1 \geq y]$. In addition, an irregular rule can be used to

[1] For example, defining $\mathcal{J}_p(n)$ to be true if a sequence of 9999 nines starts at the n^{th} decimal of π, and false otherwise, and considering a rule $f(x) \to x\ [\neg p(x)]$, we don't know whether a term $f(x)$ should reduce for all instances of x.

calculate a partial function, e.g. $\mathsf{div}(x, y) \to z\ [z * y = x]$, which cannot be easily defined otherwise. Note that such a rule does not lead to undecidability.

Similar to regularity, *standardness* is convenient: in a standard system there are no overlaps between $\to_{\mathtt{rule}}$ and $\to_{\mathtt{calc}}$. Standardness is a natural property, since symbols from $\Sigma_{theory} \setminus \Sigma_{terms}$ in terms are conceptually intended primarily as a way to do calculations. We don't use modulo reasoning: a non-standard rule $f(x + 1) \to r$ matches only terms of the form $f(s + 1)$, so *not* for instance $f(3)$. As with regularity, non-standard systems may arise during analysis, for example when a rule is reversed. Note that even in standard systems, left-hand sides of rules may contain values (or other symbols in Σ_{terms}); while we often encounter rules of the form $f(x_1, \ldots, x_n) \to r\ [\varphi]$, this is not an innate property.

Overview. Compared to the rather hairy (Example 2) or infinite (Example 3) systems obtained when encoding integer arithmetic and constraints in a normal TRS, LCTRSs offer an elegant alternative. Although LCTRSs often have infinite signatures, calculation steps make it possible to avoid infinite sets of rules.

Example 6. To demonstrate a situation where we should not use the integers as an underlying set, consider the following short imperative program:

```
1. function main() {          5.     x = x * 2;
2.   byte x = input();         6.   }
3.   while (x < 150) {         7.   return x;
4.     if (x == 0) x = 1;      8. }
```

Here a `byte` is an unsigned 8-bit integer. This program doesn't terminate: input 0 gives an infinite reduction (with `x` changing from 0 to $1, 2, 4, 8, 16, 32, 64, 128, 0$).

Using the ideas of [3], we might model this program as follows:

$$\begin{array}{lll} \mathsf{main} \to \mathsf{loop}(\mathit{input}) & & \mathsf{loop}_1(x) \to \mathsf{loop}_2(1) \quad [x = 0] \\ \mathsf{loop}(x) \to \mathsf{loop}_1(x) & [x < 150] & \mathsf{loop}_1(x) \to \mathsf{loop}_2(x) \quad [\neg(x = 0)] \\ \mathsf{loop}(x) \to \mathsf{return}(x) & [\neg(x < 150)] & \mathsf{loop}_2(x) \to \mathsf{loop}(x * 2) \end{array}$$

As bytes are not unbounded, our underlying set is *not* $\mathbb{Z}$. Rather, we consider the set $\mathcal{BV}_8$ of bit vectors of length 8, and let $\mathcal{J}$ map to corresponding notions of addition, multiplication, comparison and equality (see SMT-LIB [1]). With this theory, a reduction from start adequately simulates a reduction in the original imperative program. Note that if we had naively translated the program to an LCTRS over the integers, we could have falsely concluded (local) termination.

We can analyse systems on bitvectors in much the same way as we analyse systems over the integers. We will see some ideas for this in Section 6.

4 Constrained Terms

As discussed in Section 3, there are good reasons why the definition of rewriting requires that variables in a constraint are instantiated by values. But sometimes you may want to know whether a term of a certain form rewrites. For example, if we know that $x < -3$, this is enough to decide that $\mathsf{sum}(x)$ reduces to $\mathsf{0}$.

In this section, we will therefore consider *constrained terms*: pairs $s\ [\varphi]$ of a term s and a constraint φ. Constrained terms are harder to rewrite and analyse

than normal terms, but sometimes the need may arise. For instance in rewriting induction (see e.g. [4,9]) when proving that $f(x) \leftrightarrow^* g(x, 0)$ $[x \geq 1]$, being able to reason about the reducts of $f(x)$ $[x \geq 1]$ is very relevant.

To rewrite constrained terms, we must take several things into account. For example, given a rule $f(0) \to 1$, we should be able to reduce $f(x)$ $[x = 0]$, even though $f(x)$ itself is not matched by the left-hand side. We will also need to deal with irregular rules; given a rule $f(x) \to g(y)$ $[y > x]$, we should be able to reduce $f(x)$ $[x > 3]$ to $g(y)$ $[y > 4]$, or at least to an instance, like $g(y)$ $[x > 3 \wedge y = x+1]$.

A substitution γ *respects* a constrained term s $[\varphi]$ if $\gamma(x)$ is a value for all $x \in Var(\varphi)$ and $[\![\varphi\gamma]\!] = \top$. Two constrained terms s $[\varphi]$ and t $[\psi]$ are *equivalent*, notation s $[\varphi] \approx t$ $[\psi]$, if for all substitutions γ which respect s $[\varphi]$ there is a substitution δ which respects t $[\psi]$ such that $s\gamma = t\delta$, and vice versa.

Example 7. Examples of constrained terms over the signature of sum are:

1. $\mathsf{sum}(x)$ $[x \geq 3]$;
2. $x + y$ $[x \geq y \wedge \neg(x = y) \wedge x = 3]$;
3. $3 + z$ $[1 \geq x \wedge x + 1 \geq z]$;

Constrained terms 2 and 3 are equivalent, as the following formulas hold in $\mathbb{Z}$:

$$\forall x, y.(x \geq y \wedge \neg(x = y) \wedge x = 3) \Rightarrow \exists x', z.1 \geq x' \wedge x' + 1 \geq z \wedge x = 3 \wedge y = z$$
$$\forall x', z.(1 \geq x' \wedge x' + 1 \geq z) \Rightarrow \exists x, y.x \geq y \wedge \neg(x = y) \wedge x = 3 \wedge y = z$$

It is clear that equivalence of two constrained terms is not always easy to tell.

To be able to modify constraints, we assume that Σ_{theory} contains a symbol $\wedge : [\mathsf{bool} \times \mathsf{bool}] \Rightarrow \mathsf{bool}$ with $\mathcal{J}_\wedge$ the conjunction operator on the booleans, and for all sorts ι a symbol $=_\iota : [\iota \times \iota] \Rightarrow \mathsf{bool}$ with $\mathcal{J}_{=_\iota} := \lambda n, m \in \mathcal{I}_\iota.n = m$.

Rewriting Constrained Terms. We let s $[\varphi] \to_{\mathtt{calc}} t$ $[\varphi \wedge x = f(s_1, \ldots, s_n)]$ if $s = C[f(s_1, \ldots, s_n)]$ with $f \in \Sigma_{theory} \setminus \Sigma_{terms}$, all s_i in $Var(\varphi) \cup \mathcal{V}al$, x a fresh variable, and $t = C[x]$. Additionally, s $[\varphi] \to_{\mathtt{rule}} t$ $[\varphi]$ if φ is satisfiable, $s = C[l\gamma]$ and $t = C[r\gamma]$ for some rule $l \to r$ $[\psi]$ and substitution γ such that:

- $Dom(\gamma) = Var(l) \cup Var(r) \cup Var(\psi)$
- $\gamma(x)$ is a value or variable in $Var(\varphi)$ for all $x \in LVar(l \to r\ [\psi])$
- $\varphi \Rightarrow (\psi\gamma)$ is valid (that is, for all δ with $\varphi\delta$ valid also $\psi\gamma\delta$ is valid)

The relation $\to_{\mathcal{R}}$ on constrained terms is defined as $\approx \cdot(\to_{\mathtt{calc}} \cup \to_{\mathtt{rule}})\cdot \approx$.

Example 8. With the rule $f(0) \to 1$: $f(x)$ $[x = 0] \approx f(0)$ $[\mathsf{true}] \to_{\mathtt{rule}} 1$ $[\mathsf{true}]$.

Example 9. With the irregular rule $f(x) \to g(y)$ $[y > x]$, we have: $f(x)$ $[x > 3] \approx f(x)$ $[x > 3 \wedge y > x] \to_{\mathtt{rule}} g(y)$ $[x > 3 \wedge y > x] \approx g(y)$ $[y > 4]$. Similarly, $f(x)$ $[x > 0]$ reduces with $f(x) \to g(y)$ $[x = y + 1]$ to $f(y)$ $[y \geq 0]$. We do *not* have that $f(x)$ $[\mathsf{true}] \to g(x-1)$ $[\mathsf{true}]$, as $x-1$ cannot be instantiated to a value.

Example 10. Following on Example 5, we may reduce $\mathsf{sum}(x)$ $[x > 2]$ as follows:

$$\begin{aligned}
\mathsf{sum}(x)\ [x > 2] &\to_{\mathcal{R}} x + \mathsf{sum}(x + -1)\ [x > 2] \\
&\to_{\mathcal{R}} x + \mathsf{sum}(y)\ [x > 2 \wedge y = x + -1] \\
&\to_{\mathcal{R}} x + (y + \mathsf{sum}(y + -1))\ [x > 2 \wedge y = x + -1] \\
&\to_{\mathcal{R}} x + (y + \mathsf{sum}(z))\ [x > 2 \wedge y = x + -1 \wedge z = y + -1]
\end{aligned}$$

The notion of reduction on constrained terms is intimately tied to the notion of reduction on terms, as the following two theorems demonstrate:

Theorem 1. *If* $s \to_{\mathcal{R}} t$ *then also* $s\ [\mathsf{true}] \to_{\mathcal{R}} t\ [\mathsf{true}]$.

Theorem 2. *If* $s\ [\varphi] \to_{\mathcal{R}} t\ [\psi]$ *then for all substitutions* γ *which respect* $s\ [\varphi]$ *there is a substitution* δ *which respects* $t\ [\psi]$ *such that* $s\gamma \to_{\mathcal{R}} t\delta$.

Thus, we have a notion of constrained terms and reduction thereof. We do not consider these notions as basic; rather, using for instance Theorem 2, they can be used in analysis to find properties of unconstrained terms in the system (think for instance of an inductive proof that $\mathsf{sum}(x) \geq x$ if $x \geq \mathsf{0}$).

Determining whether a constrained term reduces, or what it reduces to, is a difficult problem. In special cases (for instance, regular rules with linear integer arithmetic) it is decidable, but in others we may have to resort to clever guessing.

5 Comparison to Existing Systems

So, we have a new formalism. Is this truly more convenient or general than existing formalisms? Here we will briefly study some formalisms from the literature, and sketch how they relate to the LCTRSs introduced here. However, a comprehensive study of all relevant formalisms is beyond the reach of this paper.

5.1 Constrained TRSs from [9,12,14]

LCTRSs are based primarily on the constrained TRSs in [9,12,14]. Like our LCTRSs, these systems have a separate theory signature, and a given interpretation mapping $\mathcal{J}$. The main difference is that they have no values or calculation steps. Instead, these features are encoded in the terms and rules, with for example the integers being represented as $\mathsf{0}, \mathsf{s}(\mathsf{0}), \mathsf{s}(\mathsf{s}(\mathsf{0})), \ldots, \mathsf{p}(\mathsf{0}), \mathsf{p}(\mathsf{p}(\mathsf{0})), \ldots$

Example 11. The sum system is implemented as a CTRS with rules:

$$\begin{array}{llllll} \mathsf{sum}(x) \to \mathsf{0} & [\mathsf{0} \geq x] & \mathsf{0} + y \to y & & \mathsf{s}(\mathsf{p}(x)) \to x \\ \mathsf{sum}(\mathsf{s}(x)) \to \mathsf{sum}(x) + \mathsf{s}(x) & [x \geq \mathsf{0}] & \mathsf{s}(x) + y \to \mathsf{s}(x + y) & & \mathsf{p}(\mathsf{s}(x)) \to x \\ & & \mathsf{p}(x) + y \to \mathsf{p}(x + y) & & \end{array}$$

Compared to their predecessors, LCTRS are far simpler to use: by having symbols for all values and using calculation steps, systems are implemented much more concisely (as demonstrated by sum). Moreover, the resulting systems are easier to analyse. For example, note that this version of sum is not a constructor system, and proving confluence or complete reducibility is difficult. Also, due to the countable nature of terms, no finite CTRS can encode Example 12:

Example 12. Using sorts int, real and bool, and addition on the real numbers denoted by $+$, we might represent the function $n \mapsto \Sigma_{i=i}^{n} \sqrt{n}$ as follows:

$$\mathsf{sumroot}(x) \to \mathsf{0.0}\ [\mathsf{0} \geq x] \qquad \mathsf{sumroot}(x) \to \mathsf{sqrt}(x) +. \mathsf{sumroot}(x - \mathsf{1})\ [\neg(\mathsf{0} \geq x)]$$

There does not seem to be an easy way to simulate CTRSs as LCTRSs or the other way around. However, initial results suggest that results for CTRSs [9,12,14] easily extend to LCTRSs, and are moreover vastly simplified by the translation.

5.2 Integer Term Rewriting Systems

In Example 3 we saw a system somewhat like the *integer term rewriting systems* in [8]. These ITRSs are innermost TRSs with an infinite signature $\Sigma \cup \Sigma_{int}$, where Σ_{int} includes $BOp = \{+, -, *, /, \%, >, \geq, <, \leq, =, \neq, \wedge, \Rightarrow\}$ and moreover true, false and all integers. $\mathcal{R}$ is defined as $R \cup \mathcal{PD}$, where $\mathcal{PD} = \{n \circ m \to k \mid n, m, k \in \mathbb{Z} \cup \mathbb{B},\ \circ \in BOp \mid n \circ m = k \text{ holds in } \mathbb{Z} \text{ and } \mathbb{B}\}$ (e.g. $1 + 2 \to 3 \in \mathcal{PD}$).

Example 13. An example ITRS in [8] has $\Sigma = \{\mathsf{log}, \mathsf{lif}\}$ and R consisting of:

$$\mathsf{log}(x, y) \to \mathsf{lif}(x \geq y \wedge y > 1, x, y) \qquad \begin{array}{l}\mathsf{lif}(\mathsf{true}, x, y) \to 1 + \mathsf{log}(x/y, y)\\ \mathsf{lif}(\mathsf{false}, x, y) \to 0\end{array}$$

Terms containing symbols $\geq$, $\wedge$, $+$ and $/$ can be rewritten using the $\mathcal{PD}$ rules.

We can model an ITRS as an LCTRS, which is finite if R is finite. ITRSs are not defined with sorts, but sorts can easily be imagined. Indeed, there seems little reason to analyse the behaviour of a term like $\mathsf{log}(\mathsf{true}, 5 + \mathsf{false})$, and for innermost termination (the primary area of interest for ITRSs) presence of sorts makes no difference [7]. The only other issue is that some elements of BOp ($/$ and $\%$) define *partial* functions, so cannot be modelled by calculations.

To define an LCTRS with the same terms and rewrite relation as a given *sorted* ITRS (with sorts int and bool assigned in the obvious way), let $\mathcal{V}al := \{\mathsf{true}, \mathsf{false} : \mathsf{bool}\} \cup \{\mathsf{n} : \mathsf{int} \mid n \in \mathbb{Z}\}$, $\Sigma_{terms} := \Sigma \cup \mathcal{V}al \cup \{/, \% : [\mathsf{int} \times \mathsf{int}] \Rightarrow \mathsf{int}\}$ and $\Sigma_{theory} := \mathcal{V}al \cup (BOp \setminus \{/, \%\})$. We use the expected interpretations for Σ_{theory}. Let $\mathcal{R} := R \cup \{x/y \to z\ [x = y{*}z],\ x\%y \to z\ [x = y{*}u{+}z \wedge 0 \leq z \wedge z < y]\}$. Then $\to_{\mathcal{R}}$ is exactly the reduction relation from the original ITRS. $\mathcal{R}$ is finite, because $\to_{\mathtt{calc}}$ and the two irregular rules take over the role of $\to_{\mathcal{PD}}$. Note that the irregularity is not an issue for computability in this case.

Example 14. The system from Example 13 becomes the following LCTRS (ignoring the $\%$ symbol which does not occur in any rule):

$$\begin{array}{ll}\mathsf{log}(x, y) \to \mathsf{lif}(x \geq y \wedge y > 1, x, y) & \mathsf{lif}(\mathsf{true}, x, y) \to 1 + \mathsf{log}(x/y, y)\\ x/y \to z\ [x = y * z] & \mathsf{lif}(\mathsf{false}, x, y) \to 0\end{array}$$

Comment: if, for whatever reason, we do want to analyse the original *unsorted* ITRS, we can also do so with an LCTRS. In this case, we assign a single sort term as suggested in Section 2, and let $\mathcal{I}_{\mathsf{term}} = \mathbb{Z} \cup \mathbb{B}$; we cannot use calculations now, because all functions are partial, but can encode $\to_{\mathcal{PD}}$ with irregular rules.

Conditional ITRSs. Integer TRSs play a role in the termination analysis of Java Bytecode employed in [13]. There, termination of JBC is reduced to termination of a *conditional* ITRS; rules look somewhat like the rules in LCTRSs:

$$\begin{array}{ll}\mathsf{log}(x, y) \to 1 + \log(x/y, y) \mid & x \geq y \wedge y > 1 \to^* \mathsf{true}\\ \mathsf{log}(x, y) \to 0 \mid & \neg(x \geq y \wedge y > 1) \to^* \mathsf{true}\end{array}$$

These systems are unravelled to ITRSs for analysis (giving the system from Example 13). However, if the elements of Σ_{int} are not root symbols of left-hand sides of R, and $/$ and $\%$ do not occur in the conditions, we can translate such systems into LCTRSs immediately (replacing conditions by constraints), and

obtain a system where constraints are not encoded. We can prove that a CITRS generates the same relation as its transformation to an innermost LCTRS.

As for the other direction, LCTRSs are not a special case of ITRS. Most importantly, ITRSs have no native treatment of constraints. These have to be encoded, and to for instance prove termination of even simple systems we need far more powerful techniques than in the LCTRS setting. Moreover, ITRSs are restricted to the integers. While Example 6 *can* be encoded, using rules like $\mathsf{loop}_2(x) \to \mathsf{loop}((x*2)\%256)$, ITRSs cannot represent for instance Example 12.

5.3 $\mathbb{Z}$-TRSs

Next, we consider a mixture of the ideas in [3,4,5,6].[2] $\mathbb{Z}$-TRSs are based on a many-sorted signature Σ, which must include $\Sigma_{\mathtt{int}} = \{0, 1 : \mathsf{int},\ - : [\mathsf{int}] \Rightarrow \mathsf{int},\ +, * : [\mathsf{int} \times \mathsf{int}] \Rightarrow \mathsf{int}\}$. *Constraints* are given by the grammar:

$$\mathsf{C} ::= \mathsf{true} \mid \mathsf{false} \mid s = t \mid s > t \mid s \geq t \mid \neg\mathsf{C} \mid \mathsf{C} \wedge \mathsf{C} \qquad s, t \text{ terms over } \Sigma_{\mathtt{int}}$$

Rules are triples $l \to r\ [\varphi]$ where φ is a constraint, l and r are terms with the same sort, and l has the form $f(l_1, \ldots, l_n)$. The left-hand sides may not contain any element of $\Sigma_{\mathtt{int}}$. The reduction relation is defined much like in LCTRSs, except that γ "respects" $l \to r\ [\varphi]$ if all variables of sort int are instantiated by (not necessarily ground) terms over $\Sigma_{\mathtt{int}}$, and $\varphi\gamma$ is valid. Note that symbols $2, 3, \ldots$ are *not* included in the signature; terms have forms like $\mathsf{sum}((1+1)+1)$.

Example 15. In [3] we see how a code snippet is translated to a $\mathbb{Z}$-TRS:

```
while (x > 0 && y > 0) {
   if (x > y) { while (x > 0) { x--; y++; } }
   else { while (y > 0) { y--; x++; } }
}
```

$$\begin{aligned}
&\mathsf{eval}_1(x,y) \to \mathsf{eval}_2(x,y) && [x > 0 \wedge y > 0 \wedge x > y]\\
&\mathsf{eval}_1(x,y) \to \mathsf{eval}_3(x,y) && [x > 0 \wedge y > 0 \wedge \neg(x > y)]\\
&\mathsf{eval}_2(x,y) \to \mathsf{eval}_2(x + -1, y + 1) && [x > 0]\\
&\mathsf{eval}_2(x,y) \to \mathsf{eval}_1(x,y) && [\neg(x > 0)]\\
&\mathsf{eval}_3(x,y) \to \mathsf{eval}_3(x + 1, y + -1) && [y > 0]\\
&\mathsf{eval}_3(x,y) \to \mathsf{eval}_1(x,y) && [\neg(y > 0)]
\end{aligned}$$

In fact, $\mathbb{Z}$-TRSs are very close to our LCTRSs, but with a fixed theory signature. Although there is no concept of "values", there is no harm to internally replacing ground terms over $\Sigma_{\mathtt{int}}$ by the corresponding value (in a tool, or when manually rewriting terms), because the symbols in Σ_{int} do not occur in any left-hand side.

For every $\mathbb{Z}$-TRS, we can define an LCTRS which is roughly the same, modulo calculation of integer values. We can do this as follows: let $\Sigma_{theory} := \{\mathsf{n} : \mathsf{int} \mid n \in \mathbb{Z}\} \cup \{\mathsf{true}, \mathsf{false} : \mathsf{bool},\ - : [\mathsf{int}] \Rightarrow \mathsf{int},\ +, * : [\mathsf{int} \times \mathsf{int}] \Rightarrow \mathsf{int},\ =, >, \geq : [\mathsf{int} \times \mathsf{int}] \Rightarrow \mathsf{int},\ \neg : [\mathsf{bool}] \Rightarrow \mathsf{bool},\ \wedge : [\mathsf{bool} \times \mathsf{bool}] \Rightarrow \mathsf{bool}\}$ and $\Sigma_{terms} := (\Sigma \setminus \Sigma_{\mathtt{int}}) \cup \{\mathsf{n} : \mathsf{int} \mid n \in \mathbb{Z}\}$. Every $\mathbb{Z}$-TRS rule is already a standard rule in this LCTRS, and every term in the original $\mathbb{Z}$-TRS is still a term.

[2] The authors use simplifications of this formalism for different applications. For example, multiplication is omitted, or custom symbols must have output sort unit.

Theorem 3. *We can derive, for all ground terms s, t:*

- *if $s \to_{\mathcal{R}} t$ in a $\mathbb{Z}$-TRS, and $s \to^*_{\mathtt{calc}} s' \in \mathcal{T}erms(\Sigma_{terms} \cup \Sigma_{theory}, \mathcal{V})$, then exists t' such that $t \to^*_{\mathtt{calc}} t'$ and $s' \to^+_{\mathcal{R}} t'$ in the corresponding LCTRS;*
- *if $s \to_{\mathtt{rule}} t$ in the corresponding LCTRS, and $s' \in \mathcal{T}erms(\Sigma, \mathcal{V})$ such that $s' \to^*_{\mathtt{calc}} s$, then there is a term t' such that $t' \to^*_{\mathtt{calc}} t$ and $s' \to_{\mathcal{R}} t'$ in the original $\mathbb{Z}$-TRS.*

Thus, results from LCTRSs typically extend to $\mathbb{Z}$-TRSs. As with ITRSs, $\mathbb{Z}$-TRSs cannot model the behaviour of LCTRSs. Being fundamentally restricted to the integers, they cannot easily represent Example 6, nor Example 12. An extension of $\mathbb{Z}$-TRSs, which admits all integers in the signature, can model a variation of Example 6, as discussed in [6]. This analysis uses extra rules to "normalise" integers to their range, e.g. $\mathsf{loop}_2(x) \to \mathsf{loop}_3(x * 2)$, $\mathsf{loop}_3(x) \to \mathsf{loop}_3(x - 256)$ $[x \geq 256]$, $\mathsf{loop}_3(x) \to \mathsf{loop}(x)$ $[x \geq 0 \wedge 256 \geq x]$.

5.4 Constrained Equational Systems

For a very different direction, let us consider a system from the further past. In [11], a framework for constrained deduction is developed, which uses constrained terms and rules. Like the current paper, the interpretation of function symbols ($\mathcal{J}_f$) is not fixed, but assumed to be given by the user. There is no notion of values, however. This fits with a typical usage of the formalism, where the underlying model is the set of terms modulo some theory.

Example 16. We consider a constrained system with symbols $* : [\mathsf{term} \times \mathsf{term}] \Rightarrow \mathsf{term}$, $\mathsf{a}, \mathsf{b} : \mathsf{term}$, $=_{\mathsf{AC}} : [\mathsf{term} \times \mathsf{term}] \Rightarrow \mathsf{bool}$. The model $\mathcal{I}_{\mathsf{term}}$ is the set of terms over $\{*, \mathsf{a}, \mathsf{b}\}$, where a, b and $*$ are interpreted as themselves and $=_{\mathsf{AC}}$ is interpreted as equality on terms modulo AC (associativity and commutativity) of $*$.

Unlike LCTRSs, this formalism has no separate "term signature": all function symbols have a meaning in the model, and may occur in both terms and constraints. Rules have the form $s \to t$ $[\varphi]$ and are used for example to simplify constrained terms (called *constrained formulas*) modulo an equational theory.

Example 17. In the signature from Example 16, we consider a rule $(x * x) * x \to \mathsf{a}$ $[\neg(x =_{\mathsf{AC}} \mathsf{a})]$, which matches modulo AC. The constrained formula $x * (\mathsf{b} * (\mathsf{b} * y)) =_{\mathsf{AC}} \mathsf{a} * \mathsf{b}$ $[x =_{\mathsf{AC}} \mathsf{b}]$ is AC-equivalent to $((x * \mathsf{b}) * \mathsf{b}) * y =_{\mathsf{AC}} \mathsf{a} * \mathsf{b}$ $[x =_{\mathsf{AC}} \mathsf{b}]$, which can be reduced to $\mathsf{a} * y =_{\mathsf{AC}} \mathsf{a} * \mathsf{b}$ $[x =_{\mathsf{AC}} \mathsf{b}]$. This notion of reduction is called *total simplification*. There is also a notion of *partial simplification*, where constrained terms are reduced to pairs. This happens when a rule does not necessarily match; for example the constrained formula $x * (\mathsf{b} * (\mathsf{b} * y)) =_{\mathsf{AC}} \mathsf{a} * \mathsf{b}$ $[\neg(x =_{\mathsf{AC}} y]$ reduces to the pair $\mathsf{a} * y =_{\mathsf{AC}} \mathsf{a} * \mathsf{b}$ $[\neg(x =_{\mathsf{AC}} y \wedge (x * x) * x =_{AC} (x * \mathsf{b}) * \mathsf{b} \wedge \neg(x =_{\mathsf{AC}} \mathsf{a})]$ and $((x * \mathsf{b}) * \mathsf{b}) * y =_{\mathsf{AC}} \mathsf{c} * \mathsf{b}$ $[\neg(x =_{\mathsf{AC}} y) \wedge \neg((x * x) * x =_{AC} (x * \mathsf{b}) * \mathsf{b} \wedge \neg(x =_{\mathsf{AC}} \mathsf{a})]$.

There are many similarities between these equational systems and LCTRSs; to a large extent they can be seen as non-standard LCTRSs. From this perspective, complete simplification is exactly constrained rewriting as we saw in Section 4. We have no notion of partial simplification, because it fundamentally relies on

the symbols from terms being moved into the constraint, but similar techniques could be defined for the special case that $\Sigma_{terms} = \emptyset$.

However, LCTRSs do not allow reasoning modulo a theory, which alters fundamental properties like computability of reduction. Moreover, the systems from [11] violate an essential rule in LCTRSs: logical terms reduce only to their value. In the presence of rules like $x + (y + z) \to y$, many analysis techniques break.

Thus, while there is an overlap in expressibility between these two formalisms, we do not claim to cover or improve on this style of constrained rewriting. The dynamics of the systems are too different, and so are their purposes: where equational systems are designed for equational reasoning in logic, LCTRSs are designed for analysing programs. In the rest of this section, we have seen how LCTRSs relate to several formalisms which share this goal.

6 Analysing LCTRSs

Several times we have alluded to the ease of analysis in LCTRSs, so it is time to give some indication of how this is done. Unfortunately, we cannot do this justice, as there are many questions for analysis and little space. To give some ideas of how common techniques extend to LCTRSs, we will now briefly study some basic confluence and termination results.

6.1 (Weak) Orthogonality

Confluence is the property that whenever $s \to^*_{\mathcal{R}} t$ and $s \to^*_{\mathcal{R}} q$ there is some w such that $t \to^*_{\mathcal{R}} w$ and $q \to^*_{\mathcal{R}} w$. We will extend the common notion of *orthogonality*, a property which implies confluence, to LCTRSs.

It is well-known that for any pair of terms which can be unified, there is a *most general unifier*. That is, if s and t have distinct variables, and $s\gamma = t\gamma$, there is some δ such that also $s\delta = t\delta$, and any unifying substitution γ can be written as $\epsilon \circ \delta$ for some ϵ (here, $(\epsilon \circ \delta)(x) = \delta(x)\epsilon$ if $x \in Dom(\delta)$ and $\epsilon(x)$ otherwise). A substitution γ *respects variables* of a rule ρ if $\gamma(x)$ is a value or variable for all x in $LVar(\rho)$. If γ respects variables of $l \to r\ [\varphi]$, then $l\gamma \to r\gamma\ [\varphi\gamma]$ is also a rule.

Definition 1 (Critical Pair). *Given rules $\rho_1 \equiv l_1 \to r_1\ [\varphi_1]$ and $\rho_2 \equiv l_2 \to r_2\ [\varphi_2]$ with distinct variables, the* critical pairs *of ρ_1, ρ_2 are all tuples $\langle s, t, \varphi \rangle$ where:*

- *l_1 can be written as $C[l'_1]$, where l'_1 is not a variable, but is unifiable with l_2;*
- *$C \neq \Box$, or not $\rho_1 = \rho_2$ modulo renaming of variables, or $Var(r_1) \not\subseteq Var(l_1)$;*
- *the most general unifier γ of l'_1 and l_2 respects variables of both ρ_1 and ρ_2;*
- *$\varphi_1\gamma \wedge \varphi_2\gamma$ is satisfiable;*
- *$s = r_1\gamma$ and $t = (C\gamma)[r_2\gamma]$ and $\varphi = \varphi_1\gamma \wedge \varphi_2\gamma$.*

The critical pairs for calculations of a rule ρ are all critical pairs of ρ with any "rule" of the form $f(x_1, \ldots, x_n) \to y\ [y = f(\boldsymbol{x})]$ with $f \in \Sigma_{theory} \setminus \mathcal{V}al$.

Note that a rule $f \to g(x)$ has a critical pair with its own renamed copy: $\langle g(x), g(y), \mathsf{true} \wedge \mathsf{true} \rangle$. This is necessary because fresh variables in the right-hand sides of rules are a very likely source of non-confluence.

Example 18. Consider the following rules:

$$\begin{array}{lll} (\rho_1) & f(x_1, y_1) \to g(x_1 + y_1) & [x_1 \geq y_1] \\ (\rho_2) & f(x_2, y_2) \to g(x_2) & [x_2 \leq y_2] \\ (\rho_3) & f(x_3, y_3) \to g(y_3) & [x_3 < y_3] \\ (\rho_4) & f(x_4, x_4 + y_4) \to g(y_4) & [x_4 > 0] \end{array}$$

There are no critical pairs between ρ_1 and ρ_3: although $f(x_1, y_1)$ and $f(x_3, y_3)$ can be unified (with most general unifier $[x_1 := x,\ x_3 := x,\ y_1 := y,\ y_3 := y]$), the formula $x \geq y \wedge x < y$ is not satisfiable. On the other hand, ρ_1 and ρ_2 do admit a critical pair: $\langle g(x+y), g(y), x \geq y \wedge x \leq y \rangle$. None of the rules $\rho_1,\ \rho_2$ or ρ_3 gives a critical pair with ρ_4, since in the resulting mgu γ we have $\gamma(y_1) = x + y$, and thus this substitution does not respect the variables of ρ_1, ρ_2, ρ_3. Finally, ρ_4 has a critical pair for calculations, $\langle g(y), f(x, z), x > 0 \wedge z = x + y \rangle$.

Definition 2 (Weak Orthogonality). *A critical pair $\langle s, t, \varphi \rangle$ is* trivial *if $s\ [\varphi] \approx t\ [\varphi]$. An LCTRS $\mathcal{R}$ is* weakly orthogonal *if the left-hand side of each rule is linear (no variable occurs more than once), and for any pair $\rho_1, \rho_2 \in \mathcal{R}$: every critical pair between ρ_1 and a variable-renamed copy of ρ_2, and every critical pair of ρ_1 for calculations, is trivial. It is* orthogonal *if there are no critical pairs.*

The following result follows much like its unconstrained counterpart:

Theorem 4. *A weakly orthogonal LCTRS is confluent.*

Example 19. sum is orthogonal, so by Theorem 4 this LCTRS is confluent.

6.2 The Recursive Path Ordering

To prove termination of a TRS, it suffices to show that its rules are included in the *recursive path ordering* [2], a well-founded ordering $\succ$ which is monotonic and stable under substitutions. We will consider a simple variation of this ordering. To deal with the possibly infinite number of values, we assume that Σ_{theory} contains a symbol $\sqsupset_\iota$ for all sorts ι occurring in $\mathcal{V}al$, which is mapped to a well-founded ordering $>_\iota$ in $\mathcal{I}_\iota$. For example, we might take $\sqsupset_{\mathsf{int}} = \lambda xy.x > y \wedge x \geq 0$. We also assume given a well-founded ordering $\rhd$ on the symbols of $\Sigma_{terms} \backslash \Sigma_{theory}$.

The recursive path ordering is defined by the following derivation rules:

1. $s \succeq t\ [\varphi]$ if one of the following holds:
   (a) $s, t \in \mathcal{T}erms(\Sigma_{theory}, \mathcal{V}ar(\varphi))$, and $\varphi \Rightarrow (s = t \vee s \sqsupset t)$ is valid
   (b) $s = f(s_1, \ldots, s_n), t = f(t_1, \ldots, t_n)$ with $f \notin \Sigma_{theory}$ and each $s_i \succeq t_i\ [\varphi]$
   (c) $s \succ t\ [\varphi]$, or $s = t$ is a variable
2. $s \succ t\ [\varphi]$ if one of the following holds:
   (a) $s, t \in \mathcal{T}erms(\Sigma_{theory}, \mathcal{V}ar(\varphi))$, and $\varphi \Rightarrow s \sqsupset t$ is valid
   (b) $s = f(s_1, \ldots, s_n)$ with $f \in \Sigma_{terms} \setminus \Sigma_{theory}$ and one of:
      i. $s_i \succeq t\ [\varphi]$ for some $i \in \{1, \ldots, n\}$
      ii. $t = g(t_1, \ldots, t_m)$ with $g \in \Sigma_{theory}$ or $f \rhd g$, and for all i: $s \succ t_i\ [\varphi]$
      iii. $t = f(t_1, \ldots, t_n)$, all $s_i \succeq t_i\ [\varphi]$ and for at least one i: $s_i \succ t_i\ [\varphi]$
      iv. $t \in \mathcal{V}ar(\varphi)$

Theorem 5. *An LCTRS $\mathcal{R}$ is terminating if we can choose a suitable $\sqsupset_\iota$ for all ι, and some well-founded $\rhd$, such that $l \succ r\ [\varphi]$ for all $l \to r\ [\varphi] \in \mathcal{R}$.*

Proof. We can define a pair $(\equiv, >)$ of an equivalence relation and a compatible ordering with $\to_{\mathtt{calc}} \subseteq \equiv$ and $C[s] > C[t]$ if $s \succ t\ [\mathsf{true}]$ and $s \notin \mathcal{T}erms(\Sigma_{theory}, \emptyset)$. Having these, we observe first that $>$ is well-founded, and second that if $l \succeq r\ [\varphi]$, then $l\gamma \succ r\gamma\ [\mathsf{true}]$ for all substitutions γ which respect $l \to r\ [\varphi]$.

Example 20. Taking $n \sqsupset_{\mathsf{int}} m$ if $n > m$ and $n \geq 0$, the sum system is terminating by the recursive path ordering: For the first rule, $\mathsf{sum}(x) \succ \mathsf{0}\ [\mathsf{0} \geq x]$ by 2(b)ii. For the second, writing $\varphi = \neg(\mathsf{0} \geq x)$, we have $\mathsf{sum}(x) \succ x + \mathsf{sum}(x + -1)\ [\varphi]$ by 2(b)ii because $\mathsf{sum}(x) \succ x\ [\varphi]$ by 2(b)iv, and $\mathsf{sum}(x) \succ \mathsf{sum}(x + -1)\ [\varphi]$ by 2(b)iii because $x \succ x + -1\ [\varphi]$ by 2a, because $\varphi \Rightarrow (x > x + -1 \wedge x \geq \mathsf{0})$ is valid.

Note that Example 3, with encoded constraints, cannot be handled by RPO.

Of course, this is a very basic version of the recursive path ordering. There are various ways to strengthen the technique, but this is left for future work.

6.3 Observations

Both when analysing confluence and termination, a pattern appears: existing techniques extend in fairly natural way, with the constraints handled by proving validity of some formula. In other techniques we have studied but omitted here (such as dependency pairs and inductive equality proofs) a similar pattern arises.

Importantly, this pattern does not depend on the kind of theory we use: analysis takes a similar form whether we reason about integer arrays, reals or bitvectors. The difference is in how to solve the resulting formulas. When automatically analysing properties of LCTRSs, it seems natural to combine a dedicated analysis tool with SMT-solvers for the theory of interest. This way, we can immediately profit from the continuing improvement of the SMT-community, without having to adjust our methods when a new theory is explored.

7 Conclusion

In this paper, we have studied *logical constrained term rewriting systems*. LCTRSs offer an approach to program analysis for a large variety of languages and analysis questions. Due to their similarity to normal term rewriting, we can easily transpose the many powerful techniques of traditional term rewriting. However, by natively handling constraints, we obtain a much simpler analysis than if we were to encode the constraints in the rules.

In conclusion, LCTRS can be summarised with four keywords: They are *natural*: values in the logic are modelled with constants, and calculations do not need to be encoded. They are *general*: LCTRSs are not restricted to for instance the integers, but can handle all kinds of theories. They are *versatile*: LCTRSs can model a wide range of problems, from termination and overflow analysis to program equivalence, and can represent examples from many existing formalisms of constrained or integer rewriting. Finally, they are *flexible*: common analysis techniques for term rewriting extend to LCTRSs without much effort.

In the future, we aim to provide a tool to rewrite and analyse LCTRSs. Such analysis would not necessarily need special treatment for the various theories: in many cases (as we saw in Section 6), an LCTRS problem can be converted into a sequence of SMT-queries which might be fed into an external solver.

In addition, we hope to extend translations of program analysis from e.g. [3,9,13] with arrays and bitvectors, thus making use of the greater generality of LCTRSs, and the power of SMT-solvers for various theories.

References

1. Community. SMT-LIB, `http://www.smtlib.org/`
2. Dershowitz, N.: Orderings for term rewriting systems. Theor. Comput. Sci. 17(3), 279–301 (1982)
3. Falke, S., Kapur, D.: A term rewriting approach to the automated termination analysis of imperative programs. In: Schmidt, R.A. (ed.) CADE 2009. LNCS, vol. 5663, pp. 277–293. Springer, Heidelberg (2009)
4. Falke, S., Kapur, D.: Rewriting induction + linear arithmetic = decision procedure. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS (LNAI), vol. 7364, pp. 241–255. Springer, Heidelberg (2012)
5. Falke, S., Kapur, D., Sinz, C.: Termination analysis of C programs using compiler intermediate languages. In: Schmidt-Schauß, M. (ed.) Proc. RTA 2011. LIPIcs, vol. 10, pp. 41–50. Dagstuhl (2011)
6. Falke, S., Kapur, D., Sinz, C.: Termination analysis of imperative programs using bitvector arithmetic. In: Joshi, R., Müller, P., Podelski, A. (eds.) VSTTE 2012. LNCS, vol. 7152, pp. 261–277. Springer, Heidelberg (2012)
7. Fuhs, C., Giesl, J., Parting, M., Schneider-Kamp, P., Swiderski, S.: Proving Termination by Dependency Pairs and Inductive Theorem Proving. Journal of Automated Reasoning 47(2), 133–160 (2011)
8. Fuhs, C., Giesl, J., Plücker, M., Schneider-Kamp, P., Falke, S.: Proving termination of integer term rewriting. In: Treinen, R. (ed.) RTA 2009. LNCS, vol. 5595, pp. 32–47. Springer, Heidelberg (2009)
9. Furuichi, Y., Nishida, N., Sakai, M., Kusakari, K., Sakabe, T.: Approach to procedural-program verification based on implicit induction of constrained term rewriting systems. IPSJ Trans. Program. 1(2), 100–121 (2008)
10. Giesl, J., Raffelsieper, M., Schneider-Kamp, P., Swiderski, S., Thiemann, R.: Automated termination proofs for Haskell by term rewriting. ACM Transactions on Programming Languages and Systems 33(2), 7:1–7:39 (2011)
11. Kirchner, C., Kirchner, H., Rusinowitch, M.: Deduction with symbolic constraints. Revue Française d'Intelligence Artificielle 4(3), 9–52 (1990)
12. Nakabayashi, N., Nishida, N., Kusakari, K., Sakabe, T., Sakai, M.: Lemma generation method in rewriting induction for constrained term rewriting systems. Computer Software 28(1), 173–189 (2010) (in Japanese)
13. Otto, C., Brockschmidt, M., von Essen, C., Giesl, J.: Automated termination analysis of java bytecode by term rewriting. In: Lynch, C. (ed.) Proc. RTA 2010. LIPIcs, vol. 6, pp. 259–276. Dagstuhl (2010)
14. Sakata, T., Nishida, N., Sakabe, T.: On proving termination of constrained term rewrite systems by eliminating edges from dependency graphs. In: Kuchen, H. (ed.) WFLP 2011. LNCS, vol. 6816, pp. 138–155. Springer, Heidelberg (2011)
15. Schneider-Kamp, P., Giesl, J., Ströder, T., Serebrenik, A., Thiemann, R.: Automated termination analysis for logic programs with cut. Theory and Practice of Logic Programming 10(4-6), 365–381 (2010)