Constrained Term Rewriting tooL *

Cynthia Kop[1] and Naoki Nishida[2]

[1] Institute of Computer Science, University of Innsbruck
[2] Graduate School of Information Science, Nagoya University

Abstract. This paper discusses Ctrl, a tool to analyse – both automatically and manually – term rewriting with logical constraints. Ctrl can be used with TRSs on arbitrary underlying logics, and automatically analyse various properties such as termination, confluence and quasi-reductivity. Ctrl also offers both a manual and automatic mode for equivalence tests using inductive theorem proving, giving support for and verification of "hand-written" term equivalence proofs.

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 every program run 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 about term rewriting systems (TRSs). Such transformations are used to analyse termination of small, constructed languages (e.g. [2]), but also real code, like Java Bytecode [13], Haskell [7] or LLVM [4]. Similar transformations are used to analyse code equivalence in [3,5].

In these works, *constraints* arise naturally. Where traditional term rewriting systems generally consider well-founded sets like the natural numbers, more dedicated techniques are necessary when dealing with for instance integers or floating point numbers. This is why, typically, *extensions* of basic term rewriting are considered, adding a (usually infinite) number of predefined symbols and rules – for instance including all integers as constant symbols, and rules such as $\mathsf{1} + \mathsf{0} \to \mathsf{1}$, $\mathsf{1} + \mathsf{1} \to \mathsf{2}$, ... – along with some way of specifying constraints. The *Logically Constrained Term Rewriting Systems* (LCTRSs) from [10] take this a step further, by not limiting interest to a fixed theory (such as the integers with standard functions and relations), but rather allowing an *arbitrary* underlying theory. This makes it possible to define systems corresponding to (and immediately obtain theoretical results for), e.g., imperative programs with arrays, or to functional programs with advanced data structures. As observed in [10], LCTRSs conservatively extend many typical forms of constrained rewriting.

To analyse LCTRSs automatically, we have created the tool Ctrl. Like the

* This research is supported by the Austrian Science Fund (FWF) international project I963 and the Graduate Program for RWDC Leaders of Nagoya University.

general LCTRS framework, Ctrl can be equipped with an arbitrary underlying theory, provided an SMT-solver is given to solve its satisfiability and validity problems. The tool has the functionality to test confluence and quasi-reductivity, extensive capability to verify termination, and both automatic and manual support for inductive theorem proving, by which one may prove equivalence of two different functions. Ctrl participated in the *Integer Transition Systems* and *Integer TRS* categories of the 2014 termination competition (no corresponding categories for other theories were present). Ctrl is open-source, and available at:

`http://cl-informatik.uibk.ac.at/software/ctrl/`

Contribution. Compared to other tools on forms of constrained rewriting (e.g. AProVE [6]), Ctrl is unique in supporting *arbitrary* theories. Of the tool's many features, only automatic equivalence proving has been presented before [11].

Structure. In this paper, we will consider the various aspects of Ctrl. In Sec. 2, we start by recalling the definition of LCTRSs. In Sec. 3, we show how these notions translate to Ctrl, and in Sec. 4 we discuss the problems Ctrl can solve. The next sections treat the two most sophisticated options: termination (Sec. 5) and term equivalence (Sec. 6). Experiments and practical usage, where relevant, are explained in the corresponding sections. Finally, we conclude in Sec. 7.

2 Logically Constrained Term Rewriting Systems

The full definition of LCTRSs is given in [10,11]. We will here explain by example.

In LCTRSs, many-sorted term rewriting is combined with pre-defined *functions* and *values* over arbitrary sets, along with *constraints* to limit reduction. For example, we might define an LCTRS to calculate the Fibonacci numbers:

$$\mathtt{fib}(n) \to \mathtt{1}\ [n \leq \mathtt{1}] \qquad \mathtt{fib}(n) \to \mathtt{fib}(n-\mathtt{1}) + \mathtt{fib}(n-\mathtt{2})\ [n > \mathtt{1}]$$

Here, the *integers* are added to term rewriting, along with functions for addition, subtraction and comparison. To be precise, we have the following symbols:

values	theory functions	TRS functions
$\mathtt{true}, \mathtt{false} : \mathtt{Bool}$	$+, - : [\mathtt{Int} \times \mathtt{Int}] \Rightarrow \mathtt{Int}$	$\mathtt{fib} : [\mathtt{Int}] \Rightarrow \mathtt{Int}$
$\mathtt{0}, \mathtt{1}, -\mathtt{1}, \mathtt{2}, \ldots : \mathtt{Int}$	$\leq, > : [\mathtt{Int} \times \mathtt{Int}] \Rightarrow \mathtt{Bool}$	

The values and theory functions each have a pre-defined *meaning* in the underlying theory of the booleans and integers. The TRS functions are used to define custom functions, like in a functional programming language (although at the moment, higher-order functions such as `map` are not permitted), but also for constructors, which make it possible to define inductive types.

Rewriting is constrained as follows: a rule may only be applied if the variables in its constraint are all instantiated by values, and the constraint evaluates to `true` in the theory. In addition, theory functions occurring inside terms are evaluated step by step. For example, $\mathtt{fib}(\mathtt{2}+(\mathtt{0}+\mathtt{1}))$ cannot be reduced with the second rule, as $\mathtt{2}+(\mathtt{0}+\mathtt{1})$ is not a value. Instead, $\mathtt{fib}(\mathtt{2}+(\mathtt{0}+\mathtt{1})) \to \mathtt{fib}(\mathtt{2}+\mathtt{1}) \to \mathtt{fib}(\mathtt{3})$ by two calculation steps, and $\mathtt{fib}(\mathtt{3}) \to \mathtt{fib}(\mathtt{3}-\mathtt{1}) + \mathtt{fib}(\mathtt{3}-\mathtt{2})$.

A key feature of LCTRSs is that we do not fix the underlying sets, theory functions or values, nor their meanings. Predicates, too, are merely functions mapping to booleans, which could be anything according to need. For instance, to model an implementation of the `strlen` function in C, we might use

$$\begin{array}{lll} \mathtt{slen}(s) \to \mathtt{u}(s,0) & \mathtt{u}(s,i) \to \mathtt{err} & [i < 0 \vee \mathtt{size}(s) \leq i] \\ & \mathtt{u}(s,i) \to \mathtt{ret}(i) & [0 \leq i < \mathtt{size}(s) \wedge \mathtt{get}(s,i) = \mathtt{c0}] \\ & \mathtt{u}(s,i) \to \mathtt{u}(s,i+1) & [0 \leq i < \mathtt{size}(s) \wedge \mathtt{get}(s,i) \neq \mathtt{c0}] \end{array}$$

and the following signature, where `Carr` is interpreted as the set $\{0,\dots,255\}^*$ and `Int` as the set $\{-2^{15},\dots,2^{15}-1\}$, with addition subject to overflow:

values	TRS functions
$\mathtt{true},\mathtt{false} : \mathtt{Bool}$	$\mathtt{slen} : [\mathtt{Carr}] \Rightarrow \mathtt{X}$
$-32768,\dots,32768 : \mathtt{Int}$	$\mathtt{u} : [\mathtt{Carr} \times \mathtt{Int}] \Rightarrow \mathtt{X}$
$\mathtt{c0},\mathtt{c1},\dots,\mathtt{c255} : \mathtt{Char}$	$\mathtt{err} : \mathtt{X}$
$\{\},\{0\},\{1,0\},\dots : \mathtt{Carr}$	$\mathtt{ret} : [\mathtt{Int}] \Rightarrow \mathtt{X}$

theory functions
$+ : [\mathtt{Int} \times \mathtt{Int}] \Rightarrow \mathtt{Int}$
$\leq,<,=,\neq : [\mathtt{Int} \times \mathtt{Int}] \Rightarrow \mathtt{Bool}$
$\vee,\wedge : [\mathtt{Bool} \times \mathtt{Bool}] \Rightarrow \mathtt{Bool}$
$\mathtt{not} : [\mathtt{Bool}] \Rightarrow \mathtt{Bool}$
$\mathtt{size} : [\mathtt{Carr}] \Rightarrow \mathtt{Int}$
$\mathtt{get} : [\mathtt{Carr} \times \mathtt{Int}] \Rightarrow \mathtt{Char}$

It is common to assume that at least all the usual boolean operators ($\wedge, \vee, \mathtt{not}$) are present in Σ_{logic} and have the standard interpretation. Quantifiers are *not* supported directly, but can typically be replaced by a theory function; e.g., turning $\forall x \in \{0, \mathtt{size}(a)\}$ $[\mathtt{select}(a,x) > 0]$ into $\mathtt{positive}(a)$, with $\mathtt{positive} : [\mathtt{IntArray}] \Rightarrow \mathtt{Bool}$ a new theory function with the meaning "all elements of the argument are greater than 0".

3 Fundamentals

Ctrl is invoked with an input file defining an LCTRS and a query, using the format in Figure 1. Each of the fields (e.g. `SOLVER` *`solver`*) can be omitted.

```
THEORY theory
LOGIC logic
SOLVER solver
SIGNATURE
   signature
RULES
   rules
QUERY query
```

Fig. 1: Input File

Ctrl follows the core idea of LCTRSs by not using a predefined theory; instead, theory functions and values are defined in a theory file, which is included using the `THEORY` field. The underlying logic is handled by an external SMT-solver (as given by the `SOLVER` field), which uses the input and output format of SMT-LIB (see `http://smtlib.cs.uiowa.edu/`). The `LOGIC` field provides the name of an SMT-LIB logic following `http://smtlib.cs.uiowa.edu/logics.shtml` or any other logic supported by the SMT-solver. For the `fib` example of Sec. 2, we would for instance use `QF_LIA`.

The signature is given by listing TRS function symbols, along with their type declaration and separated by commas or semi-colons, e.g., `err : X ; u : Carr * Int => X`. Type declarations may be omitted (writing, e.g., `err, u`), in which case types are derived automatically; if this fails, Ctrl aborts. Rules have the form $\mathtt{term}_1$ `->` $\mathtt{term}_2$ `[constraint]` where both $\mathtt{term}_1$ and $\mathtt{term}_2$ are well-typed terms on variables and declared symbols (values, theory functions or TRS functions), and `constraint` is a term of sort `Bool`, not containing TRS functions.

Rules must be separated by semi-colons, and constraints may be omitted. For example: `slen(s) -> u(s,0) ; u(s,i) -> err [i < 0 or size(s) <= i]`.

Finally, *query* determines the action Ctrl should take, as detailed in Sec. 4.

```
INCLUDE theories
DECLARE
   signature
WELLFOUNDED names
CHAIN chainings
SMT-TRANSLATIONS
   translations
```

Fig. 2: Theory File

The shape of a theory file is given in Figure 2. Theory files, in order to be used, are expected to be in the `theories/` subdirectory in the program folder. A theory can extend another theory (effectively including all its symbols) using the `INCLUDE` field, e.g. `INCLUDE ints`; it is recommended to include at least `core`, which contains symbols like `true`, `false`, `and`, `or` and `not`. The `DECLARE` field corresponds to `SIGNATURE` in an input file; here, all theory functions and values must be listed, along with their type declaration. The symbols listed after `WELLFOUNDED` should all be well-founded relations, i.e. symbols $R : [\iota \times \iota] \Rightarrow$ `Bool` such that no infinite sequence $s_1 R s_2 R \ldots$ exists; this is used for termination analysis. `CHAIN` is used to define syntactic sugar, allowing, e.g., `x > y > z` to be shorthand for `x > y and y > z`. Finally, `SMT-TRANSLATIONS` allows users to assign a meaning to custom symbols. That is, if a theory symbol was declared which is not typically supported by SMT-solvers for this theory – such as `positive(a)` from the previous section – we may instead express its meaning as an SMT-term (e.g. `(forall ((x Int)) (or (< x 0) (>= x (size a)) (> (select a x) 0)))`; of course, in this case the `LOGIC` must support quantifiers).

Note that Ctrl itself does not know much theory: aside from basic properties on the core theory (i.e. symbols like `and` and `or`) and minor reasoning on integers, all calculations and validity questions which arise during a program run are passed to the given SMT-solver (which must be present in the folder from which Ctrl is invoked), along with the given `LOGIC` field. This makes it possible to handle arbitrary theories. If no solver is given, the default SMT-solver called smtsolver in the program directory is automatically used; this is currently Z3 [1].

To support realistic systems, Ctrl provides three constructions to declare infinitely many values at once. A declaration `!INTEGER :` *sort* causes all integer symbols to be read as values with sort *sort*. Similarly, `!ARRAY!`α, with α the name of a sort, includes all sequences of the form $\{a_1 \texttt{:} \ldots \texttt{:} a_n\}$ where each a_i is a value of sort α. `!MIXED!`o`!`c, with o and c strings, includes all strings of the form $o\langle string \rangle c$.[3] The string values are passed to the SMT-solver *without* the "bracketing" o, c. As it is not needed that each integer/array/string represents a value, these constructions allow you to support arbitrary types; for instance:

- `!INTEGER : Byte` (but users should make sure the input file only includes integers in $\{0, \ldots, 255\}$, and the SMT-solver only returns such numbers);
- `!ARRAY!ARRAY!Int : Matrix` (values would be, e.g., {{`1,3`}:{`2:1`}});
- `!MIXED!"!" : Real` (values would be, e.g., `"3.14"`, and passed to the SMT-solver as `3.14`; Ctrl does not assume all values can be represented).

[3] However, to avoid ambiguity in the input parser, the brackets and individual strings in the input file may not use the protected symbols `[`, `,` and `;`, or spaces.

To demonstrate how the various fields and constructions are used, the Ctrl download at `http://cl-informatik.uibk.ac.at/software/ctrl/` contains both example input files (in the `examples/` folder) and theories (in `theories/`).

Comment: Instead of an external SMT-solver, users might set the `SOLVER` to `manual`, which indicates that they will manually perform calculations and satisfiability or validity checks; or to `internal`, which causes Ctrl to attempt simplifying formulas with booleans, integers and integer arrays itself before passing any remaining problems to smtsolver. This gives a speedup by avoiding external calls in many cases. The internal solver is consistent with the core and ints theories in `http://smtlib.cs.uiowa.edu/theories.shtml` .

4 Queries

Ctrl is a generic tool for constrained rewriting, designed to solve a variety of problems (as requested by the `QUERY` field). We consider the possibilities. Note that example uses of all queries are available in the Ctrl download.

Simplifying. The literature offers several translations from restricted imperative programs to constrained rewriting (see e.g. [2,12]), enabling the analysis of imperative languages with rewriting techniques. Initially, this often gives large and somewhat impractical systems. Ctrl's simplification module (invoked using `simplification [f1 ... fn]`) simplifies such LCTRSs, chaining together rules and removing unused arguments, but leaving the symbols $\mathtt{f}_i$ untouched. For instance, $\{f(x,y) \to g(x,0)\ [\varphi],\ g(x,y) \to h(x+y,x,x*y),\ h(x,0,y) \to f(x,x)\ [x<0]\}$ becomes $\{f(x,y) \to h(x+0,x)\ [\varphi],\ h(x,0) \to f(x,x)\ [x<0]\}$.

Reducing. Ctrl can reduce both terms (using the SMT-solver to test whether constraints are satisfied and to do calculations), and *constrained terms*, which intuitively indicates how *groups* of terms are reduced, following [11, Sec. 2.1]. For example, `fib(n) [n > 3]` → `fib(n - 1) + fib(n - 2) [n > 3]` proves that *all* terms `fib(`s`)` with s a value > 3 can be reduced as given. Ctrl reduces (constrained) terms using an innermost strategy to normal form, or until the SMT-solver simply fails to verify that any specific rule can be applied.[4] In a non-terminating LCTRS, it is possible that evaluation never ends.

Boolean Properties Ctrl tests three properties which apply to the full LCTRS:

- Confluence: $\forall s,u,v[(s \to^* u \wedge s \to^* v) \Rightarrow \exists w[u \to^* w \wedge v \to^* w]]$; put differently, *how* we reduce a term does not affect the results we can obtain.
- Quasi-reductivity: all irreducible ground terms are built entirely of *constructor symbols*: values and TRS functions f where no rule $f(\boldsymbol{\ell}) \to r\ [\varphi]$ exists.
- Termination: there is no infinite reduction $s_1 \to s_2 \to s_3 \to \ldots$.

All three are undecidable yet commonly studied properties in the world of (un-

[4] This failure is not unlikely, as constrained reduction following [11] requires validity of quantified formulas $\exists \boldsymbol{x}[\varphi(\boldsymbol{x})]$, which is hard for most solvers. To improve performance, Ctrl uses default choices for $\boldsymbol{x}$; this method is omitted here for space reasons.

constrained) TRSs. Techniques to verify them often extend naturally to LCTRS.

For confluence, Ctrl tests the sufficient condition of *orthogonality* [10]. This property is straightforward to check – testing satisfiability of formulas which are little more complicated than the rule constraints – yet captures a large and natural collection of LCTRS, as typical functional programs are orthogonal. LCTRSs obtained from imperative programs and simplified are usually orthogonal as well, provided variables are obviously instantiated before they are used.

For quasi-reductivity, Ctrl uses the nameless but powerful algorithm described in [12]. Termination uses a combination of techniques, described in Sec. 5.

Equivalence. Finally, Ctrl has a module on *inductive theorem proving*, which can help a user prove reducibility between two groups of terms, either automatically or in an interactive mode. This is explained in more detail in Sec. 6.

5 Termination

Termination is the property that, regardless of the order and position in which rules are applied, evaluation of every term ends eventually. Many termination methods for unconstrained TRSs rely on the *dependency pair framework* [8], a powerful approach which enables modular use of many sub-techniques.

While this framework extends naturally to constrained rewriting [9], the ordering methods which form a core part unfortunately do not – or rather, they are useful in theory, but automation fails in the presence of infinitely many values. Consider for example a TRS with a rule `f(x,y)` $\to$ `f(x - 1, y + 2) [x > 0]`. To see that it terminates, we must know that there is no infinite sequence $x_1, x_2, \ldots$ where each $x_i > x_{i+1}$ and $x_i > 0$. This we cannot express as a constraint over integer arithmetic: rather, it requires domain-specific knowledge.

Here, the `WELLFOUNDED` declaration comes in. Ctrl will test whether arguments decrease with respect to any given well-founded relation. To handle the example above, we may introduce a custom symbol $>! : [\texttt{Int} \times \texttt{Int}] \Rightarrow \texttt{Bool}$, and translate $x >! y$ to $x > y \wedge x \geq 0$. Currently, the stronger *polynomial interpretations* are limited to the integers, but we intend to generalise this in the near future.

Practical Results. There is no database of LCTRS termination problems, but there *are* large collections of integer TRSs (ITRSs) and transition systems (ITSs) in the *termination problem database* (see `http://termination-portal.org/wiki/TPDB`), both of which can be translated to LCTRSs.[5] Figure 3 shows Ctrl's power on these benchmarks, evaluated with a 1-minute timeout. Here, *Time* indicates the average runtime in seconds, disregarding timeouts. Ctrl currently has no non-termination functionality.

	ITRSs	**ITSs**
Yes	85	371
Maybe	29	455
Timeout	3	396
Time	0.85	6.88

Fig. 3: results on the TPDB

Ctrl's apparent weakness on ITSs is partly caused by the greater size of many benchmarks, and partly due to non-termination: many of them

[5] The translation for integer transition systems uses a variation of Marc Brockschmidt's SMT-Pushdown tool at `https://github.com/mmjb/SMTPushdown`.

only terminate if you fix a given start symbol. Ctrl proves the stronger property of termination for *all* terms. Consequently, it performs somewhat worse than dedicated tools for ITSs like T2 (`http://research.microsoft.com/en-us/projects/t2/`). In addition, neither integer rewriting nor termination are the main focus of Ctrl: the primary goal is generality. In the future, we hope to add further termination techniques; both general and theory-specific ones.

6 Equivalence

Finally, *equivalence* studies the question whether two groups of terms are reducible to each other; this is done in the form of *equations* $s \approx t\ [\varphi]$. For instance $f(x,y) \approx g(x,z)\ [x > y \wedge x > z]$ is an inductive theorem if for all values x, y, z such that $x > y \wedge x > z$ holds in the underlying theory, $f(x,y) \leftrightarrow^*_{\mathcal{R}} g(x,z)$. In a confluent, terminating system, this exactly means that they reduce to the same normal form. If $f(x_1,\ldots,x_n) \approx g(x_1,\ldots,x_n)\ [\varphi]$ is an inductive theorem, then f and g define the same function (under the conditions dictated by φ), which could be used in practice to replace (parts of) functions by optimised variations.

Unfortunately, this is a hard problem to solve automatically, even for quite simple systems. Ctrl uses *rewriting induction* [11], a method introduced in [14] which relies on termination of $\to_{\mathcal{R}}$ for the induction principle. There are a number of inference rules to simplify equations, but the key to successful rewriting induction is guessing suitable *lemma equations*, for which no single obvious method exists (although many techniques exist to capture certain kinds of systems).

Ctrl offers two ways of testing equivalence: automatic and interactive. In interactive mode, the user manually chooses inference rules to apply, using Ctrl to guard applicability of these steps and allowing "`auto`" steps to do obvious simplifications. Beside the basic steps, a lemma generation method is included: generalise, which is especially useful for the LCTRSs obtained from imperative programs, by focusing on loop counters. Figure 4 shows an example LCTRS comparing a recursive and iterative calculation of the factorial function. The Ctrl solution is: `auto`, `swap`, `expand`, `auto`, `auto`, `expand`, `auto`, `generalise`, `expand`, `auto`, `auto`. To see these commands in action, download the tool and run it on `examples/ft.ctrs`.

$$
\begin{array}{rll}
\mathtt{ft1}(x) & \to \mathtt{1} & [x \leq \mathtt{0}] \\
\mathtt{ft1}(x) & \to x * \mathtt{ft1}(x-\mathtt{1}) & [x > \mathtt{0}] \\
\mathtt{ft2}(x) & \to \mathtt{u}(x,\mathtt{1},\mathtt{1}) & \\
\mathtt{u}(x,i,z) & \to \mathtt{u}(x,i+\mathtt{1},z*i) & [i \leq x] \\
\mathtt{u}(x,i,z) & \to z & [i > x]
\end{array}
$$

Goal: $\mathtt{ft1}(x) \approx \mathtt{ft2}(x)\ [\mathtt{true}]$

Fig. 4: Example LCTRS problem.

The automatic mode requires no user interaction (although lemma equations can be added in the input file), but combines some heuristics with backtracking to obtain a proof. Ctrl can automatically handle quite complicated examples, as evidenced by the results in [11] (`http://cl-informatik.uibk.ac.at/software/ctrl/aplas14/`): on 7 groups of manually translated student homework programs, Ctrl could automatically prove correctness of two thirds. This includes array / string functions such as `strcpy` or summing array elements. To our knowledge, there are no other provers which can handle systems like Fig. 4.

7 Conclusions

We have discussed Ctrl, a versatile tool for constrained rewriting. A key focus of Ctrl is *generality*: the functionality is not limited to, e.g., linear integer arithmetic, but supports almost any theory, provided an SMT-solver is available for it. This makes it possible to use Ctrl in many different settings; once support is available, we could for instance use it to analyse confluence of oriented mathematical equations over the real number field, termination of functional programs with mappings as core objects, or equivalence of imperative string functions.

What is more, the techniques themselves are designed with extension in mind, allowing for more sophisticated techniques to be added in the future. Another obvious future work (which is already in progress) is to translate reasonable subsets of certain imperative languages into LCTRSs automatically.

The version of Ctrl used in this work, and evaluation pages for the experimental results on termination and equivalence, are available at:

`http://cl-informatik.uibk.ac.at/software/ctrl/lpar15/`

References

1. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. TACAS*, 2008.
2. S. Falke and D. Kapur. A term rewriting approach to the automated termination analysis of imperative programs. In *Proc. CADE*, 2009.
3. S. Falke and D. Kapur. Rewriting induction + linear arithmetic = decision procedure. In *Proc. IJCAR*, 2012.
4. S. Falke, D. Kapur, and C. Sinz. Termination analysis of C programs using compiler intermediate languages. In *Proc. RTA*, 2011.
5. Y. Furuichi, N. Nishida, M. Sakai, K. Kusakari, and T. Sakabe. Approach to procedural-program verification based on implicit induction of constrained term rewriting systems. *IPSJ Transactions on Programming*, 1(2):100–121, 2008. In Japanese.
6. J. Giesl, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann. Proving termination of programs automatically with AProVE. In *Proc. IJCAR*, 2014.
7. J. Giesl, M. Raffelsieper, P. Schneider-Kamp, S. Swiderski, and R. Thiemann. Automated termination proofs for Haskell by term rewriting. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 33(2):7:1–7:39, 2011.
8. J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Mechanizing and improving dependency pairs. *Journal of Automated Reasoning*, 37(3):155–203, 2006.
9. C. Kop. Termination of LCTRSs. In *Proc. WST*, 2013.
10. C. Kop and N. Nishida. Term rewriting with logical constraints. In *Proc. FroCoS*, 2013.
11. C. Kop and N. Nishida. Automatic constrained rewriting induction towards verifying procedural programs. In *Proc. APLAS*, 2014.
12. C. Kop and N. Nishida. Towards verifying procedural programs using constrained rewriting induction. Technical report, 2014. `http://arxiv.org/abs/1409.0166`.
13. C. Otto, M. Brockschmidt, C. von Essen, and J. Giesl. Automated termination analysis of Java Bytecode by term rewriting. In *Proc. RTA*, 2010.
14. Uday S. Reddy. Term rewriting induction. In *Proc. CADE*, 1990.