KBCV 2.0 – Automatic Completion Experiments[*]

Thomas Sternagel

University of Innsbruck, Innsbruck, Austria
`thomas.sternagel@uibk.ac.at`

Abstract

This paper describes the automatic mode of the new version of the Knuth-Bendix Completion Visualizer. The internally used data structures have been overhauled and the performance was dramatically improved by introducing caching, parallelization, and term-indexing in the computation of critical pairs and simplification. The new version is much faster and can complete three more systems.

1 Introduction

The *Knuth-Bendix Completion Visualizer* (KBCV) is an interactive/automatic tool for Knuth-Bendix completion and equational logic proofs. The basic functions of the previous release are described in detail in [5, 7]. This paper addresses implementation issues to improve the performance of the automatic completion mode and reports on experiments of the new release KBCV 2.0. The tool is available under the *GNU Lesser General Public License 3* at

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

In the sequel we assume familiarity with term rewriting, and completion [1]. Nevertheless we recall the basics.

Completion is a procedure which takes as input a (finite) set of equations $\mathcal{E}$ and a reduction order $>$ (or it tries to construct this reduction order on the fly with the help of an external termination tool, see [8]) and attempts to construct a terminating and confluent term rewrite system (TRS) $\mathcal{R}$ with the same equational theory as $\mathcal{E}$. In case the completion procedure succeeds, two terms are equivalent with respect to $\mathcal{E}$ if and only if they reduce to the same normal form with respect to $\mathcal{R}$, that is, $\mathcal{R}$ represents a decision procedure for the word problem of $\mathcal{E}$.

The computation is done by generating a finite sequence of intermediate TRSs which constitute approximations of the equational theory of $\mathcal{E}$. Following Bachmair and Dershowitz [2] the completion procedure can be modeled as an inference system (see Figure 1). The inference rules work on pairs $(\mathcal{E}, \mathcal{R})$ where $\mathcal{E}$ is a finite set of equations and $\mathcal{R}$ is a finite set of rewrite rules. The goal is to transform an initial pair $(\mathcal{E}, \varnothing)$ into a pair $(\varnothing, \mathcal{R})$ such that $\mathcal{R}$ is terminating, confluent and equivalent to $\mathcal{E}$. In our setting a completion procedure based on these rules may succeed (find $\mathcal{R}$ after finitely many steps), loop, or fail. In Figure 1 a reduction order $>$ is provided as part of the input. We use $s \overset{\sqsupset}{\to}_{\mathcal{R}} u$ to express that s is reduced by a rule $\ell \to r \in \mathcal{R}$ such that ℓ cannot be reduced by another rule with left-hand side s. The notation $s \dot{\approx} t$ denotes either of $s \approx t$ and $t \approx s$.

KBCV internally uses indexed equations $i\colon l \approx r$ and rules $j\colon l \to r$, where i and j are unique positive integers and l and r are terms, called the left- and right-hand side respectively.

[*]Supported by the Austrian Science Fund (FWF) international project I963 and the Japan Society for the Promotion of Science.

$$\textsc{Deduce}\ \frac{(\mathcal{E},\mathcal{R})}{(\mathcal{E}\cup\{s\approx t\},\mathcal{R})}\ \text{if } s\ {}_{\mathcal{R}}\!\leftarrow u \rightarrow_{\mathcal{R}} t \qquad \textsc{Orient}\ \frac{(\mathcal{E}\cup\{s\mathrel{\dot{\approx}}t\},\mathcal{R})}{(\mathcal{E},\mathcal{R}\cup\{s\rightarrow t\})}\ \text{if } s>t$$

$$\textsc{Compose}\ \frac{(\mathcal{E},\mathcal{R}\cup\{s\rightarrow t\})}{(\mathcal{E},\mathcal{R}\cup\{s\rightarrow u\})}\ \text{if } t\rightarrow_{\mathcal{R}} u \qquad \textsc{Delete}\ \frac{(\mathcal{E}\cup\{s\approx s\},\mathcal{R})}{(\mathcal{E},\mathcal{R})}$$

$$\textsc{Collapse}\ \frac{(\mathcal{E},\mathcal{R}\cup\{s\rightarrow t\})}{(\mathcal{E}\cup\{u\approx t\},\mathcal{R})}\ \text{if } s\ \overset{\sqsupset}{\rightarrow}_{\mathcal{R}}\ u \qquad \textsc{Simplify}\ \frac{(\mathcal{E}\cup\{s\mathrel{\dot{\approx}}t\},\mathcal{R})}{(\mathcal{E}\cup\{u\mathrel{\dot{\approx}}t\},\mathcal{R})}\ \text{if } s\rightarrow_{\mathcal{R}} u$$

Figure 1: The inference rules of *completion*.

2 Optimizing Automatic Completion

KBCV 2.0 is implemented in Scala 2.10.0,[1] an object-functional programming language which compiles to Java bytecode. For this reason KBCV is portable and runs on Windows and Linux machines. The developed term library (`scala-termlib`, available from KBCV's homepage) was completely overhauled and consists of approximately 2100 lines of code. The new KBCV builds upon this library and has an additional 5000 lines of code.

The main goal for this release was to improve the performance of KBCV especially in automatic mode. Some more details on the automatic mode can be found in [5, Section 5.2.1] and [7, Section 2.2]. Looking at the flow chart of the automatic mode depicted in Figure 2 we first had to identify critical parts, where speed-up would be possible.

1. The procedure starts in the SIMPLIFY-phase, where both sides of equations are rewritten as far as possible.

2. Trivial equations, that is, equations where both sides are the same are dropped in the DELETE-phase.

3. The third phase checks if $\mathcal{E}$ is empty and if all critical pairs between left-hand sides of rules in $\mathcal{R}$ are joinable.[2]

4. Then the procedure chooses a single equation which it tries to orient. The used heuristic is to select an equation where the length of the left- and right-hand sides is minimal. The cost for orientation mainly depends on the used termination tool.

5. Now in the COMPOSE-phase the procedure simplifies all right-hand sides of rules as far as possible.

6. After that, in the COLLAPSE-phase, it tries to simplify left-hand sides of rules.

7. Finally DEDUCE computes critical pairs and adds them to the set of equations.

From this assessment we see that (2) is trivial and already very fast and (4) mainly depends on an external program. So we focus on the remaining phases. In the sequel we will sometimes refer to (3) and (7) collectively as *critical pair computation* and to (1), (5), and (6) as *simplification*.

[1] `http://www.scala-lang.org/`

[2] The computation and check for joinability of critical pairs is only needed because of KBCV's interactive mode in which inference rules may be fired in an arbitrary order.

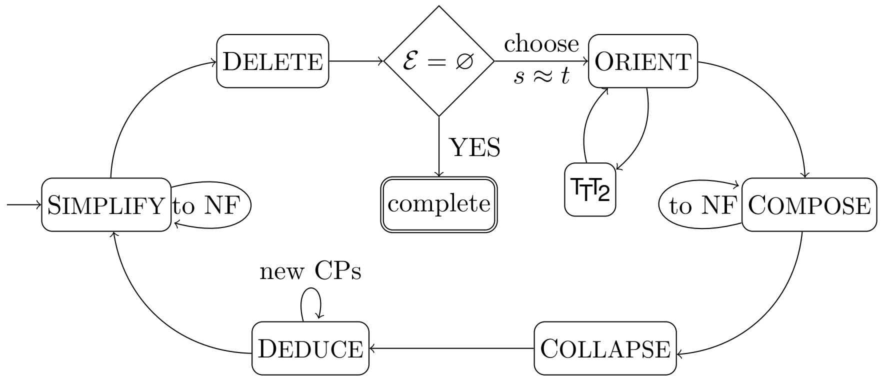

Figure 2: KBCV's automatic completion procedure.

The first idea (which was already partly implemented in versions 1.7 and 1.8 of KBCV) was to prevent re-computation by introducing caching. Next the independent parts of the procedure were parallelized to make the most of modern multi-core/processor architectures. Finally we also introduced term-indexing in order to speed up unification and matching of terms. These steps are described in some more detail in the next three sections. We compare the resulting speed-ups for various combinations of these methods in Section 3.

2.1 Caching

In order to avoid redundancy in critical pair computations and simplifications we introduced four new data structures for caching.

Each time a critical pair is computed KBCV stores the pair of indices of the overlapping rules which caused the new equation. The next time automatic completion has to compute critical pairs (in phases (3) or (7) of the procedure) it only computes critical pairs from overlaps which are not already stored.

We use three different caches for COMPOSE, COLLAPSE, and SIMPLIFY respectively. The first cache has an entry for each rule. In this entry we store the set of indices of rules which have already been tried to simplify this rule. Next time automatic completion has to simplify a rule it only tries the rules which are not cached yet. The other two caches work just in the same way.

2.2 Parallelization

While automatic completion (Figure 2) executes the single phases sequentially, within a phase there are completely independent computations which can be parallelized.

- DEDUCE: The computation of critical pairs.
- COMPOSE: The composition of rules.
- COLLAPSE: The collapsing of rules.
- SIMPLIFY: The simplification of equations.

In order to get the most out of modern multi-core architectures we re-implemented those four phases. Now each single step (e.g. the computation of critical pairs between two particular

	KBCV-b-i-u	KBCV-i-u	KBCV-b-u	KBCV-b-i	KBCV-u	KBCV-i	KBCV-b	KBCV
completed	85	87	85	85	89	90	85	90
total time	1142.6	512.8	498.0	384.5	1163.4	1321.3	321.8	1116.2
avg. time	13.4	5.9	5.9	4.5	13.1	14.7	3.8	12.4
AD93_Z22		83.9			44.7	41.8		32.9
BGK94_D16		30.5			25.7	25.6		23.2
BGK94_Z22W					598.7	220.6		201.6
LS94_G1						583.6		514.5
SK90_3.09					168.1	161.1		90.1

Table 1: Experimental results on 115 systems, timeout: 600s.

rules, or one specific rewrite step on one side of an equation) are separate computations which can be handled by a pool of worker threads. The main program waits until all results are computed and then continues with the non-parallel part of the procedure.

2.3 Term Indexing

Both unification of terms (needed for the computation of critical pairs) and matching (needed for rewriting of terms) can get very expensive for large systems with large left-hand sides of rules. To counteract that we now store the left-hand sides of rules in a discrimination tree (see for example [4]) which allows for very fast filtering of so called *candidate sets* (which are typically very small). Getting a unifiable or matching term from this candidate set is much faster than checking all left-hand sides of rules.

3 Experiments

The experiments we describe here were carried out on a 64bit GNU/Linux machine with 48 AMD Opteron^{TM} 6174 processors and 315 GB of RAM. The kernel version is 2.6.32. The version of Java on this machine is 1.7.0_03. For the JVM we limited the stack size for each thread to 10MB, set the initial heap size to 1GB, and the maximum heap size to 2GB. The test-bed we worked with consists of 115 systems from the distribution of MKBTT.[3] KBCV was launched using the following flags:

```
./kbcv -a -p -s 600 -m "./ttt2 -cpf xml - 1" <inputfile>
```

Here the `-a` flag tells KBCV to switch to automatic mode, `-p` causes KBCV to output the CPF proof of completion on `stdout`, `-s 600` sets the timeout to 600 seconds and finally `-m` sets the termination-check method to use, in our case calls to the external termination tool TTT2. There are three more flags we used in the experiments: `-b` disables caching, `-i` disables term-indexing, and `-u` disables parallelization. The tool instances where parallelization was enabled used all of the 48 processors.

The upper part of Table 1 gives the number of completed systems, the total time needed to complete them and the average time for each of the completed systems for different configurations of KBCV. The lower part lists systems which only certain configurations of KBCV could complete together with the time. The detailed experiments are available online.[4] Here each

[3]http://cl-informatik.uibk.ac.at/software/mkbtt/index.php
[4]http://cl-informatik.uibk.ac.at/software/kbcv/experiments/kbcv2/

column is labeled with KBCV plus the set flags. So the first column labeled KBCV-b-i-u gives the results for KBCV without caching, term-indexing, and parallelization, while the last column shows the results for KBCV using all three methods. What we see is that without optimization KBCV can complete 85 out of the 115 systems and the average time for that is 13.4 seconds per system. Without caching (columns three, four, and seven) we are not able to complete additional systems, although we achieve a speed-up of about 2.6 using only term-indexing or parallelization, and 3.5 using both of these methods. Only using caching (column two) already establishes two more systems with a speed-up for the initial 85 systems of about 2.9. Caching plus term-indexing (column five) already yields two more successful systems and a speed-up with respect to the 85 systems of about 3.5. If we combine caching with parallelization (column six) we get yet another system and a speed-up for the initial systems of about 4.0. Finally KBCV using all three methods achieves a speed-up of 4.5 for the initial 85 systems. All found proofs have been certified by CeTA [6].

4 Conclusion

Three different methods to enhance KBCV's automatic completion procedure have been investigated and compared. We have seen that these methods, most notably caching, achieve a huge performance boost for the automatic completion procedure of KBCV 2.0.

If we look at the 115 systems we tested, we see that most of them only consist of about 10 to 20 rules and that the left-hand sides of those are also pretty small. When we work with much larger systems with more complicated left-hand sides parallelization and term-indexing become more and more important. We for example tried to only compute critical pairs for a subset of the HOL Light [3] simpset (about 3000) rules. Without parallelization we had to cancel the experiment after several days. Using parallelization KBCV was able to compute the 300,000 critical pairs in less than two hours.

A next step to further push the procedure would be to investigate different heuristics for the selection of equations in the ORIENT-phase.

References

[1] Franz Baader and Tobias Nipkow. *Term rewriting and all that.* Cambridge University Press, 1998.

[2] Leo Bachmair and Nachum Dershowitz. Equational inference, canonical proofs, and proof orderings. *J. ACM*, 41:236–276, 1994.

[3] John Harrison. HOL Light: A tutorial introduction. In *FMCAD*, pages 265–269, 1996.

[4] I. V. Ramakrishnan, R. C. Sekar, and Andrei Voronkov. Term Indexing. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 1853–1964. Elsevier and MIT Press, 2001.

[5] Thomas Sternagel. Automatic proofs in equational logic. Master's thesis, University of Innsbruck, 2012.

[6] Thomas Sternagel, René Thiemann, Harald Zankl, and Christian Sternagel. Recording completion for finding and certifying proofs in equational logic. In *IWC'12*, pages 31–36, 2012.

[7] Thomas Sternagel and Harald Zankl. KBCV - Knuth-Bendix completion visualizer. In *Proceedings of the 6th International Joint Conference on Automated Reasoning*, volume 7364 of *LNAI*, pages 530–536. Springer-Verlag, 2012.

[8] Ian Wehrman, Aaron Stump, and Edwin Westbrook. Slothrop: Knuth-Bendix completion with a modern termination checker. In *RTA'06*, pages 287–296. Springer-Verlag, 2006.