Recent Developments in the Matchbox Termination Prover

Alexander Bau * **, Tobias Kalbitz, Maria Voigtländer, and Johannes Waldmann**

Fakultät IMN, HTWK Leipzig, Germany

Abstract

We report on recent and ongoing work on the Matchbox termination prover:

- a constraint compiler that transforms a Boolean-valued Haskell function into a Boolean satisfiability problem (SAT),
- a constraint solver for real and arctic matrix constraints that is using evolutionary optimization, and is running on massively parallel (graphics) hardware.

1998 ACM Subject Classification F.1.1 Models of Computation

Keywords and phrases rewriting, termination, constraint programming

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

1 Introduction

The program Matchbox [11] originally proved termination of string rewriting, using the method of match bounds [6]. The domain was extended to term rewriting, by adding proof methods of dependency pairs [1], matrix interpretations over the integers [4] and over arctic numbers [9].

These methods are typical instances of the following scheme: to automatically find a proof of termination, one solves constraint satisfaction problems. E.g., the precedence of function symbols, or the coefficients of polynomials and matrices, are constrained by the condition that the resulting path order, polynomial order, or matrix order, respectively, is compatible with a given rewriting system.

The constraint system could be solved by

- domain-specific methods (e.g., Matchbox computes a certificate for match-boundedness by completion of automata),
- generic search (exhaustively, randomly, or directed by some fitness function),
- transformation to another constraint domain (e. g., Matchbox transforms integer and arctic polynomial inequalities to a Boolean satisfiability problem, and solves it with Minisat [3]).

In the present paper, we report on recent and ongoing work to

- extract a general framework for constraint programming by automatic transformation to SAT,
- and (independently) add a domain-specific solver for real and arctic matrix constraints that is using evolutionary optimization, and is running on massively parallel (graphics) hardware.

* Alexander Bau is supported by an ESF grant

Conference title on which this volume is based on.
Editors: Billy Editor, Bill Editors; pp. 1–4

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

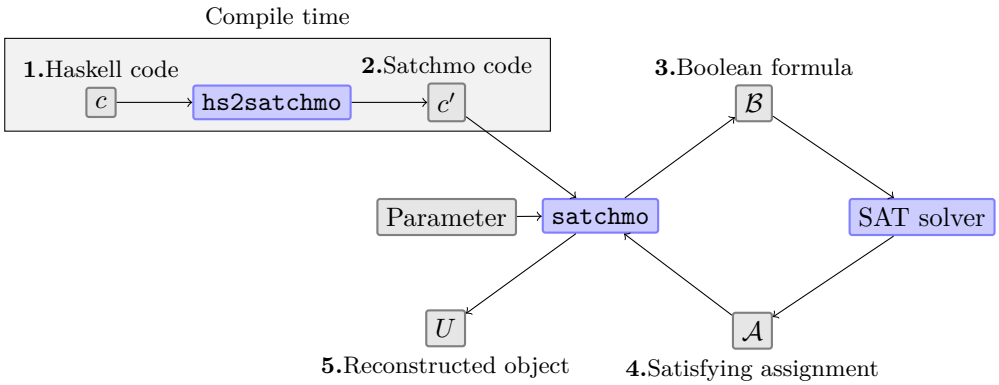
2 A Constraint Compiler

The idea behind constraint programming is to separate specification (encoded as constraint system) from implementation (the constraint solver). The obvious choice for the specification language is mathematical logic, equivalently, a pure (i.e. side-effect free) functional programming language like Haskell. A constraint system c can be seen as a function with type $c : U \to \texttt{Bool}$, where the solution is an object $s \in U$ such that $c(s)$ is True.

There are clever solvers for the case that $U' = \texttt{Bool}^*$, and c' is given by a formula in propositional logic. But typically, the application domain U is different. The translation from U to U' can be done manually by the programmer, or automatically, by some tool. The satchmo library (`http://hackage.haskell.org/package/satchmo`) used in Matchbox is an example for the "manual" approach. It is an embedded (in Haskell) domain specific language for the generation of SAT constraints. Several termination researchers built and published similar libraries for other host languages (Ocaml, Java). These interweave the generation of the boolean formula with the declaration of the constraint system in the host language. E.g., satchmo generates the formula as a side effect represented by a suitable State monad. Actually these are different processes and should be separated from each other.

Therefore Alexander Bau is building a constraint compiler that inputs source code of a Haskell function $c : U \to \texttt{Bool}$, as explained above, and produces a satchmo program. The domain U may use structured data types like tuples and lists, in addition to primitive types like booleans and integers. c may also depend on run-time parameters that are not known at compile time.

A prototypical use case is the search for a precedence that defines a lexicographical path order (LPO) that is compatible with a term rewriting system (TRS). In this case the constraint system consists of a Haskell implementation of LPO : TRS → Precedence → `Bool`. LPO applied to a TRS R and a precedence p returns `true`, iff lpo(p) is compatible with R. The first parameter (R) of LPO is known at run-time while the second (p) is not.



The constraint system is given as a program (a set of declarations) in a subset of Haskell. The constraint compiler performs a type-directed transformation, where the type system is an extension of the Damas-Milner type system [2] [10]. We additionally annotate each type constructor with a flag that indicates whether its value is known (as a parameter given at run-time) or unknown (and therefore has to be determined by the constraint solver).

We plan to extend the type system further, to take into account resource bounds [7]. E.g., we want to be statically certain that the size of the generated SAT constraint system is polynomially bounded in the size of the (known) input parameters.

3 Massively Parallel Constraint Solving

A constraint satisfaction problem can be converted into an optimization problem that is solved by evolutionary algorithms. For the domain of matrix interpretations, this approach was used by Dieter Hofbauer's termination prover *MultumNonMulta* (2006, 2007), see also [5]. We return to it now, since it allows for massive parallelisation.

In the context of numerical constraint solving by randomized, directed search, parallel processing is applicable because

- basic operations (on numbers) can be executed fast
- domain specific operations (matrix multiplications) can be sped up by parallelism (multiplication of n dimensional square matrices, using n^2 cores and n time)
- evolutionary search strategies can be sped up again, by treating several individuals in parallel (e.g., computing their fitness values)

General Purpose Graphical Processing Units (GPGPUs) provide massively parallel processing at affordable prices. Tobias Kalbitz and Maria Voigtländer are implementing matrix constraint solvers for CUDA capable graphics cards. CUDA (Compute Unified Device Architecture) [8] is a parallel programming model for NVIDIA's GPGPUs.

The following approach is used to find a strictly monotone matrix interpretation of dimension d that is compatible with a string rewriting system R over alphabet Σ (weakly compatible with each rule, and strictly compatible with at least one rule):

- A population consists of several individuals, each individual is a matrix interpretation, that is, a mapping $[\cdot] : \Sigma \to \mathbb{N}^{d\times d}$, where for each $a \in \Sigma$, the first column of $[a]$ is $(1, 0, \ldots, 0)^T$, and the last row of $[a]$ is $(0, \ldots, 0, 1)$. This condition ensures monotonicity.
- The *fitness* of an interpretation $[\cdot]$ is $\sum\{\max(0, [r]_{p,q} - [l]_{p,q})^2 \mid (l, r) \in R, 1 \leq \{p, q\} \leq d\}$, plus some very large penalty in case that $\neg\exists(l, r) \in R : [l]_{1,d} > [r]_{1,d}$. Lower fitness values are better, and value zero indicates that compatibility holds.
- An individual with fitness > 0 is changed by a *large* mutation: we randomly pick some $(l, r) \in R, 1 \leq \{p, q\} \leq d$ such that $[l]_{p,q} < [r]_{p,q}$, and we choose randomly a sequence of indices $p = p_0, p_1, \ldots, p_n = q$ with $n = |l|$, and then increase each $[a_i]_{p_{i-1},p_i}$ by one, where $l = a_1 \ldots a_n$. This ensures that $[l]_{p,q}$ increases.
- Next, this individual undergoes a series of *small* mutations where for any $a \in \Sigma, 1 \leq i, j \leq d$, the entry at $[a]_{i,j}$ is modified. We try sereval small mutations, until we find one that decreases fitness, and then repeat. The total number of small mutations is bounded.
- The resulting individual is placed back into the population, removing another individual of larger fitness.

▶ **Example 1.** With 1000 individuals, and 100 small steps after each large step, we find a compatible 5-dimensional interpretation for $a^2b^2 \to b^3a^3$ (Problem z001) with < 30.000 large steps with probability $> 50\%$. Of course, the total runtime is not bounded, as the evolution may go into a dead-end. So it is better to re-start than to wait.

Applying this idea to rational and arctic numbers, we meet the following challenges:

Real numbers are approximated by rational ("floating point" values), thus results of comparisons may be wrong. The solution is to introduce a "grid" for rounding input values, e.g. use only integer multiples of $1/2$, or $1/10$, say.

A fine grid implies a smooth objective function, and this may help evolutionary algorithms. On the other hand, a coarse grid reduces the search space, and may increase the chance that we find a solution by luck.

Note that we do not need a grid for arctic numbers, since we can use arctic integers.

On typical CUDA cards, a large number of compute cores is available (e.g., 512). They can only be used efficiently if the data that they process is stored in fast (thread-(block-)local) memory. The amount of such memory is severely limited (e.g., 16 kByte total, resulting in 300 byte per core)

CUDA cards are programmed in (a dialect of) C. This allows fine-grained control, but is highly impractical for large-scale programming. Therefore, we are isolating the low-level details in a C library, and provide it with an interface to Haskell, where we implement global flow of control. Still it is important that data stays on the card's memory, since transport to and from the host computer's memory is slow.

4 Future plans

We stress that the above is a report on ongoing work.

We plan to have an implementation ready for the termination competition in 2012. The code will be open-sourced.

Since the hardware of the competition platform does not include a GPGPU, we will run Matchbox/CUDA remotely.

References

1 Thomas Arts and Jürgen Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236(1-2):133–178, 2000.

2 Luís Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL*, pages 207–212, 1982.

3 Niklas Eén and Armin Biere. Effective preprocessing in sat through variable and clause elimination. In Fahiem Bacchus and Toby Walsh, editors, *SAT*, volume 3569 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2005.

4 Jörg Endrullis, Johannes Waldmann, and Hans Zantema. Matrix interpretations for proving termination of term rewriting. *J. Autom. Reasoning*, 40(2-3):195–220, 2008.

5 Andreas Gebhardt, Dieter Hofbauer, and Johannes Waldmann. Matrix evolutions. In Dieter Hofbauer and Alexander Serebrenik, editors, *Proc. Workshop on Termination, Paris*, 2007.

6 Alfons Geser, Dieter Hofbauer, and Johannes Waldmann. Match-bounded string rewriting systems. *Appl. Algebra Eng. Commun. Comput.*, 15(3-4):149–171, 2004.

7 Jan Hoffmann. *Types with Potential: Polynomial Resource Bounds via Automatic Amortized Analysis*. PhD thesis, Ludwig-Maximilians-Universiät München, 2011.

8 David B. Kirk and Wne mei W. Hwu. *Programming Massively Parallel Processors*. Morgan Kaufmann Publishers, 2010.

9 Adam Koprowski and Johannes Waldmann. Max/plus tree automata for termination of term rewriting. *Acta Cybern.*, 19(2):357–392, 2009.

10 Alan Mycroft. Incremental polymorphic type checking with update. In *LFCS*, pages 347–357, 1992.

11 Johannes Waldmann. Matchbox: A tool for match-bounded string rewriting. In Vincent van Oostrom, editor, *RTA*, volume 3091 of *Lecture Notes in Computer Science*, pages 85–94. Springer, 2004.