The Leipzig autotool System for Grading Student Homework

Mirko Rahn[2] and Johannes Waldmann[1]

[1] Institut für Informatik, Universität Leipzig
Augustusplatz 10, D-04109 Leipzig, Germany
joe@informatik.uni-leipzig.de
[2] Universität Karlsruhe, m.rahn@stud.uka.de

Abstract. The system `autotool` is used at Leipzig University to grade students' homework for courses in theoretical computer science, for exercises on automata (various models), grammars, recursive functions, and graph properties.
The grading involves verification and/or simulation, whose deliberately verbose protocols are sent back to the student. That way, the system allows students to explore the subject of the homework problems by experimentation.
There is an `autotool` subsystem that invites students to send in their own problem instances (within a given problem class) — to be solved by fellow students.
The system is implemented with Haskell, and mainly accessed via an email interface.
The novel aspect of `autotool` design and implementation is that students send their solutions as Haskell code, which then gets dynamically loaded into a grading program. That way, the system makes use of advantages of embedded (over interpreted) domain specific languages. In particular, it uses Haskell syntax and typing.

1 Overview

Our department faces the rapidly rising number of computer science students on the one hand, and the constant (at best) number of teaching staff on the other. Therefore, we can no longer afford the time to weekly grade students' homework.

In order to not give up grading altogether, we did automatize part of it. Now the students send their solutions of selected exercises by electronic mail to a grading program called `autotool`. [?].

The system does not just return a Yes/No answer to the student. The grading of the solution usually involves some verification and/or simulation. The system produces a rather verbose protocol and sends it back to the student.

Also, for each class of problems, we measure a typical parameter of the involved objects (like, the number of states of a finite automaton), and publish the parameters of the correct solutions received so far. This encourages students to seek "better and better" solutions (like, equivalent automata with fewer states).

Additionally, there is an `autotool` subsystem called `challenger` where students can send in their own problem instances (within a given problem class) — that should then be solved by their fellow students.

We did implement and use several sets of exercises related to models of computation (automata, grammars, recursive functions, graph problems).

The novelty in our approach is that the solution that the students send in is in fact the source code of a Haskell module. This will then get linked with the grading program, whose main routine will be executed on–the–fly.

That way we can make use of the advantages of embedded (over interpreted) domain specific languages. In particular, we use Haskell's syntax and typing. This saves us from coding up ad–hoc solutions, and moreover enforces a certain uniformity that is beneficial to the student.

Strict typing is not only a pedagocial aim. Haskell's sharp distinction between values and IO actions (that produce a value) is crucial for the safe execution of the grading program. (We don't want to execute a student's program that first wipes the hard disk, and then returns a Turing machine ...)

In this note, we first give an example of how the system is used; then we describe its architecture. We discuss the benefits and shortcomings of our approach, and emphasize the role of Haskell as the host language.

2 Example system usage

A rather typical exercise in a course on formal languages is this:

Find a context–free grammar for the set of all palindromes in $\{a, b\}^*$.

The student would send the following email:

```
import Grammar
student = Grammar
        { terminals = mkSet "ab"
        , nonterminals = mkSet "S"
        , startsymbol = 'S'
        , rules  = [ ("S", ""), ("S", "a"), ("S", "b")
                   , ("S", "aSa"), ("S", "bSb")
                   ]
        }
```

(In fact we use German identifies and messages, but they have been translated here for readability.) He would then get back an email like this:

```
You sent me
    G = Grammar { terminals = "ab", ... }
    G is indeed a context-free grammar.
Some derivations in G are:
0 steps: ["S"]
1 steps: ["","a","aSa","b","bSb"]
2 steps: ["aa","aaSaa","aaa","abSba","aba"]
```

```
         ...  5 more
...
deriving these terminal words:
0 steps: []
1 steps: ["","a","b"]
2 steps: ["aa","aaa","aba","bab","bb"]
         ...  1 more
...
I will compare L(G) with the target language,
considering only words of length up to 10
and derivations up to length 12.

All (small) words from L(G) are in the target language.
All (small) target words belong to L(G).

Congratulations. As far as I can see, you have solved the exercise.
```

Let's assume now that the student omitted the rules `("S","a"),("S","b")`. The reply would then end like this:

```
Your grammar does not generate (at least) these target words:
    ["a","aaa","aaaaa","aaaaaaa","aaaaaaaaa"]
    ... 57 more
Therefore, you did not solve the exercise.
Please think it over and try again. If you need help, please contact ...
```

To get an impression of how `autotool` is used in a standard undergraduate lecture on automata and formal languages at Leipzig university, see the exercises from [**?**].

3 System architecture

The `autotool` system can handle exercises on:

- grammars (regular, context–free, monotone, arbitrary)
- regular expressions
- automata (finite, pushdown, Turing) (possibly nondeterministic)
- pumping properties of languages
- primitive and partial recursive functions
- graph problems
- two-dimensional languages and automata
- tree languages and automata

For each model, there are *syntactic* functions (e. g. check whether a grammar is in Chomsky normal form), and a simulator for the *semantics* (e. g. generating the set of derivable words).

Note that most interesting semantic properties of these models (e. g., language equivalence of context free grammars) are undecidable, thus simulation and testing cannot be avoided. Whenever possible (e. g., for finite automata), exact algorithms have been implemented.

The system further consists of

– the dispatcher,
– the various grading programs (one per exercise).
– the scoring program (see Section ??)
– the `challenger` interface (see Section ??)

The *dispatcher*

– checks incoming mail for proper `From:` and `Subject:` lines,
– writes the body of the mail (called the *student module*) into a file,
– selects the appropriate *grading module*,
– loads both these modules (and their dependencies) into a Haskell interpreter (usually `hugs`, but `ghci` is an option)
– and calls the grading module's main function.

Each *grading program*

– performs the appropriate syntactic and semantic actions on the data defined in the students module,
– checks the outcome against the specifications of the exercise,
– and writes a report onto a file.

Finally, the dispatcher mails back this report file to the student.

4 Student Competition

While it is enough for the student to just send any correct solution within the given timespan, we maintain a separate ranking table for *small* and *early* correct entries. For each category of exercises, we define a suitable *size* parameter of a solution, like the number of states of an automaton, or the number of rules of a grammar. For each exercise, we keep a table of correct entries, sorted by size (and for equal size, sorted by date). At the end of the semester, we accumulate these tables, and we award prizes to top students.

We found that about 10 percent of all students (about 150, last year) actively took part in this (friendly, but fierce) competition. More often than not, their solutions did surprise even the instructors. On the other hand, the active competition meant heavy work for `autotool`, from time to time showing inefficiencies and bugs in the program.

5 Autonomous Student Competition

The latest addition to `autotool` is an interface, called `challenger`, that allows students to create exercise on their own, for their fellow students to solve.

For many languages (i. e., sets of objects) L, an object O is in L because it has a proof P. For example, a graph is three-colourable because it has a three-colouring; or a formula is satisfiable because there is a satisfying assignment.

Now students send in an object-proof-pair (O, P), for given L. Then `challenger` checks whether P indeed proves O, and then publishes O, *but not* P, as a new exercise. Now other students try to find P' that proves O.

The course instructor initiates the competition by pre-scribing a few objects (thus creating "classical" exercises), and encourages students to modify these problems.

Points will then be awarded for both the originator, and the solver(s) of the exercise. The longer a problem had been "open" before a solution, the more points will be given (to both originator and solver!)

As project work, accompanying a lecture in functional programming, students implemented the `challenger` control interface, and graph modules for Clique, Hamiltonian Circuit, and Colourability. For a detailed description, see [**?**] An extra feature is that the graphs are visualized by GraphViz [**?**].

6 Advantages of our approach

First of all, the system fulfills the goal it was designed for: it grades homework exercises, and thus relieves our staff from some work.

For the students, getting their homework graded automatically is in some respects even better than by a human: they can be sure that the system is always alert and objective, they get immediate response, and they can try as often as they like, and at any time of day.

Moreover, they not only practice the subject of the respective exercise, but they also get used to writing expressions that are correct with respect to syntax and typing. It is much more difficult to enforce this with the classical way of writing on paper.

Although, at present, Haskell is not taught in the basic curriculum, we found that our students are easily able to write correct Haskell code just by copying examples. Actually, all they need to write are constant expressions for records, lists and tuples — see the introductory grammar example.

But of course they can use naturally Haskell's provisions for local (`let, where`) and global definitions, whenever they want to factor out some parts of their solutions. Again this can easily be learnt from examples. Another such extension is to use list comprehensions to reduce tedious typing in some cases, as with the following description of a finite automaton:

```
ring :: Int -> NFA Int
ring n =  NFA
    { states = mkSet [ 1 .. n ]
    , starts = mkSet [ 1 ]
    , finals = mkSet [ 1 ]
    , trans  = collect
       (   (n, 'a', 1  )
       : [ (k, 'a', k+1) | k <- [1 .. n-1] ] ) }
student :: NFA Int
student = ring 5
```

We actually prefer this kind of notation (over explicit listings of states and transitions) because it shows much better the structure of the automaton.

7 Functional Notation for Pumping Properties

The following example shows how we use the functional paradigm to formulate exercises on pumping properties of languages. Recall the statement

$$\begin{aligned} L \in \text{REG} \to\ & \exists n : \forall w \in L : |w| \geq n \to \exists x, y, z \in \Sigma^* : \\ & xyz = w \wedge |xy| \leq n \wedge \forall k \geq 0 : xy^k z \in L. \end{aligned}$$

To prove the pumping property for a language L, students must give the number n, and a function $f : \Sigma^* \to (\Sigma^*)^3$ such that $f(w) = (x, y, z)$ produces a tuple that fulfills the above formula. Using a functional language, students indeed send in the source code for f.

"Officially speaking", this f is the function that is constructed during Skolemization of the formula. (The students learn about this in their logics lecture.) Now `autotool` checks the thruth of a formula by "throwing dice" (i. e., picking a random value) at each $\forall$ quantor, and applying the Skolemized function at each $\exists$ quantor.

Since our undergraduate students don't normally learn Haskell, we cannot make the above method compulsory. But they *can* write a (partial) function as a list of ordered pairs!

However this list had to be rather long (and sometimes infinite) to cover all the results of "throwing dice" that might occur. Therefore we restrict the range of the "random" values by initially seeding the random generator with a fixed value, and repeating the checking for a fixed number of times.

The seed value in fact depends (only) on the student's ID (which is known from his mail address). That way, students get "personalized" exercises. This minimizes copying.

8 Programming details

We use Haskell's type system extensively, to get readable, self explanatory code. Unsurprisingly, types `Set` and `|FiniteMap|`do occur often. The main reason is not efficiency, but rather close correspondence to textbook definitions.

Also the `Pretty` module and `Doc` document type are used throughout `autotool`. Most of the data types are instances of

```
class ToDoc a where toDoc :: a -> Doc
```

As a principle, we output each data item in a form that can also be input again, for instance, the external representation of a set is `mkSet [1, 2, 3]`. (It would be nice to automatically derive these instances.)

In the actual code, we use the hierarchical module namespace, and multi parameter type classes with functional dependencies.

The "scripting" functionality that control the overall `autotool` behaviour (like, reading configuration files, loading the student and grading modules, and doing bookkeeping) is realized with Haskell programs as well.

9 Haskell: efficient, readable, secure

The system was written with severely limited human resources. (In fact, the main goal was to free some of these resources, by delegating tedious grading work to a machine.) The choice of Haskell allowed for very efficient programming. We factored out common code patterns, like finding all descendants of some item, with respect to some (non–deterministic) relation (see the introductory example). Higher order functions were essential for this.

We found that efficiently written Haskell code is, at the same time, very readable as well. Due to referential transparency, it feels as natural as a mathematics text. Most of the model specific syntactic and semantic functions literally correspond to the definitions and algorithms that the students find in their theoretical computer science textbooks. So our code can be shown in classes, as an example of executable specification.

On the other hand, in our graduate program (after *Vordiplom*) we do teach functional programming, and students from that course can practise their skills by contributing `autotool` code.

A system that executes mailed–in code automatically seems to open a huge door to intruders. We think that Haskell nicely helps to keep control.

The foremost way to reject malicious programs is to check whether they are type correct. If they return a *value*, then they cannot have any *effect.* If they want to *do* something ugly, their type would be `IO something`, and that is easily detected by the type checker.

A program could still try to hide some effects using `unsafePerformIO`, but this requires `import IOExts`, and such dangerous imports can be detected by textually scanning the student modules.

It would be overkill however to prohibit all imports and IO operations—sure the grading program needs to do `IO` because it finally writes the report, and perhaps clever students want to import some standard library module.

With this respect it would be handy if Haskell allowed to make a distinction between "trusted" and "untrusted" modules or identifiers (like the "tainted" attribute in the language Perl). These issues also arise if one uses Haskell in other safety critical applications, see for instance the discussion of the concept of *red line* [**?**].

At present, we run the grading tool under a separate account on a separate machine, with system resources restricted by appropriate `ulimit` settings.

10 Conclusion and outlook

The system `autotool` partially replaces the classical process of grading homework solutions by an instructor. We use it since October 2000. Meanwhile, we have a accumulated a set of ≈ 100 exercises that accompanies a typical basic course on Automata Theory, Formal Languages, Computability, and Complexity, plus an advanced course on picture and tree automata. To date, `autotool` has graded ≈ 20.000 submitted solutions.

There were no safety–critical incidents. (Except that one day the system wanted to mail out infinitely long texts of blank characters, due to a bug in `ghc`'s `Pretty` printing library...)

We plan to extend the system

- in scope: add modules for logic (propositional and predicate calculus), for mathematical foundations of computer science (relations, partial orders, lattices), for tree automata, for term rewriting,
- internally: make it more robust and portable, helping students to install `autotool` at home, course instructors to add problems, and porters to add foreign language features;
- visually: add a web based graphics interface that allows students to watch simulations and to navigate through the source code

The choice of Haskell as implementation language was important:

- it serves as an embedding language, providing syntax and types,
- referential transparency allows to use clear mathematical notation,
- functional programming is efficient,
- strict typing and distinction between values and actions guarantee runtime safety.

References

[EN02] John Ellson and Stephen North. Graphviz development home page. `http://www.graphviz.org/`, 2002.

[MLK+02] Georg Martius, Manja Lindemeyer, Alexander Kiel, Jörg Endrullis, and Markus Kreuz. `autotool/challenger` web pages. `http://theopc.informatik.uni-leipzig.de/~challenger/challenger/`, 2002.

[Rei01] Alastair D. Reid. mailing list message (poll: System.exitwith behaviour). `http://www.haskell.org/pipermail/glasgow-haskell-users/2001-May/000504.%html`, 2001.

[RRW01] Mirko Rahn, Alf Richter, and Johannes Waldmann. `autotool` web pages. `http://theopc.informatik.uni-leipzig.de/~autotool/`, 2001.

[Wal01] Johannes Waldmann. Übungsaufgaben zu Automaten und Formalen Sprachen. `http://www.informatik.uni-leipzig.de/~joe/edu/ws01/as/`, 2001.