Automation for Exercises in Computer Science and Mathematics

Johannes Waldmann, HTWK Leipzig

HTWK, 3 May 2018

Example: problem instance

- topic: terms over a given many-sorted signature,
- equivalently, type-correct use of an API

```
write an expression of type  Cherry  , given

Pear c;
static Tomato a ( Pear x , Pear y );
static Tomato b
         ( Cherry x , Cherry y , Tomato z );
static Pear d ( Cherry x);
static Cherry e
         ( Tomato x , Tomato y , Pear z );
```

Example: submission and evaluation

```
infer type for expression: a (c , d (c ))
 function declaration is
  static Tomato a ( Pear x , Pear y )
  number of arguments matches declaration? Yes.
   check argument number 1 [ ... ]
   check argument number 2
    infer type for expression: d (c )
     function declaration is
       static Pear d ( Cherry x )
     number of arguments matches declaration? Yes
     check argument number 1
       infer type for expression: c
         is variable with declaration: Pear c
         has type: Pear
     type of argument matches declaration? No.
```

Example: Conf. of Instance Generator

- teacher sets these parameters

```
Conf { max_arity = 3
     , types = [ Apple, Pear
               , Orange, Cherry, Tomato ]
     , min_symbols = 5 , max_symbols = 5
     , min_size = 7 , max_size = 15
     }
```

- then a generator program will produce problem instances for students

Example: Polymorphic Typing

```
Give an expression of type
 Fozzie<Kermit, Kermit>
in the signature   class S {
 static <T2> Piggy<Piggy<Animal>>
    statler ( Piggy<T2> x , Piggy<T2> y );
 static <T2> Kermit waldorf ( Piggy<T2> x );
 static Piggy<Fozzie<Animal, Animal>> bunsen ( );
 static <T2, T1> T1
    chef ( Piggy<Piggy<T2>> x , Piggy<Piggy<T1>>
 static <T2> Fozzie<Kermit, T2>
    rowlf (T2 x, Animal y );              }
```

```
S.<Kermit>rowlf
 (S.<Fozzie<Animal,Animal>>waldorf
   (S.bunsen()), ...
```

More Examples

- graph “theory”, discrete mathematics:
  - instance: graph G,
    solution: Hamiltonian Circuit in G
  - instance: graph G, number k,
    solution: conflict-free k-colouring of G
- logic:
  - instance: propositional logic formula in CNF
    solution: a satisfying assignment
  - instance: formula in 1st order predicate logic
    solution: a model of the formula

Leipzig autotool — General Design

for each type of exercise:

- types: Config, Instance, Solution
  (each with pretty-printer, parser, API doc)
- functions:
  - grade: Instance $\times$ Solution $\to$ Bool
  - $\to$ Bool $\times$ Text
  - describe: Instance $\to$ Text
  - initial: Instance $\to$ Solution
  - generate: Config $\times$ Seed $\to$ Instance

Leipzig autotool — Components

- collection of exercise types
  as (stateless) semantics server (XML-RPC)
- plugin for Olat LMS (learning management system)
- stand-alone autotool LMS with
  - data base (problems, students, grades,. . .)
  - web front-end (for student, for teacher, . . .
  - . . . display highscores: small/early solutions)
- since $\approx$ 2000, open-source (GPL), Haskell,
  $\approx$ 1500 modules, $\approx$ 15 MB source
  `https://gitlab.imn.htwk-leipzig.de/autotool/all0`

Leipzig autotool — Applications

at HTWK Leipzig, IMN, since 2003, in lectures on

- Modellierung (discrete mathematics and logic)
- Algorithms and Data Structures
- Automata and Formal Languages
- Advanced (i.e., Functional) Programming
- Artificial Intelligence
- Principles of Programming Languages
- Theory of Computation
- Constraint Programming

Experience - Students, Teachers

- autotool is: always available, always correct, always patient
- teaching/grading assistant is: available for few hours a week only (if at all – staff costs money, which we generally don't have)
- autotool homework exercises prepare students for discussing "real homework" (that is, proofs) in classes

Experience - Implementation

- each exercise type is a domain specific language (concrete syntax, abstract syntax, semantics)
- *implementation* of the grading algorithm ($=$ semantics) is always the easiest part
- the hard part is the *design*
  - what type of exercise helps the student to understand a specific concept?
  - how can we write the instance generator?

Design Goals for Exercises

- grading:
  - should give reasonable explanation for wrong submissions (not just “it’s wrong”)
  - without giving away the correct solution
- generator:
  - each instance: non-trivial, but manageable,
  - set of instances:
    sufficiently distinct, but of similar difficulty
- concrete syntax:
  - Haskell syntax for tuples, lists, records
  - except: (model) programming languages

Design Principles for Exercises

- basic approach: verify property of an object
  example: any NP complete problem, e.g., SAT
- but this does not check whether the student used a certain algorithm to construct this object
- several exercise types implement non-deterministic algorithms ($=$ inference systems)
  student has to find an execution path (inference tree, proof), examples:
  - Resolution (derive empty clause)
  - Hilbert style deduction (derive formula)
  - (balanced) search tree operations

Example: Algorithms on Search Trees

- instance: AVL trees s, t, pattern p, e.g.,
  `[Insert 92,*,*,*,*,Insert 51,*,Delete 38]`
  solution: sequence q of operations that matches p and transforms s to t
- this exercise is not to implement operations, but to give correct (black-box) implementation so that students can explore their properties
- underlying design principle: *sudoku*,
  that is, create “holes” that students have to fill in

Design Principle: AST Sudoku

- start from any exercise type with
  grade: Instance $\times$ Solution $\to$ Bool
- build generator that produces correct pairs
- Instance $\in$ Term(Σ), Solution $\in$ Term(Γ),
  from Term to Pattern: introduce (several)
  - variables for subtrees
  - variables for function symbols
- “sudoku” variant of this exercise:
  - instance: $(p_i, p_s) \in \text{Pat}(\Sigma) \times \text{Pat}(\Gamma)$
  - solution: a correct instance of (p_i, p_s)
- unlike Sudoku, solution is not necessarily unique

Sounds Great - I Want This!

- autotool is free software (GPL):
  you can download, compile, install, use!
  source/instruction: `https://gitlab.imn.htwk-leipzig.de/autotool/all0`
- TODO (contributions welcome)
  - translation (most exercises German-only, some English-only, some have both texts)
  - more exercise types (requires: 1. design skills, 2. Haskell skills)
  - integration with other LMS (learning management systems)

Discussion (this slide added after talk)

- Q: autotool should give feedback based on models of students' learning process (and errors)

  A: Nice to have. Background see `https://www.uu.nl/staff/JTJeuring#tabPublicaties`
- Q: autotool tutorials for students? A: Concrete syntax is mostly uniform, semantics is discussed in lectures. Students have to adapt to (but that's exactly the point):
  - use textual input (not graphical)
  - read and understand error messages
- Q: tutorials for teachers? A: see `https://gitlab.imn.htwk-leipzig.de/autotool/all0#documentation-papers-talks-theses`

Discussion: Can this work?

- some properties are not decidable (equivalence of context free grammars, of programs, ...)
  - use tests instead (e.g., 1000 shortest strings and 1000 random strings)
  - do not check the property, but a formal proof of that property
    (need to define and implement syntax and semantics for proofs)
  - change the question to use a decidable approximation instead,
    e.g., program equivalence: forget states, obtain regular trace language