Automation for Exercises on Principles of Programming Languages

Johannes Waldmann, HTWK Leipzig

TUM, 27. 2. 2017

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()), ...
```

Example: Polym. Typ. — Answer

```
berechne Typ für Ausdruck:
  S.<Kermit>rowlf (S.bunsen (), S.bunsen ())
    Name rowlf hat Deklaration:
      static <T2> Fozzie<Kermit, T2> rowlf ( T2 x
    die Substitution für die Typ-Parameter ist
        listToFM [ ( T2, Kermit)]
    die instantiierte Deklaration der Funktion is
        static Fozzie<Kermit, Kermit> rowlf ( Ker
    prüfe Argument Nr. 1
        berechne Typ für Ausdruck: S.bunsen ()
            Name bunsen hat Deklaration:
              static Piggy<Fozzie<Animal, Animal>
        Ausdruck: S.bunsen ()
        hat Typ: Piggy<Fozzie<Animal, Animal>>
    Argument-Typ stimmt mit instantiierter Deklar
        No.
```

Example: Polym. Typ — Summary

- problem instance:
  - signature S
    (set of Java-like method declarations)
  - type T
- problem solution:
  - expression e of type T in S
- extra information during evaluation:
  - trace of the type checker walking the AST

Ex: Poly. Typ. — Instance Generator

- *generator* is function: Config × Seed → Instance
- s. t. instance is solvable and fulfils constraints

instance on previous slide could have been generated from:

```
Config
    { types_with_arities =
       [ ( Kermit , 0 ) , (Animal,0), ( Piggy , 1
    , type_variables = [ T1 , T2  ]
    , function_names = [ statler , waldorf , buns
    , type_expression_size_range = ( 1 , 4 ) , ar
    , solution_size_range = ( 6 , 12 ) , generato
    , generator_retries = 10
    }
```

Ex: Poly. Typ. — Discussion

alternative:

- use Java compiler to check solution
- use Java IDE to derive solution

discussion: properties of home-grown type checking

- it is extra work to define and implement abstract syntax, type checker, concrete syntax
- but no too much (“it’s just a few lines of Haskell”)
- can serve as example in Compiler Construction
- abstract syntax can be more restrictive
- type checker can be more verbose
- would need this anyway for the generator

Frames and the Static Chain

- subprogram call $\Rightarrow$ activation record (frame)
- each frame has two predecessors
  - dynamic p. (who called this subprogram?)
  - static p. (who declared this subprogram?) (in general, the frame that was active when the closure was constructed)
- exercise problem:
  - instance: relations D, S on $F = \{1, \dots, n\}$
  - sol.: program P s.t. execution of P creates frames $F_1 \dots, F_n$ with given predecessors
- ex.: $S = \{5 \to 3, 4 \to 2, 3 \to 1, 2 \to 1\}$
  $D = \{5 \to 4, 4 \to 3, 3 \to 2, 2 \to 1\}$

Frames — Example, Discussion

$S = \{5 \to 3, 4 \to 2, 3 \to 1, 2 \to 1\}$,
$D = \{5 \to 4, 4 \to 3, 3 \to 2, 2 \to 1\}$

```
function f1 () {
  f2 = function () {
     f4 = function () { } ;
     f3 ();                  } ;
  f3 = function () {
     f5 = function () { } ;
     f4() /* but it is invisible here */ }
  f2 () ;                                      }
```

- what pairs (S, D) are realizable?
  ("common domain and root, $S \cup D$ loop-free"?)
- example for the "most recent" error
  (McGowan, SIGPLAN 1972 7(1) 191–202)?

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 $\rightarrow$ Bool
  - $\rightarrow$ Bool $\times$ Text
  - describe: Instance $\rightarrow$ Text
  - initial: Instance $\rightarrow$ Solution
  - generate: Config $\times$ Seed $\rightarrow$ 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

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 inst.: sufficiently distinct, 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
- to prescribe an algorithm:
  object = list of steps of an algorithm, examples:
  - DPLL (decide, progagate, conflict, backtrack), with CDCL (learn, backjump)
  - Resolution (derive empty clause)
  - Hilbert style deduction (derive formula)

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 (sevaral)
  - 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

AST Sudoku — examples

- exercise on data structures (AVL, red/black):
  - NOT: insert $(t_1, 42)$ is ...?
  - instantiate `[Ins *, *, Del 3, *, *]`
    s.t. it transforms t_1 (given) into t_2 (given).
- exercise on polynomials:
  - instantiate $[(q_1, r_1), \ldots, (q_k, 0)]$ where
    $(q_2, r_2) = (15 \cdot x^? + ? \cdot x^?, 14 \cdot x^? + ? \cdot x^1), \ldots$
    to a complete trace of Euclid's algorithm.
  - (NB: $X = Y \cdot Q + R$ with $|R| < |Y|$ is fine)

Exercise type: Haskell Programs

- instance: Haskell module *M* with some `undefined`, and `test :: Bool`
- solution: Haskell module *M′* that matches *M* (may replace `undefined` by any expression) such that `test == True`
- example: “write function *f* as a fold”
- use property-based testing with `smallcheck`
- students can (and should!) work on exercise *as-is* in `ghci` on their machine
- security w.r.t.: cheating? attacks (DoS, leaks)?

Notes from Discussion

- 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
-