Automatisierte Bewertung und Erzeugung von Übungsaufgaben zu Prinzipien von Programmiersprachen

Johannes Waldmann

Fakultät IMN, HTWK Leipzig,
johannes.waldmann@htwk-leipzig.de

Zusammenfassung Das E-Learning/E-Assessment-System autotool ist eine Online-Plattform zur semantischen Bewertung von studentischen Lösungen von Übungsaufgaben und gestattet auch das zufällige Generieren von Aufgabeninstanzen.
Ich verwende diese Plattform für einen Teil des Übungsbetriebes zur Vorlesung *Prinzipien von Programmiersprachen* und berichte über Design, Implementierung und Erfahrung mit Aufgabentypen zur polymorphen Typisierung, zur denotationalen Semantik rekursiver Definitionen, zur (approximierten) Spur-Semantik imperativer Programme, sowie zu dynamischen und statischen Ketten bei lokalen Unterprogrammen.

1 Einleitung

In der Vorlesung *Prinzipien von Programmiersprachen* geht es um Syntax, Semantik und Pragmatik programmiersprachlicher Konzepte und ihrer Realisierungen. Welche Übungsaufgaben sollte man dazu stellen?

Naheliegend sind Bastelarbeiten in verschiedensten realen Programmiersprachen. Oft sind dabei aber die jeweils betrachteten Konzepte nicht in Reinform vorhanden und die Studenten werden durch nebensächliche Sprachspezifika abgelenkt. Die reale Sprachvielfalt täuscht auch, denn unter der Oberfläche findet man doch oft nur das imperative objektorientierte Paradigma, und die Betrachtung der $(n + 1)$-ten Instanz davon bringt dann nicht mehr viel neues.

Nützlich sind vielmehr Aufgaben zur isolierten Behandlung jeweils einer Idee. Ich realisiere solche Aufgaben als Semantikmodule für mein E-Learning/E-Assessment-System *autotool* [RW02]. Für jeden Aufgabetyp definiert dabei man eine problemspezifische Sprache. Dabei ist die Syntax entweder uniform (und entspricht der Syntax von Daten-Termen in Haskell) oder an eine bekannte Sprache angelehnt (z.B. Methodendeklarationen in Java). Die Semantik wird durch einen aufgabenspezifischen Interpreter realisiert. Dieser verarbeitet die studentische Einsendung und liefert (sofort) diese Informationen:

- Einsendung ist richtig oder falsch bzgl. Aufgabenstellung
- wenn richtig, eine Meßgröße (z.B. Eingabegröße) für eine Highscore-Wertung
- wenn falsch, eine Spur der ausgeführten Rechnung (Probe) mit Teilschritten.

Der Einsender soll aus der Fehlermeldung (Spur der Probe) und dem Wissen aus der Vorlesung erschließen, wie die Einsendung zu reparieren ist, und kann das innerhalb der vorgesehenen Bearbeitungsfrist beliebig oft versuchen. *autotool* unterscheidet sich damit grundlegend von E-„Learning“-Systemen, bei denen doch nur überprüft wird, ob an vorgegebenen Stellen Kreuzchen gemacht wurden.

Die Erfahrung zeigt, daß die Studenten vor allem die sofortigen, ausführlichen und nachvollziehbaren Rückmeldungen des Systems schätzen und die sich daraus ergebende Möglichkeit, zu beliebiger Tages- (und Nacht-)Zeit Aufgaben zu bearbeiten.

Zu vielen Aufgabentypen gibt es Instanzen-Generatoren. Diese erzeugen nach einstellbaren Parametern verschiedene, aber ähnlich schwere Aufgabeninstanzen eines Typs. Jeder Student erhält eine eigene Aufgabeninstanz und kann deren Lösung deswegen nicht bei anderen abschreiben.

Für Aufgaben mit gemeinsamer Instanz kann eine öffentliche (und pseudonymisierte) Highscore-Liste geführt werden, in der allen richtigen Einsendungen nach der Meßgröße sortiert werden und jene mit übereinstimmenden Werten nach Einsendezeitpunkt. Wer einen vorderen Platz in dieser Liste ergattern will, wird auch niemanden abschreiben lassen.

Im vorliegenden Bericht beschreibe ich einige der von mir entwickelten, implementierten und in Vorlesungen angewendeten Aufgabentypen zu Prinzipien von Programmiersprachen. Dabei gebe ich jeweils an, welchem Zweck die Aufgabe dienen soll, dann eine exakte Spezifikation von Aufgabenstellung und Korrektheit der Lösung, sodann ein Beispiel für eine Aufgabeninstanz mit Lösung. Es folgen jeweils Beispiele für Systemantworten bei inkorrekten Einsendungen, denn das ist der wohl der häufigste Anwendungsfall und somit ein wesentlicher Bestandteil des Lernprozesses. Beispiel-Aufgaben können ausprobiert werden unter `https://autotool.imn.htwk-leipzig.de/cgi-bin/Trial.cgi?lecture=213`.

Der hier vorgelegte Artikel erscheint ähnlich in den informellen Pre-Proceedings des *Workshops E-Learning* Leipzig 2015.

Ich bedanke mich bei Hans-Gert Gräbe für nützliche Diskussionen zu Vorstufen dieses Aufsatzes sowie bei vielen Teilnehmern meiner Vorlesungen in den letzten Jahren für den (unfreiwilligen) Test der Aufgaben.

2 Aufgabe: Polymorphe Typisierung

Die parametrische (generische) Polymorphie ist ein wichtiges Hilfsmittel zur Software-Wiederverwendung. Polymorphe Typen werden oft für Collections verwendet, der Typparameter ist dann der Elementtyp.

2.1 Spezifikation

- Instanz: eine polymorphe Signatur und ein Typ T
- Einsendung: ein Term t
- Bedingung: t ist korrekt typisiert und hat Typ T.
- Highscore-Parameter: die Größe von t

Die Syntax ist sehr stark an Java angelehnt. Bei jedem Aufruf einer Methode mit polymorphem Typ müssen jedoch alle Typargumente explizit angegeben werden — auch wenn das in Java (meist) inferiert werden könnte.

Bei gegebener monomorpher Signatur bilden die korrekt typisierten Terme eine reguläre Baumsprache. In polymorphen Signaturen ist die Lage viel komplizierter, man kann etwa die Lösung eines Postschen Korrespondenzproblems in der Typ-Ableitung kodieren. Das ergibt anspruchsvolle Highscore-Aufgaben. . . .

2.2 Beispiel mit Lösung

```
Gesucht ist ein Ausdruck vom Typ Fozzie<Kermit, Kermit>
in der Signatur
    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>> y );
        static <T2> Fozzie<Kermit, T2> rowlf ( T2 x , Animal y );
    }
```

Der Zieltyp ist der Resultattyp von `S.<Kermit>rowlf`. Um das aufzurufen, braucht man einen `Kermit` (erhält man von `waldorf`, dessen Argumenttyp ist egal, also kann man `bunsen()` nehmen) und ein `Animal`:

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

2.3 Typische Fehlermeldungen

```
berechne Typ für Ausdruck: S.<Fozzie<Animal, Kermit>>statler (S.bunsen (),
                           S.bunsen ())
    Name statler hat Deklaration:
      static <T2> Piggy<Piggy<Animal>> statler ( Piggy<T2> x, Piggy<T2> y )
    die Substitution für die Typ-Parameter ist
        listToFM
            [ ( T2 , Fozzie<Animal, Kermit> ) ]
    die instantiierte Deklaration der Funktion ist
        static Piggy<Piggy<Animal>> statler ( Piggy<Fozzie<Animal, Kermit>> x
                                            , Piggy<Fozzie<Animal, Kermit>> y
                                            )
    prüfe Argument Nr. 1
        berechne Typ für Ausdruck: S.bunsen ()
```

```
        Name bunsen hat Deklaration:
            static Piggy<Fozzie<Animal, Animal>> bunsen ( )
    Ausdruck: S.bunsen ()
    hat Typ: Piggy<Fozzie<Animal, Animal>>
Argument-Typ stimmt mit instantiierter Deklaration überein?
    Nein.
```

2.4 Instanzen-Generator

Hier die Konfiguration des Generators, mit der obige Instanz erzeugt wurde:

```
Conf
    { types_with_arities = [ (Kermit,0), (Animal,0), (Piggy,1) , (Fozzie,2) ]
    , type_variables = [ T1 , T2  ]
    , function_names = [ statler , waldorf , bunsen, chef, rowlf ]
    , type_expression_size_range = ( 1 , 4 ) , arity_range = ( 0 , 2 )
    , solution_size_range = ( 6 , 12 ) , generator_iterations = 500
    , generator_retries = 10
    }
```

Bei der Generierung einer Instanz wird zunächst eine Signatur gewürfelt und dann korrekt typisierte Terme in dieser Signatur der Größe nach aufgezählt. Dieser Vorgang wird einigemale wiederholt und schließlich die Instanz gewählt, die den gewünschten Parametern am nächsten kommt.

3 Aufgabe: Semantik rekursiver Programme

Die operationale Semantik beschreibt die Rechenschritte, mit denen ein Programm ausgewertet wird. Die denotationale Semantik ordnet den Bezeichnern eines Programmtextes mathematische Objekte zu. Den Unterschied kann man für (gegenseitig) rekursive Definitionen von Funktionen so erklären: operational ist ein Programm ein Ersetzungssystem, denotational beschreibt es Funktionen. Wenn man Glück hat, sind diese eindeutig und total.

In Sonderfällen besitzen rekursive Gleichungssysteme eine explizite Darstellung der Lösung durch einfache arithmetische Funktionen und Verzweigungen. Zum Beispiel zeigt man für $g(n) = \text{if } n > 0 \text{ then } 2 + g(n-1) \text{ else } 1$ leicht $g(n) = 2n+1$. In ausgesuchten anderen Fällen [Knu91] ist das ebenfalls möglich, aber überhaupt nicht offensichtlich.

3.1 Spezifikation

- Instanz: ein Gleichungssystem E, möglicherweise rekursiv, für unbekannte Funktionen $f_1, \ldots, f_n$,
- Einsendung: explizite arithmetische Ausdrücke $A_1, \ldots, A_n$
- Bedingung: die Substitution $\sigma : f_1 \mapsto A_1, \ldots$ ist eine Lösung von E
- Highscore-Parameter: $\sum |A_i|$

Die Bedingung ist nicht entscheidbar, deswegen wird wiederholt getestet. Beim Test wird *nicht* die Einsendung mit einer bekannten Lösung verglichen, *sondern* die eingesandte Substitution σ wird auf alle Gleichungen von E angewendet und die dabei entstehenden Gleichungen zwischen arithmetischen Ausdrücken werden getestet — und das geht schnell. Selbst wenn E rekursiv ist, findet beim Testen einer Einsendung keinerlei Rekursion statt.

Es ist Sache des Aufgabenstellers, darauf zu achten, daß das Gleichungssystem eine geschlossen darstellbare Lösung besitzt. Eindeutige Lösbarkeit ist nicht erforderlich.

3.2 Beispiel mit Lösung

Die Takeuchi-Funktion kann so beschrieben werden:

```
Deklarieren Sie Funktionen mit den folgenden Eigenschaften.
wobei die Variablen über alle natürlichen Zahlen laufen:
    {forall x y z . t (x  , y  , z ) ==
    if x  <= y  then y
    else t (t (x  - 1 , y  , z ) ,
           t (y  - 1 , z  , x ) ,
           t (z  - 1 , x  , y ))
    ;}
```

Eine explizite Lösung lautet

```
{ t(x,y,z) =
  if x <= y then y else if y <= z then  z else x ;}
```

3.3 Typische Fehlermeldungen

```
gelesen: {t (x , y , z) = if x  <= y then y else z  ;}
nicht erfüllte Bedingungen:
    Constraint: forall x y z .
                t (x  , y  , z ) == if x  <= y
                then y
                else t (t (x  - 1 , y  , z ) ,
                       t (y  - 1 , z  , x ) ,
                       t (z  - 1 , x  , y ))
                ;
    Belegung: x = 3 ; y = 2 ; z = 1 ;
        dabei berechnete Funktionswerte:
        t (3 , 2 , 1) = 1
        t (2 , 2 , 1) = 2
        t (1 , 1 , 3) = 1
        t (0 , 3 , 2) = 3
        t (2 , 1 , 3) = 3
```

4 Aufgabe: Approximierte Spur-Semantik

Gegenstand ist die Semantik von imperativen Programmen als Menge von möglichen Ausführungen (Spuren) und deren Anwendung beim Feststellen der Äquivalenz verschiedener Arten der Programmablaufsteuerung (wie Schleifen und Sprünge).

4.1 Spezifikation

- Instanz: ein imperatives Programm P, eine Beschreibung von erlaubten Steuerbefehlen, z.B. „nur While (kein Goto)“, „nur Goto (kein While)“.
- Einsendung: ein imperatives Programm Q,
- Bedingung: Q hat erlaubte Struktur und ist spur-äquivalent zu P
- Highscore-Parameter: Größe von Q

Programme bestehen dabei aus abstrakten elementaren Befehlen. Diese sind benannt, aber ihre Semantik ist nicht spezifiziert. Die Programmablaufsteuerung benutzt boolesche Kombinationen abstrakter Zustandsprädikate. Diese sind ebenfalls nur benannt, aber nicht spezifiziert.

Eine Spur ist eine Folge von Paaren von Zustand und Befehl. Ein Zustand ist eine Zuordnung von Prädikat-Namen zu Wahrheitswerten. Die Spuren des Programmes `if P then A; B;` sind `[(P,A), (P,B)]`, `[(P,A), (not P, B)]` und `[(not P,B)]`. Dadurch wird modelliert, daß jede Bedingsauswertung nebenwirkungsfrei ist und jede Befehlsausführung jede Zustandsinformation zerstört. Die Menge der Spuren eines Programmes ist in diesem Modell effektiv regulär und die Spuräquivalenz damit entscheidbar. Sind die so definierten Spursprachen von P und Q gleich, so sind alle Instantiierungen von P und Q als konkrete imperative Programme äquivalent.

Unter diesem strengen Äquivalenzbegriff kann nicht jedes Goto-Programm in ein äquivalentes While-Programm transformiert werden, da man keine (booleschen) Variablen zur Verfügung hat, um Wissen über frühere Zustände abzulegen.

4.2 Beispiel mit Lösung

```
Gesucht ist ein Programm,
das äquivalent ist zu
    {foo : while (a)
               {while (b)
                   {p;
                   if (c) continue foo;
                   q;}}}
und diese Bedingungen erfüllt
    And    [ Flat , No_Loops ]
```

Lösung:

```
{foo : if (!a) goto foo_end;
 bar: if (!b) goto bar_end;
 p;
 if (c) goto foo;
 q;
 goto bar;
 bar_end: skip;
 goto foo;
 foo_end: skip;}
```

4.3 Typische Fehlermeldungen

```
gelesen: {foo : if (! a) goto foo_end;
         bar : if (! b) goto bar_end;
         p;
         if (c) goto bar;
         q;
         goto bar;
         bar_end : skip;
         goto foo;
         foo_end : skip;}

Ist Spursprache des Programms aus Aufgabenstellung
Teilmenge von Spursprache des Programms aus Ihrer Einsendung ?

Nein. Wenigstens diese Wörter sind in
   Spursprache des Programms aus Aufgabenstellung
, aber nicht in
   Spursprache des Programms aus Ihrer Einsendung :
    [ [ ( state [(a,True),(b,True),(c,True)] , Execute    p )
      , ( state [(a,False),(b,True),(c,True)] , Halt ) ] ]
```

5 Aufgabe: Statische und dynamische Ketten

Zur Verwaltung von Unterprogramm-Aufrufen benutzt man Frames, die durch Zeiger verbunden sind. Der dynamische Vorgänger eines Frames ist der Frame des aufrufenden Unterprogramms und wird zur Programmablaufsteuerung benötigt. Der statische Vorgänger ist der Frame des textuell umgebenen Unterprogramms (genauer: der Frame, in dem die Closure konstruiert wurde) und dient dem Zugriff auf die Werte lokal gebundener Namen.

Lokalen Unterprogramme (Lambda-Ausdrücke) in Sprachen wie C#, Javascript und Java unterstreichen die Bedeutung dieses Konzeptes auch in der sogenannten Mainstream-Programmierung.

Die Übungsaufgabe besteht darin, einen Programmtext so zu vervollständigen, daß bei Ausführung eine vorgegebene Struktur aus dynamischen und statischen Zeigern entsteht.

Die Programme werden in (stark eingeschränktem) Java-Script notiert. Es gibt lokale Unterprogramme, formale Parameter und lokale Variablen, aber keine Bedingungsauswertung.

5.1 Spezifikation

- Instanz: ein gerichteter Kanten-2-gefärbter Graph $G = (\{1, \ldots, n\}, E_1, E_2)$, ein Lücken-Programmtext S
- Einsendung: ein vollständiges Programm P
- Bedingung: P paßt zu S und bei Ausführung von P entstehen der Reihe nach die Frames $1, \ldots, n$ mit statischer Vorgänger-Relation E_1 und dynamischer Vorgänger-Relation E_2.
- Highscore-Parameter: Größe von P

Tatsächlich sind nicht viele Graphen G so realisierbar. Jede Kante zeigt in die Vergangenheit und ist unveränderlich. Sowohl der statische als auch der dynamische Teilgraph sind also Bäume. Deren Gestalten können sich jedoch erheblich unterscheiden.

5.2 Beispiel (ohne Lösung)

```
Ersetzen Sie im Programm
    { missing ; missing ;  missing ; }
jedes 'missing' durch eine Deklaration oder einen Ausdruck,
so daß nach höchstens 25 Auswertungsschritten
die Anweisung 'halt' erreicht wird
und die Frames dann folgende Verweise enthalten:
    Frame 1 : dynamischer Vorgänger 0 , statischer Vorgänger 0 ;
    Frame 2 : dynamischer Vorgänger 1 , statischer Vorgänger 1 ;
    Frame 3 : dynamischer Vorgänger 1 , statischer Vorgänger 1 ;
    Frame 4 : dynamischer Vorgänger 3 , statischer Vorgänger 2 ;
    Frame 5 : dynamischer Vorgänger 1 , statischer Vorgänger 4 ;
```

Diese Aufgabe ist lösbar — probieren Sie es!

5.3 Typische Fehlermeldungen

Häufige semantische Fehler in Einsendungen sind: es wird ein nicht deklariertes (nicht sichtbares) Unterprogramm aufgerufen; der Frame-Graph stimmt nicht mit G überein. — Beispiel (Ausschnitte):

```
gelesen: { f = function ( ) { halt ; };
         g = function ( y ) {
            h = function ( i ) { i ( ); }; h ( f );
          };
          g ( 42 ); }
```

```
Schritt 1 (in Frame 1)
 beginne Auswertung von function ( ) { halt ; }
    Ergebnis von Schritt 1 ist
      ValClosure { link = 1
                 , body = function ( ) { halt ; } }
Schritt 2 (in Frame 1)
  beginne Auswertung von
    function ( y ) { h = function ( i ) { i ( ); }; h ( f ); }
    Ergebnis von Schritt 2 ist
      ValClosure { link = 1
        , body = function ( y )
          { h = function ( i ) { i ( ); }; h ( f ); }
        }
...
Dieser Speicherzustand wird erreicht:    Store
  { step = 11
  , max_steps = 25
  , store = listToFM
     [ ( 1 , Frame { number = 1, dynamic_link = 0, static_link = 0 , ... } )
     , ( 2 , Frame { number = 2, dynamic_link = 1, static_link = 1, ... } )
     ...
     ]
  }
...
der dynamische Vorgänger soll 1 sein:
  Frame { number = 3 , dynamic_link = 2 , static_link = 2 }
```

6 Bemerkungen zur Implementierung

Abschließend betone ich einige technische Aspekte bei der Realisierung der vorgestellten Aufgaben sowie deren Auswirkungen auf die Didaktik.

Die Implementierungssprache des autotool ist Haskell [Mar10]. Die hohe Ausdrucksstärke und Typsicherheit der Sprache erleichtert (eigentlich: ermöglicht) das effiziente Schreiben von Parsern und Interpretern für domainspezifische (d.h. hier: aufgabenspezifische) Sprachen — und hat weitere Vorteile.

Studentische Einsendungen erfolgen ausschließlich in textueller Form, es gibt absichtlich keinerlei Unterstützung für grafische Editoren, jedoch folgende textuelle Eingabehilfen: Das Texteingabefeld ist typisiert. Aus dem Typ der Eingabe wird der zu benutzende Parser bestimmt (realisiert mit `parsec` [LM01], bei Syntaxfehlern werden automatisch die möglichen nächsten Token angezeigt) sowie (durch compile-time reflection mit `Data.Typeable`) ein URL erzeugt und angezeigt, der auf die API-Dokumentation des Eingabetyps verweist. Diese wurde mit `haddock` [MW10] erzeugt und enthält Verweise auf die tatsächlichen Quelltexte.

Studenten werden dadurch angehalten, Typdeklarationen und Quelltext zu lesen und sollen dabei erkennen, daß beides sehr weitgehend den Definitionen an der Tafel entspricht. Sie werden damit auch auf die (optionale) Compilerbau-Vorlesung [Wal13] vorbereitet, wo sie domainspezifische Interpreter selbst schreiben.

Literatur

Knu91. Donald E. Knuth. Textbook Examples of Recursion. `http://arxiv.org/abs/cs/9301113`, 1991.

LM01. Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001.

Mar10. Simon Marlow. Haskell 2010 Language Report. `https://www.haskell.org/onlinereport/haskell2010/`, 2010.

MW10. Simon Marlow and David Waern. Haddock User Guide. `https://www.haskell.org/haddock/`, 2010.

RW02. Mirko Rahn and Johannes Waldmann. The Leipzig autotool System for Grading Student Homework. In Michael Hanus, Shriram Krishnamurthi, and Simon Thompson, editors, *Functional and Declarative Programming in Education (FDPE 2002)*, 2002. `http://www.informatik.uni-kiel.de/~mh/publications/reports/fdpe02/`.

Wal13. Johannes Waldmann. M****** in der Compilerbauvorlesung. In *30. Workshop der GI-Fachgruppe Programmiersprachen und Rechenkonzepte, Bad Honnef*, 2013. `http://www.imn.htwk-leipzig.de/~waldmann/talk/13/fg214/`.