Interaktive Übungen zu Grundlagen von Programmiersprachen

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

Zusammenfassung

Ich berichte über Selbsttestaufgaben für Vorlesungen *Prinzipien von Programmiersprachen, Compilerbau* (Master Informatik FH) und *Berechenbarkeit* (Bachelor Informatik Uni) im E-Learning-System *autotool*. Dabei betone ich die didaktischen Möglichkeiten, die sich durch die semantische Bewertung der studentischen Einsendungen ergeben.

1 Einleitung

Zum Studium der Informatik gehört das Studium von Berechnungsmodellen und deren Eigenschaften. Beispiele für Berechnungsmodelle sind die strukturierte imperative Programmierung (While-Programme) oder die funktionale Programmierung (einschließlich Unterprogrammen in Java). Eigenschaften von Programmen sind die Typisierung als Teil der statischen Semantik sowie die Zustandsübergangsrelation als dynamische Semantik.

Diese Fragen sind zu unterscheiden von Fragen der *Modellierung* (die Verbindung zwischen Anwendungsdaten und mathematische Objekten wie Mengen, Relationen, Graphen, Funktionen) und der *Algorithmik* (die effizienten Repräsentation und Manipulation dieser mathematischen Objekte).

Die Verbindung besteht darin, daß Algorithmen durch konkrete Programme ausgedrückt werden sollen, dazu müssen eben die dafür wichtigen Eigenschaften von Programmiersprachen, d.h. Berechnungsmodellen, bekannt sein. Diese Eigenschaften werden üblicherweise innerhalb der Grundvorlesung *Berechenbarkeit* oder *Prinzipien von Programmiersprachen* eingeführt und in Spezialvorlesungen wie *Compilerbau* vertieft.

Für diese Vorlesungen habe ich Übungsaufgaben entwickelt, für die studentische Einsendungen durch das autotool-Systems [2] *sofort* und *semantisch* bewertet werden. Das bedeutet: das System vergleicht die Einsendung *nicht* mit einer vorgegebenen Musterlösung, sondern führt für die studentische Einsendung eine *Probe* durch und zeigt deren Resultat sowie Zwischenschritte an. Diese liefern dem Einsender eine inhaltlich nachvollziehbare Begründung der Bewertung. Zusammen

mit dem Wissen aus der Vorlesung kann der Student damit einen neuen Lösungsversuch herstellen und einsenden und diesen Prozeß bis zu einer korrekten Lösung wiederholen.

Im vorliegenden Bericht beschreibe ich einige der von mir entwickelten, implementierten und in Vorlesungen angewendeten Aufgabentypen zu Grundlagen 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. Jeder Abschnitt schließt mit kurzer Erläuterung zur Erzeugung von Aufgabeninstanzen.

Der Bericht schließt mit allgemeine Thesen zur Gestaltung von E-Learning-Systemen.

Beispiel-Aufgaben können ausprobiert werden unter
`https://autotool.imn.htwk-leipzig.de/cgi-bin/Trial.cgi?lecture=223` .

Ich bedanke mich bei Hans-Gert Gräbe für nützliche Diskussionen zu Vorstufen dieser Einreichung.

2 Aufgabe: Typisierung von Termen

Ein Typ ist ein Name für eine Menge von Werten. Bei der statischen Typisierung wird jedem Teilausdruck eines Programmes ein Typ zugeordnet. Dadurch verhindert man, daß bei der Ausführung des Programms ein Unterprogramm außerhalb seines intendierten Definitionsbereiches angewendet wird.

Die statische Typsierung ist unabhängig vom Programmierparadigma und wird in dieser Aufgabe in Java-naher Syntax formuliert. Funktionssymbole sind Methodennamen. Andere Eigenschaften von Java sollen hier ignoriert werden: die Methoden sind `static` (es gibt keine Vererbung) und besitzen keine Implementierungen.

2.1 Spezifikation

Zu gegebener Signatur ist ein korrekt typisierter Term zu konstruieren.

- Instanz: eine mehrsortige Signatur Σ und eine Sorte S
- Einsendung: ein Term t
- Bedingung: t hat die Sorte S.
- Highscore-Parameter: die Größe von t.

2.2 Beispiel mit Lösung

```
Gesucht ist ein Ausdruck vom Typ Birne
in der Signatur
    Pflaume d;
    static Birne a ( Apfel x , Apfel y , Pflaume z );
    static Apfel b ( Birne x );
    static Apfel c ( Pflaume x , Pflaume y , Pflaume z );
```

Die einzige Funktion, die eine Birne liefert, ist `a`. Um diese aufzurufen, muß man Äpfel und Pflaumen konstruieren. Eine Pflaume hat man schon (`d`), einen Apfel baut man sich:

```
a (c (d , d , d ), c (d , d , d ), d )
```

Eine weitere korrekt Lösung ist

```
a (b (a (c (d , d , d ), c (d , d , d ), d )), c (d , d , d ), d )
```

2.3 Typische Fehlermeldungen

Syntaxfehler:

```
berechne Typ für Ausdruck: a (b , c )
    Funktion hat Deklaration: static Birne a ( Apfel x
                                             , Apfel y , Pflaume z )
    Anzahl der Argumente stimmt mit Deklaration überein?
        Nein.
```

Semantik-Fehler:

```
berechne Typ für Ausdruck: a (d , d , d )
    Funktion hat Deklaration: static Birne a ( Apfel x
                                             , Apfel y , Pflaume z )
    Anzahl der Argumente stimmt mit Deklaration überein?
        Ja.
    prüfe Argument Nr. 1
        berechne Typ für Ausdruck: d
            ist Variable mit Deklaration: Pflaume d
        hat Typ: Pflaume
    Argument-Typ stimmt mit Deklaration überein?
        Nein.
```

2.4 Instanzen-Generator

Parameter sind die vorkommenden Sorten sowie Anzahl und Stelligkeit der Funktionssymbole.

```
Conf { max_arity = 3 , types = [ Apfel , Birne , Pflaume  ]
    , min_symbols = 4 , max_symbols = 4 , min_size = 6 , max_size = 10 }
```

Der Generator erzeugt tatsächlich endliche Baum-Automaten und bestimmt die kleinsten Elemente der von ihnen akzeptierten Sprache.

3 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.

3.1 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 alle Typargumente explizit angegeben werden — auch wenn das in Java (meist) inferiert werden könnte.

3.2 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() ) ) )
```

3.3 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.
```

3.4 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
    }
```

4 Aufgabe: Semantik von While-Programmen

In der Berechenbarkeit betrachtet man While-Programme als Modell für die strukturierte imperative Programmierung. Imperativ bedeutet: die Programmausführung ist eine Folge von Zustandsänderungen. Strukturiert bedeutet: die Programme haben eine hierarchische Struktur. Diese wird gern optisch durch Kästen (Struktogramm) oder Einrückungen dargestellt. Mathematisch ist ein Programm ein Term. Die Menge der erlaubten Terme ist gegeben durch

```
data Program = Inc Register | Dec Register
         | Assign Register Builtin [Register]
         | Skip | Seq Program Program
         | IfZ Register Program Program
         | While Register Program
```

4.1 Spezifikation

- Instanz: ein arithmetischer Ausdruck A mit freien Variablen $x_1, \ldots, x_n$
- Einsendung: ein While-Programm P
- Bedingung: für jede Substitition $\sigma : \{x_1, \ldots\} \to \mathbb{N}$: das Programm P erreicht aus der Initialkonfiguration zur Belegung σ eine Finalkonfiguration, die das richtige Resultat $A\sigma$ enthält.
- Highscore-Parameter: die Größe von P.

Die Bedingung ist nicht entscheidbar, deswegen wird die Einsendung für verschiedene Belegungen σ getestet.

4.2 Beispiel mit Lösung

```
Konstruieren Sie eine Maschine,
die die Funktion \ [ x1 , x2 ]  -> x1 + x2 berechnet!
```

Lösung:

```
Seq (While 1 (Seq (Dec 1) (Inc 0))) (While 2 (Seq (Dec 2) (Inc 0)))
```

4.3 Typische Fehlermeldungen

```
gelesen: While 1 (Seq (Dec 1) (Inc 0))

Bei Eingabe
[ 16 , 18 ]
```

```
Startkonfiguration listToFM [ ( 1 , 16 ) , ( 2 , 18 ) ]
erreicht die Maschine die Endkonfiguration
    State
        { schritt = 65
        , memory = listToFM [ ( 0 , 16 ) , ( 1 , 0 ) , ( 2 , 18 ) ]
        , todo = [ ]
        }

Die Endkonfiguration enthält das Resultat
16
gefordert war
34

Die Rechnung beginnt so:
    State
        { schritt = 0
        , memory = listToFM [ ( 1 , 16 ) , ( 2 , 18 ) ]
        , todo = [ Seq (Dec 1) (Inc 0)
                 , While (Seq (Dec 1) (Inc 0)) ]
        }
    State
        { schritt = 1
        , memory = listToFM [ ( 1 , 16 ) , ( 2 , 18 ) ]
        , todo = [ Dec 1 , Inc 0
                 , While (Seq (Dec 1) (Inc 0)) ]
        }
```

5 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 werden dadurch Funktionen beschrieben. Wenn man Glück hat, sind diese eindeutig und total. Der Aufgabensteller muß beachten, daß auch für diesen Fall nur wenige rekursive Gleichungssysteme eine explizite Darstellung der Lösung durch einfache arithmetische Funktionen gestatten. Die Fachdidaktik [1] kennt klassische Beispiele.

5.1 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 entstehen Gleichungen zwischen arithmetischen Ausdrücken werden getestet — das geht schnell. Selbst wenn E rekursiv ist, findet beim Testen einer Einsendung keinerlei Rekursion statt.

5.2 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 ;}
```

5.3 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
```

6 Thesen

Die semantische Bewertung von Übungsaufgaben unterstützt das Verstehen der Vorlesung. Das E-Learning-System verwendet genau die Begriffe aus der Vorlesung.

Beispiel: die Semantik eines imperativen Programms ist eine Relation auf Maschinenzuständen, diese Zustände sieht man tatsächlich in der Systemantwort.

Die automatisch aus Typdeklarationen abgeleitete Dokumentation ist eine sinnvolle Eingabehilfe.

Beispiel: bei der While-Programm-Aufgabe steht unter dem Eingabefenster "gesucht ist ein Ausdruck vom Typ `http://autotool.imn.htwk-leipzig.de/docs/autotool-collection/While-Type.html#t:Program`". Dort findet der Student die (oben zitierte) Definition des Datentyps `Program`.

Das Studium der Quelltexte der Übungsaufgaben nützlich, wenn eine geeignete (d.h. deklarative) Implementierungssprache benutzt wird.

Beispiel: vom Datentyp gelangt man zum Quelltext (`While.Step`), in dem die Semantik von While-Programmen als Zustandsübergangsrelation realisiert wird:

```
data State =
  State { schritt :: Int , memory :: Memory , todo :: [Program ] }
step :: State -> [ State ]
step s = case todo s of
  [] -> []
  p : ps -> case p of
    ..
    Inc v -> stepped $ update succ s v
    Seq p1 p2 ->  stepped $ s { todo = p1 : p2 : ps }
    IfZ r p1 p2 -> stepped $
      if 0 == get (memory s) r
      then s { todo = p1 : ps }
      else s { todo = p2 : ps }
```

Die Deklarationen im Quelltext stimmen optisch und inhaltlich weitgehend überein mit der mathematischen Wahrheit an der Tafel und im Skript.

Die semantische Bewertung ist eine Grundlage für die intelligente Erzeugung von personalisierten Aufgabeninstanzen.

Durch geeignete Programmierung von Instanzengeneratoren und -Lösern für einen Aufgabentyp kann jeder Student eine andere Aufgabeninstanz erhalten, dabei sind alle sind lösbar und ähnlich schwer. Das „Abschreiben“ von Lösungen wird damit unmöglich.

Bei nicht personalisierten Aufgaben ist eine (pseudonymisierte) Highscore-Wertung für kürzeste Lösungen ein wirksamer Anreiz für intensives individuelles Arbeiten der Studenten.

Zum Beispiel ist die Berechnung arithmetischer Funktionen (wie Multiplikation, Potenzieren) auf sehr einfachen Maschinenmodellen oft umständlich, weil sie viele elementare Operationen (wie Inkrementieren, Dekrementieren) erfordert. Aber genau deswegen kann man oft durch clevere Programmierung einige dieser elementaren Operationen einsparen. Dazu muß man die Semantik des Maschinenmodells beherrschen, und genau das ist das Ziel der Übung. Wer tatsächlich den Highscore anstrebt, wird auch nicht abschreiben lassen.

Eine ausschließlich textuelle Eingabe mit uniformer Syntax unterstreicht die Rolle der mathematischen Notation in der Informatik.

Jede studentische Einsendung ist ein Text in einer aufgabenspezifischen formalen Sprache. Für viele dieser Sprachen ist die konkrete Syntax (das, was der Student tatsächlich eintippen muß) aus der abstrakten Syntax (einem algebraischen Datentyp) auf uniforme Weise abgeleitet (Code für Parser und Prettyprinter wird aus der Typdeklaration automatisch generiert). In *autotool* gibt es absichtlich keine grafischen Eingabemöglichkeiten. Das ist meine Interpretation des Diktums von Wittgenstein *Was sich überhaupt sagen läßt, läßt sich klar sagen* [3] . . . nämlich in der Sprache der Mathematik, genauer: durch Terme, d.h. Zeichenreihen, die abstrakte Syntaxbäume repräsentieren.

Literatur

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

[2] 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/`.

[3] Ludwig Wittgenstein. Logisch-Philosophische Abhandlung. *Annalen der Naturphilosophie*, 14:185–262, 1921.