(Hrsg.): ,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2021 1

Eine einfache Grafik-Programmiersprache für die Einführung in die Informatik

Johannes Waldmann[1]

Abstract: Für eine Nebenfach-Vorlesung *Einführung in die Informatik* für Erstsemester eines medientechnischen Studienganges habe ich eine konfigurierbare deklarative Grafik-Programmiersprache entwickelt und in das `autotool`-System zur automatischen Hausaufgabenstellung und -Bewertung integriert. Ich berichte über Entwurf und Anwendung.

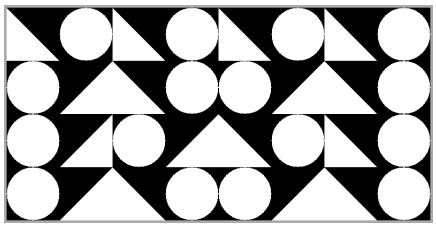
1 Einleitung

Eine einsemestrige Vorlesung *Einführung in die Informatik* als Nebenfach für Erstsemester eines medientechnischen Studienganges präsentiert ausgewählte Gebiete und Methoden der Informatik. Ein Abschnitt handelt von grundlegenden Konzepten der Programmierung und der Programmiersprachen. Die vorausgehende Themen sind digitale Informationsdarstellung (Zahlen, Zeichen, Töne, Stand- und Bewegtbilder) sowie Hardware (mit etwas Schalt-Algebra). Es folgt der Abschnitt zu Computernetzen (Kodierung, Verschlüsselung, Protokolle–offen und geschlossen, Dienste—föderiert und zentralisiert), der leider mit einer Darstellung der Methoden und gesellschaftlichen Auswirkungen der Überwachungswirtschaft abschließen muß.

Im Abschnitt zur Programmierung verwende ich eine selbstgebaute Sprache zur Beschreibung von rechteckigen Schwarz-Weiß-Grafiken.

Beispiel 1. Das Programm links beschreibt das Bild rechts. Es enthält elementare Bilder (`Triangle`, `Circle`), Operationen (Nebeneinandersetzen, Übereinandersetzen, Spiegeln), eine Unterprogrammdefinition (`f`) sowie einen wiederholten Unterprogramm-Aufruf (`iterate`).

```
let { f (x :: Pic) =  Column
       [ Row [ x, x ]
       , Row [ Transform Mirror x
               , x ]  ]
    }
in iterate f 2
  (Row [Triangle (100,100) False
       , Circle 100])
```



[1] Institut für Informatik, Fakultät Informatik und Medien, Hochschule für Technik, Wirtschaft und Kultur Leipzig, `https://www.imn.htwk-leipzig.de/~waldmann/`

Die Bildsprache verwende ich für automatisch bewertete und teilweise auch automatisch erzeugte Hausaufgaben im Auto-Grader `autotool` [RW02]. Im einfachsten Fall ist die Aufgabenstellung ein Bild und die Lösung ist ein Programmtext, der das Bild erzeugt. Die Ausdrucksmittel der Lösung können beschränkt werden (z.B., keine Unterprogramme) und Teile des Programmtextes können als verbindlicher Lückentext vorgegeben werden.

Die Manipulation von Grafiken ist nur das anschauliche Hilfsmittel. Die Studenten beschäftigen sich später im Studium ausführlich mit papiernen und elektronischen Medien sowie Soft- und Hardware zu deren Herstellung. Da will ich gar nicht vorgreifen und halte deswegen die Bilder einfach (schwarz/weiß) und auch die Sprache ist stark reduziert.

Das Ziel ist nicht die Anwendungsprogrammierung, sondern die Vermittlung grundlegender Prinzipien der Programmierung wie der Abstraktion (durch Namen) und der Programmablaufsteuerung, der Unterscheidung zwischen konkreter Syntax (Zeichenreihe) und abstrakter Syntax (Baum), der Unterscheidung zwischen statischer Semantik (Sichtbarkeitsbereiche und Typen) und dynamischer Semantik (Programm-Ausführung) und der Trennung zwischen Spezifikation und Implementierung. Diese Prinzipien sind zum Verständnis (informations)technischer System allgemein nützlich.

Im folgenden beschreibe ich zunächst die bereichsspezifischen Ausdrucksmittel der Sprache (Abschnitt 2), dann die abstrakten (Abschnitt 3). Wegen der Platzbeschränkung befinden sich im Haupttext nur Motivation, Beispiele und Diskussion (Abschnitt 4), Details sind im Anhang.

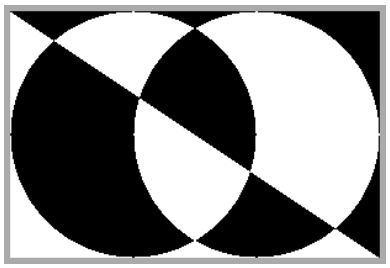
2 Aufgaben zur Erzeugung von Bildern

Jeder Programmtext ist konkrete Syntax (als Zeichenreihe) für eine baumartig strukturierte abstrakte Syntax. Die Bedeutung des Textes wird dann durch Induktion über diesen Baum („von den Blättern zur Wurzel“) definiert. Dazu betrachten wir zunächst Terme, die nur geschachtelte domainspezifische Operator-Anwendungen beschreiben, ohne weitere programmiersprachliche Ausdrucksmittel. Als Anwendungsbereich verwenden wird rechteckige Schwarz-Weiß-Bilder.

Algebraische Beschreibung von Bildern. Jedes Blatt eines Ausdrucks-Baumes bezeichnet ein elementares Bild (einfarbiges Rechteck, rechtwinkliges Dreieck, Kreis), jeder innere Knoten bezeichnet eine geometrische Transformation eines Bildes (Rotation, Spiegelung, Skalierung, Ausschnittbildung) oder eine Kombination von mehreren Bildern durch Neben- oder Übereinandersetzen oder durch punktweise Boolesche Operationen. Hier wird das Konzept der strukturierten (baumartigen) Beschreibung von aussagenlogischen Ausdrücken übernommen, die in der Vorlesung vorher behandelt wurden, und auf den neuen Gegenstandsbereich (Bilder) übertragen.

Beispiel 2. Hier sehen wir die grafischen und auch die punktweisen logischen Operationen.

```
Combine Xor
  [ Row [ Circle 200
        , Rectangle (200,100) False
        ]
  , Triangle (200,300) False
  , Row [ Rectangle (200,100) False
        , Circle 200
        ]
  ]
```



Dabei kommt ein neuer semantische Aspekt hinzu: nicht jeder syntaktisch korrekte Ausdruck hat einen Wert, denn die Verknüpfungs-Operatoren werden durch partielle Operationen interpretiert: alle Bilder sind rechteckig und man kann Bilder nur nebeneinandersetzen, wenn sie die gleiche Höhe haben. Bei vertikaler Komposition müssen die Breite übereinstimmen und bei Boolescher Verknüpfung sowohl Höhe als auch Breite. Man hätte das auch großzügiger definieren können (alles gestatten und dann ein umgrenzendes Rechteck bestimmen). Mir scheint die strenge Variante leichter verständlich und deswegen besser für Übung und Diskussion geeignet.

Aufgabenstellung und Bewertung. In `autotool` stelle ich dann Aufgaben, bei denen ein Bild P vorgegeben wird und als Lösung ein algebraischer Ausdruck E gefordert ist, dessen Semantik $S(E)$ gleich P ist. Mit Bezug auf Beispiel 2: die rechte Seite (das Bild) ist die Aufgabenstellung, die linke Seite (der Term) ist die Lösung. Die Einsendung E kann syntaktisch und semantisch eingeschränkt werden: der Aufgabensteller kann die Menge der erlaubten Operatoren vorgeben sowie die zulässige syntaktische Größe (des Terms) oder die semantische Größe (aller Teilbilder—durch die Ausschnitt-Operation könnten diese größer sein als das Gesamtbild).

Um garantiert eine lösbare Aufgabenstellung zu erhalten, wird P als $S(E_0)$ für einen (nur dem Aufgabensteller bekannten) Ausdruck E_0 erzeugt. Die Bewertung der Einsendung E ist jedoch *nicht* der Vergleich $E \stackrel{?}{=} E_0$ mit der geheimen Musterlösung, sondern der Vergleich $S(E) \stackrel{?}{=} P = S(E_0)$ der sichtbaren Semantiken. Nur dafür kann eine sinnvolle Fehlermeldung bestimmt werden (die abweichenden Bildpunkte werden farblich markiert). Andernfalls wäre die Fehlermeldung „Ihre Einsendung stimmt nicht mit der Musterlösung überein“, und daraus lernt der Einsender gar nichts.

Der rein syntaktische Vergleich mit der Musterlösung ist auch sachlich falsch, denn es gibt zahlreiche syntaktisch verschiedene, aber semantisch äquivalente Ausdrücke. Die grafischen Kompositions-Operatoren sind meist assoziativ, manche auch kommutativ, und es gelten einige Distributivgesetze. Solche Eigenschaften werden in der Vorlesung auch diskutiert.

Die Syntax einer bekannten Lösung E_0 wird für Lückentextaufgaben benutzt: aus E_0 wird ein Vorlage-Ausdruck V erzeugt durch Ersetzen einiger Teilbäume durch Lückensymbole. Eine korrekte Lösung E dieses Aufgabentyps muß dann das richtige Bild P erzeugen und syntaktisch zu V passen.

Beispiel 3. Eine Lückentextaufgabe besteht aus dem Bild aus Beispiel 2 und der Vorlage

```
Combine Xor [ Row [ Circle 200 , _ ]  , _
            , Row [ Rectangle (200,100) False , _ ] ]
```

Die pixelgenaue Prüfung gerasterter Bilder zerstört einige geometrische Äquivalenzen. Für jede nicht achsenparallele Strecke oder Kreislinie treten Rasterfehler auf, die durch Skalierung verstärkt werden. So bezeichnen die Ausdrücke `Transform (Scale (2,2)) (Circle 10)` und `Circle 20` nicht das gleiche Bild. Ich nehme das als Anlaß für den Hinweis auf andere Darstellungsformen (Vektorgrafik), ohne diese jedoch zu behandeln, denn das geschieht in anderen Lehrveranstaltungen.

Es gibt grafische Sprachen und Bibliotheken, die von vornherein Vektorgrafiken als semantischen Bereich verwenden. Die konkrete Repräsentation (Vektor oder Raster) kann man möglicherweise zunächst vor den Studenten verstecken, denn schließlich wird jedes Bild auf den Monitor gerastert. Bei der automatischen Hausaufgabenbewertung sind jedoch Grafiken zu vergleichen. Dazu scheinen Vektorgrafiken ungeeignet. Unterschiedlich repräsentierte Splines könnten doch übereinstimmende Bilder beschreiben. Einen symbolischen Vergleichsalgorithmus kann der Student im ersten Semester nicht nachvollziehen, ebensowenig wie die Details der Rasterung für einen semantischen Vergleich. Ich verwende deswegen `Scale` sehr sparsam in Aufgaben.

Automatische Erzeugung unterschiedlicher Aufgaben-Instanzen. Jeder Student erhält eine andere, gewürfelte, Aufgaben-Instanz. Der Aufgabensteller legt die Operatormenge, die Termgröße und die gewünscht Bildgröße fest. Ein Generator erzeugt pseudo-zufällige E_0 und $P = S(E_0)$ ist die Aufgabenstellung. Das Würfeln von Aufgabeninstanzen ist eine Herausforderung und regelmäßig der schwierigste Teil der Implementierung. Alle Instanzen sollen mit vernünftigem und vergleichbarem Aufwand lösbar sein. Wenn E_0 vorgegeben ist, hat man dadurch eine obere Grenze für die Größe einer Lösung. Diese kann zur Orientierung des Studenten auch angezeigt werden („diese Aufgabe hat wenigstens eine Lösung mit Größe $\leq N$“). Die Instanz könnte jedoch zu leicht sein, weil sie eine viel kleinere Lösung (ein E mit $S(E) = S(E_0)$ und $|E| \ll |E_0|$) gestattet. Das passiert etwa, wenn E_0 einen Teilausdruck enthält, der zwei gleichfarbige gleichhohe Rechtecke nebeneinandersetzt, weil das Resultat kürzer durch ein einziges Rechteck beschrieben ist. Man kann beim Würfeln von E_0 solche Trivialitäten verhindern. Eine andere von mir implementierte Methode bewertet die Bild-Kandidaten P wie folgt: es wird ein gleichmäßiges Rechteck-Gitter, z.B. mit 4×4 Zellen, über P gelegt und möglichst viele der 16 Zellen sollen nicht einfarbig sein.

3 Eine Programmiersprache selbst bauen — Wie und Warum?

In der bisher beschriebenen Bild-Sprache ist jedes „Programm“ ein Ausdruck (ein Term), der nur Literale enthält (für elementare Bilder und für Operatoren). Man bereits damit sehr gut viele syntaktische und einige semantische Konzepte von Programmiersprachen üben.

Die Sprache wird nun schrittweise um wesentliche domain-unabhängige Ausdrucksmittel erweitert (Details im Anhang)

- Abstraktion (Abschnitt 5) durch Namen, Beispiel

```
let { a = Rectangle (1,1) False
    ; b = Row [Column[a,a],Column[a,a]]
    ; c = Row [Column[b,b],Column[b,b]]
    ; d = Row [Column[c,c],Column[c,c]]
} in      Row [Column[d,d],Column[d,d]]
```

  und Unterprogramme, Beispiel

```
let { a = Rectangle (1,1) False
    ; f (x :: Pic) = Row [Column[x,x],Column[x,x]]
} in f (f (f (f a)))
```

- Programmablaufsteuerung (Abschnitt 6) durch Verzweigung und Wiederholung, in Beispiel 1 bezeichnet `iterate` die geschachtelte Anwendung einer Funktion

$$\texttt{iterate}\ f\ k\ x = f^k(x) = \underbrace{f(\ldots(f}_{k}(x))),$$

- statische Typisierung (Abschnitt 7), siehe `:: Pic` in vorigen Beispielen, weitere Typen sind `Bool` für Wahrheitswerte (zur Steuerung der Verzweigung) und `Nat` für Zahlen (zur Steuerung der Iteration). Die Typisierung ermöglicht die automatische Erzeugung von Daten für das eigenschaftsbasierte Testen (Abschnitt 8).

Wichtig ist das schrittweise Vorgehen. Die genannten Eigenschaften (bis auf die Typisierung) können für jede Aufgabe einzeln an- und abgeschaltet werden. Auch bei voller Ausdruckskraft ist die Sprache absichtlich nicht Turing-vollständig, denn Unterprogramme sind nicht rekursiv und Iteration ist nicht bedingungsgesteuert, sondern durch Zähler. Vor der Beschreibung von Einzelheiten möchte ich dieses Vorgehen motivieren und abgrenzen.

Das Typsystem bleibt erster Ordnung und monomorph. Diese Iteration ist tatsächlich eine polymorphe Funktion zweiter Ordnung, aber deren Typ läßt sich im System nicht hinschreiben, also ist sie fest eingebaut. Hier sieht man deutlich den Unterschied zur Programmierung in der Informatik als Hauptfach: dort muß es natürlich sehr früh generische Polymorphie und Funktionen höherer Ordnung geben. Dazu wird man dann auch eine Programmiersprache verwenden, die die Arbeit mit diesen Konzepten ohne Umwege ermöglicht [Wa17, SVW19].

Für die hier beschriebene Nebenfach-Informatik benutze ich absichtlich keine reale Programmiersprache. Ob funktional oder nicht, der Anfänger muß sich mit zu vielen Sprach-Eigenschaften gleichzeitig beschäftigen, von denen gerade bei den populären nicht-funktionalen Sprachen viele schlecht durchdacht sind. John C. Reynolds schreibt [Re08] „Not all programming languages are good. Most—including the most widely used—have serious design defects, so that learning such languages is less a matter of mastering a style than of learning workarounds for the language designer's mistakes.“ Das erspare ich meinen Studenten. Die hier beschriebene Lehr-Sprache hat klar begrenzte Ziele und vermeidet alles dafür unnötige. Es gibt z.B. gar keine expliziten Ein- und Ausgaben: während das für die Anwendungsprogrammierung entscheidend ist, habe die hier betrachteten Grafik-Programme als Resultat immer ein Bild, und dieses wird automatisch von der Ausführungs-Umgebung angezeigt. Das Ein/Ausgabeverhalten von Programmbausteinen kann am Unterprogrammen-Aufruf geübt werden.

Die folgenden wünschenswerten Eigenschaften von Programmieraufgaben können in jedem Fall nur durch spezielle Werkzeuge gelöst werden, nicht durch Standard-Werkzeuge (Interpreter, Compiler) realer Sprachen:

Die Bewertung von Lückentext-Aufgaben erfordert den Strukturvergleich von abstrakten Syntaxbäumen (AST). Dazu müßte die AST-Definition und der Parser aus dem Standard-Werkzeug extrahiert, oder separat nachprogrammiert werden. Das ist möglich, aber umständlich—und für eine stark eingeschränkte Sprache viel einfacher.

Die Bestimmung der statischen (Typisierung) und dynamischen (Auswertung) Semantik sollen für den Studenten nachvollziehbar sein. Dazu muß die Möglichkeit bestehen, jeden einzelne Schritt (der Typisierung als abstrakte Intepretation, der Auswertung als konkrete Interpretation) anzuzeigen. Das ist mit Standardwerkzeugen üblicherweise nicht möglich (Typisierung) oder nur sehr schwer (Debugger).

Eine Integration in ein Auto-Grading-System erfordert zusätzliche Arbeit. Die Ausführung realer Programme in Sprachen mit unsicherem Typsystem (d.h., in welchem eine Aktion nicht von ihrem Resultat unterschieden wird) ist nur unter sehr großen Vorsichts- und Isolationsmaßnahmen möglich. Ein rein funktionales Programm kann man gefahrlos und zustandslos interpretieren, deswegen braucht man dabei auch nichts zu isolieren. Der Interpreter zählt Rechenschritte mit und bricht nach vorgegebener Grenze ab.

Diese Argumente gelten wortgleich auch für die zahlreichen Online-Systeme, die beim Lernen des Programmierens (in realen Sprachen) helfen sollen. Diese scheiden zusätzlich deswegen aus, weil eine Hochschule die Benutzung von Drittanbietern ohne vetragliche Absicherung des Studentendatenschutzes nicht empfehlen kann (zum Üben) und erst recht nicht vorschreiben darf (für Pflicht-Hausaufgaben).

4 Verwandte Arbeiten und Diskussion

Das klassische Beispiel für die Lehre der imperativen Programmierung durch Grafik ist Logo [Lu72]. Der semantische Bereich sind Befehlsfolgen, die den Zeichenstift bewegen. Eine Einführung in die funktionale Programmierung durch die algebraische Beschreibung von Medien verfolgen *The Haskell School of Expression* [Hu00] und *The Haskell School of Music* [PH18]. Die Autoren verwenden angepaßte Beispiele und Bibliotheken, arbeiten jedoch mit Standardwerkzeugen (Compiler und Interpreter) und müssen deswegen den Standard-Sprachumfang weitgehend präsentieren. Das Online-Lernsystem *Code World* (das wegen des fehlenden Datenschutzes nicht in Frage kommt) bietet ebenfalls einen Einstieg in Haskell über Grafik-Programmierung. Eine „Einführung in Haskell“ (oder irgendeine andere Sprache) ist jedoch nicht mein Ziel, weder im Neben- noch im Hauptfach.

Einen für die Lehre eingeschränkten und konfigurierbaren Sprachumfang bietet *Racket* [Fe15]. Bilder werden in *Quick: An Introduction to Racket with Pictures* (`https://docs.racket-lang.org/quick/`) konstruiert. Mein hier vorgestellter Ansatz ist ganz ähnlich. Eigenschaften zugrundeliegenden Sprache (Scheme [AS85]) scheinen durch: die fehlende referentielle Transparenz, das fehlende Typsystem und die konkrete Syntax. Man kann das sicher so lehren, daß das zunächst nicht stört.

Keines der betrachteten Systeme bietet automatische Bewertung mit ausführlicher Protokollierung, das Umschalten der Gegenstandsbereiche (Bilder, aussagenlogische Funktionen, Zahlenfolgen), konfigurierbare syntaktische Einschränkungen und Lückentext-Aufgaben, Generierung von Aufgaben-Instanzen und Integration in das von mir sowieso schon zur Aufgaben- und Punkteverwaltung benutzte System `autotool`.

Die Nach-Implementierung guter fremder Ideen zum Auto-Grading von Programmier-Aufgaben zum Zweck der Benutzung im eigenen Lern-Management-System (LMS) ist ineffizient. Nützlich wäre eine wohldefinierte sprach- und systemunabhängige Schnittstelle zwischen zustandslosen Auto-Grader-Komponenten und LMS. Alle `autotool`-Aufgaben einschließlich der hier vorgestellten können über eine RPC-Schnittstelle benutzt werden [Fe10].

Ich habe die hier beschriebenen Aufgaben für die Hausaufgaben und die Prüfung im Wintersemester 2020/21 angewendet. Da wegen der Infektionsschutzmaßnahmen keine Präsenz-Lehre und Präsenz-Prüfung möglich waren, schien mir das Auto-Grading besonders nützlich. Ich habe das System auch zum freien Üben verwendet: Studenten sollten mit der Grafik-Sprache Buchstaben, Symbole oder Verkehrsschilder gestalten—und haben zusätzlich einen Weihnachtsbaum gemalt. Die Prüfung benutzte Auto-Grading-Aufgaben in einem beschränkten (klausurnahen) Zeitraum. Aufgaben-Instanzen werden gewürfelt, was die illegale Kooperation erschwert. Je Aufgabe waren mehrere Lösungsversuche gestattet, die jedoch zu Abwertung führen (jeder Syntaxfehler 1%, jeder Semantikfehler 10%). Damit wird das Hinweisgeben in einer mündlichen Prüfung nachgebildet sowie das Hinwegsehen des Prüfers über reine Schreibfehler.

Literaturverzeichnis

[AS85] Abelson, Harold; Sussman, Gerald J.: Structure and Interpretation of Computer Programs. MIT Press, 1985.

[Fe10] Felgenhauer, Bertram: , Das autOlat-Protokoll. `https://gitlab.imn.htwk-leipzig.de/autotool/all0/-/blob/master/server-interface/doc/protokoll/`, 2010.

[Fe15] Felleisen, Matthias; Findler, Robert Bruce; Flatt, Matthew; Krishnamurthi, Shriram; Barzilay, Eli; McCarthy, Jay A.; Tobin-Hochstadt, Sam: The Racket Manifesto. In (Ball, Thomas; Bodík, Rastislav; Krishnamurthi, Shriram; Lerner, Benjamin S.; Morrisett, Greg, Hrsg.): 1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA. Jgg. 32 in LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, S. 113–128, 2015.

[Hu00] Hudak, Paul: The Haskell School of Expression: Learning Functional Programming through Multimedia. Cambridge University Press, 2000.

[Lu72] Lukas, George: Uses of the LOGO programming language in undergraduate instruction. In (Donovan, John J.; Shields, Rosemary, Hrsg.): Proceedings of the ACM annual conference, ACM 1972, 1972, Volume 2. ACM, S. 1130–1136, 1972.

[Ma21] Matela, Rudy: , leancheck: Enumerative property-based testing. `https://hackage.haskell.org/package/leancheck`, 2017–2021.

[PH18] Paul Hudak, Donya Quick: The Haskell School of Music, From Signals to Symphonies. Cambridge University Press, 2018.

[Re08] Reynolds, John C.: Some Thoughts on Teaching Programming and Programming Languages. In: Proceedings of the First SIGPLAN Workshop on Undergraduate Programming Language Curricula (PLC 2008). 2008.

[RW02] Rahn, Mirko; Waldmann, Johannes: The Leipzig autotool System for Grading Student Homework. In: International Workshop on Functional and Declarative Programming in Education (FDPE 2002). 2002.

[Ry02] Rytter, Wojciech: Application of Lempel-Ziv Factorization to the Approximation of Grammar-Based Compression. In (Apostolico, Alberto; Takeda, Masayuki, Hrsg.): Combinatorial Pattern Matching, 13th Annual Symposium, CPM 2002, Fukuoka, Japan, July 3-5, 2002, Proceedings. Jgg. 2373 in Lecture Notes in Computer Science. Springer, S. 20–31, 2002.

[SVW19] Siegburg, Marcellus; Voigtländer, Janis; Westphal, Oliver: Automatische Bewertung von Haskell-Programmieraufgaben. In (Strickroth, Sven; Striewe, Michael; Rod, Oliver, Hrsg.): Proceedings of the Fourth Workshop Äutomatische Bewertung von Programmieraufgaben"(ABP 2019), Essen, Germany, October 8-9, 2019. Gesellschaft für Informatik e.V., 2019.

[Wa17] Waldmann, Johannes: How I Teach Functional Programming. In: Intl. Workshop on Functional and Logic Programming (WFLP 2017). 2017.

Anhang

5 Abstraktion durch Namen

Namen. Die Benennung von Dingen ist eine grundlegende Abstraktion. Der Name kann dann anstelle des Dinges benutzt werden. Damit ermöglicht man Nachnutzung und spart Platz. Eine konkrete Syntax für die Namensvergabe in Programmen ist

```
let { n1 = e1 ; ... ; nk = ek } in b
```

Dabei sind n_i Namen und e_i und b Ausdrücke. Der Name n_i ist in $e_{i+1}, \ldots, e_k$ und b sichtbar. Das ist nicht das `let` aus Haskell oder `letrec` aus Scheme, sondern `let*`.

Beispiel 4. Ein kleiner Programmtext kann durch Namen große Bilder beschreiben:

```
let { a = Rectangle (1,1) False
    ; b = Row [Column[a,a],Column[a,a]]
    ; c = Row [Column[b,b],Column[b,b]]
    ; d = Row [Column[c,c],Column[c,c]]
} in      Row [Column[d,d],Column[d,d]]
```

Tatsächlich stelle ich eine Aufgabe der Art „erzeugen Sie ein beliebiges Bild der Größe $N \times N$, durch einen Text der Größe $\leq G$, in dem als Zahl-Literal nur die 1 vorkommen darf“. Bei beschränkten Elementarbildern wächst die Bildgröße solcher Programme einfach exponentiell mit der Programmgröße, ohne Namen nur linear.

Programme über Folgen. In einem anderen Abschnitt der Vorlesung wird die Kompression von Daten behandelt. Dazu verwende ich die vorliegende Sprache und als Gegenstandsbereich nicht Bilder, sondern Zahlenfolgen. Ein Geradeausprogramm über Folgen ist tatsächlich eine kontextfreie Grammatik, deren jede Variable eine einelementige Sprache erzeugt. Eine solche Grammatik kann als Kompressions-Wörterbuch aufgefaßt werden [Ry02]. Die Aufgabe ist dann, für eine gegebene Zahlenfolge ein kurzes Geradeaus-Programm anzugeben. Wie bei den Aufgaben mit Bildern würfelt der Instanzen-Generator die Programme, wertet sie aus, und zeigt die resultierende Folge. Aus dem ursprüngliche Programm kann ein Lückentext erzeugt werden oder eine Vorgabe für die zulässige Gesamtgröße der Lösung. Eine solche Aufgabe ist selbst dann interessant, wenn das Alphabet der Folge nur ein Element enthält. Die richtige Kompression einer unären Folge der Länge 2^n erhält man durch fortgesetzte Verdopplung. Das gibt eine Möglichkeit der Wiederholung der Binärzahlen.

Boolesche Schaltungen Jedes Geradeausprogramm über dem semantischen Bereich der aussagenlogischen Funktionen ist eine Darstellung einer Schaltung. Jeder Name des Programms entspricht einem Knoten des (gerichteten, kreisfreien) Schaltungsgraphen. Man kann an Beispielen zeigen, daß die Schaltung kleiner sein kann als der Ausdruck mit der gleichen Semantik.

Beispiel 5. Die Majoritätsfunktion mit 5 Eingängen hat eine diskunktive Normalform mit $\binom{5}{3} + \binom{5}{4} + \binom{5}{5} = 16$ Klauseln. `autotool` kann das leicht bewerten, aber ich möchte den Studenten diese Schreibarbeit nicht zumuten. Stattdessen hatte ich die (optionale) Aufgabe gestellt, diese Funktion durch ein Geradeausprogramm mit den üblichen zweistelligen Operatoren (Konjunktion, Disjunktion, Antivalenz) zu realisieren. Ich benutzte die Highscore-Wertung im `autotool`: die Größe der richtigen Einsendungen wird auf einem anonymisierten *leader board* angezeigt.

Namen mit Parametern. Das Beispiel 4 zeigt, daß Namen nützlich sind. Die gleichartige Struktur der Zeilen motiviert eine weitergehende Abstraktion durch Definition von Unterprogrammen als Namen mit Parametern. Erst dadurch ist die Verwendung eines einzigen Programmtextes für verschiedene Daten möglich.

Beispiel 6. Das Programm aus Beispiel 4 kann damit so geschrieben werden

```
let { a = Rectangle (1,1) False
    ; f (x :: Pic) = Row [Column[x,x],Column[x,x]]
    ; b = f a ; c = f b ; d = f c
} in  f d
```

oder einfach

```
let { a = Rectangle (1,1) False
    ; f (x :: Pic) = Row [Column[x,x],Column[x,x]]
} in f (f (f (f a)))
```

Wir behalten auch bei Unterprogrammen die Einschränkung bei, daß ein Name erst in den darauffolgenden Definitionen sichtbar wird, nicht in seiner eigenen. Damit ist Rekursion ausgeschlossen. Alle Programme terminieren.

Beispiel 7. Mit Programmen dieser Art erreicht man Ausgabegrößen von $2^{2^{\Theta(n)}}$:

```
let { a = Rectangle (1,1) False ; f (x :: Pic) = Row [x,x]
    ; g (x :: Pic) = f (f x) ; h (x :: Pic) = g (g x)
    ; i (x :: Pic) = h (h x) ; j (x :: Pic) = i (i x)
} in  j (j a)
```

Eine weitere Abstraktion ist nun naheliegend: mit `twice f x = f (f x)` kann man das vorige Beispiel schreiben als `twice (twice (twice f)) a`, und `((twice twice) twice) f a` erreicht mehrfach exponentielles Wachstum. Hier ist `twice` aber eine Funktionen höherer Ordnung, die im zweiten Beispiel auch noch für unterschiedliche Typparameter instantiiert wird. Das liegt außerhalb der Nebenfach-Lehrziele.

6 Programmablaufsteuerung

Bereits algebraische Ausdrücke mit Domain-Operationen realisieren eine Art der Ablaufsteuerung, denn die Operationen werden von innen nach außen (von unten nach oben im Baum) ausgewertet und jede Operation verarbeitet das vorige Ergebnis weiter. Bei Geradeausprogrammen passiert das gleiche, die Zwischenergebnisse sind dann benannt.

Durch Unterprogramme kann ein Teilausdruck mehrfach ausgewertet werden (für unterschiedliche Belegung der formalen Parameter). Trotzdem steht die Reihenfolge der Operationen statisch fest. Die weiteren Formen der Programmablaufsteuerung sind datenabhängig: die Verzweigung von einem Wahrheitswert und die Wiederholung von einem Zähler.

Verzweigungen. Zum Üben von Verzweigungen und dem gleichzeitigen Wiederholen der Aussagenlogik stelle ich die Aufgabe, eine vorgegebene Boolesche Funktion nur durch Verzweigungen zu realisieren. Der Lückentext enthält feste Testfälle und die Konfiguration verbietet, daß in den Lücken aussagenlogische Funktionen benutzt werden.

Die Verzweigungen haben die konkrete Syntax `if` B `then` A_1 `else` A_2. Jeder Zweig muß einen Wert liefern, damit entfällt die in der imperativen Programmierung notwendige Diskussion des *dangling else*.

Wiederholungen. Die eingebaute Funktion zweiter Ordnung `iterate` mit der Semantik `iterate` $f\ k\ x = f^k(x)$ realisiert die Programmablaufsteuerung durch Wiederholung.

Beispiel 8. Das Programm aus Beispiel 6 ist äquivalent zu

```
let { a = Rectangle (1,1) False
    ; f (x :: Pic) = Row [Column[x,x],Column[x,x]]
} in  iterate f 4 a
```

Als Übung zu `iterate` lasse ich zunächst ein Schachbrett zeichnen. Eine Iteration erzeugt eine Zeile, eine weitere Iteration erzeugt das Brett.

Durch Verwenden des Gegenstandsbereiches der natürlichen Zahlen können die arithmetischen Funktionen Addition, Multiplikation, Potenzierung durch Iteration aus der Nachfolgerfunktion erzeugt werden. Mit `iterate` (im monomorphen Typsystem) beschreiben wir genau die Menge der primitiv-rekursiven Funktionen.

7 Statische Typisierung

Die Algebra zur Beschreibung von Bildern ist einsortig. Jeder Term und Teilterm beschreibt ein Bild. In dem Ausdruck `Rectangle (100,200) False` kommen Zahl- und Bool-Literale vor, an deren Stelle kann aber syntaktisch kein Term stehen.

Damit wird die Diskussion der Typisierung etwas verschoben. Ein Typ ist eine Abstraktion für eine Menge von Werten. Solange der semantische Bereich nur aus Bildern besteht, braucht man gar nichts zu abstrahieren.

Der Bedarf nach verschiedenen Sorten und ihrer statischen Unterscheidung durch ein Typsystem entsteht durch die Programmablaufsteuerung: Wahrheitswerte steuern die Verzweigung, Zahlen steuern die Wiederholung. Die Sprache bietet deswegen die Typen `Pic` für Bilder, `Seq` für Folgen, `Bool` für Wahrheitswerte, `Nat` für Zahlen.

Ich benutze die Typisierung zur Diskussion der Unterscheidung zwischen statischer und dynamischer Semantik von Programmen sowie zwischen Spezifikation und Implementierung von technischen Systemen allgemein. Dieser Aspekt wird bei der Diskussion von Netzwerk-Protokollen und Diensten wieder aufgegriffen.

Jeder Term hat einen Typ. An jeden Teilterm kann optional eine Typ-Aussage angehängt werden. Jeder Name hat einen Typ. In Konstanten-Definitionen wird der Typ inferiert. In Funktions-Definitionen müssen die Typen der formalen Parameter angegeben werden. Damit wird die explizite Notation von Funktionstypen vermieden. Es ergeben sich weitere (für den speziellen Anwendungsfall nützliche) Einschränkungen: da jedes Funktions-Argument einen der Basis-Typen hat, gibt es keine generische Polymorphie. Zudem werden Funktionen höherer Ordnung verhindert, da man keinen Funktionstyp als Argumenttyp notieren kann.

Die Verzweigung ist polymorph und die Iteration ist von zweiter Ordnung

```
if-then-else :: forall (t :: Type) . Bool -> t -> t -> t
iterate      :: forall (t :: Type) . (t -> t) -> Nat -> t -> t
```

Deren Typisierung und Implementierung sind fest verdrahtet, der Anwender kann solche Funktionen nicht schreiben. Es wird syntaktisch erzwungen, daß das erste Argument von `iterate` ein Name ist, da es keine anderen Ausdrücke gibt, die eine Funktion bezeichnen.

8 Eigenschaftsbasiertes Testen

In Lückentextaufgaben können gewünschte Programmeigenschaften leicht als Bestandteil der Vorgabe formuliert werden:

Beispiel 9. Ersetzen Sie die Lücken (Unterstriche), so daß der Ausdruck den Wert `True` hat

```
let { x = _ ; y = _ } in  (x||y) && (not x || not y)
```

Die Spezifikation ist jedoch oft eine All-Aussage, die durch Aufzählung einiger konkreter Testfälle nur unzureichend geprüft wird.

Didaktisch viel besser ist es, die All-Aussage als solche erkennbar hinzuschreiben, und praktisch ist es, daraus die Testdaten automatisch zu generieren. Wir verwenden Leancheck [Ma21] zur Implementierung des (vollständigen) Vergleichs von Booleschen Funktionen.

Beispiel 10. In diese Lückentextaufgabe ist `prop` die Spezifikation (die All-Aussage), `Majority` ist die Majoritätsfunktion, und `majority` ist zu implementieren. `check 8 prop` sagt, daß 8 Testfälle, d.h., Belegungen der formalen Parameter, überprüft werden.

```
let { f = False ; t = True
    ; majority (x :: Bool) (y :: Bool) (z :: Bool) = _
    ; prop     (x :: Bool) (y :: Bool) (z :: Bool) =
              Equals [ majority x  y  z
                     , Majority [ x , y , z ]
                     ]
} in check 8 prop
```

Eine korrekte Ersetzung der Lücke ist

```
if x then if y then t else z
     else if y then z else f
```

Das illustriert auch die Realisierung einer aussagenlogischen Funktion durch einen Entscheidungsbaum.

Einfache Aufgaben mit arithmetischen Funktionen kann man ebenso behandeln. Die Prüfung ist dann nicht vollständig, aber bei kleinen Programmtexten werden Fehler meist durch kleine Testdaten (am Anfang der Aufzählung) gefunden. Eine eigenschafts-basierte Prüfung von Bild-Operationen habe ich noch nicht implementiert, weil das in der Vorlesung nicht benötigt wurde. Es scheint leicht realisierbar, denn eine Aufzählung von Bildern als Test-Daten kann mittels einer Aufzählung der Terme erhalten werden, die für den Instanzengenerator bereits implementiert ist.