functional METAPOST

Eine Beschreibungssprache für Grafiken

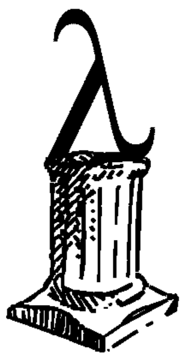
Diplomarbeit

von Joachim Korittky

Rheinische Friedrich–Wilhelms–Universität Bonn

Institut für Informatik, Abteilung III

Prof. Dr. A. B. Cremers



24. Dezember 1998

Erklärung

Ich erkläre hiermit, daß ich diese Arbeit selbstständig durchgeführt, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und Zitate kenntlich gemacht habe.

Bonn, den 24. Dezember 1998

(Joachim Korittky)

Danksagung

Ich danke meinem Betreuer Ralf Hinze für sein Engagement, seine Geduld, Tips und wertvolle Verbesserungsvorschläge.
Meinen Kommilitonen Marc Zimmermann und Bernhard Reichmann danke ich für ihre kritischen Anmerkungen zu dieser Arbeit.
Vor allem danke ich meinen lieben Eltern für ihre finanzielle und moralische Unterstützung während meines gesamten Studiums und widme ihnen diese Arbeit.

Joachim Korittky

Vorwort

In dieser Arbeit entwerfen und implementieren wir eine funktionale Bildbeschreibungssprache und einen Compiler, der aus den Bildbeschreibungen geräteunabhängige Bilder von hoher Qualität generiert.
Es existiert bereits eine vielfältige Auswahl an Bildbeschreibungssprachen die, bezogen auf spezielle Anwendungsbereiche, alle ihre Vor– aber auch Nachteile haben. Besonders reichhaltig ist die Auswahl an Makropaketen für LaTeX, die kleine Spezialsprachen für einen mehr oder weniger begrenzten Anwendungsbereich bereitstellen. Leider lassen sich die Funktionen mehrerer verschiedener solcher Makropakete meist nicht innerhalb eines Bildes kombinieren.
Viele Bildbeschreibungssprachen lassen sich nicht erweitern oder es ist für den weniger geübten Anwender zu schwierig, eigene Abstraktionen zu definieren. Das Ziel sollte sein, für oft wiederkehrende Bildelemente eine besonders einfache Beschreibung zu ermöglichen, welches Abstraktionen leisten können.
Wir werden in dieser Arbeit einen neuen Weg gehen, indem wir eine Bildbeschreibungssprache in die funktionale Programmiersprache Haskell einbetten und dies mit der Möglichkeit kombinieren, mit Hilfe von Gleichungssystemen geometrische Beziehungen zwischen Bildelementen festzulegen. Um die Universalität dieses Ansatzes zu demonstrieren, formulieren wir Erweiterungen, die auch zur Beschreibung von Bildern in Lehrbüchern der Informatik nützlich sind.
Oft unterstreichen Arbeiten, die sich mit der Implementierung von Algorithmen in funktionalen Sprachen beschäftigen, Vorteile wie Eleganz, Kürze, gute Verständlichkeit und leichte Verifizierbarkeit. Wir wollen in dieser Arbeit unser Augenmerk aber hauptsächlich auf die Möglichkeiten, die sich durch eine Einbettung der Bildbeschreibungssprache in Haskell ergeben, legen.

> *Vermittelst des äußeren Sinnes, (einer Eigenschaft unseres Gemüts), stellen wir uns Gegenstände als außer uns, und diese insgesamt im Raume vor. Darinnen ist ihre Gestalt, Größe und Verhältnis gegeneinander bestimmt oder bestimmbar.*
>
> — Immanuel Kant [Kan87]

Inhaltsverzeichnis

Kapitel 1

Einleitung

„Ein Bild sagt mehr als tausend Worte.“ heißt ein bekannter Spruch – sicher nicht zu unrecht. Trotzdem herrscht in manchen Lehrbüchern der Informatik und anderen wissenschaftlichen Publikationen ein Mangel an Illustrationen. Und wenn Bilder vorhanden sind, sind diese manchmal schief gezeichnet und in zu geringer Auflösung erzeugt. Woran liegt das? Da davon auszugehen ist, daß die Autoren bemüht sind, ihre Arbeiten mit aussagekräftigen und „akkurat“ gezeichneten Bildern zu versehen, ist die Ursache bei den zur Verfügung stehenden Werkzeugen zu suchen. Es gibt eine große Vielzahl verschiedener Möglichkeiten zur Erzeugung von Bildern. Wir können diese grob in *interaktive* und *beschreibende* Ansätze unterteilen.

In die Klasse der interaktiven Ansätze fallen Mal– oder Zeichenprogramme, die dem Anwender mit Hilfe von Eingabegeräten direkte Bildmanipulationen erlauben. Der Schwachpunkt von Zeichenprogrammen ist die fehlende Möglichkeit, eigene Abstraktionen für oft wiederkehrende Bildelemente zu definieren, von Cliparts einmal abgesehen. Es existieren zwar Abstraktionen für Spezialaufgaben, wie das Zeichnen von Organigrammen, aber universelle und vom Anwender erweiterbare Funktionalität ist nicht vorhanden.

Mit interaktiven Werkzeugen ist es schwierig, symmetrische und als ästhetisch empfundene Bilder zu erzeugen, wenn Funktionen für Ausrichtungen, Fänge oder Gruppierungen fehlen. Ein weiteres Problem, das in der Praxis auch bei sehr verbreiteten Zeichenprogrammen auftritt, sind Unterschiede in der Darstellung am Bildschirm und im gedruckten Dokument.

Beschreibende Programme erfordern dagegen ein Arbeiten in zwei Phasen: Das Bild wird zuerst in einer speziellen Bildbeschreibungssprache beschrieben und in der zweiten Phase interpretiert, wobei das beschriebene Bild in einem Format entsteht, das in ein Dokument eingebunden werden kann. Dies birgt den Nachteil, daß der Anwender entweder Erfahrung benötigt, um abzuschätzen, wie sich eine Anweisung auf das Bild auswirkt oder aus der Bildbeschreibung oft das Bild berechnen lassen muß, um das Resultat zu überprüfen. Das Fehlen einer Benutzeroberfläche vereinfacht es aber, eine Bildbeschreibungssprache als Portierung für viele Rechner– und Betriebssysteme zur Verfügung zu stehen.

Obwohl leistungsfähige beschreibende Lösungen existieren, die viele Vorteile bieten, bleibt uns noch genügend Spielraum für Verbesserungen und neue kreative Ideen. Um einen Ausgangspunkt zu erhalten, auf dem wir aufbauen können, werden wir uns im nächsten Kapitel einen Überblick über verbreitete Bildbeschreibungssprachen verschaffen.

1.1 Bisherige Lösungsansätze

In dieser Arbeit wollen wir uns auf die Möglichkeiten der Bildbeschreibung konzentrieren. Deshalb werden wir uns in den folgenden Abschnitten ein paar Lösungen für das Problem der Bildbeschreibung anschauen, die typisch für verschiedene Ansätze sind. Wir können grob drei verschiedene Klassen unterscheiden.

- Sprachen, die als Makropakete in ☞ *LaTeX*[1] integriert sind.
- Spezielle Bildbeschreibungssprachen, die eine eigene Syntax haben und einen Compiler in Form eines externen Programms benötigen.
- Bildbeschreibungssprachen, die in eine andere, meist funktionale, Sprache **eingebettet** sind. Eine Beschreibungssprache, die z.B. in ☞ *Haskell* eingebettet ist, beschreibt Bilder mit Hilfe von Haskell–Ausdrücken. Das hat gegenüber einem Ansatz, in dem Grafiken aus einer Folge von Zeichenanweisungen entstehen, den Vorteil, daß ein Bild, da es von einem Ausdruck repräsentiert wird, als Parameter einer Funktion dienen kann, um es zu manipulieren und zu einem neuen Bild zu verwandeln. Der eingebetteten Bildbeschreibungssprache steht die ganze Syntax der Sprache, in die sie eingebettet ist, zur Verfügung.

Wir stellen Lösungen aller drei Klassen vor. Jede dieser Beschreibungssprachen hat ihre Stärken und Schwächen. Aus den Stärken wollen wir lernen, was eine universelle und leicht erweiterbare Bildbeschreibungssprache ausmacht.

1.1.1 XY-pic

Als typischen Vertreter der vielen in LaTeX integrierten Sprachen wollen wir das Paket XY-pic von KRISTOFFER H. ROSE und ROSS MOORE betrachten [Ros92, GR97].

XY-pic ist eine eigene mit ☞ *TeX*–Makros implementierte Bildbeschreibungssprache mit objektorientiertem Charakter. Das Makropaket ist modular aufgebaut und ermöglicht das Einbinden sogenannter Features wie `curve`, `frame`, `cmtip`, `tips`, `line`, `rotate`, `color`, `matrix`, `arrow`, `graph` die die Sprache um zusätzliche Befehle für spezielle Anwendungen erweitern.

Die Einbettung in LaTeX bedingt eine gute Wartbarkeit der Grafiken und ein einigermaßen schnelles Übersetzen der Bilder, solange diese nicht zu kompliziert sind.

Eine Integration der Grafik in das DVI–Dokument erfolgt mittels spezieller Zeichensätze, die Linien in verschiedenen Steigungen und Symbole wie Pfeile in verschiedenen Ausführungen

```
\xymatrix{
 U \ar@@/_/[ddr]_y \ar@@/^/[drr]^x
   \ar@@{.>}[dr]|-{(x,y)}        \\
  & X \times_Z Y \ar[d]^q \ar[r]_p
                 & X \ar[d]_f    \\
  & Y \ar[r]^g   & Z  }
```

$$\begin{tikzcd}
U \arrow[drr, bend left, "x"] \arrow[ddr, bend right, "y"'] \arrow[dr, dotted, "{(x,y)}" description] & & \\
& X \times_Z Y \arrow[r, "p"'] \arrow[d, "q"] & X \arrow[d, "f"'] \\
& Y \arrow[r, "g"] & Z
\end{tikzcd}$$

Abbildung 1.1.1: Beispiel einer mit XY-pic erstellten Grafik aus [Ros96].

[1] Das Handsymbol vor einem kursiven Begriff weist auf eine nähere Erklärung im Glossar hin.

enthalten. Da hierbei zwangsläufig nicht alle, sondern nur einige diskrete Winkel berücksichtigt werden, können beim Ausdruck störende Artefakte und Treppen auftreten.

Die Bildbeschreibungen von XY-pic übertreffen in Hinsicht auf Kryptik die schlimmsten C- und Assemblerprogramme und erfordern viel Einarbeitungszeit, wie Abbildung 1.1.1 zeigt.

Die Sprache XY-pic änderte sich in der Vergangenheit leider immer wieder stark, so daß trotz Kompatibilitätsmodus ältere Bildbeschreibungen nicht mehr verarbeitet werden konnten. Das Bild aus Abbildung 1.1.1 sieht in [RM94] wie Abbildung 1.1.2 zeigt, noch ganz anders aus.

```
\xygraph{
 !M{ X \times_Z Y \="xy" \:[r]_p
       \:[d]^q
     & X \="X" \:[d]_f \\
     Y \="Y" \:[r]^g & Z}
 [ul]U   ( ? :@/^.5pc/ ^x  "X",
   ? :@{-->}   |-{(x,y)} "xy" ,
   ? :@/_.5pc/ _y  "Y" ) }
```

Abbildung 1.1.2: Programmtext einer älteren XY-pic–Version für Abbildung 1.1.1.

Die speziellen Erweiterungen von XY-pic, es gibt sogar ein Feature für das Beschreiben von mathematischen Knoten, wie in Abbildung 1.1.3 zu sehen, machen das Makropaket für den interessant, der viel damit arbeitet und für den es sich deshalb lohnt, Mühe in das Erlernen der Syntax zu investieren.

Trotzdem ist es schwierig, z.B. Bäume darzustellen. Da es für diese keine Erweiterung gibt, muß der Anwender alle Knoten in einer Matrix anordnen und dann umständlich die passenden Kanten zeichnen.

Hier zeigt sich das große Problem von in LaTeX eingebetteten Bildbeschreibungssprachen. Sie decken alle einen mehr oder weniger großen Teil von Spezialfällen ab. Man könnte, um beim obigen Problem zu bleiben, den Baum mit einem Makropaket für Bäume setzen. Die Syntax ist aber eine zu XY-pic ganz verschiedene und eine Kombination beider Pakete in einem Bild ist nicht möglich.

Es existieren also unzählige Makropakete heterogen nebeneinander die alle ein oder mehrere Probleme lösen. Dabei gibt es ebensoviele verschiedene Beschreibungssprachen, die alle gelernt werden müssen. Das Buch [GR97] gibt einen Überblick der wichtigsten Makropakete. Irgendwann fängt der Anwender an, seine Grafiken an die Möglichkeiten der Sprachen anzupassen, statt umgekehrt. Spätestens wenn ein Sonderfall nicht unterstützt wird, steht er vor einem Problem.

```
\xy 0;/r1pc/:
 ,{\vcap\vtwist
 \vunder->{a}||{b}}
\endxy
```

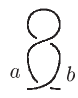


Abbildung 1.1.3: Ein Knoten, erstellt mit dem knot Feature.

1.1.2 METAPOST

METAPOST ist eine Bildbeschreibungssprache von JOHN D. HOBBY [Hob89], die sich eng an DONALD E. KNUTHS ☞ *METAFONT* [Knu86] anlehnt. Der Unterschied zwischen den beiden Programmen besteht darin, daß METAFONT Bitmaps erzeugt, während METAPOST Bildbeschreibungen in ☞ *PostScript* verwandelt. PostScript bietet den Vorteil der Geräteunabhängigkeit, denn es operiert auf einem vektorbasierten Grafikmodell mit komplexen Objekten wie Pfaden, Stiften und Zeichensätzen

[Ado85]. Die Umwandlung in ein Bitmap erfolgt erst im Ausgabegerät mit optimaler Auflösung. Um die Funktionalität von PostScript auch gut auszunutzen, ist METAPOST um Befehle zur Farbwahl, Zeichensatzwahl und Clipping erweitert.
Die Besonderheit, die METAPOST von anderen Bildbeschreibungssprachen abhebt, ist die symbolische Beschreibung von Positionen und Beziehungen mit Hilfe von Gleichungen. Auf diese Weise lassen sich lineare Gleichungssysteme formulieren, in denen unbekannte Variablen automatisch hergeleitet werden. Es existieren viele verschiedene Typen, von Zahlen, Punkten und Farben angefangen, bis hin zu Bildvariablen und Pfaden. Besonders affallend sind die vielen Möglichkeiten zur Beschreibung von Pfaden. Derartig leistungsfähige Funktionalität findet sich in keiner der anderen hier vorgestellten Bildbeschreibungssprachen.
Aber vor allem die Möglichkeit der algebraischen Bildbeschreibung qualifiziert METAPOST zu einem Werkzeug, das wir in *functional* METAPOST verwenden wollen. Deshalb werden wir uns in Kapitel 6 näher mit METAPOST beschäftigen.

1.1.3 PIC

PIC wurde 1980 von BRIAN W. KERNINGHAN an den Bell Laboratories entwickelt [Ker82]. Es implementiert eine Bildbeschreibungssprache, die Boxen, Linien, Pfeile, Kreise, Ellipsen, Kreisbögen und Splines zur Verfügung stellt, die mit Bezug auf vorherige Objekte relativ positioniert werden können.
Die Beschreibung erfolgt in einer Textdatei, mit einer Syntax, die sich, soweit möglich, an der Umgangssprache orientiert. Ein Compiler erzeugt daraus TROFF–Befehle[2]; es handelt sich also um einen ein sogenannten TROFF–Preprozessor [Oss79].
Abbildung 1.1.4 zeigt eine typisches Beispiel. Die zentralen Objekte sind Boxen, die einen Text enthalten können. Die Positionierung der Boxen ist nicht direkt angegeben. Objekte werden, wenn die Richtung nicht geändert wurde, von links nach rechts angeordnet. Die linke Seite einer Box wird an der rechten Seite des letzten Objekts plaziert. Das Beschreibungsmodell erinnert an Turtlegrafik und ist für solche einfachen linearen Anordnungen sehr praktisch.

```
.PS
arrow
box ht 0.3 wid 0.3 "x"
box same
arrow from center of last box
box same "y"
box same
arrow from center of last box
box same "z"
box same
line from last box.sw to last box.ne
.PE
```

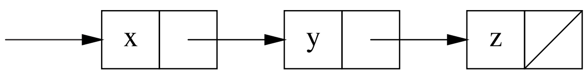


Abbildung 1.1.4: Eine Bildbeschreibung mit PIC

Boxen haben die vordefinierten Bezugspunkte `center`, `top`, `bottom`, `n`, `ne`, `e`, `se`, `s`, `sw`, `w`, `nw`, die man zu Referenzen nutzen kann.
Ferner besteht die Möglichkeit zur Benennung von Boxen und der Definition von Funktionen, die optional Parameter haben können. Eine große Einschränkung ist die fehlende Möglichkeit, die Dimensionen von Boxen automatisch an den umschlossenen Text anzupassen.
Das macht eine korrekte automatische Erstellung von Diagrammen schwierig, da der Raum, den Text

[2]Es existiert mittlerweile auch eine GNU-Implementierung des PIC-Paketes, die optional nach TEX übersetzt.

```
(define (fractal point1 point2 depth)
  (stroke (let recur ((pt1 point1) (pt2 point2) (d depth))
            (if (= d 0) (line pt1 pt2)
                (let* ((x1 (pt:x pt1))
                       (y1 (pt:y pt1))
                       (x2 (pt:x pt2))
                       (y2 (pt:y pt2))
                       (newx (+ (/ (+ x1 x2) 2)
                                (/ (- y2 y1) 2)))
                       (newy (- (/ (+ y1 y2) 2)
                                (/ (- x2 x1) 2)))
                       (newpt (pt newx newy)))
                  (compose (recur pt1 newpt (- d 1))
                           (recur newpt pt2 (- d 1))))))))
```

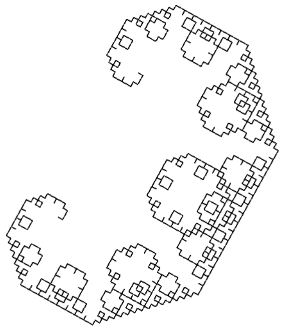


Abbildung 1.1.5: Die Bildbeschreibung eine LÉVY–Kurve mit *functional* PostScript.

einnimmt, sowohl vom Zeichensatz als auch vom Text selbst abhängt. Die Namen Modula und Eiffel haben beide sechs Buchstaben, sind aber bei Verwendung eines nicht proportionalen Zeichensatzes verschieden lang. Um etwas vorzugreifen: Dieses Problem begegnet uns auch beim Entwurf von *functional* METAPOST. Wir werden Anstrengungen unternehmen, es zu lösen.
Da das Ausgabeformat TROFF– bzw. TEX–Quellcode ist, können Farben leider nicht von PIC unterstützt werden.
PIC ist relativ unbekannt, obwohl es Teil mancher TEX–Distribution ist. Trotz der genannten Schwächen verdient das Programm mehr Bekanntheit.

1.1.4 *functional* PostScript

functional PostScript ist nach der Definition der Autoren WANDY SAE-TAN und OLIN SHIVERS ein „portables System mit dem man aus ☞*Scheme* Programmen heraus geräte– und auflösungsunabhängige Grafiken erzeugen kann“ [STS96]. Dazu definiert *functional* PostScript Scheme–Funktionen, die alle wichtigen Zeichenbefehle von PostScript bereitstellen. Die Grundobjekte von PostScript, wie Pfade und Bilder, werden auf Scheme abgebildet und das in PostScript übliche Rechnen mittels Stacks ist durch funktionale Notationen ersetzt.
Eine weiteres Konzept ist das „Ansammeln kleinerer Objekte um daraus komplexere zu definieren“. Ein Scheme Programm erzeugt während seiner Abarbeitung mit den Funktionen von *functional* PostScript eine Folge von PostScript Zeichenbefehlen. Im Falle des Beispiels aus Abbildung 1.1.5 ensteht ein Folge von Linienbefehlen. Die einfache Erzeugungsvorschrift, die zu dem Fraktal führt, läßt sich aus der Bildbeschreibung leider kaum erahnen. Andere Graphikparadigmen, wie z.B. Turtlegrafik, würden für das Bild eine wesentlich einfachere Art der Beschreibung ermöglichen.
Dem Anwender bleibt selbst überlassen, sich Abstraktionen für solche anderen Zeichenkonzepte zu entwerfen. Aber auch einfachere Funktionen, wie das Umrahmen eines Bildes, stehen nicht bereit. Damit ist *functional* PostScript nur ein einfaches Frontend zu PostScript und bietet darüber hinaus keine vordefinierten Abstraktionen für oft verwendete Grafikobjekte.

1.1.5 mlP$_\wr$cTeX

mlP$_\wr$cTeX von Emmanuel Chailloux und Ascánder Suárez [CS96] kombiniert die speziellen Fähigkeiten von LaTeX im Textsatz mit den Zeichenmöglichkeiten von PostScript. Das Paket ist in LaTeX integriert, d.h. die Bildbeschreibung kann direkt im Textdokument erscheinen. Wie in Abbildung 1.1.6 beschrieben, erzeugt eine `mlPic`–Umgebung einen LaTeX– und einen PostScript–Teil. mlP$_\wr$cTeX geht dabei in drei Schritten vor:

1. Die Übersetzung der Datei `myFile.tex` mit LaTeX erzeugt die Dateien `myFile.log` und `myFile.dvi`.
2. Das Programm mlP$_\wr$cTeX erzeugt den PostScript–Teil, indem es in der Datei `myFile.log` ☞ *Caml*–Programme ausführt, die der Befehl `\mlBody` dort hineingeschrieben hat. Die genauen Ausmaße von Textboxen müssen aus der Datei `myFile.dvi` ausgelesen werden, da diese mlP$_\wr$cTeX nicht bekannt sein können.
3. Ein weiterer Aufruf von LaTeX fügt die soeben erzeugten PostScript–Grafiken in das Dokument ein.

```
\begin{mlpic}
\mlDef{form}{$\sqrt{1+\sqrt{1+x}}$}
\mlBody{sqr}
{rotate 20.0 (oval (latexBox "form"))}
\end{mlPic}
```

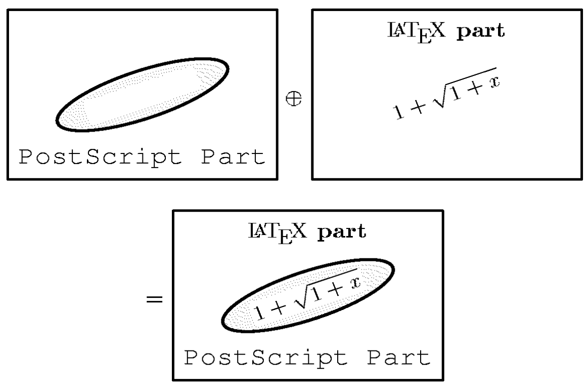


Abbildung 1.1.6: mlP$_\wr$cTeX konstruiert Bilder aus einem LaTeX– und einem PostScript–Teil.

Mit einigen Tricks kann man sogar die Ausgabe eines Caml–Programms in eine Bildbeschreibung in einem LaTeX–Dokument konvertieren und auf diese Weise visualisieren.
mlP$_\wr$cTeX stellt Befehle zum farbigen Zeichnen, zum Drehen von Bildern und auch komplexe Objekte, wie Bäume zur Verfügung. Alle Objekte besitzen standardmäßige Voreinstellungen für Eigenschaften wie Farbe oder Strichstärke. Diese Eigenschaften lassen sich mit Hilfe von Listen, die neue Eigenschaften beinhalten, verändern.

1.1.6 TkGofer

Eigentlich ist TkGofer von Koen Claessen, Ton Vullinghs und Erik Meijer [CVM97] kein Programm zur Erzeugung statischer Grafiken, sondern ein Werkzeug zur Realisierung grafischer, interaktiver Benutzeroberflächen. Trotzdem scheinen einige Ansätze auch für *functional* METAPOST sinnvoll.
TkGofer ist ein Interface zwischen dem Werkzeug Tcl/Tk und dem funktionalen Interpreter Gofer. Tcl/Tk ist eine textbasierte Scriptsprache, die die Visualisierung einer Benutzeroberfläche und die Behandlung von Interaktionsereignissen realisiert. Die elementaren Objekte, Widgets genannt, sind Fenster, Buttons, Eingabefelder, Schieberegler usw..
Viele Widgets haben gemeinsame Eigenschaften und Funktionalitäten, die sich in einer Klassenhierarchie ordnen lassen. So kann man für jedes sichtbare Widget eine Farbe festlegen. Natürlich möchte

man die gemeinsamen Eigenschaften verschiedener Widgets mit nur einer Funktion beeinflussen. Dieses Problem läßt sich mit Konstruktorklassen lösen [VSS96]. Alle Widgets, die einen statischen Text haben, wie z.B. Label oder Buttons, werden in der Klasse `HasText` vereint, die die Funktion `text` zum Setzen desselben bereitstellt. Die Klasse `HasInput` vereint alle Widgets mit Eingabefunktionalität und bietet die Funktionen `get` und `set`.
Ein nützlicher Weg Widgets zu plazieren sind Layout–Kombinatoren. Es gibt davon viele verschiedene, z.B.

```
(<<), (^^), (<|<), (^-^) :: Widget -> Widget -> Widget
```

mit folgender Bedeutung:

`v << w` plaziert Widget `w` rechts von Widget `v`.

`v ^^ w` plaziert Widget `w` unter Widget `v`.

`v <|< w` plaziert Widget `w` rechts von Widget `v`. Streckt das niedrigere Widget auf die Höhe des anderen.

`v ^-^ w` plaziert Widget `w` linksbündig unter Widget `v`.

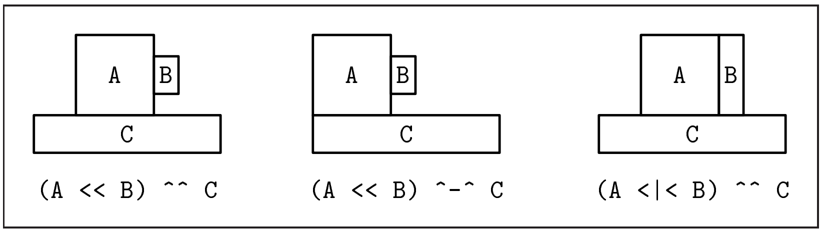


Abbildung 1.1.7: Die Layout–Kombinatoren in TkGofer ermöglichen die Beschreibung grundlegender Anordnungen von Widgets.

Mit diesen Layout–Kombinatoren lassen sich dann, wie in Abbildung 1.1.7 zu sehen, leicht die wichtigsten Layouts beschreiben.

1.1.7 Pictures

Ein Beispiel für eine in Haskell eingebettete Sprache ist Pictures von SIGBJORN FINNE und SIMON PEYTON JONES [FJ95]. Bilder werden mit Hilfe des Datentyps *Picture* beschrieben, der drei Arten von Konstruktoren enthält. Erstens solche, die primitive Bilder erzeugen wie Text, Linienzüge oder Bitmaps. Zweitens die Konstruktoren *Move*, *Transform* und *Clip* zum Verändern von Bildern und schließlich *Overlay* und *ConstrainOverlay* zur Kombination und relativen Plazierung von Teilbildern.

$$\begin{array}{lcl}
\textbf{data}\ \mathit{Picture} & = & \mathit{NullPic} \mid \mathit{Point} \mid \mathit{Text\ String} \\
 & \mid & \mathit{PolyLine}\ [\mathit{Translation}] \mid \mathit{Rect\ Size} \\
 & \mid & \mathit{Raster\ Raster} \mid \mathit{Move\ Offset\ Picture} \\
 & \mid & \mathit{Transform\ Transform\ Picture} \\
 & \mid & \mathit{Overlay\ Picture\ Picture} \\
 & \mid & \mathit{ConstrainOverlay\ RelSize\ RelSize\ Picture\ Picture} \\
 & \mid & \mathit{Clip\ Picture\ Picture} \\
 & \mid & \ldots
\end{array}$$

```
light :: Colour -> String -> Picture
light col lab =
 ConstrainOverlay
  (RelFixed True 20)
  (RelFixed True 20)
  (withColour black $ centre $ Text lab)
  (filledCircle col 2)

redTLight      = light red     "R"
orangeTLight   = light orange  "O"
greenTLight    = light green   "G"

besideSpace :: Unit -> Picture
            -> Picture -> Picture
besideSpace spc picA picB =
 beside
   picA
   (Transform (xlt (spc,0)) $
   moveWest picB)
```

```
lights =
 foldr1
   (besideSpace 10)
   [redTLight, orangeTLight,
    greenTLight]

trafficLight =
 ConstrainOverlay
   (RelFixed True 20)
   (RelFixed True 20)
   (Move (OffDir Centre)
         lights)
   (Move (OffDir Centre)
         (Rectangle (2,2)))
```

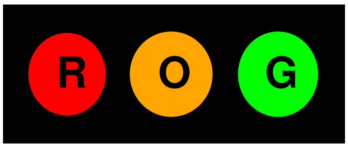


Abbildung 1.1.8: Ein Beispiel für eine Bildbeschreibung mit Pictures aus [FJ95].

Abstände und Positionen können relativ zur Bounding Box eines Teilbildes angegeben werden. Leider lassen sich Teilbildern und Punkten keine Namen zuordnen, um später darauf Bezug zu nehmen. Punkte müssen deshalb immer direkt als Koordinaten angegeben werden, was die Leistungsfähigkeit als allgemeine Bildbeschreibungssprache einschränkt.

Diese Einschränkungen spielen allerdings bei der Verwendung der Beschreibungssprache Pictures im GUI–Toolkit Haeggis keine große Rolle. Abbildung 1.1.8 zeigt, wie sich mit den Primitiven erzeugte Bilder zu komplizierteren Gebilden kombinieren lassen. An die Qualität der Textfunktion sollte man allerdings, wie sofort ins Auge fällt, keine großen Anforderungen stellen: Der Buchstabe „R“ befindet sich nicht in der Mitte des roten Kreises.

1.2 Analyse und Motivation

Nachdem wir nun einige Beispiele zur Bildbeschreibung kennengelernt haben, wollen wir mit Abbildung 1.2.1 einen zusammenfassenden Überblick geben, der die Möglichkeiten zur Abstraktion berücksichtigt. Als Grad für die Abstraktion einer Bildbeschriebungssprache können wir die Menge absoluter Positions– und Maßangaben heranziehen.

Ein hoher Abstraktionsgrad ist bei der Bildbeschreibung einem niedrigeren vorzuziehen, damit dem Anwender möglichst Aufgaben, wie absolute Positionierung und der Anpassung von Größen, wie Längen abgenommen werden. Bei einem hohem Abstraktionsniveau sind nachträgliche Veränderungen leicht möglich und erfordern im Idealfall keine manuellen Korrekturen irgendwelcher Größen. Wenn die Beschreibungssprache direkt komplexe Objekte wie z.B. Bäume unterstützt, muß der Anwender keine Mühe in deren Layout investieren, da dieses automatisch hergeleitet wird. Der Anwender kann sich mehr auf den Inhalt konzentrieren, statt auf die genaue Beschreibung des Layouts.

Wenn eine Sprache von sich aus nur einen niedrigen Abstraktionsgrad bietet, wie z.B. *functional* PostScript, ist es umso wichtiger, daß der Anwender in der Lage ist, auf einfache Weise eigene Abstraktionen für komplexere Objekte zu formulieren. In Abbildung 1.2.1 sehen wir, daß die Bildung von Abstraktionen in eingebetteten Sprachen besonders einfach ist. Deshalb werden auch wir unsere Bildbeschreibungssprache einbetten.

	Abstraktionsgrad der grafischen Elemente	Definition eigener Abstraktionen	Realisierung
XY-pic	mittel	nicht möglich	LaTeX Markosammlung
METAPOST	mittel	knifflig	externes Programm
PIC	hoch	exterm kompliziert	externes Programm
functional PostScript	mittel	sehr einfach	eingebettete Sprache
mlPicTeX	sehr hoch	sehr einfach	eingebettete Sprache
TkGofer	sehr hoch/speziell	sehr einfach	eingebettete Sprache
Pictures	hoch	sehr einfach	eingebettete Sprache

Abbildung 1.2.1: Klassifikation der vorgestellten Bildbeschreibungssprachen.

Wir wollen nun untersuchen, welche anderen Konzepte und Ideen der vorgestellten Bildbeschreibungssprachen wir für unseren Ansatz übernehmen können.

- Ein positives Vorbild sind die Erweiterungen von XY-pic, die kleine Spezialsprachen für besondere Anwendungen bereitstellen. Leider ist es nicht möglich, eigene Erweiterungen mit den Mitteln von XY-pic zu entwerfen. Negativ ist auch, daß die Syntax fast nur Sonderzeichen verwendet. Damit ergeben sich zwar kurze Bildbeschreibungen, aber die Bedeutung der Anweisungen kann sich ein Mensch kaum merken. Wir werden deshalb für unsere Sprache viele „sprechende" Textbefehle verwenden.

- Die Möglichkeit von METAPOST, mit **Gleichungssystemen** Bilder zu beschreiben, ist auf sehr natürliche Weise deskriptiv. Diese Art der Beschreibung werden wir deshalb auch anbieten.

- Die Idee, viele relative Positiononierungen vorzunehmen und dabei Bezugspunkte von Teilbildern zu referenzieren, übernehmen wir von PIC.

- Wie bei *functional* PostScript geschehen, werden wir unsere Bildbeschreibungssprache in eine funktionale Sprache **einbetten**. Wir benutzen die moderne und leistungsfähige Sprache Haskell. Von einer in Haskell eingebetten Bildbeschreibungssprache versprechen wir uns folgende Vorteile:

  – Es können leicht vom Anwender Abstraktionen für komplexe Ausdrücke definiert werden. Funktionale Sprachen haben a priori beschreibenden Charakter. Die Bildbeschreibungen sind normale Datentypen, auf denen sich auch Rechnungen formulieren lassen.

  – Das Berechnungsresultat eines funktionalen Programms kann sehr leicht visualisiert werden. Dazu genügt es, einen Konverter zu programmieren, der Ausdrücke des Berechnungsresultates in Ausdrücke der Bildbeschreibung überführt.

  – Die Syntaxprüfung der Bildbeschreibung übernimmt der Haskell–Compiler oder –Interpreter.

  – Das gilt ebenso für die Typprüfung.

- Die Sprache mlP$_{\lambda}$cTEX ist nicht nur in einer funktionalen Sprache eingebettet, sondern die Bildbeschreibung findet in der LATEX–Quelldatei an der Stelle statt, an der das Bild erscheint. Das erspart den Wechsel zwischen verschieden Programmen und kann bei einfachen Beschreibungen die Übersichtlichkeit erhöhen, für längere Beschreibungen dagegen wieder unübersichtlich werden. Wir wollen aber auch für unsere Sprache die Bildbeschreibung innerhalb der Dokumentenbeschreibung (LATEX–Quelldatei) ermöglichen.
- An der Sprache TkGofer überzeugen unter anderem die Layout–Kombinatoren. Für die wichtigsten Anordnungen von Bildern werden wir deshalb Operatoren definieren.
- Mit einem ähnlichen Datentyp, wie ihn die Bildbeschreibungssprache Pictures verwendet, beschreiben auch wir unsere Bilder. Wir werden allerding mit weniger und dafür allgemeineren Konstruktoren auskommen.

Zusammenfassend sind zwei Punkte für das Design unserer Bildbeschreibungssprache besonders wichtig: Die Einbettung in eine funktionale Programmiersprache und die Verwendung von Gleichungen zur Beschreibung geometrischer Zusammenhänge.

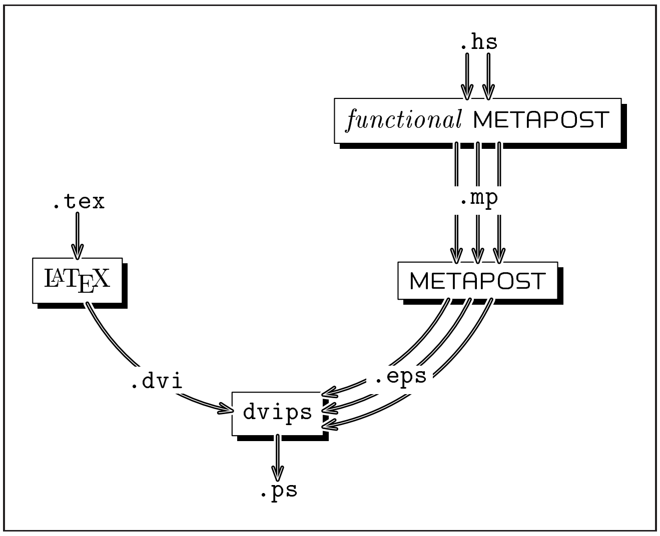


Abbildung 1.2.2: Das Programm dvips vereint ein LATEX–Dokument mit den von *functional* METAPOST generierten Bildern.

Als funktionales Front–End benutzen wir Haskell, denn es ist eine moderne funktionale Sprache, die für größere Anwendungen geeignet ist und für die effiziente Implementierungen existieren.
Das Back–End zur Generierung der Grafiken soll METAPOST bilden, da dieses Programm hochwertige PostScript–Grafiken erzeugt und in der Lage ist, lineare Gleichungssysteme zu lösen. Dies wird kombiniert mit einer Vielzahl leistungsfähiger Anweisungen, wie das Zeichnen komplexer Pfade oder die Einbindung beliebiger LATEX–Texte.
Unsere Aufgabe wird es sein, eine in Haskell eingebettete Bildbeschreibungssprache zu entwickeln und einen Compiler zu implementieren, der solche Beschreibungen in METAPOST– Programme übersetzt. METAPOST wird dann daraus PostScript–Grafiken generieren. Im letzten Schritt bindet das Programm dvips diese PostScript–Grafiken in das Dokument ein. Diesen Informationsfluß illustriert Abbildung 1.2.2. Die drei Pfeile deuten an, daß in einem Dokument mehrere Bilder vorkommen dürfen.
Wir wollen versuchen, eine universelle und möglichst kleine **Kernsprache** zu entwerfen und auf dieser Basis **Erweiterungen** für spezielle Anwendungen bereitzustellen. Der Name *functional* METAPOST, den wir unserer Bildbeschreibungssprache geben, soll die Einbettung in eine funktionale Sprache und die Verwendung von METAPOST andeuten.

1.3 Eine Gliederung der Arbeit

1 In diesem Kapitel haben wir verschiedene beschreibende Ansätze zur Erzeugung von Bildern untersucht. Die Einbettung einer Bildbeschreibungssprache in eine funktionale Sprache, in Kombination mit dem Programm METAPOST, ist ein vielversprechender Weg, den wir in dieser Arbeit gehen wollen.

2 Das zweite Kapitel ist eine kurze Einführung in die Sprache Haskell, in die wir die Bildbeschreibungssprache einbetten.

3 Das dritte Kapitel stellt unsere Bildbeschreibungssprache *functional* METAPOST in der Form eines Tutorials vor, in dem nur die wichtigsten Möglichkeiten vorgestellt werden. Damit sich der Leser trotz dieser reduzierten Darstellung ein vollständiges Bild von *functional* METAPOST machen kann, wird dieses Kapitel von Anhang A ergänzt.

4 Im vorigen Kapitel lernten wir eine bestimmte Art und Weise der Bildbeschreibung kennen. Im vierten Kapitel stellen wir Erweiterungen vor, die andere Konzepte der Beschreibung und eine Abstraktion für das Zeichnen von Bäumen bereitstellen.

5 In diesem Kapitel untersuchen wir, wie sich die Erweiterungen des letzten Kapitels mit Hilfe der in Kapitel 3 vorgestellten Kernsprache implementieren lassen.

6 Bevor wir uns mit der Implementierung der Kernsprache beschäftigen, lernen wir in Kapitel Sechs einige wichtige Grundlagen von METAPOST kennen.

7 Wir geben in diesem Kapitel Einblicke in die Implementierung von *functional* METAPOST.

8 Im achten und letzten Kapitel resümieren wir die Ergebnisse der Arbeit und geben Ideen für mögliche Erweiterungen an.

Kapitel 2

Haskell

Da *functional* METAPOST in Haskell eingebettet ist, sind zum weiteren Verständnis einige Grundkenntnisse von Haskell notwendig. Dieses Kapitel soll den Leser mit den wichtigsten Konzepten von Haskell vertraut machen. Erfahrungen mit anderen ☞ *funktionalen Programmiersprachen* sind dabei von Vorteil, aber nicht zwingend notwendig. Für eine weitergehende Einführung in Haskell empfiehlt sich das Studium des Artikels [HFP96] oder der Bücher [Bir98, Tho96].

2.1 Das Berechnungsmodell

Zu Beginn wollen wir ein Gefühl dafür vermitteln, wie Programme in Haskell aussehen und wie sie abgearbeitet werden. Wenn ein Algorithmus in einer funktionalen Programmiersprache wie Haskell entwickelt werden soll, geschieht dies in zwei Schritten.

1. Das Finden einer Repräsentation der benötigten Werte mit Hilfe
2. Datenstruktur von **Datenstrukturen** und
3. der Definition von **Rechenvorschriften** zwischen Ausdrücken dieser Datenstrukturen.

Als Beispiel schauen wir uns das Problem der Addition auf natürlichen Zahlen an. Bei der Darstellung der Zahlen orientieren wir uns an den ersten beiden PEANO Axiomen: Die Null ist eine natürliche Zahl und jede natürliche Zahl hat einen Nachfolger.

$$\mathbf{data}\ \mathit{Nat} \qquad = \mathit{Zero} | \mathit{Succ}\ \mathit{Nat}$$

Mit dem Datentyp *Nat* können wir für jede natürliche Zahl einen Ausdruck bilden.

$$\begin{array}{lcl} 0 & \equiv & \mathit{Zero} \\ 1 & \equiv & \mathit{Succ}\ \mathit{Zero} \\ 2 & \equiv & \mathit{Succ}\ \mathit{Succ}\ \mathit{Zero} \\ 3 & \equiv & \mathit{Succ}\ \mathit{Succ}\ \mathit{Succ}\ \mathit{Zero} \\ \vdots & & \vdots \end{array}$$

Die Rechenregeln der Addition orientieren sich an der Struktur des Datentyps *Nat*. Eine Regel für die Addition von *Zero* und eine zweite, die ebenso wie *Succ* rekursiv ist.

$$
\begin{array}{lll}
\mathit{add}\ \mathit{Zero}\ b & = b & \text{-- Regel 1} \\
\mathit{add}\ (\mathit{Succ}\ a)\ b & = \mathit{Succ}\ (\mathit{add}\ a\ b) & \text{-- Regel 2}
\end{array}
$$

Nachdem wir uns einen Datentyp und Regeln ausgedacht haben, kann das so definierte Programm zur Addition angewendet werden. Dazu muß zuerst ein Problem mit Hilfe eines Ausdrucks beschrieben werden. Die Lösung entsteht dann durch Ausrechnen, d.h. fortschreitende Anwendung der aufgestellten Rechenregeln. Die Anwendung einer Rechenregel heißt Reduktion.
Wir können z.B. die Summe der Zahlen Zwei (*Succ Succ Zero*) und Eins (*Succ Zero*) ausrechnen.

$$
\begin{array}{lll}
 & \mathit{add}\ (\mathit{Succ}\ (\mathit{Succ}\ \mathit{Zero}))\ (\mathit{Succ}\ \mathit{Zero}) & \text{-- reduziere mit Regel 2} \\
\Rightarrow & \mathit{Succ}\ (\mathit{add}\ (\mathit{Succ}\ \mathit{Zero})\ (\mathit{Succ}\ \mathit{Zero})) & \text{-- reduziere mit Regel 2} \\
\Rightarrow & \mathit{Succ}\ (\mathit{Succ}\ (\mathit{add}\ \mathit{Zero}\ (\mathit{Succ}\ \mathit{Zero}))) & \text{-- reduziere mit Regel 1} \\
\Rightarrow & \mathit{Succ}\ (\mathit{Succ}\ (\mathit{Succ}\ \mathit{Zero})) & \text{-- nicht weiter reduzierbar} \rightsquigarrow \text{Endergebnis}
\end{array}
$$

Solange eine Regel angewendet werden kann, schreitet die Reduktion fort, bis das Endergebnis feststeht. Haskell kennt den Datentyp *Int* für natürliche Zahlen, der gegenüber *Num* eine effizientere Darstellung und die Rechnung mit gewöhnlichen Rechenoperatoren erlaubt.
Wir wollen uns noch ein weiteres Beispiel für die Lösung von Problemen mit Haskell anschauen. Angenommen, wir wollen einen Baum konstruieren, der in jedem Knoten eine Zahl speichert. Es sind zwei Fälle zu unterscheiden: Entweder der Baum ist leer oder er besteht aus einem Knoten mit zwei Teilbäumen und einem Zahlenwert. Mit dieser Überlegung erhalten wir direkt den Datentyp:

$$
\begin{array}{lll}
\mathbf{data}\ \mathit{Tree} & = & \mathit{Empty} \\
 & \mid & \mathit{Node}\ \mathit{Tree}\ \mathit{Int}\ \mathit{Tree}
\end{array}
$$

Man nennt *Empty*, *Node*, *Zero* und *Succ* Konstruktoren. Sie beginnen genau wie Typbezeichnungen immer mit Großbuchstaben. Der Aufbau einer Datenstruktur findet sich meistens auf ähnliche Weise in den Köpfen der Regeln wieder, um auf diesem Weg eine bedingte Regelanwendung zu erreichen. Bei einer Reduktion wird nämlich immer die oberste Regel angewendet, deren Kopf ein Muster enthält, das auf den aktuellen Ausdruck paßt. Dies nennt man auch Mustervergleich (engl. pattern matching).

$$
\begin{array}{ll}
\mathit{addTree}\ \mathit{Empty} & = 0 \\
\mathit{addTree}\ (\mathit{Node}\ l\ a\ r) & = \mathit{addTree}\ l + a + \mathit{addTree}\ r
\end{array}
$$

Diese beiden Regeln, bilden die Funktion mit Namen *addTree*. Es gibt eine alternative Definition der Funktion, die mit nur einer Regel auskommt. Sie verwendet einen **case**–Ausdruck. Hier findet der Mustervergleich an anderer Stelle statt.

$$
\begin{array}{ll}
\mathit{addTree}\ a & = \mathbf{case}\ a\ \mathbf{of} \\
 & \qquad \mathit{Empty} \rightarrow 0 \\
 & \qquad \mathit{Node}\ l\ a\ r \rightarrow \mathit{addTree}\ l + a + \mathit{addTree}\ r
\end{array}
$$

2.2 Typen

Typen sind in Haskell sehr wichtig. Jedem Ausdruck und jeder Funktion läßt sich ein Typ zuordnen. Entweder ist der Typ einer Funktion vor der dieser notiert, eine sog. Typsignatur oder der Typ wird automatisch während des Übersetzungsvorgangs hergeleitet (Typinferenz).

Die Typen von *Zero*, *add* und *addTree* notiert man z.B. als

$$\begin{array}{lll} \mathit{Zero} & :: & \mathit{Nat} \\ \mathit{add} & :: & \mathit{Nat} \to \mathit{Nat} \to \mathit{Nat} \\ \mathit{addTree} & :: & \mathit{Tree} \to \mathit{Int} \end{array}$$

Der Typ $\mathit{Nat} \to \mathit{Nat} \to \mathit{Nat}$ steht für eine Funktion, die zwei Parameter des Typs *Nat* besitzt und einen Ausdruck vom Typ *Nat* zurückliefert. Der Typ $\mathit{Tree} \to \mathit{Int}$ steht für eine Funktion, mit einem Parameter des Typs *Tree*, die einen Ausdruck des Typs *Int* zurückliefert.
Warum gibt es mit Typsignaturen die Möglichkeit, den Typ einer Funktion anzugeben, wenn dieser auch automatisch hergeleitet werden kann? Dafür gibt es verschiedene Gründe. Erstens erhöht es die Lesbarkeit eines Programmtextes, wenn man auf einen Blick den Typ einer Funktion sehen kann. Zweitens kann es nützlich sein, den Typ einer Funktion anzugeben, um eine Rückmeldung zu erhalten, wenn der Compiler einen anderen Typ herleitet, was auf einen Fehler hindeuten kann.
Neben normalen Typen kennt Haskell auch polymorphe Typen. Diese sind z.B. nötig, um allgemeine Listen zu beschreiben.

$$\begin{array}{lll} \mathbf{data}\ \mathit{List}\ a & = & \mathit{NIL} \mid \mathit{Cons}\ a\ (\mathit{List}\ a) \end{array}$$

Dabei ist *a* eine sog. Typvariable, die für einen beliebigen Typ stehen kann. Das bedeutet, in der Liste können beliebige Elemente gespeichert sein, solange alle vom gleichen Typ sind (Typhomogenität).

$$\mathit{Cons}\ \texttt{'l'}\ (\mathit{Cons}\ \texttt{'i'}\ (\mathit{Cons}\ \texttt{'s'}\ (\mathit{Cons}\ \texttt{'t'}\ \mathit{NIL}))) :: \mathit{List}\ \mathit{Char}$$

Zur Konkatenation zweier Listen können wir einen Operator definieren. Operatoren bestehen aus Symbolzeichen und werden in infix–Notation angewendet. Um einen Operator in postfix–Notation, d.h. in der Schreibweise normaler Funktionen zu verwenden, ist dieser mit runden Klammern einzuschließen, d.h. a_1 & a_2 ist äquivalent zu (&) $a_1\ a_2$.

$$\begin{array}{lll} (\&) & :: & \mathit{List}\ a \to \mathit{List}\ a \to \mathit{List}\ a \\ \mathit{NIL}\ \&\ \mathit{bs} & = & \mathit{bs} \\ (\mathit{Cons}\ a\ \mathit{as})\ \&\ \mathit{bs} & = & \mathit{Cons}\ a\ (\mathit{as}\ \&\ \mathit{bs}) \end{array}$$

2.3 Listen

Haskell hat einen vordefinierten Listentyp. Die leere Liste $[\,] :: [a]$ entspricht $\mathit{NIL} :: \mathit{List}\ a$ in unserem obigen Beispiel und der Listenkonstruktor $(:) :: a \to [a] \to [a]$, der eine Liste um ein weiteres Element am Anfang erweitert, entspricht $\mathit{Cons} :: a \to \mathit{List}\ a \to \mathit{List}\ a$.

$$\begin{array}{lll} \mathbf{data}\ [a] & = & [\,] \mid a : [a] \end{array}$$

Die Konkatenation zweier Listen ist wie folgt definiert.

$$\begin{array}{lll} (+\!\!+) & :: & [a] \to [a] \to [a] \\ [\,] +\!\!+ b & = & b \\ (a : \mathit{as}) +\!\!+ b & = & a : (\mathit{as} +\!\!+ b) \end{array}$$

Die etwas umständliche Schreibweise $1 : 2 : 3 : 4 : 5 : [\,]$ läßt sich zu $[1, 2, 3, 4, 5]$ abkürzen. Die Abkürzung des Ausdrucks `'H'`:`'a'`:`'s'`:`'k'`:`'e'`:`'l'`:[ ] ist `"Haskell"`. Zeichenketten, die den Typ *String* haben sind als Listen über dem Typ *Char* repräsentiert.
Ein nützliches Feature sind arithmetische Folgen; z.B. beschreiben folgende Ausdrücke die Liste der natürlichen Zahlen, die Liste der fünf ersten natürlichen Zahlen, oder die Liste der geraden Zahlen zwischen zehn und zwanzig.

$$
\begin{aligned}
&[1\,..] :: [Int] \\
&[1\,..\,5] :: [Int] \\
&[10, 12\,..\,20] :: [Int]
\end{aligned}
$$

Zusätzlich lassen sich Bedingungen für die Elemente in der Liste festlegen. Das folgende Beispiel für eine ineffiziente Faktorisierung verdeutlicht dies.

$$
\begin{aligned}
&\mathit{factors} && :: \mathit{Int} \rightarrow [\mathit{Int}] \\
&\mathit{factors}\ n && = [m \mid m \leftarrow [1\,..\,n], \mathit{mod}\ n\ m \equiv 0]
\end{aligned}
$$

Der Ausdruck $m \leftarrow [1\,..\,n]$ ist ein sogenannter Generator, er schränkt den Definitionsbereich ein und bindet die Variable m der Reihe nach an alle Elemente der Liste $[1\,..\,n]$. Genau dann, wenn der Wächter $\mathit{mod}\ n\ m \equiv 0$ den Wert *True* ergibt, wird m ein Element der beschriebenen Liste.

2.4 Typklassen

In vielen Fällen sind Funktionen oder Operatoren für eine bestimmte Menge (oder Klasse) von Typen sinnvoll. Diese Typen kann man zu einer Typklasse zusammenfassen. Es gibt z.B. für Typen, die den Test auf Gleichheit unterstützen, die Typklasse *Eq*. Diese Klasse mit den Operatoren $(\equiv)$ und $(\not\equiv)$ ist wie folgt definiert.

$$
\begin{aligned}
&\textbf{class}\ \mathit{Eq}\ a\ \textbf{where} \\
&\qquad (\equiv) && :: a \rightarrow a \rightarrow \mathit{Bool} \\
&\qquad (\not\equiv) && :: a \rightarrow a \rightarrow \mathit{Bool} \\
&\qquad x \not\equiv y && = \neg\ (x \equiv y)
\end{aligned}
$$

Wenn wir für einen Typ eine Instanz dieser Typklasse *Eq* definieren, können wir die Vergleichsoperationen $(\equiv)$ und $(\not\equiv)$ auf diesen Typ anwenden. Es genügt, die Funktion für den Test auf Gleichheit anzugeben, da die Funktion für Ungleichheit aus dieser abgeleitet werden kann. Wir geben eine Instanz für den Typ *Tree* aus Abschnitt 2.1 an.

$$
\begin{aligned}
&\textbf{instance}\ \mathit{Eq}\ \mathit{Tree}\ \textbf{where} \\
&\qquad \mathit{Node}\ l_1\ a_1\ r_1 \equiv \mathit{Node}\ l_2\ a_2\ r_2 \\
&\qquad && = l_1 \equiv l_2 \wedge a_1 \equiv a_2 \wedge r_1 \equiv r_2 \\
&\qquad \mathit{Empty} \equiv \mathit{Empty} && = \mathit{True} \\
&\qquad \_ \equiv \_ && = \mathit{False}
\end{aligned}
$$

Die Typklasse für *Ord*, die Ordnungsrelationen definiert, benutzt den Gleichheitsoperator und setzt deshalb voraus, daß ein Typ Instanz der Klasse *Eq* ist, um Instanz der Klasse *Ord* sein zu können.

$$
\begin{aligned}
&\textbf{instance}\ (\mathit{Eq}\ a) \Rightarrow \mathit{Ord}\ a\ \textbf{where} \\
&\qquad \mathit{compare} && :: a \rightarrow a \rightarrow \mathit{Ordering} \\
&\qquad (<), (\leqslant), (\geqslant), (>) && :: a \rightarrow a \rightarrow \mathit{Bool} \\
&\qquad \mathit{max}, \mathit{min} && :: a \rightarrow a \rightarrow a \\
&\qquad \mathit{compare}\ x\ y \\
&\qquad\qquad \mid x \equiv y && = \mathit{EQ} \\
&\qquad\qquad \mid x \leqslant y && = \mathit{LT} \\
&\qquad\qquad \mid \mathit{otherwise} && = \mathit{GT} \\
&\qquad ..
\end{aligned}
$$

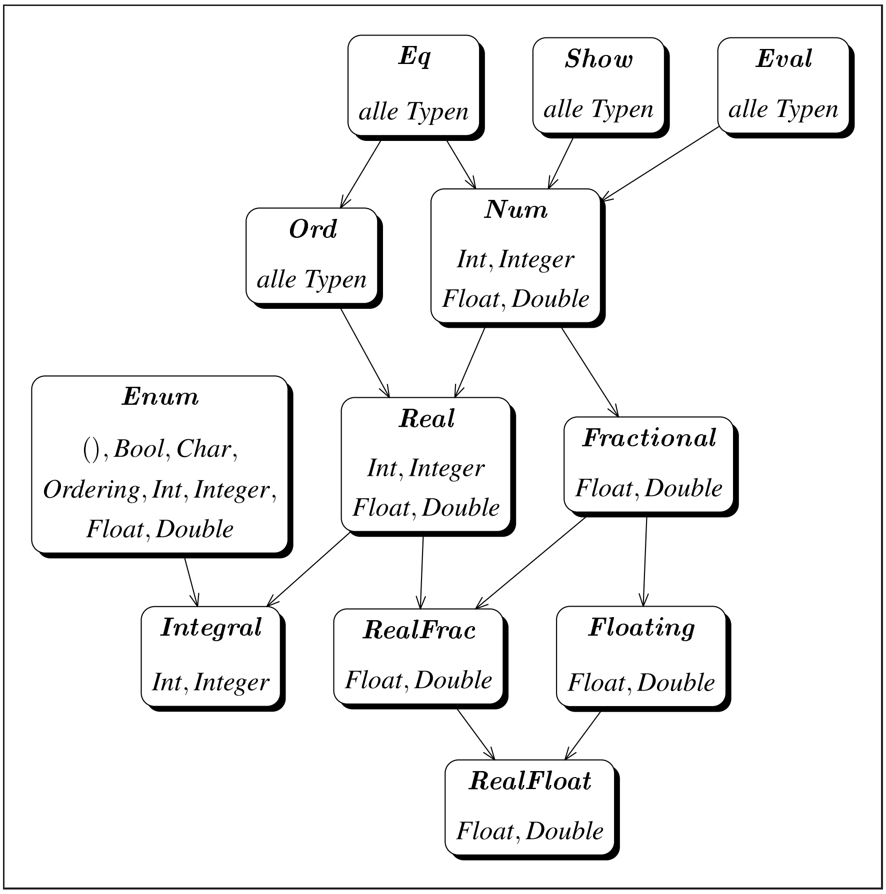


Abbildung 2.4.1: Ausschnitt aus der Klassenhierarchie für Standardtypen von Haskell. Siehe [PHe97b], S. 67.

Wenn wir die übrigen Typklassen von Haskell und ihre Anhängigkeiten betrachten, ergibt sich eine Hierarchie dieser Typklassen, die Abbildung 2.4.1 wiedergibt.
Eine Reihe von kanonischen Instanzen kann Haskell auf Wunsch automatisch herleiten. Dies gilt z.B. für die Klasse *Eq* oder die Klasse *Show*, die eine Konvertierung des Datentyps in einen String implementiert. Zur automatischen Generierung der Instanzen genügt es, die Definition des Typs um **deriving** (*Show*, *Eq*) zu ergänzen.

Kapitel 3

Einführung in *functional* METAPOST

Wie schon in Kapitel 1 erwähnt, bedeutet eine Einbettung von *functional* METAPOST in die Programmiersprache Haskell, daß wir zur Bildbeschreibung Haskell–Ausdrücke verwenden. Der Grundtyp dieser Ausdrücke, der ein Bild repräsentiert, ist *Picture*.
Die Sprache besteht aus Funktionen, die atomare Bilder wie z.B. Text erzeugen und Funktionen, mit denen sich mehrere Bilder zu einem neuen Bild kombinieren lassen. Aber es gibt auch die Möglichkeit, Bilder zu umrahmen, wobei verschiedene Rahmenformen zur Auswahl stehen. Ein Bild läßt sich um beliebige Pfade ergänzen, die zwischen Punkten verlaufen, denen wir Namen geben können. Wenn ein Pfad zwei Teilbildern verbindet, wird dieser automatisch an den Grenzen der Bilder abgeschnitten, damit der Pfad nicht durch ein Bild hindurch verläuft. Wir nennen diese Grenze eines Bildes seine Bounding Box, die wir uns in erster Annäherung als ein Rechteck vorstellen dürfen, welches das Bild umschließt.
Neben Pfaden können auch Flächen gezeichnet werden oder Bilder lassen sich affin transformieren. Nicht zuletzt ermöglicht es *functional* METAPOST dem Anwender, mit Hilfe von Gleichungssystemen, geometrische Beziehungen herzuleiten oder ein Layout, für die relativen Positionen mehrerer Bilder, zu definieren.
Um bestimmte Eigenschaften von Bildern oder Pfaden, wie Farben oder Zeichenstifte, verändern zu können, sind diese Eigenschaften als Attribute gespeichert. Die Attribute haben sinnvolle Voreinstellungen, die sich bei Bedarf, mit Hilfe der sog. Attributierungsfunktionen, ändern lassen. Der Ausdruck

$$\mathit{text}\ \texttt{"red"}\ \#\ \mathit{setColor}\ \mathit{red}$$

erzeugt z.B. einen roten Text, wobei wir den (#)–Operator als umgekehrte Applikation definieren, um eine übersichtlichere Notation zu erreichen.

$$\begin{array}{lcl} (\#) & :: & a \to (a \to b) \to b \\ a\,\#\,f & = & f\ a \end{array}$$

Die Namen der Attributierungsfunktionen, die Änderungen ermöglichen, beginnen mit *set*. Daneben gibt es Funktionen, die das Auslesen von Attributen erlauben und die mit *get* beginnen.
Dieses Kapitel soll eine Einführung sein und einen schnellen Einstieg in *functional* METAPOST ermöglichen. Deshalb enthält es lediglich oft benötigte Funktionen. Anhang A ergänzt dieses Kapitel zur Darstellung des vollständigen Funktionsumfangs.

Unser Vorgehen setzt nicht voraus, daß erst alle Anweisungen und Ausdrücke beschrieben sind, bevor wir sie verwenden, da ihre Namen genug über ihre Funktion verraten, um die Beispiele zu verstehen. Wir werden *functional* METAPOST so beschreiben, wie es in einer fiktiven funktionalen Sprache aussehen könnte. Das bedeutet, daß wir an einigen Stellen keine Rücksicht auf Einschränkungen nehmen, die uns Haskell auferlegt. In Abschnitt 3.16 gehen wir auf diese Punkte näher ein und finden eine Lösung.

3.1 Atomare Bilder

Atomare Bilder sind die Grundbausteine, aus denen wir später mittels Kombination komplexere Bilder konstruieren können. Zwei solche atomaren Konstrukte sind die Einbindung beliebiger LaTeX–Ausdrücke und das Erzeugen eines rechteckigen, leeren Bildes bestimmter Größe.
Wenn das Bild in ein LaTeX–Dokument eingefügt wird, ist es ein Vorteil, wenn sich die Texte des Bildes nicht vom übrigen Dokument unterscheiden. Da die gleichen Zeichensätze und Formatierungen zur Anwendung kommen, paßt sich die Erscheinung des Bildes sehr gut in den Text ein. Außerdem steht auch innerhalb des Bildes die volle Mächtigkeit von LaTeX zur Verfügung.

$$\begin{array}{lcl} \mathit{tex} & :: & \mathit{String} \to \mathit{Picture} \end{array}$$

Die Funktion *space* erzeugt ein leeres Bild mit einer rechteckigen Bounding Box, welche die angegebene Breite und Höhe hat.

$$\begin{array}{lcl} \mathit{space} & :: & \mathit{Numeric} \to \mathit{Numeric} \to \mathit{Picture} \end{array}$$

Leicht lassen sich aus den beiden atomaren Funktionen *tex* und *space* weitere nützliche Funktionen definieren. Z.B. zum automatischen Satz im Mathematikmodus von LaTeX

$$\begin{array}{lcl} \mathit{math} & :: & \mathit{String} \to \mathit{Picture} \\ \mathit{math}\ p & = & \mathit{tex}\ (\texttt{"\$"} +\!\!+\ p +\!\!+ \texttt{"\$"}) \end{array}$$

oder in Anlehnung an die entsprechenden Befehle in LaTeX zum Erzeugen horizontaler und vertikaler Leerräume:

$$\begin{array}{lcl} \mathit{hspace}, \mathit{vspace} & :: & \mathit{Numeric} \to \mathit{Picture} \\ \mathit{hspace}\ n & = & \mathit{space}\ n\ 0 \\ \mathit{vspace}\ n & = & \mathit{space}\ 0\ n \end{array}$$

Eine Funktion, die wie sich noch zeigen wird, auch nützlich sein kann, erzeugt das leere Bild.

$$\begin{array}{lcl} \mathit{empty} & :: & \mathit{Picture} \\ \mathit{empty} & = & \mathit{space}\ 0\ 0 \end{array}$$

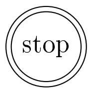
3.2 Rahmen

Oft möchte man ein Bild umrahmen, sei es um eine Abgrenzung oder eine Zusammengehörigkeit von Bildelementen auszudrücken. *functional* METAPOST bietet dazu Rahmen verschiedener Form an, die ihre Größe automatisch an das eingeschlossene Bild anpassen. Der Ausdruck

$$\mathit{box}\ (\mathit{tex}\ \texttt{"rectangular"}) \quad :: \quad \mathit{Picture} \quad \Big| \quad \boxed{\text{rectangular}}$$

bewirkt z.B. eine rechteckige Umrahmung des Textes `"rechteckig"`.[1]
Eine mehrfache Anwendung solcher Rahmenfunktionen wie im Ausdruck

$$\mathit{circle}\ (\mathit{circle}\ (\mathit{tex}\ \texttt{"stop"}))\quad ::\ \mathit{Picture}$$



erzeugt entsprechend viele Rahmen.
Für speziellere Umrahmungen gibt es noch die Funktionen *oval*, *triangle*, *rbox* und viele mehr. Der Abstand der Umrandung zum Bild läßt sich mit den Funktionen

$$\mathit{setDX}\quad ::\ \mathit{Double} \to \mathit{Picture} \to \mathit{Picture}$$

und

$$\mathit{setDY}\quad ::\ \mathit{Double} \to \mathit{Picture} \to \mathit{Picture}$$

beeinflussen. Der Ausdruck

$$\begin{array}{ll} \mathit{rbox}\ 5\ (\mathit{tex}\ \texttt{"rounded␣box"}) & \\ \#\mathit{setDX}\ 10 & \\ \#\mathit{setDY}\ 5 & ::\ \mathit{Picture} \end{array}$$

erzeugt einen Rahmen mit abgerundeten Ecken vom Radius 5, der einen etwas größeren Abstand zum Text hat. Wir können mit Hilfe des runden Rahmens auch einen Ausdruck für einen Punkt, d.h. einen kleinen gefüllten Kreis definieren.

$$\begin{array}{lll} \mathit{dot} & :: & \mathit{Picture} \\ \mathit{dot} & = & \mathit{circle}\ \mathit{empty} \\ & & \#\mathit{setBGColor}\ \mathit{black} \\ & & \#\mathit{setDX}\ 0.75 \end{array}$$

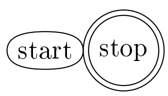
3.3 Kombination von Bildern

Bisher haben wir nur die Möglichkeit kennengelernt, atomare Bilder zu erzeugen und diese mit einem Rahmen zu versehen. Nun wollen wir zwei Bilder zu einem neuen kombinieren. Für die häufigsten Kombinationen, nämlich die Anordnung nebeneinander und übereinander gibt es zwei Kombinatoren.

$$\begin{array}{lll} (\boxbar) & :: & \mathit{Picture} \to \mathit{Picture} \to \mathit{Picture} \\ (\boxminus) & :: & \mathit{Picture} \to \mathit{Picture} \to \mathit{Picture} \end{array}$$

Nehmen wir z.B. den Ausdruck

$$\mathit{oval}\ \texttt{"start"} \boxbar (\mathit{circle}\ (\mathit{circle}\ \texttt{"stop"}))$$



Es entsteht ein neues Bild, in dem die beiden Bilder horizontal, direkt nebeneinander, angeordnet sind. Die Kombinatoren $(\boxbar)$ und $(\boxminus)$ sind assoziativ. D.h. die Klammerung spielt keine Rolle:

[1] Im weiteren Text werden wir noch oft den Ausdruck, der ein Bild beschreibt und das berechnete Bild gegenüberstellen. Wir trennen dann beides mit einer vertikalen oder horizontalen Linie.

$$
\begin{array}{lcl}
a \mathbin{◫} (b \mathbin{◫} c) & \equiv & (a \mathbin{◫} b) \mathbin{◫} c \\
a \boxminus (b \boxminus c) & \equiv & (a \boxminus b) \boxminus c
\end{array}
$$

Das leere Bild ist ein neutrales Element.

$$
\begin{array}{lcl}
a \mathbin{◫} \mathit{empty} & \equiv & a \\
a \boxminus \mathit{empty} & \equiv & a
\end{array}
$$

Wir erhalten also die zwei Gruppen $((\mathbin{◫}), \mathit{empty})$ und $((\boxminus), \mathit{empty})$.
Oft soll sich zwischen den Bildern ein kleiner Abstand befinden. Für diesen Fall können wir schon mit unseren bisherigen Sprachmitteln die entsprechenden Kombinatoren ableiten.

$$
\begin{array}{lcl}
(\mathbin{▥}) & :: & \mathit{Picture} \rightarrow \mathit{Picture} \rightarrow \mathit{Picture} \\
a \mathbin{▥} b & = & a \mathbin{◫} \mathit{hspace}\ 8 \mathbin{◫} b
\end{array}
$$

$$
\begin{array}{lcl}
(\mathbin{▤}) & :: & \mathit{Picture} \rightarrow \mathit{Picture} \rightarrow \mathit{Picture} \\
a \mathbin{▤} b & = & a \boxminus \mathit{vspace}\ 8 \boxminus b
\end{array}
$$

Abbildung 3.3.1 zeigt ein Bild, das wir nur mit Text, Rahmen und Kombinatoren erzeugen können. Dabei erzeugt die Funktion *rbox20* einen Rahmen mit runden Ecken vom Radius $\leqslant 20$.

$$
\begin{array}{lcl}
\mathit{rbox20}\ a & = & \mathit{rbox}\ 20\ a \mathbin{\#} \mathit{setDX}\ 8 \mathbin{\#} \mathit{setDY}\ 6
\end{array}
$$

Wir haben uns bisher nicht darum gekümmert, in welchen Einheiten Werte angegeben werden. Dies wollen wir jetzt präzisieren. METAPOST verwendet als Grundeinheit PostScript Punkte, was $1/72$ Inch entspricht. Der Ausdruck *hspace* 8 erzeugt also ein leeres Bild der Breite $1/9$ Inch, was ungefähr 2.82 mm entspricht. Um Abstände in anderen Einheiten angeben zu können, gibt es vordefinierte Konstanten, mit denen die Werte entsprechend zu multiplizieren sind.

$$
\begin{array}{lcl}
\mathit{mm}, \mathit{pt}, \mathit{cm} & :: & \mathit{Numeric} \\
\mathit{mm} & = & 2.83464 \\
\mathit{pt} & = & 0.99626 \\
\mathit{cm} & = & 28.34645
\end{array}
$$

Der Ausdruck $2 * \mathit{cm}$ gibt den Wert zwei Zentimeter und der Ausdruck $1 * \mathit{pt}$ den Wert eines Druckerpunktes, der $\frac{1}{72.27}$ Inch entspricht, an.
Nun wollen wir die Kombinatoren noch etwas verallgemeinern. Auch für mehr als zwei Bilder gibt es Funktionen, die eine Anordnung in einer Reihe bzw. Spalte bewirken.

$$
\begin{array}{lcl}
\mathit{row} & :: & [\mathit{Picture}] \rightarrow \mathit{Picture} \\
\mathit{column} & :: & [\mathit{Picture}] \rightarrow \mathit{Picture}
\end{array}
$$

Beide Funktionen könnte man so definieren:[2]

$$
\begin{array}{lcl}
\mathit{row} & = & \mathit{foldr}\ (\mathbin{◫})\ \mathit{empty} \\
\mathit{column} & = & \mathit{foldr}\ (\boxminus)\ \mathit{empty}
\end{array}
$$

[2]Die Funktion *foldr* setzt einen binären Operator auf eine Liste fort.

$$
\begin{array}{lcl}
\mathit{foldr}\ f\ z\ [\,] & = & z \\
\mathit{foldr}\ f\ z\ (x : \mathit{xs}) & = & f\ x\ (\mathit{foldr}\ f\ z\ \mathit{xs})
\end{array}
$$

```
rbox20 (tex "set␣of␣all␣languages"
    ⊟rbox20 (tex "rec.␣enum.␣languages␣or␣type␣0"
        ⊟rbox20 (tex "decideable␣languages"
            ⊟rbox20 (tex "contextsensitive␣or␣type␣1"
                ⊟rbox20 (tex "contextfree␣or␣type␣2"
                    ⊟rbox20 (tex "regular␣or␣type␣3"))))))
```

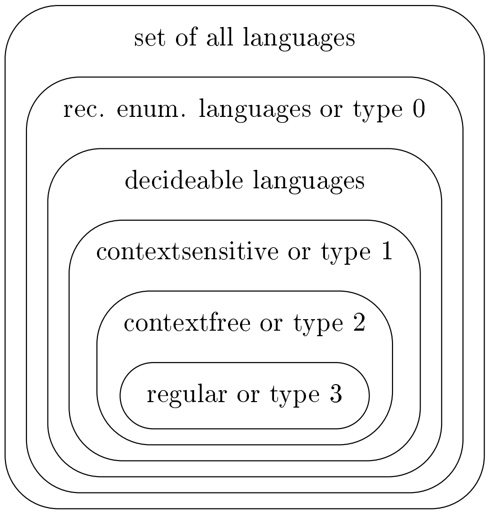


Abbildung 3.3.1: Eine Grafik mit Rahmen und Kombinatoren, aus [Sch97].

Ebenfalls nützlich sind Varianten dieser Funktionen mit wählbarem Abstand zwischen den Bildern.

$$
\begin{array}{lll}
\mathit{rowSepBy} & :: & \mathit{Numeric} \to [\mathit{Picture}] \to \mathit{Picture} \\
\mathit{columnSepBy} & :: & \mathit{Numeric} \to [\mathit{Picture}] \to \mathit{Picture}
\end{array}
$$

Analog zu *row* lassen sich die Funktionen wie folgt definieren:[3]

$$\mathit{rowSepBy}\ n \quad = \quad \mathit{foldr}\ (\lambda a\ b \to a \boxbar \mathit{hspace}\ n \boxbar b)\ \mathit{empty}$$

Wir können nun leicht das Beispiel einer Bildbeschreibung mit der Sprache Pictures aus Abbildung 1.1.8 auf Seite 8 mit unserer Beschreibungssprache nachgestalten. Abbildung 3.3.2 zeigt, wie einfach dies ist. Für ein Ampellicht definieren wir eine Funktion, die einen mit Abstand 10 rund umrahmten Text in der gewünschten Farbe erzeugt. Die drei Lichter werden in einer Spalte mit Abstand 10 kombiniert und schließlich mit Abstand 10 umrahmt.
Diese Bildbeschreibung ist in der Sprache Pictures länger und etwas weniger verständlich. Das liegt zum einen daran, daß keine vordefinierten Kombinatoren vorhanden sind und zum anderen, daß dort das Konzept von Rahmen fehlt.

[3] Der Ausdruck $\lambda a\ b \to a \boxbar \mathit{hspace}\ n \boxbar b$ ist eine sog. Lambda–Abstraktion und zu lesen als: „Die Funktion, die die Parameter a und b auf den Ausdruck $a \boxbar \mathit{hspace}\ n \boxbar b$ abbildet.“ Mit Lambda–Abstraktionen lassen sich also Ausdrücke für Funktionen angeben, die keinen Funktionsnamen haben müssen.

```
let light s c = circle (tex s)
                # setDX 10
                # setBGColor c in
box (columnSepBy 10
       [light "R" red,
        light "O" yellow,
        light "G" green])
#setBGColor black
#setDX 10
#setDY 10
```

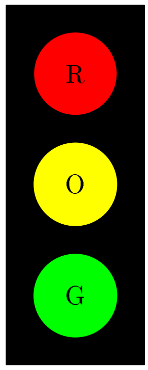


Abbildung 3.3.2: Das Bild einer Ampel. (Siehe auch Abbildung 1.1.8 und [FJ95].)

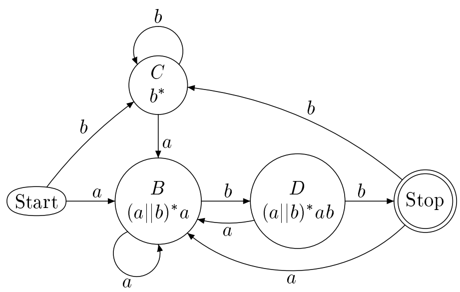
In manchen Fällen ist eine zweidimensionale Anordnung von Bildern erwünscht. Deshalb gibt es die Möglichkeit, Bilder in einem Raster auszurichten.

```
matrix :: [[Picture]] → Picture
```

Die Funktion *matrix* teilt die Ebene in Spalten und Reihen ein. Jede Spalte ist so breit, wie das breiteste Bild in ihr. Analog dazu ist jede Zeile so hoch, wie das höchste Bild in ihr. In den damit definierten rechteckigen Zellen wird jedes Bild zentriert plaziert. Wie auch bei den Funktionen *row* und *column* gibt es eine Variante, bei der sich ein zusätzlicher Abstand zwischen den Zellen horizontal wie vertikal angeben läßt.

```
matrixSepBy :: Numeric → Numeric → [[Picture]] → Picture
```

Wir können das bisher Gelernte benutzen, um in zwei Schritten das Bild eines endlichen Automaten zu konstruieren, wie es in Abbildung 52 in [Hob92] zu sehen ist. Die Bilder der Zustände Start und Stop kennen wir ja bereits aus Abschnitt 3.2. Die Funktion *stk* stellt in LaTeX zwei Zeilen übereinander.




```
stk     :: String → String → Picture
stk a b  = math ("\\matrix{" ++ a ++ "\\cr␣" ++ b ++ "\\cr}")
```

Die Bilder der Zustände können wir mit Hilfe von Rahmen erzeugen und mit der Funktion *matrixSepBy* zueinander kombinieren. Abbildung 3.3.3 zeigt das Resultat dieses ersten Schrittes. Was noch fehlt, sind die Pfeile zwischen den Zuständen des endlichen Automaten, aber das lernen wir in den nächsten Kapiteln.

```
automat   = matrixSepBy 30 20
              [[empty, circle (stk "C" "b^*")       #setName "C"],
               [oval (tex "Start")                  #setName "start",
                circle (stk "B" "(a||b)^*a")        #setName "B",
                circle (stk "D" "(a||b)^*ab")       #setName "D",
                circle (circle (tex "Stop"))        #setName "stop"]]
```

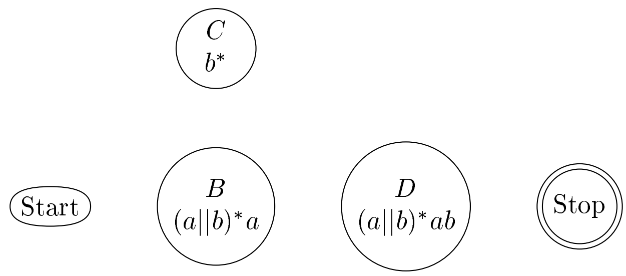
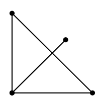


Abbildung 3.3.3: Erster Teil eines endlichen Automaten. Die Bilder der Zustände werden von der Funktion *matrixSepBy* ausgerichtet.

3.4 Pfade

Pfade sind neben Bildern die wichtigsten Objekte in *functional* METAPOST. Im Unterschied zu Bildern haben diese aber keine Bounding Box, sondern werden über Bilder gezeichnet. Ein Pfad besteht aus Punkten und Verbindungen zwischen diesen. Die Funktion *vec* bezeichnet den Punkt an den angegebenen Koordinaten. Der Pfadkonstruktor (--) verbindet die Punkte. Die einzelnen Punkte sind hier zur Deutlichkeit hervorgehoben.

vec $(20, 20)$ -- *vec* $(0, 0)$ -- *vec* $(0, 30)$
--*vec* $(30, 0)$ -- *vec* $(0, 0)$



Die Anwendung des Pfadkonstruktors (--) erzeugt ein sogenanntes Pfadsegment, das nicht unbedingt die Form einer Strecke haben muß. Es stehen vier verschiedene Pfadkonstruktoren zur Verfügung, mit denen sich unterschiedliche Formen der Pfadsegmente ergeben. [4]

[4] Die Berechnung gekrümmter Pfade geschieht mit Hilfe von BÉZIER splines. Seien $z_0, z_1, \ldots, z_n$ Punkte, dann gibt es Kontrollpunkte z_k^+ und z_{k-1}^- so daß der kubische spline zwischen den Punkten z_k und z_{k+1} definiert ist durch das BERNSTHEĬN Polynom

$$z(t) = B(z_k, z_k^+, z_{k+1}^-, z_{k+1}; t) = (1-t)^3 z_k + 3(1-t)^2 t z_k^+ + 3(1-t)t^2 z_{k+1}^- + t^3 z_{k+1}$$

für $0 \leqslant t \leqslant 1$. Siehe auch [Fel92] Kapitel 5.8.2. Wenn nicht anders angegeben, werden die Kontrollpunkte so gewählt, daß die Pfadsegmente an ihren Übergängen C^1 stetig sind.

(--) Eine gerade Verbindung.
(..) Eine gekrümmte Verbindung.
(---) Eine gerade Verbindung mit möglichst stetigen Enden.
(...) Eine gekrümmte Verbindung mit wenig Wendepunkten.

Der Verlauf von Pfaden ist aber nicht nur durch die Punkte, sondern auch wesentlich von den Verbindungen zwischen ihnen bestimmt. So sind die drei Pfade in Abbildung 3.4.1 unterschiedlich, obwohl sie durch die gleichen Punkte verlaufen. Aber auch die Pfadsegmente zwischen den Punkten z_0 und z_1 oder z_1 und z_2 unterscheiden sich in den drei Bildern obwohl die Punkte immer mit dem Pfadkonstruktor (..) verbunden sind.[5] Um einen geschlossenen, d.h. zyklischen Pfad zu konstruieren, kann man als letzten Punkt eines Pfades den Ausdruck $cycle$ angeben.

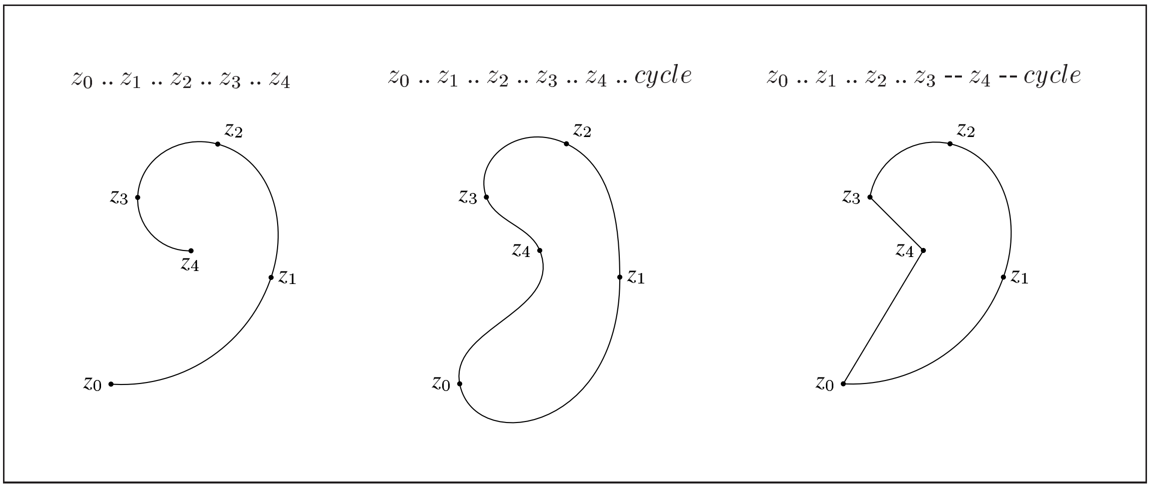


Abbildung 3.4.1: Die Wirkung verschiedener Pfadverbindungen auf die gleichen Punktmengen. (Siehe Abbildungen 3 und 4 in [Hob92])

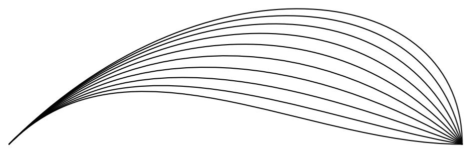
Pfade lassen sich mit Hilfe von Attributierungsfunktionen noch weiter in ihrem Erscheinungsbild beeinflussen. Dabei besitzt jedes Pfadsegment seine eigene Attributmenge! Für die kurvigen Verbindungstypen (..) und (...) verändern die Funktionen *setStartAngle* und *setEndAngle* den Winkel, in dem ein Pfadsegment einen Punkt verläßt bzw. betritt, wie die Abbildung 3.4.2 illustriert. Die Pfadkonstruktoren haben eine höhere Präzedenz, als der (#)–Operator, was bedeutet, daß der Ausdruck *a .. b .. c # setStartAngle d* zu lesen ist, wie (*a .. b .. c*) # *setStartAngle d*.
Eine andere Möglichkeit zur Beeinflussung der Kontrollpunkte der BÉZIER splines, aus denen Kurvensegmente bestehen, sind die Befehle *setStartVector* und *setEndVector*. Siehe dazu Abbildung 3.4.3, in der auch die Nützlichkeit der Pfadverbindung (...) deutlich wird, die im Gegensatz zu (..) Wendepunkte vermeidet.
Oft besteht der Wunsch, Pfade mit Beschriftungen zu versehen. Jedes Pfadsegment kann beliebig viele davon erhalten und es kann sich nicht nur um Text, sondern um ein beliebiges Bild handeln. Die Zuweisung einer Beschriftung an einen Pfad realisiert die Funktion

$$setLabel \quad :: \quad Numeric \to Dir \to Picture \to Path \to Path$$

Der erste Parameter gibt die Position auf dem Pfad an. Der Wert muß im Intervall $[0;1]$ liegen, wobei 0 für den Anfang und 1 für das Ende des Pfades steht. Der zweite Parameter gibt an, wie die

[5] Eine Eigenschaft, die aus der C^1 Stetigkeit zwischen den Segmenten und der Verwendung von BÉZIER splines folgt.

$$
\begin{array}{l}
[\mathit{vec}\ (0,0)\ ..\ \mathit{vec}\ (6 * \mathit{cm}, 0) \\
\quad \#\ \mathit{setStartAngle}\ 45 \\
\quad \#\ \mathit{setEndAngle}\ (-10 * a) \\
|\, a \leftarrow [0\,..\,9]]
\end{array}
$$



$$
\begin{array}{l}
[\mathit{vec}\ (0,0)\ ..\ \mathit{vec}\ (6 * \mathit{cm}, 0) \\
\quad \#\ \mathit{setStartAngle}\ 45 \\
\quad \#\ \mathit{setEndAngle}\ (10 * a) \\
|\, a \leftarrow [0\,..\,7]]
\end{array}
$$

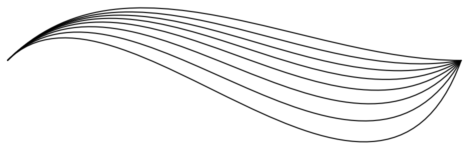


Abbildung 3.4.2: Start– und Endwinkel von Pfadsegmenten können vorgegeben werden. (Siehe Abbildungen 7 und 8 in [Hob92])

$$
\begin{array}{l}
(z_0\ ..\ z_1\ \#\ \mathit{setStartVector}\ (\mathit{vec}\ (0,1)) \\
\qquad\qquad \#\ \mathit{setEndVector}\ (\mathit{vec}\ (1,0))) \\
..z_2\ \#\ \mathit{setEndVector}\ (\mathit{vec}\ (0,-1))
\end{array}
$$

$$
\begin{array}{l}
(z_0\ ...\ z_1\ \#\ \mathit{setStartVector}\ (\mathit{vec}\ (0,1)) \\
\qquad\qquad \#\ \mathit{setEndVector}\ (\mathit{vec}\ (1,0))) \\
...z_2\ \#\ \mathit{setEndVector}\ (\mathit{vec}\ (0,-1))
\end{array}
$$

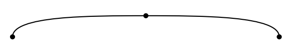


Abbildung 3.4.3: Der Unterschied zwischen den Pfadverbindungen (..) und (...). (Siehe Abbildung 9 in [Hob92])

Beschriftung zu dem Pfad ausgerichtet sein soll und der dritte Parameter ist das Bild der Beschriftung. Abbildung 3.4.4 zeigt ein Beispiel.

```
(vec (0,40) .. vec (10,50)
  .. vec (15,40) .. vec (10,7)
  .. vec (15,0) .. vec (25,7))
#setLabel 0 N (tex "start")
#setLabel 0 C dot
#setLabel 0.5 W (tex "middle")
#setLabel 0.5 C dot
#setLabel 1 W (tex "end")
#setLabel 1 C dot
```

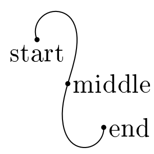


Abbildung 3.4.4: Pfade können beliebige Beschriftungen erhalten.

Das Konzept, das *functional* METAPOST zugrunde liegt, geht davon aus, daß Pfade über Bilder gezeichnet werden oder anders gesehen, ein bestehendes Bild mit Pfaden „dekoriert" wird. Wir bringen eine Reihe von Pfaden auf ein Bild auf, indem wir die Funktion

$$\textit{draw} \quad :: \quad [\textit{Path}] \rightarrow \textit{Picture} \rightarrow \textit{Picture}$$

auf dieses anwenden. Dieser Abschnitt zeigt einige Bilder von Pfaden. Wie sind diese erzeugt? Eine Möglichkeit, das Bild eines Pfades oder mehrere Pfade zu erhalten ist, diese über ein leeres Bild zu zeichnen. Das resultierende Bild hat dann allerdings die Bounding Box des Bildes *empty*, die keine Ausdehnung hat. Das Einzeichnen von Pfaden verändert die Bounding Box eines Bildes nämlich nicht. In diesem Fall soll das Bild des Pfades aber eine Bounding Box haben, die diesen umschließt. Zu diesem Zweck gibt es die Funktion

$$\textit{setTrueBoundingBox} \quad :: \quad \textit{Picture} \rightarrow \textit{Picture}$$

die einem Bild eine rechteckige Bounding Box gibt, die dieses minimal umschließt. Der folgende Ausdruck erzeugt ein Bild der Pfade *ps*.

$$\textit{setTrueBoundingBox}\ (\textit{draw ps empty})$$

In Abschnitt 3.16 lernen wir eine einfachere Schreibweise für diesen Ausdruck kennen.

3.5 Namen

Im vorigen Abschnitt haben wir Punkte, immer mit Hilfe der Funktion *vec* angegeben. Darüber hinaus gibt es aber die Möglichkeit, auf Punkte innerhalb eines Teilbildes Bezug zu nehmen. Jedes Bild hat neun Bezugspunkte vordefiniert, die die Grenzen seiner Bounding Box markieren. Diese Punkte haben die Namen *N*, *NE*, *E*, *SE*, *S*, *SW*, *W*, *NW*, entsprechend den Himmelsrichtungen und *C* für das Zentrum des Bildes. In Abbildung 3.5.1 sind die Bezugspunkte und die Bounding Box eines Bildes einmal beispielhaft eingezeichnet. Diese Bezugspunkte lassen sich mit Hilfe der Funktion

$$\begin{array}{l}\mathit{tex}\ \texttt{"let␣this␣be"}\\ \boxminus\ \mathit{tex}\ \texttt{"a␣picture"}\end{array}$$

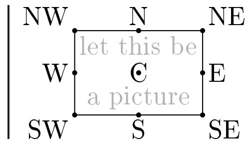


Abbildung 3.5.1: Jedes Bild hat neun Bezugspunkte.

$$\mathit{ref} \quad :: \quad \mathit{Name} \rightarrow \mathit{Point}$$

z.B. in Pfadbeschreibungen verwenden. Im folgendem Ausdruck benutzen wir die Bezugspunkte des Bildes *tex* "!", um einen Pfad zu zeichnen.

$$\begin{array}{l}\mathit{draw}\ [\mathit{ref}\ \mathit{NE} + \mathit{vec}\ (2,0)\ \text{--}\ \mathit{ref}\ S - \mathit{vec}\ (0,2)\\ \qquad \text{--}\ \mathit{ref}\ \mathit{NW} - \mathit{vec}\ (2,0)\ \text{--}\ \mathit{cycle}]\ (\mathit{tex}\ \texttt{"!"})\end{array}$$

In diesem Beispiel haben wir es mit einem einzelnen atomaren Bild zu tun. Oft möchten wir aber Bezugspunkte verschiedener Bilder referenzieren. Dies stellt ein Problem dar, denn die Bezugspunkte haben ja in jedem Bild den gleichen Namen. Wie sind sie trotzdem voneinander zu unterscheiden? Wir können dieses Problem lösen, indem wir den Bildern Namen geben und diese damit unterscheidbar machen. Dem Namen des gewünschten Bezugspunktes ist dann der Name des Bildes voranzustellen. Eine Attributierungsfunktion versieht ein Bild mit dem gewünschten Namen. Ein Bild kann auch mehrere Namen haben.

$$\mathit{setName} \quad :: \quad \mathit{Name} \rightarrow \mathit{Picture} \rightarrow \mathit{Picture}$$

Die Kombination verschiedener Namen erfolgt mit dem Konstruktor

$$(\triangleleft) \quad :: \quad \mathit{Name} \rightarrow \mathit{Name} \rightarrow \mathit{Name}$$

Wie in Abbildung 3.5.2 zu sehen, können wir als Namen Zeichenketten oder Integerzahlen verwenden. Warum dies funktioniert, besprechen wir in Abschnitt 3.16.

Hier sehen wir ein ganz besonderes Feature von Pfaden. Der Pfad verläuft zwar in Richtung der Bildzentren, erreicht diese aber nicht, sondern ist an den Bounding Boxen abgeschnitten. Dies ist die natürliche Intention, wenn wir Pfade zwischen Bildern zeichnen wollen. Der Pfadkonstruktor (--) wählt diese Funktionsweise automatisch, wenn er Bezugspunkte verbindet

$$\begin{array}{l}\mathit{draw}\ [\mathit{ref}\ (\texttt{"box1"} \triangleleft C)\ \text{--}\ \mathit{ref}\ (2 \triangleleft C)]\\ \quad ((\mathit{box}\ \texttt{"1"}\ \#\ \mathit{setName}\ \texttt{"box1"})\\ \qquad ⫴\ (\mathit{box}\ \texttt{"2"}\ \#\ \mathit{setName}\ (2 :: \mathit{Int})))\end{array}$$

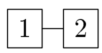


Abbildung 3.5.2: Ein Pfad zwischen zwei Bildern.

Natürlich läßt sich der Prozeß der Namensgebung auch auf Bilder anwenden, um sie dann zu größeren zu kombinieren und wieder zu benennen. Die Namen der Referenzen verlängern sich entsprechend. Dabei ist es problemlos möglich, aus der Kette der Namensbestandteile beliebige Teile wegzulassen, solange der Name noch eindeutig bleibt. Im Bild

$$(\mathit{tex}\ \texttt{"a"} \# \mathit{setName}\ \texttt{"a"})\ \square\square\ (\mathit{tex}\ \texttt{"b"} \# \mathit{setName}\ \texttt{"b"})$$
$$\#\mathit{setName}\ \texttt{"ab"}$$

bezeichnet der Ausdruck $\mathit{ref}\ (\texttt{"ab"} \triangleleft \texttt{"a"} \triangleleft C)$ das Zentrum des Bildes $\mathit{tex}\ \texttt{"a"}$, aber der Ausdruck $\mathit{ref}\ (\texttt{"a"} \triangleleft C)$ tut dies ebenso, weil nur ein Bild mit dem Namen "a" vorkommt. Die Referenz $\mathit{ref}\ (\texttt{"ab"} \triangleleft C)$ ist erlaubt, aber nicht mehr eindeutig. Mit welchen Regeln Doppeldeutigkeiten aufgelöst werden, behandelt Abschnitt 3.17.

Wir können nun den endlichen Automaten aus 3.3 vervollständigen. Für die Schleifen definieren wir allgemeine Funktionen, die nur den Namen eines Bildes benötigen und die Größe der Schleife an die Breite des Bildes anpassen.

$$\begin{array}{lcl}
\mathit{loopN} & :: & \mathit{Name} \to \mathit{Path} \\
\mathit{loopN}\ s & = & \mathit{ref}\ (s \triangleleft \mathit{NE}) \mathinner{..} \mathit{arrow}\ (\mathit{ref}\ (s \triangleleft N) + \mathit{vec}\ (0, 0.5 * \mathit{width}\ s)) \\
 & & \qquad\qquad\qquad (\mathit{ref}\ (s \triangleleft \mathit{NW}))
\end{array}$$

$$\begin{array}{lcl}
\mathit{loopSW} & :: & \mathit{Name} \to \mathit{Path} \\
\mathit{loopSW}\ s & = & \mathit{ref}\ (s \triangleleft \mathit{SW}) \mathinner{..} \mathit{arrow}\ (\mathit{ref}\ (s \triangleleft S) - \mathit{vec}\ (0.353 * \mathit{width}\ s, \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad 0.353 * \mathit{width}\ s)) \\
 & & \qquad\qquad\qquad (\mathit{ref}\ (s \triangleleft S))
\end{array}$$

Wir greifen hier schon einmal auf das Kapitel 3.11 vor, in dem wir Pfeile kennenlernen werden.

$$\begin{array}{lcl}
\mathit{arrow} & :: & \mathit{Point} \to \mathit{Point} \to \mathit{Path} \\
\mathit{arrow}\ a\ b & = & a \mathinner{..} b \# \mathit{setArrowHead}\ default
\end{array}$$

Die Funktion *width*, die die Breite eines Bildes zurückliefert, läßt sich so definieren:

$$\begin{array}{lcl}
\mathit{width} & :: & \mathit{String} \to \mathit{Numeric} \\
\mathit{width}\ s & = & \mathit{xpart}\ (\mathit{ref}\ (s \triangleleft E)) - \mathit{xpart}\ (\mathit{ref}\ (s \triangleleft W))
\end{array}$$

Dabei ist vorausgesetzt, daß der Punkt *W* eines Bildes immer die linkeste und *E* die rechteste Position markiert. Schließlich fehlt noch eine Funktion für Pfeile mit Beschriftungen.

$$\begin{array}{lcl}
\mathit{to} & :: & \mathit{Name} \to \mathit{Numeric} \to \mathit{Name} \to \mathit{String} \to \mathit{Dir} \to \mathit{Path} \\
\mathit{to}\ a\ \mathit{sa}\ b\ l\ d & = & \mathit{arrow}\ (\mathit{ref}\ (a \triangleleft C))\ (\mathit{ref}\ (b \triangleleft C)) \\
 & & \#\mathit{setStartAngle}\ \mathit{sa} \\
 & & \#\mathit{setLabel}\ 0.5\ d\ (\mathit{math}\ l)
\end{array}$$

Wir haben das Bild des endlichen Automaten in zwei Schritten konstruiert. Dies taten wir nicht nur aus didaktischen Gründen, sondern es ist für die Arbeit mit *functional* METAPOST typisch, erst verschiedene Bilder zu kombinieren und dann in dieses neue Bild Linien und Flächen zu zeichnen.

Wenn wir den endlichen Automaten mit einem Stift zeichnen sollten, würden wir wahrscheinlich zuerst einen Knotentext zeichnen und diesen dann umrahmen. Bei den weiteren Knoten gingen wir entsprechend vor. Die Pfeile würden wir normalerweise genau wir bei *functional* METAPOST erst nach den Knoten einzeichnen.

```
draw [loopN "C"
        # setLabel 0.5 S (math "b"),
        loopSW "B"
        # setLabel 0.5 N (math "a"),
        to "start" 0 "B" "a" S,
        to "C" (−90) "B" "a" W,
        to "B" 0 "D" "b" S,
        to "D" 0 "stop" "b" S,
        to "start" 55 "C" "b" SE,
        to "stop" 135 "C" "b" SW,
        to "stop" (−125) "B" "a" N,
        to "D" (−145) "B" "a" N
        ] automat
```

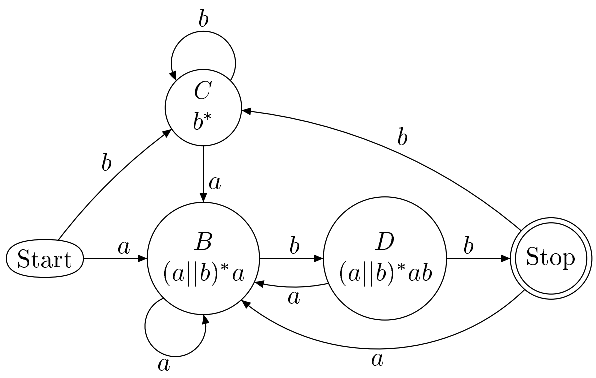


Abbildung 3.5.3: Zweiter Teil des endlichen Automaten aus Abbildung 3.3.3. Die Zustandsübergänge sind nun eingezeichnet.

3.6 Zahlen und Punkte

In *functional* METAPOST haben numerische Werte den Typ *Numeric* und Punkte den Typ *Point*. Dabei sind auf diesen Typen vielfältige Operationen definiert. In den Pfaden von Kapitel 3.4 haben wir schon die Funktion

$$\textit{vec} \quad :: \quad (\textit{Numeric}, \textit{Numeric}) \rightarrow \textit{Point}$$

verwendet, die den Punkt an den angegebenen Koordinaten zurückliefert. Umgekehrt lassen sich aus einem Punkt die Anteile der x– und der y–Dimension gewinnen.

$$\begin{aligned} &\textit{xpart} &&:: \quad \textit{Point} \rightarrow \textit{Numeric} \\ &\textit{ypart} &&:: \quad \textit{Point} \rightarrow \textit{Numeric} \end{aligned}$$

Der Typ *Point* ist Instanz der Klassen *Num*, was die Grundrechenarten $(+)$, $(-)$ und $(*)$ auf diesen anwendbar macht. Für *Numeric* sind Instanzen der Klassen *Num*, *Fractional*, *Floating* und *Enum* definiert, womit neben den vier Grundrechenarten auch Funktionen wie *sin*, *tan*, *sqrt* oder *exp* definiert sind. Weil *Numeric* auch Instanz der Klasse *Enum* ist, sind auch arithmetische Folgen über *Numeric* möglich.

Die Multiplikation einer Zahl mit einem Punkt ermöglicht der Operator $(\dot{*})$. Darüber hinaus gibt es noch weitere Funktionen, die die geometrische Konstruktion von Punkten vereinfachen. Wird die Polarkoordinatendarstellung eines Punktes benötigt, ermittelt *angle* den zugehörigen Winkel und *dist* $(\textit{vec}\ (0,0))$ den Abstand des Punktes zum Ursprung.

$$\begin{aligned} &\textit{angle} &&:: \quad \textit{Point} \rightarrow \textit{Numeric} \\ &\textit{dist} &&:: \quad \textit{Point} \rightarrow \textit{Point} \rightarrow \textit{Numeric} \end{aligned}$$

Oft liegt ein Punkt auf der Strecke zwischen zwei anderen. Die Funktion *med* [6] (mediate) drückt diese Beziehung aus. Der erste Parameter bestimmt, wo auf dieser Strecke der Punkt liegen soll. Z.B.

[6] Die Funktion *med* entspricht einem BERNSTHEĬN Polynom ersten Grades:

$$\textit{med}\ t\ z_1\ z_2 = B(z_1, z_2; t) = (1-t)z_1 + tz_2$$

beschreibt *med* $\frac{1}{3}$ z_1 z_2 den Punkt, der auf dem Drittel der Strecke von Punkt z_1 nach z_2 liegt. *med* ist in analoger Weise auch auf Zahlen anwendbar.

$$
\begin{array}{lll}
\mathit{med} & :: & \mathit{Numeric} \to \mathit{Point} \to \mathit{Point} \to \mathit{Point} \\
\mathit{med} & :: & \mathit{Numeric} \to \mathit{Numeric} \to \mathit{Numeric} \to \mathit{Numeric}
\end{array}
$$

Den größten bzw. kleinsten Wert aus einer Liste von Zahlen ermitteln die Funktionen

$$
\begin{array}{lll}
\mathit{maximum}' & :: & [\mathit{Numeric}] \to \mathit{Numeric} \\
\mathit{minimum}' & :: & [\mathit{Numeric}] \to \mathit{Numeric}
\end{array}
$$

Angenommen wir wollen einen Kreis um einen Punkt o zeichnen, der drei Punkte p_1, p_2 und p_3 umschließt und einen möglichst minimalen Radius hat. Dieser Radius läßt sich wie folgt ermitteln:

$$
\begin{array}{lll}
\mathit{radius} & :: & \mathit{Numeric} \\
\mathit{radius} & = & \mathit{maximum}'\ [\mathit{dist}\ o\ p_1, \mathit{dist}\ o\ p_2, \mathit{dist}\ o\ p_3]
\end{array}
$$

Aus den Funktionen für Punkte und Zahlen können wir noch folgende nützliche Funktionen ableiten.

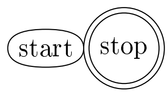
$$
\begin{array}{lll}
\mathit{dir} & :: & \mathit{Numeric} \to \mathit{Point} \\
\mathit{dir}\ a & = & \mathit{vec}\ (\mathit{cos}\ a, \mathit{sin}\ a) \\
\\
\mathit{xy} & :: & \mathit{Point} \to \mathit{Point} \to \mathit{Point} \\
\mathit{xy}\ p_1\ p_2 & = & \mathit{vec}\ (\mathit{xpart}\ p_1, \mathit{ypart}\ p_2) \\
\\
\mathit{xdist} & :: & \mathit{Point} \to \mathit{Point} \to \mathit{Numeric} \\
\mathit{xdist}\ p_1\ p_2 & = & \mathit{xpart}\ p_1 - \mathit{xpart}\ p_2 \\
\\
\mathit{ydist} & :: & \mathit{Point} \to \mathit{Point} \to \mathit{Numeric} \\
\mathit{ydist}\ p_1\ p_2 & = & \mathit{ypart}\ p_1 - \mathit{ypart}\ p_2
\end{array}
$$

Mit den Funktionen die wir hier kennengelernt haben, lassen sich Ausdrücke für zusammengesetzte Punkte und Zahlen formulieren. Im nächsten Absatz werden wir einen Schritt weiter gehen und Gleichungen formulieren.

3.7 Symbolische Gleichungen

Ein Konzept, welches *functional* METAPOST von anderen Beschreibungssprachen hervorhebt, ist die Beschreibung von Bildern mit Hilfe symbolischer Gleichungen. Gleichungen können dazu dienen, das Layout von Bildern festzulegen oder geometrische Zusammenhänge zu beschreiben.

$$
\begin{array}{l}
\textbf{let}\ \mathit{beside}\ a\ b = \mathit{overlay}\ [\mathit{ref}\ (0 \triangleleft E) \doteq \mathit{ref}\ (1 \triangleleft W)] \\
\qquad\qquad\qquad\qquad\qquad [a, b] \\
\textbf{in}\ \mathit{beside}\ (\mathit{oval}\ \texttt{"start"})\ (\mathit{circle}\ (\mathit{circle}\ \texttt{"stop"}))
\end{array}
$$



Dieses Bild ist uns schon aus Abschnitt 3.3 bekannt. Die Funktion *beside* entspricht dem Kombinator $(\llbracket)$. Wir können mit Hilfe der Funktion

$$
\begin{array}{lll}
\mathit{overlay} & :: & [\mathit{Equation}] \to [\mathit{Picture}] \to \mathit{Picture}
\end{array}
$$

Siehe dazu auch die Fußnote 4 dieses Kapitels.

alle möglichen Layouts mehrerer Bilder definieren. Dabei referenzieren wir Variablen und Bezugspunkte aus dem $n+1$-ten Bild, indem wir ihren Namen n voranstellen, d.h. das erste Bild hat den Namen 0, das zweite den Namen 1 ... das $n+1$-te den Namen n. So legen wir fest, daß sich der Bezugspunkt E des ersten Bildes an der gleichen Position befindet, wie Bezugspunkt W des zweiten Bildes. Damit ist das Layout zwischen beiden Bildern beschrieben.
Als weiteres Beispiel betrachten wir eine andere Implementierung der Funktion *rowSepBy*, als die in Abschnitt 3.3 vorgeschlagene .

$$\begin{array}{lll} \mathit{rowSepBy} & :: & \mathit{Numeric} \rightarrow [\mathit{Picture}] \rightarrow \mathit{Picture} \\ \mathit{rowSepBy}\ \mathit{hSep}\ \mathit{ps} & = & \mathit{overlay}\ [\mathit{ref}\ (i \triangleleft E) + \mathit{vec}\ (\mathit{hSep}, 0) \doteq \mathit{ref}\ (i+1 \triangleleft W) \\ & & \qquad |\, i \leftarrow [0\, . .\, \mathit{length}\ \mathit{ps} - 2]] \\ & & \qquad \mathit{ps} \end{array}$$

In ähnlicher Weise sind alle Bildkombinatoren definiert. Beim Aufstellen des Gleichungssystems ist nur zu beachten, daß für jedes Bild die relative Position zu den anderen eindeutig bestimmt ist.
In den bisherigen Beispielen haben wir nur die Gleichheit mehrerer Bezugspunkte ausgedrückt. Es besteht aber auch die Möglichkeit, eigene Punktvariablen und auch numerische Variablen zu definieren. Der Ausdruck

$$\mathit{ref}\ \texttt{"point"} :: \mathit{Point}$$

steht für die Punktvariable mit dem Namen `"point"`, wogegen

$$\mathit{var}\ \texttt{"number"} :: \mathit{Numeric}$$

eine Zahlvariable mit dem Namen `"number"` bezeichnet. Es gibt eine klare Trennung zwischen Punkt– und Zahlenvariablen. Solche Variablen fungieren als Unbekannte, für die sich mit Hilfe von Gleichungssystemen Werte herleiten lassen.
Wenn wir z.B. eine Punktvariable mit dem Namen `"target"` definieren wollen, die von einem Punkt `"point"` in der Richtung `"angle"`, `"distance"` Punkte entfernt liegt, beschreibt dies die Gleichung

$$\mathit{ref}\ \texttt{"target"} \doteq \mathit{ref}\ \texttt{"point"} + \mathit{var}\ \texttt{"distance"} \mathbin{\dot{*}} \mathit{dir}\ (\mathit{var}\ \texttt{"angle"})$$

Ein Vorkommen einer Variablen mit zusammengesetztem Namen referenziert eine Punkt– oder eine Zahlvariable. Aber wie lassen sich neue Variablen definieren? Wenn eine Variable mit normalem, d.h. nicht zusammengesetzten Namen das erste Mal in einem Gleichungssystem verwendet wird, entsteht eine neue Variable, d.h. die Variable hat ihr definierendes Vorkommen. Alle späteren Vorkommen dieser Variable in dem Gleichungssystem sind angewandte Vorkommen.
Wir werden dazu im weiteren Verlauf noch einige Beispiele sehen, doch zunächst schauen wir uns den Typ der Gleichungen genauer an. Eine Gleichung hat den Typ *Equation* und wird mit dem Operator

$$(\doteq) \quad :: \quad a \rightarrow a \rightarrow \mathit{Equation}$$

erzeugt. Die Typvariable *a* steht für *Numeric* oder *Point*, denn eine Gleichheit kann entweder zwischen numerischen Werten oder zwischen Punkten bestehen. Zusätzlich lassen sich mit

$$\mathit{equal} \quad :: \quad [a] \rightarrow \mathit{Equation}$$

auch Mehrfachgleichungen[7] der Form $x_1 \doteq x_2 \doteq \ldots \doteq x_n$ definieren. Die Gültigkeit von Gleichungen läßt sich von logischen Bedingungen abhängig machen.

[7] An dieser Stelle kann man die Typ-Homogenität der Listen, die an anderer Stelle eher störend ist, begrüßen. Sie sorgt dafür, daß alle Argumente der Funktion *equal* den gleichen Typ haben, sonst tritt schon während der Übersetzung ein Fehler auf.

$$cond \quad :: \quad Boolean \rightarrow a \rightarrow a \rightarrow a$$

Dabei ergibt die Anwendung folgender Vergleichsoperatoren auf Punkte bzw. Zahlen einen Ausdruck vom Typ *Boolean*.

$$\begin{array}{lll} (\doteq) & :: & a \rightarrow a \rightarrow Boolean \\ (\dot{\not\equiv}) & :: & a \rightarrow a \rightarrow Boolean \\ (\dot{<}) & :: & a \rightarrow a \rightarrow Boolean \\ (\dot{\leqslant}) & :: & a \rightarrow a \rightarrow Boolean \end{array}$$

Auf dem Typ *Boolean* ist die gewöhnliche BOOLESCHE Algebra implementiert, wo $(*)$ die Funktion Und, $(+)$ die Funktion Oder und *negate* die Funktion Not repräsentieren.
Wenn eine Zahl oder ein Punkt variabel sein soll, sein Name aber unbedeutend ist, weil er an keiner weiteren Stelle vorkommt, kann der Ausdruck *whatever* verwendet werden[8]. Die Gleichung

$$ref\ z_1 \doteq med\ whatever\ (ref\ z_2)\ (ref\ z_3)$$

besagt, daß der Punkt z_1 irgendwo auf der Geraden durch die Punkte z_2 und z_3 liegt.
Neben der Beschreibung des Layouts mehrerer Bilder können wir Gleichungssysteme auch dafür verwenden, um andere geometrische Zusammenhänge mit ihnen zu beschreiben. Die in einem solchen Gleichungssystem berechneten Variablen können in einem Bild dann weiterverwendet werden. Dies alles ermöglicht die Funktion

$$define \quad :: \quad [Equation] \rightarrow Picture \rightarrow Picture$$

Die Gültigkeit der definierten Variablen entspricht der des Befehls **let** .. **in**.. von Haskell. Diese Funktion ist auch auf Pfade und Flächen anwendbar.

$$\begin{array}{lll} define & :: & [Equation] \rightarrow Path \rightarrow Path \\ define & :: & [Equation] \rightarrow Area \rightarrow Area \end{array}$$

Um das Vorgehen etwas klarer zu machen, wollen wir die Funktion *define* einmal beispielhaft anwenden. Gegeben sei folgendes Problem: Zeichne zu drei Punkten den Kreis, auf dem alle Punkte liegen. Dies läßt sich in die beiden folgenden Aufgaben aufteilen: Finde den Mittelpunkt des Umkreises und errechne dann den Radius des Umkreises. Der Mittelpunkt des Umkreises ergibt sich bei Dreiecken aus dem Schnittpunkt der Mittelsenkrechten. Wir konstruieren den Ausdruck schrittweise.
Gegeben seien die Punkte mit den Namen `"p1"`, `"p2"` und `"p3"`.

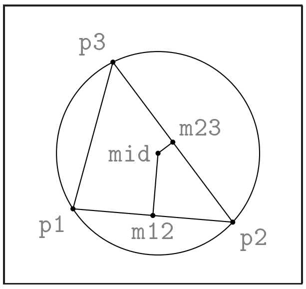


Abbildung 3.7.1: Konstruktion eines Umkreises.

$$\begin{array}{lll} points3 & :: & [Equation] \\ points3 & = & [ref\ \texttt{"p1"} \doteq vec\ (0,5), \\ & & \ ref\ \texttt{"p2"} \doteq vec\ (60,0), \\ & & \ ref\ \texttt{"p3"} \doteq vec\ (15,60)] \end{array}$$

Nun konstruieren wir die Punkte, auf denen die Mittelsenkrechten der Strecken $\overline{\texttt{p1 p2}}$ und $\overline{\texttt{p2 p3}}$.

$$\begin{array}{lll} meds3 & = & [ref\ \texttt{"m12"} \doteq med\ 0.5\ (ref\ \texttt{"p1"})\ (ref\ \texttt{"p2"}), \\ & & \ ref\ \texttt{"m23"} \doteq med\ 0.5\ (ref\ \texttt{"p2"})\ (ref\ \texttt{"p3"})] \end{array}$$

[8] Die Variable Whatever hat Ähnlichkeit mit dem Parameter *fill* in TEX, der wenn er mehr als einmal in einer Zeile auftritt, an allen Stellen den gleichen Wert annimmt.

Der Winkel zwischen den Mittelsenkrechten und den entsprechenden Strecken beträgt nach Definition 90 Grad.

$$\begin{aligned} \mathit{angles3} \quad &= [\mathit{var}\ \texttt{"a12"} \doteq 90 + \mathit{angle}\ (\mathit{ref}\ \texttt{"p1"} - \mathit{ref}\ \texttt{"p2"}), \\ &\quad\ \mathit{var}\ \texttt{"a23"} \doteq 90 + \mathit{angle}\ (\mathit{ref}\ \texttt{"p2"} - \mathit{ref}\ \texttt{"p3"})] \end{aligned}$$

Der Kreismittelpunkt *ref* `"mid"` liegt irgendwo auf der Geraden durch *ref* `"m12"` mit dem Winkel *var* `"a12"` und gleichzeitig auf der Graden durch *ref* `"m23"` mit dem Winkel *var* `"a23"`.

$$\begin{aligned} \mathit{mid3} \quad &= [\mathit{equal}\ [\mathit{ref}\ \texttt{"mid"}, \\ &\qquad\qquad \mathit{med}\ \mathit{whatever}\ (\mathit{ref}\ \texttt{"m12"}) \\ &\qquad\qquad\qquad\qquad (\mathit{ref}\ \texttt{"m12"} + \mathit{dir}\ (\mathit{var}\ \texttt{"a12"})), \\ &\qquad\qquad \mathit{med}\ \mathit{whatever}\ (\mathit{ref}\ \texttt{"m23"}) \\ &\qquad\qquad\qquad\qquad (\mathit{ref}\ \texttt{"m23"} + \mathit{dir}\ (\mathit{var}\ \texttt{"a23"}))]] \end{aligned}$$

Alle Punkte haben den gleichen Abstand zum Mittelpunkt, d.h. der Radius des Umkreises entspricht der Entfernung des Mittelpunktes zum ersten Punkt.

$$r_3 \quad = [\mathit{var}\ \texttt{"r"} \doteq \mathit{dist}\ (\mathit{ref}\ \texttt{"mid"})\ (\mathit{ref}\ \texttt{"p1"})]$$

Nachdem wir den Mittelpunkt und den Radius des Umkreises mit Hilfe von Gleichungen beschreiben haben, können wir mit beiden Werten den Umkreis zeichnen. Zum besseren Verständnis zeichnen wir zusätzlich das durch die Punkte gebildete Dreieck und die beiden Mittelsenkrechten ein. Der Operator $(\&)$ konkateniert die Gleichungslisten effizient.

$$\begin{aligned} &\mathit{define}\ (\mathit{points3}\ \&\ \mathit{meds3}\ \&\ \mathit{angles3}\ \&\ \mathit{mid3}\ \&\ r_3) \\ &\quad (\mathit{draw}\ [\mathit{ref}\ \texttt{"mid"} + \mathit{vec}\ (0, \mathit{var}\ \texttt{"r"})\ ..\ \mathit{ref}\ \texttt{"mid"} + \mathit{vec}\ (\mathit{var}\ \texttt{"r"}, 0) \\ &\qquad\quad ..\ \mathit{ref}\ \texttt{"mid"} + \mathit{vec}\ (0, -\mathit{var}\ \texttt{"r"})\ ..\ \mathit{ref}\ \texttt{"mid"} + \mathit{vec}\ (-\mathit{var}\ \texttt{"r"}, 0) \\ &\qquad\quad ..\ cycle, \\ &\qquad\quad \mathit{ref}\ \texttt{"p1"}\ \text{--}\ \mathit{ref}\ \texttt{"p2"}\ \text{--}\ \mathit{ref}\ \texttt{"p3"}\ \text{--}\ cycle, \\ &\qquad\quad \mathit{ref}\ \texttt{"m12"}\ \text{--}\ \mathit{ref}\ \texttt{"mid"}, \\ &\qquad\quad \mathit{ref}\ \texttt{"m23"}\ \text{--}\ \mathit{ref}\ \texttt{"mid"}]\ \mathit{empty}) \end{aligned}$$

3.8 Farben

Jedes sichtbare Objekt in *functional* METAPOST kann eine Farbe erhalten. Die Voreinstellung der Farbattribute ist dabei *black*. Als Farbraum kommt das RGB–Modell zur Anwendung, bei dem eine Farbe durch additives Mischen der Grundfarben rot, grün und blau beschrieben wird.

$$\mathit{color} \quad :: \mathit{Double} \to \mathit{Double} \to \mathit{Double} \to \mathit{Color}$$

Es sind schon Ausdrücke für häufig benötigte Farben vordefiniert.

$$\begin{aligned} \mathit{white} \quad &= \mathit{color}\ 1\ 1\ 1 \\ \mathit{black} \quad &= \mathit{color}\ 0\ 0\ 0 \\ \mathit{red} \quad &= \mathit{color}\ 1\ 0\ 0 \\ \mathit{green} \quad &= \mathit{color}\ 0\ 1\ 0 \\ \mathit{blue} \quad &= \mathit{color}\ 0\ 0\ 1 \\ \mathit{yellow} \quad &= \mathit{color}\ 1\ 1\ 0 \\ \mathit{cyan} \quad &= \mathit{color}\ 0\ 1\ 1 \\ \mathit{magenta} \quad &= \mathit{color}\ 1\ 0\ 1 \\ \mathit{grey}\ n \quad &= \mathit{color}\ n\ n\ n \end{aligned}$$

Ferner ist der Typ *Color* eine Instanz von *Num* und *Fractional*, was bedeutet, daß sich Farben auch addieren, subtrahieren sowie multiplizieren lassen. Diese Operationen erfolgen komponentenweise auf den roten, grünen bzw. blauen Anteilen der Farben. Der Ausdruck $red + green$ bezeichnet also die Farbe *yellow* und $cyan - blue$ bezeichnet die Farbe *green*. Weil der Typ *Color* auch die Funktion *fromRational* implementiert, ist 0.5 eine Abkürzung für grey 0.5, d.h. *color* 0.5 0.5 0.5. Die Werte der Farbanteile sollten sich im Intervall $[0;1]$ bewegen. Jeder Farbwert außerhalb dieses Intervalls wird als 0 bzw. 1 interpretiert.
Um in einem Bild den Farbwert zu ändern, existiert die Funktion

$$setColor \quad :: \quad Color \to Picture \to Picture$$

Die Attributierungsfunktion zum Setzen der Hintergrundfarbe ist

$$setBGColor \quad :: \quad Color \to Picture \to Picture$$

```
tex "colors"
#setColor green
#setBGColor 0.2
```

Abbildung 3.8.1: Attributierung zur Farbwahl

Abbildung 3.8.1 verdeutlicht, wie mit Hilfe der Attributierungsfunktionen die Farbattribute des Textbildes verändert werden.

3.9 Strichmuster

Pfade und Rahmen lassen sich mit verschiedenen Strichmustern zeichnen. Die Länge der Pfadstücke, die gezeichnet bzw. nicht gezeichnet werden, können in einer Liste frei definiert werden. [9]

$$\begin{aligned} &dashPattern &&:: [Double] \to Pattern \\ &dashPattern' &&:: [Double] \to Pattern \end{aligned}$$

Bei der Funktion *dashPattern* gibt der erste Zahlenwert die Länge des Pfades an, die gezeichnet wird, der zweite Wert die Länge des Pfades die nicht gezeichnet wird usw.. Wenn der Pfad länger ist, als das Muster, wiederholt sich dieses. Der Befehl $dashPattern'$ ist wie *dashPattern*, nur daß der erste Wert die Länge angibt, die nicht zu zeichnen ist.
Die Muster für gestrichelte und gepunktete Pfade sind bereits vordefiniert.

$$\begin{aligned} &dashed &&:: Pattern \\ &dashed &&= dashPattern\ [3,3] \end{aligned}$$

$$\begin{aligned} &dotted &&:: Pattern \\ &dotted &&= dashPattern'\ [2.5,0,2.5] \end{aligned}$$

[9] Der Komplexität des Strichmusters sind leider Grenzen gesetzt da PostScript nur eine Liste von maximal elf Werten erlaubt.

Einem Pfadsegment oder einem Rahmen weist man ein Muster mit der Attributierungsfunktion

$$setPattern \quad :: \quad Pattern \to Path \to Path$$

zu, wie Abbildung 3.10.1 demonstriert.

3.10 Zeichenstifte

Ein weiteres Attribut von Pfaden und Rahmen sind die verwendeten Zeichenstifte. Es gibt zwei Sorten: rechteckige und ovale; beide eventuell rotiert.

$$\begin{array}{lll} penSquare & :: & (Numeric, Numeric) \to Numeric \to Pen \\ penCircle & :: & (Numeric, Numeric) \to Numeric \to Pen \end{array}$$

Der erste Parameter bestimmt die Ausmaße, der zweite den Rotationswinkel. Damit lassen sich kalligraphische Effekte erreichen.

$$\begin{array}{l} \textbf{let}\ path = vec\ (0,0)\ ..\ vec\ (10,10) \\ \qquad\qquad ..\ vec\ (30,-10)\ ..\ vec\ (40,0) \\ \textbf{in}\ column\ [path\ \#\ setPattern\ dotted, \\ \qquad\qquad path\ \#\ setPattern\ dashed, \\ \qquad\qquad path\ \#\ setPattern\ (dashPattern\ [0,2,3,2]) \\ \qquad\qquad\qquad \#\ setPen\ 1, \\ \qquad\qquad path\ \#\ setPen\ (penCircle\ (0.1,5)\ 20), \\ \qquad\qquad path\ \#\ setPen\ (penSquare\ (5,5)\ 45), \\ \qquad\qquad path\ \#\ setPen\ 5] \end{array}$$

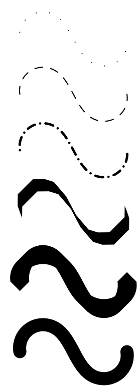


Abbildung 3.10.1: Strichmuster und kalligraphische Effekte

Wesentlich öfter dürfte aber Bedarf an einem runden Zeichenstift bestimmter Größe bestehen. Der Befehl *setPen* 1.5 wählt z.B. einen runden Stift mit einem Durchmesser von 1.5 PostScript Punkten. Siehe auch Abbildung 3.10.1.

3.11 Pfeile

Pfadsegmente könne an ihren beiden Enden Pfeilspitzen besitzen. Neben dem gewöhnlichen Pfeil besteht die Möglichkeit, die Länge der Spitze und den Spreizwinkel festzulegen. Abbildung 3.11.1 verdeutlicht das.

$$\begin{array}{lll} default & :: & PathArrowHead \\ arrowHeadSize & :: & Double \to Double \to PathArrowHead \end{array}$$

Eine große Pfeilspitze beschreibt der Ausdruck

$$\begin{array}{lll} arrowHeadBig & :: & PathArrowHead \\ arrowHeadBig & = & pathArrowHeadSize\ 8\ 4 \end{array}$$

Eine Pfeilspitze ist ein Objekt, das in zwei verschiedenen Stilarten gezeichnet werden kann. Ein gefülltes Dreieck oder eine Spitze aus Linien.

$$\begin{array}{ll} \textit{ahFilled} & :: \textit{ArrowHeadStyle} \\ \textit{ahLine} & :: \textit{ArrowHeadStyle} \end{array}$$

Eine Anwendung der folgenden Attributierungsfunktionen auf eine Pfeilspitze ändert, bzw. ermittelt dessen Stil.

$$\begin{array}{ll} \textit{setArrowHeadStyle} & :: \textit{ArrowHeadStyle} \rightarrow \textit{ArrowHead} \rightarrow \textit{ArrowHead} \\ \textit{getArrowHeadStyle} & :: \textit{ArrowHead} \rightarrow \textit{ArrowHeadStyle} \end{array}$$

Die folgenden Funktionen setzen Pfeilspitzen am Ende oder Anfang eines Pfadsegmentes, bzw. liefern eventuell eine Pfeilspitze.

$$\begin{array}{ll} \textit{setArrowHead} & :: \textit{ArrowHead} \rightarrow a \rightarrow a \\ \textit{setStartArrowHead} & :: \textit{ArrowHead} \rightarrow a \rightarrow a \end{array}$$

$$\begin{array}{ll} \textit{getArrowHead} & :: a \rightarrow \textit{Maybe ArrowHead} \\ \textit{getStartArrowHead} & :: a \rightarrow \textit{Maybe ArrowHead} \end{array}$$

Weil einfache Pfeile öfters benötigt werden, gibt es eine Funktion für diese.

$$\textit{arrow a b} \quad = a\,..\,b\;\#\;\textit{setArrowHead}\; default$$

```
let {ar2 a = setArrowHead a ∘ setStartArrowHead a;
     lineStyle = setArrowHeadStyle ahLine;
     fs = [id,
           ar2 defaultArrowHead,
           ar2 (arrowHeadSize 10 20),
           ar2 (arrowHeadSize 5 250),
           ar2 (defaultArrowHead # lineStyle),
           ar2 (arrowHeadSize 10 20 # lineStyle),
           ar2 (arrowHeadSize 5 180 # lineStyle),
           ar2 (arrowHeadSize 5 250 # lineStyle)]; }
in [f (vec (0,y) -- vec (40,y))
   | (y,f) ← zip [0,−16..] fs]
```

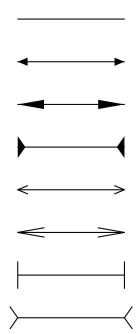


Abbildung 3.11.1: Verschiedene Pfeilvarianten. Man vergleiche den vierten und den achten Pfeil. Wenn ein Pfeil mit einem Spreizwinkel größer als 180 Grad im gefüllten Stil gezeichnet wird, verschiebt sich die Spitze von den Pfadenden weg, damit diese sichtbar bleiben. Ist der Pfeil jedoch im Linienstil, so liegt die Spitze an den Pfadenden.

3.12 Flächen

Neben dem Zeichnen von Linien benötigt eine universelle Bildbeschreibungssprache auch eine Möglichkeit Flächen auszufüllen. Flächen sind wie Pfade eigenständige Objekte mit Attributen wie Zeichenstift, Farbe und Zeichenreihenfolge. Der Unterschied zu Pfaden besteht darin, daß die Farb– und Stiftattribute für die ganze Fläche nur einmal vorhanden sind und nicht für jedes Pfadsegment einzeln. Es liegt nahe, die Pfadsyntax auch für die Beschreibung von Flächenumrissen einzusetzen. Dazu existiert eine Funktion, die einen zyklischen Pfad in eine Fläche überführt.

$$toArea \quad :: \quad Path \to Area$$

Auf das Flächenobjekt lassen sich die Attributierungsfunktionen für Farb– und Stiftwahl anwenden. Wie oben erwähnt, ist auch die Zeichenreihenfolge wählbar. Standardmäßig verdeckt die Fläche alles was unter ihr liegt, sie kann aber auch unter ein Bild gezeichnet werden. Die zugehörigen Attributierungsfunktionen sind

$$\begin{array}{lll} setBack & :: & Area \to Area \\ setFront & :: & Area \to Area \end{array}$$

let $p = ref\ (0 \triangleleft C)$ -- $ref\ (1 \triangleleft C)$
  -- $ref\ (2 \triangleleft C)$ -- $ref\ (3 \triangleleft C)$
  -- $ref\ (2 \triangleleft C)$ -- $ref\ (1 \triangleleft C)$ -- *cycle*
in *fill* [*toArea p* # *setBack* # *setPen* 13,
  toArea p # *setBack* # *setPen* 12 # *setColor* 0.9]
  binTree

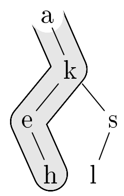


Abbildung 3.12.1: Flächen können mit Hilfe der Attributierung #*setBack* unter Bilder, statt darüber gezeichnet werden.

Die Funktion

$$fill \quad :: \quad [Area] \to Picture \to Picture$$

ergänzt eine Grafik um die gewünschten Flächen. Abbildung 3.12.1 zeigt, wie nützlich es sein kann, mit Hilfe von *setBack* Flächen hinter eine Grafik zu zeichnen. Bezüglich der resultierenden Bounding Box eines Bildes gilt grundsätzlich alles für Pfade gesagte: Das Einzeichnen von Flächen erhält die Bounding Box eines Bildes.

3.13 Clipping

Ein Effekt von METAPOST, der auch in *functional* METAPOST verwirklicht ist, ist das Abschneiden eines Bildes entlang eines zyklischen Pfades. Abbildung 3.13.1 zeigt was gemeint ist. Alles was außerhalb des angegebenen Pfades liegt, verschwindet. Die Bounding Box des resultierenden Bildes ist das minimal umschließende Rechteck.

$$clip \quad :: \quad Path \to Picture \to Picture$$

Vom angegebenen Pfad findet nur die Form Berücksichtigung. Farbinformationen, Strichmuster, Stifte und Label des Pfades werden ignoriert.

$$
\begin{array}{l}
\mathit{clip}\ (\mathit{ref}\ \mathit{SE} + \mathit{vec}\ (0,15)\ \text{--}\ \mathit{ref}\ C \\
\quad \text{--}\mathit{ref}\ \mathit{NE} - \mathit{vec}\ (0,15)\ ..\ \mathit{ref}\ W \\
\quad ..\mathit{cycle}) \\
\quad (\mathit{matrixSepBy}\ 0\ 0 \\
\qquad\qquad (\mathit{replicate}\ 8\ (\mathit{replicate}\ 4\ \texttt{"pac"})))
\end{array}
$$

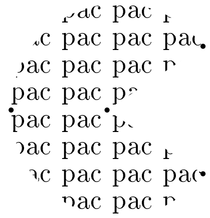


Abbildung 3.13.1: Das Bild ist auf den angegebenen Pfad zurechtgeschnitten. Die vier Punkte, durch die der Pfad verläuft, sind zur Verdeutlichung eingezeichnet.

3.14 Transformationen

Oft soll die Größe einer Grafik geändert oder ein Text gedreht werden. Um dies zu erreichen, gibt es Transformationsbefehle, die sich auf beliebige Bilder anwenden lassen.[10] Vordefiniert sind z.B. Befehle zum Skalieren, Drehen, Scheren entlang der Achsen und Spiegeln von Bildern.

$$
\begin{array}{lll}
\mathit{scale} & :: & \mathit{Numeric} \to \mathit{Picture} \to \mathit{Picture} \\
\mathit{scaleX} & :: & \mathit{Numeric} \to \mathit{Picture} \to \mathit{Picture} \\
\mathit{scaleY} & :: & \mathit{Numeric} \to \mathit{Picture} \to \mathit{Picture} \\
\mathit{rotate} & :: & \mathit{Numeric} \to \mathit{Picture} \to \mathit{Picture} \\
\mathit{skewX} & :: & \mathit{Numeric} \to \mathit{Picture} \to \mathit{Picture} \\
\mathit{skewY} & :: & \mathit{Numeric} \to \mathit{Picture} \to \mathit{Picture} \\
\mathit{reflectX} & :: & \mathit{Picture} \to \mathit{Picture} \\
\mathit{reflectY} & :: & \mathit{Picture} \to \mathit{Picture}
\end{array}
$$

Wie beim Clipping ist die Bounding Box des resultierenden Bildes das minimal umschließende Rechteck.

$$
\begin{array}{ll}
 & \mathit{rotate}\ 45\ \mathit{color1} \\
▯▯ & \mathit{scale}\ 2\ (\mathit{rotate}\ 45\ \mathit{color1}) \\
▯▯ & \mathit{reflectY}\ (\mathit{skewX}\ 1.2\ \mathit{color1}) \\
▯▯ & \mathit{rotate}\ 45\ (\mathit{skewY}\ 1.2\ \mathit{color1}) \\
▯▯ & \mathit{skewY}\ 1.2\ (\mathit{rotate}\ 45\ \mathit{color1})
\end{array}
$$

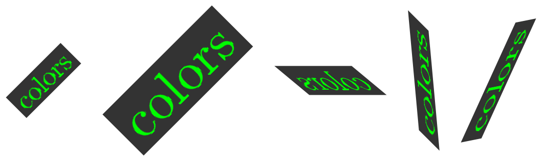


Abbildung 3.14.1: Verschiedene affine Transformationen.

Wenn in einem transformierten Bild ein Punkt benannt ist, wird bei einer Referenz auf diesen die Transformation automatisch berücksichtigt. Die ist notwendig, damit z.B. eine Linie, die durch einen solchen Punkt verläuft, nicht durch den untransformierten Punkt geht.

[10] Weil PostScript Linien immer als Flächen betrachtet, ändert sich durch eine Transformation eventuell die Strichstärke.

3.15 Bitmap Grafiken

METAPOST unterstützt leider keine Bitmap Grafiken. Trotzdem wartet *functional* METAPOST mit diesem Feature auf, was allerdings nur mit einem Trick möglich wurde, den wir in Abschnitt 7.8.2 beschreiben.
Möglich sind Bilder mit einem Bit, acht Bit Graustufen oder 24 Bit pro Bildpunkt. Ein Punkt hat, wenn die Grafik nicht skaliert wird, eine Kantenlänge von $\frac{1}{600}$ Inch. Abbildung 3.15.1 zeigt eine Bitmap mit einem Bit Tiefe um den Faktor zwanzig vergrößert.

```
let turkey = ["003B00","002700","002480","0E4940",
              "114920","14B220","3CB650","75FE88",
              "17FF8C","175F14","1C07E2","3803C4",
              "703182","F8EDFC","B2BBC2","BB6F84",
              "31BFC2","18EA3C","0E3E00","07FC00",
              "03F800","1E1800","1FF800"]
in scale 20 (image Depth1 turkey)
```

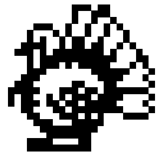


Abbildung 3.15.1: Eine Bitmap Grafik, siehe [Ado85].

3.16 Untertypen von *Picture*, *Path* und *Name*

Bisher haben wir in einer idealisierenden Herangehensweise den Typ aller Bilder als *Picture* angegeben. Aus folgenden Gründen müssen wir uns genauer mit dem Typ *Picture* beschäftigen:

1. Rahmen sind spezielle Bilder. Sie besitzen zusätzliche Attribute um den Abstand der Umrandung festzulegen und entsprechende Attributierungsfunktionen wie z.B. *setDX*, die auf normale Bilder nicht anwendbar sind. Davon abgesehen hat die Funktion *setColor* bei Rahmen die Bedeutung, das Farbattribut der Umrandung und nicht die Farbe des gesamten Bildes festzulegen. Deshalb sind Rahmen keine Objekte vom Typ *Picture*, sondern vom Typ *Frame*. Konzeptionell ist der Typ *Frame* ein Untertyp von *Picture*, der noch weitere Attribute besitzt. Leider unterstützt Haskell nicht direkt Untertypen[11].

2. Es wäre wünschenswert, daß wir Abstraktionen von komplexen Objekten mit Hilfe kleiner „Spezialsprachen“ formulieren können, die bei Bedarf automatisch in ein Bild umgewandelt werden.

Die Problematik, die der erste Punkt beschreibt, stellt sich für Pfade und Namen in ähnlicher Weise. Mit der Verwendung von Typklassen läßt sich für beide Punkte eine Lösung finden, die das Konzept von Untertypen simuliert.

[11] In C++ könnte man eine Klasse *Picture* definieren, *Frame* von dieser ableiten und dann Funktionen wie *setDX* für die Klasse *Frame* einführen. Wenn die Funktion *setColor* virtuell ist, kann sie für Rahmen die Farbe der Umrandung festlegen, während sie auf Bilder angewendet, das ganze Bild färbt.

3.16.1 *Picture*

Der Trick besteht darin, von den eigentlichen Typen der Objekte zu abstrahieren indem wir eine Typklasse *IsPicture* bilden, von der alle Untertypen von *Picture* Instanzen sind.

```haskell
class (Show a) => IsPicture a where
    toPicture          :: a -> Picture
    toPictureList      :: [a] -> Picture
    toPicture a        =  text (show a)
    toPictureList ps   =  row (map toPicture ps)
```

Für Bilder ist die Funktion *toPicture* die identische Abbildung. Die anderen Typen, die Untertypen von *Picture* sein sollen, müssen Instanz der Klasse *IsPicture* sein und damit die Funktion *toPicture* implementieren. Auf diese Weise können Ausdrücke der Untertypen jederzeit in ein Bild umgewandelt werden.
Z.B. Rahmen haben den Typ Frame und besitzen weitere Attribute. Wenn wir auf einen Rahmen eine Funktion anwenden wollen, die eigentlich ein Bild erwartet, muß der Rahmen mit *toPicture* in ein solches umgewandelt werden. Es wäre nichts gewonnen, wenn der Anwender dafür Sorge tragen müßte, daß dies geschieht. Deshalb wenden die Funktionen von *functional* METAPOST automatisch auf alle Parameter, die ein Bild erwarten, *toPicture* an. Jetzt akzeptieren diese Funktionen, alle Untertypen von *Picture* als Argumente. Der Typ der Kombinatoren, den wir bisher vereinfacht angegeben haben, lauten z.B.

```haskell
(|||)        :: (IsPicture a, IsPicture b) => a -> b -> Picture
p1 ||| p2    =  row [toPicture p1, toPicture p2]
```

Wir können nun auch eigene Untertypen für beliebige Abstraktionen bilden. Machen wir uns dies an einem Beispiel klar. Wir definieren einen Datentyp, mit dem wir Bäume darstellen können.

```haskell
data Tree    =  N Tree Picture Tree
             |  E
```

Damit der Typ *Tree* zum Untertyp von *Picture* wird, braucht es eine Instanzdeklaration folgender Art[12].

```haskell
instance IsPicture Tree where
    toPicture t  =  draw edges (overlay equations nodePics)
        where
        edges      =  ...
        equations  =  ...
        nodePics   =  ...
```

Nun können Ausdrücke vom Typ *Tree* wie normale Bilder behandelt werden.

```haskell
pic          ::  Picture
pic          =   t1 ||| t2
    where
    t1, t2   ::  Tree
    t1       =   N E (tex "2") (N E (tex "3") E)
    t2       =   N (N E (tex "1") E) (tex "2") (N E (tex "3") E)
```

[12] In Kapitel 5.3.3 ist diese Funktion ausführlich diskutiert.

In *functional* METAPOST sind schon viele einfache Typen Instanzen der Klasse *IsPicture*. Darunter befinden sich die Typen *Char*, *String*, *Int*, *Integer*, *Numeric* sowie Tupel, Tripel und Listen.

$$pic' \quad = \quad \texttt{"String"}\ ▯▯\ 2$$

Leider unterliegen Listen in Haskell der Typhomogenität, was Ausdrücke wie $[\texttt{"String"}, 2]$ verbietet. Diese Einschränkung läßt sich nicht umgehen und muß für die Typinferenz in Kauf genommen werden.
Auch die Typen *Path* und *Area* sind Instanzen der Typklasse *IsPicture*. Am Ende von Abschnitt 3.4 lernten wir eine Methode kennen, einen Pfad in ein Bild umzuwandeln. Jetzt haben wir einen einfacheren Weg und können einen Pfad, wenn wir darauf Funktionen von *functional* METAPOST anwenden, wie ein Bild betrachten.

3.16.2 *Path*

Bei den Pfadkonstruktoren ergibt sich für uns ein ähnliches Problem. Wenn diese wirklich den Typ *Path* $\rightarrow$ *Path* $\rightarrow$ *Path* hätten, ließe sich mit ihnen nicht eine Reihe von Punkten verbinden, ohne die Punkte mit Hilfe einer Funktion vorher in einen Pfad überführt zu haben. Die würde etwa so aussehen wie der Ausdruck *path* p_1 -- *path* p_2 .. *path* p_3. Der Lesbarkeit von Pfadausdrücken wäre das nicht besonders förderlich.
Um sowohl Pfade, wie auch Punkte mit den Pfadkonstruktoren verbinden zu können, beschreitet *functional* METAPOST hier einen ähnlichen Weg, wie bei Bildern. Die Argumente der Pfadkonstruktoren können jeden Typ haben, der Instanz der Klasse *IsPath* ist und damit die Funktion *toPath* bereitstellt.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{class}\ \mathit{IsPath}\ a\ \textbf{where}} \\
\quad \mathit{toPath} & :: & a \rightarrow \mathit{Path} \\
\quad \mathit{toPathList} & :: & [a] \rightarrow \mathit{Path} \\
\quad \mathit{toPathList}\ ps & = & \mathit{foldl1}\ (\text{--})\ (\mathit{map}\ \mathit{toPath}\ ps)
\end{array}
$$

Der Typ der Pfadkonstruktoren in Haskell drückt diese Forderung an die Argumente aus.

$$(\&), (..), (...), (\text{--}), (\text{---}) \quad :: \quad (\mathit{IsPath}\ a, \mathit{IsPath}\ b) \Rightarrow a \rightarrow b \rightarrow \mathit{Path}$$

Als Beispiel seien hier noch ein paar Instanzdeklarationen für wichtige Typen gegeben, die eine intuitive Notation von Pfaden ermöglichen.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{instance}\ \mathit{IsPath}\ \mathit{Path}\ \textbf{where}} \\
\quad \mathit{toPath} & = & \mathit{id} \\
\\
\multicolumn{3}{l}{\textbf{instance}\ \mathit{IsPath}\ \mathit{Point}\ \textbf{where}} \\
\quad \mathit{toPath} & = & \mathit{PathPoint} \\
\\
\multicolumn{3}{l}{\textbf{instance}\ \mathit{IsPath}\ \mathit{Name}\ \textbf{where}} \\
\quad \mathit{toPath} & = & \mathit{toPath} \circ \mathit{ref} \\
\\
\multicolumn{3}{l}{\textbf{instance}\ \mathit{IsPath}\ a \Rightarrow \mathit{IsPath}\ [a]\ \textbf{where}} \\
\quad \mathit{toPath} & = & \mathit{toPathList}
\end{array}
$$

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\textbf{instance}\ (\mathit{Num}\ a, \mathit{Num}\ b, \mathit{Real}\ a, \mathit{Real}\ b) \Rightarrow \mathit{IsPath}\ (a,b)\ \textbf{where}} \\
\quad \mathit{toPath}\ (a,b) & = & \mathit{toPath}\ (\mathit{vec}\ (\mathit{fromRational}\ \$\ \mathit{toRational}\ a, \\
 & & \qquad \mathit{fromRational}\ \$\ \mathit{toRational}\ b))
\end{array}
$$

Neben der Erleichterung Punkte verbinden zu können, dürfen wir jetzt auch einen Ausdruck *ref* n_1 -- *ref* n_2 durch n_1 -- n_2 oder $[n_1, n_2]$ abkürzen. Die letzte Deklaration ermöglicht Konstruktionen der Art $(0,0)$ -- $(10,0)$ -- $(10,10)$ -- $cycle$. Damit haben wir eine maximale Annäherung an das Erscheinungsbild der Notation von METAPOST.

3.16.3 *Name*

Namen können aus Bestandteilen der Typen *Int*, *String* oder *Dir* gebildet werden. Deshalb machen wir diese auf dem bekannten Weg zu Untertypen von *Name*. Wir kommen in Abschnitt 7.2.2 noch einmal auf genaue Implementierung zurück, da der Untertyp *String* noch eine Schwierigkeit beinhaltet.
Während der Einführung in *functional* METAPOST haben wir bisher an vielen Stellen Namensbestandteile aus Integerkonstanten gebildet, wie in dem Ausdruck $1 \lessdot C$. Das ist in Haskell leider nicht zulässig, da nicht zugesichert ist, daß die Zahlenkonstante den Typ *Int* hat. Deshalb ist der Typ entweder mit Angabe der Typsignatur $(1 :: \mathit{Int}) \triangleleft C$ zu notieren oder als $1 \lessdot C$ mit Hilfe der Funktion

$$
\begin{array}{lcl}
(\lessdot) & :: & (\mathit{IsName}\ a) \Rightarrow \mathit{Int} \rightarrow a \rightarrow \mathit{Name} \\
(\lessdot) & = & (\triangleleft)
\end{array}
$$

deren Typsignatur das erste Argument des Operators auf den Typ *Int* festlegt. Abbildung 3.16.1 gibt eine abschließende Übersicht über das System der Untertypen von *functional* METAPOST.

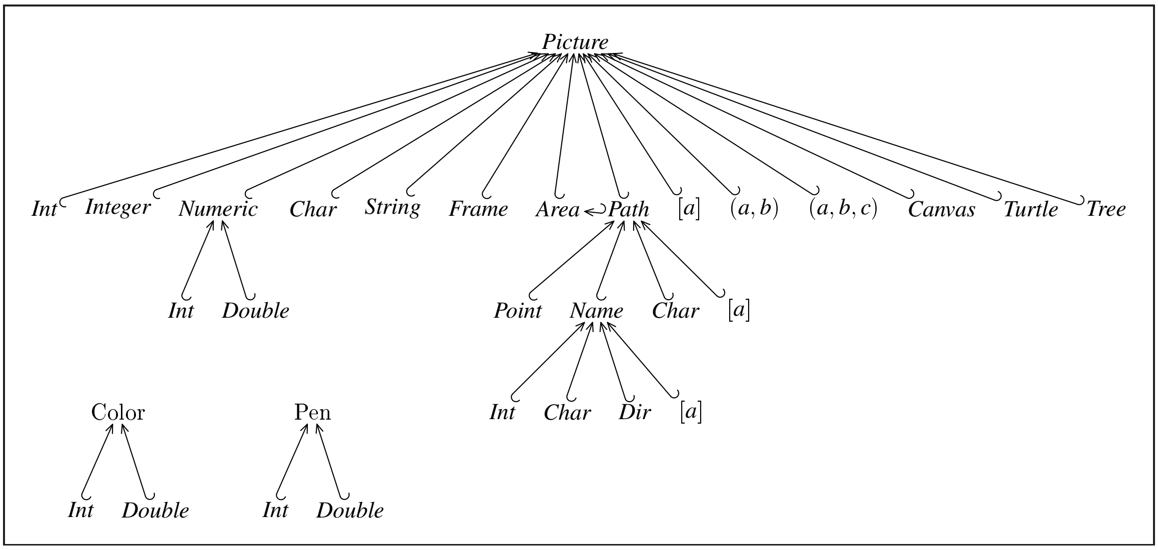


Abbildung 3.16.1: Diagramm aller Untertypen. Die Beziehung $[a] \hookrightarrow \mathit{Picture}$ gilt für alle Typen a, für die auch $a \hookrightarrow \mathit{Picture}$ gilt.

3.17 Sichtbarkeit und Verdeckung von Variablen

Variablen, die in einem Ausdruck der Form *define equations picture* innerhalb der Gleichungen *equations* ihr definierendes Vorkommen haben, sind nur im Bild *picture* sichtbar, d.h. referenzierbar.

Variablen, die in einem Ausdruck der Form *overlay equations pictures* innerhalb der Gleichungen *equations* ihr definierendes Vorkommen haben, sind global sichtbar.
Die Namen von Variablen müssen nicht eindeutig sein. Ein Beispiel sind die Namen der Bezugspunkte $C, ..., NW$, die in jedem Bild vorkommen. Daher besteht die Notwendigkeit, Regeln für die Verdeckung von Variablen festzulegen, die auf eindeutige Weise bestimmen, welche Variable ein angewandtes Vorkommen referenziert.
Wenn wir im folgenden von definierenden Variablenvorkommen reden, kürzen wir dies mit d.V. und angewandtes Vorkommen mit a.V. ab. Ein a.V. einer Variable referenziert das d.V. dieser Variable, das nicht verdeckt ist.
Wir benötigen noch den Begriff eines globalen d.V.: Ein globales d.V. sei ein solches, daß nicht im Ausdrucks des a.V. auftritt. Z.B. sind für den Ausdruck p_1, der in der Bildbeschreibung *overlay eqs* $[p_1, p_2]$ vorkommt, alle d.V. des Ausdrucks p_2 global oder für den Ausdruck *p*, der in der Bildbeschreibung *define eqs p* vorkommt, sind alle d.V. des Gleichungssystems *eqs* global. Dagegen sind für den Ausdruck *define eqs p*, die d.V. des Gleichungssystems eqs und des Bildes *p* nicht global. Bei globalen d.V. handelt es sich also um d.V. aus dem Kontext eines Ausdrucks. Wir können nun die Regeln für Verdeckungen von d.V. definieren.

1. Ein globales d.V. wird von einem nicht globalen verdeckt.

2. Für den Ausdruck *overlay eqs ps* verdecken d.V. aus dem Gleichungssystem *eqs* alle d.V. aus den Bildern *ps*.

3. Wenn der Ausdruck *p* im Kontext *define eqs p* steht, verdecken die globalen d.V. aus *eqs* alle anderen globalen d.V.

4. Für zwei d.V. in den Bildern p_i und $p_j, 1 \leqslant i, j \leqslant n$ des Ausdrucks *overlay eqs* $[p_1 \mathinner{..} p_n]$ gilt: wenn $i < j$, dann verdeckt das d.V. in p_i das d.V. in p_j.

5. In einem Gleichungssystem verdeckt das d.V. einer Variable alle anderen d.V. dieser Variable außerhalb des Gleichungssystems.

Die Definition der Verdeckungsregeln ist mit Bedacht in dieser Weise gewählt. Überlegen wir, was ohne Regel 1. passieren würde, wenn also ein globales d.V. ein d.V. innerhalb eines Ausdrucks verdecken könnte. Dieses Bild wäre dann plötzlich kontextabhängig. Das bedeutet, daß sich das Aussehen dieses Bildes ungewollt ändern kann, wenn der Name eines globalen d.V. der gleiche ist, wie der eines nicht globalen d.V und plötzlich das globale d.V. referenziert wird. Für wiederverwendbare Bildteile, die in jedem Kontext gleich aussehen sollen, wäre dieses Verhalten inakzeptabel. Die Regeln 2. und 3. formalisieren die Vorstellung, daß ein „näheres“ d.V., “entferntere“ d.V. verdeckt. Regel 4. definiert eine Ordnung für Verdeckungen der d.V. aus Argumenten der Funktion *overlay*. Da, wie wir in Kapitel 7 sehen werden, alle Bildkombinatoren auf die Funktion *overlay* zurückgehen, genügt dies. Schließlich garantiert Regel 5. daß d.V. innerhalb eines Gleichungssystems in diesem auch immer referenziert werden. Dies läßt sich auch als Spezialfall der Regel 2. verstehen.

Kapitel 4

Erweiterungen von *functional* METAPOST

Im letzten Kapitel haben wir die Bildbeschreibungssprache *functional* METAPOST kennengelernt. Dabei benutzten wir ein bestimmtes Konzept zur Beschreibung von Grafiken. Teilbilder werden relativ zueinander ausgerichtet und somit zu neuen Bildern kombiniert. Darauf können dann noch Pfade oder Flächen gezeichnet werden. Mit diesem Grafikparadigma lassen sich bestimmte Arten von Grafiken, wie z.B. Diagramme sehr gut beschreiben. Andere Grafiken, wie das Zeichnen in absoluten Koordinaten, sind dagegen in diesem Ansatz schlechter beschreibbar.
Die Kernsprache von *functional* METAPOST ist allerdings so leistungsfähig, daß wir auf ihrer Basis andere Konzepte entwerfen können. Deshalb definieren wir in diesem Kapitel Erweiterungen, die weitere Beschreibungskonzepte realisieren.
Mit Canvasgrafik, einem Konzept, das vielen Grafikschnittstellen zugrunde liegt, entsteht ein Bild durch die Aneinanderreihung einzelner Zeichenbefehle. Bei Turtlegrafik wird ein virtueller Stift über eine Zeichenfläche gesteuert. Schließlich geben wir ein Beispiel für eine komplexe Abstraktion, nämlich Bäume, die sich automatisch ästhetisch zeichnen.
Alle Grafikparadigmen sind kombinierbar und ineinander einbettbar, d.h. eine Canvasgrafik kann in einem Baum eingebettet werden und umgekehrt.

4.1 Canvasgrafik

Für viele Anwendungen, z.B. ein Funktionsplot, ist ein lokales Koordinatensystem mit potentiell unendlicher Größe praktisch. Dieser Anwendung liegt die Metapher einer Leinwand (*Canvas*) zugrunde, auf die verschiedene Zeichenoperationen angewendet werden können.
Der Zeichenbefehl bildet einen Pfad auf einen *Canvas* ab.[1]

$$cdraw \quad :: \quad Path \to Canvas$$

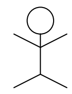
Grafiken, die aus mehr als einem Zeichenbefehl bestehen, erhalten wir mit Hilfe des Sequenzoperators.

$$(\&) \quad :: \quad Canvas \to Canvas \to Canvas$$

[1] Wir setzen unsere idealisierte Vorstellung der Typen aus dem letzten Kapitel fort und geben die Typen nicht in der Form $cdraw :: IsPath\ a \Rightarrow a \to Canvas$ an, um eine bessere Lesbarkeit zu erreichen.

Damit lassen sich schon kleine Strichmännchen zeichnen. Für diese Klasse von Anwendungen macht es Sinn, ein Bild mit absoluten Koordinaten zu beschreiben.

$$\begin{array}{ll} & \mathit{cdraw}\ (\mathit{vec}\ (0,5)\ \text{--}\ \mathit{vec}\ (0,-10)) \\ \& & \mathit{cdraw}\ (\mathit{vec}\ (10,5)\ \text{--}\ \mathit{vec}\ (0,0)\ \text{--}\ \mathit{vec}\ (-10,5)) \\ \& & \mathit{cdraw}\ (\mathit{vec}\ (10,-15)\ \text{--}\ \mathit{vec}\ (0,-10)\ \text{--}\ \mathit{vec}\ (-10,-15)) \\ \& & \mathit{cdraw}\ (\mathit{vec}\ (0,5)\ ..\ \mathit{vec}\ (-5,10)\ ..\ \mathit{vec}\ (0,15)\ ..\ \mathit{vec}\ (5,10)\ ..\ \mathit{cycle}) \end{array}$$



Es existieren weitere Befehle zum Zeichnen mehrer Pfade, zum Füllen von Flächen und Clipping. Der Typ *Canvas* ist Instanz der Klasse *IsPicture*, d.h. wir können eine Canvasgrafik wie ein gewöhnliches Bild, als Parameter der Kombinatoren oder anderer Funktionen benutzen.

$$\begin{array}{lll} \mathit{cdraws} & :: & [\mathit{Path}] \to \mathit{Canvas} \\ \mathit{cfill} & :: & \mathit{Area} \to \mathit{Canvas} \\ \mathit{cfills} & :: & [\mathit{Area}] \to \mathit{Canvas} \\ \mathit{cclip} & :: & \mathit{Path} \to \mathit{Canvas} \\ \mathit{relax} & :: & \mathit{Canvas} \end{array}$$

Ein Bild kann an einer beliebigen Position in die Canvasgrafik eingebettet werden.

$$\begin{array}{lll} \mathit{cdrop} & :: & (\mathit{Numeric},\mathit{Numeric}) \to \mathit{Path} \to \mathit{Canvas} \end{array}$$

Abbildung 4.1.1 zeigt noch ein anderes typisches Anwendungsbeispiel. Um es nochmal zu erwähnen: Alle absoluten Koordinatenangaben einer Sequenz von Zeichenbefehlen erfolgen in dem selben lokalen Koordinatensystem. Wenn die Canvasgrafik aber in ein Bild umgewandelt ist, kann dieses frei neben anderen Bildern plaziert sein.

$$\begin{array}{ll} & \mathit{cdraws}\ [\mathit{arrow}\ (\mathit{vec}\ (-10,0))\ (\mathit{vec}\ (210,0)), \\ & \qquad \mathit{arrow}\ (\mathit{vec}\ (0,-50))\ (\mathit{vec}\ (0,50))] \\ \& & \mathit{cdrop}\ (-6,50)\ (\mathit{math}\ \texttt{"y"}) \\ \& & \mathit{cdrop}\ (210,-6)\ (\mathit{math}\ \texttt{"x"}) \\ \& & \mathit{cdraw}\ (\mathit{toPath}\ [\mathit{vec}\ (i,40 * \mathit{sin}\ (3 * i)) \mid i \leftarrow [0,2\,..\,200]]) \\ \& & \mathit{cdraw}\ (\mathit{toPath}\ [\mathit{vec}\ (i,40 * \mathit{sin}\ (3 * i) ** 2) \mid i \leftarrow [0,2\,..\,200]]) \end{array}$$

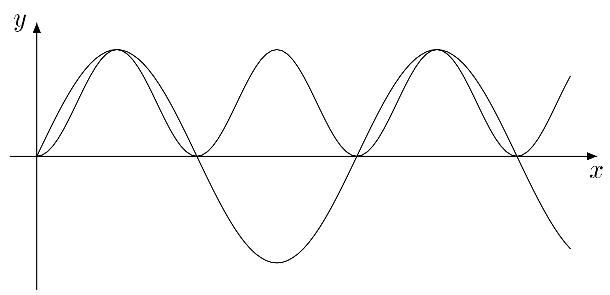


Abbildung 4.1.1: Canvasgrafik eignet sich besonders für Diagramme und Funktionsplots.

4.2 Turtlegrafik

Unter Turtlegrafik[2] [Ad82] versteht man im allgemeinen das Zeichenkonzept der Sprache LOGO [Abe85]. Zeichnungen entstehen dabei durch Bewegungen eines virtuellen Plotterstiftes, Turtle[3] genannt. Der Turtle kennt die Zustände gesenkt und angehoben. Wird er im gesenkten Zustand bewegt, hinterläßt er eine Linie auf dem Bildschirm. Drehbefehle ändern die „Laufrichtung“ des Turtles. Turtlegrafik ist immer da nützlich, wo es auf relatives Zeichnen ankommt, und die Grafik aus kleinen Bausteinen aufgebaut ist.
Die beiden grundlegenden Anweisungen zur Steuerung des Turtles bewegen ihn vorwärts bzw. drehen ihn um eine bestimmte Gradzahl.

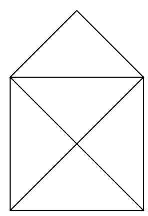
$$\begin{array}{lll} forward & :: & Numeric \to Turtle \\ turn & :: & Numeric \to Turtle \end{array}$$

Diese Anweisungen führen mit Hilfe des Sequenzoperators

$$\begin{array}{lll} (\&) & :: & Turtle \to Turtle \to Turtle \end{array}$$

zur Beschreibung eines Turtlepfades. Mit einer Sequenz aus den ersten beiden Anweisungen können wir schon einen geschlossenen Pfad zeichnen.

$$\begin{array}{ll} & forward\ 50\ \&\ turn\ 90 \\ \& & forward\ 50\ \&\ turn\ 90 \\ \& & forward\ 50\ \&\ turn\ 90 \\ \& & forward\ 50\ \&\ turn\ 135 \\ \& & forward\ (50 * sqrt\ 2)\ \&\ turn\ 90 \\ \& & forward\ (25 * sqrt\ 2)\ \&\ turn\ 90 \\ \& & forward\ (25 * sqrt\ 2)\ \&\ turn\ 90 \\ \& & forward\ (50 * sqrt\ 2) \end{array}$$



Daneben existieren Anweisungen zum Heben und Senken des Stiftes, dem Ansteuern eines Ursprungs und der Befehl, der keine Wirkung hat. Ein Befehl ohne Wirkung ist als neutrales Element nützlich, wenn wir eine Liste von Befehlen mit der Funktion *foldr* (&) *relax* in einen Turtlepfad falten wollen.

$$\begin{array}{lll} penUp & :: & Turtle \\ penDown & :: & Turtle \\ home & :: & Turtle \\ relax & :: & Turtle \end{array}$$

Wir können zusätzlich noch die oft benötigten Funktionen für Drehungen um 90 Grad ableiten.

$$\begin{array}{lll} toleft, toright & :: & Turtle \\ toleft & = & turn\ 90.0 \\ toright & = & turn\ (-90.0) \end{array}$$

Eine klassische Anwendung für Turtlegrafik sind fraktale Kurven, die sich damit besonders leicht beschreiben lassen. Siehe Abbildung 4.2.1.

[2] Es gehört wohl zu den ewigen Geheimnissen, warum in der deutschen Literatur [Abe85] oft von Igelgrafik die Rede ist. Wir benutzen dagegen den Ausdruck Turtlegrafik.

[3] Der Name Turtle ist durchaus wörtlich zu verstehen. Da die Sprache LOGO von SEYMOUR PAPERT [Pap82] unter pädagogischen Aspekten entwickelt wurde , und zur ersten Heranführung von Kindern an Programmiertechniken Verwendung findet, hat der Zeichenstift auf dem Bildschirm die Form einer Schildkröte. Dies wiederum hat seinen Ursprung in den frühen Siebzigern, als noch Kathodenstrahlröhren zu teuer waren und die Turtlebefehle einen kleinen Roboter steuerten, der wie eine Schildkröte aussah.

$$
\begin{array}{lll}
\mathit{dragon}\ g\ l & & \\
\quad |\, g \equiv 0 & = & \mathit{forward}\ l\ \#\ \mathit{setPen}\ 1 \\
\quad |\, g > 0 & = & \mathit{dragon}\ (g-1)\ l \\
 & & \&\mathit{toright} \\
 & & \&\mathit{dragon}\ (-(g-1))\ l \\
\quad |\, \mathit{otherwise} & = & \mathit{dragon}\ (-(g+1))\ l \\
 & & \&\mathit{toleft} \\
 & & \&\mathit{dragon}\ (g+1)\ l
\end{array}
$$

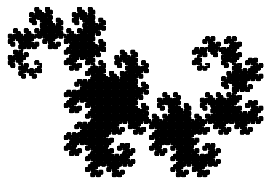


Abbildung 4.2.1: Drachen-Füllkurve der Tiefe Zwölf. Die Strichstärke entspricht der Segmentlänge. Damit entsteht die ausgefüllte Drachenkurve. Eine ausführliche Diskussion der Drachenkurve findet sich in [Man87] Kapitel 7.

Gemäß unseres Konzepts der Untertypen, ist auch der Datentyp *Turtle* eine Instanz von *IsPicture*. Die Einbettung eines beliebigen Bildes leistet die Anweisung

$$\mathit{fromPicture} \quad :: \quad \mathit{Picture} \to \mathit{Turtle}$$

die an der aktuellen Position des Turtles ein Bild einfügt.
Eine Spezialität, die normalerweise nicht zum Funktionsumfang von Turtlegrafik gehört, ist der Befehl *fork*, der an der aktuellen Turtleposition einen zweiten Pfad erzeugt. Damit entsteht eine Gabelung, die z.B. für baumartige Strukturen, wie in Abbildung 4.2.2, nützlich ist.

$$\mathit{fork} \quad :: \quad \mathit{Turtle} \to \mathit{Turtle} \to \mathit{Turtle}$$

Zum Abschluß noch ein weiteres Beispiel, in dem wir einmal eine Funktion als Parameter übergeben, die eine weitere Etage, bzw. ein Dach zeichnet. Ein rotes Dach der Breite l läßt sich leicht mit folgender Funktion beschreiben.

$$
\begin{array}{lll}
\mathit{roof} & :: & \mathit{Numeric} \to \mathit{Turtle} \\
\mathit{roof}\ l & = & \mathit{turn}\ 45\ \&\, \mathit{forward}\ (0.5 * l * \mathit{sqrt}\ 2)\ \&\ \mathit{toright} \\
 & & \&\mathit{forward}\ (0.5 * l * \mathit{sqrt}\ 2)\ \&\ \mathit{turn}\ (-45) \\
 & & \#\mathit{setColor}\ \mathit{red}
\end{array}
$$

Die Funktion *story* erwartet als Parameter eine Funktion, die die nächsthöhere Etage beschreibt und die Breite des Hauses. Abbildung 4.2.3 fügt zwei solcher Häuser zusammen. Wir haben hier auch eine Anwendung für das Heben und Senken des Turtlestiftes.

$$
\begin{array}{lll}
\mathit{story} & :: & (\mathit{Numeric} \to \mathit{Turtle}) \to \mathit{Numeric} \to \mathit{Turtle} \\
\mathit{story}\ r\ l & = & \mathit{forward}\ l\ \&\ \mathit{toleft}\ \&\, \mathit{forward}\ l\ \&\ \mathit{toleft}\ \&\, \mathit{forward}\ l \\
 & & \&\mathit{turn}\ 180\ \&\ r\ l\ \&\ \mathit{turn}\ (-45)\ \&\, \mathit{forward}\ (l * \mathit{sqrt}\ 2) \\
 & & \&\mathit{turn}\ (-135)\ \&\, \mathit{forward}\ l\ \&\ \mathit{turn}\ (-135) \\
 & & \&\mathit{forward}\ (l * \mathit{sqrt}\ 2)\ \&\ \mathit{turn}\ (-45)
\end{array}
$$

```
let canopy 0 l a _   = fork (turn a & forward l)
                            (turn (−a) & forward l)
    canopy n l a d   = fork (turn a & forward l
                             & canopy (n − 1) (l ∗ d) a d)
                            (turn (−a) & forward l
                             & canopy (n − 1) (l ∗ d) a d)
in rowSepBy 25 (map (setPen 0.2)
                 [canopy 8 30 80 0.67, canopy 8 30 100 0.62,
                  canopy 8 25 30 0.55, canopy 8 30 90 0.7])
```

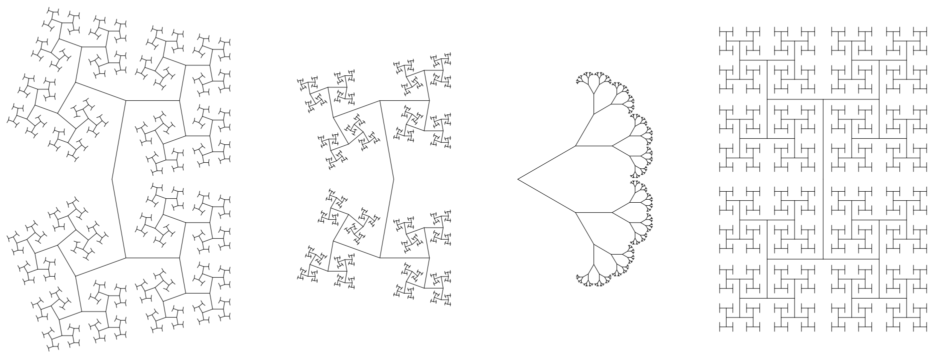


Abbildung 4.2.2: Fraktale Baldachine, siehe [Man87] Kapitel 16.

```
  story roof 20
& home & penUp
& forward 40 & fromPicture "Santa"
& forward 20 & penDown
& story (story (story roof)) 15
  #setPen 2
```

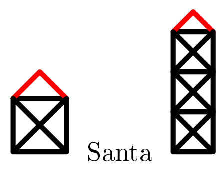


Abbildung 4.2.3: Das Bild besteht aus einem einzigen Turtlepfad. Nachdem das erste Haus gezeichnet ist, wird der Stift angehoben, ein Bild mit Text eingebettet und dann mit gesenktem Stift das rechte Haus gezeichnet.

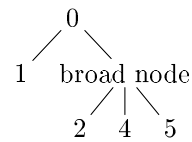
4.3 Bäume

Bäume gehören in der Informatik zweifellos zu den wichtigsten und häufig benötigten Strukturen. Deshalb stellt *functional* METAPOST sie als eigene Objekte zur Verfügung. Ein Baum kann auf diese Weise formal beschrieben werden, das Erscheinungsbild berechnet *functional* METAPOST danach selbstständig.
Ein Baum besteht aus Knoten, die beliebig viele Kanten haben können, die wiederum in Knoten münden.

$$\begin{array}{lll} \mathit{node} & :: & \mathit{IsPicture}\ a \Rightarrow a \rightarrow [\mathit{Edge}] \rightarrow \mathit{Tree} \\ \mathit{edge} & :: & \mathit{Tree} \rightarrow \mathit{Edge} \end{array}$$

Damit läßt sich ein Baum auf einfache Art und Weise beschreiben. Die Aufgabe, die einzelnen Knoten so zu plazieren, daß ein „schöner“ Baum entsteht, übernimmt *functional* METAPOST.

```
tree1 = node "0" [edge (node "1" []),
                  edge (node "broad␣node" [edge (node "2" []),
                                           edge (node "4" []),
                                           edge (node "5" [])])]
```



Der Berechnung des Layouts liegen folgende Regeln[4] zugrunde.

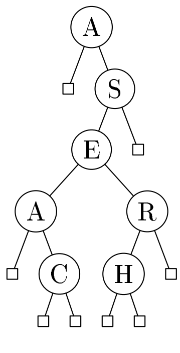
① Knoten, mit demselben Abstand zur Wurzel, liegen auf einer Geraden.

② Ein Vater ist über seinen Söhnen zentriert.

③ Gleiche Teilbäume sehen unabhängig von ihrer Position im Baum gleich aus.

Gleichzeitig paßt sich das Layout an die Breite und Höhe der Bilder im Knoten an. Dieser Punkt ist sehr wichtig, damit sich die Bilder nicht überlappen und die Abstände gleichmäßig sind.
Die Bilder der Knoten können beliebig eingerahmt sein, wie im folgenden Bild eines binären Suchbaums. Siehe [Sed92], Seite 242 für ein schlechteres Layout dieses Baumes.

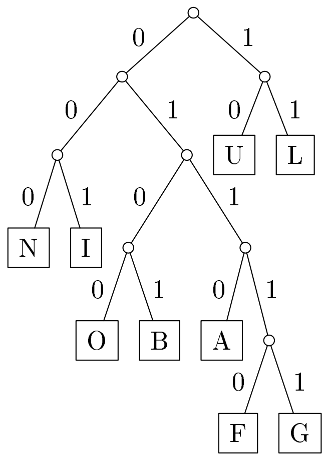
```
let enc s t = edge $ node (circle s) t
    nil = edge $ node (box empty) []
in node (circle "A")
        [nil,
         enc "S" [enc "E" [enc "A" [nil,
                                    enc "C" [nil, nil]],
                           enc "R" [enc "H" [nil, nil],
                                    nil]],
                  nil]]
```



Jede Kante mündet in einen Knoten. Da liegt der Gedanke nahe, auf die Trennung zwischen Kante und Knoten, zugunsten einer kompakteren Schreibweise, zu verzichten. Diesen Weg gehen wir nicht, da Kanten eigene Objekte mit eigenen Attributen sind. Kanten besitzen alle Attribute von Pfaden, Knoten alle Attribute von Bildern und weitere zur Beeinflussung des Layouts.
Z.B. lassen sich Kanten in einem HUFFMAN–Baum mit Markierungen versehen.

[4] Siehe Kapitel 5.3 für eine Motivation und eine genauere Beschreibung der Regeln.

```
let inner l r = node (circle empty)
                     [edge l # setLabel 0.4 SE "0",
                      edge r # setLabel 0.4 SW "1"]
    leaf s = node (box s) []
in inner (inner (inner (leaf "N")
                       (leaf "I"))
                (inner (inner (leaf "O")
                              (leaf "B"))
                       (inner (leaf "A")
                              (inner (leaf "F")
                                     (leaf "G")))))
         (inner (leaf "U")
                (leaf "L"))
```

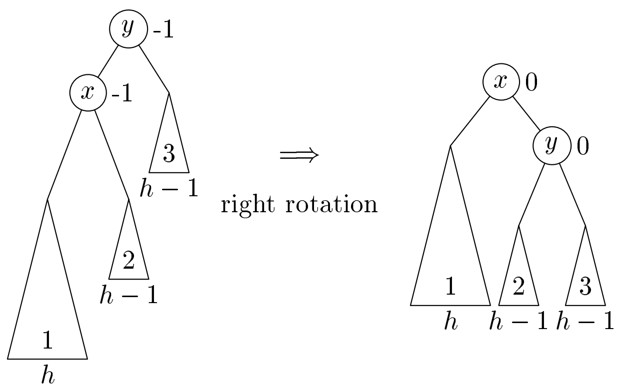


Unser Konzept der Baumbeschreibung ermöglicht auch die Einbeziehung von Querkanten. Diese Kanten erzeugt die Funktion *cross*.

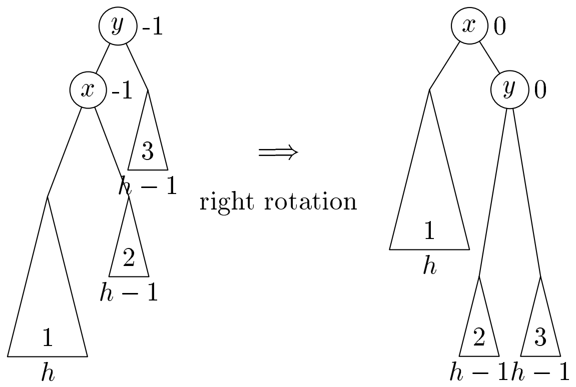
```
cross :: Point → Edge
```

Querkanten können einfach in die Kantenliste eines Knotens aufgenommen werden und verlaufen von diesem Knoten zum angegebenen Punkt. Das Baumlayout wird davon nicht beeinflußt.

Die automatische Layoutgenerierung von Bäumen ist zwar sehr leistungsfähig, aber es gibt Situationen, in denen ein gezielter, vielleicht sehr lokaler manueller Eingriff notwendig ist. Betrachten wir hierzu das folgende Beispiel.



Ohne manuelle Eingriffe würde die automatische Layoutgenerierung nicht das gewünschte Resultat erzeugen.



Im linken Baum ist der horizontale Abstand zwischen den Söhnen der Wurzel zu gering. Beim rechten Baum liegen die unteren Kanten der Dreiecke nicht alle auf einer Geraden.
Um solche Probleme zu lösen, lassen sich sich die Abstände zwischen Knoten horizontal und vertikal beeinflussen.

$$
\begin{array}{lll}
\mathit{setDistH} & :: & \mathit{Distance} \rightarrow \mathit{Tree} \rightarrow \mathit{Tree} \\
\mathit{getDistH} & :: & \mathit{Tree} \rightarrow \mathit{Distance} \\[1ex]
\mathit{setDistV} & :: & \mathit{Distance} \rightarrow \mathit{Tree} \rightarrow \mathit{Tree} \\
\mathit{getDistV} & :: & \mathit{Tree} \rightarrow \mathit{Distance}
\end{array}
$$

Dabei gibt es zwei verschiedene Möglichkeiten Abstände anzugeben. Die erste meint den Abstand zwischen den Rahmen der Knotenbilder. Dies wird standardmäßig angenommen wird, wenn *setDistV* oder *setDistH* mit einer Zahl als Parameter aufgerufen wird.

$$
\mathit{distBorder} \quad :: \quad \mathit{Numeric} \rightarrow \mathit{Distance}
$$

Die andere Definition eines Abstandes ist der, zwischen den Zentren der Knotenbilder. Damit lassen sich die Knoten in einem Raster anordnen, aber das Layout paßt sich nicht mehr automatisch an breitere Bilder an. Eventuell kann es zu ungewollten Überlappungen kommen.

$$
\mathit{distCenter} \quad :: \quad \mathit{Numeric} \rightarrow \mathit{Distance}
$$

Mit diesen Funktionen gerüstet, können wir die Beispielgrafik beschreiben. Zunächst definieren wir Funktionen, um die dreieckigen Knoten mit festen Höhen und Breiten zu erzeugen.

$$
\begin{array}{lll}
\mathit{tri} & :: & \mathit{String} \rightarrow \mathit{Picture} \\
\mathit{tri}\ \texttt{"1"} & = & \mathit{triangle}\ \texttt{"1"}\ \#\ \mathit{setHeight}\ 60\ \#\ \mathit{setWidth}\ 30 \\
 & & \#\mathit{label}\ S\ (\mathit{math}\ \texttt{"h"}) \\
\mathit{tri}\ s & = & \mathit{triangle}\ s\ \#\ \mathit{setHeight}\ 30\ \#\ \mathit{setWidth}\ 15 \\
 & & \#\mathit{label}\ S\ (\mathit{math}\ \texttt{"h-1"})
\end{array}
$$

Am linken Baum ist nur ein geringer Eingriff nötig. Die beiden Söhne der Wurzel sollten einen etwas größeren Abstand haben. Dies geschieht, indem die Attributierungsfunktion *setDistH* 16 auf die Wurzel angewendet wird.

$$
\begin{array}{lll}
\mathit{notRotated} & :: & \mathit{Tree} \\
\mathit{notRotated} & = & \mathit{node}\ (\mathit{circle}\ \texttt{"\$y\$"}\ \#\ \mathit{label}\ E\ \texttt{"-1"}) \\
 & & \qquad [\mathit{edge}\ (\mathit{node}\ (\mathit{circle}\ \texttt{"\$x\$"}\ \#\ \mathit{label}\ E\ \texttt{"-1"})
\end{array}
$$

```
                                        [edge (node (tri "1") []),
                                         edge (node (tri "2") [])]),
                              edge (node (tri "3") [])]
                       #setDistH 16
```

Die Schwierigkeit beim zweiten Baum besteht darin, die unteren Seiten aller Dreiecke auf eine Gerade zu setzen. Deshalb sorgt die Attributierung mit *setDistV* (*distCenter* 30) dafür, daß der vertikale Abstand der Dreiecksspitzen zum Zentrum des Kreises y 30 Punkte beträgt. Weil das erste Dreieck 60 Punkte hoch und die beiden anderen halb so hoch sind, liegen die unteren Seiten auf einer Geraden.

```
rightRotated           :: Tree
rightRotated           =  node (circle "$x$" # label E "0") [
                                edge (node (tri "1") []),
                                edge (node (circle "$y$" # label E "0")
                                          [edge (node (tri "2") []
                                                  # setDistV (distCenter 30)),
                                           edge (node (tri "3") []
                                                  # setDistV (distCenter 30))]
                                      # setDistH 10)]
                          #setDistH 16
```

Damit ist der Ausdruck des gesamten Bildes leicht zu formulieren.

```
rowSepBy 10 [toPicture notRotated,
             "$\\Longrightarrow$" ⊟ "right␣rotation",
             toPicture rightRotated]
```

Wenn Bäume eines bestimmten Typs öfter vorkommen, wird man für diesen eine Abstraktion formulieren. Wir tun dies für Binärbäume.

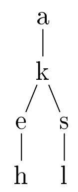
```
data BinTree           =  BNode BinTree Picture BinTree
                       |  BEmpty
```

Eine Konvertierungsfunktion kann einen Binärbaum in den Typ *Tree* umwandeln.

```
bin                    :: BinTree → Tree
bin BEmpty             =  node "empty␣bin" []
bin (BNode BEmpty p BEmpty)
                       =  node p []
bin (BNode l p BEmpty) =  node p [edge (bin l)]
bin (BNode BEmpty p r) =  node p [edge (bin r)]
bin (BNode l p r)      =  node p [edge (bin l), edge (bin r)]
```

Das Ergebnis dieses Versuchs ist ernüchternd. Wenn ein Knoten nur einen Sohn hat ist nicht ersichtlich, ob es sich um den linken oder rechten Sohn handelt. Der Layoutalgorithmus positioniert den Vater genau über den Sohn.

$$
\begin{array}{l}
\textbf{let}\ e = \mathit{BEmpty} \\
\quad\ n = \mathit{BNode} \\
\textbf{in}\ \mathit{bin}\ (n\ e \\
\qquad\qquad (\mathit{tex}\ \texttt{"a"}) \\
\qquad\qquad (n\ (n\ e\ (\mathit{tex}\ \texttt{"e"})\ (n\ e\ (\mathit{tex}\ \texttt{"h"})\ e)) \\
\qquad\qquad\quad (\mathit{tex}\ \texttt{"k"}) \\
\qquad\qquad\quad (n\ (n\ e\ (\mathit{tex}\ \texttt{"l"})\ e)\ (\mathit{tex}\ \texttt{"s"})\ e)))
\end{array}
$$



In solchen Situationen hilft nur ein lokaler Eingriff in das Layout. Ein Knotenattribut steuert die Ausrichtung der Söhne. Wenn wir dieses Attribut verändern, berücksichtigt dies der Layoutalgorithmus.

$$
\begin{array}{lll}
\mathit{setAlign} & :: & \mathit{AlignSons} \to \mathit{Tree} \to \mathit{Tree} \\
\mathit{getAlign} & :: & \mathit{Tree} \to \mathit{AlignSons}
\end{array}
$$

Für den Fall von Binärbäumen gibt es die beiden Ausrichtungen

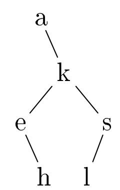
$$
\mathit{alignLeftSon}, \mathit{alignRightSon} \quad :: \quad \mathit{AlignSons}
$$

mit denen ein einzelner Sohn entweder nach rechts oder links verzweigt. Wenn ein Knoten mehrere Söhne hat, werden diese aber wie gewohnt zentriert ausgerichtet.

$$
\begin{array}{lll}
\mathit{bin}' & :: & \mathit{BinTree} \to \mathit{Tree} \\
\mathit{bin}'\ \mathit{BEmpty} & = & \mathit{node}\ \texttt{"empty␣bin"}\ [\,] \\
\mathit{bin}'\ (\mathit{BNode}\ \mathit{BEmpty}\ p\ \mathit{BEmpty}) & & \\
 & = & \mathit{node}\ p\ [\,] \\
\mathit{bin}'\ (\mathit{BNode}\ l\ p\ \mathit{BEmpty}) & = & \mathit{node}\ p\ [\mathit{edge}\ (\mathit{bin}'\ l)] \\
 & & \#\mathit{setAlign}\ \mathit{alignLeftSon} \\
\mathit{bin}'\ (\mathit{BNode}\ \mathit{BEmpty}\ p\ r) & = & \mathit{node}\ p\ [\mathit{edge}\ (\mathit{bin}'\ r)] \\
 & & \#\mathit{setAlign}\ \mathit{alignRightSon} \\
\multicolumn{3}{l}{\mathit{bin}'\ (\mathit{BNode}\ l\ p\ r) = \mathit{node}\ p\ [\mathit{edge}\ (\mathit{bin}'\ l), \mathit{edge}\ (\mathit{bin}'\ r)]}
\end{array}
$$

Mit diesen Hilfen ist das Ergebnis der gewünschte Binärbaum.

$$
\begin{array}{l}
\textbf{let}\ e = \mathit{BEmpty} \\
\quad\ n = \mathit{BNode} \\
\textbf{in}\ \mathit{bin}'\ (n\ e \\
\qquad\qquad (\mathit{tex}\ \texttt{"a"}) \\
\qquad\qquad (n\ (n\ e\ (\mathit{tex}\ \texttt{"e"})\ (n\ e\ (\mathit{tex}\ \texttt{"h"})\ e)) \\
\qquad\qquad\quad (\mathit{tex}\ \texttt{"k"}) \\
\qquad\qquad\quad (n\ (n\ e\ (\mathit{tex}\ \texttt{"l"})\ e)\ (\mathit{tex}\ \texttt{"s"})\ e)))
\end{array}
$$

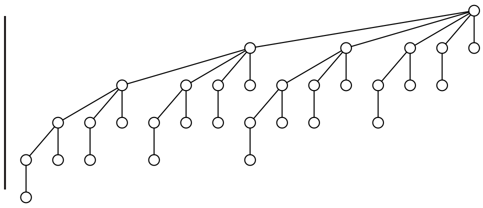


Eine andere Ausrichtung ist für Binomialbäume nützlich. Die Ausrichtungstypen

$$
\mathit{alignLeft}, \mathit{alignRight} \quad :: \quad \mathit{AlignSons}
$$

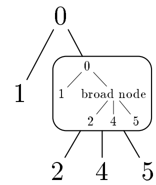
plazieren die Söhne links bzw. rechtsbündig zum Vater.

```
let ce = circle empty
    binom 0 = node ce []
    binom n = node ce [edge (binom i)
                        | i ← [(n − 1), (n − 2) .. 0]]
              # setAlign AlignRight
in binom 5
```



Noch einmal besonders hervorzuheben ist die Tatsache, daß ein Knoten ein beliebiges Bild aufnehmen kann und sich das Layout des Baumes automatisch auf die Größe der Knoten anpaßt. Damit sind sogar Knoten möglich, die wiederum Bäume sind, wie man es bei bestimmten Datenstrukturen (data–structural bootstrapping [BO96]) kennt.

```
node "0" [edge (node "1" []),
          edge (node (rbox 5 (scale 0.5 tree1)) [edge (node "2" []),
                                                 edge (node "4" []),
                                                 edge (node "5" [])])]
```



Die Bäume der bisherigen Beispiele waren ziemlich klein. Um auch einmal ein Beispiel für ein etwas umfangreicheres Exemplar zu geben, folgt in Abbildung 4.3.1 das Bild des Baums aus Abbildung 44.2 in [Sed92] mit 153 Knoten. Die Baumbeschreibung ist lang aber so einfach, daß sie hier nicht abgedruckt ist.

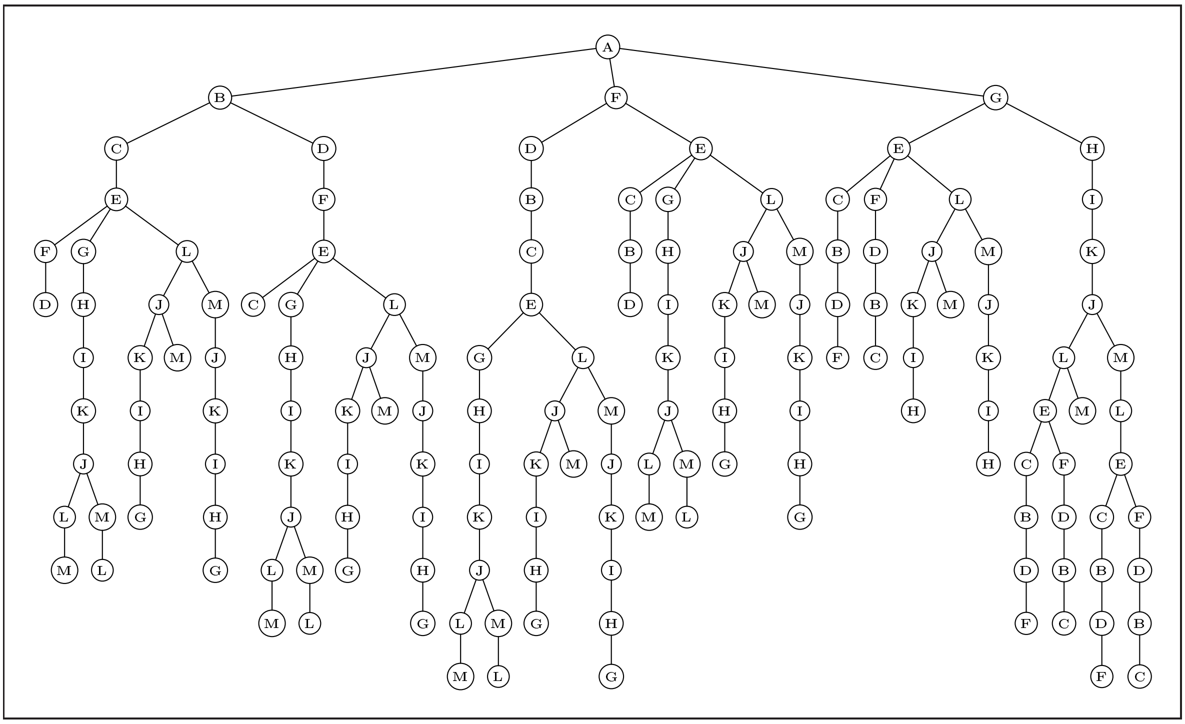


Abbildung 4.3.1: Tiefensuche in einem Graphen zum Finden eines HAMMILTON–Zyklus.

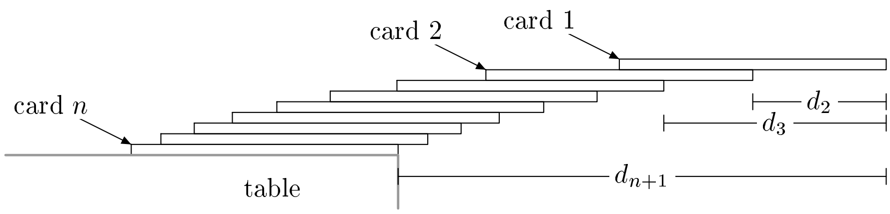
Wie wir gesehen haben, eignet sich die Erweiterung zum Baumlayout nicht nur zur automatischen Generierung regelmäßiger Bäume, sondern erlaubt auch manuelle Eingriffe in das Layout.

4.4 Weitere Beispiele

Am Schluß dieser Einfühung in *functional* METAPOST wollen wir uns noch einige umfangreichere Beispiele anschauen für die in den letzten beiden Kapiteln bisher kein Platz war.

Beispiel 1

Diese Beispiel soll noch einmal verdeutlichen, wie wir von der Einbettung in Haskell profitieren. Immer, wenn sich Teile einer Grafik mit einer Formel berechnen lassen, können wir dazu Haskell verwenden.
Im Buch [GKP92] findet sich auf Seite 259 eine nette Grafik, die das Problem illustriert, n Spielkarten so zu stapeln, daß die oberste möglichst weit über die Tischkante hinausragt.



Man kann zeigen, daß die Abstände d_2 bis d_{n+1} von der Reihe

$$d_{k+1} = \frac{(d_1+1)+(d_2+1)+\ldots+(d_k+1)}{k} \qquad \text{, für } 1 \leqslant k \leqslant n$$

beschrieben sind. Für diese Reihe gilt wiederum $\mathcal{H}_k = d_{k+1}$, wobei $\mathcal{H}_k$ die harmonische Reihe

$$\mathcal{H}_n = \sum_{k=1}^{n} \frac{1}{k} \qquad \text{, für } n \geqslant 0$$

ist. In Haskell läßt sich leicht ein Ausdruck für die harmonische Reihe angeben.[5]

$$\begin{array}{lcl} \mathit{harmonic} & :: & [\mathit{Double}] \\ \mathit{harmonic} & = & 0 : [h + 1 \mathbin{/} k \mid (h,k) \leftarrow \mathit{zip}\ \mathit{harmonic}\ [1\,..]] \end{array}$$

Im nächsten Schritt können wir mit Hilfe der harmonischen Reihe den Kartenstapel zeichnen. Die Ausmaße der einzelnen Karten sollen nachträglich veränderbar sein, also definieren wir Konstanten dafür.

$$\begin{array}{lcl} \mathit{cardW}, \mathit{cardH} & :: & \mathit{Numeric} \\ \mathit{cardW} & = & 100 \\ \mathit{cardH} & = & 4 \end{array}$$

Die horizontale Koordinate der k-ten Karte ergibt sich aus $\mathcal{H}_k$ multipliziert mit der Kartenbreite von rechts nach links.

[5] Diese Definition besitzt ein lineares Laufzeitverhalten. Ein naives Vorgehen wird mit quadratischer Laufzeit bestraft:

$$\begin{array}{lcl} \mathit{harmonic}' & = & [h\ n \mid n \leftarrow [0\,..]] \\ \quad \textbf{where} & & \\ \quad h\ 0 & = & 0 \\ \quad h\ k & = & 1 \mathbin{/} k + h\ (k-1) \end{array}$$

$$
\begin{array}{ll}
\mathit{cardX} & :: \mathit{Numeric} \to \mathit{Numeric} \\
\mathit{cardX}\ k & = -\mathit{cardW} * \mathit{fromDouble}\ (\mathit{harmonic}\ !!\ (\mathit{fromEnum}\ k))
\end{array}
$$

Der Kartenstapel besteht aus neun Karten, die als Rechtecke von oben nach unten mit der entsprechenden horizontalen Position gezeichnet werden.

$$
\begin{array}{ll}
\mathit{cards} & :: \mathit{Canvas} \\
\mathit{cards} & = \mathit{foldl}\ (\&)\ \mathit{relax} \\
& \qquad [\mathit{cdraw}\ ((\mathit{cardX}\ n, \mathit{cardH} * (1 - n)) \\
& \qquad\qquad \text{--}\ (\mathit{cardX}\ n + \mathit{cardW}, \mathit{cardH} * (1 - n)) \\
& \qquad\qquad \text{--}\ (\mathit{cardX}\ n + \mathit{cardW}, -\mathit{cardH} * n) \\
& \qquad\qquad \text{--}\ (\mathit{cardX}\ n, -\mathit{cardH} * n)\ \text{--}\ \mathit{cycle}) \\
& \qquad |\, n \leftarrow [1\,..\,9]]
\end{array}
$$

Der Tisch ist genauso wie der Kartenstapel als Canvasgrafik realisiert.

$$
\begin{array}{ll}
\mathit{table} & :: \mathit{Canvas} \\
\mathit{table} & = \mathit{cdraw}\ ((-330, -9 * \mathit{cardH}) \\
& \qquad \text{--}\ (\mathit{cardW} + \mathit{cardX}\ 9, -9 * \mathit{cardH}) \\
& \qquad \text{--}\ (\mathit{cardW} + \mathit{cardX}\ 9, -14 * \mathit{cardH}) \\
& \qquad \#\ \mathit{setPen}\ 1\ \#\ \mathit{setColor}\ 0.6) \\
& \&\ \mathit{cdrop}\ (-230, -12.0 * \mathit{cardH})\ \texttt{"table"}
\end{array}
$$

Die folgende Funktion ist allgemein für Bemaßungen nützlich. An den Enden befinden sich Pfeile, deren Spreizwinkel 180 Grad beträgt. In der Mitte der Linie ist ein Textlabel mit weißem Hintergrund plaziert.

$$
\begin{array}{ll}
\mathit{dimension} & :: (\mathit{IsPath}\ b, \mathit{IsPath}\ a) \Rightarrow a \to b \to \mathit{String} \to \mathit{Path} \\
\mathit{dimension}\ a\ b\ s & = a\ \text{--}\ b \\
& \quad \#\mathit{setArrowHead}\ \mathit{marker} \\
& \quad \#\mathit{setStartArrowHead}\ \mathit{marker} \\
& \quad \#\mathit{setLabel}\ 0.5\ C\ (\mathit{math}\ s\ \#\ \mathit{setBGColor}\ \mathit{white}) \\
\quad \textbf{where} & \\
\quad \mathit{marker} & = \mathit{arrowHeadSize}\ 3\ 180\ \#\ \mathit{setArrowHeadStyle}\ \mathit{ahLine}
\end{array}
$$

Auch die Bemaßungen konstruieren wir als Canvasgrafik unter Zuhilfenahme der Funktion *cardX*.

$$
\begin{array}{ll}
\mathit{dimensions} & :: \mathit{Canvas} \\
\mathit{dimensions} & = \mathit{cdraw}\ (\mathit{dimension}\ (\mathit{cardW} + \mathit{cardX}\ 2, -4 * \mathit{cardH}) \\
& \qquad\qquad (0, -4 * \mathit{cardH})\ \texttt{"d\_2"}) \\
& \&\ \mathit{cdraw}\ (\mathit{dimension}\ (\mathit{cardW} + \mathit{cardX}\ 3, -6 * \mathit{cardH}) \\
& \qquad\qquad (0, -6 * \mathit{cardH})\ \texttt{"d\_3"}) \\
& \&\ \mathit{cdraw}\ (\mathit{dimension}\ (\mathit{cardW} + \mathit{cardX}\ 9, -11 * \mathit{cardH}) \\
& \qquad\qquad (0, -11 * \mathit{cardH})\ \texttt{"d\_\{n+1\}"})
\end{array}
$$

Auch für die häufig benötigte Funktionalität eines beschreibenden Textes von dem ein Pfeil ausgeht erstellen wir eine Funktion.

$$
\begin{array}{ll}
\mathit{description} & :: (\mathit{IsPicture}\ c, \mathit{IsPath}\ a, \mathit{IsPath}\ b) \\
& \Rightarrow a \to b \to c \to \mathit{Path} \\
\mathit{description}\ p_1\ p_2\ l & = \mathit{arrow}\ p_1\ p_2 \\
& \quad \#\mathit{setLabel}\ 0\ C\ (\mathit{toPicture}\ l\ \#\ \mathit{setBGColor}\ \mathit{white})
\end{array}
$$

Die Variante für unseren Fall mit speziellem Text und einem konstanten Pfeilvektor formulieren wir nochmals separat.

$$\begin{array}{lll}
\mathit{describeCard} & :: & \mathit{Point} \to \mathit{String} \to \mathit{Path} \\
\mathit{describeCard}\ p\ l & = & \mathit{description}\ (p + \mathit{vec}\ (-30, 15))\ p\ (\texttt{"card␣"} +\!\!+\ l)
\end{array}$$

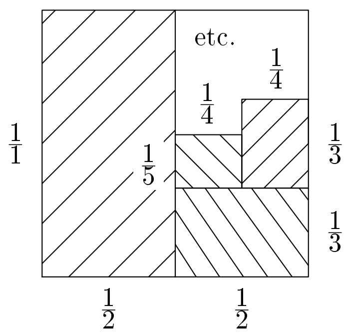
Das fertige Bild besteht aus einer Canvasgrafik, die den Kartenstapel, den Tisch und die Bemaßungen enthält und den darauf gezeichneten Pfeilen mit den Beschreibungen.

$$\begin{array}{l}
\mathit{draw}\ [\mathit{describeCard}\ (\mathit{vec}\ (\mathit{cardX}\ 1, 0))\ \texttt{"\$1\$"}, \\
\qquad \mathit{describeCard}\ (\mathit{vec}\ (\mathit{cardX}\ 2, -\mathit{cardH}))\ \texttt{"\$2\$"}, \\
\qquad \mathit{describeCard}\ (\mathit{vec}\ (\mathit{cardX}\ 9, -8 * \mathit{cardH}))\ \texttt{"\$n\$"}] \\
\quad (\mathit{cards}\ \&\ \mathit{table}\ \&\ \mathit{dimensions})
\end{array}$$

In diesem Beispiel kann man schön sehen, wie *functional* METAPOST von der Einbettung in Haskell profitiert. Werte, wie hier die harmonische Reihe, können leicht in Haskell berechnet und dann direkt in der Bildbeschreibung verwendet werden. Die Größe der einzelnen Karten ist auch nachträglich variabel. Mit einem Zeichenprogramm wäre diese Grafik nicht so einfach zu erstellen, jedenfalls wenn die Abstände maßstabsgerecht sein sollen.

Beispiel 2

Auf Seite 66 des Buches [GKP92] befindet sich ein Bild zu dem Problem, das Einheitsquadrat mit Rechtecken der Seitenlängen $\frac{1}{1} \times \frac{1}{2}$, $\frac{1}{2} \times \frac{1}{3}$, $\frac{1}{3} \times \frac{1}{4}$, … zu füllen.
PostScript und damit auch METAPOSTunterstützt leider nicht direkt Schraffuren oder sonstige Füllmuster für Flächen. Das Buch [Ado88] gibt zu diesem Problem den Lösungsvorschlag, das Füllmuster in einen rechteckigen Bereich, der die gewünschte Fläche enthält, zu zeichnen und dann mit Clipping die Fläche „auszustanzen“. Dieser Weg läßt sich auch genauso mit *functional* METAPOST verwirklichen.



Wir definieren eine Funktion *patternBox*, die ein schraffiertes Rechteck zeichnet. Die ersten beiden Parameter geben die Koordinaten des linken unteren und des rechten oberen Punktes an. Die nächsten beiden Parameter sind Funktionen, die die Richtung der Schraffur beeinflussen.

$$\begin{array}{lll}
\mathit{patternBox} & :: & (\mathit{Numeric}, \mathit{Numeric}) \to (\mathit{Numeric}, \mathit{Numeric}) \\
 & \to & (\mathit{Numeric} \to (\mathit{Numeric}, \mathit{Numeric})) \\
 & \to & (\mathit{Numeric} \to (\mathit{Numeric}, \mathit{Numeric})) \to \mathit{Canvas} \\
\multicolumn{3}{l}{\mathit{patternBox}\ (ax, ay)\ (bx, by)\ \mathit{fa}\ \mathit{fb}} \\
 & = & \mathit{cdrop}\ (0.5 * (ax + bx), 0.5 * (ay + by))
\end{array}$$

$$
\begin{array}{lll}
 & & (cdraws\ [fa\ i\ \text{--}\ fb\ i \mid i \leftarrow [-50, -40\, ..\, 200]] \\
 & & \&cclip\ p \\
 & & \&cdraw\ p) \\
\textbf{where} & & \\
p & = & vec\ (ax, ay)\ \text{--}\ vec\ (bx, ay)\ \text{--}\ vec\ (bx, by) \\
 & & \text{--}vec\ (ax, by)\ \text{--}\ cycle
\end{array}
$$

Die vielen Brüche sollen in einer etwas größeren Schriftart als normal erscheinen. Ihr Hintergrund muß weiß sein, damit die Schraffur um den Bruch $\frac{1}{5}$ herum verläuft.

$$
\begin{array}{lll}
frac & :: & Show\ a \Rightarrow (Numeric, Numeric) \rightarrow a \rightarrow Canvas \\
frac\ p\ n & = & cdrop\ p\ (math\ (\texttt{"\\\\frac\{\\\\textstyle␣1\}"} \\
 & & \qquad\qquad +\!\!+ \texttt{"\{\\\\textstyle␣"} +\!\!+ show\ n +\!\!+ \texttt{"\}"}) \\
 & & \qquad \#\ setBGColor\ white)
\end{array}
$$

$$
\begin{array}{llll}
 & patternBox\ (0,0)\ (50,100) & (\lambda i \rightarrow (0, 200 - i * 1.5)) & (\lambda i \rightarrow (i * 1.5, 200)) \\
\& & patternBox\ (50,0)\ (100,33.3) & (\lambda i \rightarrow (0, i)) & (\lambda i \rightarrow (i, -50)) \\
\& & patternBox\ (75,33.3)\ (100,66.6) & (\lambda i \rightarrow (0, 100 - i)) & (\lambda i \rightarrow (i, 100)) \\
\& & patternBox\ (50,33.3)\ (75,53.3) & (\lambda i \rightarrow (0, i)) & (\lambda i \rightarrow (i, 0)) \\
\& & \multicolumn{2}{l}{cdraw\ (vec\ (50,100)\ \text{--}\ vec\ (100,100)\ \text{--}\ vec\ (100,50))} & \text{-- upper right corner} \\
\& & frac\ (-10,50)\ 1 & & \\
\& & frac\ (25,-12)\ 2\ \&\ frac\ (75,-12)\ 2 & & \\
\& & frac\ (110,17)\ 3\ \&\ frac\ (110,50)\ 3 & & \\
\& & frac\ (62,65)\ 4\ \&\ frac\ (88,78)\ 4 & & \\
\& & frac\ (40,42)\ 5 & & \\
\& & cdrop\ (65,90)\ \texttt{"etc."} & &
\end{array}
$$

Wie wir gesehen haben, lassen sich mit Hilfe von Clipping mehr Effekte erzielen, als es auf den ersten Blick scheinen mag. Trotzdem haben wir dieses Bild nicht sehr deskriptiv beschrieben, denn es enthält viele absolute Maßangaben. Aber in manchen Fällen kann dies die einfachste Lösung sein.

Beispiel 3

Gerade im Bereich der Artikel über Datenstrukturen werden oft spezielle Baumtypen benötigt. Wir wollen hier deshalb exemplarisch Abstraktionen für zwei–drei–vier–Bäume und rot–schwarz–Bäume definieren wie sie z.B. im Artikel [Oka98] vorkommen. Gleichzeitig ist dies ein Beispiel dafür, wie man den Verlauf von Baumkanten an eigene Bedürfnisse anpassen kann.
Beim Entwurf der Datenstruktur richten wir uns nach der nach der Definition von zwei–drei–vier–Bäumen. Ein Knoten enthält keine, zwei, drei oder vier Elemente.

$$
\begin{array}{lll}
\textbf{data}\ Tree234\ a & = & Nil \\
 & \mid & Two\ (Tree234\ a)\ a\ (Tree234\ a) \\
 & \mid & Three\ (Tree234\ a)\ a\ (Tree234\ a)\ a\ (Tree234\ a) \\
 & \mid & Four\ (Tree234\ a)\ a\ (Tree234\ a)\ a\ (Tree234\ a)\ a\ (Tree234\ a) \\
 & & \textbf{deriving}\ (Show)
\end{array}
$$

Mit dieser Datenstruktur lassen sich die Bäume aus Abbildung 4.4.1 wie folgt notieren.

$$
rbtree \quad :: \quad Tree234\ String
$$

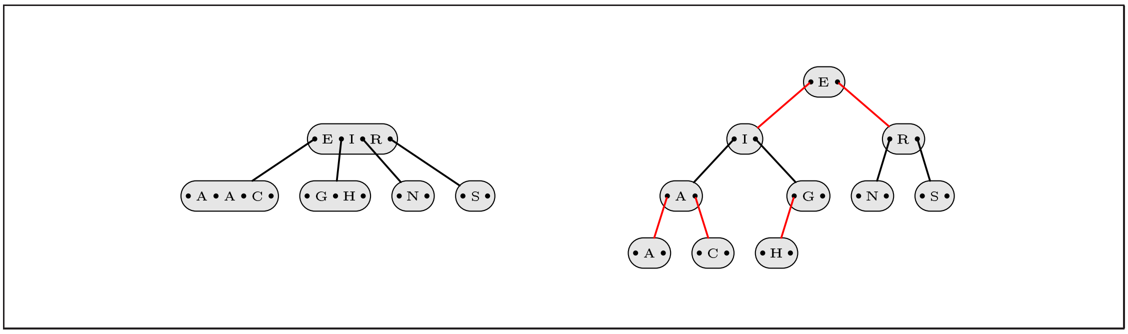


Abbildung 4.4.1: Die gleiche Information als zwei–drei–vier–Baum und als rot–schwarz–Baum dargestellt.

rbtree = *Four* (*Four Nil* `"A"` *Nil* `"A"` *Nil* `"C"` *Nil*)
`"E"`
(*Three Nil* `"G"` *Nil* `"H"` *Nil*)
`"I"`
(*Two Nil* `"N"` *Nil*)
`"R"`
(*Two Nil* `"S"` *Nil*)

Wir wollen nun zwei kleine Funktionen entwerfen, die einen Baum des Typs *Tree234 String* in einen Baum des Typs *Tree*, konvertieren; entweder im zwei–drei–vier– oder im rot–schwarz–Layout.
Die Funktion zum Erzeugen kleinen Textes können wir für beide Konverter einsetzen.

tiny :: *String* → *Picture*
tiny a = *tex* (`"\\tiny␣"` ++ *a*)

Das Besondere an den beiden Baumarten ist die Tatsache, daß die Kanten nicht von den Knotenzentren, sondern verschiedenen Punkten ausgehen. Das stellt für die Baumbeschreibungssprache aber kein Problem dar. Kanten müssen nicht unbedingt zwischen den Zentren der Knotenbilder verlaufen. Wir müssen die gewünschten Punkte nur benennen können.

dotName :: *String* → *Frame*
dotName n = *dot* # *setName n*

Die Knotenbilder bestehen aus abgerundeten Rechtecken mit entsprechendem Inhalt.

tbox :: *String* → *Frame*
tbox s = *rbox* 8 (*dotName* `"p1"` ||| *tiny s* ||| *dotName* `"p2"`)
tbox2 s_1 s_2 = *rbox* 8 (*dotName* `"p1"` ||| *tiny* s_1 ||| *dotName* `"p2"`
||| *tiny* s_2 ||| *dotName* `"p3"`)
tbox3 s_1 s_2 s_3 = *rbox* 8 (*dotName* `"p1"` ||| *tiny* s_1 ||| *dotName* `"p2"`
||| *tiny* s_2 ||| *dotName* `"p3"` ||| *tiny* s_3
||| *dotName* `"p4"`)

Wir können nun spezielle Kanten definieren, die vom aktuellen Sohn zum Zentrum des Bildes mit den Namen `'p'` : *show n* des Vaters verlaufen. Dabei zeigt sich die Nützlichkeit, Namen aus verschiedenen Bestandteilen bilden zu können. Die speziellen Namen *This* und *Parent* sind Platzhalter und ändern ihre Referenz im Kontext einer bestimmten Kante. Dies erlaubt es, eine Kante allgemein zu beschreiben, ohne genaue Namen zu verwenden. Anhang A.16.3 erläutert dies genauer.

$$
\begin{array}{lll}
\textit{edgeN} & :: & \textit{Int} \to \textit{Tree} \to \textit{Edge} \\
\textit{edgeN}\ n & = & \textit{edge}'\ (\textit{ref}\ (\textit{This} \triangleleft C)\ \text{--}\ \textit{ref}\ (\textit{Parent} \triangleleft \texttt{'p'} : \textit{show}\ n \triangleleft C))
\end{array}
$$

Bei der Konvertierung eines Knotens ist es wichtig, die richtige Kante entsprechend der Nummer des Sohnes zu wählen.

$$
\begin{array}{lll}
\textit{convert234} & :: & \textit{Tree234 String} \to \textit{Tree} \\
\textit{convert234 Nil} & = & \textit{node}\ (\textit{box empty})\ [\,] \\
\textit{convert234}\ (\textit{Two}\ t_1\ a\ t_2) & = & \textit{node}\ (\textit{tbox}\ a)\ (\textit{edge234}\ 1\ t_1\ (\textit{edge234}\ 2\ t_2\ [\,])) \\
\textit{convert234}\ (\textit{Three}\ t_1\ a_1\ t_2\ a_2\ t_3) & & \\
 & = & \textit{node}\ (\textit{tbox2}\ a_1\ a_2)\ (\textit{edge234}\ 1\ t_1 \\
 & & \qquad (\textit{edge234}\ 2\ t_2 \\
 & & \qquad (\textit{edge234}\ 3\ t_3\ [\,]))) \\
\textit{convert234}\ (\textit{Four}\ t_1\ a_1\ t_2\ a_2\ t_3\ a_3\ t_4) & & \\
 & = & \ldots \\
 & & \\
\textit{edge234} & :: & \textit{Int} \to \textit{Tree234 String} \to [\textit{Edge}] \to [\textit{Edge}] \\
\textit{edge234}\ n\ \textit{Nil cont} & = & \textit{cont} \\
\textit{edge234}\ n\ t\ \textit{cont} & = & \textit{edgeN}\ n\ (\textit{convert234}\ t) : \textit{cont}
\end{array}
$$

Bei der Konvertierung in der rot–schwarz–Darstellung wird ein Knoten eventuell auf mehrere Knoten eines Binärbaumes abgebildet. Im Falle eines *Three*–Knotens ist darauf zu achten, daß der linke Sohn auch nach links ausgerichtet wird.

$$
\begin{array}{lll}
\textit{convertRB} & :: & \textit{Tree234 String} \to \textit{Tree} \\
\textit{convertRB Nil} & = & \textit{node}\ (\textit{box empty})\ [\,] \\
\textit{convertRB}\ (\textit{Two}\ t_1\ a\ t_2) & = & \textit{node}\ (\textit{tbox}\ a)\ (\textit{edgeRB}\ 1\ t_1\ (\textit{edgeRB}\ 2\ t_2\ [\,])) \\
\textit{convertRB}\ (\textit{Three}\ t_1\ a_1\ t_2\ a_2\ t_3) & & \\
 & = & \textit{node}\ (\textit{tbox}\ a_1)\ (\textit{map}\ (\textit{setColor red}) \\
 & & \qquad\qquad (\textit{edgeRB}\ 1\ (\textit{Two}\ t_1\ a_2\ t_2) \\
 & & \qquad (\textit{edgeRB}\ 2\ t_3\ [\,])))\ \#\ \textit{setAlign alignLeftSon} \\
\textit{convertRB}\ (\textit{Four}\ t_1\ a_1\ t_2\ a_2\ t_3\ a_3\ t_4) & & \\
 & = & \textit{node}\ (\textit{tbox}\ a_1) \\
 & & \quad (\textit{map}\ (\textit{setColor red})\ (\textit{edgeRB}\ 1\ (\textit{Two}\ t_1\ a_2\ t_2) \\
 & & \qquad\qquad (\textit{edgeRB}\ 2\ (\textit{Two}\ t_3\ a_3\ t_4)\ [\,])))
\end{array}
$$

Wir wollen in beiden Bäumen eine etwas dickere Strichstärke der Kanten und für die Knotenbilder eine Hintergrundschattierung erreichen. Hierfür sind die Funktionen *forEachPic* und *forEachEdge* nützlich, die eine Funktion auf die Knotenbilder bzw. Kanten eines Baumes fortsetzen.

$$
\begin{array}{ll}
 & \textit{forEachPic}\ (\textit{setBGColor}\ 0.9)\ (\textit{forEachEdge}\ (\textit{setPen}\ 0.75)\ (\textit{convert234 rbtree})) \\
[\!|\!] & \textit{hspace}\ 50 \\
[\!|\!] & \textit{forEachPic}\ (\textit{setBGColor}\ 0.9)\ (\textit{forEachEdge}\ (\textit{setPen}\ 0.75)\ (\textit{convertRB rbtree}))
\end{array}
$$

Beispiel 4

Schließlich soll dieses letzte Beispiel noch einmal die Nützlichkeit und Leistungsfähigkeit von Gleichungssystemen demonstrieren. Die Aufgabe besteht darin, eine geschwungene Klammer zwischen zwei Punkte zu zeichnen, wobei ein Bild als Label angegeben werden kann. Für die eigentliche Klammer soll ein kalligraphischer Stift verwendet werden, damit die Enden spitz und die Geraden dicker

sind. Dazu ist der Stift im passenden Winkel zu drehen. Eine weitere Schwierigkeit ist die richtige Plazierung des Labels. Obwohl dieses eine beliebige Größe haben kann, soll es so an der Spitze in der Mitte der Klammer ausgerichtet sein, daß die Spitze auf das Label zeigt. In Abbildung 4.4.2 sehen wir, wie sich die Ausrichtung des Labels an die verschiedenen Winkel, in denen die Klammer auftritt, anpaßt und wie die Strichstärke bei kleinen Klammern abnimmt.
Wir beginnen mit dem Ausdruck, der den Pfad der Klammer beschreibt. Mit Hilfe der Funktion *define* legen wir in einem Gleichungssystem die Positionen der fünf Punkte fest, durch die der Pfad verläuft. Die Variable *var* `"angle"` gibt den Winkel, den die Punkte *pl* und *pr* bilden und zwischen denen die Klammer verläuft, an. Die Breite, *var* `"d"` des Stiftes, beträgt 5 bp oder wenn der Abstand der Punkte *pl* und *pr* kleiner als 20 bp ist, ein Viertel dieses Wertes. Aus diese Weise wirken kleine Klammern nicht klobig, sondern filligran.
Was wir für die Positionierung des Labels brauchen, ist die vom Winkel *var* `"angle"` abhängige Ausrichtung. Leider können wir diese Bedingung aber nicht innerhalb des Pfades formulieren. Deshalb plazieren wir ein verändertes Label am Bezugspunkt *C*, in der wir den Bezugspunkt *C* entsprechend an die Ränder verschieben.
Zu diesem Zweck benutzen wir die Funktion *overlay*$'$, die eine Wahl der Bounding Box für das kombinierte Bild erlaubt. Wir kombinieren das leere Bild mit dem des Labels und wählen mit dem Parameter (Just 0) die Bounding Box des ersten Bildes, also die von *empty*. Die Funktion *label* erzeugt damit ein Bild, dessen Bezugspunkt *C* (und alle anderen Bezugspunkte ebenso), vom Winkel *var* `"angle"` abhängig, auf den Bezugspunkten *N*, *NE* bis *NW* des Bildes *l* liegt.
Auf diese Weise haben wir eine Funktion für einen Pfad, in Gestalt einer geschwungenen Klammer, mit einer gewissen „Intelligenz“ erhalten. Der Anwender kann diese Abstraktion wie alle anderen Pfade einsetzen und muß sich nicht um die Plazierung des Labels oder die Wahl der Strichstärke kümmern.

```
let bracket :: IsPicture a ⇒ a → (Point, Point) → Path
    bracket l (pl, pr) = define [var "angle" ≐ angle (pl − pr),
                                 var "d" ≐ cond (dist pl pr <̇ 20) (dist pl pr / 4) 5,
                                 ref "vecl" ≐ var "d" *̇ dir (var "angle" − 135),
                                 ref "vecr" ≐ var "d" *̇ dir (var "angle" − 45),
                                 ref "mid" ≐ med 0.5 pl pr
                                             + (1.41 ∗ var "d") *̇ dir (var "angle" − 90),
                                 ref "midl" ≐ ref "mid" − ref "vecl",
                                 ref "midr" ≐ ref "mid" − ref "vecr"]
                                (pl ... pl + ref "vecl" --- ref "midl" ... ref "mid"
                                 & ref "mid" ... ref "midr" --- pr + ref "vecr" ... pr
                                 # setPen (penCircle (0.001, var "d" / 5) (var "angle"))
                                 # setLabel 0.5 C label)
      where
      label = overlay′ [var "angle" ≐ angle (pl − pr),
                        ref (0 ◃ C) ≐ cond (var "angle" <̇ −175.5
                                            + 175.5 <̇ var "angle") (ref (1 ◃ S))
                                       (cond (var "angle" <̇ −112.5) (ref (1 ◃ SE))
                                        (cond (var "angle" <̇ −67.5) (ref (1 ◃ E))
                                         (cond (var "angle" <̇ −22.5) (ref (1 ◃ NE))
                                          (cond (var "angle" <̇ 22.5) (ref (1 ◃ N))
                                           (cond (var "angle" <̇ 67.5) (ref (1 ◃ NW))
                                            (cond (var "angle" <̇ 112.5) (ref (1 ◃ W))
                                                                         (ref (1 ◃ SW))
                                       ))))))] (Just 0) [empty, toPicture l]
in [bracket a (5 *̇ dir a, 80 *̇ dir a) | a ← [0, 45 .. 315]]
  ++ [bracket y (vec (x, 0), vec (x, y)) | (x, y) ← zip [0, 30 ..] (5 : 10 : 15 : [20, 30 .. 60])]
```

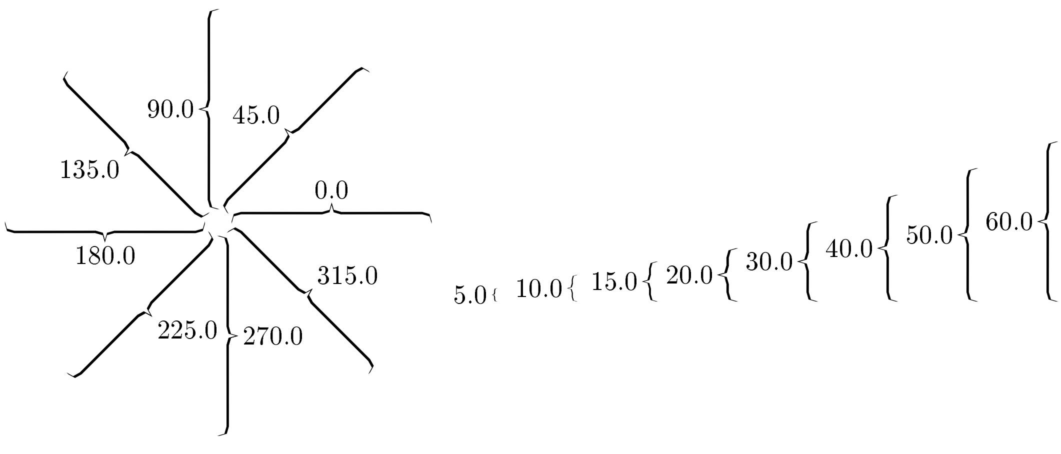


Abbildung 4.4.2: Zwischen den beiden Punkten *pl* und *pr* wird ein Pfad in Gestalt einer Klammer erzeugt und das Bild *l* als Label daran plaziert.

Kapitel 5

Implementierung der Erweiterungen

Beim Entwurf von *functional* METAPOST waren eine möglichst große Universalität und Allgemeinheit Designkriterien. Wir haben bewußt darauf verzichtet, die Erweiterungen direkt in die Kernsprache einzubauen und stattdessen eine Sprache vorgestellt, die so leistungsfähig ist, daß wir Erweiterungen mit ihren Sprachmitteln beschreiben können. Der Aufwand, der dazu nötig ist, ist als Maßstab dafür anzusehen, wie erfolgreich dieser Ansatz ist.

Dieses Kapitel beschreibt die Implementierung der Erweiterungen Canvasgrafik, Turtlegrafik und Bäume, die in Kapitel 4 vorgestellt wurden. Im Kapitel 3 haben wir uns mit der eigentlichen Kernsprache von *functional* METAPOST beschäftigt und damit alle nötigen Kenntnisse zur Implementierung der Erweiterungen erworben.

Unser allgemeines Vorgehen bei der Implementierung der kleinen Spezialsprache einer Erweiterung wird folgendes sein: Wir bilden die Operationen und Anweisungen auf Konstruktoren eines Datentyps ab. Diesen Datentyp machen wir zu einem Untertyp von *Picture*, indem wir für ihn eine Instanz der Klasse *IsPicture* definieren. Die Funktion *toPicture* bildet einen Interpreter für den Datentyp der Spezialsprache und wandelt Ausdrücke dieses Datentyps in ein Bild um.

Warum bilden wir die Befehle nicht direkt auf ein Bild ab, sondern gehen den Umweg über einen Datentyp, der von einem Interpreter schließlich auch in ein Bild umgewandelt wird? Dafür gibt es folgende Gründe: Ein Interpreter ermöglicht es, einen Zustand zu verwalten, der durch die Operationen der Spezialsprache verändert wird. Mit anderen Worten sind die Operationen der Sprache kontextsensitiv sind und der Interpreter muß ganzen Kontext kennen. Dies wird bei der Turtlegrafik besonders deutlich, bei der die Wirkung der Anweisung *forward n* von der Orientierung und Position des Turtles abhängt, die sich aus den vorangegangenen Anweisungen ergibt.

Der Umweg über einen Datentyp ermöglicht es erst, Attribute persistent zu verwalten und vom Typ *Picture* abweichende Attributierungsfunktionen anzubieten.

5.1 Canvasgrafik

Bei Canvasgrafik handelt es sich um eine kleine, leicht zu implementierende Anwendung, an der wir uns unser Vorgehen gut verdeutlichen können. Der Datentyp zur Canvasgrafik enthält Konstruktoren, auf die wir alle Befehle der kleinen Spezialsprache abbilden können. Spezielle Attribute müssen wir hier nicht speichern.

$$
\begin{array}{lll}
\textbf{data}\ \mathit{Canvas} & = & \mathit{CDraw}\ [\mathit{Path}] \\
 & | & \mathit{CClip}\ \mathit{Path} \\
 & | & \mathit{CFill}\ [\mathit{Area}] \\
 & | & \mathit{CDrop}\ (\mathit{Numeric}, \mathit{Numeric})\ \mathit{Picture} \\
 & | & \mathit{CConcat}\ \mathit{Canvas}\ \mathit{Canvas} \\
 & | & \mathit{CRelax} \\
 & & \textbf{deriving}\ \mathit{Show}
\end{array}
$$

Der Typ *Canvas* ist Instanz der Typklassen *HasConcat* und *HasRelax*, was den in *functional* META-POST üblichen Operator der Konkatenation und den Ausdruck *relax* erschließt.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{instance}\ \mathit{HasConcat}\ \mathit{Canvas}\ \textbf{where}} \\
\quad (\&) & = & \mathit{CConcat}
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{instance}\ \mathit{HasRelax}\ \mathit{Canvas}\ \textbf{where}} \\
\quad \mathit{relax} & = & \mathit{CRelax}
\end{array}
$$

Die Abbildung der Zeichenbefehle auf die Konstruktoren ist trivial. Wir geben nur ein Beispiel, da *cfills*, *cclip* und *cdrop* analog definiert sind.

$$
\begin{array}{lll}
\mathit{cdraws} & :: & \mathit{IsPath}\ a \Rightarrow [a] \to \mathit{Canvas} \\
\mathit{cdraws} & = & \mathit{CDraw} \circ \mathit{map}\ \mathit{toPath}
\end{array}
$$

Interessant ist die Umwandlung vom Datentyp *Canvas* nach *Picture*, die in der Funktion *toPicture* stattfindet. In der Kernsprache lassen sich Linien und Flächen nur zu einem bestehenden Bild hinzufügen. Deshalb muß eine kaskadierende Folge von Bildern erzeugt werden, die mit dem leeren Bild beginnt. Die Sequenz der Canvas–Befehle muß umgekehrt werden, damit die zuletzt angegebenen Bilder auch zuletzt gezeichnet werden. [1]
Die Funktion *setTrueBoundingBox* ermittelt die tatsächlichen Ausmaße des erzeugten Bildes und macht die Objekte, die absolut positioniert wurden verschiebbar.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{instance}\ \mathit{IsPicture}\ \mathit{Canvas}\ \textbf{where}} \\
\quad \mathit{toPicture}\ p & = & \mathit{setTrueBoundingBox}\ (\mathit{foldr}\ \mathit{canvas2Pic}\ \mathit{empty} \\
 & & \qquad\qquad (\mathit{reverse}\ (\mathit{flatten}\ p\ [\,])))
\end{array}
$$

Die Funktion *canvas2Pic* fügt, für ein einzelnes Canvasobjekt, den entsprechenden Zeichenbefehl der Kernsprache zu einem Bild hinzu. Erläuterung bedarf wohl nur die Behandlung des Musters $\mathit{CanvasDrop}\ \mathit{pos}\ p'$. Es gibt nur eine Gleichung für den Befehl $\mathit{overlay}'$, die das übergebene Bild mit dem Zentrum auf den gewünschten Punkt festlegt. Der optionale zweite Parameter von $\mathit{overlay}'$ gibt an, von welchem Bild die neue Bounding Box stammt. Die Festlegung auf das erste Bild erfolgt nur um die Berechnung des minimal umschließenden Rechtecks einzusparen. Die korrekte Bounding Box der Canvasgrafik wird zuletzt sowieso von *setTrueBoundingBox* garantiert.

[1] Die Wichtigkeit der Zeichenreihenfolge ergibt sich einzig und allein durch die Möglichkeit, mit verschiedenen Farben zu zeichnen. Wenn alle Zeichenbefehle nur schwarze Spuren auf der Zeichenfläche hinterließen, spielte die Zeichenreihenfolge keine Rolle.

$$
\begin{array}{lcl}
\textit{canvas2Pic} & :: & \textit{Canvas} \rightarrow \textit{Picture} \rightarrow \textit{Picture} \\
\textit{canvas2Pic}\ (\textit{CDraw ps}) & = & \textit{draw ps} \\
\textit{canvas2Pic}\ (\textit{CFill as}) & = & \textit{fill}\ (\textit{map setFront as}) \\
\textit{canvas2Pic}\ (\textit{CClip p}) & = & \textit{clip p} \\
\textit{canvas2Pic}\ (\textit{CDrop pos } p') & = & \lambda p \rightarrow \textit{overlay}\ [\textit{ref}\ (1 \triangleleft C) \doteq \textit{vec pos}]\ [p, p'] \\
\textit{canvas2Pic}\ \_ & = & \textit{id}
\end{array}
$$

Um aus einer Sequenz von Canvasobjekten eine Liste zu erhalten, muß die Baumstruktur, die sich aus den Konstruktoren *CConcat* ergibt, inorder durchlaufen werden.

$$
\begin{array}{lcl}
\textit{flatten} & :: & \textit{Canvas} \rightarrow [\textit{Canvas}] \rightarrow [\textit{Canvas}] \\
\textit{flatten}\ (\textit{CConcat}\ c_1\ c_2)\ \textit{cs} & = & \textit{flatten}\ c_1\ (\textit{flatten}\ c_2\ \textit{cs}) \\
\textit{flatten}\ c\ \textit{cs} & = & c : \textit{cs}
\end{array}
$$

Dies genügt schon für die Definition einer nützlichen Erweiterung. Im nächsten Abschnitt steigert sich der Schwierigkeitsgrad etwas.

5.2 Turtlegrafik

Für die Sprache dieser Anwendung werden wir einen etwas komplizierteren Interpreter entwerfen müssen. Ein Turtlepfad wird sequentiell abgearbeitet, wobei sich der Zustand des Stiftes ändert. Neben der sequentiellen Abarbeitung mit Verwaltung eines Zustandes müssen Attributierungen, die auf einen Ausschnitt des Turtlepfades wirken, ebenfalls effizient berücksichtigt werden.
Zu Beginn definieren wir wieder den Datentyp, hier *Turtle*, wobei jeder Befehl seine Entsprechung in einem Konstruktor findet.

$$
\begin{array}{lcl}
\textbf{data}\ \textit{Turtle} & = & \textit{TConc Turtle Turtle} \\
 & | & \textit{TDropPic Picture} \\
 & | & \textit{TColor Color Turtle} \\
 & | & \textit{TPen Pen Turtle} \\
 & | & \textit{THide Turtle} \\
 & | & \textit{TForward Numeric} \\
 & | & \textit{TTurn Numeric} \\
 & | & \textit{TPenUp} \\
 & | & \textit{TPenDown} \\
 & | & \textit{THome} \\
 & | & \textit{TFork Turtle Turtle} \\
 & & \textbf{deriving}\ \textit{Show}
\end{array}
$$

Der Konkatenationsoperator und der Befehl *relax* soll auch auf Turtlepfade anwendbar sein. Also ist *Turtle* Instanz der Klassen *HasConcat* und *HasRelax*.

$$
\begin{array}{lcl}
\textbf{instance}\ \textit{HasConcat Turtle}\ \textbf{where} & & \\
\quad (\&) & = & \textit{TConc}
\end{array}
$$

$$
\begin{array}{lcl}
\textbf{instance}\ \textit{HasRelax Turtle}\ \textbf{where} & & \\
\quad \textit{relax} & = & \textit{TTurn}\ 0.0
\end{array}
$$

Die Klasse *HasPicture* stellt mit *fromPicture* eine Funktion zum Einfügen eines Bildes an der aktuellen Turtleposition bereit. Der Interpreter hat dafür Sorge zu tragen, daß diese Bilder an den entsprechenden Positionen gezeichnet werden.

$$
\begin{array}{ll}
\textbf{instance } \textit{HasPicture Turtle } \textbf{where} & \\
\quad \textit{fromPicture} & = \textit{TDropPic} \circ \textit{toPicture}
\end{array}
$$

Weiterhin sollen die Attribute für Farbe und Stift von Pfaden auch für Turtlepfade gelten. Ein Problem, das in gleicher Weise für normale Pfade gilt, ist die Bedeutung der *get* – Funktionen. Die abgefragten Attribute müssen nicht homogen für den gesamten Turtlepfad gelten. Teilpfade können z.B. verschiedene Farben haben. Das Ergebnis der *get* – Funktionen kann also nur Sinn ergeben, wenn das entsprechende Attribut auf dem ganzen Pfad den gleichen Wert besitzt.

$$
\begin{array}{ll}
\textbf{instance } \textit{HasColor Turtle } \textbf{where} & \\
\quad \textit{setColor} & = \textit{TColor} \\
\quad \textit{setDefaultColor} & = \textit{setColor DefaultColor} \\
\quad \textit{getColor } (\textit{TColor } c \; \_) & = c \\
\quad \textit{getColor } \_ & = \textit{DefaultColor}
\end{array}
$$

Die Befehle zur Steuerung des Turtles lassen sich einfach auf die Konstruktoren abbilden. Z.B. für die Funktion *forward*:

$$
\begin{array}{ll}
\textit{forward} & :: \textit{Numeric} \to \textit{Turtle} \\
\textit{forward} & = \textit{TForward}
\end{array}
$$

Die Überführung eines Turtlepfades in ein Bild ist etwas komplizierter. Die Pfade und eventuell darin enthaltene Bilder, die mit der Funktion *fromPicture* erzeugt sind, werden getrennt gezeichnet. Die Funktion *figure* bildet einen Turtlepfad auf eine Liste normaler Pfade und eine Liste eventuell eingefügter Bilder mit ihren Positionen ab. Mit Hilfe von Canvasgrafik werden zuerst die Pfade und danach die Bilder gezeichnet. Das ist sinnvoll, da es sich bei eingefügten Bildern meistens um Textlabel handeln dürfte, die nicht von den Pfaden überzeichnet werden sollten.

$$
\begin{array}{ll}
\textbf{instance } \textit{IsPicture Turtle } \textbf{where} & \\
\quad \textit{toPicture tp} & = \textit{toPicture } (\textit{cdraws paths} \,\&\, \textit{foldl } (\&) \textit{ relax cs}) \\
\qquad \textbf{where} & \\
\qquad \textit{cs} & = [\textit{cdrop pos pic} \mid (\textit{pos}, \textit{pic}) \leftarrow \textit{pics}] \\
\qquad (\textit{paths}, \textit{pics}) & = \textit{figure tp stdPathElemDescr}
\end{array}
$$

Für die Speicherung der Bildpositionen verwenden wir Ausdrücke des Typs

$$
\textbf{type } \textit{PicPos} \quad = ((\textit{Numeric}, \textit{Numeric}), \textit{Picture})
$$

Die Umwandlung eines Turtlepfades in Pfade ist in zwei Schritte aufgeteilt.

1. Zuerst verteilt die Funktion *spreadAttrib* die Farb– und Stiftinformationen auf die Teilpfade und linearisiert den Pfad. Die Verteilung der Attribute ist wichtig, damit der zweite Schritt, der den Pfad berechnet, für jeden Teilpfad die richtige Farbe und den Stift kennt.

   Der Typ *TurtleAttrib* speichert die Farb– und Stiftänderungen, der Funktionen *setColor* und *setPen*. Records vom Typ *PathElemDescr* speichern auch in normalen Pfaden die Attribute. Den Konstruktor *TAttribFork* benötigen wir, um mit *fork* erzeugte Verzweigungen als solche zu erhalten, denn in diesem Fall müssen wir erkennen daß mehrere Pfade zu erzeugen sind.

$$
\begin{array}{lll}
\textbf{data}\ \mathit{TurtleAttrib} & = & \mathit{TAttrib\ PathElemDescr\ Turtle} \\
 & \mid & \mathit{TAttribFork}\ [\mathit{TurtleAttrib}]\ [\mathit{TurtleAttrib}] \\
 & & \textbf{deriving}\ \mathit{Show}
\end{array}
$$

Ein Turtlepfad besitzt die Struktur eines binären Baumes. Die Knoten bildet der Konstruktor *TurtleConc*, die übrigen Konstruktoren sind Blätter. Abbildung 5.2.1 zeigt ein Beispiel. Wie dort zu sehen ist, wird ein Pfad, der auf einen *fork* – Befehl folgt, an beide mit *fork* erzeugten Pfade angehängt.

$$
\begin{array}{l}
\textbf{let}\ \{\mathit{forkT} = \mathit{fork}\ (\mathit{toleft}\ \&\ \mathit{fw}\ 20) \\
\qquad\qquad\qquad\qquad (\mathit{toright}\ \&\ \mathit{fw}\ 20); \\
\qquad \mathit{endPic} = \mathit{tex}\ \texttt{"end"} \\
\qquad\qquad\qquad \#\ \mathit{setBGColor\ white}; \\
\qquad \mathit{fw} = \mathit{forward}\} \\
\textbf{in}\ (\mathit{fw}\ 5\ \#\ \mathit{setPen}\ 2) \\
\qquad \&\ \mathit{forkT} \\
\qquad \&\ (\mathit{forkT}\ \#\ \mathit{setColor\ green}) \\
\qquad \&\ \mathit{fromPicture\ endPic}
\end{array}
$$

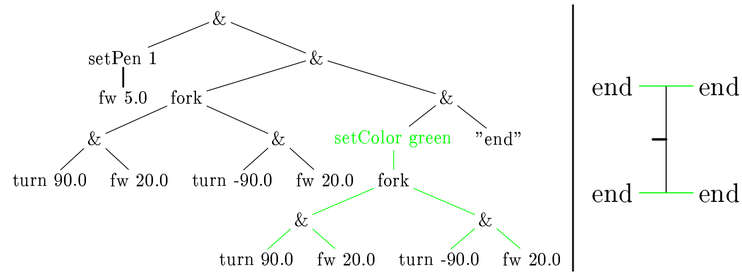
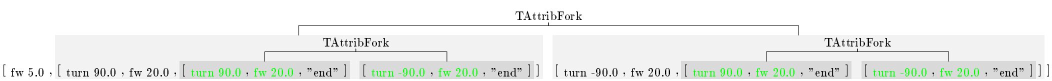


Abbildung 5.2.1: Ein Ausdruck, der einen Turtlepfad beschreibt, seine Darstellung als Baum und das berechnete Bild des Pfades.

Die Informationen über Farben und Stifte werden top–down verteilt. Verzweigungen von *fork* dürfen wie gesagt, nicht linearisiert werden, da diese Information noch im zweiten Schritt benötigt wird.

$$
\begin{array}{lll}
\mathit{spreadAttrib} & :: & \mathit{PathElemDescr} \rightarrow \mathit{Turtle} \rightarrow [\mathit{TurtleAttrib}] \\
 & \rightarrow & [\mathit{TurtleAttrib}] \\
\multicolumn{3}{l}{\mathit{spreadAttrib\ ped}\ (\mathit{TConc}\ p_1\ p_2)\ \mathit{ps}} \\
 & = & \mathit{spreadAttrib\ ped}\ p_1\ (\mathit{spreadAttrib\ ped}\ p_2\ \mathit{ps}) \\
\multicolumn{3}{l}{\mathit{spreadAttrib\ ped}\ (\mathit{TColor}\ c\ p)\ \mathit{ps}} \\
 & = & \mathit{spreadAttrib}\ (\mathit{setColor}\ c\ \mathit{ped})\ p\ \mathit{ps} \\
\multicolumn{3}{l}{\mathit{spreadAttrib\ ped}\ (\mathit{TPen\ pen}\ p)\ \mathit{ps}} \\
 & = & \mathit{spreadAttrib}\ (\mathit{setPen\ pen\ ped})\ p\ \mathit{ps} \\
\multicolumn{3}{l}{\mathit{spreadAttrib\ ped}\ (\mathit{THide}\ p)\ \mathit{ps}} \\
 & = & \mathit{spreadAttrib}\ (\mathit{hide\ ped})\ p\ \mathit{ps} \\
\multicolumn{3}{l}{\mathit{spreadAttrib\ ped}\ (\mathit{TFork}\ p_1\ p_2)\ \mathit{ps}} \\
 & = & [\mathit{TAttribFork}\ (\mathit{spreadAttrib\ ped}\ p_1\ \mathit{ps}) \\
 & & \qquad\qquad\quad (\mathit{spreadAttrib\ ped}\ p_2\ \mathit{ps})] \\
\mathit{spreadAttrib\ ped}\ p\ \mathit{ps} & = & \mathit{TAttrib\ ped}\ p : \mathit{ps}
\end{array}
$$

Wenn wir *spreadAttrib* auf unseren Beispielausdruck anwenden, erhalten wir die Liste:



2. Im zweiten Schritt interpretiert die Funktion *renderPath* die Liste aus Schritt 1. und berechnet daraus die Pfade. Der aktuelle Zustand des Turtles wird in einem Record gespeichert.

$$\begin{array}{lll} \textbf{data}\ \mathit{TurtleDescr} & = & \mathit{TurtleDescr}\{\mathit{tPos} :: (\mathit{Numeric}, \mathit{Numeric}), \\ & & \qquad\qquad\quad \mathit{tOrientation} :: \mathit{Numeric}, \\ & & \qquad\qquad\quad \mathit{tColor} :: \mathit{Maybe\ Color}, \\ & & \qquad\qquad\quad \mathit{tPen} :: \mathit{Maybe\ Pen}, \\ & & \qquad\qquad\quad \mathit{tPenDown} :: \mathit{Bool}\} \end{array}$$

Schauen wir uns die Funktion *renderPath* für die wichtigsten Konstruktoren genauer an. Die Variable *ped* enthält die Attributierungen aus dem ersten Schritt, die top–down erzeugt wurden. Die Variable *td* speichert dagegen die Informationen, die sich in einem linearen Lauf durch die Liste verändern.

Bei dem Befehl zur Drehung ist die Orientierung des Turtles um *d* zu inkrementieren.

$$\begin{array}{lll} \mathit{renderPath} & :: & \mathit{TurtleDescr} \to [\mathit{TurtleAttrib}] \to ([\mathit{Path}], [\mathit{PicPos}]) \\ & \to & ([\mathit{Path}], [\mathit{PicPos}]) \\ \multicolumn{3}{l}{\mathit{renderPath}\ \mathit{td}\ (\mathit{TAttrib}\ \_\ (\mathit{TTurn}\ d) : \mathit{ps})\ \mathit{tp}} \\ & = & \mathit{renderPath}\ \mathit{td}'\ \mathit{ps}\ \mathit{tp} \\ \quad \textbf{where} & & \\ \quad \mathit{td}' & = & \mathit{td}\{\mathit{tOrientation} = \mathit{tOrientation}\ \mathit{td} + d\} \end{array}$$

Bei einer Vorwärtsbewegung ergibt sich die neue Position des Turtles durch Verschiebung der alten Position um *d* Schritte in Richtung *phi*. Mit dem Pfadkonstruktor *PathJoin* wird der Pad um die neue Position verlängert. Dabei läßt ein gehobener Stift den Pfad unsichtbar werden.

$$\begin{array}{lll} \multicolumn{3}{l}{\mathit{renderPath}\ \mathit{td}\ (\mathit{TAttrib}\ \mathit{ped}\ (\mathit{TForward}\ d) : \mathit{ps})\ \mathit{tp}} \\ & = & (\mathit{PathJoin}\ (\mathit{actualPos}\ \mathit{td})\ \mathit{ped}'\ \mathit{rp} : \mathit{rps}, \mathit{pics}) \\ \quad \textbf{where} & & \\ \quad (\mathit{rp} : \mathit{rps}, \mathit{pics}) & = & \mathit{renderPath}\ \mathit{td}'\ \mathit{ps}\ \mathit{tp} \\ \quad \mathit{td}' & = & \mathit{td}\{\mathit{tPos} = (x + d * \mathit{cos}\ \mathit{phi}, y + d * \mathit{sin}\ \mathit{phi})\} \\ \quad (x, y) & = & \mathit{tPos}\ \mathit{td} \\ \quad \mathit{phi} & = & \mathit{tOrientation}\ \mathit{td} \\ \quad \mathit{ped}' & = & \textbf{if}\ (\mathit{tPenDown}\ \mathit{td}) \\ & & \qquad \textbf{then}\ \mathit{ped} \\ & & \qquad \textbf{else}\ \mathit{hide}\ \mathit{ped} \end{array}$$

Der Befehl zum Heben des Stiftes ändert nur dem entsprechenden Wert im Record *td*.

$$\begin{array}{lll} \multicolumn{3}{l}{\mathit{renderPath}\ \mathit{td}\ (\mathit{TAttrib}\ \_\ \mathit{TPenUp} : \mathit{ps})\ \mathit{tp}} \\ & = & \mathit{renderPath}\ \mathit{td}\{\mathit{tPenDown} = \mathit{False}\}\mathit{ps}\ \mathit{tp} \end{array}$$

Wird ein Bild mit Hilfe der Funktion *fromPicture* eingefügt, so ist es zusammen mit der aktuellen Turtleposition in die Bilderliste aufzunehmen.

$$
\begin{aligned}
&\mathit{renderPath}\ \mathit{td}\ (\mathit{TAttrib}\ \_\ (\mathit{TDropPic}\ p) : \mathit{ps})\ \mathit{tp} \\
&\qquad\qquad\qquad = (\mathit{rps}, (\mathit{tPos}\ \mathit{td}, p) : \mathit{pics}) \\
&\qquad\mathbf{where} \\
&\qquad(\mathit{rps}, \mathit{pics}) = \mathit{renderPath}\ \mathit{td}\ \mathit{ps}\ \mathit{tp}
\end{aligned}
$$

Die Verzweigung bewirkt eine Berechnung der beiden Teilpfade und hängt anschließend an beide den weiteren Pfad an; wie im Beispiel aus Abbildung 5.2.1.

$$
\begin{aligned}
&\mathit{renderPath}\ \mathit{td}\ (\mathit{TAttribFork}\ \mathit{ta1}\ \mathit{ta2} : \mathit{ps})\ \_ \\
&\qquad\qquad\qquad = (\mathit{actualPos}\ \mathit{td} : \mathit{rps1} +\!\!+\ \mathit{rps2}, \mathit{pics1} +\!\!+\ \mathit{pics2}) \\
&\qquad\mathbf{where} \\
&\qquad(\mathit{rps1}, \mathit{pics1}) = \mathit{renderPath}\ \mathit{td}\ \mathit{ta1}\ (\mathit{renderPath}\ \mathit{td}\ \mathit{ps}\ ([\,], [\,])) \\
&\qquad(\mathit{rps2}, \mathit{pics2}) = \mathit{renderPath}\ \mathit{td}\ \mathit{ta2}\ (\mathit{renderPath}\ \mathit{td}\ \mathit{ps}\ ([\,], [\,]))
\end{aligned}
$$

Da der Pfad, der nur aus der aktuellen Position besteht, mehrmals benötigt wird, definieren wir hierfür eine Hilfsfunktion.

$$
\begin{aligned}
&\mathit{actualPos} && :: \mathit{TurtleDescr} \rightarrow \mathit{Path} \\
&\mathit{actualPos}\ \mathit{td} && = \mathit{toPath}\ \$\ \mathit{vec}\ \$\ \mathit{tPos}\ \mathit{td}
\end{aligned}
$$

Diese beiden Schritte, Verteilung der Attribute und Berechnen der Pfade, kombiniert die Funktion

$$
\begin{aligned}
&\mathit{figure} && :: \mathit{Turtle} \rightarrow \mathit{PathElemDescr} \rightarrow ([\mathit{Path}], [\mathit{PicPos}]) \\
&\mathit{figure}\ t\ \mathit{ped} && = \mathit{renderPath}\ \mathit{default}\ (\mathit{spreadAttrib}\ \mathit{ped}\ t\ [\,])\ ([\,], [\,])
\end{aligned}
$$

Die Funktion *figure* liefert auf unser Beispiel angewendet, den Ausdruck

$$
\begin{aligned}
&([\mathit{vec}\ (0,0)\ \text{--}\ \mathit{vec}\ (5,0) && \#\mathit{setPen}\ 2, \\
&\ \ \mathit{vec}\ (5,0)\ \text{--}\ \mathit{vec}\ (5,20), && \\
&\ \ \mathit{vec}\ (5,20)\ \text{--}\ \mathit{vec}\ (-15,20) && \#\mathit{setColor}\ \mathit{green}, \\
&\ \ \mathit{vec}\ (5,20)\ \text{--}\ \mathit{vec}\ (25,20) && \#\mathit{setColor}\ \mathit{green}, \\
&\ \ \mathit{vec}\ (5,0)\ \text{--}\ \mathit{vec}\ (5,-20), && \\
&\ \ \mathit{vec}\ (5,-20)\ \text{--}\ \mathit{vec}\ (25,-20) && \#\mathit{setColor}\ \mathit{green}, \\
&\ \ \mathit{vec}\ (5,-20)\ \text{--}\ \mathit{vec}\ (-15,-20) && \#\mathit{setColor}\ \mathit{green}], \\
&[((-15,20), \mathit{tex}\ \texttt{"end"} && \#\mathit{setBGColor}\ \mathit{white}), \\
&\ ((25,20), \mathit{tex}\ \texttt{"end"} && \#\mathit{setBGColor}\ \mathit{white}), \\
&\ ((25,-20), \mathit{tex}\ \texttt{"end"} && \#\mathit{setBGColor}\ \mathit{white}), \\
&\ ((-15,-20), \mathit{tex}\ \texttt{"end"} && \#\mathit{setBGColor}\ \mathit{white})])
\end{aligned}
$$

den unser Interpreter *toPicture Turtle* leicht in ein Bild umwandeln kann.
Wir haben jetzt zwei Anwendungen implementiert und dabei die Einbettung von *functional* METAPOST in Haskell ausgenutzt. Bisher bestand aber nicht die Notwendigkeit, Gleichungssysteme zu formulieren, um geometrische Beziehungen auszudrücken. In der nächsten Anwendung werden wir dies nachholen.

5.3 Bäume

Mit dem Modul *Tree* zeigt sich die Eignung von *functional* METAPOST zur Formulierung komplexerer Abhängigkeiten und damit die konzeptionelle Überlegenheit zu anderen Beschreibungssprachen, die keine Formulierung von Gleichungssystemem erlauben.

5.3.1 Eine kurze Geschichte des Baumlayouts

Bäume sind Strukturen, die in der Informatik sehr häufig vorkommen, und viele Algorithmen machen regen Gebrauch von ihnen. Eine Definition für allgemeine Bäume ist: Ein Baum ist ein zusammenhängender zyklenfreier Graph. Da wir uns aber mit gerichteten Bäumen (Arboreszenzen) beschäftigen wollen, d.h. der Baum hat genau eine Wurzel, verwenden wir die rekursive Definition gerichteter Graphen: Ein Baum ist entweder

- der leere Baum,
- oder ein Knoten mit einer endlichen Zahl verknüpfter Teilbäume.

Den Knoten, der nicht zu einem echten Teilbaum gehört, nennen wir Wurzel. Wenn k ein Knoten ist und $w_0, \ldots, w_m$ die Wurzeln seiner verknüpften Teilbäume, dann nennen wir k den Vater von $w_i, 0 \leqslant i \leqslant m$ und $w_0, \ldots, w_m$ die Söhne von k. Ferner sind $w_0, \ldots, w_m$ Brüder. Knoten ohne verknüpfte Teilbäume sind Blätter.

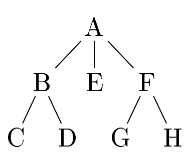
Bäume haben leider die Eigenschaft, sich schriftlich nur wenig übersichtlich notieren zu lassen. Es gibt zwar Notationen, die auf Klammerung oder Einrückung basieren, aber die Darstellungen

```
A
  B
    C
    D
  E
  F
    G
    H
```

oder

```
(A(B(C, D), E, F(G, H)))
```

erschließen sich unserer Anschauung nicht in dem Maße in dem es ein Bild zu leisten vermag.



Dies gilt umso mehr, je größer die Bäume werden. Große Bäume zu zeichnen kann aber mühselig sein, jedenfalls wenn der Baum nicht vollständig ist, die Breite aber möglichst gering sein soll und die Knoten gleichmäßige Abständene aufweisen sollen. Es wäre also von großem Nutzen einen Algorithmus für das Layout von Bäumen zu besitzen. Die Tatsache, daß es vom ersten Versuch gerechnet, ungefähr zwanzig Jahre dauerte, bis ein Algorithmus gefunden war, der wirklich ästhetische Layouts erzeugte, läßt vermuten, daß es sich nicht um ein einfaches Problem handelt, aber beginnen wir die Geschichte am Anfang.

Als erster beschäftigte sich – wie so oft – DONALD E. KNUTH mit dem Problem, einen binären Baum auf einem alphanumerischen Display auszugeben [Knu71] [2].

Wie es der Konvention entspricht, ist die Wurzel der oberste Knoten. Die vertikale Position der Knoten ergibt sich proportional aus ihrem Niveau, die horizontale Position aus ihrer inorder Numerierung. KNUTHs Algorithmus erzeugt lediglich ein Layout, das einen Baum irgendwie darstellt, ohne die Breite zu minimieren oder irgendwelche ästhetische Regeln zu beachten. Wie in Abbildung 5.3.1 zu sehen, nehmen wenig ausgeglichene Bäume zu viel Platz ein, da jede horizontale Position von nur einem Knoten benutzt wird.

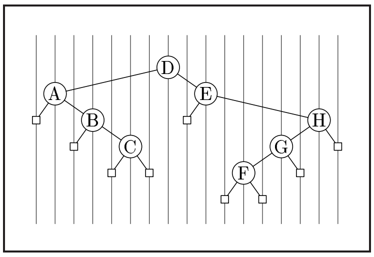


Abbildung 5.3.1: Mit KNUTHs Algorithmus erstelltes Baumlayout

CHARLES WETHERELL und ALFRED SHANNON geben in ihrer Arbeit [WS79] als erste Regeln für ein ästhetisches Layout binärer Bäume an:

① Knoten mit demselben Abstand zur Wurzel sollten auf einer Geraden liegen. Diese Geraden, die das Niveau bestimmen, sollten parallel sein.

② In einem binären Baum sollte der linke Sohn links vom Vater und der rechte rechts davon liegen.

③ Ein Vater sollte über seinen Söhnen zentriert sein.

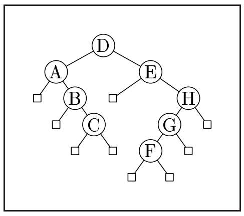


Abbildung 5.3.2: Mit WETHERELL/SHANNONs Algorithmus erstelltes Baumlayout

Die generierten Bäume sind allerdings nicht immer optimal in ihrer Breite und wie in Abbildung 5.3.2 anhand des Abstandes zwischen den Söhnen von Knoten E zu sehen ist, entstehen manchmal unnötige Zwischenräume.

EDWARD M. REINGOLD und JOHN S. TILFORD analysieren in [RT81] die Schwächen von WETHERELL/SHANNONs Algorithmus und kommen zu der Erkenntnis, daß man bei der Positionierung von Knoten stärker die Form der zugehörigen Teilbäume berücksichtigen muß. Der resultierende Algorithmus positioniert, jeweils von den Blättern beginnend, den linken und den rechten Teilbaum möglichst dicht nebeneinander und den Vater in der Mitte darüber.

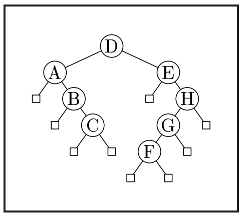


Abbildung 5.3.3: Mit REINGOLD/TILFORDs Algorithmus erstelltes Baumlayout

Damit ist das Problem des Baumlayouts für binäre Bäume zufriedenstellend gelöst. Für den allgemeinen Fall m–ärer Bäume geben REINGOLD und TILFORD eine Modifikation ihres Algorithmus an, die wiederum keine optimalen Bäume erzeugt, wie in Abbildung 5.3.4 zu sehen, ist das Layout nicht symmetrisch.

[2] KNUTH gibt hier lediglich ein ALGOL W Programm an, ohne eine Beschreibung des Algorithmus zu geben. Eine solche findet sich in [Wir75]

GERALD M. RADACK erkennt dies und fügt eine vierte Ästhetikregel hinzu [Rad88].

④ Wenn ein Baum aus einer Spiegelung eines anderen hervorgeht, dann sollte dies auch für das Layout der Bäume gelten. Weiterhin sollte ein Teilbaum unabhängig von seiner Position im Baum immer gleich aussehen.

Die Forderung nach dem immer gleichen Aussehen von gleichen Teilbäumen ermöglicht ein schnelleres visuelles klassifizieren des Baumes. Gleiche Bildteile können schneller als solche erkannt werden als Teilbäume, die verschieden dargestellt und nur logisch gleich sind. REIGOLD/TILFORDs Algorithmus erfüllt diese Forderung d.h., den zweiten Teil von Regel ④ schon. Für den ersten Teil findet RADACK eine Modifikation des Algorithmus, die eine gleichmäßige horizontale Positionierung der Söhne eines Knotens sicherstellt. Auch diese Forderung, die im Grunde den Grad der Symmetrie erhöht, bedingt ein besseres Verstehen des Bildes.

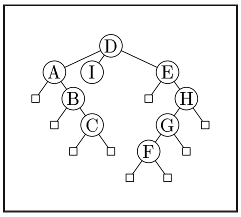


Abbildung 5.3.4: Mit REINGOLD/TILFORDS verallgemeinertem Algorithmus erstelltes Baumlayout

HELEN PURCHASE hat den Einfluß verschiedener ästhetischer Regeln beim Graphlayout auf das richtige und das schnelle Erkennen in [Pur97] untersucht. Sie kommt zu dem Ergebnis, daß ein höherer Grad an Symmetrie zwar nicht das korrekte Erkennen eines Graphen, sehr wohl aber die Geschwindigkeit des Erkennens fördert.

In Abbildung 5.3.5 sieht man, die Auswirkungen der Regel ④ auf das Layout unseres Beispielbaumes. Knoten I ist nun genau zwischen Knoten A und E positioniert.

Bleibt noch anzumerken, daß RADACKs Algorithmus nicht immer optimal schmale Bäume erzeugt, da dies im Widerspruch zu den Regeln ③ und ④ steht[3].

Da wir unser Baumlayout im Modul FMPTree auf RADACKs Algorithmus aufbauen wollen, werden wir diesen im folgenden genauer beschreiben.

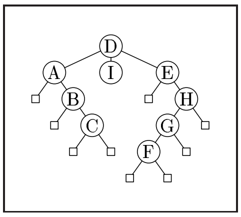


Abbildung 5.3.5: Mit RADACKs Algorithmus erstelltes Baumlayout

5.3.2 Der Algorithmus von RADACK

Der Algorithmus behandelt, bei der Wurzel angefangen, jeden Teilbaum nach folgenden Schritten.

1. Berechne das Layout der Teilbäume der Wurzel mit dem Algorithmus.

2. Plaziere die Teilbäume unter Beachtung der Regeln ①-④ relativ zur Wurzel möglichst dicht nebeneinander. Siehe Abbildung 5.3.6 (a) (b) und (c).

Die Schwierigkeit besteht darin zu gewährleisten, daß sich in Schritt 2. die Teilbäume nicht überschneiden, sondern immer ein Abstand gegeben ist. Deshalb wird zu jedem Teilbaum eine Hülle gespeichert. Für die Hüllen genügt es, zu jedem Niveau zwei Zahlen für den horizontalen Offset des linkesten und rechtesten Knotens, bezogen auf die Wurzel, zu speichern. Nachdem in Schritt 2. die Teilbäume plaziert wurden, ergibt sich die neue Hülle aus den um die Plazierungen verschobenen Teilhüllen, plus der Hülle der Wurzel. Wir erhalten damit den letzten Schritt:

[3] WETHERELL und SHANNON geben in ihrer Arbeit [WS79] eine Modifikation ihres Algorithmus an, die minimale Breite auf Kosten von Regel ③ gewährleistet.

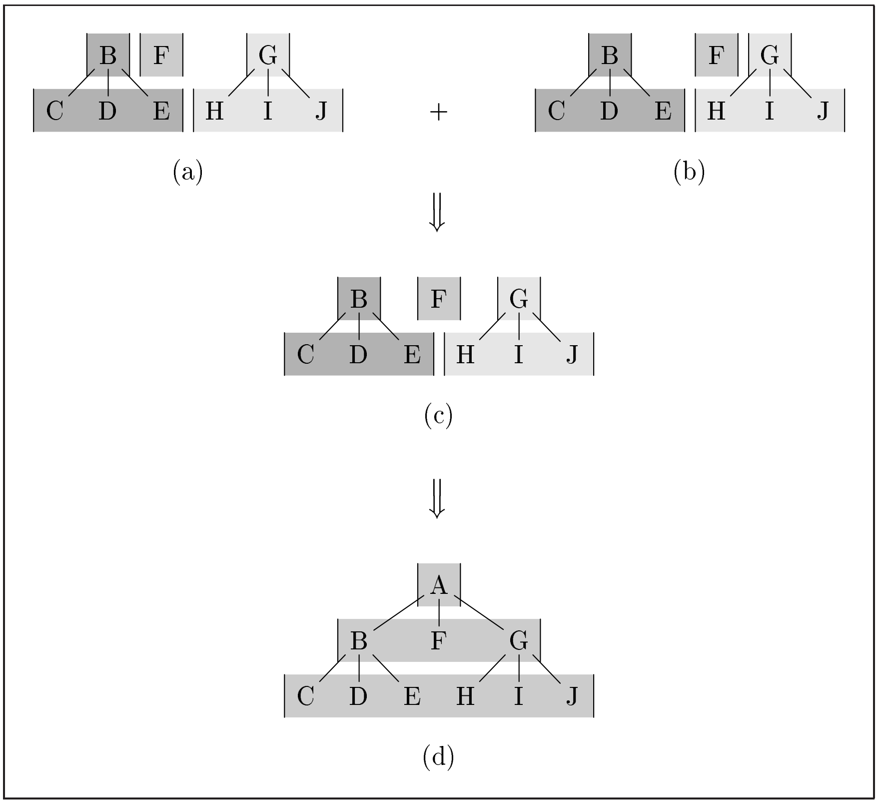


Abbildung 5.3.6: Radacks Idee für eine symmetrischere Positionierung. Die grauen Bereiche symbolisieren die Hüllen der Teilbäume auf ihren verschiedenen Ebenen. (a) Die Bäume sind von links nach rechts gepackt. (b) Die Bäume sind von rechts nach links gepackt. (c) Das arithmetische Mittel der Plazierungen (a) und (b). (d) Die neue Hülle des soeben konstruierten Teilbaums wird gespeichert.

3. Bilde aus den Hüllen der Söhne und der Wurzel eine neue Hülle. Siehe Abbildung 5.3.6 (d).

Bei einem naiven, möcglichst dichten Packen der Teilbäume von links nach rechts, wie bei Reingold/Tilford, ist die Symmetrie, die Regel ④ fordert, nicht gewährleistet. Radack kam auf die Idee [Rad88], die Teilbäume einmal von links nach rechts und ein zweites Mal von rechts nach links zu packen und dann die Mittelwerte beider Positionen zu verwenden.

5.3.3 Implementierung einer Erweiterung des Algorithmus von Radack

Andrew Kennedy liefert in [Ken93] eine deskriptive Implementierung des ursprünglich imperativen Algorithmus von Radack in der funktionalen Programmiersprache ML. Wir werden darauf aufbauen und folgende Erweiterungen einbauen. Wie alle hier vorgestellten Algorithmen geht auch Kennedys Algorithmus von gleichhohen und gleichbreiten Knoten aus und berücksichtigt deshalb bei der Wahl von Abständen nicht die Knotengrößen. Wir werden diese Größen in unsere Berechnungen einfließen lassen. Dabei handeln wir uns allerdings ein Problem ein: Die Größe der Knoten kennen wir erst, nachdem METAPOST diese berechnet hat. Dies liegt an der Möglichkeit, TeX einzubinden sowie der Möglichkeit von affinen Transformationen, die die Bounding Box eines Bildes so

verändern können, daß ihr Bild schwierig zu berechnen ist.[4] Deshalb müssen wir alle Rechenoperationen auf den Termen von *functional* METAPOST statt mit normalen Zahlen ausführen. Diese Terme rechnet später das Programm METAPOST aus, das alle Größen ermitteln kann. Wenn wir dabei naiv vorgehen und den Algorithmus, wie er in [Ken93] angegeben ist übernehmen, wächst die Termlänge exponentiell, da die Ergebnisse aus dem letzten Rekursionsschritt an mehreren Stellen des neuen Ergebnisses auftreten. Deshalb werden wir an geeigneter Stelle Zwischenergebnisse an Variablen binden und dann mit diesen Variablen weiterrechnen.
Die nächste Erweiterung ist die Möglichkeit, in das Layout lokal einzugreifen. Dies ermöglicht z.B. konstante Knotenabstände, und andere Effekte wie sie z.B. für die Abbildungen 5.3.1-5.3.5 notwendig sind. Schließlich wollen wir als Kanten beliebige Pfade inklusive Querkanten erlauben.

Beginnen wir unsere Aufgabe mit der Definition der Datenstrukturen. Jeder Knoten speichert neben dem Knotenbild und den von ihm ausgehenden Kanten, eine kleine Attributmenge.

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Tree} & = & \mathit{Node\ Picture\ NodeDescr}\ [\mathit{Edge}] \\
 & & \textbf{deriving}\ \mathit{Show}
\end{array}
$$

Kanten können entweder in einen neuen Teilbaum münden oder quer zu beliebigen Knoten verlaufen.

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Edge} & = & \mathit{Edge\ Path\ Tree} \\
 & \mid & \mathit{Cross\ Path} \\
 & & \textbf{deriving}\ \mathit{Show}
\end{array}
$$

Die Attributierungen, die das Layout festlegen werden von einem Record gespeichert.

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{NodeDescr} & = & \mathit{NodeDescr}\{\mathit{nEdges} :: [\mathit{Path}], \\
 & & \qquad\qquad\ \mathit{nAlignSons} :: \mathit{AlignSons}, \\
 & & \qquad\qquad\ \mathit{nDistH}, \mathit{nDistV} :: \mathit{Distance}\} \\
 & & \textbf{deriving}\ \mathit{Show}
\end{array}
$$

Jetzt lassen sich die Attributierungsfunktionen, analog zu denen für Turtlegrafik, leicht definieren. Beispielsweise sieht die Farbattributierung für Knoten und Kanten so aus:

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\textbf{instance}\ \mathit{HasColor\ Tree}\ \textbf{where}} \\
\quad \mathit{setColor}\ c\ (\mathit{Node}\ p\ \mathit{nd}\ \mathit{es}) & & \\
 & = & \mathit{Node}\ (\mathit{setColor}\ c\ p)\ \mathit{nd}\ \mathit{es} \\
\quad \mathit{setDefaultColor}\ t & = & \mathit{setColor}\ default\ t \\
\quad \mathit{getColor}\ (\mathit{Node}\ p\ \_\ \_) & = & \mathit{getColor}\ p
\end{array}
$$

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\textbf{instance}\ \mathit{HasColor\ Edge}\ \textbf{where}} \\
\quad \mathit{setColor}\ c\ (\mathit{Edge}\ e\ \mathit{ts}) & = & \mathit{Edge}\ (\mathit{setColor}\ c\ e)\ \mathit{ts} \\
\quad \mathit{setColor}\ c\ (\mathit{Cross}\ e) & = & \mathit{Cross}\ (\mathit{setColor}\ c\ e) \\
\quad \mathit{setDefaultColor}\ (\mathit{Edge}\ e\ \mathit{ts}) & & \\
 & = & \mathit{Edge}\ (\mathit{setDefaultColor}\ e)\ \mathit{ts} \\
\quad \mathit{setDefaultColor}\ (\mathit{Cross}\ e) & & \\
 & = & \mathit{Cross}\ (\mathit{setDefaultColor}\ e) \\
\quad \mathit{getColor}\ (\mathit{Edge}\ e\ \_) & = & \mathit{getColor}\ e \\
\quad \mathit{getColor}\ (\mathit{Cross}\ e) & = & \mathit{getColor}\ e
\end{array}
$$

[4] Die Form einer Bounding Box kann ja beliebig sein.

Die Umwandlung eines Baumes in ein Bild geschieht in zwei Schritten. Zuerst werden alle Bilder der Knoten nach dem Algorithmus von KENNEDY mit der Funktion *overlay* plaziert. Danach ergänzt ein *draw*–Befehl den Baum um die Kanten.

$$
\begin{array}{lll}
\textbf{instance}\ \mathit{IsPicture\ Tree}\ \textbf{where} & & \\
\quad \mathit{toPicture}\ t & = & \mathit{draw\ edgePaths} \\
 & & \quad (\mathit{overlay}\ (\mathit{widthsL}\ \&\ \mathit{widthsR} \\
 & & \qquad\quad \&\ \mathit{heightsTop}\ \&\ \mathit{heightsBot} \\
 & & \qquad\quad \&\ \mathit{voffs}\ \&\ \mathit{hoffs} \\
 & & \qquad\quad \&\ \mathit{placements}) \\
 & & \qquad (\mathit{enumPics\ nodePics})) \\
\qquad \textbf{where} & & \\
\qquad \mathit{widthsL} & = & [\mathit{widthL}\ i \doteq \mathit{xpart}\ (\mathit{ref}\ (i \triangleleft W) - \mathit{ref}\ (i \triangleleft C)) \\
 & & \mid i \leftarrow [0\,..\,\mathit{length\ nodePics} - 1]] \\
\qquad \mathit{widthsR} & = & [\mathit{widthR}\ i \doteq \mathit{xpart}\ (\mathit{ref}\ (i \triangleleft E) - \mathit{ref}\ (i \triangleleft C)) \\
 & & \mid i \leftarrow [0\,..\,\mathit{length\ nodePics} - 1]] \\
\qquad \mathit{heightsTop} & = & [\mathit{heightT}\ l \doteq \mathit{maximum}'\ (\mathit{map\ heightTop\ ns}) \\
 & & \mid (ns, l) \leftarrow \mathit{zip}\ (\mathit{levels\ nt})\ [1\,..]] \\
\qquad \mathit{heightsBot} & = & [\mathit{heightB}\ l \doteq \mathit{maximum}'\ (\mathit{map\ heightBot\ ns}) \\
 & & \mid (ns, l) \leftarrow \mathit{zip}\ (\mathit{levels\ nt})\ [1\,..]] \\
\qquad \mathit{voffs} & = & [\mathit{voff}\ l \doteq -\mathit{heightT}\ (l+1) - \mathit{heightB}\ l \\
 & & \mid l \leftarrow (\mathit{tail}\ [0\,..\,\mathit{length}\ (\mathit{levels\ nt}) - 1])] \\
\qquad \mathit{heightTop}\ n & = & \mathit{ypart}\ (\mathit{ref}\ (n \triangleleft N) - \mathit{ref}\ (n \triangleleft C)) \\
\qquad \mathit{heightBot}\ n & = & \mathit{ypart}\ (\mathit{ref}\ (n \triangleleft C) - \mathit{ref}\ (n \triangleleft S)) \\
\qquad \mathit{nt} & = & \mathit{number}\ t \\
\qquad \mathit{hoffs} & = & \mathit{design\ nt} \\
\qquad \mathit{placements} & = & \mathit{tail}\ (\mathit{relPlacements\ nt}) \\
\qquad \mathit{nodePics} & = & \mathit{extractPics}\ t \\
\qquad \mathit{edgePaths} & = & \mathit{edges}\ [\,]\ \mathit{nt}
\end{array}
$$

Für die Baumbschreibungssprache, ist es sinnvoll, Knoten und Kanten zu trennen. Bei der Berechnung des Layouts ist es aber einfacher, die Information über eingehende Kanten zusammen mit einer Numerierung im Knoten zu speichern. Deshalb verwenden wir für die Berechnung des Baumlayouts den Typ

$$
\textbf{data}\ \mathit{Tree}'\ a \quad = \quad \mathit{Node}'\ a\ \mathit{NodeDescr}\ [\mathit{Tree}'\ a]
$$

Die Funktion *number* verteilt die Kanteninformationen auf die Knoten und bildet einen Baum aus Tripeln. Die erste Zahl eines Tripels gibt die Nummer des Vaterknotens an, die zweite die Nummer des Knotens in preorder–Numerierung und die dritte das Niveau des Knotens, wie in Abbildung 5.3.7 zu sehen ist.

$$
\begin{array}{lll}
\mathit{number} & :: & \mathit{Tree} \rightarrow (\mathit{Tree}'\ (\mathit{Int}, \mathit{Int}, \mathit{Int})) \\
\mathit{number}\ t & = & \mathit{snd}\ (\mathit{traverse}\ (-1)\ 0\ 0\ t\ [\,]) \\
\quad \textbf{where} & & \\
\quad \mathit{traverse}\ j\ k\ l\ (\mathit{Node}\ \_\ \mathit{nd\ ts})\ \mathit{pe} & & \\
 & = & (k', \mathit{Node}'\ (j,k,l)\ \mathit{nd}\{\mathit{nEdges} = \mathit{edges\ ts}\}\mathit{nts})
\end{array}
$$

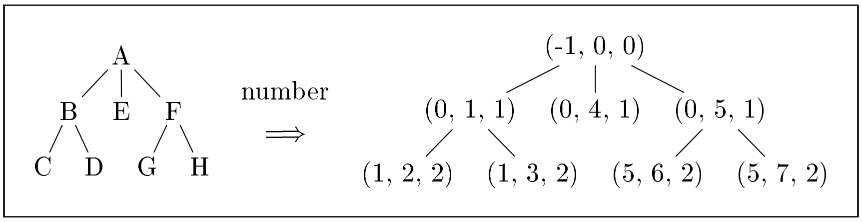


Abbildung 5.3.7: Die Funktion *number* markiert die Knoten mit Informationen, die zur Formulierung eines Gleichungssystems, das den Baum beschreibt, wichtig sind.

$$\begin{array}{lcl}
\textbf{where} & & \\
\quad \mathit{edges}\ [\,] & = & \mathit{pe} \\
\quad \mathit{edges}\ (\mathit{Edge}\ _\ _ : \mathit{es}) & = & \mathit{edges}\ \mathit{es} \\
\quad \mathit{edges}\ (\mathit{Cross}\ e : \mathit{es}) & = & e : \mathit{edges}\ \mathit{es} \\
\quad \mathit{sons}\ [\,] & = & [\,] \\
\quad \mathit{sons}\ (\mathit{Edge}\ e\ s : \mathit{es}) & = & (e, s) : \mathit{sons}\ \mathit{es} \\
\quad \mathit{sons}\ (_ : \mathit{es}) & = & \mathit{sons}\ \mathit{es} \\
\quad (k', \mathit{nts}) & = & \mathit{traverses}\ k\ (k+1)\ (l+1)\ (\mathit{sons}\ \mathit{ts}) \\
\quad \mathit{traverses}\ _\ k\ _\ [\,] & = & (k, [\,]) \\
\quad \mathit{traverses}\ j\ k\ l\ ((e,t) : \mathit{ts}) & & \\
 & = & (k'', \mathit{nt} : \mathit{nts}) \\
\qquad \textbf{where} & & \\
\qquad (k', \mathit{nt}) & = & \mathit{traverse}\ j\ k\ l\ t\ [e] \\
\qquad (k'', \mathit{nts}) & = & \mathit{traverses}\ j\ k'\ l\ \mathit{ts}
\end{array}$$

Mit einem so markierten Baum lassen sich leicht die Gleichungen für die relative Knotenpositionierung aufstellen. Für diese ist nur die formale Struktur des Baumes von Bedeutung, deshalb enthält der numerierte Baum keine Informationen über die Knotenbilder. Die Bilder werden nur als Argument der *overlay*–Funktion benötigt. Zu diesem Zweck erzeugen wir aus dem Baum eine Bilderliste, analog zur Knotennumerierung, in preorder–Reihenfolge.

$$\begin{array}{lcl}
\mathit{extractPics} & :: & \mathit{Tree} \rightarrow [\mathit{Picture}] \\
\mathit{extractPics}\ (\mathit{Node}\ p\ _\ \mathit{ts}) & = & p : \mathit{pics}\ \mathit{ts} \\
\quad \textbf{where} & & \\
\quad \mathit{pics}\ [\,] & = & [\,] \\
\quad \mathit{pics}\ (\mathit{Edge}\ _\ t : \mathit{es}) & = & \mathit{extractPics}\ t \mathbin{+\!\!+} \mathit{pics}\ \mathit{es} \\
\quad \mathit{pics}\ (_ : \mathit{es}) & = & \mathit{pics}\ \mathit{es}
\end{array}$$

Um die entstehenden Terme möglichst klein zu halten, speichern wir oft benötigte Werte, wie die Breite der rechten und linken Knotenhälften, in Hilfsvariablen. Wenn der Punkt *C* eines Bildes immer genau in der Mitte liegen würde, käme man ohne eine Unterscheidung der linken (*widthL*) und rechten (*widthR*) Breite aus. Aber wir müssen mit Ausnahmen rechnen. In einem mit *triangle* erzeugten Rahmen fallen z.B. die Punkte *C* und *N* zusammen.
Die Hilfsvariablen sollen durchnumeriert sein. Es bietet sich also an, ihnen Namen vom Typ *Int* zu geben. Solche Variablen werden in konstanter Zeit verwaltet. Somit verschlechtert sich die Laufzeit des Algorithmus nicht unnötig durch die Variablenverwaltung.

$$
\begin{aligned}
&\mathit{hoff}, \mathit{voff}, \mathit{widthL}, \mathit{widthR}, \mathit{heightT}, \mathit{heightB} :: \mathit{Int} \to \mathit{Numeric}\\
&\mathit{hoff}\ i && = \mathit{var}\ (6 * i)\\
&\mathit{voff}\ i && = \mathit{var}\ (6 * i + 1)\\
&\mathit{widthL}\ i && = \mathit{var}\ (6 * i + 2)\\
&\mathit{widthR}\ i && = \mathit{var}\ (6 * i + 3)\\
&\mathit{heightT}\ i && = \mathit{var}\ (6 * i + 4)\\
&\mathit{heightB}\ i && = \mathit{var}\ (6 * i + 5)
\end{aligned}
$$

Die Variablen *heightT l* und *heightB i* speichern vertikale Abstände zwischen den verschiedenen Baumniveaus wie in Abbildung 5.3.8 zu sehen.

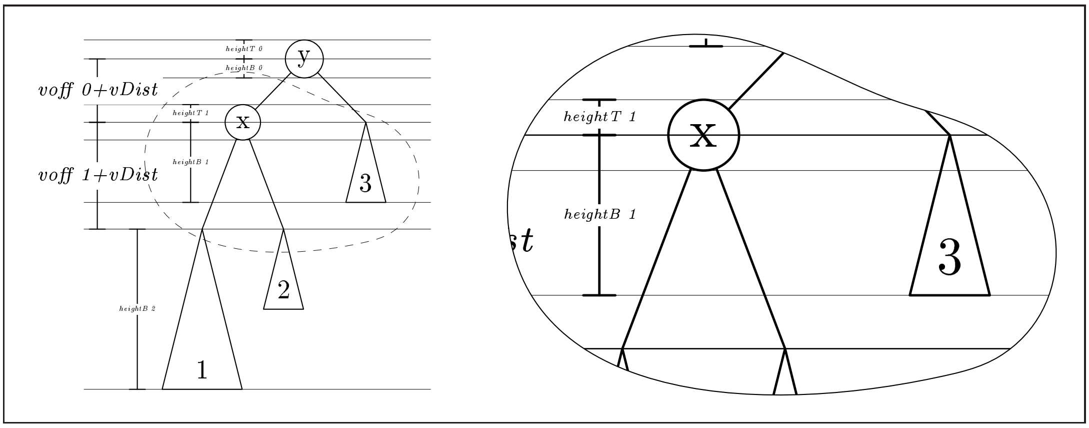


Abbildung 5.3.8: Die vertikalen Abstände zwischen den Knoten berechnen sich aus dem Maximum der unteren Knotenhöhen einer Baumebene plus dem Maximum der oberen Knotenhöhen der nächsten Ebene plus einem Wert *vSep*, den das entsprechende Knotenattribut festlegt.

Um diese Werte zu erhalten, müssen Maxima über alle Knotenhöhen eines Niveaus gebildet werden. Die Funktion *levels* erzeugt eine Liste, wo das n-te Element die Liste der Knotennummern des n-ten Niveaus ist. Für den Baum aus Abbildung 5.3.7 ergäbe sich $[[0],[1,4,5],[2,3,6,7]]$, für Abbildung 5.3.8 $[[0],[1,4],[2,3]]$.

$$
\begin{aligned}
&\mathit{levels} && :: \mathit{Tree}'\ (a,b,c) \to [[b]]\\
&\mathit{levels}\ (\mathit{Node}'\ (\_,a,\_)\ \_\ [\,]) && = [[a]]\\
&\mathit{levels}\ (\mathit{Node}'\ (\_,a,\_)\ \_\ \mathit{ts}) && = [a] : \mathit{foldl}\ \mathit{zipLists}\ [\,]\ (\mathit{map}\ \mathit{levels}\ \mathit{ts})\\
&\qquad \textbf{where}\\
&\qquad \mathit{zipLists}\ [\,]\ l && = l\\
&\qquad \mathit{zipLists}\ l\ [\,] && = l\\
&\qquad \mathit{zipLists}\ (l : \mathit{ls})\ (l' : \mathit{ls}') && = (l +\!\!+\ l') : \mathit{zipLists}\ \mathit{ls}\ \mathit{ls}'
\end{aligned}
$$

Die Gleichungen für die relativen Knotenpositionen definieren jeweils die Position eines Knotens relativ zu seinem Vater. Dabei müssen wir die zwei verschiedenen Arten vertikale Abstände anzugeben berücksichtigen. Für *SepBorder* gehen die Knotenhöhen in die Berechnung mit ein, bei *SepCenter* soll dagegen der vertikale Abstand der Knotenbildmittelpunkte definiert werden.

$$
\begin{aligned}
&\mathit{relPlacements} && :: \mathit{Tree}'\ (\mathit{Int},\mathit{Int},\mathit{Int}) \to [\mathit{Equation}]\\
&\mathit{relPlacements}\ (\mathit{Node}'\ (a,b,l)\ \mathit{nd}\ \mathit{ts})
\end{aligned}
$$

$$
\begin{aligned}
= \ [\textbf{case}\ \mathit{nDistV}\ \mathit{nd}\ \textbf{of}\\
&\quad \mathit{DistBorder}\ v \rightarrow \mathit{ref}\ (b \triangleleft C) \doteq \mathit{ref}\ (a \triangleleft C)\\
&\qquad\qquad\qquad\qquad +\mathit{vec}\ (\mathit{hoff}\ b, \mathit{voff}\ l - v)\\
&\quad \mathit{DistCenter}\ v \rightarrow \mathit{ref}\ (b \triangleleft C) \doteq \mathit{ref}\ (a \triangleleft C)\\
&\qquad\qquad\qquad\qquad +\mathit{vec}\ (\mathit{hoff}\ b, -v)]\\
\&\mathit{map}\ (\mathit{equations} \circ \mathit{relPlacements})\ \mathit{ts}
\end{aligned}
$$

Bleibt jetzt nur noch das eigentliche Problem des Baumlayouts, in dem die Gleichungen festzulegen sind, die eine ästhetische relative Knotenpositionierung beschreiben und den Variablen *hoff* zugewiesen sind. Zur besseren Lesbarkeit geben wir den Typen für relative Positionen und Hüllen besondere Namen.

$$
\begin{aligned}
&\textbf{type}\ \mathit{Position} && = \mathit{Numeric}\\
&\textbf{type}\ \mathit{Extent} && = [(\mathit{Position}, \mathit{Position})]
\end{aligned}
$$

Die Funktion *fit* berechnet den notwendige horizontalen Abstand, den zwei Bäume voneinander haben müssen, damit ihre Hüllen an der dichtesten Stelle *hsep* Einheiten voneinander entfernt sind. Siehe Abbildung 5.3.9.

$$
\begin{aligned}
&\mathit{fit} && :: \mathit{Numeric} \rightarrow \mathit{Extent} \rightarrow \mathit{Extent} \rightarrow \mathit{Position}\\
&\mathit{fit}\ \mathit{hDist}\ \mathit{ps}\ \mathit{qs} && = \mathit{maximum}'\ \mathit{dists}\\
&\qquad \textbf{where}\ \mathit{dists} && = [r - l + \mathit{hDist} \mid ((\_, r), (l, \_)) \leftarrow \mathit{zip}\ \mathit{ps}\ \mathit{qs}]
\end{aligned}
$$

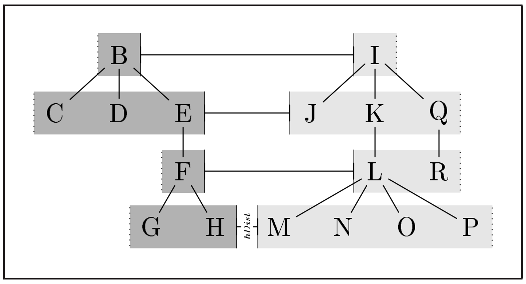


Abbildung 5.3.9: Die Funktion *fit* berechnet den Wert um den ein Baum zu verschieben ist, s.d. der kleinste Abstand zwischen den Bäumen *hsep* beträgt

Die Funktion *fitLeft* erzeugt für eine Liste von Hüllen die notwendigen Verschiebungen, die den Positionen der Wurzeln entsprechen, s.d. die benachbarten Hüllen den Abstand *hsep* voneinander haben. Dabei konstruiert die Funktion eine Hülle von links nach rechts, indem sie an der bisherigen Hülle die jeweils nächste, entsprechend verschoben, anhängt. Siehe Abbildung 5.3.10. Analog hierzu konstruiert *fitRight* die Hülle und gleichzeitig die Liste von Positionen, durch Packen von rechts nach links.

$$
\begin{aligned}
&\mathit{fitLeft} && :: \mathit{Numeric} \rightarrow [\mathit{Extent}] \rightarrow [\mathit{Position}]\\
&\mathit{fitLeft}\ \mathit{hDist}\ \mathit{es} && = \mathit{traverse}\ \mathit{hDist}\ [\,]\ \mathit{es}\\
&\quad \textbf{where} &&\\
&\quad \mathit{traverse} && :: \mathit{Numeric} \rightarrow \mathit{Extent} \rightarrow [\mathit{Extent}] \rightarrow [\mathit{Position}]\\
&\quad \mathit{traverse}\ \_\ \_\ [\,] && = [\,]\\
&\quad \mathit{traverse}\ \mathit{hDist}\ \mathit{acc}\ (e : \mathit{es}) && = x : \mathit{traverse}\ \mathit{hDist}\ (\mathit{merge}\ \mathit{acc}\ (\mathit{moveExtent}\ x\ e))\ \mathit{es}\\
&\qquad \textbf{where}\ x && = \mathit{fit}\ \mathit{hDist}\ \mathit{acc}\ e
\end{aligned}
$$

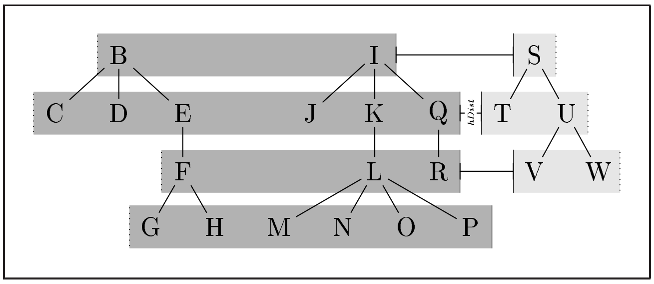


Abbildung 5.3.10: Die Funktion *fitLeft* packt eine Liste von Hüllen von links nach rechts.

Nach RADACKs Idee erzeugt das Mittel der Ergebnisse von *fitLeft* und *fitRight* eine symmetrische Positionierung.

$$\begin{array}{lcl} \mathit{fitMany} & :: & \mathit{Numeric} \to [\mathit{Extent}] \to [\mathit{Position}] \\ \mathit{fitMany\ hDist\ es} & = & [(x+y)\ /\ 2 \\ & & \mid (x,y) \leftarrow \mathit{zip}\ (\mathit{fitLeft\ hDist\ es})\ (\mathit{fitRight\ hDist\ es})] \end{array}$$

Um zwei Hüllen zu mischen genügt es, für jedes Niveau den linken Wert des ersten Baumes und den rechten des zweiten Baumes zu berücksichtigen.

$$\begin{array}{lcl} \mathit{merge} & :: & \mathit{Extent} \to \mathit{Extent} \to \mathit{Extent} \\ \mathit{merge}\ [\,]\ \mathit{qs} & = & \mathit{qs} \\ \mathit{merge\ ps}\ [\,] & = & \mathit{ps} \\ \mathit{merge}\ ((l,\_) : \mathit{ps})\ ((\_,r) : \mathit{qs}) & = & (l,r) : \mathit{merge\ ps\ qs} \end{array}$$

Wir können die Funktion *merge* auf Listen fortsetzen.

$$\begin{array}{lcl} \mathit{mergeMany} & :: & [\mathit{Extent}] \to \mathit{Extent} \\ \mathit{mergeMany} & = & \mathit{foldr\ merge}\ [\,] \end{array}$$

Vor dem Mischen zweier Hüllen muß eine von beiden, auf die von *fit* ermittelte Position verschoben werden. Dafür ist der entsprechende Wert auf die gesamte Hülle aufzuaddieren.

$$\begin{array}{lcl} \mathit{moveExtent} & :: & \mathit{Position} \to \mathit{Extent} \to \mathit{Extent} \\ \mathit{moveExtent}\ x & = & \mathit{map}\ (+(x,x)) \end{array}$$

Aus numerierten Bäumen können wir mit Hilfe des Algorithmus von RADACK ein ästhetisches Layout berechnen. Wir wollen, anders als dieser, allerdings nicht die Baumknoten mit ihren relativen Positionen markieren, sondern direkt Gleichungen erzeugen, die diese Informationen beschreiben.

$$\begin{array}{lcl} \mathit{design} & :: & \mathit{Tree}'\ (\mathit{Int},\mathit{Int},\mathit{Int}) \to [\mathit{Equation}] \\ \mathit{design}\ t & = & \mathit{fst}\ (\mathit{design}'\ t) \end{array}$$

Die Funktion *design*$'$ berechnet ein Tupel, welches im ersten Parameter das Layout eines Teilbaums und im zweiten seine Hülle enthält. Wir können hier die drei Schritte aus Kapitel 5.3.2 wiederfinden. in Schritt 1. wird die Funktion rekursiv auf alle Bäume der Söhne angewendet. Im zweiten Schritt

folgt die Berechnung der relativen Positionen der Teilbäume, wobei die Hüllen aus Schritt 1. Verwendung finden. Zuletzt muß im dritten Schritt die neue Hülle gebildet werden. Die Variable *eqs* enthält die Gleichungen für die relativen horizontalen Positionen der Söhne. Diese Positionen sind Variablen mit den Namen *hoff m* zugewiesen, wo *m* die Nummer des Sohnes ist. Sehr wichtig ist, daß wir mit den Variablen *hoff m* weiterrechnen, wenn wir mit *moveExtent* die Hüllen verschieben, da die Länge der Terme sonst exponentiell wachsen würde.
Da die horizontale Positionierung nicht nur unter Berücksichtigung der Knotenbreite, sondern optional als Abstand zwischen den Knotenzentren erfolgen kann, muß die Hülle der Wurzel, diesen Fällen entsprechend, so breit sein, wie die Wurzel oder keine Ausmaße haben. Dies regelt die Funktion *topExtent*.

$$
\begin{array}{lll}
\multicolumn{2}{l}{design'} & :: \; Tree' \; (Int, Int, Int) \rightarrow ([Equation], Extent) \\
\multicolumn{2}{l}{design' \; (Node' \; (n,m,l) \; nd \; ts)} & = \; (foldl \; (\&) \; [\,] \; designedTrees \; \& \; eqs, \\
 & & \quad topExtent \; (nDistH \; nd) : mergedExtent) \\
 & \textbf{where} & \\
 & (designedTrees, es) & = \; unzip \; [design' \; t | t \leftarrow ts] \\
 & relPositions & = \; calcPos \; nd \; es \; l \\
 & mergedExtent & = \; mergeMany \; [moveExtent \; h \; e | (h,e) \leftarrow zip \; hoffVars \; es] \\
 & eqs & = \; [h \doteq rp | (h, rp) \leftarrow zip \; hoffVars \; relPositions] \\
 & hoffVars & = \; [hoff \; m | Node' \; (\_, m, \_) \; \_ \; \_ \leftarrow ts] \\
 & topExtent \; (DistBorder \; \_) & \\
 & & = \; (widthL \; m, widthR \; m) \\
 & topExtent \; \_ & = \; (0,0)
\end{array}
$$

Die Art und Weise, in der die Söhne eines Knotens positioniert werden, bestimmt das Knotenattribut *nAlignSons*.

$$
\begin{array}{ll}
calcPos & :: \; NodeDescr \rightarrow [Extent] \rightarrow Int \rightarrow [Position] \\
calcPos \; nd \; es \; l & = \; calcPos' \; (nAlignSons \; nd) \; nd \; es \; l
\end{array}
$$

Die Positionierung erfolgt normalerweise mit *fitMany* wie in der Originalarbeit von ANDREW KENNEDY [Ken93]. Aber wir wollen hier etwas flexibler sein und lokale Eingriffe in das Layout erlauben. Wir geben hier nur einen Ausschnitt der vordefinierten Anordnungen wieder. Die restlichen sind entsprechend implementiert.

$$
\begin{array}{lll}
\multicolumn{2}{l}{calcPos'} & :: \; AlignSons \rightarrow NodeDescr \rightarrow [Extent] \rightarrow Int \rightarrow [Position] \\
\multicolumn{2}{l}{calcPos' \; DefaultAlign \; nd \; es \; \_} & = \; fitMany \; (getHDist \; nd) \; es \\
\multicolumn{2}{l}{calcPos' \; AlignLeft \; nd \; es \; \_} & = \; fitLeft \; (getHDist \; nd) \; es \\
\multicolumn{2}{l}{calcPos' \; AlignLeftSon \; nd \; [((l, \_) : \_)] \; \_} & \\
 & & = \; [l - 0.5 * getHDist \; nd] \\
\multicolumn{2}{l}{calcPos' \; AlignLeftSon \; nd \; es \; \_} & = \; fitMany \; (getHDist \; nd) \; es \\
\multicolumn{2}{l}{calcPos' \; (AlignAngles \; ds) \; nd \; es \; h} & \\
 & & = \; take \; (length \; es) \; (calcOffsets \; ds \\
 & & \qquad\qquad + \!\! + \; fitLeft \; (getHDist \; nd) \\
 & & \qquad\qquad\qquad (drop \; (length \; ds) \; es)) \\
 & \textbf{where} & \\
 & calcOffsets \; [\,] & = \; [\,] \\
 & calcOffsets \; (d : ds) & = \; offset \; d : calcOffsets \; ds
\end{array}
$$

$$\begin{array}{lcl} \quad \mathit{offset}\ d & = & (\mathit{voff}\ (h+1) - \mathit{getVDist}\ \mathit{nd}) * \cos d \,/\, \sin d \\ \mathit{calcPos}'\ (\mathit{AlignFunction}\ f)\ \mathit{nd}\ \mathit{es}\ h & & \\ & = & f\ \mathit{nd}\ \mathit{es}\ h \end{array}$$

Bei der Beschreibung der Kanten tritt das Problem auf, die zu verbindenden Punkte zu beschreiben, da für eine allgemeine Angabe keine speziellen Knotennamen verwendet werden können. Zur Lösung diese Problems orientieren wir uns an der Art und Weise, wie wir den Verlauf von Baumkanten verbal beschreiben würden. Nämlich z.B. ”Die Kante verläuft von der Mitte des Knotens zur Mitte des Vaters des Knotens“, wobei „Knoten“ und „Vater des Knoten“ im jeweiligen Kontext des betrachteten Knotens verschiedene Knoten referenzieren.
Also werden wir einen neuen Namenstyp definieren, den wir in allgemeinen Pfadbeschreibungen verwenden können. Diese Platzhalter werden dann im jeweiligen Kontext durch die aktuellen Knotennamen ersetzt.

$$\begin{array}{lcl} \mathbf{data}\ \mathit{NodeName} & = & \mathit{Parent} | \mathit{This} | \mathit{Root} | \mathit{Up}\ \mathit{Int} | \mathit{Son}\ \mathit{Int} \\ & & \mathbf{deriving}\ \mathit{Show} \end{array}$$

Wir müssen die Platzhalter auf Strings abbilden, da sich der Typ *Name* leider nicht erweitern läßt.

$$\begin{array}{lcl} \multicolumn{3}{l}{\mathbf{instance}\ \mathit{IsName}\ \mathit{NodeName}\ \mathbf{where}} \\ \quad \mathit{toName}\ a & = & \mathit{toName}\ (\mathit{show}\ a) \end{array}$$

In allen Kantenpfaden des Baumes sind alle Vorkommen von Platzhaltern durch die, im richtigen Kontext gültigen Namen zu ersetzen. Dabei wird für jeden Knoten eine Tabelle gebildet, welche Ersetzungsregeln definiert.
Um auch den Platzhalter *Up n* auf den Namen des entsprechenden Knotens abbilden zu können, ist es notwendig, die Knotennummern des Pfades von der Wurzel bis zum betrachteten Knoten, zu speichern. Die Funktion *edges* sammelt diesen Pfad in ihrem ersten Parameter an.

$$\begin{array}{lcl} \mathit{edges} & :: & [\mathit{Int}] \to \mathit{Tree}'\ (\mathit{Int},\mathit{Int},\mathit{Int}) \to [\mathit{Path}] \\ \mathit{edges}\ \mathit{path}\ (\mathit{Node}'\ (a,b,\_)\ \mathit{nd}\ \mathit{ts}) & & \\ & = & [\mathit{replacePath}\ \mathit{edge}\ \mathit{aliases} | \mathit{edge} \leftarrow \mathit{nEdges}\ \mathit{nd}] \\ & +\!\!+ & \mathit{concat}\ (\mathit{map}\ (\mathit{edges}\ (b : \mathit{path}))\ \mathit{ts}) \\ \qquad\qquad \mathbf{where} & & \\ \qquad\qquad \mathit{aliases} & = & [(\mathit{toName}\ \mathit{Parent}, \mathit{toName}\ a), \\ & & \ (\mathit{toName}\ \mathit{This}, \mathit{toName}\ b), \\ & & \ (\mathit{toName}\ \mathit{Root}, \mathit{toName}\ (0 :: \mathit{Int}))] \\ & +\!\!+ & [(\mathit{toName}\ (\mathit{Up}\ n), \mathit{toName}\ u) | (u,n) \leftarrow \mathit{zip}\ (b : \mathit{path})\ [0\,..]] \\ & +\!\!+ & [(\mathit{toName}\ (\mathit{Son}\ n), \mathit{toName}\ s) \\ & & | (\mathit{Node}'\ (\_,s,\_)\ \_\ \_, n) \leftarrow \mathit{zip}\ \mathit{ts}\ [0\,..]] \end{array}$$

Die Ersetzungsfunktion durchläuft einen Pfad komplett und führt alle Ersetzungen der Platzhalterliste aus.

$$\begin{array}{lcl} \mathit{replacePath} & :: & \mathit{Path} \to [(\mathit{Name},\mathit{Name})] \to \mathit{Path} \\ \mathit{replacePath}\ (\mathit{PathPoint}\ p)\ \mathit{al} & = & \mathit{PathPoint}\ (\mathit{replacePoint}\ p\ \mathit{al}) \\ \mathit{replacePath}\ \mathit{PathCycle}\ \_ & = & \mathit{PathCycle} \\ \mathit{replacePath}\ (\mathit{PathJoin}\ p_1\ \mathit{pj}\ p_2)\ \mathit{al} & & \end{array}$$

$$
\begin{array}{lll}
 & = & \mathit{PathJoin}\ (\mathit{replacePath}\ p_1\ \mathit{al}) \\
 & & \qquad (\mathit{replacePathElemDescr}\ \mathit{pj}\ \mathit{al}) \\
 & & \qquad (\mathit{replacePath}\ p_2\ \mathit{al}) \\
\multicolumn{3}{l}{\mathit{replacePath}\ (\mathit{PathEndDir}\ p\ d)\ \mathit{al}} \\
 & = & \mathit{PathEndDir}\ (\mathit{replacePoint}\ p\ \mathit{al})\ (\mathit{replaceDir}'\ d\ \mathit{al}) \\
\multicolumn{3}{l}{\mathit{replacePath}\ (\mathit{PathBuildCycle}\ p_1\ p_2)\ \mathit{al}} \\
 & = & \mathit{PathBuildCycle}\ (\mathit{replacePath}\ p_1\ \mathit{al})\ (\mathit{replacePath}\ p_2\ \mathit{al}) \\
\multicolumn{3}{l}{\mathit{replacePath}\ (\mathit{PathTransform}\ t\ p)\ \mathit{al}} \\
 & = & \mathit{PathTransform}\ t\ (\mathit{replacePath}\ p\ \mathit{al}) \\
\multicolumn{3}{l}{\mathit{replacePath}\ (\mathit{PathDefine}\ \mathit{eqs}\ p)\ \mathit{al}} \\
 & = & \mathit{PathDefine}\ (\mathit{replaceEquations}\ \mathit{eqs}\ \mathit{al}) \\
 & & \qquad (\mathit{replacePath}\ p\ \mathit{al})
\end{array}
$$

Die Ersetzung setzt sich auf Punkten fort, sowie auf den Typen *Numeric*, *PathElemDescr*, Dir' und *CutPic*.

$$
\begin{array}{lll}
\mathit{replacePoint} & :: & \mathit{Point} \rightarrow [(\mathit{Name}, \mathit{Name})] \rightarrow \mathit{Point} \\
\multicolumn{3}{l}{\mathit{replacePoint}\ (\mathit{PointVar}\ \mathit{name})\ \mathit{al}} \\
 & = & \mathit{PointVar}\ (\mathit{replaceName}\ \mathit{name}\ \mathit{al}) \\
\multicolumn{3}{l}{\mathit{replacePoint}\ (\mathit{PointPPP}\ c\ a\ b)\ \mathit{al}} \\
 & = & \mathit{PointPPP}\ c\ (\mathit{replacePoint}\ a\ \mathit{al})\ (\mathit{replacePoint}\ b\ \mathit{al}) \\
\multicolumn{3}{l}{\mathit{replacePoint}\ (\mathit{PointVec}\ (a, b))\ \mathit{al}} \\
 & = & \mathit{PointVec}\ (\mathit{replaceNumeric}\ a\ \mathit{al}, \mathit{replaceNumeric}\ b\ \mathit{al}) \\
\multicolumn{3}{l}{\mathit{replacePoint}\ (\mathit{PointMediate}\ a\ b\ c)\ \mathit{al}} \\
 & = & \mathit{PointMediate}\ (\mathit{replaceNumeric}\ a\ \mathit{al}) \\
 & & \qquad (\mathit{replacePoint}\ b\ \mathit{al}) \\
 & & \qquad (\mathit{replacePoint}\ c\ \mathit{al}) \\
\multicolumn{3}{l}{\mathit{replacePoint}\ (\mathit{PointDirection}\ a)\ \mathit{al}} \\
 & = & \mathit{PointDirection}\ (\mathit{replaceNumeric}\ a\ \mathit{al}) \\
\mathit{replacePoint}\ (\mathit{PointNeg}\ a)\ \mathit{al} & = & \mathit{PointNeg}\ (\mathit{replacePoint}\ a\ \mathit{al}) \\
\mathit{replacePoint}\ a\ \_ & = & a
\end{array}
$$

Wenn eine Variable gefunden wird, ist zu prüfen, ob einer ihrer Namensbestandteile in der Platzhalterliste vorhanden ist, um dort eine Ersetzung vorzunehmen.

$$
\begin{array}{lll}
\mathit{replaceName} & :: & \mathit{Name} \rightarrow [(\mathit{Name}, \mathit{Name})] \rightarrow \mathit{Name} \\
\mathit{replaceName}\ (\mathit{Hier}\ n\ n')\ \mathit{al} & = & \mathit{Hier}\ (\mathit{replaceName}\ n\ \mathit{al})\ (\mathit{replaceName}\ n'\ \mathit{al}) \\
\mathit{replaceName}\ n\ \mathit{al} & = & \mathit{replaceName}'\ n\ \mathit{al} \\
\quad \mathbf{where} & & \\
\quad \mathit{replaceName}'\ n\ [\,] & = & n \\
\quad \mathit{replaceName}'\ n\ ((n', r) : \mathit{al}) & & \\
\qquad\qquad |\, n \equiv n' & = & r \\
\qquad\qquad |\, \mathit{otherwise} & = & \mathit{replaceName}'\ n\ \mathit{al} \\
\mathit{replaceName}\ a\ \_ & = & a
\end{array}
$$

Damit haben wir eine sehr flexible und leistungsfähige Abstraktion für beliebige Bäume definiert. Werte, die wir nicht vorher kennen, haben wir mit Hilfe symbolischer Terme angegeben, die später während der Berechnung des Bildes durch METAPOST ausgerechnet werden. Der Originalalgorithmus von ANDREW KENNEDY [Ken93] markiert die Knoten eines Baumes mit dem Wert des horizontalen Abstandes zum Vaterknoten. Unser Algorithmus liefert keinen markierten Baum zurück, sondern eine Liste von Gleichungen, die die horizontalen Abstände der Knoten zum Vater beschreiben.

5.3.4 Gedanken zur Laufzeit

Die theoretische Laufzeit der Funktion *design* liegt bei $O(n^2)$, wobei n der Anzahl der Knoten entspricht. Jeder Knoten ist einmal zu besuchen und seine Hülle höchstens einmal zu verschieben. Dieses Verschieben mit der Funktion *moveExtent* geht in $O(n)$, womit sich die Laufzeit der Funktion ergibt.
In der Schlußbetrachtung der Arbeit [Ken93] hat KENNEDY die Idee, die Hülle nicht als Tupelliste absoluter Positionen zu verwalten, sondern mit Hilfe relativer Positionen. Dann ließe sich diese in konstanter Zeit verschieben, und somit eine Gesamtlaufzeit von $O(n)$ erreichen. Dabei wird der Algorithmus aber auch wesentlich unleserlicher. Die Funktionen *fit* und *merge* wären davon besonders betroffen. Eventuell müßten wir hier zusätzlich sehr viele Zwischenergebnisse einfügen, denn um die Abstände von Hüllen mit *maximum'* vergleichen zu können, müssen die relativen Werte aufaddiert sein. Viele Zwischenergebnisse bedeuten aber mehr Arbeit für METAPOST und Zeit, die im Layoutalgorithmus gespart wird, kann sich später bei der Berechnung der Grafik mit METAPOST wieder aufbrauchen. Aber auch aus einem anderen Grund erscheint eine Implementierung mit relativer Hüllenrepräsentierung nicht vielversprechend.
In die Rechenzeit zur Erstellung eines Baumes geht die der Funktion *edges*, die in den Kantenpfaden Platzhalter durch Knotennamen ersetzt, mit ein. Diese Funktion hat im worst–case eine Laufzeit von $O(n^2)$, wenn nicht zu viele Querkanten existieren. Dieser Wert ergibt sich aus *n* betrachteten Knoten und der Länge der Tabelle von Platzhaltern, die bedingt durch den Namen *Up* mit der Höhe des Baumes wächst.
Benchmarks mit zufälligen, allgemeinen Bäumen ergaben eine Laufzeit für den average–case von etwa $O(n \log n)$. Die Größe eines Blattes Papier beschränkt die sinnvolle Anzahl von Knoten auf natürliche Art und Weise. Bäume mit etwa 1000 Knoten sind noch in weniger als einer Stunde zu berechnen, aber erkennen lassen sich die einzelnen Knoten auch bei 600 DPI nicht mehr.

Kapitel 6

Die Grundbegriffe von METAPOST

Dieses Kapitel soll zum einen kurz die Möglichkeiten von METAPOST erläutern, zum anderen aber auch auf Einschränkungen hinweisen, die Auswirkungen auf das Design von *functional* METAPOST haben.

METAPOST [Hob89] basiert in den grundlegenden Funktionen auf Syntax und Semantik von METAFONT[Knu86]. Unterschiede liegen in den Tatsache begründet, daß METAFONT auf Rasterbildern operiert, während METAPOST PostScriptobjekte erzeugt, also eher mit Vektoren rechnet.

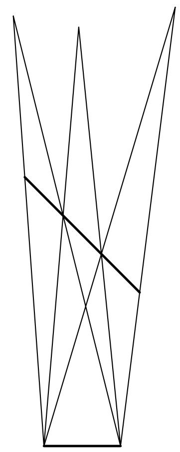
METAPOST unterscheidet die Datentypen `boolean`, `numeric`, `pair`, `color`, `path`, `pen`, `picture` und `transform`. Auf diesen Typen sind vielfältige Operationen definiert, die wir teilweise schon in Kapitel 3 kennengelernt haben. Eine genaue Kenntnis aller Befehle ist zum Verständnis später folgender Beispiele nicht notwendig. Daher wollen wir hier keine genaue Beschreibung von METAPOST geben, wie sie in [Hob92] zu finden ist, sondern die zugrundeliegende Philosophie vermitteln.

6.1 Ein einführendes Beispiel

Um ein Gefühl dafür zu bekommen, wie eine Bildbeschreibung in METAPOST aussieht, schauen wir uns ein erstes Beispiel an.

```
beginfig(1);
z1                  = -z2                 = (.2in,0);
xpart z3            = -xpart z6           =  .3in;
xpart z3 + ypart z3 = xpart z6 + ypart z6 = 1.1in;
z4  = 1/3[z3,z6];
z5  = 2/3[z3,z6];
z20 = whatever[z1,z3] = whatever[z2,z4];
z30 = whatever[z1,z4] = whatever[z2,z5];
z40 = whatever[z1,z5] = whatever[z2,z6];

draw z1--z20--z2--z30--z1--z40--z2;
draw z1--z2 withpen pencircle scaled 1pt;
draw z3--z6 withpen pencircle scaled 1pt;
endfig; end;
```



An diesem Beispiel läßt sich gut die typische Gliederung in zwei Phasen erkennen:

- Die **Layoutphase**, in der die beschreibenden Gleichungen aufgeführt sind und alle Werte ausgerechnet werden.
- Die **Zeichenphase**, die alle Zeichenbefehle enthält, die auf eventuell in der Layoutphase berechnetes zurückgreifen. Es ist wichtig, daß die Layoutphase hier abgeschlossen ist, d.h. allen Variablen Werte zugewiesen wurden. Der Versuch einen Pfad zu zeichnen, in dem irgendein Wert mit den bisher gelesenen Gleichungen nicht herleitbar ist, erzeugt unweigerlich einen Fehler.

6.2 Lineare und nichtlineare Gleichungssysteme

METAPOST erlaubt die Formulierung komplexer Gleichungssysteme, mit allen wichtigen Funktionen, sogar trigonometrischen. Diese Gleichungen versucht METAPOST zu lösen, indem es einen Abhängigkeitsgraph erzeugt und von Konstanten ausgehend, die Werte möglichst aller unbekannten Variablen herleitet. Wenn alle verwendeten Gleichungen linearer Natur sind, treten keine Probleme auf und die Reihenfolge, in der sie notiert wurden, spielt keine Rolle. In folgendem Beispiel erhält die Variable `a` den korrekten Wert, obwohl `b` und `c` erst später definiert sind.

```
a=b+c; b=2; c=3;
```

Probleme treten aber auf, wenn eine Gleichung nicht linear ist oder zwei Variablen multipliziert werden sollen, deren Werte beide noch nicht bekannt sind.

```
a=b*c; b=2; c=3;
>> b
>> c
! Not implemented: (unknown numeric)*(unknown numeric).
<to be read again>
                   ;
l.2 a=b*c;
```

Warum tritt hier die Fehlermeldung auf? Wenn der Gleichungslöser auf die Gleichung `a=b*c` trifft, kennt er die Werte von `b` und `c` noch nicht und das Lösungsverfahren muß abbrechen. Notiert der Anwender die Gleichungen jedoch in einer anderen Reihenfolge, kann die Gleichung `a=b*c` wie eine Zuweisung an `a` behandelt werden.

```
b=2; c=3; a=b*c;
```

Die Reihenfolge der Gleichungen kann also von entscheidender Bedeutung sein. Besondere Aufmerksamkeit auf die Reihenfolge ist nicht nur bei der Multiplikation zu legen, sondern auch bei vielen anderen Funktionen wie `max`, `sin`, `if` ... Diese Eigenschaft ist für eine deskriptive Sprache nicht besonders wünschenswert, aber auch Sprachen wie Prolog sind in dieser Hinsicht nicht unproblematisch.

Ein weiterer Grund für Reihenfolgeabhängigkeit ist die Existenz eines Zuweisungsoperators. Eine Zuweisung bindet den Wert erst nach ihrer Abarbeitung an die Variable und dieser Wert kann sich später in einer erneuten Zuweisung wieder ändern. Er gilt also lokal, d.h. nur zu einem bestimmten Zeitpunkt der Programmabarbeitung.

Eine andere Fehlerquelle sind überbestimmte Gleichungssysteme, die mit einer sofortigen Fehlermeldung quittiert werden. Jede Gleichung sollte nur neue Informationen transportieren.

```
x=0; y=10; y=x+10;
! Redundant equation.
<to be read again>
                   ;
l.2 x=0; y=10; y=x+10;
```

Natürlich müssen die Gleichungen auch konsistent sein.

```
x=0; y=10; y=x+20;
! Inconsistent equation (off by 10).
<to be read again>
                   ;
l.2 x=0; y=10; y=x+20;
```

Aus den ersten Blick wirken diese Eigenschaften vielleicht wie starke Einschränkungen. Es ist aber nicht schwer, entsprechende Fehler zu vermeiden.

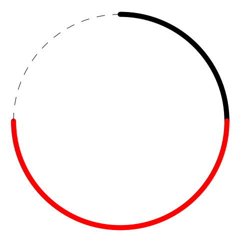
6.3 Pfade

Die Funktionalität von Pfaden entspricht der in *functional* METAPOST implementierten; abgesehen davon, daß ein Pfad nur Informationen über seine Form und keine Farb– oder Stiftattribute speichert. Ein Zeichenbefehl kann einen Pfad nur in einer homogenen Farbe mit einem bestimmten Stift und Muster zeichnen. Das bedeutet, daß Teilpfade mit unterschiedlichen Farben oder Stiften mit einzelnen Anweisungen zu zeichnen sind.

```
beginfig(20);
path p;

p = (0,40) .. (40,0) .. (0,-40) .. (-40,0) .. cycle;

draw subpath(0,1) of p withpen pencircle scaled 2pt;
draw subpath(1,3) of p withcolor (1,0,0)
                       withpen pencircle scaled 2pt;
draw subpath(3,4) of p withpen pencircle scaled 0.1pt
                       dashed evenly;
endfig; end;
```



6.4 Bildvariablen

Bildvariablen bedürfen noch einer besonderen Besprechung, da wir sie verwenden werden, um Teilbilder zwischenzuspeichern. Sie entsprechen etwa dem Datentyp *Picture* in *functional* METAPOST. Eine besondere Bildvariable ist `currentpicture`. Sie repräsentiert die aktuelle Zeichenfläche. Die Zuweisung `p:=currentpicture` speichert die Zeichenfläche, `currentpicture:=p` stellt sie wieder her.

6.5 Rahmenboxen

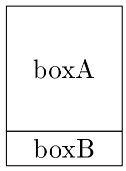
Von der Sprache PIC [Ker82] inspiriert, stellt METAPOST eine Makrosammlung zur Verfügung, mit der sich Bildvariablen in rechteckigen Boxen speichern und plazieren lassen.

```
boxit.boxA(btex boxA etex);
boxit.boxB(btex boxB etex);

boxA.dx = 10; boxA.dy = 20;
boxA.e - boxA.w = boxB.e - boxB.w;
fixsize(boxA, boxB);

boxA.s = boxB.n;
fixpos(boxA, boxB);

drawboxed(boxA, boxB);
```



Die Anweisung `boxit.`⟨*name*⟩`(`⟨*picture*⟩`)` erzeugt eine Rahmenbox mit dem Namen ⟨*name*⟩, die das Bild ⟨*picture*⟩ enthält. Eine solche Rahmenbox kann man als ein Objekt verstehen, welches neben dem Bild auch den Pfad der Umrandung, Variablen für die Bezugspunkte `c, .., nw` und die Variablen `dx` und `dy` enthält. Die Funktionen `pic <Name>` und `bpath <Name>` ermöglichen einen Zugriff auf das eingerahmte Bild bzw. auf den Pfad des Rahmens.
Bei einer neu erzeugten Rahmenbox sind die Werte der Variablen noch nicht vollständig bestimmt. Die Variablen `dx` und `dy` geben als eine Art dehnbare Länge den Abstand des Rahmens zum Bild an. Diese beiden Variablen können entweder direkt verändert werden, oder ihre Werte ergeben sich indirekt, wie durch die Wahl der Breite von `boxB` im Beispiel. Der Aufruf der Funktion `fixsize` legt die Werte von `dx` und `dy`, wenn dies noch nicht geschehen ist, fest; wie bei der Höhe von `boxB`. Die Anweisung `fixsize(`⟨*name*⟩`);` führt die folgenden Anweisungen aus:

```
if unknown ⟨name⟩.dx:  ⟨name⟩.dx=defaultdx; fi;
if unknown ⟨name⟩.dy:  ⟨name⟩.dy=defaultdy; fi;
```

Damit sind die wechselseitigen Abstände aller Bezugspunkte einer Rahmenbox festgelegt, nicht aber die absoluten Werte dieser Punkte. Eine solche Rahmenbox läßt sich jetzt noch absolut plazieren, indem man einem der Bezugspunkte einen absoluten Wert zuordnet oder zwei Rahmenboxen lassen sich mit einer relative Beziehung von Bezugspunkten „aneinanderketten". Z.B. die Beziehung `boxA.s = boxB.n;` plaziert die Rahmenboxen übereinander.
Die Anweisung `fixpos(boxA, boxB);` legt noch unbekannte Werte für Positionen fest, d.h. für die erste Rahmenbox wird festgelegt `boxA.c=(0,0)` und die Position der zweiten Rahmenbox ist damit determiniert.
Schließlich können die Rahmenboxen mit den Anweisung `drawboxed(boxA, boxB);` gerahmt oder mit `drawunboxed(boxA, boxB);` ungerahmt gezeichnet werden.
Da die Makrosammlung, die METAPOST bereitstellt nur einfache Rahmenformen kennt, müssen wir, um beliebige Rahmen zu erhalten, eigene Makros hierfür entwerfen. Auch Farbverläufe müssen wir genau wie Bitmaps aus einfacheren Befehlen zusammenbauen.

Kapitel 7

Implementierung der Kernsprache

Nachdem wir in den letzten Kapiteln die Bildbeschreibungssprache *functional* METAPOST kennengelernt und die Implementierung der Erweiterungen besprochen haben, kommen wir nun zu der Definition der Kernsprache und der Beschreibung der Bilderzeugung.
Das System der Bilderzeugung von *functional* METAPOST, setzt sich aus verschiedenen Sprachebenen zusammen, die sich, wie Abbildung 7.0.1 zeigt ineinander schachteln. Die erste Sprachebene bildet die Programmiersprache Haskell, in der Bildbeschreibung stattfindet, und in der wir auch den Compiler von *functional* METAPOST implementieren. Dieser Teil des Systems zur Bilderzeugung, den wir in dieser Arbeit entwerfen, wird vom gestrichelten Rahmen umschlossen. Der Sprachstandard unserer Beschreibungssprache besteht aus der Kernsprache und den Anwendungen, die auf diese zurückgreifen. In Kapitel 5 haben wir die Implementierung der Erweiterungen bereits besprochen. Die grauen Bereiche und die dunklen Pfeile definieren die verbleibende Aufgabenstellung dieses Kapitels. Wir werden zuerst eine Idee vermitteln, wie die Datentypen und Funktionen der Kernsprache definiert sind. Danach entwerfen wir einen Compiler, der Bildbeschreibungen in eine abstrakte Zwischensprache übersetzt, die mit einem speziellen Haskell–Datentyp eine Repräsentierung von METAPOST–Programmen erlaubt. Aus dieser abstrakten Sprache wird mit Hilfe eines Pretty Printers auf effiziente Weise der Quellcode einer METAPOST–Bildbeschreibung generiert.

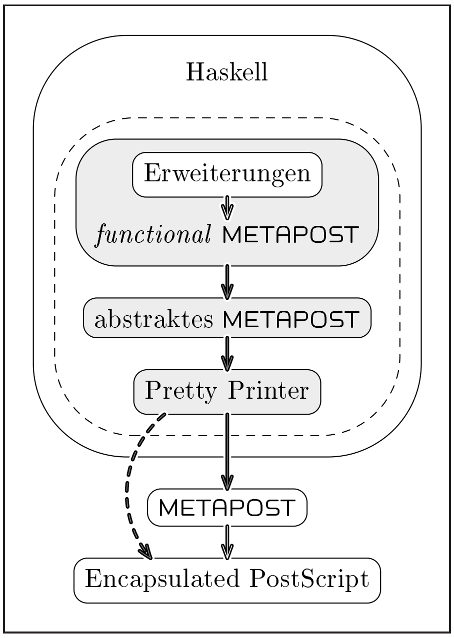


Abbildung 7.0.1: Die verschiedenen Sprachebenen.

Das Programm METAPOST erzeugt aus dieser Bildbeschreibung ein Encapsulated ☞ *PostScript*–Bild. Wir werden METAPOST um zusätzliche Funktionen für Farbverläufe und Bitmaps erweitern. Im Fall von Bitmaps müssen wir die Sprachschicht von METAPOST überspringen und direkt in die Generierung des Encapsulated PostScript–Bildes eingreifen. Dies symbolisiert der gestrichelte Pfeil.

7.1 Die Verteilung der Aufgaben auf Module

Wir beginnen mit einem Überblick der verschiedenen zu lösenden Problembereiche und verteilen die anfallende Aufgaben auf überschaubare Module. Abbildung 7.1.1 zeigt, in welcher Weise diese Module aufeinander aufbauen.

Die Definitionen der Datenstrukturen und Funktionen der Sprache *functional* METAPOST sind auf die drei Module verteilt, die der Anwender als Schnittstelle zur Bildbeschreibung sieht. In Abbildung 7.1.1 sind diese Module grau dargestellt.

FMPTypes führt die Typen *Point*, *Numeric* und *Name* ein.

FMPColor definiert den Typ *Color* und einige Typinstanzen davon.

FMPPicture enthält die übrigen Typen wie *Picture*, *Path*, *Area*, ... zusätzlich grundlegende Funktionen wie (⫴), die Pfadkonstruktoren, sowie Klassen und Instanzdefinitionen der Attributierungsfunktionen.

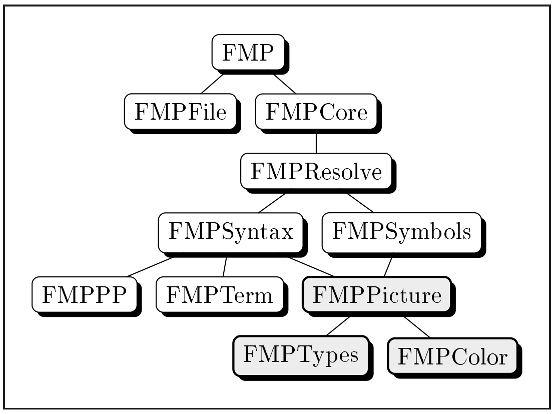


Abbildung 7.1.1: Die Module des Compilers und deren Import–Abhängigkeiten.

Während der Übersetzung werden Ausdrücke der Sprache *functional* METAPOST in Ausdrücke einer abstrakten METAPOST–Sprache überführt. Siehe dazu Abbildung 7.0.1 der vorigen Seite. Die Definition dieser abstrakten Sprache verteilt sich auf zwei Module.

FMPTerm exportiert den Typ *Term* für abstrakte Terme.

FMPSyntax definiert den Typ *MetaPost*, mit dem sich METAPOST–Programme darstellen lassen und stellt Funktionen zur Überführung der *functional* METAPOST–Typen in die abstrakte Sprache bereit. Ferner sind in diesem Modul Funktionen zur Umwandlung der abstrakten Sprache in den Typ *Doc*, der eine effiziente und komfortable Ausgabe erlaubt, enthalten.

FMPPP bietet einen speziell für unser Problem optimierten Pretty Printer

Die nächsten beiden Module verwalten die Punkt– und numerischen Variablen und die Benennungen von Bildern.

FMPSymbols definiert die Typen zur Variablenverwaltung.

FMPResolve enthält Funktionen zur Verwaltung von Variablen und der Auflösung von Referenzen.

Wir benötigen noch eine Funktion, die Ausdrücke des Typs *Picture* in eine Folge von abstrakten METAPOST–Befehlen überführt.

FMPCore erledigt diese Aufgabe der Codegenerierung.

Schließlich sind noch verschiedene Aufgaben von Generierung der Bilder, bis zum Aufruf des Programms METAPOST zu koordinieren.

FMPFile verwaltet eine Datei, die allgemeine Voreinstellungen enthält.

FMP exportiert die Funktion, die der Anwender aufruft, um ein Bild zu erzeugen.

Bei der nachfolgenden Diskussion der Implementierung werden wir uns auf die wichtigen und interessanten Teile beschränken. Die Intention besteht nicht darin, das komplette Programm aufzulisten, das schon ohne Kommentare ca. 100 Seiten füllt, sondern eine Idee seiner grundlegenden Funktionsweise zu vermitteln.

7.2 Die Sprachdefinition von *functional* METAPOST

In diesem Abschnitt betrachten wir die Datentypen, Typklassen und Funktionen zur Bildbeschreibung, die für den Anwender von *functional* METAPOST sichtbar sind. Diese Definitionen verteilen sich auf die drei Module FMPColor, FMPTypes und FMPPicture. Eine stärkere Gliederung ist mit dem Interpreter ☞*Hugs* leider nicht möglich, da dieser keine zyklischen Abhängigkeiten zwischen Funktionen und Datentypen verschiedener Module verwalten kann.

7.2.1 FMPColor

Es gibt drei verschiedene Arten von Farben. Die Farbe *DefaultColor* zeigt an, daß für ein Bild oder einen Pfad keine Veränderung der Farbe erfolgen soll. Der Konstruktor *Color* repräsentiert eine normale Farbe im RGB–Farbraum, während *Graduate* einen linearen Farbverlauf zwischen zwei Farben unter einem bestimmten Winkel in n Abstufungen angibt.

$$
\begin{array}{lll}
\textbf{data}\ \mathit{Color} & = & \mathit{DefaultColor} \\
 & | & \mathit{Color\ Double\ Double\ Double} \\
 & | & \mathit{Graduate\ Color\ Color\ Double\ Int} \\
 & & \textbf{deriving}\ (\mathit{Eq}, \mathit{Show}, \mathit{Read})
\end{array}
$$

Zur Farbattributierung aller denkbaren Objekte, nicht nur Bildern, sondern auch Bäumen oder Turtlepfaden, exportiert diese Modul die Klassen *HasColor* und *HasBGColor*.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{class}\ \mathit{HasColor}\ a\ \textbf{where}} \\
\quad \mathit{setColor} & :: & \mathit{Color} \to a \to a \\
\quad \mathit{setDefaultColor} & :: & a \to a \\
\quad \mathit{getColor} & :: & a \to \mathit{Color}
\end{array}
$$

Neben Konstanten für einige oft benötigte Farben wie *green*, gibt es auch Funktionen, um Farbverläufe anzugeben.

$$
\begin{array}{lll}
\mathit{graduate} & :: & \mathit{Color} \to \mathit{Color} \to \mathit{Double} \to \mathit{Int} \to \mathit{Color} \\
\mathit{graduate}\ c_1\ c_2\ a\ n & = & \mathit{Graduate}\ c_1\ c_2\ a\ n
\end{array}
$$

Um nicht immer die Anzahl der Zwischenfarben angeben zu müssen, existieren Funktionen für Farbverläufe niedriger (*graduateLow*), mittlerer (*graduateMed*) und hoher Qualität.

$$
\begin{array}{lll}
\mathit{graduateHigh} & :: & \mathit{Color} \to \mathit{Color} \to \mathit{Double} \to \mathit{Color} \\
\mathit{graduateHigh}\ c_1\ c_2\ a & = & \mathit{graduate}\ c_1\ c_2\ a\ 256
\end{array}
$$

Damit wir mit Farben auch rechnen können, wird *Color* Instanz von *Num* und *Fractional*. Eine Instanz von *Fractional* ist auf den ersten Blick nicht nötig, da eine Division von Farben nur beschränkt Sinn macht. Aber es ermöglicht durch die Funktion *fromRational* eine automatische Umwandlung von Brüchen und Fließkommazahlen in Grautöne. Somit wählt *setColor* 0.5 ein mittleres Grau.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{instance } \mathit{Fractional\ Color} \textbf{ where}} \\
\quad \mathit{Color}\ r_1\ g_1\ b_1\ /\ \mathit{Color}\ r_2\ g_2\ b_2 & & \\
 & = & \mathit{Color}\ (r_1\ /\ r_2)\ (g_1\ /\ g_2)\ (b_1\ /\ b_2) \\
\quad \mathit{Graduate}\ c_1\ c_2\ a\ n\ /\ c_3@(\mathit{Color}\ \_\ \_\ \_) & & \\
 & = & \mathit{Graduate}\ (c_1\ /\ c_3)\ (c_2\ /\ c_3)\ a\ n \\
\quad c_3@(\mathit{Color}\ \_\ \_\ \_)\ /\ \mathit{Graduate}\ c_1\ c_2\ a\ n & & \\
 & = & \mathit{Graduate}\ (c_3\ /\ c_1)\ (c_3\ /\ c_2)\ a\ n \\
\quad \mathit{Graduate}\ c_1\ c_2\ a\ n\ /\ \mathit{Graduate}\ c_3\ c4\ a'\ n' & & \\
 & = & \mathit{Graduate}\ (c_1\ /\ c_3)\ (c_2\ /\ c4) \\
 & & \qquad ((a + a')\ /\ 2)\ (\mathit{maximum}\ [n, n']) \\
\quad a\ /\ \_ & = & a \\
\quad \mathit{recip}\ (\mathit{Color}\ r\ g\ b) & = & \mathit{Color}\ (\mathit{recip}\ r)\ (\mathit{recip}\ g)\ (\mathit{recip}\ b) \\
\quad \mathit{recip}\ a & = & a \\
\quad \mathit{fromRational}\ i & = & \mathit{Color}\ f\ f\ f \\
\qquad\qquad \textbf{where}\ f & = & \mathit{fromRational}\ i
\end{array}
$$

Es kann manchmal nützlich sein, Farben im HSV–Farbraum anzugeben. Deshalb gibt es eine Konvertierungsfunktion in Anlehnung an [Fel92] Kapitel 3.5.

$$
\begin{array}{lll}
\mathit{hsv2rgb} & :: & (\mathit{Double}, \mathit{Double}, \mathit{Double}) \rightarrow \mathit{Color} \\
\mathit{hsv2rgb}\ (\_, 0, v) & = & \mathit{Color}\ v\ v\ v \\
\mathit{hsv2rgb}\ (h, s, v) & = & \textbf{case}\ i'\ \textbf{of} \\
 & & \quad 0 \rightarrow \mathit{Color}\ v\ t_3\ t_1 \\
 & & \quad 1 \rightarrow \mathit{Color}\ t_2\ v\ t_1 \\
 & & \quad 2 \rightarrow \mathit{Color}\ t_1\ v\ t_3 \\
 & & \quad 3 \rightarrow \mathit{Color}\ t_1\ t_2\ v \\
 & & \quad 4 \rightarrow \mathit{Color}\ t_3\ t_1\ v \\
 & & \quad \_ \rightarrow \mathit{Color}\ v\ t_1\ t_2 \\
\qquad \textbf{where} & & \\
\qquad h' & = & h\ /\ 60.0 \\
\qquad i' & = & \mathit{mod}\ (\mathit{floor}\ h')\ 6 \\
\qquad i & = & \mathit{floor}\ h' \\
\qquad \mathit{fract} & = & h' - \mathit{fromInt}\ i \\
\qquad t_1 & = & v * (1 - s) \\
\qquad t_2 & = & v * (1 - s * \mathit{fract}) \\
\qquad t_3 & = & v * (1 - s * (1 - \mathit{fract}))
\end{array}
$$

Diese Funktionen reichen für eine komfortable Beschreibung von Farben aus, wenn auch noch eine Erweiterung um zusätzliche Farbmodelle denkbar wäre. Leider ist es mit PostScript nicht möglich, (halb–)transparente Farben zu definieren, sonst würde das noch professionellere Bildmanipulationen ermöglichen.

7.2.2 FMPTypes

Im Modul FMPTypes sind alle Datentypen der Sprache *functional* METAPOST definiert, die vom Datentyp *Picture* unabhängig sind, d.h. *Picture* kommt in keinem ihrer Konstruktoren direkt oder indirekt vor.

Immer, wenn ein Funktionsname für verschiedene Typen anwendbar sein soll, können wir das erreichen, indem wir Typklassen bilden. Der Ausdruck *relax* steht z.B. immer für ein neutrales Element, d.h. ein leeres Bild, ein leerer Turtlepfad usw.

$$
\begin{array}{lll}
\textbf{class}\ \mathit{HasRelax}\ a\ \textbf{where} & & \\
\quad \mathit{relax} & :: & a
\end{array}
$$

Auch die Funktionen *med* und *cond* wollen im Zusammenhang mit Werten der Typen *Numeric* und *Point* benutzen können.

$$
\begin{array}{lll}
\textbf{class}\ \mathit{HasMed}\ a\ \textbf{where} & & \\
\quad \mathit{med} & :: & \mathit{Numeric} \to a \to a \to a
\end{array}
$$

$$
\begin{array}{lll}
\textbf{class}\ \mathit{HasCond}\ a\ \textbf{where} & & \\
\quad \mathit{cond} & :: & \mathit{Boolean} \to a \to a \to a
\end{array}
$$

Namen dürfen aus den drei Typen *Int*, *String*, und *Dir* gebildet werden. Der Konstruktor *Hier* ermöglicht die hierarchische Kombination von Namen und *Global* zeigt an, daß es sich nicht um ein definierendes Variablenvorkommen, sondern um ein angewandtes Vorkommen, also eine Referenz handelt.

$$
\begin{array}{lll}
\textbf{data}\ \mathit{Name} & = & \mathit{NameInt}\ \mathit{Int} \\
 & \mid & \mathit{NameStr}\ \mathit{String} \\
 & \mid & \mathit{NameDir}\ \mathit{Dir} \\
 & \mid & \mathit{Hier}\ \mathit{Name}\ \mathit{Name} \\
 & \mid & \mathit{Global}\ \mathit{Name} \\
 & & \textbf{deriving}\ (\mathit{Show}, \mathit{Read}, \mathit{Eq}, \mathit{Ord})
\end{array}
$$

Ähnlich wie bei Bildern, wollen wir auch für Namen das Konzept von Unterklassen, mit einer automatischen Umwandlung aus anderen Typen in Namen, erreichen. Deshalb bilden wir eine Typklasse, die eine Abstraktion für diese Untertypen bildet. Siehe hierzu auch Abschnitt 3.16.

$$
\begin{array}{lll}
\textbf{class}\ \mathit{IsName}\ a\ \textbf{where} & & \\
\quad \mathit{toName} & :: & a \to \mathit{Name} \\
\quad \mathit{toNameList} & :: & [a] \to \mathit{Name} \\
\quad \mathit{toNameList}\ [l] & = & \mathit{toName}\ l \\
\quad \mathit{toNameList}\ (l : ls) & = & \mathit{Hier}\ (\mathit{toName}\ l)\ (\mathit{toNameList}\ ls)
\end{array}
$$

Etwas umständlich ist dieser Weg für Strings, die den Typ $[\mathit{Char}]$ haben. Die Definition

$$\textbf{instance}\ \mathit{IsName}\ [\mathit{Char}]\ \textbf{where}..$$

ist im aktuellen Haskell Sprachstandard nicht zulässig, da es sich um einen zusammengesetzen Typ handelt und Instanzen nur für einfache Typen definiert werden können. Stattdessen gehen wir einen Umweg und geben eine Instanz für den allgemeinen Listentyp an.

$$
\begin{array}{lll}
\textbf{instance}\ (\mathit{IsName}\ a) \Rightarrow \mathit{IsName}\ [a]\ \textbf{where} & & \\
\quad \mathit{toName} & = & \mathit{toNameList}
\end{array}
$$

Wenn wir für den Typ *Char* ebenfalls eine Instanz von *IsName* angeben, wird bei der Reduktion des Ausdrucks *toName* `"picture_name"`, die Funktion *toNameList* der Instanz für *Char* verwendet.

$$\begin{array}{l}\textbf{instance}\ \mathit{IsName}\ \mathit{Char}\ \textbf{where}\\ \qquad \mathit{toName}\ n \qquad = \mathit{NameStr}\ [n]\\ \qquad \mathit{toNameList} \qquad = \mathit{NameStr}\end{array}$$

Nachdem wir die Darstellung von Namen besprochen haben, wollen wir nun noch die Definition der beiden wichtigen Typen *Numeric* und *Point* vorstellen.
Im Datentyp für numerische Werte können wir Konstruktoren für die Darstellung von Variablen, Konstanten, den Ausdruck *whatever* und einige andere Funktionen finden, siehe Abbildung 7.2.1. Die Aufgabe der beiden ersten Konstruktoren *NumericVar'* und *NumericArray'* wird uns erst in Abschnitt 7.5 beschäftigen, in dem wir die Verwaltung der Variablen besprechen. Aber warum existieren keine Konstruktoren für die Addition von Zahlen und derartige Operationen und was bedeuten Konstruktoren wie *NumericPN*?

$$\begin{array}{lcl}\textbf{data}\ \mathit{Numeric} & = & \mathit{NumericVar'}\ \mathit{Int}\ \mathit{Int}\\ & \mid & \mathit{NumericArray'}\ \mathit{Int}\ \mathit{Int}\\ & \mid & \mathit{NumericVar}\ \mathit{Name}\\ & \mid & \mathit{Numeric}\ \mathit{Double}\\ & \mid & \mathit{NumericWhatever}\\ & \mid & \mathit{NumericDist}\ \mathit{Point}\ \mathit{Point}\\ & \mid & \mathit{NumericMediate}\ \mathit{Numeric}\ \mathit{Numeric}\ \mathit{Numeric}\\ & \mid & \mathit{NumericPN}\ \mathit{FunPN}\ \mathit{Point}\\ & \mid & \mathit{NumericNN}\ \mathit{FunNN}\ \mathit{Numeric}\\ & \mid & \mathit{NumericNNN}\ \mathit{FunNNN}\ \mathit{Numeric}\ \mathit{Numeric}\\ & \mid & \mathit{NumericNsN}\ \mathit{FunNsN}\ [\mathit{Numeric}]\\ & \mid & \mathit{NumericCond}\ \mathit{Boolean}\ \mathit{Numeric}\ \mathit{Numeric}\\ & & \textbf{deriving}\ (\mathit{Eq}, \mathit{Show}, \mathit{Read}, \mathit{Ord})\end{array}$$

Abbildung 7.2.1: Der Datentyp *Numeric*.

Damit hat es folgende Bewandtnis: Wenn wir für jede Funktion, die *functional* METAPOST kennt, einen Konstruktor definieren würden, wären dies 25 Stück, die beim Durchlaufen der Datenstruktur viele Mustervergleiche nach sich zögen. Das wäre weder schnell noch elegant. Einige dieser Konstruktoren würden sich allerdings nur durch ihren Namen und nicht die Typen ihrer Argumente unterscheiden. Wir fassen diese Fälle dann zu einem zusammen. Es existiert nur ein Konstruktor *NumericPN* für alle Operatoren des Typs $\mathit{Point} \to \mathit{Numeric}$, *NumericNN* für den Typ $\mathit{Numeric} \to \mathit{Numeric}$, *NumericNNN* für $\mathit{Numeric} \to \mathit{Numeric} \to \mathit{Numeric}$ und *NumericNsN* für $[\mathit{Numeric}] \to \mathit{Numeric}$. Es gibt z.B. drei Funktionen des Typs $\mathit{Point} \to \mathit{Numeric}$.

$$\begin{array}{lcl}\textbf{data}\ \mathit{FunPN} & = & \mathit{PNXPart}\\ & \mid & \mathit{PNYPart}\\ & \mid & \mathit{PNAngle}\\ & & \textbf{deriving}\ (\mathit{Eq}, \mathit{Show}, \mathit{Read}, \mathit{Ord})\end{array}$$

und sechs Funktionen des Typs $Numeric \to Numeric \to Numeric$.

$$
\begin{array}{lll}
\textbf{data}\ \mathit{FunNNN} & = & \mathit{NNNAdd} \\
 & | & \mathit{NNNSub} \\
 & | & \mathit{NNNMul} \\
 & | & \mathit{NNNDiv} \\
 & | & \mathit{NNNPyth} \\
 & | & \mathit{NNNPower} \\
 & & \textbf{deriving}\ (\mathit{Eq}, \mathit{Show}, \mathit{Read}, \mathit{Ord})
\end{array}
$$

Die Funktion *xpart* notieren wir mit diesem Ansatz wie folgt.

$$
\begin{array}{lll}
\mathit{xpart} & :: & \mathit{Point} \to \mathit{Numeric} \\
\mathit{xpart}\ (\mathit{PointVec}\ (a, \_)) & = & a \\
\mathit{xpart}\ a & = & \mathit{NumericPN}\ \mathit{PNXPart}\ a
\end{array}
$$

Wir haben zwar auf der einen Seite die Zahl der Konstruktoren verringern können, auf der anderen Seite ist aber ein zusätzlicher Parameter für den Funktionsnamen hinzugekommen. Trotzdem ist unser Ansatz sehr sinnvoll, denn Funktionen wie *replacePoint* auf Seite 86 werden jetzt wesentlich kürzer. Analog zu numerischen Ausdrücken definieren wir den Datentyp für Punkte, wie in Abbildung 7.2.2 zu sehen ist. Auch hier fassen wir Operationen, die auf den gleichen Typen stattfinden, in einem einzigen Konstruktor zusammen.

$$
\begin{array}{lll}
\textbf{data}\ \mathit{Point} & = & \mathit{PointPic'}\ \mathit{Int}\ \mathit{Dir} \\
 & | & \mathit{PointVar'}\ \mathit{Int}\ \mathit{Int} \\
 & | & \mathit{PointVarArray'}\ \mathit{Int}\ \mathit{Int} \\
 & | & \mathit{PointTrans'}\ \mathit{Point}\ [\mathit{Int}] \\
 & | & \mathit{PointVar}\ \mathit{Name} \\
 & | & \mathit{PointVec}\ (\mathit{Numeric}, \mathit{Numeric}) \\
 & | & \mathit{PointMediate}\ \mathit{Numeric}\ \mathit{Point}\ \mathit{Point} \\
 & | & \mathit{PointDirection}\ \mathit{Numeric} \\
 & | & \mathit{PointWhatever} \\
 & | & \mathit{PointPPP}\ \mathit{FunPPP}\ \mathit{Point}\ \mathit{Point} \\
 & | & \mathit{PointNMul}\ \mathit{Numeric}\ \mathit{Point} \\
 & | & \mathit{PointNeg}\ \mathit{Point} \\
 & | & \mathit{PointCond}\ \mathit{Boolean}\ \mathit{Point}\ \mathit{Point} \\
 & & \textbf{deriving}\ (\mathit{Eq}, \mathit{Show}, \mathit{Read}, \mathit{Ord})
\end{array}
$$

Abbildung 7.2.2: Der Datentyp *Point*.

Gleichungen können zwischen numerischen Werten oder Punkten bestehen. Mehrere Gleichungen lassen sich mit Hilfe des Konstruktors *Equations* effizient konkatenieren und *EquationCond* ermöglicht bedingte Gleichungen.

$$
\begin{array}{lll}
\textbf{data}\ \mathit{Equation} & = & \mathit{NEquations}\ [\mathit{Numeric}] \\
 & | & \mathit{PEquations}\ [\mathit{Point}] \\
 & | & \mathit{Equations}\ [\mathit{Equation}] \\
 & | & \mathit{EquationCond}\ \mathit{Boolean}\ \mathit{Equation}\ \mathit{Equation} \\
 & & \textbf{deriving}\ (\mathit{Eq}, \mathit{Show}, \mathit{Read})
\end{array}
$$

Der Gleichheitsoperator soll für numerische Werte und Punkte der gleiche sein. Deshalb definieren wir eine Klasse und die entsprechenden Instanzen für *Numeric* und *Point*.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{class}\ \mathit{IsEquation}\ a\ \textbf{where}} \\
\quad (\doteq) & :: & a \to a \to \mathit{Equation} \\
\quad \mathit{equal} & :: & [a] \to \mathit{Equation}
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{instance}\ \mathit{IsEquation}\ \mathit{Point}\ \textbf{where}} \\
\quad p_1 \doteq p_2 & = & \mathit{PEquations}\ [p_1, p_2] \\
\quad \mathit{equal} & = & \mathit{PEquations}
\end{array}
$$

Den Typ *Boolean* und die Operatoren darauf definieren wir ähnlich, wie wir es für den Typ *Numeric* taten. Von der Typklasse

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{class}\ \mathit{HasCond}\ a\ \textbf{where}} \\
\quad \mathit{cond} & :: & \mathit{Boolean} \to a \to a \to a
\end{array}
$$

können wir Instanzen der Typen bilden, für die wir die Formulierung von Bedingungen zulassen wollen. Dies sind Typen *Equation*, *Numeric* und *Point*.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{instance}\ \mathit{HasCond}\ \mathit{Equation}\ \textbf{where}} \\
\quad \mathit{cond}\ b\ t\ e & = & \mathit{EquationCond}\ b\ t\ e
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{instance}\ \mathit{HasCond}\ \mathit{Point}\ \textbf{where}} \\
\quad \mathit{cond}\ b\ t\ e & = & \mathit{PointCond}\ b\ t\ e
\end{array}
$$

Weiterhin bilden wir die booleschen Operationen auf die Operatoren der Grundrechenarten ab, indem der Typ *Boolean* Instanz der Klasse *Num* wird. So oder ähnlich sind alle Funktionen und Datentypen dieses Moduls definiert, so daß wir uns nun der Konstruktion von Bildern zuwenden werden.

7.2.3 FMPPicture

In diesem Modul versammeln sich alle Datentypen und Funktionen, die so eng mit dem Datentyp *Picture* verbunden sind, daß sie sich nicht in andere Module auslagern lassen, ohne eine wechselseitige Modulabhängigkeit zu verursachen, denn solche wechselseitigen Abhängigkeiten zwischen verschiedenen Modulen kann der Interpreter Hugs leider nicht verwalten. Hier finden sich etwa vierzig Datentypen und weit über hundert Funktionen, so daß wir uns auf drei Aspekte beschränken wollen: Die Verwaltung der Attribute, die Beschreibung von Rahmenkonstruktoren und die Pfadkonstruktoren. Wir tun dies jedoch nicht, ohne vorher mit Abbildung 7.2.3 den Typ *Picture* vorgestellt zu haben. Die Funktion der einzelnen Konstruktoren liegt nach der Einführung in *functional* METAPOST auf der Hand, abgesehen von *Attributes*. Mit *Attributes* erfolgt die Attributierung der Bilder. Ein Record des Typs *Attrib* speichert eventuell Namen, Farbe und Hintergrundfarbe eines Bildes.

$$
\begin{array}{lll}
\textbf{data}\ \mathit{Picture} & = & \mathit{Attributes\ Attrib\ Picture} \\
 & | & \mathit{Overlay}\ [\mathit{Equation}]\ (\mathit{Maybe\ Int})\ [\mathit{Picture}] \\
 & | & \mathit{Define}\ [\mathit{Equation}]\ \mathit{Picture} \\
 & | & \mathit{Frame\ FrameAttrib}\ [\mathit{Equation}]\ \mathit{Path\ Picture} \\
 & | & \mathit{Draw}\ [\mathit{Path}]\ \mathit{Picture} \\
 & | & \mathit{Fill}\ [\mathit{Area}]\ \mathit{Picture} \\
 & | & \mathit{Clip\ Path\ Picture} \\
 & | & \mathit{Empty\ Numeric\ Numeric} \\
 & | & \mathit{Tex\ String} \\
 & | & \mathit{Text\ String} \\
 & | & \mathit{BitLine\ Point\ BitDepth\ String} \\
 & | & \mathit{PTransform\ Transformation\ Picture} \\
 & | & \mathit{TrueBox\ Picture} \\
 & & \textbf{deriving}\ (\mathit{Eq}, \mathit{Show}, \mathit{Read})
\end{array}
$$

Abbildung 7.2.3: Der Datentyp *Picture*.

$$
\begin{array}{lll}
\textbf{data}\ \mathit{Attrib} & = & \mathit{Attrib}\{\mathit{aNames} :: [\mathit{Name}], \\
 & & \qquad \mathit{aColor} :: \mathit{Color}, \\
 & & \qquad \mathit{aBGColor} :: \mathit{Color}\} \\
 & & \textbf{deriving}\ (\mathit{Eq}, \mathit{Read})
\end{array}
$$

Warum enthält dann nicht jeder *Picture*–Konstruktor diesen Record? Auf diese Weise müssen die Attributierungsfunktionen nicht für jeden *Picture*–Konstruktor ein eigenes Muster, bzw. eine eigene Regel anbieten. Erst bei der ersten Veränderung eines Attributes, werden die Attributierungsinformationen in den Ausdruck aufgenommen, indem der Konstruktor *Attributes* mit den Attributinformationen vor den Ausdruck gestellt wird.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{instance}\ \mathit{HasColor\ Picture}\ \textbf{where}} \\
\quad \mathit{setColor}\ c\ (\mathit{Attributes\ as\ p}) & & \\
 & = & \mathit{Attributes\ as}\{\mathit{aColor} = c\}p \\
\quad \mathit{setColor}\ c\ p & = & \mathit{Attributes\ stdAttrib}\{\mathit{aColor} = c\}p \\
\quad \mathit{setDefaultColor} & = & \mathit{setColor\ DefaultColor} \\
\quad \mathit{getColor}\ (\mathit{Attributes\ as}\ \_) & & \\
 & = & \mathit{aColor\ as} \\
\quad \mathit{getColor}\ \_ & = & \mathit{DefaultColor}
\end{array}
$$

Kommen wir nun zu Rahmen. Der Rahmenkonstruktor *Frame* speichert neben Rahmenattributen ein Gleichungssystem, das die Form und Ausmaße des Rahmens berechnet und den Rahmenpfad. Die Attributmenge ist für Rahmen etwas umfangreicher als für Bilder. Attribute wie *faColor*, *faPen*, *faPattern* oder *faVisible* beeinflussen die Gestalt des Rahmenpfades.

$$
\begin{array}{lll}
\textbf{data}\ \mathit{FrameAttrib} & = & \mathit{FrameAttrib}\{\mathit{faNames} :: [\mathit{Name}],\\
 & & \qquad \mathit{faColor},\\
 & & \qquad \mathit{faBGColor} :: \mathit{Color},\\
 & & \qquad \mathit{faPen} :: \mathit{Pen},\\
 & & \qquad \mathit{faPattern} :: \mathit{Pattern},\\
 & & \qquad \mathit{faShadow} :: \mathit{Maybe}\ (\mathit{Numeric}, \mathit{Numeric}),\\
 & & \qquad \mathit{faVisible} :: \mathit{Bool}\}\\
 & & \textbf{deriving}\ (\mathit{Eq}, \mathit{Read})
\end{array}
$$

Attribute, die die Rahmengröße speichern könnten, fehlen bewußt, da die Art und Weise, die Form eines Rahmens mit den Attributen *dx* und *dy* oder durch Angabe von Höhe und Breite festzulegen, schon eine gewisse Rahmenform impliziert. Es wären aber Rahmentypen denkbar, die neben den Attributierungsfunktionen *setDX*, *setDY*, *setWidth* und *setHeight* noch weitere Eingriffsmöglichkeiten bieten, oder diese nicht bereitstellen. Deshalb speichert nur der Typ *Frame* solche speziellen Informationen.

$$
\begin{array}{lll}
\textbf{data}\ \mathit{Frame} & = & \mathit{Frame'}\ \mathit{FrameAttrib}\ \mathit{ExtentAttrib}\ \mathit{Path}\ \mathit{Picture}\\
 & & \textbf{deriving}\ \mathit{Show}
\end{array}
$$

Theoretisch ließe sich zusätzlich noch ein Typ *RoundFrame* definieren, der andere Möglichkeiten bietet, die Rahmenform zu verändern und daher auch eine andere Attributmenge hierfür besitzt. In unserem Fall speichern die Attribute *eaX* und *eaX* die gewüschte Form des Rahmens. Die restlichen Variablen des Records sind keine veränderbaren Attribute, sondern speichern Gleichungen, die je nach der gewählten Form, zusammen mit den Gleichungen *eaEqs* zu einer vollständigen Beschreibung des Rahmens kombiniert werden.

$$
\begin{array}{lll}
\textbf{data}\ \mathit{ExtentAttrib} & = & \mathit{ExtentAttrib}\{\mathit{eaX}, \mathit{eaY} :: \mathit{AbsOrRel},\\
 & & \qquad \mathit{eaEqsDX} :: [\mathit{Equation}],\\
 & & \qquad \mathit{eaEqsDY} :: [\mathit{Equation}],\\
 & & \qquad \mathit{eaEqsWidth} :: [\mathit{Equation}],\\
 & & \qquad \mathit{eaEqsHeight} :: [\mathit{Equation}],\\
 & & \qquad \mathit{eaEqs} :: [\mathit{Equation}]\}\\
 & & \textbf{deriving}\ \mathit{Show}
\end{array}
$$

Damit wir die beiden Möglichkeiten Breite und Höhe eines Rahmen zu beeinflussen modellieren können, definieren wir hierfür einen Datentyp. Die erste Möglichkeit ist die Angabe einer absoluten Breite oder Höhe mit den Attributierungsfunktionen *setWidth* und *setHeight*. Die zweite Moglichkeit, die wir durch die Funktionen *setDX* und *setDY* erhalten, beschreibt die Ausmaße relativ zum umrahmten Bild.

$$
\begin{array}{lll}
\textbf{data}\ \mathit{AbsOrRel} & = & \mathit{AORAbs}\ \mathit{Numeric}\\
 & \mid & \mathit{AORRel}\ \mathit{Numeric}\\
 & \mid & \mathit{AORDefault}\\
 & & \textbf{deriving}\ \mathit{Show}
\end{array}
$$

Bei der Umwandlung eines Rahmens in ein Bild, werden je nach relativer oder absoluter Wahl der Ausmaße, die entsprechenden Gleichungen verwendet, die wir hier zu einem Gleichungssystem kombinieren, das den Rahmen vollständig beschreibt.

$$
\begin{array}{lll}
\textbf{instance}\ \mathit{IsPicture\ Frame}\ \textbf{where} & & \\
\quad \mathit{toPicture}\ (\mathit{Frame'}\ \mathit{fa}\ \mathit{ea}\ \mathit{path}\ p) & & \\
 & = & \mathit{Frame}\ \mathit{fa}\ \mathit{eqs}\ \mathit{path}\ p \\
\qquad\qquad \textbf{where}\ \mathit{eqs} & = & \mathit{equations}\ (\textbf{case}\ \mathit{eaX}\ \mathit{ea}\ \textbf{of} \\
 & & \quad \mathit{AORRel}\ \mathit{dx} \to (\mathit{var}\ \texttt{"dx"} \doteq \mathit{dx}) : \mathit{eaEqsDX}\ \mathit{ea} \\
 & & \quad \mathit{AORDefault} \to (\mathit{var}\ \texttt{"dx"} \doteq 2) : \mathit{eaEqsDX}\ \mathit{ea} \\
 & & \quad \mathit{AORAbs}\ w \to (\mathit{var}\ \texttt{"width"} \doteq w) : \mathit{eaEqsWidth}\ \mathit{ea}) \\
 & & {:}\mathit{equations}\ (\textbf{case}\ \mathit{eaY}\ \mathit{ea}\ \textbf{of} \\
 & & \quad \mathit{AORRel}\ \mathit{dy} \to (\mathit{var}\ \texttt{"dy"} \doteq \mathit{dy}) : \mathit{eaEqsDY}\ \mathit{ea} \\
 & & \quad \mathit{AORDefault} \to (\mathit{var}\ \texttt{"dy"} \doteq 2) : \mathit{eaEqsDY}\ \mathit{ea} \\
 & & \quad \mathit{AORAbs}\ h \to (\mathit{var}\ \texttt{"height"} \doteq h) : \mathit{eaEqsHeight}\ \mathit{ea}) \\
 & & {:}\mathit{eaEqs}\ \mathit{ea}
\end{array}
$$

Bei der Definition des rechteckigen Rahmens müssen wir alle Möglichkeiten der Breiten und Höhenwahl in den Gleichungen *eaEqsDX*, *eaEqsDY*, *eaEqsWidth* und *eaEqsHeight* berücksichtigen, während in der Variable *eqEqs* die Gleichungen stehen, die unabhängig von der Wahl der Form immer gelten sollen. Der Name `"last"` bezeichnet das zu umrandende Bild, d.h. das Bild, auf das die Funktion *box* angewendet wird. Dies ist ein spezieller Name, der nur dazu dient, sich auf das Bild beziehen zu können, ohne ihm dafür einen Namen geben zu müssen.

$$
\begin{array}{lll}
\mathit{box} & :: & \mathit{IsPicture}\ a \Rightarrow a \to \mathit{Frame} \\
\mathit{box}\ p & = & \mathit{Frame'}\ \mathit{stdFrameAttrib} \\
 & & \qquad \mathit{stdExtentAttrib}\{\mathit{eaEqs} = \mathit{eqs}, \\
 & & \qquad\qquad \mathit{eaEqsDX} = \mathit{eqsDX}, \\
 & & \qquad\qquad \mathit{eaEqsDY} = \mathit{eqsDY}, \\
 & & \qquad\qquad \mathit{eaEqsWidth} = \mathit{eqsWidth}, \\
 & & \qquad\qquad \mathit{eaEqsHeight} = \mathit{eqsHeight}\} \\
 & & \qquad \mathit{path} \\
 & & \qquad (\mathit{toPicture}\ p) \\
\quad \textbf{where}\ \mathit{eqsDX} & = & [\mathit{ref}\ E \doteq \mathit{ref}\ (\texttt{"last"} \triangleleft E) + \mathit{vec}\ (\mathit{var}\ \texttt{"dx"}, 0), \\
 & & \ \mathit{ref}\ W \doteq \mathit{ref}\ (\texttt{"last"} \triangleleft W) - \mathit{vec}\ (\mathit{var}\ \texttt{"dx"}, 0)] \\
\qquad \mathit{eqsDY} & = & [\mathit{ref}\ N \doteq \mathit{ref}\ (\texttt{"last"} \triangleleft N) + \mathit{vec}\ (0, \mathit{var}\ \texttt{"dy"}), \\
 & & \ \mathit{ref}\ S \doteq \mathit{ref}\ (\texttt{"last"} \triangleleft S) - \mathit{vec}\ (0, \mathit{var}\ \texttt{"dy"})] \\
\qquad \mathit{eqsWidth} & = & [\mathit{ref}\ E \doteq \mathit{ref}\ W + \mathit{vec}\ (\mathit{var}\ \texttt{"width"}, 0), \\
 & & \ \mathit{ref}\ C - \mathit{ref}\ W \doteq \mathit{ref}\ E - \mathit{ref}\ C] \\
\qquad \mathit{eqsHeight} & = & [\mathit{ref}\ N \doteq \mathit{ref}\ S + \mathit{vec}\ (0, \mathit{var}\ \texttt{"height"}), \\
 & & \ \mathit{ref}\ C - \mathit{ref}\ S \doteq \mathit{ref}\ N - \mathit{ref}\ C] \\
\qquad \mathit{eqs} & = & [\mathit{ref}\ C \doteq \mathit{ref}\ (\texttt{"last"} \triangleleft C), \\
 & & \ \mathit{xpart}\ (\mathit{ref}\ \mathit{NE}) \doteq \mathit{xpart}\ (\mathit{ref}\ \mathit{SE}), \\
 & & \ \mathit{ypart}\ (\mathit{ref}\ \mathit{NW}) \doteq \mathit{ypart}\ (\mathit{ref}\ \mathit{NE}), \\
 & & \ \mathit{ref}\ W \doteq \mathit{med}\ 0.5\ (\mathit{ref}\ \mathit{NW})\ (\mathit{ref}\ \mathit{SW}), \\
 & & \ \mathit{ref}\ S \doteq \mathit{med}\ 0.5\ (\mathit{ref}\ \mathit{SW})\ (\mathit{ref}\ \mathit{SE}), \\
 & & \ \mathit{ref}\ E \doteq \mathit{med}\ 0.5\ (\mathit{ref}\ \mathit{NE})\ (\mathit{ref}\ \mathit{SE}), \\
 & & \ \mathit{ref}\ N \doteq \mathit{med}\ 0.5\ (\mathit{ref}\ \mathit{NE})\ (\mathit{ref}\ \mathit{NW})] \\
\qquad \mathit{path} & = & \mathit{ref}\ \mathit{NE}\ \text{--}\ \mathit{ref}\ \mathit{SE}\ \text{--}\ \mathit{ref}\ \mathit{SW}\ \text{--}\ \mathit{ref}\ \mathit{NW}\ \text{--}\ \mathit{cycle}
\end{array}
$$

Zum Schuß betrachten wir noch die Operatoren für Pfadverbindungen. Die Konkatenation ist sehr einfach.

$$
\begin{array}{lll}
(\&) & :: & (\mathit{IsPath}\ a, \mathit{IsPath}\ b) \Rightarrow a \to b \to \mathit{Path} \\
p_1\ \&\ p_2 & = & \mathit{PathJoin}\ (\mathit{toPath}\ p_1) \\
 & & \qquad\qquad (\mathit{stdPathElemDescr}\ \#\ \mathit{setJoin}\ \mathit{BJCat}) \\
 & & \qquad\qquad (\mathit{toPath}\ p_2)
\end{array}
$$

Die anderen Verbindungstypen haben dagegen eine praktische Funktion integriert. Sie sorgen automatisch dafür, daß ein Pfad, der durch einen Punkt der Form *ref* $(\mathit{Name} \triangleleft \mathit{Dir})$ verläuft, an der Bounding Box des Bildes abgeschnitten wird, zu dem der Punkt gehört.

$$
\begin{array}{rll}
(\text{--}) & :: & (\mathit{IsPath}\ a, \mathit{IsPath}\ b) \Rightarrow a \to b \to \mathit{Path} \\
p_1\ \text{--}\ p_2 & = & \mathit{PathJoin}\ \mathit{p1'}\ (\mathit{stdPathElemDescr} \\
 & & \qquad\qquad\quad \#\ \mathit{setJoin}\ \mathit{BJStraight} \\
 & & \qquad\qquad\quad \#\ \mathit{defaultStartCut}\ \mathit{p1'} \\
 & & \qquad\qquad\quad \#\ \mathit{defaultEndCut}\ \mathit{p2'})\ \mathit{p2'} \\
\textbf{where}\ \mathit{p1'} & = & \mathit{toPath}\ p_1 \\
\mathit{p2'} & = & \mathit{toPath}\ p_2
\end{array}
$$

Die Aufgabe besteht darin, aus den beiden zu verbindenden Pfaden den letzten bzw. ersten Punkt zu extrahieren. Wenn der jeweilige Punkt die Form *ref* $(\mathit{Name} \triangleleft \mathit{Dir})$ besitzt, können wir die Attributierungsfunktion *setStartCut Name* bzw. *setEndCut Name* auf das Pfadsegment anwenden.

$$
\begin{array}{lll}
\mathit{defaultStartCut} & :: & \mathit{HasStartEndCut}\ a \Rightarrow \mathit{Path} \to (a \to a) \\
\mathit{defaultStartCut}\ (\mathit{PathPoint}\ p) & = & \mathit{defaultCut}\ \mathit{setStartCut}\ p \\
\mathit{defaultStartCut}\ (\mathit{PathEndDir}\ p\ \_) & & \\
 & = & \mathit{defaultCut}\ \mathit{setStartCut}\ p \\
\mathit{defaultStartCut}\ (\mathit{PathJoin}\ \_\ \_\ p) & & \\
 & = & \mathit{defaultStartCut}\ p \\
\mathit{defaultStartCut}\ \_ & = & \mathit{id}
\end{array}
$$

Nur wenn der Punkt aus einer einzigen Referenz besteht, müssen wir prüfen, ob diese Referenz die Form *Name* $\triangleleft$ *Dir* hat, im anderen Fall wollen und können wir nichts am Pfadsegment ändern.

$$
\begin{array}{lll}
\mathit{defaultCut} & :: & (\mathit{Name} \to a \to a) \to \mathit{Point} \to (a \to a) \\
\mathit{defaultCut}\ f\ (\mathit{PointVar}\ \mathit{name}) & & \\
\qquad \mid \mathit{lastNameIsDir}\ \mathit{name} & = & f\ \$\ \mathit{withoutDir}\ \mathit{name} \\
\qquad \mid \mathit{otherwise} & = & \mathit{id} \\
\mathit{defaultCut}\ \_\ \_ & = & \mathit{id}
\end{array}
$$

Wir können prüfen, ob der letzte Namensbestandteil ein Name vom Typ *Dir* ist.

$$
\begin{array}{lll}
\mathit{lastNameIsDir} & :: & \mathit{Name} \to \mathit{Bool} \\
\mathit{lastNameIsDir}\ (\mathit{Hier}\ \_\ \mathit{name}) & = & \mathit{lastNameIsDir}\ \mathit{name} \\
\mathit{lastNameIsDir}\ (\mathit{NameDir}\ \_) & = & \mathit{True} \\
\mathit{lastNameIsDir}\ \_ & = & \mathit{False}
\end{array}
$$

Wenn dies der Fall ist, können wir aus dem Namen den letzten Bestandteil entfernen und erhalten damit den Namen des Bildes, an dessen Bounding Box das Pfadsegment abgeschnitten werden soll.

$$
\begin{array}{ll}
\textit{withoutDir} & :: \textit{Name} \rightarrow \textit{Name} \\
\textit{withoutDir}\ (\textit{Hier name}\ (\textit{NameDir}\ \_)) & \\
 & = \textit{name} \\
\textit{withoutDir}\ (\textit{Hier}\ n_1\ n_2) & = \textit{Hier}\ n_1\ (\textit{withoutDir}\ n_2) \\
\textit{withoutDir name} & = \textit{name}
\end{array}
$$

Damit wollen wir die Diskussion der Funktionen zur Bildbeschreibung abschließen, obwohl wir nur einen kleinen, aber hoffentlich interessanten, Bruchteil kennengelernt haben.

7.3 Pretty Printer

Ein Pretty Printer bildet eine Zwischenschicht mit Befehlen zur Beschreibung von Dokumenten und einer Funktionalität, diese Beschreibung auf einem Ausgabegerät umzusetzen. Es gibt verschiedene Vorteile, die den Einsatz eines Pretty Printers nahelegen.

- Die Beschreibung des Dokumentes kommt ohne die Angabe (geräteabhängiger) Steuerzeichen wie z.B. Carriage Return aus. Stattdessen existieren steuernde Ausdrücke, die der Pretty Printer bei der Berechnung des Layouts in entsprechende Steuerzeichen des Ausgabegerätes umsetzt.

- Damit verbunden entsteht für die Beschreibung des Dokumentes eine Unabhängigkeit vom Ausgabegerät, welches der Anwender bei der Beschreibung des Layouts nicht kennen muß.

- Wenn gewünscht, kann der Pretty Printer auch eine maximale Zeilenlänge garantieren und den Anwender auf diese Weise von der Notwendigkeit Zeilenumbrüche zu setzen, befreien.

- Es ist möglich von der gleichen Dokumentbeschreibung verschiedene Layouts zu generieren. Soll das Layout des Dokuments später einmal variiert werden, genügt eine lokale Änderung des Pretty Printers.

- Das Kombinieren verschiedener Textelemente kann ohne die langsame Listenkonkatenation erfolgen. Dies kommt dem Laufzeitverhalten zugute, das linear zur Länge des auszugebenden Textes sein sollte.

In den Systembibliotheken des Haskell Interpreters Hugs und des Compilers ☞ *GHC* ist ein Pretty Printer von JOHN HUGHES [Hug95] enthalten. Es stehen verschiedene Layoutstile, wie die Ausgabe in eine einzige Zeile oder das Ignorieren von Einrückungen, zur Verfügung.
Für wichtige Datentypen existieren Funktionen, die diese darstellen. Z.B. *text* für Strings oder *integer* für ganze Zahlen. Mit Kombinatoren lassen sich Dokumentteile untereinander anordnen. Die wichtigsten Kombinatoren sind ($$) zum Erzeugen einer Zeilenumbruchs, (<>) zur horizontalen Anordnung und (<+>) zur horizontalen Anordnung mit Zwischenraum. Weiterhin ordnet *sep* eine Liste von Dokumenten, wenn es die Gesamtbreite zuläßt, horizontal oder sonst vertikal an. [1]
Die Operatoren bilden eine Algebra mit Regeln der Assoziativität und Kommutativität für ($$):

$$
\begin{array}{lll}
\langle a_1 \rangle & (x \mathbin{\$\$} y) \mathbin{\$\$} z & = x \mathbin{\$\$} (y \mathbin{\$\$} z) \\
\langle a_2 \rangle & \textit{empty} \mathbin{\$\$} x & = x \\
\langle a_3 \rangle & x \mathbin{\$\$} \textit{empty} & = x
\end{array}
$$

und für <> und <+>:

[1] Man vergleiche dazu die Bildkombinatoren von *functional* METAPOST, die wir in Abschnitt 3.3 vorstellten. Wir können unsere Bildbeschreibungssprache also auch als einen sehr komfortablen Pretty Printer verstehen.

$$\begin{array}{lll}
\langle b_1\rangle & (x <> y) <> z & = x <> (y <> z) \\
\langle b_2\rangle & \mathit{empty} <> x & = x \\
\langle b_3\rangle & x <> \mathit{empty} & = x
\end{array}$$

für Konkatenation von Text gilt der Homomorphismus:

$$\begin{array}{lll}
\langle t_1\rangle & \mathit{text}\ s <> \mathit{text}\ t & = \mathit{text}\ (s +\!\!+ t) \\
\langle t_2\rangle & \mathit{text}\ \texttt{""} <> x & = x
\end{array}$$

PHILIP WADLER untersucht in [Wad98] die Laufzeit von HUGHEs Pretty Printer und stellt fest, daß dieser zwar optimal die Breite der Ausgabe begrenzt, aber nicht gebunden ist. Gebunden meint, daß die Entscheidung an welcher Stelle eine Zeile umzubrechen ist, mit dem Lesen von höchstens w Zeichen fällt, wo w die Zeilenbreite ist. Die Ungebundenheit von HUGHEs Pretty Printer entsteht durch den Operator *sep*, der eine Wahlmöglichkeit beinhaltet, die in manchen Situationen eine weitere Konstruktion der Ausgabe erfordert, die dann eventuell wieder verworfen wird. Die Laufzeit ist dann nicht mehr linear.

Also schlägt WADLER einen eigenen Pretty Printer vor, der optimal und gebunden ist. Dazu müssen allerdings beim Aufbau des Ausdrucks, der das Dokument repräsentiert, optionale Zeilenumbrüche angegeben sein, die der Pretty Printer bei Bedarf nutzen kann.

Beide Pretty Printer sind für unseren Verwendungszweck unnötig ausdrucksstark. Die Vergleiche zur Wahl der günstigsten Ausgabe kosten zu viel Zeit. In vielen Messungen wurde klar, daß ein ineffizienter Pretty Printer leicht mehr Zeit beanspruchen kann, als die Berechnung der Ausgabe selbst. Wenn sehr lange Terme auszugeben sind, muß eine Zeile oft umgebrochen werden. Für unser Problem ist es ausreichend, wenn keine Zeile zu lang ist und nicht schlimm, wenn einige Zeilen etwas zu früh umgebrochen sind. Es geht nur darum, eine Zeilenlänge von weniger als 1000 Zeichen zu erhalten, damit METAPOST den generierten Quellcode verarbeiten kann. Dabei sollten aber nicht zu viele Zeilenumbrüche eingefügt werden, um die Länge des Quellcodes nicht unnötig zu vergrößern. Wir entwickeln deshalb einen kleinen Pretty Printer, der eine Zeile spätestens umbricht, wenn sie 100 Token enthält.

Der Datentyp für die Dokumente kommt mit wenigen Konstruktoren aus. Das leere Dokument, horizontale Anordnung, Ausgabetext und Zeilenumbruch genügen.

$$\begin{array}{lll}
\textbf{data}\ \mathit{Doc} & = & \mathit{Empty} \\
 & \mid & \mathit{Doc}\ `\mathit{Beside}`\ \mathit{Doc} \\
 & \mid & \mathit{Text}\ \mathit{String} \\
 & \mid & \mathit{Return}\ \mathit{Doc}
\end{array}$$

Bei den zur Verfügung gestellten Ausdrücken wollen wir uns an den Pretty Printer von HUGHES halten und eine Untermenge davon implementieren.

$$\begin{array}{lll}
\mathit{empty} & :: & \mathit{Doc} \\
\mathit{empty} & = & \mathit{Empty} \\
\mathit{int} & :: & \mathit{Show}\ a \Rightarrow a \rightarrow \mathit{Doc} \\
\mathit{int}\ n & = & \mathit{text}\ (\mathit{show}\ n) \\
\mathit{char} & :: & \mathit{Char} \rightarrow \mathit{Doc} \\
\mathit{char}\ c & = & \mathit{Text}\ [c] \\
\mathit{text} & :: & \mathit{String} \rightarrow \mathit{Doc} \\
\mathit{text}\ s & = & \mathit{Text}\ s \\
\mathit{parens} & :: & \mathit{Doc} \rightarrow \mathit{Doc} \\
\mathit{parens}\ p & = & \mathit{char}\ \texttt{'('} <> p <> \mathit{char}\ \texttt{')'}
\end{array}$$

Wir benötigen nur drei Operatoren zur Anordnung der Dokumente.

$$
\begin{array}{lcl}
(<>) & :: & \mathit{Doc} \to \mathit{Doc} \to \mathit{Doc} \\
p <> q & = & p \text{ `}\mathit{Beside}\text{` } q \\[1ex]
(<+>) & :: & \mathit{Doc} \to \mathit{Doc} \to \mathit{Doc} \\
p <+> q & = & (p \text{ `}\mathit{Beside}\text{` } \mathit{Text} \texttt{"␣"}) \text{ `}\mathit{Beside}\text{` } q \\[1ex]
(\$\$) & :: & \mathit{Doc} \to \mathit{Doc} \to \mathit{Doc} \\
p \$\$ q & = & p \text{ `}\mathit{Beside}\text{` } \mathit{Return}\ q
\end{array}
$$

Der Typ *Doc* soll eine Instanz der Klasse *Show* sein, um so die Umwandlung einer Dokumentbeschreibung in eine Zeichenkette zu erreichen.

$$
\begin{array}{l}
\textbf{instance } \mathit{Show\ Doc} \textbf{ where} \\
\qquad \mathit{showsPrec}\ \_\ \mathit{doc\ cont} = \mathit{fst}\ (\mathit{showDoc\ doc}\ 0\ \mathit{cont})
\end{array}
$$

Verschiedene Layouttypen sind nicht nötig. Die maximale Anzahl der Texttoken pro Zeile ist fest auf 100 festgelegt. Dies garantiert zwar noch nicht eine theoretische maximale Zeilenlänge, reicht in der Praxis aber aus, da die Länge der einzelnen Token in unserer Anwendung der Codegenerierung begrenzt ist.

$$
\begin{array}{lcl}
\mathit{showDoc} & :: & \mathit{Doc} \to \mathit{Int} \to \mathit{String} \to (\mathit{String}, \mathit{Int}) \\
\mathit{showDoc}\ (d_1 \text{ `}\mathit{Beside}\text{` } d_2)\ n\ c & = & (d1', n'') \\
\qquad \textbf{where } (d1', n') & = & \mathit{showDoc}\ d_1\ n\ d2' \\
\qquad\qquad\quad (d2', n'') & = & \mathit{showDoc}\ d_2\ n'\ c \\
\mathit{showDoc}\ (\mathit{Text}\ s)\ n\ \mathit{cont} & = & \textbf{if } n > 100 \\
 & & \quad \textbf{then } (\mathit{showString}\ (\texttt{'\textbackslash n'} : s)\ \mathit{cont}, 0) \\
 & & \quad \textbf{else } (\mathit{showString}\ s\ \mathit{cont}, n+1) \\
\mathit{showDoc}\ (\mathit{Return}\ d)\ n\ c & = & (\mathit{showString}\ \texttt{"\textbackslash n"}\ d', n') \\
\qquad \textbf{where } (d', n') & = & \mathit{showDoc}\ d\ 0\ c \\
\mathit{showDoc\ Empty}\ n\ \mathit{cont} & = & (\mathit{cont}, n)
\end{array}
$$

Die Definition eines eigenen Pretty Printers war wirklich lohnend, denn die Zeit zur Generierung eines Bildes verkürzte sich, gegenüber dem anfangs verwendeten Pretty Printer von JOHN HUGHES, um zwanzig bis fünfzig Prozent! Der Gewinn ist dabei für sehr komplexe Bilder besonders groß. Für unseren Compiler bedeutet eine derart große Laufzeiteinsparung, die wir durch eine so lokale Änderung in der Ausgabe erzielen können, daß die Codegenerierung sehr effizient sein muß.

7.4 Eine abstrakte Zwischensprache

Bisher haben wir den Sprachstandard von *functional* METAPOST definiert und einen Pretty Printer beschrieben, der eine effiziente Ausgabe garantiert. Wir könnten im nächsten Schritt einen Übersetzer entwerfen, der *Picture*–Ausdrücke in Ausdrücke des Pretty Printers übersetzt. Stattdessen fügen wir jedoch eine Zwischenschicht in Form einer abstrakten METAPOST–Sprache ein. Dieses Vorgehen bietet einige Vorteile.

- Wenn die Übersetzung der Zwischenschicht in den Zielcode getestet ist und korrekt läuft, vermindert sich die Wahrscheinlichkeit von Syntaxfehlern, wie sie fehlende Leerzeichen und Klammern verursachen.

- Die Codegenerierung gestaltet sich wesentlich übersichtlicher.
- Auf der Zwischenschicht lassen sich Optimierungen vornehmen, wie die Zusammenfassung einiger Codeteile oder deren Umgruppierung.
- Die Befehle der Zwischenschicht können durchaus eine komplexere Folge von Befehlen der tieferen Schicht erzeugen und lassen sich später jederzeit verändern, ohne die darüberliegende Schicht anzutasten.

Mit Abbildung 7.0.1 von Seite 93 gaben wir einen Überblick der Sprachschichten, die wir für den Übersetzungsvorgang verwenden. Die verschiedenen Typen von *functional* METAPOST haben alle ihre Entsprechung in der abstrakten Zwischensprache und schließlich in METAPOST, siehe dazu Abbildung 7.4.1.
Die Zwischensprache ist etwas schwächer getypt als *functional* METAPOST, denn der Datentyp *Term* dient der Darstellung von `pair`, `numeric` und `boolean`. Dies spart Konstruktoren und damit Programmcode, denn Operationen wie z.B. Addition können auf Punkte oder Zahlen angewendet werden.

functional METAPOST	$\leadsto$	abstraktes METAPOST	$\leadsto$	METAPOST
Picture	$\leadsto$	MetaPost	$\leadsto$	Befehle
Point	$\leadsto$	Term	$\leadsto$	`pair`
Numeric	$\leadsto$	Term	$\leadsto$	`numeric`
Boolean	$\leadsto$	Term	$\leadsto$	`boolean`
Path	$\leadsto$	MPPath	$\leadsto$	`path`
Arrow	$\leadsto$	MPArrow	$\leadsto$	`path`[a]
Color	$\leadsto$	MPColor	$\leadsto$	`color`
Pen	$\leadsto$	MPPen	$\leadsto$	`pen`
Pattern	$\leadsto$	MPPattern	$\leadsto$	`picture`[b]
Transformation	$\leadsto$	MPTransform	$\leadsto$	`transform`

[a] Ein Makro erzeugt eine gefüllte Fläche.
[b] Ein Füllmuster wird als spezielles Bild repräsentiert, das auf die y-Achse projeziert ist.

Abbildung 7.4.1: Datentypen in den verschiedenen Sprachen

7.4.1 FMPTerm

Dieses Modul definiert den Datentyp *Term* und Funktionen, die auf Ausdrücken dieses Typs Optimierungen vornehmen, siehe Abbildung 7.4.2. Um mit den gewohnten Rechenoperationen Terme notieren zu können, soll der Typ *Term* eine Instanz der Typklasse *Num* sein.

instance *Num Term* **where**

$$
\begin{aligned}
a + b &= add\ a\ b \\
a - b &= sub\ a\ b \\
a * b &= mul\ a\ b \\
negate\ a &= neg\ a \\
abs\ (Neg\ a) &= abs\ a \\
abs\ a &= a
\end{aligned}
$$

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Term} & = & \mathit{Const\ Double} \\
 & | & \mathit{Id\ String} \\
 & | & \mathit{Add\ Term\ Term} \\
 & | & \mathit{Sub\ Term\ Term} \\
 & | & \mathit{Mul\ Term\ Term} \\
 & | & \mathit{Div\ Term\ Term} \\
 & | & \mathit{Neg\ Term} \\
 & | & \mathit{Parens\ Term} \\
 & | & \mathit{Pythagoras\ Term\ Term} \\
 & | & \mathit{Power\ Term\ Term} \\
 & | & \mathit{Sin\ Term} \\
 & | & \mathit{Cos\ Term} \\
 & | & \mathit{Sqrt\ Term} \\
 & | & \mathit{Ln\ Term} \\
 & | & \mathit{Exp\ Term} \\
 & | & \mathit{Round\ Term} \\
 & | & \mathit{Ceil\ Term} \\
 & | & \mathit{Floor\ Term} \\
 & | & \mathit{Angle\ Term} \\
 & | & \mathit{XPart\ Term} \\
 & | & \mathit{YPart\ Term} \\
 & | & \mathit{Pair\ Term\ Term} \\
 & | & \mathit{Max}\ [\mathit{Term}] \\
 & | & \mathit{Min}\ [\mathit{Term}] \\
 & | & \mathit{Identity} \\
 & | & \mathit{CurrentPicture} \\
 & | & \mathit{Pic\ String} \\
 & | & \mathit{Infix\ Term\ String\ Term} \\
 & | & \mathit{LLCorner\ Term} \\
 & | & \mathit{URCorner\ Term} \\
 & | & \mathit{Shifted\ Term\ Term} \\
 & | & \mathit{Transform}\ [\mathit{Int}]\ \mathit{Term} \\
 & | & \mathit{Transformed\ Term\ Term} \\
 & | & \mathit{TransformedM\ Term\ Term\ Term\ Term\ Term\ Term} \\
 & | & \mathit{Mediate\ Term\ Term\ Term} \\
 & | & \mathit{Pos\ Int\ Int} \\
 & | & \mathit{TDot\ String\ Dir} \\
 & | & \mathit{IfElse\ Term\ Term\ Term} \\
 & | & \mathit{VerbFunction\ String\ Term} \\
 & | & \mathit{Dirop\ Term} \\
 & & \textbf{deriving}\ (\mathit{Eq}, \mathit{Show}, \mathit{Read})
\end{array}
$$

Abbildung 7.4.2: Der Datentyp *Term*.

$$
\begin{aligned}
&\mathit{signum}\ 0 &&= 0\\
&\mathit{signum}\ \_ &&= 1\\
&\mathit{fromInteger}\ a &&= \mathit{Const}\ (\mathit{fromInteger}\ a)\\
&\mathit{fromInt}\ a &&= \mathit{Const}\ (\mathit{fromInt}\ a)
\end{aligned}
$$

Die Funktionen *neg*, *add*, ... sind sogenannte smarte Konstruktoren. Statt der Definition $\mathit{negate} = \mathit{Neg}$ ist es sinnvoll, schon bei der Konstruktion eine doppelte Negation abzufangen. Unter Berücksichtigung der Rechenregeln kann man auf diese Weise manches Anwachsen des Ausdrucks vermeiden.

$$
\begin{aligned}
&\mathit{neg} &&:: \mathit{Term} \to \mathit{Term}\\
&\mathit{neg}\ (\mathit{Neg}\ a) &&= a\\
&\mathit{neg}\ (\mathit{Const}\ a) &&= \mathit{Const}\ (-a)\\
&\mathit{neg}\ a &&= \mathit{Neg}\ a
\end{aligned}
$$

Eine Addition von Konstanten muß den Ausdruck nicht weiter aufblähen. Die Summe können wir in diesem Fall sofort berechnen. Wenn wir die Addition einer negativen Zahl auf die Subtraktion zurückführen, spart das einen Konstruktor ein.

$$
\begin{aligned}
&\mathit{add} &&:: \mathit{Term} \to \mathit{Term} \to \mathit{Term}\\
&\mathit{add}\ 0\ b &&= b\\
&\mathit{add}\ a\ 0 &&= a\\
&\mathit{add}\ (\mathit{Const}\ a)\ (\mathit{Const}\ b) &&= \mathit{Const}\ (a+b)\\
&\mathit{add}\ a\ (\mathit{Neg}\ b) &&= \mathit{sub}\ a\ b\\
&\mathit{add}\ a\ b &&= \mathit{Add}\ a\ b
\end{aligned}
$$

Entsprechend sind *sub* und *mul* definiert. Wir können das Prinzip, die Ausdrücke klein zu halten noch weiter treiben, indem wir auch die Funktion *max'*[2] so definieren, daß falls mehrere Konstanten in der Liste vorkommen, nur ihre größte berücksichtigt wird.

$$
\begin{aligned}
&\mathit{max}' &&:: [\mathit{Term}] \to \mathit{Term}\\
&\mathit{max}'\ [\,] &&= 0\\
&\mathit{max}'\ [a] &&= a\\
&\mathit{max}'\ \mathit{as} &&\\
&\qquad |\, \mathit{null}\ c \wedge \mathit{null}\ v &&= 0\\
&\qquad |\, \mathit{null}\ c &&= \mathit{Max}\ v\\
&\qquad |\, \mathit{null}\ v &&= \mathit{Const}\ (\mathit{maximum}\ c)\\
&\qquad |\, \mathit{otherwise} &&= \mathit{Max}\ (\mathit{Const}\ (\mathit{maximum}\ c) : v)\\
&\qquad \mathbf{where} &&\\
&\qquad (c, v) &&= \mathit{max2}\ \mathit{as}\\
&\qquad \mathit{max2}\ [\,] &&= ([\,], [\,])\\
&\qquad \mathit{max2}\ ((\mathit{Const}\ a) : \mathit{as}) &&= (a : \mathit{fst}\ (\mathit{max2}\ \mathit{as}), \mathit{snd}\ (\mathit{max2}\ \mathit{as}))\\
&\qquad \mathit{max2}\ (a : \mathit{as}) &&= (\mathit{fst}\ (\mathit{max2}\ \mathit{as}), a : \mathit{snd}\ (\mathit{max2}\ \mathit{as}))
\end{aligned}
$$

Mit wesentlich mehr Aufwand ließen sich wahrscheinlich noch mehr konstante Terme durch algebraische Umformungen finden. Dafür wäre aber so viel Rechenzeit notwendig, daß es sich nicht bezahlt macht.

[2] *max* hat in Haskell schon eine andere Bedeutung.

7.4.2 FMPMpsyntax

Mit dem Datentyp *Term* können wir nun Ausdrücke bilden, die in METAPOST den Typen `pair`, `numeric` und `boolean` entsprechen. Die Datentypen, die wir benötigen, um METAPOST–Programme darstellen zu können, definieren wir in diesem Abschnitt.
Wir benötigen neben diesen Typen Funktionen, die Konvertierungen zwischen den verschiedenen Sprachschichten vornehmen. Abbildung 7.4.3 zeigt den Ausschnitt aus Abbildung 7.0.1, den wir hier betrachten. Erstens formulieren wir Funktionen, die Ausdrücke der Typen *Path*, *Pen*, *Point*, ..., also der *functional* METAPOST–Schicht, in die entsprechenden Ausdrücke der abstrakten Zwischensprache umwandeln. Zweitens Funktionen, die diese Ausdrücke dann weiter in den Typ *Doc* der Pretty Printer–Schicht übersetzen. Die Namen der Funktionen, die die erste Aufgabe erfüllen, haben das Prefix `mp` und als Suffix den Namen des umzuwandelnden Typs. Für die Funktionen, die eine Umwandlung in den Typ *Doc* vornehmen, gibt es die Typklasse:[3]

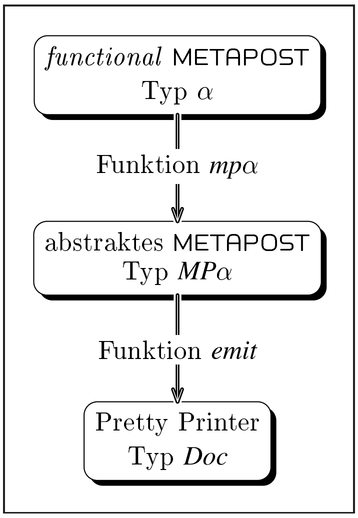


Abbildung 7.4.3: Konvertierungen zwischen den Sprachschichten.

```
class HasEmit a where
    emit        :: a -> Doc
```

Abbildung 7.4.4 zeigt den vollständigen Datentyp zur Repräsentierung von METAPOST–Programmen. In METAPOST gibt es zwar mehr Befehle als Konstruktoren in diesem Datentyp, allerdings entstehen viele dieser Befehle als Makros und sind auf einfachere reduzierbar. Uns genügen diese Konstruktoren jedenfalls für unsere Sprache. Mit Hilfe des Konkatenationsoperators lassen sich auf einfache Weise Sequenzen von Befehlen notieren.

```
instance HasConcat MetaPost where
    MPRelax & a     = a
    a & MPRelax     = a
    a & b           = MPConc a b
```

Alle Datentypen der abstrakten Syntax, die in Abbildung 7.4.1 aufgezählt sind, finden sich in diesem Modul; z.B. für Stifte.

```
data MPPen          = MPDefaultPen
                    | MPPenCircle (Term, Term) Term
                    | MPPenSquare (Term, Term) Term
                      deriving Eq
```

Für alle diese Typen finden sich Konvertierungsfunktionen, die Ausdrücke aus der Sprachebene von *functional* METAPOST in die abstrakte Sprache übersetzen. Deren Namen bilden sich aus dem Prefix „*mp*“ und dem zu konvertierenden Typnamen. Die Konvertierung in die abstrakte Zwischensprache ist einfach, da sich die Datenstrukturen der *functional* METAPOST Ebene nicht sehr von denen der Zwischensprache unterscheiden.

[3]Für Funktionen, des ersten Schrittes können wir keine Typklasse formulieren, da der Parameter– und der Ergebnistyp beide variablel sind. Uns fehlt dafür das Konzept der mehrparametrigen Typklassen. (multi parameter type classes [JJM97]).

```
data MetaPost  = MPAssign Term Term
               | MPAssignPath String MPPath
               | MPBoxit String MetaPost
               | MPBitLine (Term, Term) Term String
               | MPCloneit String String
               | MPClearIt
               | MPClip MPPath
               | MPComment String
               | MPConc MetaPost MetaPost
               | MPDef String Term
               | MPDefineTrans String MPTransform
               | MPDraw MPArrow MPPath MPPattern
                        MPColor MPPen
               | MPDrawAHead MPArrow MPPath
                             MPColor MPPen
               | MPDrawPic MPColor Term
               | MPDrawUnBoxed [String]
               | MPEquals [Term]
               | MPFigure Int MetaPost
               | MPFill MPPath MPColor MPPen
               | MPFixPos [String]
               | MPFixSize [String]
               | MPGraduate MPColor MPColor MPPath
                            Int Double
               | MPGraduatePic MPColor MPColor Term
                               Int Double
               | MPGraduatePath MPArrow MPColor MPColor
                                        MPPath MPPattern
                                        MPPen Int Double
               | MPGroup MetaPost
               | MPIfElse Term MetaPost MetaPost
               | MPImage String MetaPost
               | MPRelax
               | MPShapeit String
               | MPSubBox Int MetaPost
               | MPTex String
               | MPText String
               | MPVerbatim String
               deriving Eq
```

Abbildung 7.4.4: Mit Ausdrücken des Typs *MetaPost* lassen sich METAPOST–Programme beschreiben.

$$
\begin{array}{lcl}
\mathit{mpPen} & :: & \mathit{Pen} \rightarrow \mathit{MPPen} \\
\mathit{mpPen}\ \mathit{DefaultPen} & = & \mathit{MPDefaultPen} \\
\mathit{mpPen}\ (\mathit{PenCircle}\ (a,b)\ c) & = & \mathit{MPPenCircle}\ (\mathit{mpNumeric}\ a, \mathit{mpNumeric}\ b)\ (\mathit{mpNumeric}\ c) \\
\mathit{mpPen}\ (\mathit{PenSquare}\ (a,b)\ c) & = & \mathit{MPPenSquare}\ (\mathit{mpNumeric}\ a, \mathit{mpNumeric}\ b)\ (\mathit{mpNumeric}\ c)
\end{array}
$$

Interessanter sind diese Funktionen für die Typen *Numeric* und *Point*, die wir in den Abbildungen 7.2.1 und 7.2.2 vorgestellt haben. Für die Werte, die in den Konstruktoren vorkommen, müssen wir die Konvertierung rekursiv fortsetzen.

$$
\begin{array}{lcl}
\mathit{mpPoint} & :: & \mathit{Point} \rightarrow \mathit{Term} \\
\mathit{mpPoint}\ (\mathit{PointVarArray}'\ n\ m) & = & \mathit{Id}\ (\texttt{"pvi"} +\!\!+\ \mathit{show}\ n +\!\!+ \texttt{"␣"} +\!\!+\ \mathit{show}\ m) \\
\mathit{mpPoint}\ (\mathit{PointVar}'\ n\ m) & = & \mathit{Id}\ (\texttt{"pv"} +\!\!+\ \mathit{show}\ n +\!\!+ \texttt{"␣"} +\!\!+\ \mathit{show}\ m) \\
\mathit{mpPoint}\ (\mathit{PointPic}'\ n\ d) & = & \mathit{tdot}\ (\mathit{suff}\ n)\ d \\
\mathit{mpPoint}\ (\mathit{PointTrans}'\ p\ [\,]) & = & \mathit{mpPoint}\ p \\
\mathit{mpPoint}\ (\mathit{PointTrans}'\ p\ ts) & = & \mathit{Transform}\ ts\ (\mathit{mpPoint}\ p) \\
\mathit{mpPoint}\ (\mathit{PointPPP}\ \mathit{PPPAdd}\ p_1\ p_2) & & \\
 & = & \mathit{mpPoint}\ p_1 + \mathit{mpPoint}\ p_2 \\
\mathit{mpPoint}\ (\mathit{PointPPP}\ \mathit{PPPSub}\ p_1\ p_2) & & \\
 & = & \mathit{mpPoint}\ p_1 - \mathit{mpPoint}\ p_2 \\
\mathit{mpPoint}\ (\mathit{PointPPP}\ \mathit{PPPDiv}\ p_1\ p_2) & & \\
 & = & \mathit{mpPoint}\ p_1\ /\ \mathit{mpPoint}\ p_2 \\
\mathit{mpPoint}\ (\mathit{PointDirection}\ a) & = & \mathit{Dirop}\ (\mathit{mpNumeric}\ a) \\
\mathit{mpPoint}\ (\mathit{PointVec}\ (ox, oy)) & = & \mathit{Pair}\ (\mathit{mpNumeric}\ ox)\ (\mathit{mpNumeric}\ oy) \\
\mathit{mpPoint}\ (\mathit{PointMediate}\ o\ p_1\ p_2) & & \\
 & = & \mathit{Mediate}\ (\mathit{mpNumeric}\ o)\ (\mathit{mpPoint}\ p_1)\ (\mathit{mpPoint}\ p_2) \\
\mathit{mpPoint}\ (\mathit{PointNMul}\ n\ p) & = & \mathit{Mul}\ (\mathit{mpNumeric}\ n)\ (\mathit{mpPoint}\ p) \\
\mathit{mpPoint}\ (\mathit{PointNeg}\ p) & = & -\mathit{mpPoint}\ p \\
\mathit{mpPoint}\ \mathit{PointWhatever} & = & \mathit{Id}\ \texttt{"whatever"} \\
\mathit{mpPoint}\ (\mathit{PointCond}\ b\ t\ e) & = & \mathit{IfElse}\ (\mathit{mpBoolean}\ b)\ (\mathit{mpPoint}\ t)\ (\mathit{mpPoint}\ e)
\end{array}
$$

Für die Klasse von Funktionen, die Ausdrücke mittels der Funktion *emit*, aus der abstrakten Sprache in den Dokumententyp des Pretty Printers übersetzt, geben wie die Instanzen der Typen *Term* und *MetaPost* auszugsweise, in den Abbildungen 7.4.5 und 7.4.6, wieder. Eine vollständigere Auflistung würde nichts weiteres zum Verständnis beitragen.

In der Instanz von *Term* sehen wir, im Fall der Regeln für die Konstruktoren *Const* und *Max*, wie die Behandlung von Sonderfällen helfen kann, eine kürzere Ausgabe zu erzielen.

Auch die Instanz des Typs *MetaPost* versucht, wo möglich Ausdrücke zusammenzufassen. Beispielsweise werden die aufeinanderfolgenden Anweisungen `drawunboxed(b1);` und `drawunboxed(b2);` zu einer Anweisung `drawunboxed(b1, b2);` zusammengefaßt, da dieses Muster häufig in den generierten Programmen auftritt. An anderen Stellen, die hier, da es sich nur um einen Auszug handelt, nicht aufgeführt sind, finden ähnliche Optimierungen statt.

Erwähnenswert ist noch der Konstruktor *MPSubBox*, der eine Rahmenbox mit dem Namen `b`*n* erzeugt, die das Bild enthält, das durch Ausführen der Anweisungen *mp* entsteht. Die Rahmenbox umschließt das Bild minimal, was u.a. für den Befehl *setTrueBoundingBox* benötigt wird. Der Punkt *shiftRefPoint n* speichert die untere rechte Ecke des Bildes, da dieses später an eine andere Stelle gezeichnet werden kann und sich auf diese Weise der Translationsvektor bestimmen läßt. Diesen benötigen wir, um Referenzen auf Punkte innerhalb des verschobenen Bildes ebenfalls um den Translationsvektor zu verschieben. Am Ende des Vorgangs wird die unrsprüngliche Zeichenfläche wiederhergestellt und das mit *mp* gezeichnete Bild befindet sich in der Bildvariablen der Rahmenbox b*n*. Es ist wichtig, daß wir die Zeichenfläche in einem Array p*n* zwischenspeichern und nicht in einer Variable p, damit der Konstruktor auch geschachtelt auftreten kann.

```
instance HasEmit Term where
    emit (Const 0)          = text "0"
    emit (Const n)          = if n < 0
                                then text ('(' : (showFFloat (Just 4) n ")"))
                                else text (showFFloat (Just 4) n "")
    emit (Pos n m)          = text (pos n m)
    emit (Max [])           = char '0'
    emit (Max [a])          = emit a
    emit (Max (t : ts))     = text "max(" <> emit t
                            <> hcat [comma <> emit t' | t' ← ts] <> char ')'
    emit (Neg a)            = text "(-(" <> emit a <> text "))"
    emit (Add a b)          = emit a <> char '+' <> emit b
    emit (Sub a b)          = emit a <> text "-(" <> emit b <> char ')'
    emit (Mul a b)          = char '(' <> emit a <> text ")*(" <> emit b <> char ')'
    emit (Pair a b)         = char '(' <> emit a <> char ','
                            <> emit b <> char ')'
    emit (XPart a)          = text "(xpart␣(" <> emit a <> text "))"
    emit (Id a)             = text a
    emit (Pythagoras a b)   = char '(' <> emit a <> text "++"
                            <> emit b <> char ')'
    emit CurrentPicture     = text "currentpicture"
    emit (Infix a b c)      = emit a <+> text b <+> emit c
    emit (LLCorner a)       = text "llcorner␣(" <> emit a <> char ')'
    emit (Pic a)            = text "pic" <+> text a
    emit (Transformed a b)
                            = char '(' <> emit a <> text ")␣transformed␣("
                            <> emit b <> char ')'
    emit (Sin a)            = text "sind(" <> emit a <> char ')'
    emit (Power a b)        = text "((" <> emit a <> text ")**("
                            <> emit b <> text "))"
    emit (Angle a)          = text "angle(" <> emit a <> char ')'
    emit (Dirop a)          = text "(dir␣(" <> emit a <> text "))"
    emit (Mediate a b c)    = emit a <> brackets (emit b <> comma <> emit c)
    emit (TDot a d)         = text (a ++ emitDir d)
    emit (Parens a)         = parens (emit a)
    emit (VerbFunction a b)
                            = text a <> parens (emit b)
    emit (IfElse b t e)     = text "if" <+> emit b <> char ':' <> emit t
                            <+> text "else:" <> emit e <+> text "fi␣"
```

Abbildung 7.4.5: Instanz des Typs *Term* der Klasse *HasEmit*.

```
instance HasEmit MetaPost where
    emit (MPAssign l r)     = emit l <+> text ":=" <+> emit r <> semi
    emit (MPBoxit s pic)    = text "boxit." <> text s <> parens (emit pic) <> semi
    emit (MPClip path)      = text "clip␣currentpicture␣to␣" <+> emit path <> semi
    emit (MPDrawUnBoxed [])
                            = empty
    emit (MPConc (MPDrawUnBoxed s1) (MPConc (MPDrawUnBoxed s2) mp))
                            = emit (MPDrawUnBoxed (s1 ++ s2) & mp)
    emit (MPDrawUnBoxed s)
                            = text "drawunboxed" <> parens (emitL s) <> semi
    emit (MPConc a b)       = emit a $$ emit b
    emit (MPDraw ar p d c pen)
                            = text "draw" <+> parens (emit p)
                           <> emit d <> emit c <> emit pen <> semi
                           <+> emit (MPDrawAHead ar p c pen)
    emit (MPDrawPic c p)    = text "draw" <+> emit p <+> emit c <> semi
    emit (MPFill path c p)  = text "fill" <+> emit path <+> emit c <+> emit p <> semi
    emit (MPGraduate c1 c2 path q a)
                            = text "graduate"
                           <> parens (emitColor' c1 <> comma
                                   <+>emitColor' c2 <> comma
                                   <+>emit path <> comma
                                   <+>int q <> comma
                                   <+>double a) <> semi
    emit (MPIfElse b t e)   = text "if" <+> emit b <> colon
                           $$   emit t $$ text "else" <> colon
                           $$   emit e $$ text "fi" <> semi
    emit (MPRelax)          = empty
    emit (MPSubBox n mp)
                            = text "p" <> int n <+> text "␣:=␣currentpicture" <> semi
                           <+> text "clearit" <> semi
                           $$  emit mp
                           $$  emit (shiftRefPoint n)
                                   <> text "␣:=␣llcorner␣currentpicture" <> semi
                           $$  text ("boxit." ++ suff n
                                   ++ "(currentpicture)") <> semi
                           $$  text (suff n ++ ".dx␣=␣0") <> semi
                           <+> text (suff n ++ ".dy␣=␣0") <> semi
                           $$  text "currentpicture␣:=␣p" <> int n <> semi
    emit (MPTex s)          = text "btex" <+> text s <+> text "etex"
    emit (MPVerbatim a)     = text a
    emit (MPImage s mp)     = text (s ++ ":=image(") $$ emit mp $$ char ')' <> semi
```

Abbildung 7.4.6: Instanz des Typs *MetaPost* der Klasse *HasEmit*.

7.5 Die Verwaltung der Variablen

Die Aufgabe der Variablenverwaltung ist es, alle in Picture–Ausdrücken vorkommenden Variablen auf entsprechende METAPOST–Variablen im generierten Quelltext abzubilden. Grundsätzlich wenden wir bei der Variablenverwaltung traditionelle Techniken, wie sie z.B. in [ASU88] Kapitel 7.6 beschrieben sind, an. Die Verwaltung ist in zwei Phasen aufgeteilt.

1. Die erste Phase baut eine Symboltabelle auf, die eine Abbildung von *functional* METAPOST–Variablen auf METAPOST–Variablen realisiert. Eine Funktion durchsucht einen Ausdruck nach Variablen, die in der Symboltabelle noch nicht eingetragen sind. Bei einem Vorkommen einer unbekannten Variable, wird ihr ein freier Speicherplatz zugeordnet und ihr Name zusammen mit der Information über den Speicherplatz in eine Symboltabelle eingetragen. D.h. es wurde ein definierendes Variablenvorkommen gefunden.

2. In einer zweiten Phase werden in allen Ausdrücken, die Vorkommen von Variablen in der Symboltabelle gesucht und durch Referenzen auf den jeweiligen Speicherplatz ersetzt. Diese Referenzen repräsentieren die Konstruktoren *NumericVar*′, *NumericArray*′, *PointPic*′ und die weiteren Konstruktoren der Typen *Numeric* und *Point* (siehe Abbildungen 7.2.1 und 7.2.2), die mit einem Hochkomma enden.

Bei der Abbildung einer Variable auf Speicherplatz in METAPOST wollen wir drei Dinge beachten:

- Die Abbildung muß injektiv, d.h. eindeutig sein.
- Die Abbildung soll möglichst einfach sein, wir wollen den freien Speicher mit weinig Aufwand verwalten.
- Variablen mit Namen vom Typ *Int* und *Dir* sollen, wenn sich der Name nicht aus weiteren Bestandteilen zusammensetzt, in konstanter Zeit abgebildet werden.

Wir erfüllen diese Anforderungen, indem wir Variablen auf zweidimensionalen Arrays in METAPOST abbilden. Es gibt insgesamt vier verschiedene solcher Arrays. Jeweils zwei für numerische Werte und zwei für Punkte.

```
numeric nv[][];
numeric nvi[][];
pair pv[][];
pair pvi[][];
```

Die einzelnen Teilbilder, die entweder atomar sind oder durch Kombination entstanden, werden während der Übersetzung von *functional* METAPOST eindeutig numeriert. Die Nummer des Bildes, in dem die Variable definiert ist, korrespondiert zur ersten Dimension der Arrays. Die Arrays `pvi[][]` und `nvi[][]` bilden Variablen mit Namen vom Typ *Int* in konstanter Zeit ab, wobei die zweite Dimension dem Zahlenwert des Namens (wir können ihn als Index ansehen) entspricht. Die zweite Dimension der Arrays `pv[][]` und `nv[][]`, die alle Variablen mit den restlichen Namen speichern, ergibt sich durch eine Numerierung der definierenden Variablenvorkommen innerhalb eines Gleichungssytems eines Bildes. Variablen mit Namen vom Typ *Dir*, die einen Bezugspunkt bezeichnen, können wir direkt auf den entsprechenden Namen des Bezugspunktes in METAPOST abbilden, da die Rahmenboxen diese Punkte zur Verfügung stellen.
Um diese Abbildung etwas anschaulicher zu machen, wollen wir etwas vorgreifen und anhand eines Beispiels zeigen, wie die Variablen aus der Bildbeschreibung der Abbildung 3.7.1 von Seite 34 auf

die zweidimensionalen Arrays abgebildet werden. Zur Erinnerung zeigt Abbildung 7.5.1 neben dem Ausschnitt aus dem METAPOST–Programm nochmals die Bildbeschreibung.

$$
\begin{array}{l}
\mathit{define}\ [\mathit{ref}\ \texttt{"p1"} \doteq \mathit{vec}\ (0,5),\\
\qquad \mathit{ref}\ \texttt{"p2"} \doteq \mathit{vec}\ (60,0),\\
\qquad \mathit{ref}\ \texttt{"p3"} \doteq \mathit{vec}\ (15,60),\\
\qquad \mathit{ref}\ \texttt{"m12"} \doteq \mathit{med}\ 0.5\ (\mathit{ref}\ \texttt{"p1"})\ (\mathit{ref}\ \texttt{"p2"}),\\
\qquad \mathit{ref}\ \texttt{"m23"} \doteq \mathit{med}\ 0.5\ (\mathit{ref}\ \texttt{"p2"})\ (\mathit{ref}\ \texttt{"p3"}),\\
\qquad \mathit{var}\ \texttt{"a12"} \doteq 90 + \mathit{angle}\ (\mathit{ref}\ \texttt{"p1"} - \mathit{ref}\ \texttt{"p2"}),\\
\qquad \mathit{var}\ \texttt{"a23"} \doteq 90 + \mathit{angle}\ (\mathit{ref}\ \texttt{"p2"} - \mathit{ref}\ \texttt{"p3"}),\\
\qquad \mathit{equal}\ [\mathit{ref}\ \texttt{"mid"}, \mathit{med}\ \mathit{whatever}\ (\mathit{ref}\ \texttt{"m12"})\ (\mathit{ref}\ \texttt{"m12"} + \mathit{dir}\ (\mathit{var}\ \texttt{"a12"})),\\
\qquad\qquad \mathit{med}\ \mathit{whatever}\ (\mathit{ref}\ \texttt{"m23"})\ (\mathit{ref}\ \texttt{"m23"} + \mathit{dir}\ (\mathit{var}\ \texttt{"a23"}))],\\
\qquad \mathit{var}\ \texttt{"r"} \doteq \mathit{dist}\ (\mathit{ref}\ \texttt{"mid"})\ (\mathit{ref}\ \texttt{"p1"})]\\
\quad (\mathit{draw}\ [\mathit{ref}\ \texttt{"mid"} + \mathit{vec}\ (0, \mathit{var}\ \texttt{"r"})\ ..\ \mathit{ref}\ \texttt{"mid"} + \mathit{vec}\ (\mathit{var}\ \texttt{"r"}, 0)\\
\qquad\qquad ..\ \mathit{ref}\ \texttt{"mid"} + \mathit{vec}\ (0, -\mathit{var}\ \texttt{"r"})\ ..\ \mathit{ref}\ \texttt{"mid"} + \mathit{vec}\ (-\mathit{var}\ \texttt{"r"}, 0)\\
\qquad\qquad ..\ \mathit{cycle},\\
\qquad\ \mathit{ref}\ \texttt{"p1"}\ \text{--}\ \mathit{ref}\ \texttt{"p2"}\ \text{--}\ \mathit{ref}\ \texttt{"p3"}\ \text{--}\ \mathit{cycle},\\
\qquad\ \mathit{ref}\ \texttt{"m12"}\ \text{--}\ \mathit{ref}\ \texttt{"mid"},\\
\qquad\ \mathit{ref}\ \texttt{"m23"}\ \text{--}\ \mathit{ref}\ \texttt{"mid"}]\ \mathit{empty})
\end{array}
$$

```
pv1 0 = (0,5);
pv1 1 = (60,0);
pv1 2 = (15,60);
pv1 3 = 0.5[pv1 0,pv1 1];
pv1 4 = 0.5[pv1 1,pv1 2];
nv1 5 = 90+angle(pv1 0-(pv1 1));
nv1 6 = 90+angle(pv1 1-(pv1 2));
pv1 7 = whatever[pv1 3,pv1 3+(cosd(nv1 5),sind(nv1 5))]
      = whatever[pv1 4,pv1 4+(cosd(nv1 6),sind(nv1 6))];
nv1 8 = ((xpart (pv1 7-(pv1 0)))++(ypart (pv1 7-(pv1 0))));
boxit.n3();
n3.e = n3.w+(0,0);
n3.n = n3.s+(0,0);
cloneit.n2(n3);
cloneit.n1(n2);
fixpos(n3);
tempPath := pv1 7+(0,nv1 8)..pv1 7+(nv1 8,0)
        ..pv1 7+(0,(-(nv1 8)))..pv1 7+((-(nv1 8)),0)..cycle;
draw (subpath (0,4) of tempPath);
tempPath := pv1 0--pv1 1--pv1 2--cycle;
draw (subpath (0,3) of tempPath);
tempPath := pv1 3--pv1 7;
draw (subpath (0,1) of tempPath);
tempPath := pv1 4--pv1 7;
draw (subpath (0,1) of tempPath);
```

Abbildung 7.5.1: Die Bildbeschreibung der Abbildung 3.7.1 und der generierte METAPOST–Quelltext. Dieses Beispiel zeigt gut die Abbildung von Variablen eines Gleichungssystems auf Arrays in METAPOST.

7.5.1 FMPSymbols

Dieses Modul definiert eine Datenstruktur, zur Speicherung der Symbolinformationen und auf dieser, Funktionen zum Einfügen einer neuen Information und Mischen zweier solcher Datenstrukturen.
Unsere Anforderungen an diese Datenstruktur sind Einfügen und Mischen in konstanter Zeit, sowie die Möglichkeit, alle Symbole eines Teilbildes als transformiert oder zusätzlich benannt zu kennzeichnen. Ferner müssen die Regeln für die Verdeckung von Variablen, die wir in Abschnitt 3.17 definiert haben, umgesetzt werden. Ein schnelles Lookup wäre zwar wünschenswert, ist aber nicht so wichtig, wie bei einem Compiler für Maschinensprache, wo es viel wahrscheinlicher ist, daß eine Variable sehr oft referenziert wird.
Wir können wegen dieser speziellen Anforderungen keine bekannten effizienten Datenstrukturen wie Tries oder sortierte Bäume verwenden. Stattdessen begnügen wir uns mit einfachen, unsortierten Bäumen.[4] Die Suche verläuft in inorder–Reihenfolge von links nach rechts. Die Verdeckungsregeln der Variablenvorkommen können wir damit verwirklichen, daß Symbole, die andere Symbole verdecken sollen, in den linken Teilbaum gemischt und daher beim Lookup vor anderen Vorkommen gefunden werden. Für Punkt– und numerische Variablen verwenden wir jeweils eigene Symboltabellen, die wir in einem Record zusammenfassen.

$$
\begin{array}{lll}
\textbf{data}\ \mathit{Symbols} & = & \mathit{Symbols}\{\ \mathit{symPnts} :: \mathit{SymPoint}, \\
 & & \qquad\qquad \mathit{symNums} :: \mathit{SymNum}\} \\
 & & \textbf{deriving}\ (\mathit{Eq}, \mathit{Show})
\end{array}
$$

Die eigentlichen Informationen über die Zuordnung einer Variable zu einem Arrayelement in METAPOST, wie es im letzten Abschnitt beschrieben ist, speichert der Konstruktor *SymPName* bzw. *SymNName*. Die Benennung eines Bildes modelliert der Konstruktor *SymPHier*. Ein weiterer Punkt, den wir beachten müssen, sind Bildtransformationen wie sie durch die Funktionen *fill*, *clip*, *transform* und *truebox* auftreten können. Dabei kann ein Teilbild gegenüber dem globalen Koordinatensystem verschoben oder sogar allgemein affin transformiert werden. Wenn wir ein Bild also transformieren, müssen wir dies in der Symboltabelle für alle Punkte dieses Bildes vermerken. Wie wir später sehen werden, sind alle Teilbilder fortlaufend numeriert. Es genügt für eine Zuordnung zu einer Transformation somit die Nummer des transformierten Bildes zu speichern.

$$
\begin{array}{lll}
\textbf{data}\ \mathit{SymPoint} & = & \mathit{SymPName\ Name\ Int\ Int} \\
 & | & \mathit{SymPHier\ Name\ Int\ SymPoint} \\
 & | & \mathit{SymPUnion\ SymPoint\ SymPoint} \\
 & | & \mathit{SymPUnion3\ SymPoint\ SymPoint\ SymPoint} \\
 & | & \mathit{SymPTrans\ SymPoint\ Int} \\
 & | & \mathit{SymPRelax} \\
 & & \textbf{deriving}\ (\mathit{Eq}, \mathit{Show}) \\
 & & \\
\textbf{data}\ \mathit{SymNum} & = & \mathit{SymNName\ Name\ Int\ Int} \\
 & | & \mathit{SymNHier\ Name\ Int\ SymNum} \\
 & | & \mathit{SymNUnion\ SymNum\ SymNum} \\
 & | & \mathit{SymNUnion3\ SymNum\ SymNum\ SymNum} \\
 & | & \mathit{SymNRelax} \\
 & & \textbf{deriving}\ (\mathit{Eq}, \mathit{Show})
\end{array}
$$

[4] Obwol wir Bäume verwenden, wollen wir trotzdem von einer Symboltabelle reden.

Um die Konkatenation von Symboltabellen übersichtlicher zu notieren, wird *Symbols* Instanz der Klasse *HasConcat*.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{instance } \mathit{HasConcat\ Symbols} \textbf{ where}} \\
\quad a \,\&\, b & = & a\{\mathit{symPnts} = \mathit{symPnts}\ a \,\&\, \mathit{symPnts}\ b, \\
 & & \quad \mathit{symNums} = \mathit{symNums}\ a \,\&\, \mathit{symNums}\ b\}
\end{array}
$$

Neben den Konstruktoren *SymPUnion* und *SymNUnion* für eine normale Verschmelzung zweier Symboltabellen, benötigen wir zusätzlich die Konstruktoren *SymPUnion3* und *SymNUnion3*. Das hängt mit einem Aspekt der Verdeckung von Variablen zusammen, den wir bisher ignoriert haben. Die Namen von Variablen lassen sich ja aus verschiedenen Namensbestandteilen hierarchisch aufbauen. Einzelne Namensbestandteile können aber auch weggelassen werden, um den Namen abzukürzen. Wir wollen erreichen, daß lokale Vorkommen alle anderen verdecken, und daß vollständig angegebene Namen alle abgekürzten verdecken. Deshalb speichert der Konstruktor *SymPUnion3* im ersten Argument die Symboltabelle der lokalen Vorkommen, im zweiten Argument die Symbole, deren Namen durch die Benennung eines Bildes mit *SymPHier* erweitert wurden und das letzte Argument enthält die Abkürzungen der Namen und die restlichen Symbole. Nun können wir beim Mischen zweier Symboltabellen diese zusätzliche Verdeckungsregel beachten. Dieser Fall tritt beim Mischen der verschiedenen Symboltabellen der Bilder eines *overlay*–Befehls auf.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{instance } \mathit{HasConcat\ SymPoint} \textbf{ where}} \\
\quad \mathit{SymPRelax} \,\&\, b & = & b \\
\quad a \,\&\, \mathit{SymPRelax} & = & a \\
\multicolumn{3}{l}{\quad \mathit{SymPUnion3}\ a_1\ a_2\ a_3 \,\&\, \mathit{SymPUnion3}\ b_1\ b_2\ b_3} \\
 & = & \mathit{SymPUnion3}\ (a_1 \,\&\, b_1)\ (a_2 \,\&\, b_2)\ (a_3 \,\&\, b_3) \\
\quad a \,\&\, b & = & \mathit{SymPUnion}\ a\ b
\end{array}
$$

7.5.2 FMPResolve

In der ersten Phase der Variablenverwaltung geht es darum, ein Gleichungssystem nach definierenden Variablenvorkommen zu durchsuchen und diese dann in eine Symboltabelle einzutragen. In jedem Gleichungssystem startet der Vorgang mit einer leeren Symboltabelle. Für jedes Variablenvorkommen wird geprüft, ob die Variable in der Symboltabelle gefunden wird. Wenn nicht, handelt es sich um ein definierendes Vorkommen und die Variable wird in die Symboltabelle eingefügt.
Wir durchsuchen Gleichungen, numerische Ausdrücke und Punkte nach neuen Variablen. Dafür übergeben wir die Nummer des Bildes (Variable n), einen Zähler für die Anzahl der bisher gefundenen definierenden Variablenvorkommen in diesem Gleichungssystem (Variable m) und die bisherige Symboltabelle s. Für Punkte gilt z.B.: Bei eine Punktvariablen, deren Name mit *Global* beginnt, wird der Name nicht in die Symboltabelle aufgenommen, da es sich um ein angewandtes Vorkommen handelt. Andere Variablen werden ggf. in die Symboltabelle aufgenommen. Für die anderen Konstruktoren gilt, wir müssen alle Ausdrücke rekursiv nach definierenden Variablenvorkommen absuchen und dabei die Zähler und die Symboltabelle verwalten.

$$
\begin{array}{lll}
\mathit{symPoint} & :: & (\mathit{Int}, \mathit{Int}, \mathit{Symbols}) \to \mathit{Point} \to (\mathit{Int}, \mathit{Symbols}) \\
\multicolumn{3}{l}{\mathit{symPoint}\ (n, m, s)\ (\mathit{PointVar}\ (\mathit{Global}\ \_))} \\
 & = & (m, s) \\
\multicolumn{3}{l}{\mathit{symPoint\ nms}\ (\mathit{PointVar\ name})} \\
 & = & \mathit{insertPntName\ nms\ name}
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{symPoint}\ (n,m,s)\ (\mathit{PointPPP}\ \_\ a\ b)} \\
 & = & \mathit{symPoint}\ (n,m',s')\ b \\
\quad \textbf{where}\ (m',s') & = & \mathit{symPoint}\ (n,m,s)\ a \\
\multicolumn{3}{l}{\mathit{symPoint}\ (n,m,s)\ (\mathit{PointVec}\ (a,b))} \\
 & = & \mathit{symNumeric}\ (n,m',s')\ b \\
\quad \textbf{where}\ (m',s') & = & \mathit{symNumeric}\ (n,m,s)\ a \\
\multicolumn{3}{l}{\mathit{symPoint}\ (n,m,s)\ (\mathit{PointMediate}\ a\ b\ c)} \\
 & = & \mathit{symPoint}\ (n,m'',s'')\ c \\
\quad \textbf{where}\ (m',s') & = & \mathit{symNumeric}\ (n,m,s)\ a \\
\qquad\qquad (m'',s'') & = & \mathit{symPoint}\ (n,m',s')\ b \\
\multicolumn{3}{l}{\mathit{symPoint}\ (n,m,s)\ (\mathit{PointNMul}\ a\ b)} \\
 & = & \mathit{symPoint}\ (n,m',s')\ b \\
\quad \textbf{where}\ (m',s') & = & \mathit{symNumeric}\ (n,m,s)\ a \\
\multicolumn{3}{l}{\mathit{symPoint}\ \mathit{nms}\ (\mathit{PointDirection}\ a)} \\
 & = & \mathit{symNumeric}\ \mathit{nms}\ a \\
\mathit{symPoint}\ \mathit{nms}\ (\mathit{PointNeg}\ a) & = & \mathit{symPoint}\ \mathit{nms}\ a \\
\multicolumn{3}{l}{\mathit{symPoint}\ (n,m,s)\ (\mathit{PointCond}\ b\ t\ e)} \\
 & = & \mathit{symPoint}\ (n,m'',s'')\ e \\
\quad \textbf{where}\ (m',s') & = & \mathit{symBoolean}\ (n,m,s)\ b \\
\qquad\qquad (m'',s'') & = & \mathit{symPoint}\ (n,m',s')\ t \\
\mathit{symPoint}\ (\_,m,s)\ \_ & = & (m,s)
\end{array}
$$

Für die neun Bezugspunkte tragen wir keine Symbolinformationen ein, da wir sie gesondert behandeln und sie immer vordefiniert sind. Bei allen anderen Punktnamen prüfen wir, ob sie schon in der Symboltabelle für Punkte eingetragen sind. Wenn nicht, erzeugen wir einen neuen Eintrag und inkrementieren den Zähler der lokalen definierenden Variablenvorkommen.

$$
\begin{array}{lll}
\mathit{insertPntName} & :: & (\mathit{Int},\mathit{Int},\mathit{Symbols}) \to \mathit{Name} \to (\mathit{Int},\mathit{Symbols}) \\
\mathit{insertPntName}\ (n,m,s)\ \mathit{name} & = & \textbf{if}\ \neg\ (\mathit{lastNameIsDir}\ \mathit{name}) \\
 & & \wedge\ \mathit{resolvePoint}\ (n,s)\ (\mathit{PointVar}\ \mathit{name}) \equiv \mathit{Nothing} \\
 & & \qquad \textbf{then}\ (m+1, \mathit{addPDef}\ (\mathit{SymPName}\ \mathit{name}\ n\ m)\ s) \\
 & & \qquad \textbf{else}\ (m,s)
\end{array}
$$

In der zweiten Phase der Variablenverwaltung benutzen wir die Symboltabelle, um in einem Gleichungssystem alle Referenzen aufzulösen. Es existiert, wie in der ersten Phase, für jeden Datentyp der *functional* METAPOST–Sprachebene eine eigene Funktion, die alle Konstruktoren berücksichtigt und komplette Ausdrücke durchläuft.

Wie schauen uns wieder das Beispiel für Punkte an. Um einen Fehler für den Fall zu erhalten, daß ein Variablenname nicht in der Symboltabelle gefunden wird, benutzen wir den Typ *Maybe*. Bei einem Fehler ist das Ergebnis *Nothing*, sonst *Maybe a*, wobei *a* das eigentliche Ergebnis ist.

Das Prefix *Global* vor einem Namen ist nur für die erste Phase von Bedeutung und kann hier ignoriert werden. Der Punkt *PointVar* (*NameInt m*) ist eine lokale Variablenreferenz, die wir auflösen können ohne die Symboltabelle zu durchsuchen. Wir bilden die Variable wie in Abschnitt 7.5 besprochen, auf ein zweidimensionales Array ab, in dem die erste Dimension die Nummer *n* des Bildes und die zweite der Name *m* der Variable ist. Bei lokalen Referenzen auf Bezugspunkte verfahren wir analog. Wir sehen hier auch die besondere Behandlung des Namens `"last"`, der in der Beschreibung von Rahmen benötigt wird. Für alle anderen Variablennamen durchläuft die Funktion *resolvePointName* die Symboltabelle für Punkte von links nach rechts, um die Referenz des Namens nachzuschlagen. Der Name wird mit *flattenName* vorher in eine Normalform gebracht, z.B. wird aus $\mathit{Hier}\ (\mathit{Hier}\ a\ b)\ c$ der Ausdruck $\mathit{Hier}\ a\ (\mathit{Hier}\ b\ c)$.

$$
\begin{array}{lcl}
\mathit{resolvePoint} & :: & (\mathit{Int}, \mathit{Symbols}) \to \mathit{Point} \to \mathit{Maybe\ Point} \\
\mathit{resolvePoint\ ns}\ (\mathit{PointVar}\ (\mathit{Global\ name})) & & \\
 & = & \mathit{resolvePoint\ ns}\ (\mathit{PointVar\ name}) \\
\mathit{resolvePoint}\ (n, \_)\ (\mathit{PointVar}\ (\mathit{NameInt}\ m)) & & \\
 & = & \mathit{Just}\ (\mathit{PointVarArray}'\ n\ m) \\
\mathit{resolvePoint}\ (n, \_)\ (\mathit{PointVar}\ (\mathit{NameDir}\ d)) & & \\
 & = & \mathit{Just}\ (\mathit{PointPic}'\ n\ d) \\
\mathit{resolvePoint}\ (n, \_)\ (\mathit{PointVar}\ (\mathit{Hier}\ (\mathit{NameStr}\ \texttt{"last"})\ (\mathit{NameDir}\ d))) & & \\
 & = & \mathit{Just}\ (\mathit{PointPic}'\ (n+1)\ d) \\
\mathit{resolvePoint}\ (\_, s)\ (\mathit{PointVar\ name}) & & \\
 & = & \mathit{resolvePointName}\ (\mathit{symPnts}\ s)\ (\mathit{flattenName\ name}) \\
\mathit{resolvePoint\ ns}\ (\mathit{PointVec}\ (\mathit{nx}, \mathit{ny})) & & \\
 & = & \mathit{maybe2}\ (\lambda a\ b \to \mathit{PointVec}\ (a, b)) \\
 & & \qquad (\mathit{resolveNumeric\ ns\ nx}, \mathit{resolveNumeric\ ns\ ny}) \\
\mathit{resolvePoint\ ns}\ (\mathit{PointPPP}\ c\ p_1\ p_2) & & \\
 & = & \mathit{maybe2}\ (\mathit{PointPPP}\ c) \\
 & & \qquad (\mathit{resolvePoint\ ns}\ p_1, \mathit{resolvePoint\ ns}\ p_2) \\
\mathit{resolvePoint\ ns}\ (\mathit{PointMediate}\ a\ p_1\ p_2) & & \\
 & = & \mathit{maybe3\ PointMediate}\ (\mathit{resolveNumeric\ ns}\ a, \\
 & & \qquad\qquad\qquad \mathit{resolvePoint\ ns}\ p_1, \\
 & & \qquad\qquad\qquad \mathit{resolvePoint\ ns}\ p_2) \\
\mathit{resolvePoint\ ns}\ (\mathit{PointNMul}\ a\ p) & & \\
 & = & \mathit{maybe2\ PointNMul}\ (\mathit{resolveNumeric\ ns}\ a, \mathit{resolvePoint\ ns}\ p) \\
\mathit{resolvePoint\ ns}\ (\mathit{PointNeg}\ p) & = & \mathit{maybe}'\ \mathit{PointNeg}\ (\mathit{resolvePoint\ ns}\ p) \\
\mathit{resolvePoint\ ns}\ (\mathit{PointDirection}\ p) & & \\
 & = & \mathit{maybe}'\ \mathit{PointDirection}\ (\mathit{resolveNumeric\ ns}\ p) \\
\mathit{resolvePoint\ ns}\ (\mathit{PointCond}\ b\ t\ e) & & \\
 & = & \mathit{maybe3\ PointCond}\ (\mathit{resolveBoolean\ ns}\ b, \\
 & & \qquad\qquad\qquad \mathit{resolvePoint\ ns}\ t, \\
 & & \qquad\qquad\qquad \mathit{resolvePoint\ ns}\ e) \\
\mathit{resolvePoint}\ \_\ p & = & \mathit{Just}\ p
\end{array}
$$

Die Hilfsfunktionen *maybe′* und *maybe2* dienen der Weiterpropagierung von Fehlern und kürzen die Funktion *resolvePoint* etwas ab. Wenn eine Referenz für einen Teil eines Ausdrucks nicht aufgelöst werden kann, ergibt sich für den gesamten Ausdruck der Wert *Nothing*, ansonsten wird der angegebene Konstruktor auf die Teilausdrücke angewendet.

$$
\begin{array}{lcl}
\mathit{maybe}' & :: & (a \to b) \to \mathit{Maybe}\ a \to \mathit{Maybe}\ b \\
\mathit{maybe}'\ f\ (\mathit{Just}\ a) & = & \mathit{Just}\ (f\ a) \\
\mathit{maybe}'\ \_\ \_ & = & \mathit{Nothing}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{maybe2} & :: & (a \to b \to c) \to (\mathit{Maybe}\ a, \mathit{Maybe}\ b) \to \mathit{Maybe}\ c \\
\mathit{maybe2}\ f\ (\mathit{Just}\ a, \mathit{Just}\ b) & = & \mathit{Just}\ (f\ a\ b) \\
\mathit{maybe2}\ \_\ (\_, \_) & = & \mathit{Nothing}
\end{array}
$$

7.6 Die Generierung von abstraktem METAPOST–Code

Nachdem wir eine abstrakte Zwischensprache definiert haben, in der wir METAPOST–Programme darstellen können und die Variablenverwaltung besprachen, können wir uns nun der Aufgabe zuwenden, Bildbeschreibungen vom Typ *Picture* in die abstrakte Zwischensprache zu übersetzen. Dabei sind verschiedene Aspekte zu beachten: Die Symboltabellen müssen so verwaltet werden, daß die Sichtbarkeits– und Verdeckungsregeln für Variablen nicht verletzt werden. Wenn in der Bildbeschreibung Transformationen vorkommen, sind referenzierte Punkte in gleicher Weise zu transformieren, wie das Bild, in dem sie definiert sind usw.

Auch die Generierung des abstrakten METAPOST–Zwischencodes stellt uns vor Herausforderungen. Es dürfen z.B. keine noch unbekannten Variablen oder Variablen, an die noch kein Wert gebunden ist, in Zeichenbefehlen auftreten. In bestimmten Situationen müssen Teile der Zeichenphase schon in die Designphase vorgezogen werden. Dies ist der Fall, wenn die Größe eines Teilbildes erst nach dem Zeichnen des Bildes bestimmbar ist, aber schon in der Designphase für Berechnungen benötigt wird. Wir zeichnen dann das Bild in der Designphase, erhalten die Größeninformationen und speichern das Bild danach in einer Bildvariable bevor wir es löschen, um diese Bildvariable später in der Zeichenphase wieder zu zeichnen. Die Verwaltung der Attribute ist dagegen eine leichte Aufgabe.

Für unser weiteres Vorgehen ist es anschaulicher, wenn wir einen *Picture*–Ausdruck als einen Baum betrachten. Die Knoten bilden die Konstruktoren des Datentyps *Picture*, siehe Abbildung 7.2.3 auf Seite 101. In einem solchen Baum haben alle Knoten, abgesehen von *Overlay*, höchstens einen Sohn, da *Overlay* die einzige Möglichkeit darstellt, mehrere Bilder zu kombinieren. Abbildung 7.6.1 zeigt z.B. die Baumdarstellung des Beispiels aus Abbildung 3.3.2 von Seite 24.

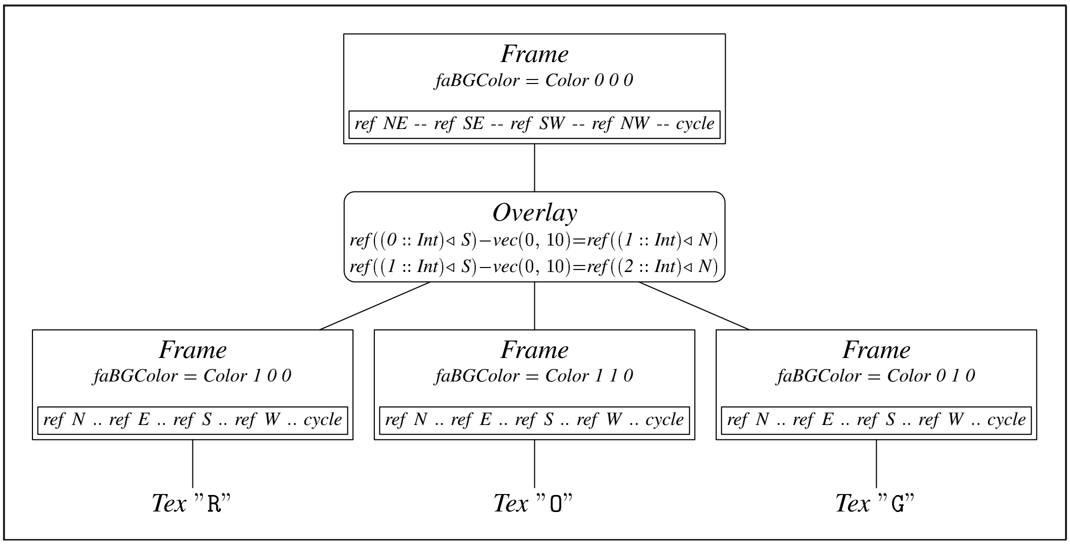


Abbildung 7.6.1: Baumdarstellung des Ausdrucks aus Abbildung 3.3.2. Die interessanten Parameter und Attribute sind zusätzlich angegeben.

Wir werden einen Ein–Pass–Compiler entwerfen, der diesen Baum abarbeitet und daraus METAPOST-Zwischencode generiert. Bei diesem Prozeß erhält jeder Knoten, außer *Attributes*, eine fortlaufende Nummer, die bei einigen wichtigen Aufgaben, wie z.B. der Variablenverwaltung, eine Identifikation der Knoten ermöglicht. Siehe Abbildung 7.6.2.

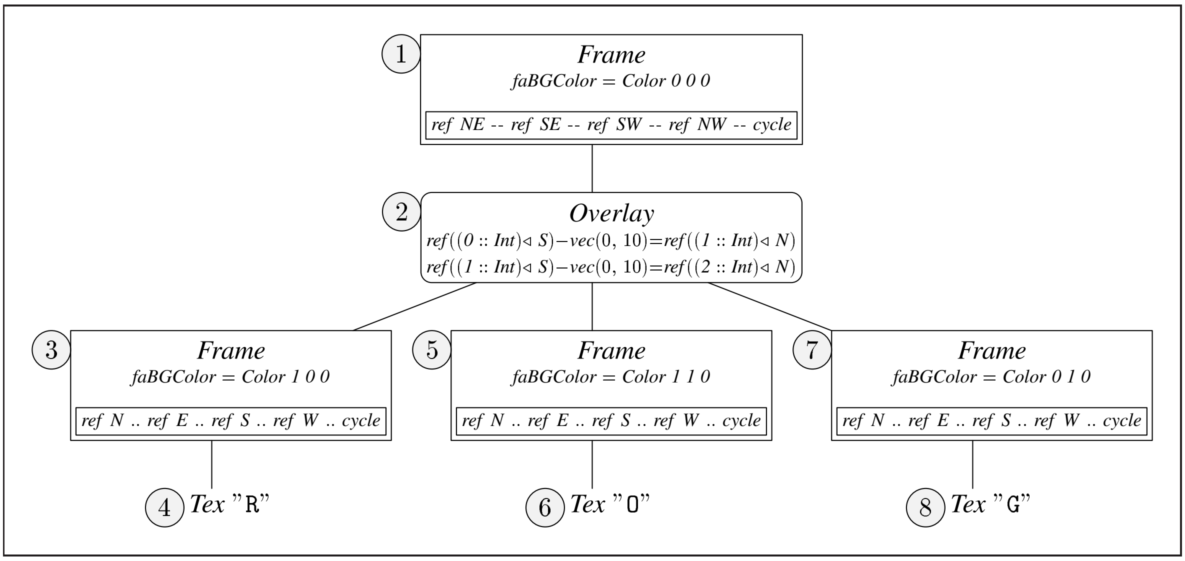


Abbildung 7.6.2: Die Teilbilder einer Bildbeschreibung erhalten eine eindeutige Nummer.

Die Funktion, die einen *Picture*–Ausdruck durchläuft und den Compiler realisiert, hat den Typ

$$mp \quad :: \quad Picture \rightarrow MPArg \rightarrow MPResult$$

Als Argument erhält sie die Nummer des nächsten Bildes und die bisherige Symboltabelle in der die Symbolinformationen stehen, die im Baum abwärts propagiert werden. Dies sind genau die globalen definierenden Variablenvorkommen. Der Rückgabewert umfaßt zusätzlich den generierten Zwischencode, getrennt nach Design– und Zeichenphase.

$$\begin{array}{lcl} \textbf{type}\ MPArg & = & (Int, Symbols) \\ \textbf{type}\ MPResult & = & (Int, Symbols, MetaPost, MetaPost) \end{array}$$

Um sicherzustellen, daß wir stets Code erzeugen, der korrekt von METAPOST interpretiert wird, benötigen wir ein Konzept, welches dies gewährleistet. Wir bedienen uns dabei der Rahmenboxen, die wir in Kapitel 6.5 vorgestellt haben. Diese nehmen uns einen Teil der Arbeit ab, denn eine Rahmenbox speichert die Bounding Box des Bildes, hat die Bezugspunkte `C, .., NW` vordefiniert, ist frei plazierbar und speichert bei Bedarf sogar ein Bild. Wir definieren Invarianten, die bei jeder Anwendung der Funktion *mp* einzuhalten sind. Sei *n* die Nummer eines Teilbildes, dann gelten für dieses folgende Invarianten.

1. In der Designphase dieses Bildes wird eine Box mit dem Namen `b`*n* erzeugt.
2. Nach der Designphase dieses Bildes muß die **Größe** der Box durch Gleichungsbeziehungen determiniert sein.
3. Nach der Designphase dieses Bildes darf die **Position** der Box **nicht** determiniert sein.
4. Wenn das Bild Nachfolger hat, muß nach der Designphase die relative Position des Bildes zu diesen bestimmt sein.

Die Übersetzung der Bildbeschreibung erfolgt, genau wie die Numerierung der Teilbilder, in preorder–Reihenfolge. Invariante 4. gewährleistet uns, daß die relativen Positionen aller Boxen, wenn die Designphase des letzten Bildes abgeschlossen ist, d.h. noch vor der Zeichenphase, determiniert sind. Wir besprechen nun die Übersetzung der wichtigsten Konstruktoren und vertiefen an gegebener Stelle weitere Fragen der Codeerzeugung und des Symbol–Informationsflusses.

7.6.1 TEX–Text

Die Behandlung von Texten bereitet keine Schwierigkeiten, siehe Abbildung 7.6.3. Wir erzeugen eine Box mit Namen `b`n, die das Bild des gewünschten Textes enthält (Invariante 1.). Deshalb ist der Bildzähler zu inkrementieren. Im nächsten Schritt wird Größe der Box bestimmt (Inv. 2.), indem den Variablen `b`n`.dx` und `b`n`.dy` konstante Werte zugewiesen werden. Die Position der Box steht nicht fest (Inv. 3.). Diese wird später in Anwendung von Invariante 4. des Vaterknotens bestimmt. Die Designphase ist damit abgeschlossen, Invariante 4. ist erfüllt, da der Knoten ein Blatt ist. In der Zeichenphase sorgt der Befehl `drawunboxed` dafür, daß der Text gezeichnet wird.

$$
\begin{array}{lcl}
\mathit{mp}\ (\mathit{Tex}\ s)\ (n, \mathit{symDown}) & = & (n+1, \mathit{relax}, \\
 & & \quad \mathit{MPBoxit}\ (\mathit{suff}\ n)\ (\mathit{MPTex}\ s) \\
 & & \quad \&\mathit{MPEquals}\ [\mathit{Id}\ (\mathit{suff}\ n +\!\!+ \texttt{".dx"}), \mathit{txtDX}] \\
 & & \quad \&\mathit{MPEquals}\ [\mathit{Id}\ (\mathit{suff}\ n +\!\!+ \texttt{".dy"}), \mathit{txtDY}], \\
 & & \quad \mathit{MPDrawUnBoxed}\ [\mathit{suff}\ n])
\end{array}
$$

Abbildung 7.6.3: Die Übersetzung von TEX–Text.

7.6.2 Das leere Bild

Bei einem leeren Bild sind, im Gegensatz zum Konstruktor *Tex*, numerischen Werte für die Breite w und Höhe h des Bildes angegeben, siehe Abbildung 7.6.4. In diesen Ausdrücken können Variablennamen vorkommen, die wir mit Hilfe der Symboltabelle auflösen müssen.

$$
\begin{array}{rcl}
\mathit{mp}\ (\mathit{Empty}\ w\ h)\ (n, \mathit{symDown}) & = & (n+1, \mathit{relax}, \\
 & & \quad \mathit{MPBoxit}\ (\mathit{suff}\ n)\ \mathit{relax} \\
 & & \quad \&\mathit{width}\ (\mathit{getDefault}\ (\mathit{resolveNumeric}\ (n, \mathit{symDown})\ w)\ 0) \\
 & & \quad \&\mathit{hight}\ (\mathit{getDefault}\ (\mathit{resolveNumeric}\ (n, \mathit{symDown})\ h)\ 0), \\
 & & \quad \mathit{MPDrawUnBoxed}\ [\mathit{suff}\ n]) \\
\textbf{where}\ \mathit{width}\ w & = & \mathit{MPEquals}\ [\mathit{tdot}\ (\mathit{suff}\ n)\ E, \\
 & & \qquad \mathit{tdot}\ (\mathit{suff}\ n)\ W + \mathit{pair}\ (\mathit{mpNumeric}\ w)\ 0] \\
\mathit{hight}\ h & = & \mathit{MPEquals}\ [\mathit{tdot}\ (\mathit{suff}\ n)\ N, \\
 & & \qquad \mathit{tdot}\ (\mathit{suff}\ n)\ S + \mathit{pair}\ 0\ (\mathit{mpNumeric}\ h)]
\end{array}
$$

Abbildung 7.6.4: Die Erzeugung eines leeren Bildes.

Der Ausdruck *getDefault* $(resolveNumeric\ (n, symDown)\ w)$ 0 liefert die Breite w mit aufgelösten Variablenreferenzen oder wenn eine Referenz nicht auflösbar ist, den Wert 0. Die Invarianten werden auch hier alle erfüllt.

7.6.3 Die Attributierung

Mit Hilfe des Konstruktors *Attributes* erfolgt die Attributierung der Bilder. Abbildung 7.6.5 zeigt, in welcher Weise Attribute ein Bild verändern. Bilder besitzen nur Attribute für die Farbe, die Hintergrundfarbe und eine Listen von Namen, die dem Bild gegeben wurden. Die Funktion *symNames* trägt diese Namen zusammen mit der Nummer n des betrachteten Teilbildes in die Symboltabelle ein. Wenn einem Bild keine besondere Farbe zugewiesen ist, muß nur noch der Hintergrund des Bildes berücksichtigt werden, bevor das Bild gezeichnet werden kann.

$$
\begin{array}{lll}
\multicolumn{3}{l}{mp\ (Attributes\ as\ p)\ (n, symDown)} \\
 & = & (n', symNames\ (aNames\ as)\ n\ symUp', l', \\
 & & \quad \textbf{case}\ aColor\ as\ \textbf{of} \\
 & & \quad DefaultColor \rightarrow drawBC\ as\ n\ \&\ z' \\
 & & \quad Graduate\ c_1\ c_2\ a\ num \\
 & & \qquad\quad \rightarrow MPImage\ rememberPic\ z'\ \&\ drawBC\ as\ n \\
 & & \qquad\quad\ \ \&\ MPGraduatePic\ (mpColor\ c_1)\ (mpColor\ c_2) \\
 & & \qquad\qquad\qquad\qquad (Id\ rememberPic)\ num\ a \\
 & & \quad c \rightarrow MPImage\ rememberPic\ z'\ \&\ drawBC\ as\ n \\
 & & \qquad\quad\ \ \&\ MPDrawPic\ (mpColor\ c)\ (Id\ rememberPic)) \\
\quad \textbf{where} & & \\
\quad (n', symUp', l', z') & = & mp\ p\ (n, symDown) \\
\quad rememberPic & = & \texttt{"r"} +\!\!+\ show\ n
\end{array}
$$

Abbildung 7.6.5: Die Umsetzung der Attributierung.

Hat ein Bild dagegen eine andere Farbe als *DefaultColor*, wird es ganz in dieser gezeichnet und die Farben innerhalb des Bildes bleiben unberücksichtigt.[5] Das Vorgehen ist bei Farbverläufen und homogenen Farben das Gleiche. Wir speichern das Bild der Zeichenphase z' in der Bildvariable *rememberPic*, zeichnen eventuell den Hintergrund und zeichnen letztendlich das Bild der Zeichenphase z' graduiert oder homogen.

Die Funktion *drawBC* zeichnet einen Hintergrund, wenn in der Attributierung eine Farbe hierfür gewählt ist, siehe Abbildung 7.6.6. Dabei hat die Fläche des Hintergrundes die Form des Bounding Box des Bildes, die ein beliebiger zyklischer Pfad sein kann. An diesem Punkt zeigt sich, wie nützlich die Verwendung der Rahmenboxen von METAPOST für uns ist. Den Pfad, der Bounding Box müssen wir auf diese Weise nicht explizit verwalten. Diesen erhalten wir in METAPOST mit der Funktion `bpath` $(suff\ \ n)$. Der entsprechende Ausdruck lautet in unserer Zwischensprache $MPBPath\ (id\ (suff\ \ n))$.

Weil für das Zeichnen der Attributierung kein neues Bild erzeugt wird, bleiben die Invarianten gewahrt.

[5]Dies ist der Grund, weshalb es den Konstruktor *DefaultColor* gibt und die voreingestellte Farbe nicht *black* ist. Sonst bestünde jedes Bild aus nur einer Farbe.

$$
\begin{array}{lll}
\mathit{drawBC} & :: & \mathit{Attrib} \rightarrow \mathit{Int} \rightarrow \mathit{MetaPost} \\
\mathit{drawBC}\ a\ n & = & \textbf{case}\ \mathit{aBGColor}\ a\ \textbf{of} \\
& & \quad \mathit{DefaultColor} \rightarrow \mathit{relax} \\
& & \quad \mathit{Graduate}\ c_1\ c_2\ a\ n' \\
& & \quad\quad \rightarrow \mathit{MPGraduate}\ (\mathit{mpColor}\ c_1)\ (\mathit{mpColor}\ c_2) \\
& & \quad\quad\quad\quad (\mathit{MPBPath}\ (\mathit{Id}\ (\mathit{suff}\ n)))\ n'\ a \\
& & \quad a \rightarrow \mathit{MPFill}\ (\mathit{MPBPath}\ (\mathit{Id}\ (\mathit{suff}\ n))) \\
& & \quad\quad\quad (\mathit{mpColor}\ a) \\
& & \quad\quad\quad \mathit{MPDefaultPen}
\end{array}
$$

Abbildung 7.6.6: Der Hintergrund eines Bildes.

7.6.4 Zeichenoperationen

Unsere Aufgabe besteht beim Zeichnen von Pfaden nicht nur darin, die Zeichenbefehle für den Pfad generieren. Wie wir in Abbildung 7.6.7 sehen können, gestaltet sich die Aufgabe etwas schwieriger, weil die Pfade in ihren Labeln Bilder enthalten können, die noch bezüglich der Numerierung und des erzeugten Codes berücksichtigt sein wollen. Deshalb müssen wir die Symboltabellen und den abstrakten METAPOST–Code aus Labeln vermischen. Für die Funktion *constructPath* tritt noch ein weiteres Problem auf. Wir haben die Eindeutigkeit für die Abbildung der Variablen auf Arrays in METAPOST bisher dadurch sichergestellt, daß pro Bild nur ein Gleichungssystem auftreten konnte. Damit konnte die Numerierung der definierenden Variablenvorkommen innerhalb eines Gleichungssystems immer von Null beginnen und wir mußten uns diesen Zähler daher nicht merken. Für Pfade haben wir allerdings die Anweisung *define* zugelassen, die ein Gleichungssystem für die Beschreibung von Punkten innerhalb eines Pfades ermöglicht und in dem Pfad beliebig oft vorkommen kann. Dies bedeutet aber, daß wir pro Bild, ja sogar pro Pfad, mehrere Gleichungssysteme haben können. Deshalb müssen wir den Zähler *m* der d.V. verwalten und als zusätzlichen Parameter übergeben. Die Anweisung *MPCloneit* erzeugt eine exakte Kopie, des Bildes, auf das die Pfade gezeichnet werden. Damit dürfen wir den Zähler *n* für das aktuelle Bild verwenden und die Invarianten sind alle erfüllt.

Die Behandlung des Konstruktors *Fill* erfolgt ähnlich, so daß wir diesen hier nicht noch gesondert besprechen müssen. Der einzige Unterschied besteht darin, daß die Flächen nach solchen sortiert werden, deren Attributierung besagt, daß sie unter das Bild zu zeichnen sind und nach solchen, die über das Bild gezeichnet werden.

7.6.5 *TrueBox*

Der Konstruktor *TrueBox* erzeugt zu einem Bild eine rechteckige Bounding Box, die dieses minimal umschließt, siehe Abbildung 7.6.8. Hierzu wird das Bild *p* schon in der Layoutphase gezeichnet, da die tatsächlichen Ausmaße der Bounding Box nur auf diese Weise ermittelt werden können. (Wir erinnern uns, daß *draw*– und *fill*–Befehle die Bounding Box des Bildes, auf das sie gemalt werden, nicht verändern. Deshalb kann ein Pfad oder eine Fläche außerhalb einer Bounding Box verlaufen.) Wir müssen also eine neue Rahmenbox erzeugen, die die korrekte Bounding Box besitzt. Der Konstruktor *MPSubBox* n z', siehe Abbildung 7.4.6 auf Seite 115, implementiert eine Art kleiner Unterprozedur, in der die Zeichenphase z' auf einer leeren Zeichenfläche ausgeführt wird. Die Größe des resultierenden

$$
\begin{array}{lcl}
mp\ (Draw\ ls\ p)\ (n, symDown) & = & (n'', symUp'' \ \&\ symUp', \\
 & & \quad l' \ \&\ l'' \ \&\ MPCloneit\ (suff\ n)\ (suff\ (n+1)), \\
 & & \quad z' \ \&\ z'') \\
\quad \textbf{where} & & \\
\quad nThis & = & n \\
\quad (n', symUp', l', z') & = & mp\ p\ (n+1, symDown) \\
\quad (n'', \_, symUp'', l'', z'') & = & paths\ ls\ n'\ 0 \\
\quad sym & = & symUp' \ \&\ symDown \\
\quad paths\ [\,]\ n\ m & = & (n, m, relax, relax, relax) \\
\quad paths\ (l : ls)\ n\ m & = & (n'', m'', symUp2' \ \&\ symUp2'', l' \ \&\ l'', z' \ \&\ z'') \\
\quad\quad \textbf{where} & & \\
\quad\quad (n', m', symUp2', l', z') & = & constructPath\ sym\ l\ n\ nThis\ m \\
\quad\quad (n'', m'', symUp2'', l'', z'') & = & paths\ ls\ n'\ m'
\end{array}
$$

Abbildung 7.6.7: Das Zeichnen von Pfaden.

Bildes können wir ermitteln und das Bild in eine neue Rahmenbox von korrekter Größe einfügen. Die Rahmenbox mit der Nummer n enthält nun das Bild, das wir aber erst endgültig in der Zeichenphase mit der Anweisung $MPDrawUnBoxed\ [suff\ n]$ zeichnen.

$$
\begin{array}{lcl}
mp\ (TrueBox\ p)\ (n, symDown) & = & (n', symTrans\ symUp'\ n, \\
 & & \quad defShift \ \&\ l' \ \&\ MPSubBox\ n\ z', \\
 & & \quad MPDrawUnBoxed\ [suff\ n]) \\
\quad \textbf{where} & & \\
\quad (n', symUp', l', z') & = & mp\ p\ (n+1, symDown) \\
\quad defShift & = & MPDef\ (tr\ n)\ (Shifted\ Identity\ (LLCorner\ (Pic\ (suff\ n)) \\
 & & \qquad\qquad -shiftRefPoint\ n))
\end{array}
$$

Abbildung 7.6.8: Die Ermittlung einer Bounding Box.

Abbildung 7.6.9 zeigt eine typische Situation, in der wir den Konstruktor *TrueBox* benötigen, nämlich wenn wir die Bounding Box eines Pfades oder einer Fläche ermitteln wollen. Das Beispiel ist extra so gewählt, daß ein Zeichenbefehl einen Punkt, innerhalb des Bildes mit der neuen Bounding Box, referenziert. Dieser Punkt muß dann nämlich verschoben werden. In Abbildung 7.6.9 sehen wir das Ergebnis des Übersetzungsvorgangs wo wir die Translation des Punktes in Zeile 21 entdecken können. Auch die Bounding Box `bpath b5` des Zeichens `"x"` muß für das Clipping in Zeile 23 verschoben werden. In den Zeilen 9 bis 18 befindet sich die vorgezogene Zeichenphase des Kreises.

7.6.6 Transformationen

Bei der Transformation eines Bildes können wir die Bounding Box des transformierten Bildes nicht berechnen ohne dieses zu zeichnen. Deshalb gehen wir, wie in Abbildung 7.6.11 zu sehen ist, analog

```
draw [ref NW -- ref ("x" ◁ C)]
     (setTrueBoundingBox
          (draw [(0, 30) .. (30, 0) .. (0, −30)
                .. (−30, 0) .. cycle
                # setLabel 0.4 C (tex "\\sf␣x"
                                  # setName "x")]
           empty))
```

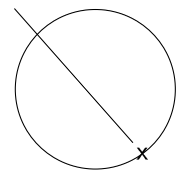


Abbildung 7.6.9: Wenn die Zeichenphase schon in der Layoutphase stattfindet, wird das Teilbild eventuell später an einer anderen Position gezeichnet, d.h. verschoben. Referenzen auf Punkte in diesem Bild müssen deshalb genauso verschoben werden.

```
def _t_DC = (identity) shifted (llcorner (pic b2)-(s2)) enddef;
boxit.b4();
b4.e = b4.w+(0,0);
b4.n = b4.s+(0,0);
boxit.b5(btex \sf x etex);
b5.dx = txtDX;
b5.dy = txtDY;
cloneit.b3(b4);
p2 := currentpicture; clearit;
drawunboxed(b4);
tempPath := (0,30)..(30,0)..(0,(-30))..((-30),0)..cycle;
draw (subpath (0,4) of tempPath);
b5.c = point 0.4*length(tempPath) of (tempPath);
drawunboxed(b5);
s2 := llcorner currentpicture;
boxit.b2(currentpicture);
b2.dx = 0; b2.dy = 0;
currentpicture := p2;
cloneit.b1(b2);
drawunboxed(b2);
tempPath := b1.nw--b5.c transformed _t_DC ;
draw (subpath (0,1) of tempPath cutbefore bpath b1
                                cutafter bpath b5 transformed _t_DC );
```

Abbildung 7.6.10: Das Compilat des Beispiels aus Abbildung 7.6.9.

zur der Übersetzung des Konstruktors *TrueBox* vor. Der Unterschied besteht darin, daß die angegebene Transformation, kombiniert mit der Verschiebung, die wir schon von *TrueBox* kennen, an eine Variable zugewiesen wird, um damit auch Punkte transformieren zu können, die innerhalb des Bildes liegen und ebenfalls transformiert werden müssen.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{mp}\ (\mathit{PTransform}\ (\mathit{Transformation}\ \mathit{xx}\ \mathit{xy}\ \mathit{yx}\ \mathit{yy}\ \mathit{dx}\ \mathit{dy})\ p)\ (n, \mathit{symDown})} \\
 & = & (n', \mathit{symTrans}\ \mathit{symUp}'\ n, \\
 & & \quad \mathit{defShift}\ \&\ l'\ \&\ \mathit{MPSubBox}\ n\ (\\
 & & \qquad z'\ \&\ \mathit{MPDefineTrans}\ \mathit{trn} \\
 & & \qquad\quad (\mathit{MPTransform}\ (\mathit{mpNumeric}\ \mathit{xx})\ (\mathit{mpNumeric}\ \mathit{xy}) \\
 & & \qquad\qquad (\mathit{mpNumeric}\ \mathit{yx})\ (\mathit{mpNumeric}\ \mathit{yy}) \\
 & & \qquad\qquad (\mathit{mpNumeric}\ \mathit{dx})\ (\mathit{mpNumeric}\ \mathit{dy})) \\
 & & \qquad \&\ \mathit{MPAssign}\ \mathit{CurrentPicture} \\
 & & \qquad\qquad (\mathit{Transformed}\ \mathit{CurrentPicture}\ (\mathit{Id}\ \mathit{trn}))), \\
 & & \quad \mathit{MPDrawUnBoxed}\ [\mathit{suff}\ n]) \\
\quad \textbf{where} & & \\
\quad (n', \mathit{symUp}', l', z') & = & \mathit{mp}\ p\ (n+1, \mathit{symDown}) \\
\quad \mathit{trn} & = & \texttt{"tr"} +\!\!+\ \mathit{show}\ n \\
\quad \mathit{defShift} & = & \mathit{MPDef}\ (\mathit{tr}\ n) \\
 & & \qquad (\mathit{Shifted}\ (\mathit{Transformed}\ \mathit{Identity}\ (\mathit{Id}\ \mathit{trn})) \\
 & & \qquad\qquad (\mathit{LLCorner}\ (\mathit{Pic}\ (\mathit{suff}\ n)) - \mathit{shiftRefPoint}\ n))
\end{array}
$$

Abbildung 7.6.11: Die Übersetzung affiner Transformationen.

7.6.7 Gleichungssysteme

Wir beschreiben nun noch die Übersetzung der Konstruktoren *Define*, *Overlay* und *Frame*, die eine Formulierung von Gleichungssystemen erlauben.

7.6.7.1 *Define*

Wir beginnen mit dem einfachsten Fall, nämlich dem von *Define*, wo lediglich ein Gleichungssystem zur Berechnung von Variablen auszuwerten ist, siehe Abbildung 7.6.12. Wir müssen hier die zwei Phasen der Variablenverwaltung, die wir in Abschnitt 7.5 beschrieben haben, koordinieren. Der Konstruktor *Define* erzeugt ein neues Bild mit der Nummer n. Warum ist das notwendig, wenn nicht das Bild, sondern lediglich die Symbolinformationen verändert werden? Der Grund liegt in der Art und Weise begründet, in der wir Variablenvorkommen auf die Arrays in METAPOST abbilden. Die eindeutigkeit der Abbildung ergibt sich aus der Nummer des Bildes und der Nummer des definierenden Vorkommens einer Variable, innerhalb des Gleichungssystems dieses Bildes. Wenn *Define* zweimal hintereinander vorkommt, muß die Abbildung der Variablen auf Arrays injektiv bleiben und deshalb müssen wir die Nummer des Bildes auch für *Define* inkrementieren.
Wir können nun beschreiben, wie die Symbolverwaltung vonstatten geht. Wir erinnern uns: Der erste Parameter der Funktion *symEquations* gibt die Nummer des betrachteten Bildes an, der zweite ist der Zähler für die definierenden Variablenvorkommen und der dritte Parameter akkumuliert die Tabelle für die neuen Symbole.

In der ersten Phase erzeugt die Funktion *symEquations* eine Symboltabelle, die alle definierenden Variablenvorkommen des Gleichungssystems *eqs* enthält.
Für die zweite Phase der Variablenverwaltung müssen in allen Gleichungen die Variablenvorkommen durch die Referenzen, wie sie in der Symboltabelle *sym* definiert sind, ersetzt werden. Die Symboltabelle *sym* setzt sich aus den neuen d.V. des Gleichungssystems (*newSym*), der Symboltabelle des Bildes *p* (*symUp*$'$) und aus den globalen d.V. (*symDown*) zusammen. Die Konkatenation der Symboltabellen in dieser Reihenfolge garantiert die Berücksichtigung der Verdeckungsregeln 1.,3. und 5. aus Abschnitt 3.17, da beim Lookup die Tabelle von links nach rechts durchsucht wird.

$$
\begin{array}{lcl}
\multicolumn{3}{l}{mp\ (Define\ eqs\ p)\ (n, symDown)} \\
 & = & (n', symUp', \\
 & & \ \ mpEquations\ (maybes2List\ eqs')\ \&\ l' \\
 & & \ \ \&MPCloneit\ (suff\ n)\ (suff\ (n+1)), z') \\
\quad \textbf{where} & & \\
\quad (n', symUp', l', z') & = & mp\ p\ (n+1, newSym\ \&\ symDown) \\
\quad newSym & = & snd\ (symEquations\ (n, 0, relax)\ eqs) \\
\quad eqs' & = & map\ (resolveEquation\ (n, sym))\ eqs \\
\qquad\qquad \textbf{where}\ sym & = & newSym\ \&\ symUp'\ \&\ symDown
\end{array}
$$

Abbildung 7.6.12: Die Übersetzung des Konstruktors *Define*.

7.6.7.2 *Overlay*

Mit Hilfe des Konstruktors *Overlay* können wir mehrere Bilder kombinieren und dabei deren relative Positionen beschreiben. Dabei sind folgende Aufgaben zu bewältigen: Die Symbolinformationen müssen verteilt werden, d.h. Symbolinformationen eines Sohnes (hier benutzen wir wieder die Sicht eines Baumes) müssen auch für alle anderen Söhne sichtbar sein. Wir müssen beachten, daß im Gleichungssytem *eqs* die Bilder der Söhne unter den Namen 0 .. *n* referenzierbar sind, wenn wir die zwei Phasen der Symbolverwaltung implementieren. Die Layout und Zeichenphasen der Söhne sind geeignet zu mischen, die Numerierung der Söhne muß erfolgen und die neue Bounding Box ist zu bestimmen. Abbildung 7.6.13 zeigt, wie wir diese Aufgaben formulieren können.
Für jeden Sohn wird nacheinander rekursiv, mit der Funktion *mp*, Code erzeugt und von der Funktion *mergeResult* vermischt. Dabei erfolgt gleichzeitig die Numerierung und ein Ansammeln der Symboltabellen. in der Variable *symUp*$'$ bilden wir die Vereinigung dieser Symboltabellen, die wir an alle Söhne übergeben.
Im Ausdruck *symLocalNames* konstruieren wir eine Symboltabelle, in der die Bilder der Söhne unter den Namen $0 \mathinner{..} n-1$ eingetragen und die Namen in den Symboltabellen der Söhne, am Anfang durch den Konstruktor *symHier* verlängert sind. Um eine Zuordnung des *n*-ten Sohnes auf den Bildzähler *n* und die Symboltabelle dieses Sohnes zu erhalten, weisen wir diese Information der Variable *mapping* zu.
Die Funktion *newBBox* wählt die Größe der Bounding Box so, daß sie alle Bounding Boxen der Söhne umschließt, wenn keine andere Wahl getroffen wurde. Sonst wird die gewünschte Bounding Box für das aktuelle Bild *n* kopiert.
Der Anwender hat innerhalb des Gleichungssystems die Möglichkeit gegen die invarianten 3. und 4.

zu verstoßen, indem er Bilder an absoluten Koordinaten positioniert, oder die relative Position von Bildern nicht bestimmt. Dies ist der Nachteil den wir durch die Leistungsfähigkeit der Gleichungssysteme erkaufen.

```
mp (Overlay _ _ []) ns          = mp (Empty 0 0) ns
mp (Overlay eqs bbox ps) (n, symDown)
                                = (n', newSym & symUp',
                                   l' & mpEquations (maybes2List eqs') & newBBox bbox,
                                   z')
  where
  (n' : ns', symSons', l', z')  = foldr (λj i → mergeResult (mp j (params i)) i)
                                        ([n + 1], [], relax, relax) ps
  mergeResult (n, sym, l, z) (ns', syms', l', z')
                                = (n : ns', sym : syms', l & l', z & z')
  params (n : _, _, _, _)       = (n, symUp' & symDown)
  params ([], _, _, _)          = (n, symUp' & symDown)
  mapping                       = zip3 [0 . .] ns' symSons'
  symUp'                        = symUnions symSons'
  symLocalNames                 = symUnion3
                                        (symUnions [addPDef (SymPName (toName n) m 0)
                                                               relax
                                                   |(n, m, _) ← mapping])
                                        (symUnions [symHier n m sym
                                                   |(n, m, sym) ← mapping])
                                        symUp'
  newSym                        = snd (symEquations (n, 0, relax) eqs)
  eqs'                          = map (resolveEquation (n, sym)) eqs
              where sym         = newSym & symLocalNames & symDown
  newBBox Nothing               = MPBoxit (suff n) relax
                                  &MPEquals [tdot (suff n) SW,
                                            pair (Min [XPart (tdot (suff n) W)|n ← ns'])
                                                 (Min [YPart (tdot (suff n) S)|n ← ns'])]
                                  &MPEquals [tdot (suff n) NE,
                                            pair (Max [XPart (tdot (suff n) E)|n ← ns'])
                                                 (Max [YPart (tdot (suff n) N)|n ← ns'])]
                                  &MPFixSize [suff n]
  newBBox (Just b)              =  let nBBox = head ([n|(m, n, _) ← mapping, m ≡ b] ++ [0])
                                   in MPCloneit (suff n) (suff nBBox)
```

Abbildung 7.6.13: Die Übersetzung des Konstruktors *Overlay*.

7.6.7.3 Rahmen

Das Gleichungssystem eines Rahmens dient einzig dazu, die Positionen der neun Bezugspunkte und der Punkte des Rahmenpfades zu bestimmen. Aus diesem Grund sind alle anderen Variablen des Gleichungssystems nur in diesem lokal und nicht außerhalb sichtbar. Wie Abbildung 7.6.14 zeigt, müssen wir daher für die Symbolinformation, die wir nach obem im Baum propagieren, lediglich die

Symbole aus Bild *p* um die Namen ergänzen, die eventuell in den den Attributen *fa* vermerkt sind.
In der Layoutphase erzeugen wir mit dem Konstruktor *MPShapeit* eine Rahmenbox, die sich von der rechteckigen Rahmenbox dadurch unterscheidet, daß sie den Pfad der Umrandung in einer Variable des Namens *suff* n ++ `".p"` speichert. Aber diese Speicherung der Beschreibung des Pfades erfolgt als Textstring und nicht in mit einer Pfadvariable. Dieser Umweg ist notwendig, da wir einen Pfad nicht an eine Variable zuweisen können, solange nicht alle Punkte in ihm bekannt sind, was in der Layoutphase nicht der Fall ist. METAPOST würde bei der Zuweisung versuchen, den genauen Verlauf des Pfades festzulegen, was zu einem Fehler führen würde. Erst bevor der Rahmenpfad in der Zeichenphase gezeichnet wird und alle Punkte determiniert sind, wandelt ein Aufruf der Funktion `bpath` mittels `shapepath`, den String in einen Pfad um. Diesen Trick benutzen teilweise auch die Makros von METAPOST.

```
vardef shapeit@# =
  beginbox_("shapepath_","sizebox_",@#,"");
  generic_declare(string) _n.p;
  generic_declare(pair) _n.n,_n.ne,_n.e,_n.se,_n.s,_n.sw,_n.w,_n.nw,_n.c;
enddef;

def shapepath_(suffix $) = scantokens $.p enddef;
```

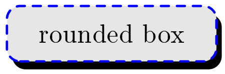
Die Funktion *drawFrameBC* ist ähnlich implementiert, wie *drawBC*, außer, daß eventuell ein Schatten am Anfang der Zeichenphase, also unter den Rahmen gezeichnet wird. Mit der Funktion *drawBorder* wird das Bild zuletzt noch um den Rahmen ergänzt.
Abbildung 7.6.15 zeigt die Bildbeschreibung eines Rahmens mit einigen Attributsveränderungen und den daraus generierten Quellcode.

$$
\begin{array}{lll}
\multicolumn{3}{l}{mp\ (Frame\ fa\ eqs\ path\ p)\ (n, symDown)} \\
 & = & (n', symNames\ (faNames\ fa)\ n\ symUp', \\
 & & \ l'\ \&\ MPShapeit\ (suff\ n)\ \&\ mpEquations\ eqs'\ \&\ assignPath, \\
 & & \ drawFrameBC\ sym\ fa\ n\ \&\ z'\ \&\ drawBorder\ fa\ n) \\
\quad \textbf{where} & & \\
\quad (n', symUp', l', z') & = & mp\ p\ (n+1, relax) \\
\quad sym & = & newSym\ \&\ symUp'\ \&\ symDown \\
\quad newSym & = & snd\ (symEquations\ (n, 0, relax)\ eqs) \\
\quad eqs' & = & maybes2List\ (map\ (resolveEquation\ (n, sym))\ eqs) \\
\quad assignPath & = & \textbf{case}\ resolvePath\ (n, 0, sym)\ path\ \textbf{of} \\
 & & \qquad Nothing \rightarrow relax \\
 & & \qquad Just\ (\_, p') \rightarrow MPAssign\ (Id\ (suff\ n +\!\!\!+ \texttt{".p"})) \\
 & & \qquad\qquad (Id\ \$\ show\ \$\ show\ \$\ emit\ \$\ mpPath\ p')
\end{array}
$$

Abbildung 7.6.14: Die Übersetzung von Rahmen.

Nachdem wir nun Ausdrücke unserer Bildbeschreibung in METAPOST–Code übersetzen können, auf Seite 166 befindet sich ein längeres Beispiel, benötigen wir noch eine Funktion, die den Übersetzungsvorgang kontrolliert.

rbox 5 "rounded␣box"
#*setDX* 10 # *setDY* 5 # *setColor blue* # *setBGColor* 0.9
#*setPattern dashed* # *setPen* 1 # *setShadow* $(2, -3)$




```
boxit.b2(btex rounded box etex);
b2.dx = txtDX;
b2.dy = txtDY;
shapeit.b1;
nv1 0 = 10;
b1.e = b2.e+(nv1 0,0);
b1.w = b2.w-((nv1 0,0));
nv1 1 = 5;
b1.n = b2.n+(0,nv1 1);
b1.s = b2.s-((0,nv1 1));
b1.c = b2.c;
(xpart (b1.ne)) = (xpart (b1.se));
(ypart (b1.nw)) = (ypart (b1.ne));
b1.w = 0.5[b1.nw,b1.sw];
b1.s = 0.5[b1.sw,b1.se];
b1.e = 0.5[b1.ne,b1.se];
b1.n = 0.5[b1.ne,b1.nw];
nv1 2 = min(5,(0.5)*((ypart (b1.n-(b1.s)))), (0.5)*((xpart (b1.e-(b1.w)))));
b1.p := "b1.sw+(nv1 2,0)---b1.se-((nv1 2,0))..b1.se+(0,nv1 2)
        ---b1.ne-((0,nv1 2))..b1.ne-((nv1 2,0))
        ---b1.nw+(nv1 2,0)..b1.nw-((0,nv1 2))
        ---b1.sw+(0,nv1 2)..cycle";
fill bpath b1 shifted (2,(-3)) withcolor (0,0,0)  ;
fill bpath b1 withcolor (0.9,0.9,0.9)  ;
drawunboxed(b2);
draw (bpath b1) dashed dashpattern (on 3 off 3 )
                withcolor (0,0,1)
                withpen pencircle scaled (1) ;
```

Abbildung 7.6.15: Die Beschreibung eines Rahmens und der generierte METAPOST–Code, hier mit manuell zusätzlich eingefügten Zeilenumbrüchen.

7.7 Die Generierung der Bilder

Wenn wir das Haskellprogramm `lhs2TeX` verwenden, können wir Bildbeschreibungen im Dokumenttext speichern. Bei diesem Konzept der Vermischung von Programm und erläuterndem Text spricht man von „literate programming“ [Knu83]. Um den Programmcode vom Dokumenttext unterscheiden zu können, beginnt jede Programmzeile mit dem Zeichen “>“. Wenn Hugs eine solche Datei mit der Endung `.lhs` läd, ignoriert der Scanner alle Zeilen, die nicht mit “>“ als Programmzeile gekennzeichnet sind. Das Programm `lhs2TeX` erzeugt aus einer solchen Quelldatei eine TEX–Datei, indem es Funktionsdefinitionen sauber formatiert und Text unverändert läßt. Diese Arbeit ist komplett mit diesem Programm formatiert.

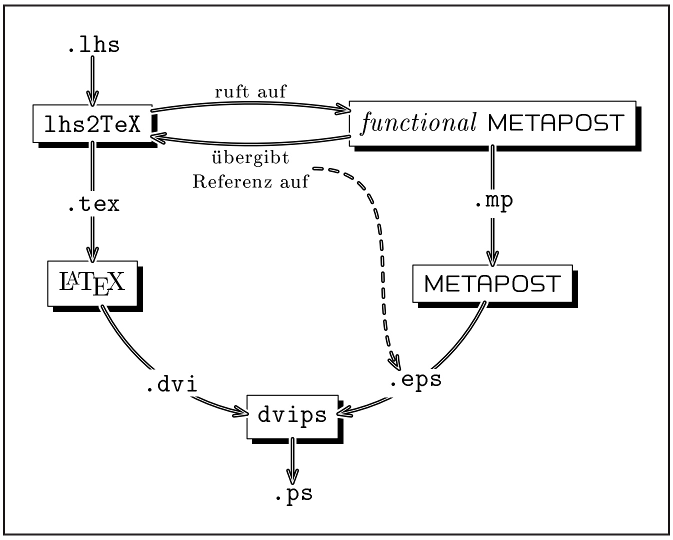
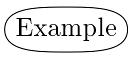


Abbildung 7.7.1: Bildbeschreibungen können direkt in Textdokumenten stehen. Die Bilder können dann automatisch generiert werden.

Sehr interessant und praktisch ist der Befehl `\perform{`⟨*Haskell–Ausdruck*⟩`}`, der `lhs2TeX` anweist, Hugs mit dem aktuell bearbeiteten Modul aufzurufen und den Ausdruck auszuwerten. Wenn diese Auswertung einen String ergibt, wird dieser an die Stelle des `\perform`–Befehls in den Text gesetzt.

Schauen wir uns diesen Vorgang an einem Beispiel näher an. Wenn wir folgendes Bild in unser Dokument einfügen wollen,



können wir dies erreichen, indem der Quelltext zu dieser Stelle lautet:

```
Wenn wir folgendes Bild in unser Dokument einf"ugen wollen,

\perform{generate "NameOfExample" 1 (oval "Example")}

k"onnen wir dies erreichen, indem der Quelltext zu dieser Stelle lautet:
```

Trifft `lhs2TeX` auf den Befehl `\perform{generate "NameOfExample" 1 (oval "Example")}`, ruft es Hugs auf und startet die Generierung des Bildes *oval* `"Example"` an deren Ende das Bild mit dem Dateinamen `NameOfExample.1` ensteht. Die Funktion *generate* liefert, nachdem METAPOST das Bild erfolgreich generiert hat, die Zeichenkette `"\includegraphics{NameOfExample.1}"` zurück, s.d. an dieser Stelle in der LATEX–Datei eine Referenz auf die Grafik erscheint, siehe Abbildung 7.7.1.

```
Wenn wir folgendes Bild in unser Dokument einf"ugen wollen,

\includegraphics{NameOfExample.1}

k"onnen wir dies erreichen, indem der Quelltext zu dieser Stelle lautet:
```

Auf diese Weise können erstens Bildbeschreibungen direkt in Textdokumenten stehen und zweitens die Bilder automatisch generiert werden.

7.7.1 Dateifunktionen

Es gibt einige Parameter, die der Anwender vielleicht anpassen möchte. Z.B. ist der Dateiname von METAPOST systemabhängig. Deshalb gibt es die Möglichkeit in einer Datei namens `.FuncMP` solche Informationen abzulegen. Die Funktion *getParam* liest die Einstellungen und setzt die entsprechenden Felder in einem Record. Wenn die Datei nicht existiert oder ein Wert nicht angegeben ist, gelten folgende Voreinstellungen:

$$
\begin{array}{lll}
\mathit{stdParameters} & :: & \mathit{Parameters} \\
\mathit{stdParameters} & = & \mathit{Parameters}\{\,\mathit{mpBin} = \texttt{"virmp"}, \\
 & & \qquad \mathit{textDX} = 2, \\
 & & \qquad \mathit{textDY} = 2, \\
 & & \qquad \mathit{newmp} = \mathit{False}, \\
 & & \qquad \mathit{prolog} = \mathit{prolog}', \\
 & & \qquad \mathit{epilog} = \texttt{"\\\\end"}\}
\end{array}
$$

Die Zeichenkette *prolog′* enthält Text, der an den Anfang des METAPOST–Quellcodes angefügt wird. Die Anweisungen `\input boxes` und `\input FuncMP` laden die METAPOST–Makrosammlung für Rahmenboxen und die speziellen Makros unserer Bildbeschreibungssprache.
Das Programm METAPOST ruft LaTeX auf, um die Größe der LaTeX–Texte herauszufinden. Dazu wird eine LaTeX–Datei erzeugt, die alle Texte eines Bildes enthält und dann mit LaTeX übersetzt. Aus der resultierenden DVI–Datei lassen sich die benötigten Informationen auslesen. Mit Hilfe des Textes zwischen den Schlüssenwörtern *verbatimtex* und *etex*, der an den Anfang der LaTeX–Datei gestellt wird, können für die Texte Formatierungen eingestellt und LaTeX–Makros geladen werden.

```
prolog'   :: String
prolog'   =  "verbatimtex\n\
             \\\documentclass[11pt]{report}\n\
             \\\begin{document}\n\
             \etex\n\n\
             \input␣boxes\n\
             \input␣FuncMP"
```

Die Variablen *textDX* und *textDY* definieren den Abstand den die Bounding Box für Bilder, die mit *tex* entstanden zum Text hat. Ein Wert von 2 bp hat sich empirisch bewährt, ist aber trotzdem veränderbar. Pfade werden mit diesem Wert früh genug vor dem Bild abgeschnitten und Rahmen haben auch einen wohldefinierten Abstand zum Text.
Das Flag *newmp* dient noch zur Berücksichtigung einer Inkompatibilität. Eine neuere Version von METAPOST hängt die Nummer des Bildes noch zusätzlich an den Namen der erzeugten PostScript–Datei, die alte Version nicht.

7.7.2 Die Hauptfunktion

Wir können nun die Funktion *generate* definieren, welche die gesamte Kontrolle der Bildgenerierung übernimmt. Wie in Abbildung 7.7.2 zu sehen ist, werden für jedes Bild die Parameter aus der Datei

`.FuncMP` ausgelesen, bevor das Ergebnis der Codegenerierung in die entsprechende METAPOST–Datei geschrieben wird. Wenn der Systemaufruf des Programms METAPOST mit Erfolg beendet wurde, wird ein LaTeX–Befehl zur Einbindung der Grafik in die Standardausgabe geschreiben, sonst erscheint eine Warnung.

```
generate                :: IsPicture a => String -> Int -> a -> IO ()
generate prefix n pic   = getParameters >>= doOutput
    where
    filePath            = prefix ++ "." ++ show n
    fileName param      = if newmp param
                              then prefix ++ ".mp"
                              else filePath ++ ".mp"
    mpDoc param         = emit (metaPost n (toPicture pic) param)
    doOutput param      = do writeFile (fileName param)
                              (show (mpDoc param))
                            err <- system (mpBin param ++ " "
                                           ++ fileName param
                                           ++ " >> /dev/null")
                            if err == ExitSuccess
                               then putStr ("\\includegraphics{"
                                           ++ filePath ++ "}")
                               else putStr ("Generation of picture "
                                          ++ filePath ++ " failed!")
```

Abbildung 7.7.2: Die Funktion *generate* steuert die gesamte Übersetzung.

Die Funktion *metaPost*, die in Abbildung 7.7.3 dargestellt ist, fügt das durch Aufruf der Funktion *mp* compilierte Bild, zwischen den Text *prolog* und *epilog* aus Datei `.FuncMP` ein. Die Anweisung `batchmode;` weist METAPOST an, Statusinformationen zu unterdrücken und im Falle eines Fehlers nicht auf eine Eingabe zu warten, damit keine Endlosschleifen entstehen. Die Anweisung `warningcheck := 0` unterdrückt eine Ausgabe von Warnungen bei Zahlen, die größer als 4096 sind. Variablen müssen, wenn sie nicht vom Typ `numeric` sind, am Anfang der Bildbeschreibung deklariert sein.

7.8 Erweiterungen von METAPOST

Wie wir bereits in Kapitel 6 feststellten, unterstützt METAPOST nicht alle Funktionen unserer Bildbeschreibungssprache. Wir werden deshalb eine Sammlung von Makros entwerfen, um diese Funktionalität auf effiziente Art und Weise bereitzustellen.

7.8.1 Farbverläufe

Alle Objekte, die man mit den Zeichenfunktionen von METAPOST erzeugt, haben eine homogene Farbe. Wenn wir Farbverläufe erhalten wollen, müssen wir das Bild aus vielen diskreten Farben zusammensetzen. Wenn wir für diese Farben entsprechend viele Zeichenbefehle generieren, vergrößert sich der METAPOST–Quellcode stark. Deshalb werden wir METAPOST–Makros entwickeln, die

metaPost	::	*Int* → *Picture* → *Parameters* → *MetaPost*
metaPost n t param	=	*MPVerbatim* `"batchmode;"`
		&*MPVerbatim* (*prolog param*)
		&*MPFigure n*
		(*MPAssign* (*Id* `"warningcheck"`) 0
		&*MPVerbatim* `"picture␣p[],␣q[],␣r[];"`
		&*MPVerbatim* `"transform␣t[],tr[];"`
		&*MPVerbatim* `"pair␣s[];"`
		&*MPVerbatim* `"pair␣pv[][];"`
		&*MPVerbatim* `"pair␣pvi[][];"`
		&*MPVerbatim* `"numeric␣nv[][];"`
		&*MPVerbatim* `"numeric␣nvi[][];"`
		&*MPVerbatim* `"path␣tempPath;"`
		&*MPAssign txtDX* (*Const* (*textDX param*))
		&*MPAssign txtDY* (*Const* (*textDY param*))
		&*l* & *z*)
		&*MPVerbatim* (*epilog param*)
where $(\_,\_,l,z)$	=	*mp t* $(1, relax)$

Abbildung 7.7.3: Die Funktion *metaPost* generiert den METAPOST–Code.

eine entsprechende Zahl verschiedenfarbiger Zeichenbefehle erzeugen. Im Quelltext rufen wir einfach das entsprechende Makro auf. Die Größe der generierten PostScriptdatei vergrößert sich leider trotzdem.[6]

Zuerst untersuchen wir, welche Funktionalität Farbverläufe verlangen. Es sind drei verschiedene Situationen, zu berücksichtigen.

1. Eine **Fläche** soll mit *fill* gezeichnet werden.
2. Ein komplettes **Bild** soll mit einem Farbverlauf versehen werden.
3. Ein **Pfad** wird entweder als Umrandung eines Rahmens oder von *draw* gezeichnet.

Überlegen wir kurz, wie wir diese Punkte implementieren können. Dazu sei n die Anzahl der Abstufungen von der Startfarbe bis zur Zielfarbe. α sei der Winkel des Farbverlaufs.

1. Das Füllen einer Fläche ist der einfachste Fall. Abbildung 7.8.1 zeigt die vier Schritte zur Konstruktion der graduierten Fläche.
   (a) Zeichne den Umriß der Fläche um $-\alpha$ rotiert in eine Bildvariable.
   (b) Frage die Ausmaße des Bildes ab. Zeichne in n Schritten den Farbverlauf aus horizontalen Strichen die genau übereinanderpassen, d.h. mit ausreichender Dicke. Damit entsteht ein Rechteck das den gedrehten Umriß genau umschließt.

[6]Daran ist nichts zu ändern. METAPOST erzeugt PostScript Level 2, aber erst PostScript Version 3 kann mit Farbverläufen umgehen.

(c) Wende die Funktion clip auf die Bildvariable an und stanze damit die Form des rotierten Umrisses aus.

(d) Zeichne die Bildvariable um α rotiert.

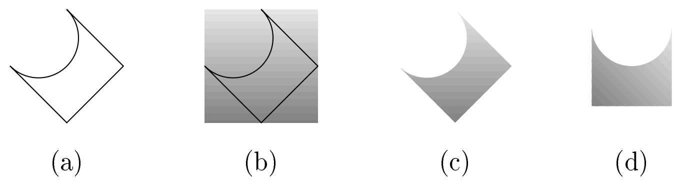


Abbildung 7.8.1: Zur Konstruktion einer Fläche mit Farbverlauf vom Drehwinkel α, sind vier Schritte notwendig

2. Das Zeichnen eines kompletten Bildes mit einer Graduierung funktioniert genauso aber anstatt n mal eine horizontale Linie in der nächsten Farbe zu zeichnen, zeichne n mal das komplette Bild geclipt an einem Rechteck, von der Breite des Bildes und der Höhe $\frac{1}{n}$, das von unten nach oben über das Bild wandert, mit aufsteigenden Farben.

3. Man könnte meinen, das Zeichnen von graduierten Pfaden sei am einfachsten. Das stimmt nur unter der Voraussetzung, daß der Pfad kein Strichmuster besitzt. Dann lassen sich n Teilpfade mit aufsteigender Farbe zeichnen. Aber bei Strichmustern funktioniert diese Methode nicht, weil das Muster bei jedem gezeichneten Teilpfad von neuem beginnt und nicht gleichmäßig weiterläuft.

   Wenn beim Zeichnen von n Teilpfaden Probleme mit Strichmustern auftreten, bleibt nur noch die Möglichkeit den gesamten Pfad n mal zu zeichnen und die Teilpfade mittels Clipping zu erhalten. In zahlreichen Experimenten haben sich als Clipping–Pfad Kreise als geeignet erwiesen. Abbildung 7.8.3 zeigt, wie acht Kreise den Pfad in acht Teilpfade verschiedener Farbe aufteilen. Ähnlich dem Vorgehen bei 2. zeichnet das METAPOST–Makro n mal den kompletten Pfad in aufsteigender Farbe in eine leere Bildvariable und führt dann das Clipping durch. Die Bildvariablen werden schließlich zum fertigen Bild vereinigt und gezeichnet.

   Ein Effekt tritt allerdings bei großen Zeichenstiften auf, wenn gleichzeitig die Anzahl der Kreise groß ist und damit ihr Durchmesser schrumpft. In diesem Fall schneidet das Clipping eventuell zu viel vom Pfad weg. Um trotzdem Pfade mit größerem Durchmesser zeichnen zu können, zeichnet das Makro, wenn kein Strichmuster vorliegt, mit der ersten Idee, n Teilpfade aufsteigender Farbe.

7.8.2 Bitmaps

Bitmap–Grafiken lassen sich in METAPOST nur sehr schwierig darstellen. Es bedarf schon besonderer Ideen um dies in befriedigender Weise zu implementieren. Wie wollen diese Ideen schrittweise entwickeln.

```
def graduate(expr a, b, border, n, alpha) =
begingroup
  picture pict,pictt;
  numeric h;

  pict:=nullpicture;
  pictt:=nullpicture;
  addto pictt doublepath border rotated -alpha;
  h:=ypart (ulcorner pictt-llcorner pictt);
  for i=0 upto n:
    addto pict doublepath llcorner pictt + (0,i/n*h)
                          -- lrcorner pictt + (0,i/n*h)
                 withcolor i/n*a+(n-i)/n*b
                 withpen pencircle scaled (h/n+1);
  endfor
  pict:=pict rotated alpha;
  clip pict to border;
  draw pict;
endgroup
enddef;
```

Abbildung 7.8.2: Das METAPOST–Makro zur Konstruktion einer graduierten Fläche

(a) (b) (c)

Abbildung 7.8.3: Alternativen zum Zeichnen von Farbverläufen in Pfaden. Die Teilpfade werden mit Clipping an Kreisen erzeugt. (a) zeigt, wie sich ein graduierter Farbverlauf zusammensetzt; (b) zeigt das Ergebnis eines solchen Vorgehens bei dicken Strichstärken; (c) zeigt die korrekte Graduierung bei dicken Strichstärken.

1. Eine Möglichkeit, die vielleicht als erste einfällt, ist das Bild aus einzelnen Punktzeichenbefehlen zusammenzusetzen. Im METAPOST–Quelltext stünde dann eine Folge von `draw (x,y);` Befehlen; für jeden Bildpunkt ein Befehl. Das läßt die Größe des METAPOST–Quelltextes explodieren. Damit ist dieser Ansatz vollständig unpraktikabel.

2. Die Größe des METAPOST–Quelltextes kann man stark begrenzen, indem jeweils eine ganze Zeile des Bildes hexadezimal kodiert an ein Makro übergeben wird, welches die Punkte erzeugt. Ein Befehl wie `bitline(1.2,"003B00");` erzeugt fünf Punkte[7], wobei der erste Parameter die y–Koordinate festlegt. Für den Speicherplatz ergibt sich die Abschätzung von höschstens $\frac{1}{4}$ Byte pro Bildpunkt. Wenn man bedenkt, daß bei einer nicht hexadezimal kodierten, unkomprimierten Speicherung, ein Bildpunkt $\frac{1}{8}$ Byte beansprucht, ist eine Verdoppelung des Speicherplatzes noch zu vertreten.

   Nachdem die Kodierung auf Seiten von METAPOST gelöst ist, betrachten wir den generierten PostScript–Code. Hier sieht die Sache noch genauso schlimm wie in der ersten Lösung aus. Jeder Bildpunkt muß mit einem Zeichenbefehl inklusive Angabe der Koordinaten gezeichnet werden. Betrachtet man ein solches Bild am Bildschirm, dauert es sehr sehr lange, bis es dargestellt ist. Ein schnelles Blättern im Dokoment ist unmöglich; nicht zuletzt aufgrund der immensen Dateigröße, aber auch weil das Zeichnen einzelner Punkte in den PostScript–Interpretern nicht besonders optimiert ist.

   Was fehlt, ist die Ausnutzung von Redundanzen. Alle Punkte einer Bildreihe liegen auf einer y–Koordinate. Wenn wir das berücksichtigen könnten und einen Mechanismus nutzten, der auch von den Interpretern optimiert ist, wären wir am Ziel.

3. Es geht nun nicht mehr nur darum, in den METAPOST–Quelltext, sondern auch in den erzeugten PostScript–Code einzugreifen. Allerdings gibt es nur einen einzigen Befehl in METAPOST, `special` genannt, der etwas zum PostScript–Code hinzufügt, aber immer nur am Dateianfang. Wie kann aber dann an einer bestimmten Stelle ein Bild erscheinen? Die Funktionalität von METAPOST, Text in einem bestimmten Zeichensatz an einer bestimmten Stelle auszugeben ist hier von großem Nutzen.

   Wir können die PostScriptfunktion `fshow`, die Text ausgibt, umdefinieren und etwas intelligenter machen. Die Idee dahinter ist folgende: Wenn wir eine Bildzeile ausgeben wollen, zeichnen wir stattdessen Text, der eine Bildzeile codiert und teilen dies der Funktion durch Wahl eines speziellen Zeichensatzes mit. Wenn die neue Funktion zur Ausgabe von Text sieht, daß dieser Zeichensatz gewählt wurde, interpretiert sie die Zeichen als Daten eines `image`–Befehls. Ansonsten ruft sie die alte Funktion zur Textausgabe auf. Abbildung 7.8.5 zeigt diese Funktion, die gleichzeitig noch eine weitere Schwäche von METAPOST ausgleicht: Bilder, die Text enthalten können nicht direkt gedruckt oder am Bildschirm betrachtet werden, da die nötigen Zeichensatzinformationen nur bereitstehen, wenn diese vom Programm dvips erzeugt wurden. Wenn die Funktion `ch-xoff`[8] nicht definiert ist, werden Fehlermeldungen für fehlende Zeichensätze unterdrückt und alle Textausgaben benutzen den normalen Times Zeichensatz von PostScript. Damit lassen sich jetzt auch nicht in DVI–Dateien eingebundene Grafiken drucken, wenn auch nur in einem anderen Zeichensatz.

[7] Die Zahl 3B hat in ihrer Binärdarstellung fünf Einsen.

[8] An Stelle dieser Funktion hätten wir auch eine beliebige andere nehmen können. Wichtig ist dabei, daß nur dvips sie definiert, um zuverlässig festzustellen, ob das Bild in einer DVI–Datei eingebunden ist.

Wenn METAPOSTeine Datei übersetzt, in der Text vorkommt, dann müssen für den entsprechenden Zeichensatz Metrikinformationen bereitstehen. Aus diesem Grund sind wir gezwungen, diese Metrikinformationen zu erzeugen. Wir definieren für alle drei Farbtiefen je einen Zeichensatz, in dem die Größe eines Zeichens einem Bildpunkt entspricht. Wie in Abbildung 7.8.6 zu sehen ist, sind die Zeichen leer. METAPOST benötigt lediglich die Information, wie viel Raum jedes Zeichen einnimmt.

Fassen wir zusammen, was die Lösung erreicht.

- Die Länge des METAPOST–Quelltextes bleibt in erträglichen Grenzen.
- Die Länge des PostScript–Codes ist klein.
- Die Geschwindigkeit der Darstellung ist hoch genug, da pro Bildzeile ein PostScriptbild erzeugt wird.

Bisher haben wir nur eine Lösung für monochrome Bilder beschrieben. Das Vorgehen ist bei Graustufen– und Echtfarbgrafiken analog, bis auf den Unterschied, daß ein Zeichen einen einzigen Bildpunkt, bzw. drei Zeichen einen Bildpunkt repräsentieren. Abbildung 7.8.4 zeigt ein Makro, das eine hexadezimal kodierte Bildzeile als Text in speziellen Zeichnsätzen zeichnet.

```
vardef bitline(expr x, y, d)(text s) =
 save i,ha,hb,h;
 string line;

 line := "";
 for i=0 upto (length s-1) div 2:
   ha   := hex(substring(i*2,  i*2+1) of s);
   hb   := hex(substring(i*2+1,i*2+2) of s);
   h    := ha*16 + hb;
   line := line & char h;
 endfor;
 if d=24:      draw (line infont "fmp24") shifted (x, y);
 else: if d=8: draw (line infont "fmp8")  shifted (x, y);
       else:   draw (line infont "fmp1")  shifted (x, y);
       fi;
 fi;
enddef;
```

Abbildung 7.8.4: Dieses METAPOST–Makro liest eine hexadezimal kodierte Bildzeile und zeichnet sie beginnend bei Position (x,y) unter Verwendung des Zeichensatzes aus Abbildung 7.8.5.

```
/ch-xoff where {pop} {errordict begin /undefined {} bind def end} ifelse
/fmp1 /fmp1 def /fmp8 /fmp8 def /fmp24 /fmp24 def
/bitline1 {gsave pop dup scale dup length 8 mul 1 true currentpoint translate
           [66.66666 0 0 66.66666 0 0] 4 index imagemask pop grestore} bind def
/bitline8 {gsave pop dup scale dup length 1 8 currentpoint translate
           [8.333333 0 0 8.333333 0 0] 4 index image pop grestore} bind def
/bitline24 {gsave pop dup scale dup length 3 idiv 1 8 currentpoint translate
            [2.777777 0 0 2.777777 0 0] 4 index false 3 colorimage pop grestore}
           bind def
/XDVIfshow  {findfont exch scalefont setfont show} bind def
/DVIPSfshow {pop exch gsave 72 TeXDict /Resolution get div -72 TeXDict
             /VResolution get div scale 1 DVImag div dup scale get cvx exec
             show grestore} bind def
/fshowText {/ch-xoff where {DVIPSfshow} {XDVIfshow} ifelse} def
/fshow {exch dup /fmp1 eq {bitline1}
                          {dup /fmp8 eq {bitline8}
                                        {dup /fmp24 eq {bitline24}
                                                       {fshowText} ifelse}
                                        ifelse}
                          ifelse} def
```

Abbildung 7.8.5: Der PostScipt–Code, der die Funktion zur Anzeige von Texten umdefiniert.

```
mode_setup; font_size 8bp#;
font_identifier:="fmp1";
font_coding_scheme:="UNSPECIFIED";

u#=0.12pt# ; cap_height#=0.96pt# ;
define_pixels (u) ;

for i=0 upto 255:
  beginchar( i ,cap_height#, u#, 0) ; endchar ;
endfor;

fontmaking:=1;
end
```

Abbildung 7.8.6: Diese METAFONT–Datei definiert die Zeichenmetrik

Kapitel 8

Resümee

8.1 Ergebnisse

Das Ziel dieser Arbeit bestand darin, eine Sprache zur Bildbeschreibung zu entwerfen, die in einer funktionalen Programmiersprache eingebettet ist und die eine Beschreibung eines Bildes mit Hilfe von Gleichungssystemen ermöglicht. Um nicht eine der vielen Bildbeschreibungssprachen mit speziellem und begrentem Anwendungsgebiet zu erhalten, definierten wir eine universelle Kernsprache. Mit den Mitteln dieser Kernsprache formulierten wir dann Erweiterungen, um Möglichkeiten für die Bildbeschreibung mit verschiedenen Grafikparadigmen zu erhalten. Der Anwender hat die Möglichkeit, leicht eigene Erweiterungen für die Lösung seiner Probleme zu entwerfen. Dabei kann er in Haskell besonders einfach Abstraktionen bilden und die Ausdrücke zur Bildbeschreibung auch zu Berechnungen nutzen, da Datentypen und Funktionen in funktionalen Sprachen gleichwertig sind.

Mit der Erweiterung für Bäume haben wir gezeigt, wie Leistungsfähig eine Bildbeschreibung mit Hilfe von Gleichungssystemen ist. Die Kombination einer in Haskell eingebetteten Sprache, mit der Beschreibung durch Gleichungen hat sich damit als sehr fruchtbar erwiesen.

Vorteilhaft ist weiterhin das Konzept, Objekte mit voreingestellten Attributen zu versehen, die meistens den Bedürfnissen des Anwenders entsprechen und nur selten verändert werden müssen. Ein nachträgliche Erweiterung dieser Attributmengen würde bestehende Bildbeschreibungen in keiner Weise beeinflussen. Um die neuen Attribute zu benutzen, können weitere Typklassen mit neuen Attributierungsfunktionen Verwendung finden. Andere Konzepte, etwa die Steuerung von Attributen über Funktionsparameter, böten den Vorteil der leichten Erweiterbarkeit nicht, wenn etwa Zeichenbefehle um zusätzliche Möglichkeiten ergänzt werden sollen, müssen deshalb neue Zeichenfunktionen mit zusätzlichen Parametern hinzukommen.

Auch die Idee der Unterklassen und Abstraktionen, die sich mit Hilfe der Funktion *toPicture* bei Bedarf in ein Bild verwandeln, ermöglicht eine natürliche Art der Bildbeschreibung, da wir die Untertypen wie ein Bild betrachten können.

Wir haben als Ergebnis dieser Arbeit ein leistungsfähiges, vielseitig verwend– und erweiterbares Werkzeug zur Bildbeschreibung erhalten. Die Sprache *functional* METAPOST hat dies schon in der Praxis unter Beweis gestellt [Hin99]. Auch alle Bilder dieser Arbeit, ab Kapitel 2 und außer Abbildung 8.5.1, wurden mit *functional* METAPOST erstellt.

8.2 Die Entwicklungsgeschichte von *functional* METAPOST

Den Anfang der Entwicklung bildete eine Implementierung RADACKs Layoutalgorithmus für Bäume. In dieser frühen Version hat *functional* METAPOST schon mit symbolischen Ausdrücken gerechnet, konnte aber nur Bild von Bäumen generieren. Im nächsten wurde die Spache um Operatoren zur horizontalen und vertikalten Anordnung zweier Bilder ergänzt, dann folgte die Möglichkeit, Bilder rechteckig einzurahmen. Zu diesem Zeitpunkt entstand die Idee der Attributierungen. Die Sprache wuchs weiter um die restlichen Befehle und es wurde deutlich, wie wichtig es ist, alle METAPOST–Befehle in die richtige Reihenfolge zu bringen, um bestimmte Rechenoperationen mit noch unbekannten Größen, die zu Fehlern führen, zu vermeiden. Die Klasse *IsPicture* entstand um Untertypen von Bildern zu simulieren. im folgenden konnten Bilder mit Namen versehen werden und eine erste Verwaltung dieser Namen entstand; von Anfang an hierarchisch organisiert. Vor allem die Einführung von affinen Transformationen stellte neue Anforderungen an die Codegenerierung. Design– und Layoutphase mußten stellenweise gemischt werden. Jetzt wurde die *overlay*–Funktion eingeführt , um die bisher speziellen Kombinatoren zu verallgemeinern. Neue Variablen konnten in den Gleichungssystemen definiert werden. Die Funktion *define* folgte später.

Damit war *functional* METAPOST mächtig genug, um den Layoutalgorithmus für Bäume, mit den eigenen Sprachmitteln, als Erweiterung zu definieren. Die Erweiterungen für Canvas– und Turtlegrafik folgten. Am Ende wichen auch die vier bisher festen Rahmentypen *box*, *circle*, *oval* und *triangle* einem allgemeinen Konzept.

Die gesamte Sprache ist mit der Zeit stetig leistungsfähiger und universeller geworden. Viel Arbeit ist in das Design der Beschreibungssprache geflossen, um die etwa 300 Funktionen und Ausdrücke, die der Anwender von *functional* METAPOST zur Verfügung hat, auszuwählen. An den meisten Stellen war dies ein sehr evolutionärer Prozeß, da stets viele Beispielgrafiken erstellt wurden und dabei gesammelte Erfahrungen die Sprache wieder beeinflußten.

8.3 Betrachtungen zur Laufzeit

Von Beginn an sind in die Entwicklung von *functional* METAPOST immer wieder die Ergebnisse von Benchmarks eingeflossen. So ist das Programm mit der Zeit nicht nur immer leistungsfähiger, sondern auch immer etwas schneller geworden. Bei der Bewertung der Effizienz unseres Compilers ist das Verhältnis der Rechenzeit zur Berechnung des METAPOST–Quellcodes zu der Zeit, die METAPOST benötigt, um daraus eine PostScript–Grafik zu erstellen, besonders interessant. Führt man Benchmarks auf einer Mischung von vielen kleinen und einigen sehr komplexen Bildbeschriebungen aus, zeigt sich daß *functional* METAPOST etwa drei mal soviel Rechenzeit verbraucht, wie METAPOST, siehe Abbildung 8.3.1. Dabei schneidet *functional* METAPOST bei komplexen Grafiken wesentlich besser ab. Für das Bild aus Abbildung 4.3.1 verbrauchen *functional* METAPOST und METAPOST etwa gleich viel Rechenzeit, nämlich etwa 45 Sekunden auf einem Rechner mit 166 Mhz. Das ist etwas verwun-

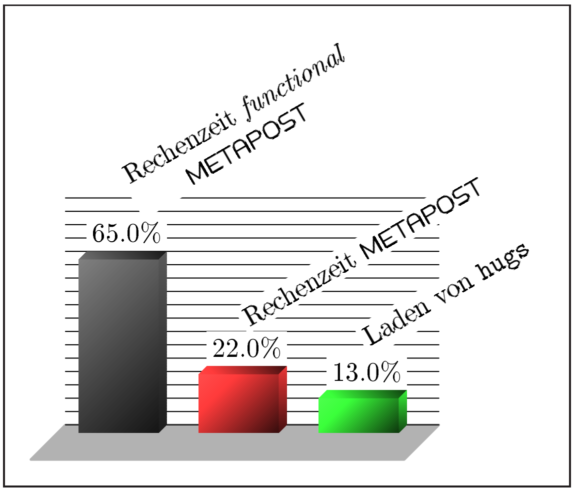


Abbildung 8.3.1: Die Verhältnisse der Laufzeiten.

derlich, denn METAPOST implementiert seine Variablenverwaltung mit Hash–Tabellen, die ausreichend dimensioniert sind. Auf der anderen Seite ist es ein gutes Zeichen für das Laufzeitverhalten von *functional* METAPOST.

8.4 Wünsche an die Sprache Haskell

Haskell eignet sich gut für den Bau von Compilern. Eine Implementierung dieser Arbeit hätte in einer imperativen Sprache wie C++ schätzungsweise vier– bis fünfmal so viel Programmcode erfordert. Trotzdem gibt es noch Punkte, die man verbessern könnte und die diese Arbeit an einigen Stellen vereinfacht hätten.

Der Interpreter Hugs erlaubt leider keine wechselseitigen Abhängigkeiten zwischen Funktionen verschiedener Module (mutually recursive modules). Das hat den Aufbau von *functional* METAPOST etwas beeinflußt, da z.B. das Modul FMPPicture ohne diese Einschränkung, in kleinere logische Einheiten aufgeteilt werden könnte.

Sehr nützlich wären auch mehrparametrige Typklassen [JJM97] (multi parameter type classes), mit denen wir einige Funktionen zur Attributierung allgemeiner hätten gestalten können. Z.B. eine Typklasse für Objekte, die sich verschieden ausrichten lassen:

class *HasAlign a b* **where**
  setAlign :: $a \to b \to b$
  getAlign :: $b \to a$

Für Ausdrücke, deren Typ Instanz dieser Klasse ist, können wir jetzt mit der Funktion *setAlign* ein Attribut, das das Layout beeinflußt setzen und abfragen, wobei der Typ des Attributs unterschiedlich sein kann. Instanzen wären für Baumknoten und Elemente einer Matrix sinnvoll.

instance *HasAlign AlignSons Tree* **where**..
instance *HasAlign Dir Cell* **where**..

Die Einschränkungen in Instanzdeklarationen bzgl. Typparameter sollten weniger restriktiv definiert sein. Wenn Instanzen nicht nur für einfache Typen sondern ebenfalls für zusammengesetzte, wie $[Char]$ erlaubt wären, könnten wir uns etwas Aufwand aus Abschnitt 7.2.2, Seite 97, sparen und die Instanz direkt definieren.

instance *IsName* $[Char]$ **where**..

Aber wir könnten das Prinzip auf etwas übertreiben, indem wir z.B. die Instanzen *HasColor Path*, *HasPen Path* und *HasPattern Path* durch *HasAttribute Color Path*, *HasAttribute Pen Path* und *HasAttribute Pattern Path* ersetzen würden. Die Klasse

class *HasAttribute a b* **where**
  set :: $a \to b \to b$
  get :: $b \to a$

ist jetzt zu allgemein gefaßt. Haskell jetzt kann nur noch anhand der Typen entscheiden, welche Instanz gemeint ist. Je mehr dies der Fall ist, desto mehr Typsignaturen sind notwendig. Um z.B. die Farbe eines Pfades auf grau zu setzen, muß im Ausdruck $p \# set\ (0.5 :: Color)$ die Typsignatur für die Farbe angegeben werden.

Bei der lebhaften Weiterentwicklung des Haskell Sprachstandards und der Entwicklungswerkzeuge ist anzunehmen, daß alle Wünsche in naher Zukunft umgesetzt sind.

8.5 Mögliche Erweiterungen von *functional* METAPOST

Die Möglichkeit Gleichungssysteme anzugeben, die dann von METAPOST bei der Generierung der Grafik gelöst werden, bringt — wie wir in Kapitel 6 gesehen haben — die Notwendigkeit einer bestimmten Reihenfolge der Gleichungen. Das weckt den Wunsch nach einer automatischen Abhängigkeitsanalyse mit anschließender topologischen Sortierung der Gleichungen um Abhängigkeiten zu entflechten.

Eine automatische Umordnung von Gleichungen muß nicht unbedingt Vorteile bringen: Problematisch ist die Tatsache, daß weiterhin zyklische Abhängigkeiten möglich sind. Wenn der Anwender ein Gleichungssystem mit zyklischen Abhängigkeiten aufgestellt hat, ist dies kein Problem; er kann eine Fehlermeldung erhalten. Aber wenn in einer Erweiterung große Gleichungssysteme automatisch generiert werden sollen, wie wir dies für die Baumlayouts gemacht haben, können solche zyklischen Abhängigkeiten plötzlich für einen Sonderfall auftreten. Damit dies nicht passiert, muß der Programmierer der Erweiterung entsprechende Vorkehrungen treffen. Die sicherste Methode ist ein konstruktiver Beweis, der eine Reihenfolge, die der topologischen Sortierung der Abhängigkeiten entspricht, liefert. In diesem Fall können die Gleichungen auch direkt in der richtigen Reihenfolge notiert werden und die Rechenzeit zur erneuten topologischen Sortierung wäre verschwendet.

Eine radikale Möglichkeit alle diese Probleme zu lösen, wäre die Neuimplementierung von METAPOST in Haskell, eventuell mit einem mächtigeren Gleichungslöser, der auch Ungleichungen beherrscht. Allerdings sind dafür, optimistisch geschätzt, zwei bis drei Mannjahre anzusetzen, da von der PostScript–Generierung bis zur komplizierten Wahl der Kontrollpunkte von Pfaden alles nachprogrammiert werden müßte.

Eventuell ließe sich eine effizientere Datenstruktur für die Variablenverwaltung entwickeln. Allerdings sind dabei so viele Rahmenbedingungen einzuhalten, daß der kreative Spielraum gering ausfällt.

Eine Ergänzung, die das Arbeiten mit *functional* METAPOST noch weiter vereinfachen kann, ist die Verwaltung von Fehlermeldungen. Bisher werden Bildelemente, wenn sie eine unbekannte Variable enthalten, einfach nicht gezeichnet. Es ist denkbar, in einer Logdatei solche Fehler zu protokollieren.

Schließlich ist ein weiteres Attribut für Pfade geplant, aber noch nicht endgültig realisiert und getestet, mit dem Wellenformen und Zickzackmuster möglich sind. Solche deformierten Pfade sind in der Darstellung von FEYNMAN–Diagrammen [Ohl96] besonders verbreitet. Für derartige Effekte berechnet ein Makro zusätzliche Punkte links und rechts neben dem angegebenen Pfad und legt durch diese Punkte eine neue Kurve. Die Schwierigkeit, ist die Wahl der Frequenz, so daß sich die Amplitude an den Enden im Nullpunkt befindet. Ansonsten sehen Pfeilspitzen, die sich neben dem Pfad befinden oder nicht in die richtige Richtung zeigen, nicht besonders hübsch aus. Hierfür müßte die Länge des Pfades genau zu ermitteln sein, was aber bei kurvigen Pfaden nicht richtig funktioniert.

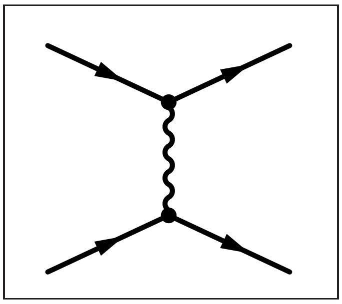


Abbildung 8.5.1: Ein Feynman–Diagramm.

Die Entwickler des Interpreters ☞ *Hugs* und des Compilers ☞ *GHC* haben angekündigt, eine gemeinsame Laufzeitumgebung für beide Systeme zu implementieren. Ein Programm kann dann teilweise aus interpretiertem oder compiliertem Code bestehen. Wenn *functional* METAPOST in den zeitkritischen Modulen compiliert ist, sind Geschwindigkeitssteigerungen vom Faktor fünf bis acht zu erwarten.

Anhang A

Weitere Befehle von *functional* METAPOST

Dieses Kapitel ergänzt die Einführung in *functional* METAPOST aus Kapitel 3, wo aus didaktischen Gründen nur die wichtigsten Möglichkeiten erwähnt sind, zu einer vollständigen Vorstellung von *functional* METAPOST. Am Anfang jedes Abschnittes werden die, zu den thematisch passenden Typen, definierten Instanzen genannt. Darauf folgen die wichtigsten exportierten Datentypen und die fehlenden Funktionen. Wir verwenden eine weitgehend aufzählende Darstellungsweise. Bei interessanten und wichtigen Aspekten geben wir aber eine kurze Beschreibung oder ein Beispiel.
Die Reihenfolge der einzelnen Abschnitte korrespondiert weitgehend zu der von Kapitel 3.

A.1 Atomare Bilder

Picture ist Instanz von: *IsPicture*, *HasColor*, *HasBGColor*, *HasName* und *HasDefine*.

Instanzen von *IsPicture* sind: *Picture*, *Char*, *Int*, *Integer*, *Numeric*, $IsPicture\ a \Rightarrow IsPicture\ [a]$, $()$, $(IsPicture\ a, IsPicture\ b) \Rightarrow IsPicture\ (a,b)$, $(IsPicture\ a, IsPicture\ b, IsPicture\ c) \Rightarrow IsPicture\ (a,b,c)$, *Path*, *Area*, *Frame*, *Tree*, *Canvas*, *Turtle*, …

$$
\begin{array}{lcl}
\textbf{data}\ Picture & = & Attributes\ Attrib\ Picture \\
 & | & Overlay\ [Equation]\ (Maybe\ Int)\ [Picture] \\
 & | & Define\ [Equation]\ Picture \\
 & | & Frame\ FrameAttrib\ [Equation]\ Path\ Picture \\
 & | & Draw\ [Path]\ Picture \\
 & | & Fill\ [Area]\ Picture \\
 & | & Clip\ Path\ Picture \\
 & | & Empty\ Numeric\ Numeric \\
 & | & Tex\ String \\
 & | & Text\ String \\
 & | & BitLine\ Point\ BitDepth\ String \\
 & | & PTransform\ Transformation\ Picture \\
 & | & TrueBox\ Picture \\
 & & \textbf{deriving}\ (Eq, Show, Read)
\end{array}
$$

$$
\begin{array}{lll}
\textbf{class } \mathit{HasPicture}\ a \textbf{ where} & & \\
\quad \mathit{fromPicture} & :: & (\mathit{IsPicture}\ b) \Rightarrow b \to a
\end{array}
$$

Es existieren noch mehr Konstanten für das Rechnen in Längeneinheiten, nämlich Didôt–Punkt, Big Point (PostScript Punkt), Pica, Cicero, und Inch.

$$
\begin{array}{lll}
dd, bp, pc, cc, inch & :: & \mathit{Numeric} \\
dd & = & 1.06601 \\
bp & = & 1 \\
pc & = & 11.95517 \\
cc & = & 12.79213 \\
inch & = & 72
\end{array}
$$

A.2 Rahmen

Frame ist Instanz von: *IsPicture*, *HasColor*, *HasBGColor*, *HasPen*, *HasPattern*, *HasShadow*, *HasDXY*, *HasExtent*, *HasName* und *IsHideable*.

$$
\begin{array}{lll}
\textbf{data } \mathit{Frame} & = & \mathit{Frame}'\ \mathit{FrameAttrib}\ \mathit{ExtentAttrib}\ \mathit{Picture} \\
 & & \textbf{deriving } \mathit{Show}
\end{array}
$$

$$
\begin{array}{lll}
\textbf{class } \mathit{HasDXY}\ a \textbf{ where} & & \\
\quad \mathit{setDX} & :: & \mathit{Numeric} \to a \to a \\
\quad \mathit{getDX} & :: & a \to \mathit{Maybe}\ \mathit{Numeric} \\
\quad \mathit{setDY} & :: & \mathit{Numeric} \to a \to a \\
\quad \mathit{getDY} & :: & a \to \mathit{Maybe}\ \mathit{Numeric}
\end{array}
$$

$$
\begin{array}{lll}
\textbf{class } \mathit{HasExtent}\ a \textbf{ where} & & \\
\quad \mathit{setWidth} & :: & \mathit{Numeric} \to a \to a \\
\quad \mathit{removeWidth} & :: & a \to a \\
\quad \mathit{getWidth} & :: & a \to \mathit{Maybe}\ \mathit{Numeric} \\
\quad \mathit{setHeight} & :: & \mathit{Numeric} \to a \to a \\
\quad \mathit{removeHeight} & :: & a \to a \\
\quad \mathit{getHeight} & :: & a \to \mathit{Maybe}\ \mathit{Numeric}
\end{array}
$$

$$
\begin{array}{lll}
\textbf{class } \mathit{HasShadow}\ a \textbf{ where} & & \\
\quad \mathit{setShadow} & :: & (\mathit{Numeric}, \mathit{Numeric}) \to a \to a \\
\quad \mathit{clearShadow} & :: & a \to a \\
\quad \mathit{getShadow} & :: & a \to \mathit{Maybe}\ (\mathit{Numeric}, \mathit{Numeric})
\end{array}
$$

$$
\begin{array}{lll}
\textbf{class } \mathit{HasExtent}\ a \textbf{ where} & & \\
\quad \mathit{setWidth} & :: & \mathit{Numeric} \to a \to a \\
\quad \mathit{removeWidth} & :: & a \to a \\
\quad \mathit{getWidth} & :: & a \to \mathit{Maybe}\ \mathit{Numeric} \\
\quad \mathit{setHeight} & :: & \mathit{Numeric} \to a \to a \\
\quad \mathit{removeHeight} & :: & a \to a \\
\quad \mathit{getHeight} & :: & a \to \mathit{Maybe}\ \mathit{Numeric}
\end{array}
$$

$$
\begin{array}{lll}
\textbf{class } \mathit{IsHideable}\ a \textbf{ where} & & \\
\quad \mathit{hide} & :: & a \to a
\end{array}
$$

Zusätzlich zu den bisher besprochenen Rahmen gibt es noch einen Rahmen für Dreiecke, bei denen der Winkel der Spitze wählbar ist, und einen um 45 Grad gedrehten quadratischen Rahmen.

$$\mathit{triAngle} :: \mathit{IsPicture}\ a \Rightarrow \mathit{Numeric} \rightarrow a \rightarrow \mathit{Frame}$$
$$\mathit{diamond} :: \mathit{IsPicture}\ a \Rightarrow a \rightarrow \mathit{Frame}$$

Weil der Rahmenpfad völlig frei wählbar ist, sind auch etwas exotischere Rahmentypen möglich. Ein Bild kann z.B. flockig umrahmt werden, wobei zwei Parameter Anfangswerte eines Zufallsgenerators festlegen. Die Form ist auf diese Weise variabel, der Anwender muß sie aber nicht genau festlegen.

$$\mathit{fuzzy} :: \mathit{IsPicture}\ a \Rightarrow \mathit{Int} \rightarrow \mathit{Int} \rightarrow a \rightarrow \mathit{Frame}$$

Im Bereich Datenbanken werden oft kleine Tonnen benötigt. Der Rahmen *drum* stellt solche Tonnen zur Verfügung.

$$\mathit{drum} :: \mathit{IsPicture}\ a \Rightarrow a \rightarrow \mathit{Frame}$$

Natürlich sind auch Schatten und Clipping an solchen Rahmen möglich, da sie nicht grundsätzlich anders implementiert sind, als ein rechteckiger Rahmen. Die Abbildung A.2.1 zeigt die vier weiteren Rahmentypen.

```
rowSepBy cm [triAngle 90 (tex "\\texttt{triAngle␣90}") # setDX 16,
             diamond (tex "\\texttt{diamond}"),
             fuzzy 1 2 (tex "\\texttt{fuzzy␣1␣2}"),
             drum (tex "\\texttt{drum}")]
```

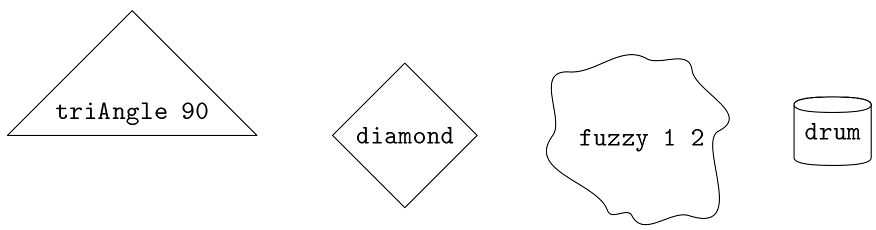


Abbildung A.2.1: Vier weitere Rahmentypen.

A.3 Kombination von Bildern

Der Befehl *matrix* ist nur ein Spezialfall der allgemeineren Funktion *matrixAlignSepBy*, die es erlaubt, die einzelnen Felder der Matrix an den Rändern auszurichten, siehe Abbildung A.3.1.

$$\begin{array}{lll} \textbf{data}\ \mathit{Cell} & = & \mathit{Cell}\ \mathit{Dir}\ \mathit{Picture} \\ & & \quad\textbf{deriving}\ \mathit{Show} \\ \\ \mathit{cell} & :: & \mathit{IsPicture}\ a \Rightarrow a \rightarrow \mathit{Cell} \\ \mathit{cell} & = & \mathit{cell}'\ C \end{array}$$

```
matrixAlignSepBy 10 10 [[cell' C (math "\\setminus"), cell' W "linksb\\\"undig",
                         cell' W "mittig", cell' W "rechtsb\\\"undig"],
                        [cell' W ("vertikal" ⊟ "oben"),
                         cell' NW "NW", cell' N "N", cell' NE "NE"],
                        [cell' W ("vertikal" ⊟ "mittig"),
                         cell' W "W", cell' C "C", cell' E "E"],
                        [cell' W ("vertikal" ⊟ "unten"),
                         cell' SW "SW", cell' S "S", cell' SE "SE"]]
```

\	linksbündig	mittig	rechtsbündig
vertikal oben	NW	N	NE
vertikal mittig	W	C	E
vertikal unten	SW	S	SE

Abbildung A.3.1: Die Felder einer Matrix können einzeln ausgerichtete werden. Zur Verdeutlichung der Zwischenräume sind diese grau eingezeichnet.

```
cell'              :: IsPicture a ⇒ Dir → a → Cell
matrixSepBy        :: IsPicture a ⇒ Numeric → Numeric → [[a]] → Picture
matrixAlign        :: [[Cell]] → Picture
matrixAlignSepBy   :: Numeric → Numeric → [[Cell]] → Picture
rowAlign           :: [Cell] → Picture
columnAlign        :: [Cell] → Picture
rowAlignSepBy      :: Numeric → [Cell] → Picture
columnAlignSepBy   :: Numeric → [Cell] → Picture
```

```
at                 :: (IsPicture a, IsPicture b)
label              :: (IsPicture a, IsPicture b) ⇒ Dir → a → b → Picture
ooalign            :: IsPicture a ⇒ [a] → Picture       -- siehe LaTeX
```

A.4 Pfade

Instanzen von *IsPath*: *Path*, *Point*, *Name*, *IsPath a* ⇒ *IsPath* [*a*], *Char*, (*Num a*, *Num b*, *Real a*, *Real b*) ⇒ *IsPath* (*a*, *b*).

Path ist Instanz der Klassen: *IsPicture*, *IsPath*, *HasLabel*, *HasConcat*, *HasColor*, *HasPattern*, *HasPen*, *IsHideable*, *HasArrowHead*, *HasStartEndDir*, *HasJoin*, *HasStartEndCut* und *HasDefine*.

$$
\begin{array}{lll}
\textbf{class } \mathit{HasLabel}\ a\ \textbf{where} & & \\
\quad \mathit{setLabel} & :: & \mathit{IsPicture}\ b \Rightarrow \mathit{Double} \rightarrow \mathit{Dir} \rightarrow b \rightarrow a \rightarrow a \\
\quad \mathit{removeLabel} & :: & a \rightarrow a
\end{array}
$$

$$
\begin{array}{lll}
\textbf{class } \mathit{HasConcat}\ a\ \textbf{where} & & \\
\quad (\&) & :: & a \rightarrow a \rightarrow a
\end{array}
$$

Einzelne Pfadsegmente lassen sich ausblenden, wenn die Funktion *hide* auf sie angewendet wird.

$$
\begin{array}{lll}
\textbf{class } \mathit{IsHideable}\ a\ \textbf{where} & & \\
\quad \mathit{hide} & :: & a \rightarrow a
\end{array}
$$

Die Stützpunkte von Kurven können am Anfang und Ende eines Pfadsegments vielfältig beeinflußt werden. Ein höherer Wert von curl als eins bewirkt eine stärkere Krümmung als normal, ein Wert kleiner eins eine schwächere. Siehe zu *setEndCurl* auch Abbildung A.14.1.

$$
\begin{array}{lll}
\textbf{class } \mathit{HasStartEndDir}\ a\ \textbf{where} & & \\
\quad \mathit{setStartAngle} & :: & \mathit{Numeric} \rightarrow a \rightarrow a \\
\quad \mathit{setEndAngle} & :: & \mathit{Numeric} \rightarrow a \rightarrow a \\
\quad \mathit{setStartCurl} & :: & \mathit{Numeric} \rightarrow a \rightarrow a \\
\quad \mathit{setEndCurl} & :: & \mathit{Numeric} \rightarrow a \rightarrow a \\
\quad \mathit{setStartVector} & :: & \mathit{Point} \rightarrow a \rightarrow a \\
\quad \mathit{setEndVector} & :: & \mathit{Point} \rightarrow a \rightarrow a \\
\quad \mathit{removeStartDir} & :: & a \rightarrow a \\
\quad \mathit{removeEndDir} & :: & a \rightarrow a
\end{array}
$$

$$
\begin{array}{lll}
\textbf{class } \mathit{HasJoin}\ a\ \textbf{where} & & \\
\quad \mathit{setJoin} & :: & \mathit{BasicJoin} \rightarrow a \rightarrow a \\
\quad \mathit{getJoin} & :: & a \rightarrow \mathit{BasicJoin}
\end{array}
$$

Pfadsegmente können an ihrem Anfang und Ende an beliebigen Bounding Boxen von Bildern abgeschnitten werden. Wenn die Pfadkonstruktoren Bezugspunkte verbinden, werden automatisch die Funktionen *setStartCut* bzw. *setEndStartCut* aufgerufen, um den Pfad an den Bounding Boxen des Bildes, dem der Bezugspunkt zugeordnet ist, abzuschneiden. Wenn diese Funktion nicht gewünscht ist, weil des Pfadsegment über das Bild verlaufen soll, können die Funktionen *removeStartCut* bzw. *removeEndCut* dies bewirken.
Der Anfang des Pfadsegments $\mathit{ref}\ (n_1 \triangleleft C)$ -- $(\mathit{ref}\ (n_2 \triangleleft C) + \mathit{vec}\ (4,0))$ wird z.B. an der Bounding Box des Bildes mit Namen n_1 abgeschnitten, aber das Ende nicht am Bild n_2. Um dies zu erreichen muß die Attributierungsfunktion *setEndCut* n_2 auf das Pfadsegment angewendet werden. Wenn wir die Kanten der Bäume aus Abbildung 4.4.1 nicht bis zu den Punkten durchzeichnen wollen, genügt es, als Definition der Kanten die Funktion

$$
\begin{array}{lll}
\mathit{edgeN} & :: & \mathit{Int} \rightarrow \mathit{Tree} \rightarrow \mathit{Edge} \\
\mathit{edgeN}\ n & = & \mathit{edge}'\ (\mathit{ref}\ (\mathit{This} \triangleleft C)\ \text{--}\ \mathit{ref}\ (\mathit{Parent} \triangleleft \texttt{'p'} : \mathit{show}\ n \triangleleft C) \\
 & & \qquad \#\ \mathit{setEndCut}\ \mathit{Parent})
\end{array}
$$

zu verwenden. Abbildung A.4.1 zeigt das Ergebnis dieser Variante.

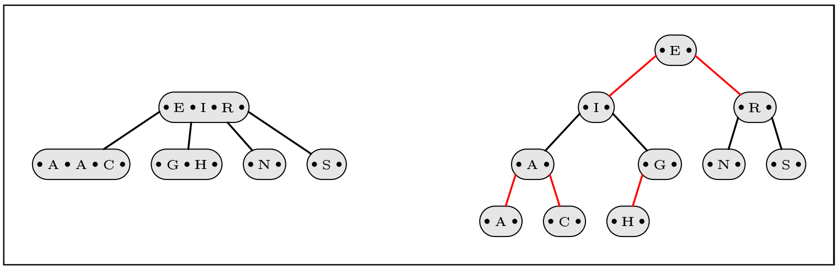


Abbildung A.4.1: Eine Variante von Abbildung 4.4.1.

```
class HasStartEndCut a where
      setStartCut          :: IsName b ⇒ b → a → a
      removeStartCut       :: a → a
      setEndCut            :: IsName b ⇒ b → a → a
      removeEndCut         :: a → a

data Path                  =  PathBuildCycle Path Path
                           |  PathTransform Transformation Path
                           |  PathPoint Point
                           |  PathCycle
                           |  PathJoin Path PathElemDescr Path
                           |  PathEndDir Point Dir'
                           |  PathDefine [Equation] Path
                              deriving (Eq, Show, Read)

data Dir'                  =  DirEmpty
                           |  DirCurl Numeric
                           |  DirDir Numeric
                           |  DirVector Point
                              deriving (Eq, Show, Read)


data PathElemDescr         =  PathElemDescr{
                                      peColor :: Color,
                                      pePen :: Pen,
                                      peArrowHead :: Maybe ArrowHead,
                                      peSArrowHead :: Maybe ArrowHead,
                                      pePattern :: Pattern,
                                      peVisible :: Bool,
                                      peStartCut,
                                      peEndCut :: Maybe CutPic,
                                      peStartDir,
                                      peEndDir :: Dir',
                                      peJoin :: BasicJoin,
                                      peLabels :: [PathLabel]}
                              deriving (Eq, Read)
```

$$
\begin{array}{lll}
\mathit{joinCat}, \mathit{joinFree}, \mathit{joinBounded}, & & \\
\mathit{joinStraight}, \mathit{joinTense} & :: & \mathit{BasicJoin} \\
\mathit{joinTension} & :: & \mathit{Tension} \to \mathit{BasicJoin} \\
\mathit{joinTensions} & :: & \mathit{Tension} \to \mathit{Tension} \to \mathit{BasicJoin} \\
\mathit{joinControl} & :: & \mathit{Point} \to \mathit{BasicJoin} \\
\mathit{joinControls} & :: & \mathit{Point} \to \mathit{Point} \to \mathit{BasicJoin}
\end{array}
$$

$$
\begin{array}{lll}
\mathbf{data}\ \mathit{BasicJoin} & = & \mathit{BJCat} \\
 & | & \mathit{BJFree} \\
 & | & \mathit{BJBounded} \\
 & | & \mathit{BJStraight} \\
 & | & \mathit{BJTense} \\
 & | & \mathit{BJTension\ Tension} \\
 & | & \mathit{BJTension2\ Tension\ Tension} \\
 & | & \mathit{BJControls\ Point} \\
 & | & \mathit{BJControls2\ Point\ Point} \\
 & & \mathbf{deriving}\ (\mathit{Eq}, \mathit{Show}, \mathit{Read})
\end{array}
$$

$$
\begin{array}{lll}
\mathit{tension}, \mathit{tensionAtLeast} & :: & \mathit{Numeric} \to \mathit{Tension}
\end{array}
$$

$$
\begin{array}{lll}
\mathbf{data}\ \mathit{Tension} & = & \mathit{Tension\ Numeric} \\
 & | & \mathit{TensionAtLeast\ Numeric} \\
 & & \mathbf{deriving}\ (\mathit{Eq}, \mathit{Show}, \mathit{Read})
\end{array}
$$

Zwei Pfade können geschnitten werden. Zwischen zwei Schnittpunkten entsteht ein zyklischer Pfad, siehe Abbildung A.4.2.

$$
\begin{array}{lll}
\mathit{buildCycle} & :: & (\mathit{IsPath}\ a, \mathit{IsPath}\ b) \Rightarrow a \to b \to \mathit{Path}
\end{array}
$$

$$
\begin{array}{lll}
\mathit{pathLength} & :: & \mathit{Num}\ a \Rightarrow \mathit{Path} \to a \\
\mathit{forEachPath} & :: & (\mathit{PathElemDescr} \to \mathit{PathElemDescr}) \to \mathit{Path} \to \mathit{Path} \\
\mathit{line} & :: & (\mathit{IsPath}\ a, \mathit{IsPath}\ b) \Rightarrow a \to b \to \mathit{Path} \\
\mathit{curve} & :: & (\mathit{IsPath}\ a, \mathit{IsPath}\ b) \Rightarrow a \to b \to \mathit{Path} \\
\mathit{arrow} & :: & (\mathit{IsPath}\ b, \mathit{IsPath}\ a) \Rightarrow a \to b \to \mathit{Path}
\end{array}
$$

Transformationen lassen nicht nur auf Bilder, sondern auch auf Pfade anwenden, siehe auch Abbildung A.4.2.

$$
\begin{array}{lll}
\mathit{transformPath} & :: & \mathit{Transformation} \to \mathit{Path} \to \mathit{Path} \\
\mathit{fullcircle}, \mathit{halfcircle}, & & \\
\mathit{quartercircle}, \mathit{unitsquare} & :: & \mathit{Path}
\end{array}
$$

```
bsp5            = box (math "U" ||| ooalign [toPicture [cArea a 0.7,
                                                        cArea b 0.7,
                                                        cArea ab 0.4],
                                             bOverA])
       where
       cArea a c = toArea a # setColor c
       bOverA    = column [math "B" # setBGColor white,
                           vspace 50,
                           math "A" # setBGColor white]
       a         = transformPath (scaled 30) fullcircle
       b         = transformPath (scaled 30 & shifted (0, −30))
                          fullcircle
       ab        = buildCycle a b
```

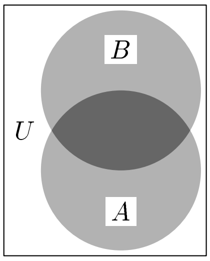


Abbildung A.4.2: Die Funktion *buildCycle* schneidet zwei Pfade. Siehe Abbildung 22 in [Hob92].

A.5 Namen

Instanzen von *IsName*: *Name*, *Int*, *Char*, *Dir* und *IsName a* ⇒ *IsName* [*a*].

Instanzen von *HasName*: *Picture*, *Frame* und *Tree*.

```
class HasName a where
    setName     :: IsName b ⇒ b → a → a
    getNames    :: a → [Name]
```

Wenn eine Liste von n Bildern durchnumeriert werden soll, kann dies mit Hilfe der Funktion *enumPics* geschehen. Das erste Bild erhält den Namen 0 das letzte den Namen $n - 1$.

```
enumPics        :: HasName a ⇒ [a] → [a]
```

In der Abbildung A.5.1 sehen wir ein weiteres Beispiel für die Art, die gewünschten Variablen mit Hilfe hierarchischer Namensgebung zu referenzieren.

```
let pointerChain dx ps = draw (backarrow : chainarrows)
                              (rowSepBy dx [b # setName (i :: Int)
                                           | (b, i) ← zip (map recBox ps) [0 ‥]])
    where
    n = length ps
    backarrow = arrow (ref (n − 1 ⊲ "bullet" ◃ C))
                      (ref (n − 1 ⊲ "bullet" ◃ C) + vec (0, 20))
                      --- arrow (ref (0 ⊲ W) + vec (0, 20)) (ref (0 ⊲ W))
    chainarrows = [arrow (ref (i ⊲ "bullet" ◃ C)) (ref (i + 1 ⊲ W))
                  | i ← [0 ‥ n − 2]]
    recBox a = (box a # setHeight 16)
               ⊞ (box (bullet # setName "bullet") # setHeight 16 # setWidth 16)
in pointerChain 25 ["42", "2", "3", "1109"]
```

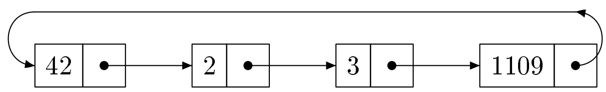


Abbildung A.5.1: Darstellung einer verketteten Feldstruktur

A.6 Zahlen und Punkte

```
data Point            = PointPic′ Int Dir
                      | PointVar′ Int Int
                      | PointVarArray′ Int Int
                      | PointTrans′ Point [Int]
                      | PointVar Name
                      | PointVec (Numeric, Numeric)
                      | PointMediate Numeric Point Point
                      | PointDirection Numeric
                      | PointWhatever
                      | PointPPP FunPPP Point Point
                      | PointNMul Numeric Point
                      | PointNeg Point
                      | PointCond Boolean Point Point
                        deriving (Eq, Show, Read, Ord)

data Numeric          = NumericVar′ Int Int
                      | NumericArray′ Int Int
                      | NumericVar Name
                      | Numeric Double
                      | NumericWhatever
                      | NumericDist Point Point
```

$$
\begin{array}{lll}
 & | & \mathit{NumericMediate\ Numeric\ Numeric\ Numeric} \\
 & | & \mathit{NumericPN\ FunPN\ Point} \\
 & | & \mathit{NumericNN\ FunNN\ Numeric} \\
 & | & \mathit{NumericNNN\ FunNNN\ Numeric\ Numeric} \\
 & | & \mathit{NumericNsN\ FunNsN}\ [\mathit{Numeric}] \\
 & | & \mathit{NumericCond\ Boolean\ Numeric\ Numeric} \\
 & & \textbf{deriving}\ (\mathit{Eq}, \mathit{Show}, \mathit{Read}, \mathit{Ord})
\end{array}
$$

$$
\begin{array}{lll}
\textbf{class}\ \mathit{HasCond}\ a\ \textbf{where} & & \\
\quad \mathit{cond} & :: & \mathit{Boolean} \to a \to a \to a
\end{array}
$$

$$
\begin{array}{lll}
(\dot{*}) & :: & \mathit{Numeric} \to \mathit{Point} \to \mathit{Point}
\end{array}
$$

$$
\begin{array}{lll}
\mathit{boolean} & :: & \mathit{Bool} \to \mathit{Boolean}
\end{array}
$$

Die Konstanten *up*, *right*, *down* und *left* bezeichnen die entsprechenden Einheitsvektoren.

$$
\begin{array}{lll}
\mathit{up}, \mathit{down}, \mathit{left}, \mathit{right} & :: & \mathit{Point} \\
\mathit{up} & = & \mathit{vec}\ (0, 1) \\
\mathit{down} & = & \mathit{vec}\ (0, -1) \\
\mathit{left} & = & \mathit{vec}\ (-1, 0) \\
\mathit{right} & = & \mathit{vec}\ (1, 0)
\end{array}
$$

A.7 Symbolische Gleichungen

Instanzen von *HasDefine*: *Picture*, *Path* und *Area*.

$$
\begin{array}{lll}
\mathit{equations} & :: & [\mathit{Equation}] \to \mathit{Equation} \\
\mathit{width}, \mathit{height} & :: & \mathit{IsName}\ a \Rightarrow a \to \mathit{Numeric}
\end{array}
$$

Soll eine Variable in einem anderen Gleichungssystem gemeint sein, und keine neue lokale Variable erzeugt werden, so kann dem Variablennamen die Funktion *global* vorangestellt werden.

$$
\begin{array}{lll}
\mathit{global} & :: & \mathit{IsName}\ a \Rightarrow a \to \mathit{Name}
\end{array}
$$

Die Funktion *overlay* erzeugt eine Bounding Box, die alle übergebenen Bilder umschließt. Das ist nicht in jeder Situation erwünscht. Wenn z.B. ein Bild mit einem Label versehen wird, kann es sinnvoll sein, die Bounding Box nicht um die des Labels zu vergrößern, sondern zu belassen, damit ein zweites Label nicht plötzlich mit einem Abstand plaziert wird. Deshalb hat die Funktion *overlay'* einen weiteren Parameter. Wenn dieser den Wert *Nothing* hat, erhalten wir die Funktionalität von *overlay*. Wenn der Wert *Just b* ist, ist die resultierende Bounding Box die, des $b + 1$–ten Bildes der Parameterliste.

$$
\begin{array}{lll}
\mathit{overlay}' & :: & \mathit{IsPicture}\ a \Rightarrow [\mathit{Equation}] \to \mathit{Maybe\ Int} \to [a]
\end{array}
$$

$$
\begin{array}{lll}
\mathit{overlay} & :: & \mathit{IsPicture}\ a \Rightarrow [\mathit{Equation}] \to [a] \to \mathit{Picture} \\
\mathit{overlay\ eqs\ ps} & = & \mathit{overlay}'\ \mathit{eqs\ Nothing\ ps}
\end{array}
$$

A.8 Farben

Color ist Instanz von: *Num* und *Fractional*.

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Color} & = & \mathit{DefaultColor} \\
 & \mid & \mathit{Color\ Double\ Double\ Double} \\
 & \mid & \mathit{Graduate\ Color\ Color\ Double\ Int} \\
 & & \textbf{deriving}\ (\mathit{Eq}, \mathit{Show}, \mathit{Read})
\end{array}
$$

$$
\begin{array}{lcl}
\textbf{class}\ \mathit{HasColor}\ a\ \textbf{where} & & \\
\quad \mathit{setColor} & :: & \mathit{Color} \to a \to a \\
\quad \mathit{setDefaultColor} & :: & a \to a \\
\quad \mathit{getColor} & :: & a \to \mathit{Color}
\end{array}
$$

$$
\begin{array}{lcl}
\textbf{class}\ \mathit{HasBGColor}\ a\ \textbf{where} & & \\
\quad \mathit{setBGColor} & :: & \mathit{Color} \to a \to a \\
\quad \mathit{setDefaultBGColor} & :: & a \to a \\
\quad \mathit{getBGColor} & :: & a \to \mathit{Color}
\end{array}
$$

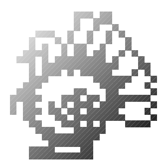
$$
\mathit{hsv2rgb} \quad :: \quad (\mathit{Double}, \mathit{Double}, \mathit{Double}) \to \mathit{Color}
$$

Wir wollen zwei weitere Beispiele für Farbverläufe geben. Im Zusammenhamg mit einem Pfad.

$$
\begin{array}{l}
(30,0)\ ..\ (0,-10)\ ..\ (-40,0)\ ..\ (5,20)\ ..\ \mathit{cycle} \\
\#\mathit{setColor}\ (\mathit{graduateMed\ white\ black}\ 30) \\
\#\mathit{setPen}\ 3
\end{array}
$$

Und ein Farbverlauf, der sich über ein ganzes Bild erstreckt.

let *turkey* = ["003B00", "002700", "002480", "0E4940",
"114920", "14B220", "3CB650", "75FE88",
"17FF8C", "175F14", "1C07E2", "3803C4",
"703182", "F8EDFC", "B2BBC2", "BB6F84",
"31BFC2", "18EA3C", "0E3E00", "07FC00",
"03F800", "1E1800", "1FF800"]
in *scale* 20 (*image Depth1 turkey*) # *setColor* (*graduateMed white black* 45)



$$
\begin{array}{lcl}
\mathit{graduate} & :: & \mathit{Color} \to \mathit{Color} \to \mathit{Double} \to \mathit{Int} \to \mathit{Color} \\
\mathit{graduate}\ c_1\ c_2\ a\ n & = & \mathit{Graduate}\ c_1\ c_2\ a\ n
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{graduateLow} & :: & \mathit{Color} \to \mathit{Color} \to \mathit{Double} \to \mathit{Color} \\
\mathit{graduateLow}\ c_1\ c_2\ a & = & \mathit{graduate}\ c_1\ c_2\ a\ 16
\end{array}
$$

Abbildung A.8.1 zeigt eine Anwendung der Funktion *hsv2rgb*.

$$
\begin{array}{lcl}
\textbf{let}\ \mathit{rad}\ n & = & 60 * (1 \mathbin{/} 1.2) ** n \\
\quad \mathit{pol}\ r\ a & = & r \mathbin{\dot{*}} \mathit{dir}\ (\mathit{fromDouble}\ a) \\
\quad \mathit{color}\ 0 & = & [\,] \\
\quad \mathit{color}\ n & = & \mathit{areas}\ (\mathit{rad}\ n)\ (\lambda i \to \mathit{hsv2rgb}\ (i,1,1)) + \!\! + \ \mathit{bw}\ (n-1) \\
\quad \mathit{bw}\ 0 & = & [\,] \\
\quad \mathit{bw}\ n & = & \mathit{areas}\ (\mathit{rad}\ n)\ (\lambda i \to \mathit{grey}\ (\mathit{abs}\ (i-180) \mathbin{/} 180)) + \!\! + \ \mathit{color}\ (n-1) \\
\quad \mathit{areas} & :: & \mathit{Numeric} \to (\mathit{Double} \to \mathit{Color}) \to [\mathit{Area}] \\
\quad \mathit{areas}\ r\ \mathit{cf} & = & [\mathit{toArea}\ [\mathit{pol}\ r\ i, \mathit{pol}\ r\ (2+i), \\
 & & \qquad\qquad \mathit{pol}\ (1.15 * r)\ (2+i), \mathit{pol}\ (1.15 * r)\ i] \\
 & & \quad \#\ \mathit{setColor}\ (\mathit{cf}\ i)\ \#\ \mathit{setPen}\ 0.1 \\
 & & \mid i \leftarrow [0, 4 \mathinner{..} 356]] \\
\textbf{in}\ \mathit{transform}\ (\mathit{affine}\ (1,0,0.2,0.85,0,0))\ (\mathit{color}\ 9) & &
\end{array}
$$

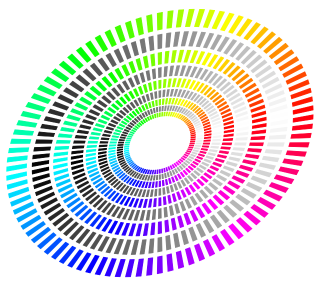


Abbildung A.8.1: Ein Farbkreis, siehe [Ado85].

A.9 Strichmuster

Instanzen von *HasPattern*: *Frame*, *Path* und *PathElemDescr*.

$$
\begin{array}{lcl}
\textbf{class}\ \mathit{HasPattern}\ a\ \textbf{where} & & \\
\quad \mathit{setPattern} & :: & \mathit{Pattern} \to a \to a \\
\quad \mathit{setDefaultPattern} & :: & a \to a \\
\quad \mathit{getPattern} & :: & a \to \mathit{Pattern}
\end{array}
$$

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Pattern} & = & \mathit{DefaultPattern} \\
 & \mid & \mathit{DashPattern}\ [\mathit{Double}] \\
 & & \textbf{deriving}\ (\mathit{Eq}, \mathit{Show}, \mathit{Read})
\end{array}
$$

A.10 Zeichenstifte

Instanzen von *HasPen*: *Frame*, *Path* und *PathElemDescr*.

Pen ist Instanz von: *Num* und *Fractional*

```
class HasPen a where
    setPen             :: Pen → a → a
    setDefaultPen      :: a → a
    getPen             :: a → Pen
```

```
data Pen               = DefaultPen
                       | PenSquare (Numeric, Numeric) Numeric
                       | PenCircle (Numeric, Numeric) Numeric
                         deriving (Eq, Show, Read)
```

A.11 Pfeile

Instanzen von *HasArrowHead*: *Path* und *PathElemDescr*.

```
class HasArrowHead a where
    setArrowHead         :: ArrowHead → a → a
    removeArrowHead      :: a → a
    getArrowHead         :: a → Maybe ArrowHead
    setStartArrowHead    :: ArrowHead → a → a
    removeStartArrowHead :: a → a
    getStartArrowHead    :: a → Maybe ArrowHead
```

```
data ArrowHead         = DefaultArrowHead
                       | ArrowHead (Maybe Double) (Maybe Double)
                                   ArrowHeadStyle
                         deriving (Eq, Show, Read)
```

```
data ArrowHeadStyle    = AHFilled
                       | AHLine
                         deriving (Eq, Show, Read)
```

A.12 Flächen

Instanzen von *IsArea*: *IsPath a* ⇒ *IsArea* [*a*], *Path* und *Area*.

Area ist Instanz von: *IsPicture*, *HasDefine*, *HasColor*, *HasPen*, *HasLayer*, *Show*, *Eq*, *Read*.

```
class IsArea a where
    toArea             :: a → Area
```

```
class HasLayer a where
    setBack            :: a → a
    setFront           :: a → a
    getLayer           :: a → Layer
```

```
data Area              = Area AreaDescr Path
                         deriving (Eq, Show, Read)
```

```
data AreaDescr        = AreaDescr{ arColor :: Color,
                                   arLayer :: Layer,
                                   arPen :: Pen}
                        deriving (Eq, Read)

stdAreaDescr          :: AreaDescr
stdAreaDescr          = AreaDescr{ arColor = black,
                                   arLayer = Front,
                                   arPen = DefaultPen}

data Layer            = Front | Back
                        deriving (Eq, Show, Read)
```

A.13 Clipping

Mit Clipping lassen sich interessante Effekte erzielen, wie in Abbildung A.13.1 zu sehen. Die Bounding Box eines mit der Funktion *clip* erzeugten Bildes ist immer rechteckig.

```
let a = clip (ref NW -- ref NE -- ref SW -- cycle)
             (box "clip"
               # setBGColor (grey 0.8))
    b = clip (ref SW -- ref NE -- ref SE -- cycle)
             (box (tex "clip" # setColor white)
               # setBGColor black)
in column [a, vspace (−5), math "\\oplus", vspace (−5),
           b, math "\\downarrow", ooalign [a, b]]
```

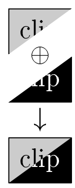


Abbildung A.13.1: Clipping ermöglicht interessante Effekte.

A.14 Transformationen

Für oft benötigte Transformationen existieren neben den schon erwähnten, zwei weitere vordefinierte Funktionen, die auf Bilder anwendbar sind.

$$\mathit{reflectX}, \mathit{reflectY} \quad :: \quad \mathit{IsPicture}\ a \Rightarrow a \to \mathit{Picture}$$

Wenn mehrere Transformationen hintereinander auf ein Bild angewendet werden sollen, ist es effizienter, die Funktion

$$\mathit{transform} \quad :: \quad \mathit{IsPicture}\ a \Rightarrow \mathit{Transformation} \to a \to \mathit{Picture}$$

zu verwenden und die Transformationsmatrizen mit Hilfe des Operators (&) zu verknüpfen. Eine Streckung entlang der x–Achse um den Faktor zwei mit anschließender Drehung um 30 Grad ergibt sich mit der Funktion *transform* (*scaledX* 2 & *rotated* 30). Zu diesem Zweck sind auch einige Transformationsmatrizen vordefiniert. Beliebige affine Transformationsmatrizen definiert die Funktion *affine*.

$$
\begin{array}{lcl}
\mathit{shifted} & :: & (\mathit{Numeric}, \mathit{Numeric}) \rightarrow \mathit{Transformation} \\
\mathit{reflectedX}, \mathit{reflectedY} & :: & \mathit{Transformation} \\
\mathit{rotated}, \mathit{scaledX}, \mathit{scaledY}, & & \\
\mathit{scaled}, \mathit{skewedX}, \mathit{skewedY} & :: & \mathit{Numeric} \rightarrow \mathit{Transformation} \\
\mathit{affine} & :: & (\mathit{Numeric}, \mathit{Numeric}, \mathit{Numeric}, \mathit{Numeric}, \mathit{Numeric}, \mathit{Numeric}) \\
 & \rightarrow & \mathit{Transformation}
\end{array}
$$

$$
\begin{array}{lcl}
\textbf{let } \mathit{rek}\ 0\ \mathit{pic} & = & \mathit{pic} \\
\quad \mathit{rek}\ n\ \mathit{pic} & = & \mathit{ooalign}\ [\mathit{draw}\ [p]\ [\mathit{toArea}\ a\ \#\ \mathit{setColor}\ 0.6, \\
 & & \qquad\qquad\qquad \mathit{toArea}\ p\ \#\ \mathit{setColor}\ \mathit{white}], \\
 & & \qquad\qquad \mathit{rotate}\ 90\ (\mathit{scale}\ (1\ /\ 3)\ (\mathit{rek}\ (n-1)\ \mathit{pic}))] \\
\quad p & = & \mathit{transformPath}\ (\mathit{scaled}\ 30)\ \mathit{fullcircle} \\
\quad a & = & (\mathit{vec}\ (90,0)\ ..\ \mathit{vec}\ (0,30)\ ..\ \mathit{vec}\ (-90,0)\ \#\ \mathit{setEndCurl}\ 1) \\
 & & \quad ..\ \mathit{vec}\ (0,-30)\ ..\ \mathit{cycle}\ \#\ \mathit{setEndCurl}\ 1 \\
\textbf{in } \mathit{rek}\ 6\ \mathit{empty} & &
\end{array}
$$

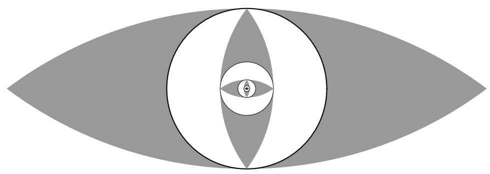


Abbildung A.14.1: Eine rekursive Grafik (Siehe figure 28 in [Hob92]).

A.15 Bitmap Grafiken

$$
\begin{array}{lcl}
\textbf{data } \mathit{BitDepth} & = & \mathit{Depth1} \mid \mathit{Depth8} \mid \mathit{Depth24} \\
 & & \textbf{deriving } (\mathit{Eq}, \mathit{Show}, \mathit{Read})
\end{array}
$$

Abbildung A.15.1 zeigt, wie sich Bitmapgrafiken in *functional* METAPOST verwenden lassen.

$$\mathit{image} :: \mathit{BitDepth} \rightarrow [\mathit{String}] \rightarrow \mathit{Picture}$$

A.16 Erweiterungen

A.16.1 Canvasgrafik

Canvas ist Instanz von: *IsPicture*, *HasRelax* und *HasConcat*.

A.16.2 Turtlegrafik

Turtle ist Instanz von: *IsPicture*, *IsHideable*, *HasRelax* und *HasConcat HasPicture*, *HasColor*, *HasPen*.

rowSepBy 10 [*rbox* 15 (*scale* 3 (*image Depth24 fruits*)) # *setDX* 10 # *setDY* 10,
rbox 15 (*scale* 6 (*image Depth8 tiger*)) # *setDX* 10 # *setDY* 10,
fuzzy 4 5 (*scale* 2 (*image Depth1 woodpecker*))]

Abbildung A.15.1: Drei verschiedene Bitmaps mit 24, 8 und einem Bit Farbtiefe.

data *Turtle*	$=$	*TConc Turtle Turtle*
	$\mid$	*TDropPic Picture*
	$\mid$	*TColor Color Turtle*
	$\mid$	*TPen Pen Turtle*
	$\mid$	*THide Turtle*
	$\mid$	*TForward Numeric*
	$\mid$	*TTurn Numeric*
	$\mid$	*TPenUp*
	$\mid$	*TPenDown*
	$\mid$	*THome*
	$\mid$	*TFork Turtle Turtle*
		deriving *Show*

Es ist nützlich, noch Funktionen zu definieren, die bei einem positiven Winkel in die gewünschte Richtung drehen, denn man vergißt leicht, ob ein positiver Winkel gegen oder mit dem Uhrzeigersinn verläuft.

turnl	::	*Numeric* $\rightarrow$ *Turtle*
turnl a	$=$	*TTurn a*

turnr	::	*Numeric* $\rightarrow$ *Turtle*
turnr a	$=$	*TTurn* $(-a)$

A.16.3 Bäume

Tree ist Instanz von: *IsPicture* und *HasName*.

Edge ist Instanz von: *HasColor*, *HasLabel*, *HasPen*, *HasPattern*, *HasArrowHead HasStartEndDir*, *IsHideable*

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Tree} & = & \mathit{Node\ Picture\ NodeDescr}\ [\mathit{Edge}] \\
 & & \textbf{deriving}\ \mathit{Show} \\[1ex]
\textbf{data}\ \mathit{Edge} & = & \mathit{Edge\ Path\ Tree} \\
 & | & \mathit{Cross\ Path} \\
 & & \textbf{deriving}\ \mathit{Show} \\[1ex]
\textbf{data}\ \mathit{NodeDescr} & = & \mathit{NodeDescr}\{\mathit{nEdges} :: [\mathit{Path}], \\
 & & \qquad\qquad \mathit{nAlignSons} :: \mathit{AlignSons}, \\
 & & \qquad\qquad \mathit{nDistH}, \mathit{nDistV} :: \mathit{Distance}\} \\
 & & \textbf{deriving}\ \mathit{Show} \\[1ex]
\mathit{stdNodeDescr} & :: & \mathit{NodeDescr} \\
\mathit{stdNodeDescr} & = & \mathit{NodeDescr}\{\mathit{nEdges} = [\,], \\
 & & \qquad\qquad \mathit{nAlignSons} = \mathit{DefaultAlign}, \\
 & & \qquad\qquad \mathit{nDistH} = 8, \\
 & & \qquad\qquad \mathit{nDistV} = 10\} \\[3ex]
\textbf{data}\ \mathit{Distance} & = & \mathit{DistCenter\ Numeric} \\
 & | & \mathit{DistBorder\ Numeric} \\
 & & \textbf{deriving}\ (\mathit{Eq}, \mathit{Show}) \\[3ex]
\textbf{data}\ \mathit{NodeName} & = & \mathit{Parent} | \mathit{This} | \mathit{Root} | \mathit{Up\ Int} | \mathit{Son\ Int} \\
 & & \textbf{deriving}\ \mathit{Show}
\end{array}
$$

Es existieren Funktionen, die Attributierungsfunktionen auf einen gesamten Baum fortsetzen. Entweder, es wird eine Funktion auf alle Knoten des Baumes angewandt oder nur auf alle Knoten eines bestimmten Levels. Attributierungen können auch auf alle Bilder oder alle Kanten des Baumes angewendet werden.

$$
\begin{array}{lcl}
\mathit{forEachNode} & :: & (\mathit{Tree} \to \mathit{Tree}) \to \mathit{Tree} \to \mathit{Tree} \\
\mathit{forEachLevelNode} & :: & \mathit{Int} \to (\mathit{Tree} \to \mathit{Tree}) \to \mathit{Tree} \to \mathit{Tree} \\
\mathit{forEachPic} & :: & (\mathit{Picture} \to \mathit{Picture}) \to \mathit{Tree} \to \mathit{Tree} \\
\mathit{forEachEdge} & :: & (\mathit{Path} \to \mathit{Path}) \to \mathit{Tree} \to \mathit{Tree}
\end{array}
$$

Wenn keine Attributsänderung an einem Knoten vorgenommen werden soll, mündet die Kante in einen normalen Knoten. Wir fassen für diesen Fall Kante und Knoten in einer Funktion zusammen, was etwas Patz sparen kann.

$$
\begin{array}{lcl}
\mathit{enode} & :: & \mathit{IsPicture}\ a \Rightarrow a \to [\mathit{Edge}] \to \mathit{Edge} \\
\mathit{enode}\ p\ ts & = & \mathit{edge}\ (\mathit{node}\ p\ ts)
\end{array}
$$

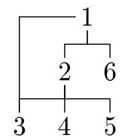
Die Art in der Kanten gezeichnet werden, läßt sich vollständig an die Bedrürfnisse anpassen. Treppenförmige Kanten z.B. lassen sich mit der Funktion *stair* leicht realisieren.

```
cross'          :: Path → Edge
cross'          =  Cross

edge'           :: Path → Tree → Edge
edge'           =  Edge

stair           :: Point → Point → Path
stair p1 p2     =  p1 -- p1 + vec (0, 0.5 * ydist p2 p1)
                   --p2 − vec (0, 0.5 * ydist p2 p1) -- p2
```

```
let sedge = edge' (stair (ref (This ◁ C)) (ref (Parent ◁ C)))
    scross p = cross' ((ref (This ◁ C)) -- xy p (ref (This ◁ C)) -- p)
in node "1" [sedge (node "2" [sedge (node "3" [] # setName "3"),
                              sedge (node "4" []),
                              sedge (node "5" [])]),
             sedge (node "6" []),
             scross (ref ("3" ◁ C))]
```



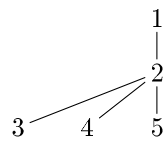
Für die Ausrichtung der Söhne eines Knotens sind acht verschiedene Optionen vordefiniert und die Möglichkeit, eigene Ausrichtungsfunktionen zu verwenden.

```
data AlignSons  =  DefaultAlign
                |  AlignLeft
                |  AlignRight
                |  AlignLeftSon
                |  AlignRightSon
                |  AlignOverN Int
                |  AlignAngles [Numeric]
                |  AlignConst Numeric
                |  AlignFunction (NodeDescr → [Extent] → Int → [Numeric])
                   deriving Show
```

Es kann notwendig sein sein, den Vaterknoten über dem n-ten Sohn zu plazieren.

```
alignOverN      :: Int → AlignSons
```

```
node "1" [edge (node "2" [edge (node "3" []),
                          edge (node "4" []),
                          edge (node "5" [])]
                # setAlign (alignOverN 3))]
```

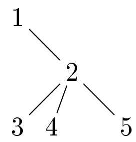


Wenn mehrere Kanten im gleichen Winkel verlaufen sollen, kann die Positionierung auch anhand der einzelnen Kantenwinkel erfolgen.

```
alignAngles     :: [Double] → AlignSons
```

Die Kanten (1,2) und (2,5) liegen im folgenden Beispiel genau auf einer Linie. Die wäre auch dann noch der Fall, wenn die horizontalen Abstände zwischen den Niveaus sehr unterschiedlich wären.

```
node "1" [edge (node "2" [edge (node "3" []),
                          edge (node "4" []),
                          edge (node "5" [])]
                 # setAlign (alignAngles [45, 70, −45]))]
#setAlign (alignAngles [−45])
```

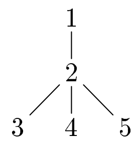


Sollen die Knoten an regelmäßigen horizontalen Abständen ausgerichtet sein, hilft der Ausdruck

$$alignConst \quad :: \quad Double \rightarrow AlignSons$$

Der Vater ist mittig über seinen Söhnen angeordnet.

```
node "1" [edge (node "2" [edge (node "3" []),
                          edge (node "4" []),
                          edge (node "5" [])]
                 # setAlign (alignConst 10))]
```



Die neunte Ausrichtungsart ermöglicht die Formulierung eigener Funktionen, die die Positionen der Söhne relativ zu ihrem Vater bestimmen. Der erste Parameter des Typs *NodeDescr* ist nützlich, um Informationen über die Attributierung zu erlangen, in der z.B. die gewünschten Knotenabstände gespeichert sind. Die Liste vom Typ [*Extent*] enthält alle Umrisse der zu positionierenden Teilbäume. Der letzte Parameter beschreibt das Niveau im Baum, um auch davon die Positionierung abhängig machen zu können. Als Ergebnis muß die Funktion eine Liste von Zahlen zurückliefern in der für jede Teilbaumwurzel die relative Position zum Vater steht.

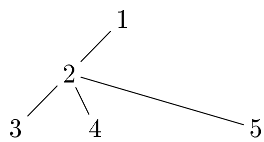
$$alignFunction \quad :: \quad (NodeDescr \rightarrow [Extent] \rightarrow Int \rightarrow [Numeric]) \rightarrow AlignSons$$

Die Funktion, die z.B. nicht von den vordefinierten Positionierungen abgedeckt wird, ist die direkte Übergabe der relativen Positionen bezogen auf den Vater. Dies läßt sich leicht selbst definieren.

$$alignConsts\ cs \quad = \quad alignFunction\ (\lambda\_\ \_\ \_ \rightarrow cs)$$

Die Umrisse der Teilbäume bleiben, wie die anderen Parameter, gänzlich unberücksichtigt. Deshalb ist natürlich nicht gewährleistet, daß die auf diese Weise positionierten Teilbäume sich nicht überdecken.

```
node "1" [edge (node "2" [edge (node "3" []),
                          edge (node "4" []),
                          edge (node "5" [])]
                 # setAlign (alignConsts [−10, 5, 35]))]
#setAlign (alignConsts [−10])
```



Leider kommt es zu einem Fehler, wenn mehr Teilbäume zu plazieren sind, als in der Liste Positionen angegeben sind. Um die Funktion etwas robuster gegen diesen Fehler zu gestalten, ist dieser Fall abzufangen und sind gegebenenfalls Werte hinzuzufügen, wie in folgender Variante.

$$
\begin{array}{lll}
\mathit{alignConsts}'\ \mathit{cs} & = & \mathit{alignFunction}\ (\lambda\_\ \mathit{es}\ \_ \\
 & & \quad \rightarrow \mathbf{let}\ n = \mathit{length}\ \mathit{es} \\
 & & \quad\quad \mathbf{in}\ \mathbf{if}\ n > \mathit{length}\ \mathit{cs} \\
 & & \quad\quad\quad \mathbf{then}\ \mathit{resumeList}\ n\ \mathit{cs} \\
 & & \quad\quad\quad \mathbf{else}\ \mathit{cs}) \\
\quad \mathbf{where} & & \\
\quad \mathit{resumeList}\ n\ [\,] & = & [0] \\
\quad \mathit{resumeList}\ n\ [c] & = & \mathit{take}\ n\ [c, 2 * c\,..] \\
\quad \mathit{resumeList}\ n\ \mathit{cs} & = & \mathit{take}\ n\ (\mathit{cs} +\!\!+ [\mathit{last}\ \mathit{cs} + d, \mathit{last}\ \mathit{cs} + 2 * d\,..]) \\
\quad\quad \mathbf{where} & & \\
\quad\quad d & = & \mathit{last}\ \mathit{cs} - \mathit{last}\ (\mathit{init}\ \mathit{cs})
\end{array}
$$

A.17 Ein Beispiel für generierten METAPOST–Quellcode

Um auch einmal ein vollständiges und etwas längeres Beispiel für das Ergebnis einer Übersetzung zu geben, zeigen wir hier den METAPOST–Quellcode für den linken Baum der Abbildung 5.3.7.

```
batchmode;
verbatimtex
\documentclass[10pt,oneside,a4paper]{report}
\usepackage[german,english]{babel}
\usepackage{amsfonts}
\SetMathAlphabet{\mathrm}{normal}{OT1}{ptm}{m}{n}
\SetMathAlphabet{\mathbf}{normal}{OT1}{ptm}{bx}{n}
\SetMathAlphabet{\mathit}{normal}{OT1}{ptm}{m}{it}
\begin{document}
etex

input boxes
input FuncMP
beginfig(101);
warningcheck := 0;
picture p[], q[], r[];
transform t[],tr[];
pair s[];
pair pv[][];
pair pvi[][];
numeric nv[][];
numeric nvi[][];
path tempPath;
defDX := 3;
defDY := 3;
txtDX := 2;
txtDY := 2;
boxit.b10(btex A etex);
b10.dx = txtDX;
b10.dy = txtDY;
boxit.b9(btex B etex);
b9.dx = txtDX;
b9.dy = txtDY;
boxit.b8(btex C etex);
```

```
 b8.dx = txtDX;
 b8.dy = txtDY;
 boxit.b7(btex D etex);
 b7.dx = txtDX;
 b7.dy = txtDY;
 boxit.b6(btex E etex);
 b6.dx = txtDX;
 b6.dy = txtDY;
 boxit.b5(btex F etex);
 b5.dx = txtDX;
 b5.dy = txtDY;
 boxit.b4(btex G etex);
 b4.dx = txtDX;
 b4.dy = txtDY;
 boxit.b3(btex H etex);
 b3.dx = txtDX;
 b3.dy = txtDY;
 nvi2 2 = (xpart (b10.w-(b10.c)));
 nvi2 8 = (xpart (b9.w-(b9.c)));
 nvi2 14 = (xpart (b8.w-(b8.c)));
 nvi2 20 = (xpart (b7.w-(b7.c)));
 nvi2 26 = (xpart (b6.w-(b6.c)));
 nvi2 32 = (xpart (b5.w-(b5.c)));
 nvi2 38 = (xpart (b4.w-(b4.c)));
 nvi2 44 = (xpart (b3.w-(b3.c)));
 nvi2 3 = (xpart (b10.e-(b10.c)));
 nvi2 9 = (xpart (b9.e-(b9.c)));
 nvi2 15 = (xpart (b8.e-(b8.c)));
 nvi2 21 = (xpart (b7.e-(b7.c)));
 nvi2 27 = (xpart (b6.e-(b6.c)));
 nvi2 33 = (xpart (b5.e-(b5.c)));
 nvi2 39 = (xpart (b4.e-(b4.c)));
 nvi2 45 = (xpart (b3.e-(b3.c)));
 nvi2 10 = (ypart (b10.n-(b10.c)));
 nvi2 16 = max((ypart (b9.n-(b9.c))),(ypart (b6.n-(b6.c))),(ypart (b5.n-(b5.c))))!
!;
 nvi2 22 = max((ypart (b8.n-(b8.c))),(ypart (b7.n-(b7.c))),(ypart (b4.n-(b4.c))),!
!(ypart (b3.n-(b3.c))));
 nvi2 11 = (ypart (b10.c-(b10.s)));
 nvi2 17 = max((ypart (b9.c-(b9.s))),(ypart (b6.c-(b6.s))),(ypart (b5.c-(b5.s))))!
!;
 nvi2 23 = max((ypart (b8.c-(b8.s))),(ypart (b7.c-(b7.s))),(ypart (b4.c-(b4.s))),!
!(ypart (b3.c-(b3.s))));
 nvi2 7 = (-(nvi2 16))-(nvi2 11);
 nvi2 13 = (-(nvi2 22))-(nvi2 17);
 nvi2 12 = ((-(nvi2 15-(nvi2 20)+8)))/(2);
 nvi2 18 = (nvi2 15-(nvi2 20)+8)/(2);
 nvi2 36 = ((-(nvi2 39-(nvi2 44)+8)))/(2);
 nvi2 42 = (nvi2 39-(nvi2 44)+8)/(2);
 nvi2 6 = ((-(max(nvi2 9-(nvi2 26-(nvi2 27-(nvi2 32)+8))+8,nvi2 21+nvi2 18-(nvi2 !
!38+nvi2 36)+8))))/(2);
 nvi2 24 = (nvi2 9-(nvi2 26)+8-(nvi2 27-(nvi2 32)+8))/(2);
 nvi2 30 = (max(nvi2 27+nvi2 9-(nvi2 26)+8-(nvi2 32)+8,nvi2 21+nvi2 18-(nvi2 38+n!
```

```
!vi2 36)+8))/(2);
 b9.c = b10.c+(nvi2 6,nvi2 7-(10));
 b8.c = b9.c+(nvi2 12,nvi2 13-(10));
 b7.c = b9.c+(nvi2 18,nvi2 13-(10));
 b6.c = b10.c+(nvi2 24,nvi2 7-(10));
 b5.c = b10.c+(nvi2 30,nvi2 7-(10));
 b4.c = b5.c+(nvi2 36,nvi2 13-(10));
 b3.c = b5.c+(nvi2 42,nvi2 13-(10));
 boxit.b2();
 b2.sw = (min((xpart (b10.w)),(xpart (b9.w)),(xpart (b8.w)),(xpart (b7.w)),(xpart!
! (b6.w)),(xpart (b5.w)),(xpart (b4.w)),(xpart (b3.w))),min((ypart (b10.s)),(ypar!
!t (b9.s)),(ypart (b8.s)),(ypart (b7.s)),(ypart (b6.s)),(ypart (b5.s)),(ypart (b4!
!.s)),(ypart (b3.s))));
 b2.ne = (max((xpart (b10.e)),(xpart (b9.e)),(xpart (b8.e)),(xpart (b7.e)),(xpart!
! (b6.e)),(xpart (b5.e)),(xpart (b4.e)),(xpart (b3.e))),max((ypart (b10.n)),(ypar!
!t (b9.n)),(ypart (b8.n)),(ypart (b7.n)),(ypart (b6.n)),(ypart (b5.n)),(ypart (b4!
!.n)),(ypart (b3.n))));
 fixsize(b2);
 cloneit.b1(b2);
 drawunboxed(b10, b9, b8, b7, b6, b5, b4, b3);
 tempPath := b9.c--b10.c;
 draw (subpath (0,1) of tempPath cutbefore bpath b9 cutafter bpath b10);
 tempPath := b8.c--b9.c;
 draw (subpath (0,1) of tempPath cutbefore bpath b8 cutafter bpath b9);
 tempPath := b7.c--b9.c;
 draw (subpath (0,1) of tempPath cutbefore bpath b7 cutafter bpath b9);
 tempPath := b6.c--b10.c;
 draw (subpath (0,1) of tempPath cutbefore bpath b6 cutafter bpath b10);
 tempPath := b5.c--b10.c;
 draw (subpath (0,1) of tempPath cutbefore bpath b5 cutafter bpath b10);
 tempPath := b4.c--b5.c;
 draw (subpath (0,1) of tempPath cutbefore bpath b4 cutafter bpath b5);
 tempPath := b3.c--b5.c;
 draw (subpath (0,1) of tempPath cutbefore bpath b3 cutafter bpath b5);
 endfig;


 \end
```

Anhang B

ASCII–Repräsentierung der Operatoren

Um eine bessere Lesbarkeit dieser Arbeit zu erreichen, sind einige Operatoren etwas kompakter gedruckt, als dies im Haskell Programmtext möglich ist. Das Programm `lhs2TeX` von Ralf Hinze nimmt diese Umwandlung bei der Übersetzung des Programms in einen LaTeX–Quelltext automatisch vor, wenn die entsprechenden Ersetzungsregeln angegeben sind.

<table>
<tr><th>Pretty Printed</th><th>ASCII</th><th></th></tr>
<tr><td>$\vert\vert\vert$</td><td><code>|||</code></td><td rowspan="4">Ausrichtungen</td></tr>
<tr><td>$\vert\vert\vert\vert$</td><td><code>||||</code></td></tr>
<tr><td>$\boxminus$</td><td><code>|-|</code></td></tr>
<tr><td>$\boxed{=}$</td><td><code>|=|</code></td></tr>
<tr><td>--</td><td><code>.-.</code></td><td rowspan="4">Pfadkonstruktoren</td></tr>
<tr><td>---</td><td><code>.--.</code></td></tr>
<tr><td>..</td><td><code>...</code></td></tr>
<tr><td>...</td><td><code>....</code></td></tr>
<tr><td>$\triangleleft$</td><td><code><+</code></td><td rowspan="2">Namenskonstruktoren</td></tr>
<tr><td>$\triangleleft\!\!\ast$</td><td><code><*</code></td></tr>
<tr><td>$\doteq$</td><td><code>.=</code></td><td>Gleichheit</td></tr>
<tr><td>$\dot{<}$</td><td><code>.<</code></td><td rowspan="4">Vergleichsoperatoren</td></tr>
<tr><td>$\dot{\leqslant}$</td><td><code>.<=</code></td></tr>
<tr><td>$\dot{\equiv}$</td><td><code>.==</code></td></tr>
<tr><td>$\dot{\not\equiv}$</td><td><code>./=</code></td></tr>
<tr><td>$\dot{\ast}$</td><td><code>.*</code></td><td>Multiplikation Zahl – Punkt</td></tr>
<tr><td>$<\!\!+\!\!>$</td><td><code><+></code></td><td rowspan="2">Pretty Printer</td></tr>
<tr><td>$\$\$$</td><td><code>$$</code></td></tr>
<tr><td>λ</td><td><code>\</code></td><td>Lambda Abstraktion</td></tr>
<tr><td>$cycle$</td><td><code>cycle'</code></td><td rowspan="2"></td></tr>
<tr><td>$default$</td><td><code>default'</code></td></tr>
</table>

Anhang C

Glossar

Caml: ML-Implementierung von ASCÁNDER SUÁREZ am Institut National de Recherche en Informatique et en Automatique (INRIA) [Ler95]. Die Sprache hat Ähnlichkeit mit standard ML, weicht aber im Modulkonzept und einigen kleineren Punkte davon ab.

GHC: Glosgow Haskell Compiler ist ein, an der Universität Glasgow, von SIMON PEYTON JONES und vielen anderen, entwickelter Haskell–Compiler [Tea98].

Haskell: Für die Implementierung von *functional* METAPOST kommt die Version 1.4 des Haskell Sprachstandards [PHe97b, PHe97a] zum Einsatz.

Hugs: Haskell Users Gofer System. Ursprünglich aus der Sprache Gofer [Jon94] hervorgegangener Interpreter der Universitäten Nottingham und Yale. Wir benutzen in dieser Arbeit die Version vom November 98, die den Sprachstandard Haskell 1.4 implementiert.

LaTeX: Ein von LESLIE LAMPORT entwickeltes System von Makros und Formatierungsprozeduren, das den Umgang mit dem Satzsystem ☞ *TeX* erleichtert [Lam94]. Es hat sich dank der vielen gestalterischen Möglichkeiten aber auch aufgrund der Systemunabhängigkeit, im universitäten Umfeld als Standardwerkzeug zur Erstellung mathematisch–technischer Dokumente entwickelt.

METAFONT: Dieses von DONALD E. KNUTH entwickelte Programm, ermöglicht eine komfortable geräteunabhängige Beschreibung von Zeichensätzen [Knu86]. Das Programm erzeugt Zeichensatz und Metrikdateien, die das Textsatzsystem ☞ *TeX* benötigt.

PostScript: Eine Seitenbeschreibungssprache [Ado85] der Firma Adobe, die mit der Zeit zu einem Standarddokumentformat avancierte. PostScript ist auflösungs– sowie geräteunabhängig und bietet eine vollständige Programmiersprache mit Schleifen, Prozeduren und vielen Datentypen. Die Sprache arbeitet stackorientiert und mit Postfix-Notation. Speziell zum für die Integration von Bildern in Texte, hat Adobe das Format Encapsulated PostScript definiert (EPS), das in Kommentarzeilen zusätzliche Informationen über Größe und Auflösung des Bildes verfügbar macht.

Scheme: Ein Dialekt der Sprache Lisp mit statischer Bindung [CRe91].

TEX: Im Jahr 1977 begann DONALD E. KNUTH mit der Entwicklung diese Text–Satzsystems, das die Formatierung von Dokumenten in einer Qualität erlaubt, die der eines professionellen Setzers nahe kommt [Knu84]. Das Dokument kann in eine Programmiersprache beschrieben werden, die sich mit Makrosammlungen, wie ☞ *LATEX*, ergänzen laßt. Das Ausgabeformat DVI (DeVice Independent) ist, wie der Name andeutet, geräteunabhängig und das gesamte Satzsystem, zu dem auch ☞ *METAFONT* gehört, ist auf alle wichtigen Systemplattformen portiert.

Abbildungsverzeichnis

Literaturverzeichnis

[Abe85] H. Abelson. *Einführung in LOGO*. IWT-Verlag, Vaterstetten bei München, second edition, 85.

[Ad82] H. Abelson and A. A. diSessa. *TurtleGeometry*. MIT Press, third edition, February 1982.

[Ado85] Adobe Systems, Inc. *PostScript ® Language Reference Manual*. Adobe Systems Incorporated, 1985.

[Ado88] Adobe Systems, Inc. *PostScript ® Language Program Design*. Adobe Systems Incorporated, 1988.

[ASU88] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilerbau*, volume 1. Addison-Wesley, Reading, MA, USA, 1988.

[Bir98] Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall Europe, London, second edition, 98.

[BO96] Gerth Stølting Brodal and Chris Okasaki. Optimal purely functional priority queues. *Journal of Functional Programming*, 6(6):839–857, November 1996.

[CRe91] William Clinger and Jonathan Rees (editors). The revised^4 report on the algorithmic language scheme. *In ACM Lisp Pointers*, 4(3):pages 1–55, 1991.

[CS96] Emmanuel Chailloux and Ascánder Suárez. mlPicTEX, a picture enxironment for LaTeX. In *Record of the fifth ACM SIGPLAN workshop on ML ans it's Applications*, 1996.

[CVM97] Koen Claessen, Ton Vullinghs, and Erik Meijer. Structuring graphical paradigms in tkgofer. In *International Conference on Functional Programming*. ACM, June 1997.

[Fel92] W. D. Fellner. *Computergrafik*. BI-Wissenschaftsverlag, 1992.

[FJ95] Sigbjorn Finne and Simon Peyton Jones. Pictures: A simple structured graphics model. In *Proceedings of the Glasgow Functional Programming Group Workshop*, Jule 1995.

[GKP92] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics*. Addison-Wesley, Reading, MA, USA, eigth edition, 1992.

[GR97] Michael Goosens and Sebastian Rahts. *The LaTeX Graphics Companion*. Addison-Wesley, 1997.

[HFP96] P. Hudak, J. Fasel, and J. Peterson. A gentle introduction to haskell. Technical Report YALEU/DCS/RR-901, Yale University, May 1996.

[Hin99] Ralf Hinze. Constructing red-black trees. 1999. in preparation.

[Hob89] J. D. Hobby A METAFONT-like system with postscript output. *TUGboat*, 10(2):505–512, 1989.

[Hob92] J. D. Hobby A user's manual for METAPOST. Computing Science Technical Report no. 162, AT&T Bell Laboratories, Murray Hill, New Jersey, 1992.

[Hug95] John Hughes. The design of a pretty–printer library. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925. Springer Verlag, 1995.

[JJM97] Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: an exploration of the design space. Technical report, University of Glasgow, Oregan Graduate Institute, University of Nottingham and University of Utrecht, May 1997.

[Jon94] Mark Jones. The implementation of the gofer functional programming system. Technical report, Yale University, New Haven, May 1994.

[Kan87] Immanuel Kant. *Kritik der reinen Vernunft*, chapter 1. Transzendentale Elementarlehre, ERSTER TEIL. Die transzendentale Ästhetik, 1. Abschnitt von dem Raume, page B 37. Meiner, second edition, 1787.

[Ken93] Andrew Kennedy. Drawing trees – a case study in functional programming. Technical Report 303, University of Cambridge Computer Laboratory, June 1993.

[Ker82] Brian W. Kerninghan. Pic – a language for typesetting graphics. *Software Practice and Experience*, 12:1–21, 1982.

[Knu71] Donald E. Knuth. Optimum binary search trees. *Acta Informatica*, 1:14–25, 1971.

[Knu83] Donald E. Knuth. Literate programming. Report STAN-CS-83-981, Stanford University, Department of Computer Science, Stanford, CA, USA, 1983.

[Knu84] Donald E. Knuth. *The TEXbook*. Addison-Wesley, Reading, MA, USA, 1984.

[Knu86] Donald E. Knuth. The METAFONTbook. Addison-Wesley, Reading, MA, USA, 1986.

[Lam94] Leslie Lamport. *LATEX: A Document Preparation System: User's Guide and Reference Manual*. Addison-Wesley, Reading, MA, USA, second edition, 1994.

[Ler95] Xavier Leroy. The caml light system, release 0.74. Technical report, INRIA, `http://pauillac.inria.fr/caml/`, 1995.

[Man87] Benoît B. Mandelbrot. *Die fraktale Geometrie der Natur*, chapter 7, page 78. Birkhäuser, Basel, 1987.

[Ohl96] Thorsten Ohl. `feynMF`: Drawing feynman diagrams with LATEX and METAFONT. Technical report, Technische Hochschule Darmstadt, December 1996.

[Oka98] Chris Okasaki. Functional pearls: Constructing red-black trees in a functional setting. *Journal of functional programming*, 1998. to appear.

[Oss79] J. F. Ossanna. NFOFF/TROFF user's manual. Technical Report Section 22, UNIX Programmer's Manual 2, January 1979.

[Pap82] Seymour Papert. *Mindstorms: Kinder, Computer und neues Lernen*. Birkhäuser Verlag, Basel—Boston—Stuttgart, 1982.

[PHe97a] J. Peterson and K. Hammond (editors). The Haskell library report version 1.4. Research Report YALEU/DCS/RR-1105, Yale University, Department of Computer Science, April 1997.

[PHe97b] J. Peterson and K. Hammond (editors). Report on the Programming Language Haskell 1.4, A Non-strict Purely Functional Language. Research Report YALEU/DCS/RR-1106, Yale University, Department of Computer Science, April 1997.

[Pur97] Helen Purchase. Which aesthetic has the greatest effect on human understanding? In G. Di Battissa, editor, *5th International Symposium, GD '97*, volume 1353, pages 248–261, Rome, Italy, September 1997. Lecture Notes in Computer Science.

[Rad88] G. M. Radack. Tidy drawing of m-ary trees. Technical Report CES-88-24, Department of Computer Engineering and Science, Case Western Reserve University, Cleveland, Ohio, November 1988.

[RM94] Kristoffer H. Rose and Ross Moore. *XY-pic Reference Manual*, 2.12/3β edition, October 1994.

[Ros92] Kristoffer H. Rose. Typsetting diagramms with XY-pic: User's manual. In *Proceedings of the 7th European TEX Conference*, pages 273–292, Prague, September 1992.

[Ros96] Kristoffer H. Rose. *XY-pic User's Guide*, 3.3 edition, December 1996.

[RT81] Edward M. Reingold and John S. Tilford. Tidier drawings of trees. *IEEE Trans. Software Eng.*, SE-7(2):223–228, 1981.

[Sch97] Uwe Schöning. *Theoretische Informatik kurz gefasst*. Spektrum, third edition, 1997.

[Sed92] Robert Sedgewick. *Algorithmen*. Addison-Wesley Verlag, Bonn, Germany, 1992.

[STS96] Wandy Sae-Tan and Olin Shivers. *Functional PostScript Reference Manual*. ftp://ftp-swiss.ai.mit.edu/pub/su/scsh/contrib/fps/fps-1.0.tar.gz, 1996.

[Tea98] The GHC Team. The Glorious Glasgow Haskell Compilation System, version 4, user's guide. Technical report, University of Glasgow, Department of Computing Science, G12 8QQ, 1998.

[Tho96] Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, Harlow England, 96.

[VSS96] T. Vullinghs, W. Schulte, and T. Schwinn. *The Design of a Functional GUI Library Using Constructor Classes*. Springer-Verlag, 1996.

[Wad98] Philip Wadler. A prettier printer. Technical report, Bell Labs, Lucent Technologies, March 1998.

[Wir75] Niklaus Wirth. *Algorithmen und Datenstrukturen*, chapter 4.4.10. B.G. Teubner Stuttgart, 1975.

[WS79] C. Wetherell and A. Shannon. Tidy drawings of trees. *IEEE Trans. Software Eng.*, SE-5(5):514–520, 1979.

Index

Namen sind, wie in der gesamten Arbeit, in der Schriftart Small Caps gedruckt, Funktionsnamen und Datentypen dagegen *kursiv*.