GENERISCHE PROGRAMMIERUNG
Eine Anwendung: digitale Suchbäume

RALF HINZE

Institut für Informatik III, Universität Bonn
Email: `ralf@informatik.uni-bonn.de`
Homepage: `http://www.informatik.uni-bonn.de/~ralf/`

Oktober, 2000

(Die Folien sind elektronisch verfügbar: `.../~ralf/talks.html#T17`.)

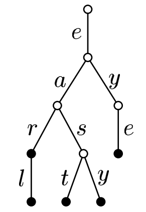
Überblick

Digitale Suchbäume

Digitale Suchbäume, auch *Tries* genannt, nutzen die *Struktur* von Suchschlüsseln für die Organisation von Daten aus.

Tries sind ursprünglich entwickelt worden, um *Mengen* von Strings zu verwalten (A. Thue, *Über die gegenseitige lage gleicher teile gewisser zeichenreihen*, 1912).

$\{ear, earl, east, easy, eye\} \Longrightarrow$



Tries in Haskell—Signatur

Strings in Haskell:

$$\mathbf{data}\ String = nil \mid cons\ Char\ String.$$

Tries sind auch geeignet, um *endliche Abbildungen* (dictionaries, look-up tables) darzustellen: der Datentyp $MapS\ V$ repräsentiert Abbildungen des Typs $String \rightarrow_{\mathrm{fin}} V$.

$$\begin{aligned} &\mathbf{data}\ MapS :: \star \rightarrow \star \\ &lookupS \quad :: \forall V\,.\, String \rightarrow MapS\ V \rightarrow Maybe\ V\,. \end{aligned}$$

Beachte: $MapS$ ist ein Typkonstruktor, eine Funktion über Typen.

Fehlerbehandlung in Haskell

Partielle Funktionen von S nach T können in Haskell als Funktionen des Typs $S \to \mathit{Maybe}\ T$ implementiert werden.

$$\mathbf{data}\ \mathit{Maybe}\ A = \mathit{nothing} \mid \mathit{just}\ A$$

Komposition partieller Funktionen:

$$\begin{aligned}&(\diamond) && :: (A \to \mathit{Maybe}\ B) \to (B \to \mathit{Maybe}\ C) \to (A \to \mathit{Maybe}\ C)\\ &(f \diamond g)\ a && = \mathbf{case}\ f\ a\ \mathbf{of}\ \{\mathit{nothing} \to \mathit{nothing}; \mathit{just}\ b \to g\ b\}.\end{aligned}$$

Tries in Haskell—Implementierung

Ein Trie selbst ist eine Komposition endlicher Abbildungen.

$$\begin{aligned}&\mathbf{data}\ \mathit{MapS}\ V = \mathit{nodeS}\ (\mathit{Maybe}\ V) && \text{-- } \mathit{nil}\\ &\qquad\qquad\qquad\qquad\ (\mathit{MapC}\ (\mathit{MapS}\ V)) && \text{-- } \mathit{cons}\end{aligned}$$

$$\begin{aligned}&\mathit{lookupS}\ :: \forall V\ .\ \mathit{String} \to \mathit{MapS}\ V \to \mathit{Maybe}\ V\\ &\mathit{lookupS}\ \mathit{nil}\ (\mathit{nodeS}\ \mathit{tn}\ \mathit{tc})\\ &\qquad = \mathit{tn}\\ &\mathit{lookupS}\ (\mathit{cons}\ c\ s)\ (\mathit{nodeS}\ \mathit{tn}\ \mathit{tc})\\ &\qquad = (\mathit{lookupC}\ c \diamond \mathit{lookupS}\ s)\ \mathit{tc}\end{aligned}$$

Beachte: MapC und $\mathit{lookupC}$ sind vorgegeben (Suchbäume, Patricia Tries).

Verallgemeinerte Tries

Tries können nicht nur zur Implementierung endlicher Abbildungen des Typs $\mathit{String} \to_{\mathrm{fin}} V$ verwendet werden.

Die Idee läßt sich auf beliebige Schlüsseltypen verallgemeinern (R.H. Connelly und F.L. Morris, *A generalization of the trie data structure*, 1995).

Grundlegende Erkenntnis: Tries basieren auf den *Potenzgesetzen*.

$$\begin{aligned}&1 \to_{\mathrm{fin}} V && \cong V\\ &(K_1 + K_2) \to_{\mathrm{fin}} V && \cong (K_1 \to_{\mathrm{fin}} V) \times (K_2 \to_{\mathrm{fin}} V)\\ &(K_1 \times K_2) \to_{\mathrm{fin}} V && \cong K_1 \to_{\mathrm{fin}} (K_2 \to_{\mathrm{fin}} V)\end{aligned}$$

Überblick

- √ *Ouvertüre:* Digitale Suchbäume (2–6).
- ☐ *Intermezzo:* Generische Programmierung. (8–22).
  - ∘ Motivation.
  - ∘ Sorten und Typen.
  - ∘ Generische Definitionen.
  - ∘ Spezializierung.
- ☐ Generische, digitale Suchbäume (24–36).
- ☐ Schlußbemerkungen (39–40).

Motivation

Die meisten modernen Programmiersprachen sind *streng getypt*.

Typsysteme bieten

- *Sicherheit:* „well-typed programs do not go wrong".
- *Dokumentation:* Typangaben dienen der Spezifikation von Schnittstellen und der Dokumentation des Quellcodes.
- *Flexibilität:* Polymorphe Typsysteme ermöglichen die Definition von Funktionen wie $\mathit{length} :: \forall A \,.\, \mathit{List}\ A \rightarrow \mathit{Int}$.

ABER, manchmal sind selbst polymorphe Typsysteme nicht flexibel genug.

So ist es nicht möglich, eine polymorphe Gleichheitsfunktion zu definieren:

$$\mathit{equal} :: \forall A \,.\, A \rightarrow A \rightarrow \mathit{Bool}.$$

Eine Funktion dieses Typs muß notwendigerweise beide Argumente ignorieren („parametricity theorem").

Die generische Programmierung löst dieses Problem.

Grundlegende Idee: die Gleichheitsfunktion wird durch *Induktion über die Struktur von Typen* definiert.

Weitere Beispiele für generische Funktionen:

- „Pretty printer" (Haskells show Funktion),
- Parser (Haskells read Funktion),
- Datenkompression,
- Datenkonversion (String, Bitstrings, XML et cetera),
- für Containertypen: map, size, toList et cetera.

Sorten und Typen

In Haskell werden neue Datentypen durch $\mathbf{data}$-Deklarationen eingeführt.

Beispiel: parametrisierte Listen und Mehrwegbäume.

$$\begin{aligned} &\mathbf{data}\ \mathit{List}\ A\ = [\,] \mid A : \mathit{List}\ A \\ &\mathbf{data}\ \mathit{Rose}\ A = \mathit{node}\ A\ (\mathit{List}\ (\mathit{Rose}\ A)) \end{aligned}$$

Der Datentyp List verallgemeinert String: es gilt $\mathit{String} \cong \mathit{List}\ \mathit{Char}$.

Haskell erlaubt darüber hinaus, von Typkonstruktoren zu abstrahieren.

$$\mathbf{data}\ \mathit{GRose}\ F\ A = \mathit{gnode}\ A\ (F\ (\mathit{GRose}\ F\ A))$$

data-Deklarationen

Eine **data**-Deklaration hat die allgemeine Form:

$$\textbf{data}\ T\ A_1\ \dots\ A_m = c_1\ T_{11}\ \dots\ T_{1m_1} \mid \cdots \mid c_n\ T_{n1}\ \dots\ T_{nm_n}.$$

Dieses Konstrukt vereinigt nicht weniger als vier verschiedene Features: Typabstraktion, Typrekursion, n-äre Summen und n-äre Produkte.

Die Typen auf der rechten Seite werden aus Typkonstanten und Typvariablen mittels Typapplikation zusammengesetzt.

Haskells Datentypen entsprechen Termen des einfach getypten λ-Kalküls.

Das einfach getypte λ-Kalkül

Sorten (die Typen der Typen):

$$\tau ::= \ \star \mid (\tau \to \tau).$$

Typen:

$$T ::= C \mid A \mid (T\ T) \mid (\Lambda A\,.\,T) \mid (\mathit{Fix}\ T).$$

Zu den primitiven Typen zählen: 1, Char, '$+$' und '$\times$' mit

$$\begin{aligned} &\textbf{data}\ 1 &&= () \\ &\textbf{data}\ A + B &&= \mathit{inl}\ A \mid \mathit{inr}\ B \\ &\textbf{data}\ A \times B &&= (A, B). \end{aligned}$$

Die **data**-Deklarationen

$$\begin{aligned} &\textbf{data}\ \mathit{String} &&= \mathit{nil} \mid \mathit{cons}\ \mathit{Char}\ \mathit{String} \\ &\textbf{data}\ \mathit{Expr} &&= \mathit{var}\ \mathit{String} \mid \mathit{if}\ \mathit{Expr}\ \mathit{Expr}\ \mathit{Expr} \\ &\textbf{data}\ \mathit{List}\ A &&= [\,] \mid A : \mathit{List}\ A \\ &\textbf{data}\ \mathit{GRose}\ F\ A &&= \mathit{gnode}\ A\ (F\ (\mathit{GRose}\ F\ A)) \end{aligned}$$

lassen sich somit wie folgt 'aufbrechen':

$$\begin{aligned} \mathit{String} &= \mathit{Fix}\ (\Lambda S\,.\,1 + \mathit{Char} \times S) \\ \mathit{Expr} &= \mathit{Fix}\ (\Lambda E\,.\,\mathit{String} + E \times E \times E) \\ \mathit{List} &= \Lambda A\,.\,\mathit{Fix}\ (\Lambda L\,.\,1 + A \times L) \\ \mathit{GRose} &= \Lambda F\,.\,\Lambda A\,.\,\mathit{Fix}\ (\Lambda T\,.\,A \times F\ T). \end{aligned}$$

Generische Definitionen

Zur Erinnerung: eine generische Funktion wird durch Induktion über die Struktur von Typen definiert.

Die gute Nachricht: es müssen nur primitive Typen berücksichtigt werden.

Das Induktionsschema wird durch die folgende Grammatik vorgegeben.

$$T_\star ::= 1 \mid \mathit{Char} \mid (T_\star + T_\star) \mid (T_\star \times T_\star)$$

Beispiel: generische Gleichheit

$$\begin{array}{ll} equal\langle T :: \star\rangle & :: T \to T \to Bool \\ equal\langle 1\rangle\ ()\ () & = true \\ equal\langle Char\rangle\ c_1\ c_2 & = equalChar\ c_1\ c_2 \\ equal\langle A+B\rangle\ (inl\ a_1)\ (inl\ a_2) & = equal\langle A\rangle\ a_1\ a_2 \\ equal\langle A+B\rangle\ (inl\ a_1)\ (inr\ b_2) & = false \\ equal\langle A+B\rangle\ (inr\ b_1)\ (inl\ a_2) & = false \\ equal\langle A+B\rangle\ (inr\ b_1)\ (inr\ b_2) & = equal\langle B\rangle\ b_1\ b_2 \\ equal\langle A\times B\rangle\ (a_1,b_1)\ (a_2,b_2) & = equal\langle A\rangle\ a_1\ a_2 \wedge equal\langle B\rangle\ b_1\ b_2 \end{array}$$

Beachte: die spitzen Klammern umfassen das Typargument.

Generische Definitionen

Eine generische Definition hat die folgende *allgemeine* Form.

$$\begin{array}{ll} poly\langle T :: \star\rangle & :: Poly\ T \\ poly\langle 1\rangle & = poly_1 \\ poly\langle Char\rangle & = poly_{Char} \\ poly\langle A+B\rangle & = poly_+\ (poly\langle A\rangle)\ (poly\langle B\rangle) \\ poly\langle A\times B\rangle & = poly_\times\ (poly\langle A\rangle)\ (poly\langle B\rangle). \end{array}$$

Beachte: $poly$ ist induktiv über die Struktur von $T_\star$ definiert.

Spezialisierung

Wie können wir $equal\langle T\rangle$ spezialisieren?

$$equal\langle List\ Char\rangle \qquad equal\langle List\ (List\ Int)\rangle$$

Die Spezialisierung sollte kompositional sein.

Idee: die spezialisierte Funktion folgt der Struktur des jeweiligen Datentyps.

$$equalList\ equalChar \qquad equalList\ (equalList\ equalInt)$$

Beachte: $equalList$ bildet $equal\langle A\rangle$ auf $equal\langle List\ A\rangle$ ab.

Spezialisierung als Interpretation

Die Spezialisierung von $poly$ auf Typen beliebiger Sorten (!) ist somit gegeben durch

$$\begin{array}{ll} poly\langle\!\langle C\rangle\!\rangle\varrho & = poly_C \\ poly\langle\!\langle A\rangle\!\rangle\varrho & = \varrho(A) \\ poly\langle\!\langle T\ U\rangle\!\rangle\varrho & = (poly\langle\!\langle T\rangle\!\rangle\varrho)\ (poly\langle\!\langle U\rangle\!\rangle\varrho) \\ poly\langle\!\langle \Lambda A\,.\,T\rangle\!\rangle\varrho & = \lambda poly_A\,.\,poly\langle\!\langle T\rangle\!\rangle\varrho(A := poly_A) \\ poly\langle\!\langle Fix\ T\rangle\!\rangle\varrho & = fix\ (poly\langle\!\langle T\rangle\!\rangle\varrho). \end{array}$$

Typapplikation wird durch Wertapplikation, Typabstraktion durch Wertabstraktion und Typrekursion durch Wertrekursion interpretiert.

Der Typ der Spezialisierung lautet (T is geschlossen):

$$poly\langle\!\langle T :: \tau \rangle\!\rangle :: Poly\langle \tau \rangle\ T,$$

wobei $Poly\langle\tau\rangle$ induktiv über die Struktur von Sorten definiert ist:

$$\begin{array}{ll} Poly\langle\tau\rangle & :: \tau \to \star \\ Poly\langle\star\rangle\ T & = Poly\ T \\ Poly\langle\alpha \to \beta\rangle\ T & = \forall A\,.\,Poly\langle\alpha\rangle\ A \to Poly\langle\beta\rangle\ (T\ A). \end{array}$$

Beispiel: Spezialisierung von $equal\langle List\rangle$

$$\begin{array}{ll} equalList & :: \forall A\,.\,(A \to A \to Bool) \\ & \qquad\qquad \to (List\ A \to List\ A \to Bool) \\ equalList\ equalA\ [\,]\ [\,] & = true \\ equalList\ equalA\ [\,]\ (a_2 : x_2) & = false \\ equalList\ equalA\ (a_1 : x_1)\ [\,] & = false \\ equalList\ equalA\ (a_1 : x_1)\ (a_2 : x_2) & \\ & = equalA\ a_1\ a_2 \wedge equalList\ equalA\ x_1\ x_2 \end{array}$$

Beispiel: Spezialisierung von $equal\langle GRose\rangle$

$$\begin{array}{l} equalGRose :: \forall F\,.\,(\forall B\,.\,(B \to B \to Bool) \\ \qquad\qquad\qquad\qquad\qquad \to (F\ B \to F\ B \to Bool)) \\ \qquad\qquad\qquad \to (\forall A\,.\,(A \to A \to Bool) \\ \qquad\qquad\qquad\qquad\qquad\quad \to (GRose\ F\ A \to GRose\ F\ A \to Bool)) \\ equalGRose\ equalF\ equalA\ (gnode\ a_1\ ts_1)\ (gnode\ a_2\ ts_2) \\ \qquad\quad = equalA\ a_1\ a_2 \\ \qquad\qquad \wedge\ equalF\ (equalGRose\ equalF\ equalA)\ ts_1\ ts_2 \end{array}$$

Beachte: die Signatur hat den *Rang* 2 ($equalGRose$ hat ein polymorphes Argument).

Überblick

- √ *Ouvertüre:* Digitale Suchbäume (2–6).
- √ *Intermezzo:* Generische Programmierung (8–22).
- ☐ Generische, digitale Suchbäume (24–36).
- ☐ Schlußbemerkungen (39–40).

Generische, digitale Suchbäume

Wir verwenden die Technik der generischen Programmierung, um folgende Typen und Funktionen zu implementieren:

$$
\begin{array}{ll}
\mathit{Map}\langle K :: \star\rangle :: \star \to \star & \\
\mathit{empty}\langle K\rangle & :: \forall V\,.\,\mathit{Map}\langle K\rangle\ V \\
\mathit{single}\langle K\rangle & :: \forall V\,.\,K \to V \to \mathit{Map}\langle K\rangle\ V \\
\mathit{lookup}\langle K\rangle & :: \forall V\,.\,K \to \mathit{Map}\langle K\rangle\ V \to \mathit{Maybe}\ V \\
\mathit{merge}\langle K\rangle & :: \forall V\,.\,(V \to V \to V) \\
 & \qquad\qquad \to (\mathit{Map}\langle K\rangle\ V \to \mathit{Map}\langle K\rangle\ V \to \mathit{Map}\langle K\rangle\ V).
\end{array}
$$

Beachte: $\mathit{merge}\langle K\rangle\ c\ t_1\ t_2$ vereinigt die Tries t_1 und t_2 (wenn t_1 und t_2 Bindungen für die gleichen Schlüssel enthalten, dann kombiniert c die assoziierten Werte).

Eine generische Definition von Map

Zur Erinnerung: Tries basieren auf den Potenzgesetzen.

$$
\begin{array}{ll}
1 \to_{\mathrm{fin}} V & \cong V \\
(K_1 + K_2) \to_{\mathrm{fin}} V & \cong (K_1 \to_{\mathrm{fin}} V) \times (K_2 \to_{\mathrm{fin}} V) \\
(K_1 \times K_2) \to_{\mathrm{fin}} V & \cong K_1 \to_{\mathrm{fin}} (K_2 \to_{\mathrm{fin}} V)
\end{array}
$$

Da $\mathit{Map}\langle K\rangle\ V$ den Typ $K \to_{\mathrm{fin}} V$ repräsentiert, definieren wir

$$
\begin{array}{ll}
\mathit{Map}\langle K :: \star\rangle & :: \star \to \star \\
\mathit{Map}\langle 1\rangle & = \Lambda V\,.\,\mathit{Maybe}\ V \\
\mathit{Map}\langle K_1 + K_2\rangle & = \Lambda V\,.\,\mathit{Map}\langle K_1\rangle\ V \times \mathit{Map}\langle K_2\rangle\ V \\
\mathit{Map}\langle K_1 \times K_2\rangle & = \Lambda V\,.\,\mathit{Map}\langle K_1\rangle\ (\mathit{Map}\langle K_2\rangle\ V).
\end{array}
$$

Spezialisierung von Map

Die Spezialisierung generischer Definitionen funktioniert in gleicher Weise für generische Datentypen. Für String, Expr und List erhalten wir

$$
\begin{array}{ll}
\mathit{MapS} & = \mathit{Fix}\ (\Lambda \mathit{MS}\,.\,\Lambda V\,.\,\mathit{Maybe}\ V \times \mathit{MapC}\ (\mathit{MS}\ V)) \\
\mathit{MapE} & = \mathit{Fix}\ (\Lambda \mathit{ME}\,.\,\Lambda V\,.\,\mathit{MapS}\ V \times \mathit{ME}\ (\mathit{ME}\ (\mathit{ME}\ V))) \\
\mathit{MapL} & = \Lambda \mathit{MapA}\,.\,\mathit{Fix}\ (\Lambda \mathit{ML}\,.\,\Lambda V\,.\,\mathit{Maybe}\ V \times \mathit{MapA}\ (\mathit{ML}\ V)).
\end{array}
$$

Als Haskell Datentypen:

$$
\begin{array}{ll}
\mathbf{data}\ \mathit{MapS}\ V & = \mathit{nodeS}\ (\mathit{Maybe}\ V)\ (\mathit{MapC}\ (\mathit{MapS}\ V)) \\
\mathbf{data}\ \mathit{MapE}\ V & = \mathit{nodeE}\ (\mathit{MapS}\ V)\ (\mathit{MapE}\ (\mathit{MapE}\ (\mathit{MapE}\ V))) \\
\mathbf{data}\ \mathit{MapL}\ \mathit{MapA}\ V & = \mathit{nodeL}\ (\mathit{Maybe}\ V)\ (\mathit{MapA}\ (\mathit{MapL}\ \mathit{MapA}\ V)).
\end{array}
$$

Eine generische Definition von empty

$$
\begin{array}{ll}
\mathit{empty}\langle K\rangle & :: \forall V\,.\,\mathit{Map}\langle K\rangle\ V \\
\mathit{empty}\langle 1\rangle & = \mathit{nothing} \\
\mathit{empty}\langle K_1 + K_2\rangle & = (\mathit{empty}\langle K_1\rangle, \mathit{empty}\langle K_2\rangle) \\
\mathit{empty}\langle K_1 \times K_2\rangle & = \mathit{empty}\langle K_1\rangle
\end{array}
$$

Beachte: empty verwendet *polymorphe Rekursion*.

Spezialisierung von *empty*—*String*, *Expr* und *List*

$$
\begin{aligned}
&\mathit{emptyS} :: \forall V \,.\, \mathit{MapS}\ V\\
&\mathit{emptyS} = \mathit{nodeS}\ \mathit{nothing}\ \mathit{emptyC}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{emptyE} :: \forall V \,.\, \mathit{MapE}\ V\\
&\mathit{emptyE} = \mathit{nodeE}\ \mathit{emptyS}\ \mathit{emptyE}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{emptyL} &&:: \forall \mathit{MapA} \,.\, (\forall W \,.\, \mathit{MapA}\ W) \to (\forall V \,.\, \mathit{MapL}\ \mathit{MapA}\ V)\\
&\mathit{emptyL}\ \mathit{emptyA} &&= \mathit{nodeL}\ \mathit{nothing}\ \mathit{emptyA}
\end{aligned}
$$

Eine generische Definition von *single*

$$
\begin{aligned}
&\mathit{single}\langle K\rangle &&:: \forall V \,.\, K \to V \to \mathit{Map}\langle K\rangle\ V\\
&\mathit{single}\langle 1\rangle\ ()\ v &&= \mathit{just}\ v\\
&\mathit{single}\langle K_1 + K_2\rangle\ (\mathit{inl}\ k_1)\ v &&= (\mathit{single}\langle K_1\rangle\ k_1\ v, \mathit{empty}\langle K_2\rangle)\\
&\mathit{single}\langle K_1 + K_2\rangle\ (\mathit{inr}\ k_2)\ v &&= (\mathit{empty}\langle K_1\rangle, \mathit{single}\langle K_2\rangle\ k_2\ v)\\
&\mathit{single}\langle K_1 \times K_2\rangle\ (k_1, k_2)\ v &&= \mathit{single}\langle K_1\rangle\ k_1\ (\mathit{single}\langle K_2\rangle\ k_2\ v)
\end{aligned}
$$

Beachte: *single* stützt sich auf *empty* ab.

Spezialisierung von *single*—*String*, *Expr* und *List*

$$
\begin{aligned}
&\mathit{singleS} :: \forall V \,.\, \mathit{String} \to V \to \mathit{MapS}\ V\\
&\mathit{singleS}\ \mathit{nil}\ v\\
&\qquad = \mathit{nodeS}\ (\mathit{just}\ v)\ \mathit{emptyC}\\
&\mathit{singleS}\ (\mathit{cons}\ c\ s)\ v\\
&\qquad = \mathit{nodeS}\ \mathit{nothing}\ (\mathit{singleC}\ c\ (\mathit{singleS}\ s\ v))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{singleE} :: \forall V \,.\, \mathit{Expr} \to V \to \mathit{MapE}\ V\\
&\mathit{singleE}\ (\mathit{var}\ s)\ v\\
&\qquad = \mathit{nodeE}\ (\mathit{singleS}\ s\ v)\ \mathit{emptyE}\\
&\mathit{singleE}\ (\mathit{if}\ e_1\ e_2\ e_3)\ v\\
&\qquad = \mathit{nodeE}\ \mathit{emptyS}\ (\mathit{singleE}\ e_1\ (\mathit{singleE}\ e_2\ (\mathit{singleE}\ e_3\ v)))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{singleL} :: \forall K\ \mathit{MapA} \,.\, (\forall W \,.\, \mathit{MapA}\ W)\\
&\qquad\qquad \to (\forall W \,.\, K \to W \to \mathit{MapA}\ W)\\
&\qquad\qquad \to (\forall V \,.\, \mathit{List}\ K \to V \to \mathit{MapL}\ \mathit{MapA}\ V)\\
&\mathit{singleL}\ \mathit{emptyA}\ \mathit{singleA}\ ([\,], v)\\
&\qquad = \mathit{nodeL}\ (\mathit{just}\ v)\ \mathit{emptyA}\\
&\mathit{singleL}\ \mathit{emptyA}\ \mathit{singleA}\ (k : \mathit{ks}, v)\\
&\qquad = \mathit{nodeL}\ \mathit{nothing}\ (\mathit{singleA}\ (k, \mathit{singleL}\ \mathit{emptyA}\ \mathit{singleA}\ (\mathit{ks}, v)))
\end{aligned}
$$

Eine generische Definition von $lookup$

$$
\begin{array}{ll}
lookup\langle K\rangle & :: \forall V\,.\,K \to Map\langle K\rangle\ V \to Maybe\ V\\
lookup\langle 1\rangle\ ()\ t & = t\\
lookup\langle K_1+K_2\rangle\ (inl\ k_1)\ (t_1,t_2) & = lookup\langle K_1\rangle\ k_1\ t_1\\
lookup\langle K_1+K_2\rangle\ (inr\ k_2)\ (t_1,t_2) & = lookup\langle K_2\rangle\ k_2\ t_2\\
lookup\langle K_1\times K_2\rangle\ (k_1,k_2)\ t_1 & = (lookup\langle K_1\rangle\ k_1 \diamond lookup\langle K_2\rangle\ k_2)\ t_1
\end{array}
$$

Spezialisierung von $lookup$—$String$, $Expr$ und $List$

$$
\begin{array}{l}
lookupS\ :: \forall V\,.\,String \to MapS\ V \to Maybe\ V\\
lookupS\ nil\ (nodeS\ tn\ tc)\\
\qquad = tn\\
lookupS\ (cons\ c\ s)\ (nodeS\ tn\ tc)\\
\qquad = (lookupC\ c \diamond lookupS\ s)\ tc
\end{array}
$$

$$
\begin{array}{l}
lookupE\ :: \forall V\,.\,Expr \to MapE\ V \to Maybe\ V\\
lookupE\ (var\ s)\ (nodeE\ tv\ ti)\\
\qquad = lookupS\ s\ tv\\
lookupE\ (if\ e_1\ e_2\ e_3)\ (nodeE\ tv\ ti)\\
\qquad = (lookupE\ e_1 \diamond lookupE\ e_2 \diamond lookupE\ e_3)\ ti
\end{array}
$$

Beachte: $lookupE$ verwendet *polymorphe Rekursion*.

$$
\begin{array}{l}
lookupL :: \forall K\ MapA\,.\,(\forall W\,.\,K \to MapA\ W \to Maybe\ W)\\
\qquad\qquad \to (\forall V\,.\,List\ K \to MapL\ MapA\ V\\
\qquad\qquad\qquad \to Maybe\ V)\\
lookupL\ lookupA\ [\,]\ (nodeL\ tn\ tc)\\
\qquad = tn\\
lookupL\ lookupA\ (k : ks)\ (nodeL\ tn\ tc)\\
\qquad = (lookupA\ k \diamond lookupL\ lookupA\ ks)\ tc
\end{array}
$$

Eine generische Definition von $merge$

$$
\begin{array}{l}
merge\langle K\rangle\ :: \forall V\,.\,(V \to V \to V)\\
\qquad\qquad \to (Map\langle K\rangle\ V \to Map\langle K\rangle\ V \to Map\langle K\rangle\ V)\\
merge\langle 1\rangle\ c\ t\ t' \qquad = combine\ c\ t\ t'\\
merge\langle K_1+K_2\rangle\ c\ (t_1,t_2)\ (t_1',t_2')\\
\qquad\qquad = (merge\langle K_1\rangle\ c\ t_1\ t_1', merge\langle K_2\rangle\ c\ t_2\ t_2')\\
merge\langle K_1\times K_2\rangle\ c\ t\ t' = merge\langle K_1\rangle\ (merge\langle K_2\rangle\ c)\ t\ t',
\end{array}
$$

wobei $combine$ wie folgt definiert ist

$$
\begin{array}{ll}
combine\ c\ nothing\ nothing & = nothing\\
combine\ c\ nothing\ (just\ v_2) & = just\ v_2\\
combine\ c\ (just\ v_1)\ nothing & = just\ v_1\\
combine\ c\ (just\ v_1)\ (just\ v_2) & = just\ (c\ v_1\ v_2).
\end{array}
$$

Spezialisierung von $merge$—$String$, $Expr$ und $List$

$$
\begin{aligned}
&mergeS :: \forall V\,.\,(V \to V \to V)\\
&\qquad\qquad\to (MapS\ V \to MapS\ V \to MapS\ V)\\
&mergeS\ c\ (nodeS\ tn\ tc)\ (nodeS\ tn'\ tc')\\
&\qquad = nodeS\ (combine\ c\ tn\ tn')\\
&\qquad\qquad (mergeC\ (mergeS\ c)\ tc\ tc')
\end{aligned}
$$

$$
\begin{aligned}
&mergeE :: \forall V\,.\,(V \to V \to V)\\
&\qquad\qquad\to (MapE\ V \to MapE\ V \to MapE\ V)\\
&mergeE\ c\ (nodeE\ tv\ ti)\ (nodeE\ tv'\ ti')\\
&\qquad = nodeE\ (mergeS\ c\ tv\ tv')\\
&\qquad\qquad (mergeE\ (mergeE\ (mergeE\ c))\ ti\ ti')
\end{aligned}
$$

$$
\begin{aligned}
&mergeL :: \forall MapA\,.\,(\forall W\,.\,(W \to W \to W)\\
&\qquad\qquad\qquad\qquad \to (MapA\ W \to MapA\ W \to MapA\ W))\\
&\qquad\qquad\qquad \to (\forall V\,.\,(V \to V \to V)\\
&\qquad\qquad\qquad\qquad \to (MapL\ MapA\ V \to MapL\ MapA\ V\\
&\qquad\qquad\qquad\qquad\qquad \to MapL\ MapA\ V))\\
&mergeL\ mergeA\ c\ (nodeL\ tn\ tc)\ (nodeL\ tn'\ tc')\\
&\qquad = nodeL\ (combine\ c\ tn\ tn')\\
&\qquad\qquad (mergeA\ (mergeL\ mergeA\ c)\ tc\ tc')
\end{aligned}
$$

Überblick

√ *Ouvertüre:* Digitale Suchbäume (2–6).

√ *Intermezzo:* Generische Programmierung (8–22).

√ Generische, digitale Suchbäume (24–36).

□ Schlußbemerkungen (39–40).

Schlußbemerkungen

Wir sind gewohnt,

□ Werte zu definieren, die mit Werten parametrisiert sind (sogenannte *Funktionen*),

□ Typen zu definieren, die mit Typen parametrisiert sind (sogenannte *Typkonstruktoren*).

Die generische Programmierung bietet darüber hinaus die Möglichkeit,

□ Werte durch Induktion über die Struktur von Typen zu definieren (sogenannte *generische Funktionen*),

□ Typen durch Induktion über die Struktur von Typen zu definieren (sogenannte *generische Typen*).

Und sonst?

- Generische Beweise: Fixpunktinduktion, „logical relations“.
- Generische Spezifikation.
- Implementierung: Glasgow Haskell Compiler.