Betreuer: Prof. Dr. Michael Hanus
Autor: Daniel Jung

Seminar: Context Patterns in Haskell

Verwendete Literatur:

1. Markus Mohnen, Context Patterns in Haskell, in Selected Papers from the 8th International Workshop on Implementation of Functional Languages, LNCS 1268, Springer, 1996

2. Markus Mohnen, Context Patterns, Part II, in 1997 International Workshop on Implementation of Functional Languages, September 10th - 12th 1997, St. Andrews, Scotland

3. `http://www.cse.unsw.edu.au/~chak/haskell/ghc/comm/`

4. `http://www.haskell.org/onlinereport/index98.html`

5. `http://www-i2.informatik.rwth-aachen.de/Staff/Current/mohnen/CP/`

letzte Änderung: 16. Januar 2006

1 Einleitung

In Haskell wird für die Selektion von Datenstrukturen Pattern-Matching verwendet. Dabei wird auf den Datenkonstruktor gematched und Ausdrücke können an Variablen gebunden werden. Dies hat den Vorteil, dass mit Definition einer Datenstruktur gleichzeitig eine einfache Möglichkeit der Selektion zur Verfügung steht.
Soll in der Nähe der Wurzel gematched werden, ist dies mit einer eleganten und kurzen Deklaration möglich (hier als Beispiel anhand einer Liste):

```
-- (1)
head' :: [a] -> a
head' [] = error "Die Liste ist leer!"
head' ( v : _ ) = v
```

Wie man sieht, ergibt sich eine kurze und klare Definition und es besteht ein direkter Zusammenhang zu der Datenstruktur des Beispiels. Daher ist es ebenso einfach, ein Element an der Wurzel mittels des entsprechenden Konstruktors einzufügen:

```
-- (2)
insertHead :: [a] -> a -> [a]
insertHead vs v = v : vs
```

Ein Nachteil ist jedoch, dass die Selektion damit von der Art wie eine Datenstruktur erstellt wird, abhängt.
Somit kann nur in einem festen Bereich um die Wurzel der Datenstruktur direkt gematched werden. Ein Binden des Kontexts ist ebenfalls nicht, oder nur eingeschränkt, möglich. Soll in einem unbestimmten Abstand oder fern der Wurzel gematched werden, muss in der Regel die Datenstruktur rekursiv durchlaufen werden:

```
-- (3)
last' :: [a] -> a
last' [] = error "Die Liste ist leer!"
last' ( v : [] ) = v
last' ( _ : vs ) = last' vs
```

Eine Lösung für dieses Problem bieten Context Pattern. Sie erlauben ein Matching von Pattern unabhängig von der Distanz zur Wurzel. Im Folgenden werden die Vorzüge von Context Pattern und deren mögliche Integration in Haskell vorgestellt, sowie deren Funktionsweise erläutert.

2 Grundlagen

2.1 Pattern-Matching

In Haskell wird Pattern-Matching zur Selektion verwendet. Man unterscheidet folgende Standard-Pattern:

- v: eine Variable als Pattern matched immer. Die Variable v wird dabei an den aktuellen Ausdruck gebunden.
- Cons $p_1 ... p_n$: hierbei ist Cons ein n-stelliger Konstruktor und p_1 bis p_n sind die jeweiligen Pattern, die auf die entsprechende Stelle im Konstruktor matchen sollen. Hat der aktuelle Ausdruck die Form (Cons $t_1 ... t_n$) und lassen sich alle Pattern p_1 bis p_n erfolgreich mit t_1 bis t_n matchen, so matched das gesamte Pattern, ansonsten schlägt das Matching fehl.
- _: oder "wildcard". Dieses Pattern matched immer, es findet jedoch keine Bindung des Ausdrucks statt.
- v@p: oder "as-pattern". Matched das Pattern p, so wird v an diesen Ausdruck gebunden.
- n + k: matched, falls der Ausdruck größer oder gleich k ist. n wird dabei an den Wert des Ausdrucks minus k gebunden.
- ~p: dieses Pattern matched immer. ~p wird als Lazy-Pattern bezeichnet und erst beim Zugriff auf das Argument überprüft. Wenn also eine Variable aus dem Pattern p später genutzt wird, so wird dieser Teil an den Ausdruck gebunden, an den er gebunden wäre, wenn p sofort gematched hätte.

n + k Pattern werden bei der Einführung der Context Pattern nicht berücksichtigt. Das Pattern-Matching in Haskell unterliegt folgenden Einschränkungen:

1. Als erstes ist festzustellen, dass die Stelligkeit des Konstruktors mit der Anzahl der Pattern übereinstimmen muss. Insbesondere kann nicht gegen einen partiell applizierten Konstruktor gematched werden.
2. Des weiteren müssen Pattern linear sein. Folgende Definition ist zum Beispiel nicht zulässig:

   ```
   f (p, p) = p
   ```

Die Erweiterung um Context Pattern wird zu einem späteren Zeitpunkt vorgestellt.

Pattern können in Lambda Abstraktionen, Funktionsdefinitionen, Pattern-Bindungen, List Comprehensions und do- oder case-Ausdrücken vorkommen. Da sich jedoch alle Fälle in case-Ausdrücke übersetzen lassen, ist es ausreichend, nur die Semantik von case-Ausdrücken zu betrachten.

2.2 Guards

Guards sind Bedingungen, die in Haskell standardmäßig am Ende von Pattern erlaubt sind. Damit die Regel einer Funktionsdefinition angewendet wird, muss nach dem Matchen des Patterns diese Bedingung zusätzlich erfüllt sein. Wird kein Guard angegeben, ist von einem Guard mit der Bedingung `True` auszugehen.

2.3 Semantik des case-Ausdrucks

Im Folgenden wird kurz auf die Semantik von case-Ausdrücken eingegangen. Der Verständlichkeit halber wird bei den Erläuterungen nicht zwischen den neuen Variablen und dem ursprünglichen Wert unterschieden, z. B. wird im Fall von: `let {y = e0}` weiterhin von e_0 gesprochen[1].

```
(a) case e0 of {p1 mat1; ...; pn matn}
    = case e0 of {p1 mat1;
                  _ -> ... case e0 of {pn matn;
                                       _ -> error "No match"} ...}
```

wobei jedes mat_i folgende Form hat:

```
| gi,1 -> ei,1; ...; | gi,mi -> ei,mi where {decls}
```

Regel (a) zerlegt case-Ausdrücke, so dass sie sequentiell von (b) verarbeitet werden können. Als Alternative e' [siehe (b)] wird dabei das jeweils nächste Pattern übergeben, falls das gerade zu matchende Pattern fehl schlägt. Nachdem alle Pattern fehlgeschlagen sind, wird als letzte Alternative der Fehler: `error "No Match"` angehängt. Enthält ein mat_i keine Guards, wird *True* für die Guards $g_{i,j}$ substituiert.

```
(b) case e0 of {p | g1 -> e1; ... | gn -> en where {decls}
                _                  -> e'}
    = let {y = e'}
      in case e0 of {p -> let {decls}
                          in if g1 then e1...
                                   else if gn then en else y
                     _ -> y}
```

wobei y eine neue Variable ist.

Regel (b) entfernt zusätzliche Guards. Es bleibt noch zu prüfen, ob das Pattern p auf e_0 matched. Ist dies der Fall, werden die Guards durch eine if - then - else Struktur überprüft. Schlägt das Matchen des Patterns fehl oder liefert keiner der Guards *True*, wird die Alternative e' ausgewertet.

```
(c) case e0 of { ~p -> e; _ -> e'}
    = let {y = e0}
      in let {x1' = case y of {p -> x1}}
         in ... let {xn' = case y of {p -> xn}}
                in e[x1'/x1, ..., xn'/xn]
```

[1] vgl. `http://haskell.org/onlinereport/exps.html#sect3.17.3`
`Abschnitt: 3.17.3 Formal Semantics of Pattern Matching`

wobei x_1, ..., x_n Variablen in p und y, x_1', ..., x_n' neue Variablen sind.

In Regel (c) werden Lazy-Pattern ersetzt. Dabei werden die Variablen x_1 bis x_n sukzessive von links nach rechts gebunden, wenn das Pattern matched. Ansonsten wird über (a) → (b) [mit Guard = `True`] beim tatsächlichen Binden (Verwendung der gebundenen Variablen) eine Fehlermeldung erzeugt.

```
(d) case e0 of {x@p -> e; _ -> e'}
    = let {y = e0} in case y of {p -> (\x -> e) y; _ -> e'}
```

wobei y eine neue Variable ist.

Regel (d) behandelt as-Pattern. Es wird überprüft, ob sich das Pattern p auf e_0 matchen lässt. Ist dies der Fall, wird jedes Vorkommen von x in e durch e_0 ersetzt. Konnte p nicht auf e_0 gematched werden, wird die Alternative e' ausgewertet.

```
(e) case e0 of {_ -> e; _ -> e'} = e
```

Das Matchen eines Wildcard-Patterns ist immer erfolgreich und es wird nichts gebunden.

```
(f) case e0 of {K p1 ... pn -> e; _ -> e'}
    = let {y = e'}
      in case e0 of {K x1 ... xn -> case x1 of {
                                      p1 -> ... case xn of {pn -> e;
                                                            _  -> y}
                                      _  -> y}
                    _            -> y}
```

mindestens ein p_i ist keine Variable; y, x_1, ...,x_n sind neue Variablen.

Regel (f) behandelt geschachtelte Pattern. Dabei wird von links nach rechts vorgegangen, d. h., es wird zuerst überprüft, ob sich Pattern p_1 auf die erste Stelle von K matchen lässt. Schlägt dies fehl, schlägt der gesamte matching Prozess fehl und es wird die Alternative e' ausgewertet. Ansonsten wird mit dem nächsten Pattern fortgefahren, bis alle Pattern überprüft worden sind. Lässt sich der entstandene Ausdruck auf e_0 matchen, wird e ausgewertet; schlug das Matching fehl, kommt die Alternative e' zur Auswertung.

```
(g) case e0 of {x -> e; _ -> e'} = case e0 of {x -> e}
```

In Regel (g) wird überprüft, ob sich x auf e_0 matchen lässt und dementsprechend e ausgewertet.

(h) `case` e_0 `of {x -> e} = (\x -> e)` e_0

Regel (h) ersetzt schließlich alle Vorkommen von x in e durch e_0.

3 Context Pattern

3.1 Die Einführung von Context Pattern

Die Motivation, Context Pattern einzuführen war, eine einfache Form des Pattern-Matching bereitzustellen, unabhängig von der Entfernung zur Wurzel. Daher soll noch einmal das Beispiel (3) aus der Einleitung betrachtet werden:

```
last' :: [a] -> a
last' [] = error "Die Liste ist leer!"
last' ( [v] ) = v
last' ( _ : vs ) = last' vs
```

Eine einfache Formulierung, was die Funktion last' leisten soll wäre:
Überprüfe, ob die Liste leer ist. Wenn nicht, gebe das letzte Element zurück.

Mit Context Pattern lässt sich die Funktion last' analog zu dieser Formulierung schreiben als:

```
-- (4)
lastCP :: [a] -> a
lastCP [] = error "Die Liste ist leer!"
lastCP ( c [v] ) = v
```

c wird dabei an den Kontext gebunden. Erhält die Funktion `lastCP` also als Argument eine Liste aus n Elementen [a_1, ..., a_n] mit $n \in \mathbb{N}_{>1}$, so wird \x → (a_1: ...(a_{n-1}:x) ...) an den Kontext `c` und `v` an a_n gebunden (siehe Abb. 1).

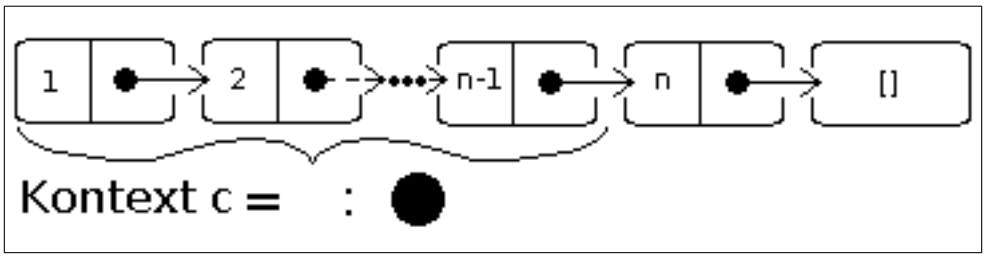

Abbildung 1: Binden des Kontexts

Context Pattern haben dabei folgenden Syntax:

cpat → var pat_1 ... pat_k

Ein Context Pattern matched also einen Wert `v`, wenn es eine Funktion f und Werte $\texttt{v}_1, \ldots, \texttt{v}_k$ gibt, so dass pat_i auf $\texttt{v}_i$ matched und gilt:

$$(*)\ f\ \texttt{v}_1\ \ldots\ \texttt{v}_k\ \texttt{=}\ \texttt{v}$$

Diese Funktion f repräsentiert einen Konstruktor-Kontext, indem die einzelnen Argumente des Konstruktors durch die Funktionsargumente modelliert werden. Allgemein hat die Funktion f daher folgende Form:

$$f = \lambda h_1 ... \lambda h_k . C[h_1, ..., h_k]$$

Dabei ist C ein Konstruktorterm und $h_1, ..., h_k$ sind die benötigten "Löcher", die zum modifizieren der Datenstruktur nötig sind. Werden der Funktion f nun als Argumente die Werte $\texttt{v}_1, \ldots, \texttt{v}_k$ übergeben, nachdem alle pat_i auf $\texttt{v}_i$ gematched wurden, so ist die Forderung $(*)$ erfüllt, da:

$$(\lambda h_1 ... \lambda h_k . C[h_1, ..., h_k])\texttt{v}_1\ \ldots\ \texttt{v}_k \rightsquigarrow C\ [\text{v}_1, ..., \text{v}_k]$$

Aus diesem Grund wird die Funktion f an `v` gebunden, falls alle Pattern pat_i auf $\texttt{v}_i$ matchen.

In unserem Beispiel entstehen also folgende Bindungen:

$$[\text{v}/\text{a}_n,\ \text{c}/\backslash\text{h->}(\text{a}_1\text{: ...}(\text{a}_{n-1}\text{:h})\ ...)]$$

Der Konstruktorterm hat in diesem Fall ein Loch, durch das die Datenstruktur modifiziert werden kann. Um die ursprüngliche Liste zu erhalten, kann `c` als Argument z. B. `[v]` übergeben werden.
Viel interessanter ist jedoch, dass dem Kontext `c` auch ein anderes Argument übergeben werden kann. Damit lässt sich durch Modifikation der letzten Zeile in dem Beispiel (4) `lastCP` die Funktion verändern (der Rückgabetyp der Funktion ändert sich dabei zu [a]):

```
-- (5)
lastCP' ( c [v] ) = c []
```

Nun gibt die Funktion die Liste zurück, der das letzte Element entnommen wurde.

```
-- (6)
lastCP'' ( c [v] ) = c [v, v]
```

Hier wurde das letzte Element der Liste verdoppelt. An diesen Beispielen sieht man, das Context Pattern auch zum transformieren von Datenstrukturen gut geeignet sind.

3.2 Eigenschaften von Context Pattern

Bei der Einführung der Context Pattern entsteht ein Konflikt. So könnte der Ausdruck:

```
let x (y:ys) = e1 in e2
```

sowohl eine Definition der Funktion `x` sein, die das Pattern `(y:ys)` nutzt, als auch ein Pattern-Matching auf das Ergebnis von e_1, wobei `x` ein Context Pattern wäre. Dieser Konflikt wird gelöst, indem Funktionsdefinitionen bevorzugt werden.

Zusätzlich zu den in 2.1 erwähnten Einschränkungen für Pattern in Haskell wird für Context Pattern gefordert:

3. Wenn eine Kontext-Variable *var* vom funktionalen Typ $t_1 \rightarrow ... \rightarrow t_k \rightarrow t$ ist, dann muss ein Wert v des Typs t und Werte v_i des Typs t_i existieren, so dass alle v_i unabhängige Argumente von v sind und in der Reihenfolge v_1, ..., v_k in einem oben $\rightarrow$ unten, links $\rightarrow$ rechts[2] Durchlauf von v vorkommen.

Diese Forderung folgt direkt aus 3.1 ($*$) und sichert zu, dass das Pattern mindestens einen Wert matchen kann. Gäbe es ein t_i, das kein Argument von v sein kann, gäbe es keinen Wert v des Typs t mit dem Argument v_i des Typs t_i und das Pattern könnte niemals matchen.
Die Unabhängigkeit der Argumente muss ebenfalls gefordert werden, da nach dem erfolgreichen Matchen eines Arguments nicht überprüft wird, ob ein Teil des gebundenen Bereichs vielleicht auch dem nächsten Argument zugeordnet werden könnte. Folgendes Beispiel wird dies verdeutlichen:

```
-- (7)
foo :: [a] -> [a]
foo (c (x:xs) (y:ys)) = e
```

Diese Deklaration ist nicht zulässig und verstößt gegen die Unabhängigkeit der Argumente. Nach dem Matchen von `(x:xs)` wäre die Eingabe bereits vollständig "verbraucht". `a` würde dabei an das erste Element der Liste gebunden sein und `xs` an die Restliste. Es wird nicht überprüft, ob sich Teile aus `xs` sich dem nächsten Argument `(y:ys)` zuordnen lassen. Außerdem kann eine Liste keine zwei, nicht überlappenden Teillisten enthalten; es ließe sich also auch kein Konstruktor für dieses Pattern finden.

3.2.1 Context-Wildcards

Wie auch im Haskell Standard-Pattern-Matching ist es möglich, Context-Wildcards zu verwenden, wenn der Kontext auf der rechten Seite der Funktionsdefinition nicht benutzt wird. In dem anfangs vorgestelltem Beispiel wird der Kontext nicht benötigt. Daher lässt sich die Funktion (4) auch wie folgt modifizieren:

[2] vgl. 3.2.3 und Abb. 2

```
lastCP ( _ [v] ) = v
```

"_" ist dabei immer noch ein Context Pattern; es findet jedoch keine Bindung des Kontexts statt.

3.2.2 Guards

Im Gegensatz zu den Haskell Standard-Pattern ist es für Context Pattern nicht ausreichend, Guards nur am Ende der Pattern zur Verfügung zu stellen. Dies wird klar, wenn man bedenkt, dass Context Pattern einer rekursiven Suche entsprechen. Ließe man Guards nur am Ende von Context Pattern zu, würde der Guard erst *nach* dem Ende der Suche wirksam.
Deutlich wird dies an folgendem Beispiel:

```
-- (8)
member :: Eq a => a -> [a] -> Bool
member _ [] = False
member y (x:xs)
 | y == x = True
 | otherwise = member y xs
```

Die Funktion `member` überprüft, ob ein Element in einer Liste enthalten ist und gibt einen Wahrheitswert zurück. Guards werden nur am Ende der Pattern verwendet. Werden Context Pattern verwendet und lässt man Guards nur am Ende der Pattern zu, so verändert man das Verhalten der Funktion:

```
-- (9)
memberCP' y (_ (x:xs)) | y == x = True
memberCP' _ _ = False
```

Hat die Liste mindestens ein Element, ist das Matchen erfolgreich und die Variablen werden gebunden. Dann wird überprüft, ob das erste Element gleich `y` ist. Ist dies der Fall, wird *True* zurück gegeben, ansonsten schlägt das matchen der Regel fehl, da der Guard *False* liefert, und es wird die zweite Funktionsdefinition gewählt, die *False* zurück gibt. Insbesondere wird die Liste nicht weiter durchsucht.

Um diesen Mangel zu beheben, wurden die Context Pattern um Guards erweitert. Guards innerhalb von Context Pattern werden *nicht* durch "|", sondern durch "if (Bedingung)" deklariert. Jetzt lässt sich die Funktion `member` wie folgt definieren:

```
-- (10)
memberCP y (c (x:xs) if x==y ) = True
memberCP _ _ = False
```

Die Überprüfung, ob der Guard erfüllt ist, findet jetzt statt *bevor* die Variablen gebunden werden. Es wird also in der Liste gesucht, ob es ein Element gibt, das gleich `y` ist. Gibt es ein solches und matched das Pattern, werden die Variablen gebunden und `True` zurück gegeben, ansonsten schlägt die Regel wie zuvor fehl.
Wird die Funktion `memberCP` zum Beispiel mit folgenden Parametern aufgerufen, entstehen bei erfolgreichem Matchen die Bindungen (es sei y = a_i und y $\neq$ a_q, für alle $q_{\in}\{1, ..., (i-1)\}$ vorausgesetzt):

`memberCP y [`a_1`, ..., `a_{i-1}`, `a_i`, `a_{i+1}`, ...,`a_n`]`
`x`/a_i, `xs`/[a_{i+1}, ..., a_n] und `c`/(\h->(a_1: ...(a_{i-1}:h) ...))

Es wird also das erste Pattern gematched, das den Guard erfüllt[3].
In Kapitel 3.6.1 wird die Funktion der Guards und die Funktionsweise der Context Pattern noch einmal ausführlich geschildert.

3.2.3 Explizite Typdeklaration

Im Gegensatz zu den Haskell Standard-Pattern ermöglichen Context Pattern eine direkte Typdeklaration innerhalb der Pattern. Um dies zu veranschaulichen wird die Datenstruktur eines Baums betrachtet, die eine Liste von Attributen und eine Liste von Nachfolgern hat:

```
data Tree a = Node [a] [Tree a]
```

Nun soll zum Beispiel das letzte Element einer nicht leeren Nachfolgerliste gematched werden. Ein erster Ansatz ohne Typdeklaration wäre Beispielsweise:

```
-- (11)
lastSuc :: Tree a -> Tree a
lastSuc (c1 (Node as (c2 [s]))) = s
```

Dabei hat der Kontext `c1` den Typ `Tree a` → `Tree a` und der Kontext `c2` ist vom Typ `[Tree a]` → `[Tree a]`.
Schränkt man den Typ innerhalb des Context Patterns ein, lässt sich die Funktion noch eleganter und klarer definieren:

```
-- (12)
lastSuc' (c [s]::[Tree a] ) = s
```

Durch die Einschränkung auf einen Typen wird eindeutig festgelegt, auf welche Liste gematched werden soll. Ohne diese Einschränkung des Typs würde auf die erste Liste, also die Liste der Attribute, gematched. Der Kontext `c` ist nun vom Typ `[Tree a]` → `Tree a` und enthält die gesamte Datenstruktur, bis auf den Teil, der an `s` gebunden wurde und einen fehlenden Listenkonstruktor.
Nun ist auch klar, warum in 3.2 die Durchlaufrichtung festgelegt wurde, denn der Kontext kann unter Umständen sehr komplexe Strukturen enthalten[4].

[3] vgl. 3.3 Eindeutigkeit und Vollständigkeit der Lösung
[4] vgl. 3.3 Eindeutigkeit und Vollständigkeit der Lösung

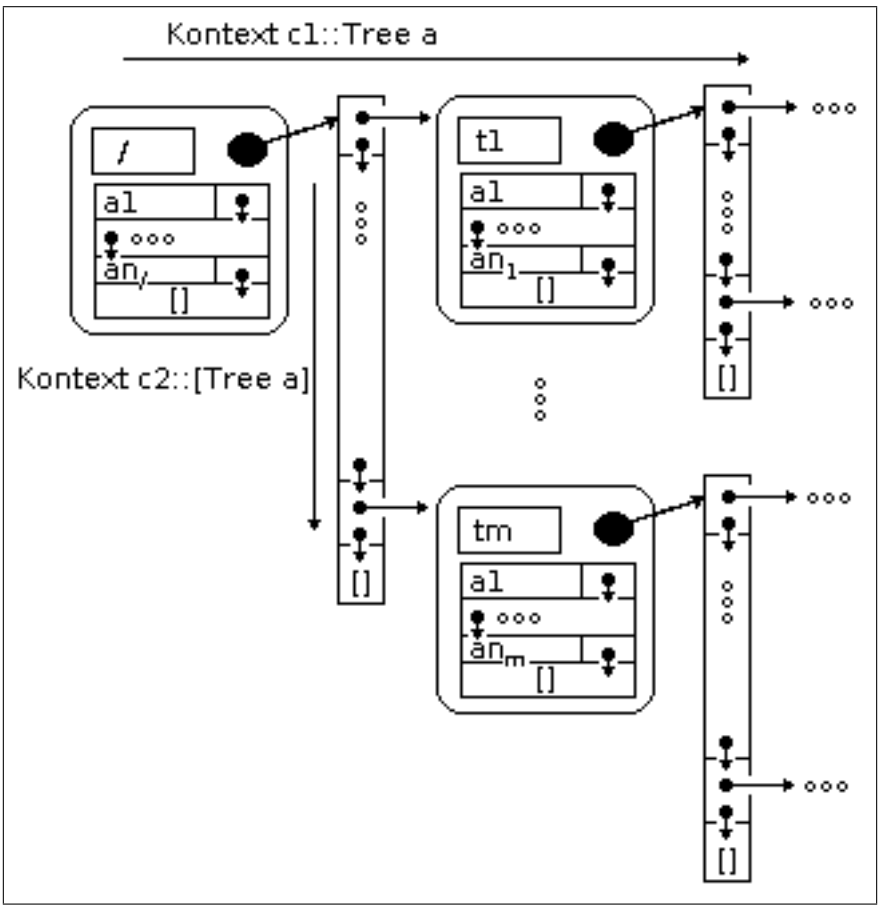

Abbildung 2: Veranschaulichung der Datenstruktur: `Tree`

3.2.4 Typinferenz

Während der Typinferenz wird jedem Argument der Kontext-Funktion ein vorläufiger Typ zugeordnet. Abhängigkeiten zwischen den Typen werden vermerkt und genutzt um den Typ des Arguments näher zu bestimmen. Kontext-sensitive Bedingungen können an dieser Stelle zusätzliche Abhängigkeiten schaffen. Angewandt auf das Beispiel (6) (`lastCP'' :: [a] → [a]`) ergeben sich durch Typinferenz für den Fall:

```
lastCP'' ( c [v] ) = c [v, v]
```

die vorläufigen Typen:

- `a` für das Context Pattern `c [v]`
- `[b] → a` für die Kontext-Funktion `c`
- `b` für die Variable `v`

Um die zusätzliche Bedingung für Context Pattern zu erfüllen[5], werden die Typen `[b]` und `a` unifiziert. Daraus ergeben sich dann folgende Typen:

[5] vgl. 3. in: 3.2 Eigenschaften von Context Pattern

- `[b]` für das Context Pattern `c [v]`
- `[b]` → `[b]` für die Kontext-Funktion `c`
- `b` für die Variable `v`

Insbesondere wird hier auch Bedingung 3 (aus 3.2 Eigenschaften von Context Pattern) überprüft. Dabei wird getestet, ob es möglich ist einen Wert v zu konstruieren, so dass wenn eine Kontext-Variable *var* vom funktionalen Typ $t_1 \rightarrow ... \rightarrow t_k \rightarrow t$ ist, v den Typ t hat und Werte v_i des Typs t_i existieren, so dass alle v_i unabhängige Argumente von v sind und in der Reihenfolge $v_1, ..., v_k$ in einem oben → unten, links → rechts Durchlauf von v vorkommen.
Für Context Pattern wird Bedingung 3 mittels folgender Inferenzregeln überprüft:

$$\frac{}{A \vdash t \rightarrow (t)} \text{ REF}$$

$$\frac{A(con) = t_1 \rightarrow ... \rightarrow t_n \rightarrow t}{A \vdash t \rightarrow (t_1, ..., t_n)} \text{ TYCON}$$

$$\frac{A \vdash t \rightarrow (t_1, ..., t_n) \;\; A \vdash t_i \rightarrow (t_{i,1}, ..., t_{i,m_i})(1 \leq i \leq n)}{A \vdash t \rightarrow (t_{1,1}, ..., t_{1,m_1}, ..., t_{n,1}, ...t_{n,m_n})} \text{ COMB}$$

REF handhabt dabei Typvariablen und Funktionstypen. In beiden Fällen wird der Ausdruck nicht weiter untersucht, da für Typvariablen keine Bindungen auftreten und die Argumente von Funktionstypen nicht durchsucht werden können.
TYCON steht für die Verwendung eines Typkonstruktors. Für jeden Konstruktor eines Typs wird diese Regel somit einmal angewendet, um alle Möglichkeiten abzudecken, den benötigten Typ zu erstellen.
COMB kombiniert die (Teil-)Ergebnisse der Konstruktoren.

Für den Typ t mit den entsprechenden Typvariablen t_1, ..., t_k, der den Typkonstruktor *TYCON* verwendet, lässt sich Bedingung 3 folgendermaßen formulieren:

$$A \vdash t \rightarrow (t_1, ..., t_k)$$

wobei A eine Umgebung für (Daten-) Konstruktoren und (t_1, ..., t_k) eine Liste unabhängiger Argumentetypen von t ist, die in einem oben → unten, links → rechts Durchlauf vorkommen.
Kann nun für eine Kontextvariable des Typs t_1 -> ... → t_k → t aus $A \vdash t \rightarrow (t_1, ..., t_k)$ $A \vdash t \rightarrow (t_1, ..., t_k, t_{k+1}, ..., t_{k+l})$ abgeleitet werden, erfüllt das Context Pattern Bedingung 3.

3.2.5 Abstrakte Grenzen

Da Context Pattern Funktionen zur Modellierung des Kontextes nutzen, können sie abstrakte Datentypen nur durchsuchen, wenn auch andere Funktionen von außen auf diese zugreifen können. Daher ist es Context Pattern nicht möglich, abstrakte Grenzen zu umgehen. Gleiches gilt für polymorphe Komponenten.

3.3 Eindeutigkeit und Vollständigkeit der Lösung

Wie zuvor bereits erwähnt, ist das matchen von Context Pattern gewissen Regeln unterworfen. Dazu soll noch einmal kurz das Beispiel (9) `memberCP'` betrachtet werden:

```
memberCP' :: Eq a => a -> [a] -> Bool
memberCP' y (c (x:xs)) | y == x = True
memberCP' _ _ = False
```

In 3.2.2 wurde festgestellt, dass diese Funktion das erste Element der Liste mit `y` vergleicht. Dabei wäre es auch durchaus denkbar, einen Teil der Liste in dem Context Pattern `c` zu matchen, solange eine Liste mit mindestens einem Element übrig bleibt, um von `(x:xs)` gematched zu werden. Daraus ließe sich dann eine Liste *aller* möglichen Matches erstellen und man erhielte die Vollständigkeit der Lösung. Diese Liste würde in Haskell natürlich auch lazy berechnet und damit nur so weit wie benötigt ausgewertet. Ein weiterer Vorteil neben der Vollständigkeit der Lösung wäre, dass Context Pattern ohne Guards auskämen. So ließe sich in dem obigen Beispiel auf der rechten Seite überprüfen, ob die Ergebnisliste für `x` das gesuchte Element `y` enthält (auch wenn dann in *diesem Beispiel* die Funktion `memberCP'` überflüssig wäre).

Dieser Ansatz wurde aus den folgenden Gründen jedoch nicht gewählt:

- Die Unvollständigkeit ist auch in den Haskell Pattern vorhanden. Dies wird an einem kurzen Beispiel deutlich:
  Gegeben sei eine Liste vom Typ `(Int, Char)`, also z. B.:
  `(2,'a'):(2,'a'):[]`
  Sollen beispielsweise alle Zeichen zum Integerwert `2` ausgegeben werden, so ist die Lösung unvollständig, da die Pattern in Haskell in einem oben → unten, links → rechts Durchlauf matchen und somit das Pattern `((i, c):ics)` das erste Element der Liste zuerst matched. (Ergebnis: `"ab"`).

  ```
  two :: [(Int, Char)] -> [Char]
  two []           = []
  two ((i, c):ics) | i == 2    = c : (two ics)
                   | otherwise = two ics
  ```

  In Abhängigkeit von der Implementierung mag als Ergebnis auch `"ba"` herauskommen, die vollständige Lösung muss jedoch beide Möglichkeiten enthalten.

- Die Variablen wären auf der rechten Seite nicht mehr frei verwendbar, sondern müssten z. B. über eine Ergebnisliste extrahiert werden. Für das Beispiel ergäbe sich vielleicht folgender Ergebnistyp: `[(Typ y, Typ c, Typ x, Typ xs)]`. Damit ist der Zugang zu den gebunden Werten jedoch wieder kompliziert geworden und es entstehen weitere Unklarheiten, wie z. B. die Reihenfolge der Bindungen.

Statt dessen wurde sich an den Haskell Standard-Pattern orientiert. Es wird jeweils der erste mögliche Match eines oben → unten, links → rechts Durchlaufs gewählt[6]. Demzufolge kann das Matching in linearer Zeit und in Abhängigkeit von der Größe des Werts erfolgen. In dem Beispiel `memberCP'` bedeutet dies, dass folgende Bindungen entstehen:

`memberCP' y' [`a_1`, `a_2`, ..., `a_n`]`
`y`/`y'`, `x`/a_1, `xs`/`[`a_2`, ...,`a_n`]` und `c`/(\h->h)

Somit wahrt die Lösung die laziness, sie ist eindeutig (das erste mögliche Match bei fester Durchlaufreihenfolge und -richtung) und *nicht* vollständig.

3.4 Eine Erweiterung der Context Pattern: der Extended Context

Angenommen, es soll in einer Liste aus Integern jeder Wert, der gleich *sieben* ist, mit *der Anzahl der bis dahin gefundenen siebenen* addiert werden.

```
-- (13)
modSeven :: Int -> [Int] -> [Int]
modSeven o (c 7) = modSeven (o+1) (c (7+o))
modSeven _ l = l
```

Bei dieser Implementierung wird nach jedem Match auf `7` die Liste modifiziert, dann aber *erneut am Anfang der Liste* begonnen, anstatt mit dem nächsten Element fort zu fahren. Daraus resultiert quadratische Komplexität, obwohl die Aufgabe nur lineare Komplexität benötigt. Aus diesem Grund werden die Context Pattern um den Extended Context `c_e` erweitert. Dazu wird auf der rechten Seite der Funktionsdefinition zusätzlich zu dem (hier allgemeinen) Kontext `c` vom Typ $\texttt{t}_1 \rightarrow \ldots \rightarrow \texttt{t}_k \rightarrow \texttt{t}$ der erweiterte Kontext $\texttt{c}_e$ des Typs $\texttt{(t} \rightarrow \texttt{t)} \rightarrow \texttt{t}_1 \rightarrow \ldots \rightarrow \texttt{t}_k \rightarrow \texttt{t}$ zur Verfügung gestellt. Dieser erweiterte Kontext nimmt als zusätzliches Argument eine Funktion, die auf die verbliebenen, also nicht besuchten, Werte hinter dem aktuellen Match angewendet werden. Damit lässt sich die Funktion wie folgt deklarieren:

```
-- (14)
modSeven :: Int -> [Int] -> [Int]
modSeven o (c 7) = c_e (modSeven (o+1)) (7+o)
modSeven _ l = l
```

Dabei entstehen bei Anwendung der Funktion und erfolgreichem Match folgende Bindungen:

```
modSeven 0 [2, 7, 3]
```

[6] wie in 3.2.3 und Abb. 2 veranschaulicht

`c`/(\h -> (2 : h : 3 : []))
`c_e`/(\f h -> (2 : h : (f (3 : []))))

wobei `f` in diesem Beispiel der rekursive Aufruf der Funktion ist, der als Argument die Restliste des erfolgreichen Matchs übergeben wird (in dem Beispiel also [3]).
Im Gegensatz zu (13) ist diese Implementierung korrekt. Beide Implementierungen erhöhen beim ersten Match auf `7` den Zähler `o` von `0` auf `1`. Da das erste Vorkommen der `7` jedoch mit `0` addiert wurde, wird in (13) die erste `7` ein zweites mal gematched. Somit wird der erweiterte Kontext für die Effizienz von Funktionen benötigt und erweitert zusätzlich den Funktionsumfang der Context Pattern.

Insbesondere gilt:

$$\texttt{c} \equiv \texttt{c\_e (\textbackslash i -> i)}$$

wie sich auch an dem Beispiel zeigen lässt:
`c_e` (\i -> i) = (\f h -> (2 : h : (f (3 : [])))) (\i -> i)
$\rightsquigarrow$ (\h -> (2 : h : ((\i -> i) (3 : []))))
$\rightsquigarrow$ (\h -> (2 : h : 3 : [])) $\equiv$ `c`

3.5 Implementierung der Context Pattern in Haskell

Die Möglichkeit einer Implementierung der Context Pattern in Haskell soll anhand der Semantik erläutert werden. Dabei wird ein Ausdruck mit Context Pattern in einen case-Ausdruck *ohne* Context Pattern transformiert.
Für alle auftretenden Context Pattern wird dabei im folgenden davon ausgegangen, dass sie in einem einfachen `case`-Ausdruck vorliegen[7]:

$$\texttt{case } e_0 \texttt{ of \{c } p_1 \texttt{ if } g_1 \texttt{ ... } p_n \texttt{ if } g_n \texttt{ -> e; \_ -> e'\}}$$

wobei:
$e_0 :: t_0$,
t_1, ..., t_m die Typen aller Unterausdrücke von e_0 und
$p_i :: t_{p_i}$ für alle Pattern p_i, mit $i \in \{1, ..., n\}$ ist.
Um nun e_0 in einer oben $\rightarrow$ unten, links $\rightarrow$ rechts Richtung zu durchlaufen werden ($\mathbf{m + 1}$) Funktionen zur Verfügung gestellt, eine für jeden Unterausdruck von e_0 und eine zusätzliche für e_0. Es wird *nicht* eine Funktion für jedes Pattern bereit gestellt. Daher kann es vorkommen, dass eine Funktion auch mehrere Pattern handhabt. Jede dieser Funktionen durchläuft einen Unterausdruck von e_0 und hat folgenden Typ:

$$\text{chk}_{t_j} :: (\text{Int}, t_{pxs}) \texttt{ -> } t_j \texttt{ -> } (\text{Int}, t_{pxs}, (t_0 \texttt{ -> } t_0) \texttt{ -> } t_1 \texttt{ -> ... -> } t_n \texttt{ -> } t_j)$$

wobei:

[7] Gerechtfertigt wird dies durch die Betrachtungen in 2.1 Pattern-Matching und 2.3 Semantik des case-Ausdrucks.

- t_{pxs} ein Tupel aller Typen ist, die in dem Pattern p_i vorkommen,
- in (Int, t_{pxs}) der Integerwert das Pattern bezeichnet, in dem als nächstes gesucht werden soll,
- t_j der zu durchlaufende Typ ist und
- (Int, t_{pxs}, $(t_0 \to t_0) \to t_1 \to ... \to t_n \to t_j$) der Rückgabetyp der Funktion ist.
  Der Integerwert enthält nun die Information, ob das Matchen erfolgreich war. Bei erfolgreichem Match wurde dieser Wert um eins erhöht, kann also als Eingabeparameter für den nächsten Funktionsaufruf verwendet werden. Insbesondere kann am Ende die letzte Funktion überprüfen, ob alle untergeordneten Pattern p_1 ... p_n erfolgreich gematched wurden (Bedingung: Integerwert = n+1).
  Die Funktion wird als lokale erweiterte Kontextfunktion bezeichnet und nimmt, analog zum Extended Context Pattern `e_c`, eine Funktion des Typs $t_0 \to t_0$, sowie ein Argument für jedes Pattern p_i und liefert ein Ergebnis vom Typ p_j. Somit enthält das Ergebnis der letzten, äußersten Funktion $\texttt{chk}_{t_0}$ eine Bindung für den Extended Context `e_c` (und damit auch für den Kontext `c`[8]).

Bindungen, die noch nicht gefunden wurden, werden mit dem vorläufigen Typ: `undefined = error` initialisiert (abkürzend auch als $\perp$ (Bottom) bezeichnet).
Damit kann die Transformation nun folgendermaßen beschrieben werden:

```
case e0 of {c p1 if g1 ... pn if gn -> e; _ -> e'}
⇝let {<chk_t0-decl>;
      ...;
      <chk_tm-decl>}
  in case (chk_t0 (1, ⊥, ..., ⊥) e0) of {
       (n+1, x_1,1, ..., x_n,kp, c_e) -> let {c = c_e id} in e;
       _                              -> e'}
```

wobei:
$\texttt{x}_{1,1}$, ..., $\texttt{x}_{n,k_p}$ die Variablen im Pattern p_i sind.

Es werden also erst die Funktionen $\texttt{chk}_{t_0}$, ..., $\texttt{chk}_{t_m}$ deklariert und anschließend auf das Ergebnis der Funktion $\texttt{chk}_{t_0}$ mit den Parametern:
("erstes zu Matchendes Pattern (`1`)", "alle Variablen dieses Patterns sind undefiniert ($\perp$)") "zu durchlaufender Wert ($\texttt{e}_0$)"
gematched.
Wurden (`n+1`) Pattern gematched, so enthalten $\texttt{x}_{1,1}$, ..., $\texttt{x}_{n,k_p}$ die Bindungen für die Pattern, die innerhalb des Context Patterns verwendet wurden und `c_e` enthält die Bindung der erweiterten Kontext-Funktion.

[8] vgl. 3.4 Eine Erweiterung der Context Pattern: der Extended Context

An dieser Stelle sollen noch ein paar verwendete Variablen eingeführt und erläutert werden:

- n^i: wobei der Index ^{i} für die Anzahl der *Argumente* des Konstruktors steht.
- $\overline{y}^i := y^i_{1,1}, ..., y^i_{n,k_n}$: Vektoren für ungenutzte Variablen, wobei der obere Index ^{i} für die Anzahl der *Argumente* des Konstruktors,
  der erste untere Index (1, ..., n) für die Anzahl der Argumente der Kontext-Funktion und
  der zweite untere Index (1, ..., k_n) für die Anzahl der Konstruktoren steht.
- $\overline{z} := z_1, ..., z_n$: Vektor für die Argumente der Kontext-Funktion.

Die Funktion chk_{t_j} sieht dabei wie folgt aus:

$$
\langle \mathrm{chk}_{t_j}\text{-decl}\rangle \rightsquigarrow \texttt{chk}_{t_j}\ \texttt{(}n^0\texttt{, } \overline{y}^0\texttt{) x = case } n^0 \texttt{ of}
\\
\texttt{1 -> } \langle \mathrm{chk}_{t_j,1}\rangle
\\
\texttt{...}
\\
\texttt{n -> } \langle \mathrm{chk}_{t_j,n}\rangle
\\
\texttt{n+1 -> } \langle \mathrm{rchk}_{t_j}\rangle
$$

Wurden alle Pattern gefunden, wird die Funktion <rchk$_{t_j}$> aufgerufen. Ansonsten wird überprüft, ob das Pattern p_i im Wert des Typs t_j gefunden werden kann. Dies geschieht mittels Typinferenz[9] und dem Ausdruck $A \vdash t_j \to (..., t_{p_i}, ...)$.

$$
\langle \mathrm{chk}_{t_j,i}\rangle \rightsquigarrow \begin{cases}
\texttt{case } x^0 \texttt{ of \{} \\
\quad \langle p_i\text{-}\mathrm{chk}_{t_j}\rangle \\
\quad \langle r_{t_j}\text{-}\mathrm{chk}\rangle \\
\quad \texttt{\_ -> (i, } \overline{y}^0\texttt{, \textbackslash f -> \textbackslash}\overline{z}\texttt{ -> } x^0\texttt{)} \\
\texttt{\}} & \text{; falls } A \vdash t_j \to (..., t_{p_i}, ...) \\
 & \\
\texttt{(i, } \overline{y}^0\texttt{, \textbackslash f -> \textbackslash}\overline{z}\texttt{ -> } x^0\texttt{)} & \text{; sonst}
\end{cases}
$$

Ist ein Matchen des Patterns nicht möglich, wird der Zähler für das aktuelle Pattern unverändert zurückgegeben, ebenso die Variablen $\overline{y}^0$. Die lokale Kontext-Funktion gibt den unveränderten Wert der Eingabe zurück.
Kann das Pattern jedoch in diesem Typen gefunden werden, wird als erstes von der Funktion <p_i-chk$_{t_j}$> untersucht, ob das Pattern direkt vorkommt.
Ist dies nicht der Fall, sucht die Funktion <r_{t_j}-chk> rekursiv in den Unterausdrücken.

[9] siehe 3.2.4 Typinferenz

$$<\mathrm{p}_i\text{-chk}_{t_j}> \rightsquigarrow \begin{cases} \mathtt{p}_i \ \mathtt{|}\ \mathtt{g}_i \ \mathtt{->}\ \mathtt{(}\ \mathtt{i+1,} \\ \ \mathtt{y}^0_{1,1}\mathtt{,}\ \ldots\mathtt{,}\ \mathtt{y}^0_{i-1,k_{i-1}}\mathtt{,} \\ \ \mathtt{x}^0_{i,1}\mathtt{,}\ \ldots\mathtt{,}\ \mathtt{x}^0_{i,k_i}\mathtt{,} \\ \ \mathtt{y}^0_{i+1,1}\mathtt{,}\ \ldots\mathtt{,}\ \mathtt{y}^0_{n,k_n}\mathtt{,} \\ \ \mathtt{\backslash f\ ->\ \backslash}\overline{z}\ \mathtt{->\ \backslash}\overline{z}_i\mathtt{);} & ;\ \text{falls } \mathrm{t}_j = \mathrm{t}_i \\ \\ \varepsilon & ;\ \text{sonst} \end{cases}$$

Matched das Pattern p_i direkt, der Context Pattern Guard (*if* g_i) wird dabei in den Haskell Standard Guard ($\mathtt{|\ g}_i$) übersetzt, wird der Parameter für das Pattern um eins erhöht. Die unbenutzten Variablen der vorherigen Pattern ($\mathtt{y}^0_{1,1}\mathtt{,}\ \ldots\mathtt{,}\ \mathtt{y}^0_{i-1,k_{i-1}}$) und die noch nicht initialisierten Variablen der nachfolgenden Pattern ($\mathtt{y}^0_{i+1,1}\mathtt{,}\ \ldots\mathtt{,}\ \mathtt{y}^0_{n,k_n}$) werden unverändert übergeben. Die Bindungen ($\mathtt{x}^0_{i,1}\mathtt{,}\ \ldots\mathtt{,}\ \mathtt{x}^0_{i,k_i}\mathtt{,}$) enthalten die Werte der Variablen des aktuellen Patterns.
Die lokale Kontext-Funktion gibt den aktuellen Parameter des übergeordneten Konstruktors zurück.

Falls das Pattern nicht direkt gematched werden kann oder nicht matched, wird rekursiv gesucht. Dazu müssen alle Konstruktoren des Typs t_j bekannt sein. Im Folgenden wird von n Konstruktoren für den Typ t_j ausgegangen, die jeweils a_{j,n_j} Argumente benötigen.

$$\begin{aligned} <\mathrm{r}_{t_j}\text{-chk}> \rightsquigarrow\ & \mathrm{K}_{t_j,1}\ \ \mathrm{w}_1\ \ldots\ \mathrm{w}_{a_{j,1}}\ \ \mathtt{->}\ <\mathrm{chkrck}_{t_j,1}>\mathtt{;} \\ & \ldots \\ & \mathrm{K}_{t_j,n_j}\ \mathrm{w}_1\ \ldots\ \mathrm{w}_{a_{j,n_j}}\ \mathtt{->}\ <\mathrm{chkrck}_{t_j,n_j}>\mathtt{;} \end{aligned}$$

In der Funktionsdefinition wird K_{t_j,n_j} durch K und $\mathrm{a}_{j,k}$ durch a abgekürzt. Im Fall (a = 0) ist nichts zu tun, andernfalls werden alle Argumente des Konstruktors K von links nach rechts durchlaufen.

$$<\mathrm{chkrek}_{t_j,k}> \rightsquigarrow \begin{cases} \mathtt{(n}^0\mathtt{,}\ \overline{y}^0\mathtt{,}\ \mathtt{\backslash f\ ->\ \backslash}\overline{z}\ \mathtt{->\ x}^0\mathtt{)} & ;\ \text{falls a} = 0 \\ \\ \mathtt{let\ (n}^1\mathtt{,}\ \overline{y}^1\mathtt{,}\ \mathtt{c\_e}^1\mathtt{)\ =\ chk}_{t_{l_1}}\ \mathtt{(n}^0\mathtt{,}\ \overline{y}^0\mathtt{)\ w}_1 \\ \ \ldots \\ \mathtt{(n}^a\mathtt{,}\ \overline{y}^a\mathtt{,}\ \mathtt{c\_e}^a\mathtt{)\ =\ chk}_{t_{l_a}}\ \mathtt{(n}^{a-1}\mathtt{,}\ \overline{y}^{a-1}\mathtt{)\ w}_a \\ \mathtt{in\ (n}^a\mathtt{,}\ \overline{y}^a\mathtt{,}\ \mathtt{\backslash f->\backslash}\overline{z}\mathtt{->\ K\ (c\_e}^1\ \mathtt{f}\ \overline{z}\mathtt{)\ldots(c\_e}^a\ \mathtt{f}\ \overline{z}\mathtt{)} & ;\ \text{sonst} \end{cases}$$

wobei:

- t_{l_1}, ..., t_{l_a}: die Argumenttypen des Konstruktors K sind und

- (c_e^1 f $\overline{z}$) ... (c_e^a f $\overline{z}$): die benötigten Argumente des Konstruktors liefern (vgl. <p$_i$-chk$_{t_j}$>)

Nachdem alle Pattern der Kontext-Funktion gematched haben, sucht die Funktion <rchk$_{t_j}$> als letztes nach den verbliebenen Teilen von t_0.

$$<\text{rchk}_{t_j}> \rightsquigarrow \begin{cases} \texttt{(n+1, } \overline{y}^0\texttt{, \textbackslash f->\textbackslash}\overline{z}\texttt{-> (f x}^0\texttt{))} & ; \text{falls } t_j = t_0 \\ <\text{r}_{t_j}\text{-chk}> & ; \text{falls } t_i \neq t_0,\ A \vdash t_j \rightarrow (..., t_0, ...) \\ \texttt{(n+1, } \overline{y}^0\texttt{, \textbackslash f -> \textbackslash}\overline{z}\texttt{ -> x}^0\texttt{)} & ; \text{sonst} \end{cases}$$

Weil bei erfolgreichem Matchen dieser Teil jeweils entfernt und bei erfolglosem Matchen der Wert zurück gegeben wurde, ist:

- im Fall $t_j = t_0$

  x_0 der unbesuchte Teil vom Typ t_0

  \f->$\overline{z}$-> (f x^0) die erweiterte Kontext-Funktion `e_c`

  $\overline{y}^0$ die Bindungen für die Parameter der Kontext-Funktion

- im Fall $t_i \neq t_0$, $A \vdash t_j \rightarrow (..., t_0, ...)$

  dies der Einstiegspunkt der Funktion, vor dem Matchen des Patterns

- im letzten Fall

  das Pattern nicht zu matchen.

  Insbesondere werden die Bindungen noch den Wert $\perp$ enthalten.

3.6 Weitere Beispiele für Context Pattern

3.6.1 Sortieren

In diesem Beispiel soll der Sortieralgorithmus "Bubble Sort" mit Hilfe von Context Pattern implementiert werden. Die Elemente einer Liste sollen nach dem Terminieren des Algorithmus aufsteigend sortiert und mit dem kleinsten Element beginnend vorliegen[10]:

```
sort :: Ord a => [a] -> [a]
sort (outer (x:inner (y:zs) if y<x)) =
    sort (outer (y:inner (x:zs)))
sort zs = zs
```

[10] verschachtelte Context Pattern sind erlaubt

Als erstes wird `x` auf das erste Element der Liste gematched, `y` auf das zweite Element der Liste und `zs` auf die Restliste. Die beiden Kontext-Funktionen `outer` und `inner` sind die Identität, sie enthalten also keinen Kontext.
Nun wird der Guard geprüft. Liefert er *True* werden die Variablen gebunden, sowie `x` und `y` vertauscht und der Algorithmus beginnt erneut am Anfang. Ansonsten wird `y` um ein Element weiter verschoben und das zweite Element der Liste liegt nun in der Kontext-Funktion `inner`. Dies wird so fortgesetzt, bis `y` das Ende der Liste erreicht hat. Nun wandert das erste Element in den Kontext `outer` und `x` wird auf das zweite Element gematched. `y` testet wieder alle Elemente der Liste zwischen `x` und dem Ende der Liste. Wird kein erfolgreiches Match mit erfülltem Guard gefunden, ist die Liste sortiert und die nächste Regel der Funktionsdefinition gibt das Ergebnis zurück.
Die Kontext-Funktionen `inner` und `outer` haben beide den Typ
`Ord a => [a] → [a]`

Da der Konstruktor ":" nur dazu dient sicherzustellen, dass `x` und `y` Elemente sind, kann er auch ausgelassen werden:

```
sort :: Ord a => [a] -> [a]
sort (outer x (inner y if y<x)) =
    sort (outer y (inner x))
sort zs = zs
```

Die Kontext-Funktion `outer` ist nun vom Typ `Ord a => a → [a] → [a]` und enthält unverändert alle Elemente vor `x`. Dagegen hat die Kontext-Funktion `inner` wie zuvor den Typ `Ord a => a → [a]`, enthält jetzt jedoch alle Elemente zwischen `x` und `y`, sowie die Restliste.
Nun treffen jedoch zwei Kontexte direkt aufeinander, also ohne trennendes Standard-Pattern. Daher lassen sich die Kontext-Funktionen zu einer Funktion verschmelzen:

```
sort :: Ord a => [a] -> [a]
sort (c x y if y<x)) = sort (c y x)
sort zs = zs
```

Die Kontext-Funktion `c` ist nun vom Typ `Ord a => a → a → [a]` und enthält alle Elemente außer `x` und `y`. Wenn zwei Kontexte direkt aufeinander treffen, ist ein solches Verschmelzen der Kontexte immer möglich.

3.6.2 Desugarer

Der Desugarer (oder Entsüßer) erhielt diesen Namen, weil er unter anderem den syntaktischen Zucker (wie Pattern Matching und list comprehensions) aus dem Code entfernt. Er übersetzt das Haskell Programm in die Zwischensprache *Core*, in der alle weiteren Compiler-Phasen durchgeführt werden, mit Ausnahme der Code-Generierung. Der Desugarer lebt in einer Monade, welche die benötigte Umgebung liefert. Dort

werden komplexe Konstrukte, wie verschachteltes Pattern-Matching mit Guards oder einfache Bedingungen durch flache case-Ausdrücke ersetzt[11].
Um nun alle "`if`" Bedingungen zu entfernen und durch "`case`" Bedingungen zu ersetzen, können Context Pattern verwendet werden:

```
uncond :: Expr -> Expr
uncond (c ( E_if con e_true e_false)) =
    uncond (c (E_case con [pt, pf]))
    where pt = (PCon "True"[], e_true)
          pf = (PCon "False"[], e_false)
uncond e = e
```

4 Zusammenfassung

In dieser Ausarbeitung wurde über Pattern Matching und deren Erweiterung durch Context Pattern berichtet. Es wurde dargelegt, wie diese in Haskell implementiert werden können und sich in die gegebenen Strukturen einbetten lassen. Dabei konnte beobachtet werden, dass besonders Funktionen die rekursiv Datenstrukturen durchsuchen oder verändern von Context Pattern profitieren, da sich deren Implementierung teilweise von der Datenstruktur löst und Pattern Matching sehr viel intuitiver wird. Dies führt zu weniger Veränderungen an den Funktionen, wenn die Datenstruktur modifiziert wird und zu kompaktem Code, da innerhalb der Funktionen weniger Rekursionen auftreten.
Eine erste Implementierung der Context Pattern auf Basis des Glasgow Haskell Compilers (ghc 2.01 für sparc-sun-solaris2) ist unter:

`http://www-i2.informatik.rwth-aachen.de/Staff/Current/mohnen/CP/ghc.html`

zu finden.

[11] vgl. 2.1 Pattern-Matching und 2.3 Semantik von case-Ausdrücken