Arbeitsbereich DBIS
Fachbereich Informatik
Universität Hamburg
D-2000 Hamburg 13
Germany

Tycoon

Title: # P-Quest Benutzerhandbuch

Author: Claudia Niederée
Sven Müßig
Florian Matthes

Identification: DBIS Tycoon Report 102-92

Status: Version 1.0

Date: February 27, 1992

Description: This document provides a smooth introduction into the language concepts of P-Quest an the underlying persistence model (in German).

Related Documents: The Quest Language and System (Tracking Draft) [Car90]
Tycoon Library Manual (*P-Quest* Version) [Mat91]

Inhaltsverzeichnis

1 Einführung **2**

2 *P-Quest*-Typkonzepte **2**

2.1 Die Ebene der Werte 3

2.2 Die Ebene der Typen 4

2.3 Die Ebene der Kinds 4

3 Funktionen in *P-Quest* **4**

3.1 Einfache Funktionen 5

3.2 Rekursive Funktionen 5

3.3 Funktionen höherer Ordnung 6

3.4 Polymorphe Funktionen 6

4 Datentypen und Typoperatoren **7**

4.1 Funktionstypen 8

4.2 Rekursive Datentypen 8

4.3 Typoperatoren 8

4.4 Abstrakte Datentypen 9

5 Subtypisierung und Subtyppolymorphismus **10**

6 Imperative Programmierung **12**

6.1 Veränderbare Variablen 12

6.2 Sequenzen und Schleifen 13

6.3 Der Datentyp *Array* 13

7 Module und Schnittstellen **14**

8 Ausnahmebehandlung **15**

9 Persistenz **16**

9.1 Dynamische Datentypen 16

9.2 Der Objektspeicher von *P-Quest* 17

1 Einführung

Quest[1] ist eine strikt typisierte, polymorphe, funktionale Programmiersprache, in der jedoch auch imperative Eigenschaften zu finden sind. Die Typüberprüfung findet statisch zur Übersetzungszeit statt.

Die Sprache *Quest* wurde von Luca Cardelli bei DEC SRC in Palo Alto, USA [Car89] entwickelt. Im Rahmen einer Diplomarbeit an der Universität Hamburg [Mü91] wurde die Sprache durch Hinzufügen eines Persistenzkonzeptes zu *P-Quest* erweitert. In diesem Report wird in die relevanten Konzepte der Sprache *P-Quest* eingeführt und danach das Persistenzkonzept des *P-Quest*-Systems erläutert. Eine vollständige Einführung in die Sprache *Quest* findet sich in [Car90].

Das *P-Quest*-System besteht aus mehreren Komponenten. Die Komponenten sind eine interaktive Programmierumgebung, eine Reihe von Bibliotheken (z.Zt. Standard-, Bulk Data Type- und Grafik-Bibliothek), sowie eine in Modula-3 [CDG+88] realisierte Abstrakte Maschine. Die Abstrakte Maschine interpretiert den vom *P-Quest*-Compiler erzeugten Bytecode.

Mit *P-Quest* werden viele neuere Programmierkonzepte realisiert, die in den nachfolgenden Abschnitten anhand von Beispielen vorgestellt werden sollen.

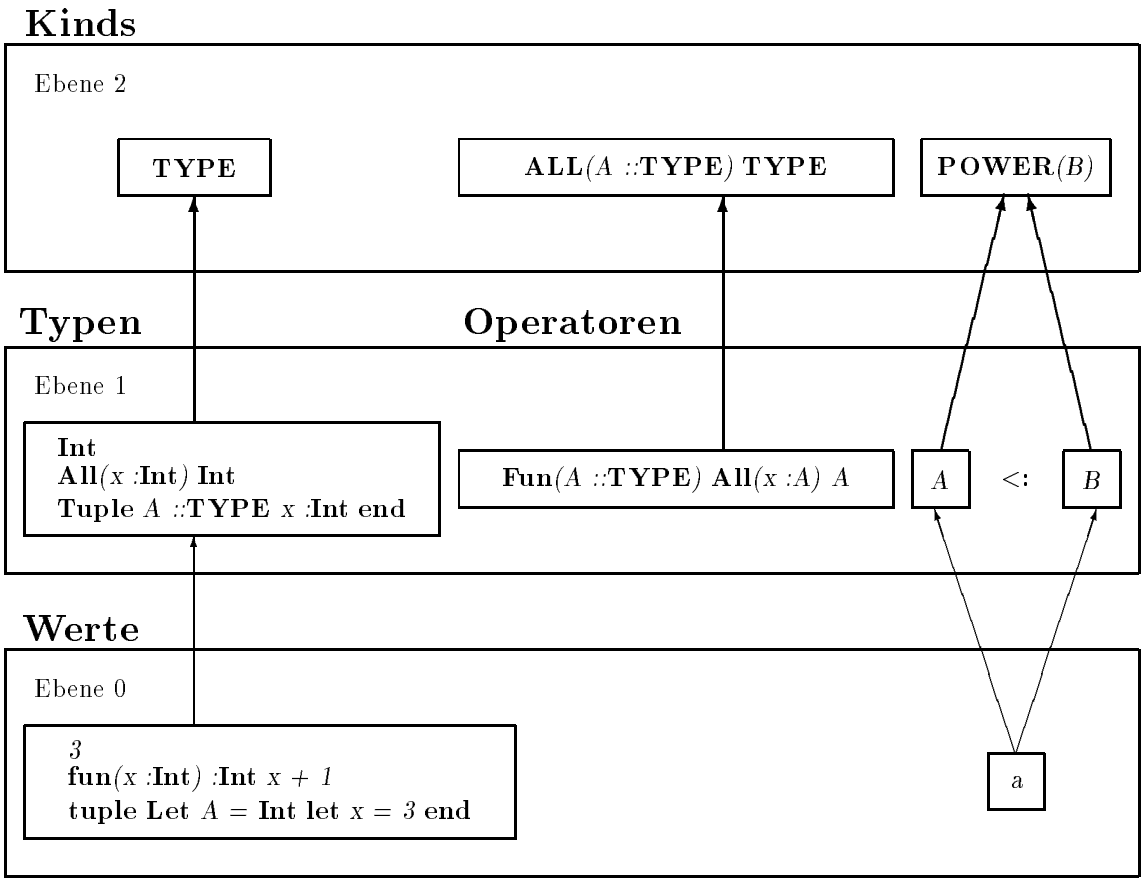
Die Konzepte sind im einzelnen:

- Erweitertes Typkonzept (Werte, Typen, Kinds)
- Funktionen höherer Ordnung
- Subtypisierung
- Parametrischer und Subtyppolymorphismus
- Abstrakte Datentypen
- Module und Schnittstellen
- Funktionen und Module als Objekte *erster Klasse*
- Ausnahmebehandlung

2 *P-Quest*-Typkonzepte

Traditionelle Progammiersprachen unterscheiden nur zwischen *Werten* und *Typen*. Werte stellen die Ebene 0 und Typen die Ebene 1 dar. Wie in der nachfolgenden Abbildung zu sehen ist, gibt es in *P-Quest* noch eine weitere Ebene (Ebene 2), die Ebene der *Kinds*. Kinds klassifizieren Typen in der gleichen Weise, wie Typen Werte. Die Ebene der Kinds wird benötigt, um die in *P-Quest* ungewöhnlich reiche Typebene zu strukturieren. Die drei Ebenen werden in den nachfolgenden Abschnitten anhand von Beispielen beschrieben.

[1] *Quest* steht für *Quantifiers and Subtypes*.



Die obige Abbildung zeigt nicht nur die im Typkonzept von *P-Quest* vorhandenen drei Ebenen, sondern sie verdeutlicht auch die in Abschnitt 2.3 aufgeführte Typisierung von Operatoren in der Kindebene und die in Abschnitt 5 beschriebene Subtypisierung.

Bevor auf die drei Ebenen eingegangen wird, sei hier die verwendete Konvention für die Schreibweise von Schlüsselworten und die Schreibweise von Bezeichnern in den Beispielen erläutert. Schlüssselworte im Zusammenhang mit Werten werden klein geschrieben, Schlüsselworte im Zusammenhang mit Typen beginnen mit einem Großbuchstaben und Schlüsselworte im Zusammenhang mit Kinds werden ganz in Versalien geschrieben. In dieser Arbeit wird zur besseren Unterscheidung zwischen den Ebenen die Konvention von Cardelli übernommen, diese Regel auch auf die Schreibweise von Bezeichnern für Objekte in der jeweiligen Ebene anzuwenden.

2.1 Die Ebene der Werte

Zur Ebene der Werte gehören Ausprägungen der in *P-Quest* angebotenen Basistypen, strukturierte Werte, wie Tupel und Optionen, aber auch Funktionen und polymorphe Funktionen.

Ein Wert wird mit dem Schlüsselwort *let* an eine Variable gebunden. Im ersten Beispiel wird eine Ausprägung des Basistyps Integer an die Variable *x* gebunden.

let *x = 3*
let *sven* = **tuple let** *name = "sven"* **let** *age = 25* **end**
let *name* = **fun**(*p :Person*) :**String** *p.name*
let *identity* = **fun**(*A ::***TYPE** *a :A*) *:A a*

Im zweiten Beispiel wird ein strukturierter Wert, und zwar ein *tuple*, an die Variable *sven* gebunden. In den folgenden Beispielen werden eine Funktion und eine polymorphe Funktion definiert. Wie am vierten Beispiel zu sehen ist, können bereits auf dieser Ebene Typen auftreten, und zwar z.B. als Komponenten von Tupeln und als Parameter von polymorphen Funktionen.

2.2 Die Ebene der Typen

Die Typebene enthält Typen und Typoperatoren. Zu den Typen gehören neben den Basistypen wie z.B. *Int*, *Real* und *Ok*[2], strukturierte Typen, wie z.B. Tupeltypen und Funktionstypen. Typoperatoren (Typfunktionen) sind Funktionen, die es ermöglichen, Typen auf Typen abzubilden. Objekte der Typebene können durch das Vorstellen des Schlüsselworts *Let* an Namen gebunden werden. Es folgen einige Beispiele für die Definition und Benennung von Typen.

Let *Number* = **Int**
Let *Person* = **Tuple** *name* :**String** *age* :**Int end**
Let *Name* = **All**(*:Person*) **String**
Let *Identity* = **Fun**(*A ::***TYPE**) *::***TYPE** *A*

Die Beispiele korrespondieren mit den Beispielen auf der Ebene der Werte. Das erste Beispiel führt einen neuen Namen für den Typ der ganzen Zahlen ein. *Person* ist der Typ der Variablen *sven* und *Name* der Typ der Funktion *name*. Das vierte Beispiel definiert einen Typoperator. Es handelt sich dabei um eine Identitätsfunktion auf der Typebene.

2.3 Die Ebene der Kinds

Kinds sind die Typen von Typen. Sie dienen zur Strukturierung der Menge der Typen. Der allgemeinste Kind ist *TYPE*; das ist der Kind, der alle Typen enthält. Weitere Kinds können vom Benutzer definiert werden. Die Definition von Kinds geschieht durch Voranstellen des Schlüsselwortes *DEF*.

DEF *TYPOP* = **ALL**(*A ::***TYPE**) **TYPE**
DEF *PERSON* = **POWER**(**Tuple** *name* :**String end**)

Im ersten Beispiel wird der Kind der einstelligen Typoperatoren definiert und an den Namen *TYPOP* gebunden. Das Schlüsselwort *ALL* dient zur Definition von Typen von Typoperatoren. Der im zweiten Beispiel definierte Kind *PERSON* repräsentiert alle Tupeltypen, deren erste Komponente den Attributnamen *name* hat und vom Typ *String* ist, die aber eventuell weitere Komponenten besitzen. Wenn *T* ein Typ ist, dann ist *POWER(T)* ein Kind, und zwar die Menge aller Subtypen von *T*.

3 Funktionen in *P-Quest*

Neben den aus anderen Programmiersprachen bekannten *einfachen* und *rekursiven* Funktionen gibt es in *P-Quest* zusätzlich Funktionen *höherer Ordnung* und

[2] *Ok* ist ein Typ, der nur die Konstante *ok* beinhaltet. Entspricht *void* in C.

polymorphe Funktionen. Auf diese Arten von Funktionen und ihre Definition in *P-Quest* wird im folgenden geanauer eingegangen.

3.1 Einfache Funktionen

Da das Konzept der einfachen Funktionen aus anderen Programmiersprachen hinreichend bekannt ist, soll es hier genügen, ihre Syntax in *P-Quest* einzuführen. Diese unterscheidet sich allerdings ein wenig von der in traditionellen Programmiersprachen üblichen. Funktionen werden durch das Schlüsselwort *fun* eingeleitet, und die einzelnen Parameter werden nicht durch Kommata, sondern durch Leerzeichen voneinander getrennt, wie an den beiden folgenden Beispielen zu sehen ist:

```
let succ = fun(a :Int) :Int a + 1
let add = fun (a :Real b :Real) :Real a ++ b
```

Die erste Funktion *succ* erwartet einen Parameter vom Typ *Int* und liefert als Ergebnis auch einen Wert vom Typ *Int* zurück. Sie berechnet für einen ganzzahligen Eingabewert den direkten Nachfolger. Die zweite Funktion *add* addiert zwei reelle Zahlen. Sie erwartet zwei Parameter vom Typ *Real* und liefert einen Wert vom Typ *Real* zurück. Die Syntax von *P-Quest* läßt auch folgende verkürzte Schreibweisen zu:

```
let succ(a :Int) :Int = a + 1
let add(a, b :Real) :Real = a ++ b
```

In *P-Quest* gibt es keine Operatorüberladungen. Deshalb gibt es je nach Parametertyp unterschiedliche Additionsoperatoren. Während zwei ganze Zahlen mit einem Pluszeichen (+) addiert werden, benutzt man für die Addition zweier reeller Zahlen ein doppeltes Pluszeichen (++).

3.2 Rekursive Funktionen

Rekursive Funktionen werden durch das Schlüsselwort *rec* gekennzeichnet. Ein Beispiel für rekursive Funktionen ist die bekannte Fakultätsberechnung:

```
let rec fac(n :Int) :Int =
  if n is 0
    then 1
    else n * fac(n - 1)
  end
```

Bei diesem Beispiel ist außerdem zu beachten, daß der Test auf Gleichheit zweier Werte nicht durch das Gleichheitszeichen, sondern durch den Operator *is* geschieht. Das dient der Vermeidung von Operatorüberladungen[3]. Der Operator *is* testet einfache Werte, wie z.B. Zahlen auf Gleichheit, strukturierte Werte hingegen, wie z.B. Tupel, auf Identität.

Um wechselseitig rekursive Funktionen zu schreiben, müssen Funktionen *parallel* definiert werden. In *P-Quest* verbindet man hierbei die Definitionen mit dem Schlüsselwort *and*. Als Beispiel ist hier ein Paritätstest aufgeführt.

```
let rec even(n :Int) :Bool =
  if n is 0 then
    true
  elsif n < 0 then
    even(int.abs(n))
```

[3] Das Gleichheitszeichen dient der Bindung an eine Variable.

```
  else
    odd(n - 1)
  end
and odd(n :Int) :Bool =
  not(even(n))
```

3.3 Funktionen höherer Ordnung

Funktionen höherer Ordnung sind Funktionen, denen Funktionen als Parameter übergeben werden bzw. die Funktionen als Ergebnis zurückliefern können.

Ein Beispiel für Funktionen höherer Ordnung ist die Funktion *double*.

```
let double(f(a :Int) :Int a :Int) :Int = f(f(a))
```

Sie erhält als ersten Parameter eine Funktion, die ganzzahlige Werte auf ganzahlige Werte abbildet, und als zweiten Parameter einen ganzzahligen Wert übergeben. *double* wendet die übergebene Funktion zweimal hintereinander auf den übergebenen Wert an.

Ein Aufruf der Funktion *double* mit der Funktion *succ* und der Zahl 4 als Parameter (*double(succ 4)*) zum Beispiel, berechnet den Nachfolger vom Nachfolger von vier, also die Zahl Sechs.

3.4 Polymorphe Funktionen

Eine Funktion wird zu einer polymorphen oder generischen Funktion, indem man ihre Signatur durch einen oder mehrere Typparameter erweitert (*parametrischer Polymorphismus*). Die Typparameter werden beim Aufruf durch Typen instanziiert.

Ein einfaches Beispiel für ein polymorphe Funktion ist eine polymorphe Identitätsfunktion. Sie soll hier in zwei Versionen betrachtet werden:

```
let identity1(A ::TYPE)(a :A) :A = a
let identity2(A ::TYPE a :A) :A = a
```

Die Versionen unterscheiden sich dadurch, daß die erste in zwei Stufen parametrisiert werden kann, während die zweite in einer Stufe parametrisiert werden muß.

Die erste Version kann in der ersten Stufe nur mit einem Typ T parametrisiert werden. Man erhält dann eine Identitätsfunktion auf dem Typ T, was hier am Beispiel des Typs *Integer* gezeigt wird:

```
let intId = identity1(:Int)
```

Die zweite Version muß sofort mit einem Typ T und mit einem Wert t dieses Typs parametrisiert werden und liefert als Ergebnis t zurück. In der Regel kann dabei jedoch die Angabe des Typs weggelassen werden, da dieser vom *P-Quest*-System durch Typinferenz aus dem übergebenen Wert hergeleitet werden kann.

Folgende Aufrufe der definierten Funktionen liefern alle das gleiche Ergebnis und zeigen noch einmal die verschiedenen Parametrisierungen der beiden Versionen der polymorphen Identitätsfunktion:

```
intId(4)
identity1(:Int)(4)
identity2(:Int 4)
identity2(4)
```

Für die Instanziierung von Typparametern sind nicht nur die Basistypen, sondern auch beliebige benutzerdefinierte Typen zulässig.

Let *Person* = **Tuple** *name* :**String** *age* :**Int end**
let *selfId* = *identity1(:Person)*

Der große Vorteil von polymorphen Funktionen ist, daß man eine Funktion schreiben kann, die auf allen Typen arbeitet, anstatt für jeden Typ eine eigene Version dieser Funktion zu schreiben. Sie eignen sich also besonders zur Beschreibung von Verhalten, das typunabhängig ist.

Eine besondere Stärke von *P-Quest* liegt in der Möglichkeit, das Konzept der polymorphen und der höheren Funktionen miteinander zu kombinieren. Dies soll am Beispiel einer polymorphen Sortierfunktion veranschaulicht werden. Das eigentliche Sortieren, d.h. das Umordnen der Elemente ist typunabhängig, kann also im Rahmen einer polymorphen Funktion beschrieben werden. Das Vergleichen der Elemente jedoch, das beim Sortieren durchgeführt werden muß, ist typabhängig. Diese Aufgabe kann gelöst werden, indem man der polymorphen Sortierfunktion eine Funktion als Parameter übergibt, deren Aufruf den Vergleich von zwei Elementen durchführt.

Die Signatur einer polymorphen Sortierfunktion, die Felder von Elementen sortiert, könnte dann wie folgt aussehen:

let *sort(A ::***TYPE** *order(a,b :A)* :**Bool** *a* :**Array**(*A*)) :**Array**(*A*) = ...

Um ein konkretes Feld zu sortieren, muß man noch eine Vergleichsfunktion auf dem Elementtyp schreiben. Mit dieser Funktion legt man das Sortierkriterium und die Sortierreihenfolge fest. Als Beispiel sei hier die Sortierung von Personen nach ihrem Alter gewählt:

let *older(a, b :Person)* :**Bool** = *a.age* > *b.age*

Der folgende Aufruf der Funktion *sort* sortiert dann ein Feld von Personen, und zwar nach ihrem Alter.

sort(:Person older personArray)

In den Abschnitten 4.3 und 4.4 sind Beispiele für die Implementation polymorpher Funktionen zu finden.

4 Datentypen und Typoperatoren

Die Sprache *P-Quest* enthält neben den Basistypen *Ok*, *Bool*, *Char String*, *Int* und *Real* die Typkonstruktoren *Tuple*, *Option*, *Array* und *Exception*[4]. Dabei entspricht der Typ *Tuple* dem Record traditioneller Programmiersprachen und der Typ *Option* dem varianten Record. Definitionen von Tupel- bzw. Optionstypen und zugehöriger Wert sehen wie folgt aus:

Let *Point* = **Tuple** *x* :**Int** *y* :**Int end**
let *point* = **tuple let** *x* = *5* **let** *y* = *7* **end**

Let *Figure* =
  Option
    circle **with** *center :Point radius* :**Int end**
    triangle **with** *point1, point2, point3 :Point* **end**
    rectangle **with** *point1, point2, point3 :Point* **end**

[4] Auf die Typkonstruktoren *Array* und *Exception* wird in späteren Abschnitten genauer eingegangen.

```
    end
let circle =
    option circle of Figure with let center = point let radius = 1 end
```

Der Zugriff auf Komponenten eines Tupels geschieht mit Hilfe der Punktnotation. Optionen werden im allgemeinen mit einer *case*-Anweisung inspiziert.

Soviel zu den grundlegenden Typen in *P-Quest*. Im folgenden wird auf die spezielleren Typen und auf die Typoperatoren eingegangen.

4.1 Funktionstypen

In *P-Quest* ist jeder Funktion (auch den polymorphen und denen höherer Ordnung) ein Typ zugeordnet. Funktionstypen werden mit dem Schlüsselwort *All* eingeleitet. Zur Veranschaulichung sind anschließend die Typen einiger der im Abschnitt 3 definierten Funktionen aufgeführt:

```
succ, fac :All(:Int) Int
add :All(:Real :Real) Real
double :All(:All(:Int) Int :Int) Int
identity1 :All(A ::TYPE) All(:A) A
identity2 :All(A ::TYPE :A) A
sort :All(A ::TYPE :All(:A :A) Bool :Array(A)) Array(A)
```

4.2 Rekursive Datentypen

Rekursive Datenstrukturen, wie Listen, Mengen und Bäume spielen eine wichtige Rolle in der Informatik. *P-Quest* bietet deshalb auch eine Möglichkeit zur Definition von rekursiven Datentypen, mit denen rekursive Datenstrukturen auf einfache Weise realisiert werden können. Als Beispiel sei hier die Definition einer Liste für ganze Zahlen gezeigt:

```
Let Rec IntegerList ::TYPE =
  Option
    nil
    cons with head :Int tail :IntegerList end
  end
```

Die Definition von rekursiven Typen wird durch *Rec* eingeleitet.

An diesem Beispiel läßt sich die Einführung von Typoperatoren motivieren, die im nächsten Abschnitt beschrieben werden. Ähnlich wie beim Übergang zu polymorphen Funktionen ist es auch hier wünschenswert, gemeinsame, typunabhängige Muster herauszukristallisieren und für diese Muster generischen Code zu schreiben, der für die einzelnen Typen instanziiert werden kann. Im konkreten Fall wäre das so etwas wie eine polymorphe Liste, die mit einem Typ E instanziert eine Liste über Elementen vom Typ E ergibt.

4.3 Typoperatoren

Typoperatoren sind Funktionen, die Typen auf Typen abbilden. Sie ermöglichen es, Typdeklarationen zu parametrisieren.

Ein einfaches Beispiel für einen Typoperator ist eine der weiter oben eingeführten Identitätsfunktion entsprechende Funktion auf der Typebene:

```
Let Identity = Fun(E ::TYPE) ::TYPE E
```

oder kürzer (ähnlich wie bei den Funktionen):

```
Let Identity(E ::TYPE) ::TYPE = E
```

Der Operator *Identity* bildet den als Parameter übergebenen Typ auf sich selbst ab.

Wie im vorigen Abschnitt bereits erwähnt, soll jetzt ein Typoperator *List* betrachtet werden, der einen Typ E auf den Typ einer Liste mit Elementen des Typs E abbildet. Dazu wird in die Listendefinition des vorigen Abschnittes ein Typparameter eingeführt.

```
Let List(E ::TYPE) ::TYPE =
  Rec(L)
    Option
      nil
      cons with head :E tail :L end
    end
```

Der Typoperator *List* bildet den Typ E auf einen rekursiven Optionstyp L ab, wobei die erste Komponente vom Typ E und die zweite vom Typ L ist[5]. Die notwendigen Listenoperationen können dann durch polymorphe Funktionen implementiert werden.

```
let new(E ::TYPE) :List(E) =
  option nil of List(E) end

let cons(E ::TYPE head :E tail :List(E)) :List(E) =
  option cons of List(E) with head tail end
```

Mit der polymorphen Funktion *new* können leere Listen beliebigen Typs erzeugt werden, und über die Funktion *cons* können Elemente an den Listenanfang angefügt werden.

Der daraus resultierende Vorteil ist, daß im Gegensatz zu traditionellen Programmiersprachen generische Listen-, Mengen- oder Baumtypen sowie polymorphe Operationen auf diesen definiert werden können, was die Anzahl der zu implementierenden Funktionen stark reduziert. Desweiteren können die Implementationen dieser Strukturen in einfacher Weise modifiziert und ausgetauscht werden, ohne Programme dadurch zu invalidieren.

Die Konzepte der polymorphen Funktionen und der Typoperatoren ergänzen sich also ideal und ermöglichen die Verwendung von generischem Code sowohl bei der Beschreibung von Strukturen als auch bei der Beschreibung von Verhalten.

4.4 Abstrakte Datentypen

Ein Abstrakter Datentyp (ADT) besteht aus einem Datentyp und einem Satz von Operationen auf diesem Datentyp. Dabei wird dem Benutzer nur der Name des Typs und die Signaturen und Namen der Operationen zur Verfügung gestellt. Ein ADT verbirgt die Implementation des Typs und der Operationen nach außen. Die zur Verfügung gestellten Operationen sind die einzig möglichen und zulässigen auf dem abstrakten Datentyp. Dadurch werden die Elemente des ADTs vor unzulässiger Benutzung geschützt.

Da die Implementation verborgen bleibt, kann sie nachträglich geändert werden, ohne dadurch Programme zu invalidieren, die den ADT benutzen. Auch kann,

[5] Hierbei ist zu beachten, daß der Operator *List* einen Typ E auf einen rekursiven Typ abbildet. Der eigentliche Typoperator ist jedoch nicht rekursiv. Rekursive Typoperatoren sind in *P-Quest* nicht zulässig, da sie zu Entscheidbarkeitsproblemen führen.

falls mehrere Implementationen eines ADT zur Verfügung stehen, zwischen diesen dynamisch gewählt werden.

Als Beispiel sei hier ein allgemeiner, funktionaler Kellerspeicher mit Operationen auf demselben vorgestellt. Der Kellerspeicher wird in Form eines polymorphen, abstrakten Datentyps deklariert.

```
Let FunStack =
  Tuple
    T ::ALL(E ::TYPE) TYPE
    new(E ::TYPE) :T(E)
    empty(E ::TYPE stack :T(E)) :Bool
    push(E ::TYPE element :E stack :T(E)) :T(E)
    pop(E ::TYPE stack :T(E)) :T(E)
    top(E ::TYPE stack :T(E)) :E
  end
```

Zu dieser Schnittstelle wird im folgenden eine auf dem Modul *list*[6] basierende Implementation vorgestellt.

```
let listStack :FunStack =
  tuple
    Let T(E ::TYPE) ::TYPE = list.T(E)
    let new(E ::TYPE) :T(E) = list.new(:E)
    let empty(E ::TYPE stack :T(E)) :Bool = list.empty(stack)
    let push(E ::TYPE element :E stack :T(E)) :T(E) =
      list.cons(element stack)
    let pop(E ::TYPE stack :T(E)) :T(E) = list.tail(stack)
    let top(E ::TYPE stack :T(E)) :E = list.head(stack)
  end
```

Man sieht, daß in diesem Beispiel die Operationen *new*, *empty*, *push*, *pop* und *top* als polymorphe Funktionen implementiert wurden, so daß Kellerspeicher für beliebige Datentypen erzeugt werden können.

5 Subtypisierung und Subtyppolymorphismus

In *P-Quest* gibt es eine Subtypbeziehung (<:) zwischen Typen. Sie ist wie folgt definiert: Wenn x vom Typ A ist und es gilt $A <: B$ (A ist Subtyp von B), dann ist x auch vom Typ B. Die Intuition hinter dieser Subtypisierung ist Mengeninklusion.

Die Menge aller Subtypen eines Typs T wird mit *POWER(T)* bezeichnet. A *::POWER(B)* und $A <: B$ haben damit die gleiche Bedeutung und die zweite wird als Abkürzung für die erste Schreibweise betrachtet.

Zwischen den Basistypen gibt es keine Subtypbeziehungen außer den trivialen. Für strukturierte Typen sind die Subtypbeziehungen induktiv definiert. Dazu gibt es für alle Typoperatoren Subtypisierungsregeln. Aus Platzgründen wird an dieser Stelle jedoch nur die Subtypisierung bei Tupel- und Optionstypen betrachtet:

Tupel: Ein Tupeltyp A ist Subtyp eines Tupeltyps B, wenn A mindestens alle Komponenten von B enthält oder genauer, wenn für alle Komponenten von B gilt: Die Namen der i-ten Komponente in A und B stimmen überein und der Typ der i-ten Komponente in A muß ein Subtyp der i-ten Komponente in B

[6] *list* ist ein Modul der Standardbibliothek, das eine polymorphe Liste mit dazugehörigen Operationen anbietet.

sein. Da die Komponenten von Tupeln geordnet sind, kommt es dabei auf die Reihenfolge der Komponenten an.

So ist in folgendem Beispiel *Employee* ein Subtyp von *Person*, nicht aber *Student*:

Let *Person* = **Tuple** *name* :**String** *age* :**Int** **end**
Let *Employee* = **Tuple** *name* :**String** *age* :**Int** *company* :**String** **end**
Let *Student* = **Tuple** *name* :**String** *matrNr* :**Int** *age* :**Int** **end**

Optionen: Ein Optionstyp A ist Subtyp eines Optionstyps B, wenn B mindestens alle Komponenten von A enthält oder genauer, wenn für alle Komponenten von A gilt: Die Namen der i-ten Komponente in A und B stimmen überein und er Typ der i-ten Komponente in A muß ein Subtyp der i-ten Komponente in B sein (im Sinne der Subtypisierung bei Tupeln). Auch die Komponenten von Optionen sind geordnet. Es kommt also auch hier auf die Reihenfolge der Komponenten an.

Der Typ *WeekDay* in folgendem Beispiel ist offensichtlich ein Subtyp vom Typ *Day*. Es gilt die Beziehung *WeekDay* **<:** *Day*.

Let *Day* =
  Option *mon tue wed thu fri sat sun* **end**

Let *WeekDay* =
  Option *mon tue wed thu fri* **end**

Die Subtypisierungsregeln für die anderen Konstruktoren sehen ähnlich, aus werden aber, wie oben bereits erwähnt, hier nicht behandelt.

Ein großer Vorteil der Subtypisierung ist die Tatsache, daß Funktionen, die auf einem Typ arbeiten, auch Werte jedes beliebigen Subtyps dieses Typs akzeptieren. Subtypisierung erleichtert somit die nachträgliche Erweiterung von Programmen, insbesondere die Erweiterung von Datenstrukturen um neue Komponenten. Funktionen, die für den ursprünglichen Typ geschrieben worden sind, arbeiten dann auch auf Werten des neuen Typs, weil sie als Instanzen des alten Typs erkannt werden.

Folgendes Beispiel zeigt die Anwendung, aber auch die Begrenzungen dieses Konzeptes. Seien folgende Tupel definiert:

```
let florian :Person =
  tuple
    let name = "Florian"
    let age = 27
  end
```

```
let gabi :Employee =
  tuple
    let name = "Gabi"
    let age = 29
    let company = "ibm"
  end
```

Es wird jetzt eine Funktion *show* betrachtet, die den Inhalt eines Personentupels anzeigen kann.

```
let show(p :Person) :Person = p
show(florian)
tuple name = "Florian" age = 27 end :Person
```

Da der Typ *Employee* ein Subtyp des Typs *Person* ist, kann man dieser Funktion auch ein Angestelltentupel als Parameter übergeben.

show(gabi)
tuple *name = "Gabi" age = 29* **end** *:Person*

Allerdings ist der Rückgabewert der Funktion vom Typ *Person*. Es geht also Typinformation verloren, was zur Folge hat, daß die Firma von Gabi nicht ausgegeben wird. Damit auch dieses Attribut bei der Ausgabe berücksichtigt wird, ist es notwendig, die Funktion *show* als polymorphe Funktion zu implementieren.

let *show2(A <:Person a :A) :A = a*

show2(gabi)
tuple *name = "Gabi" age = 29 company = "ibm"* **end** *:Employee*

Da genügend Typinformation in die Funktionsdefinition integriert wurde, werden nun alle Attribute ausgegeben. Durch die Angabe von *A <: Person* wird die Funktion *show* zu einer polymorphen Funktion, wobei jedoch der Polymorphismus im Gegensatz zum parametrischen Polymorphismus auf Subtypen des Typs *Person* beschränkt wird. Diesen auf Subtypen eines Typs eingeschränkten Polymorphismus bezeichnet man als *Subtyppolymorphismus*[7].

6 Imperative Programmierung

Die imperative Programmierung basiert auf veränderbaren Werten in einem globalem Speicher. Der Kontrollfluß wird durch Konstrukte für Sequenzen und Schleifen gesteuert.

6.1 Veränderbare Variablen

Die Bindung eines Namens an einen Wert durch das *let*-Konstrukt entspricht der Definition einer Konstanten, da der dem Namen zugewiesene Wert nicht durch eine Zuweisung modifiziert werden kann. Wird an einen bereits existierenden Namen mit *let* ein neuer Wert gebunden, so bewirkt dies die Erzeugung einer neuen Variablen (eigentlich Konstanten).

Soll eine Variable modifizierbar sein, so muß sie bei der Bindung explizit als modifizierbar gekennzeichnet werden. Dies geschieht durch Angabe des Schlüsselworts *var* bei der Definition.

let *a = 7*
let var *b = 9*
b := b + 4

Nur für die Variable *b* ist eine Zuweisung möglich. Eine Zuweisung an die Variable *a* führt bei der Übersetzung zu einem Typfehler, da modifizierbare Werte nicht vom Typ T, sondern von dem speziellen Typ *Var(T)* sind.

Es ist zu beachten, daß in der gegenwärtigen Implementation von *P-Quest* für beliebige Typen *T* weder globale Variablen vom Typ *Var(T)* innerhalb von Funktionen verändert werden können, noch Werte vom Typ *Var(T)* als Parameter von Funktionen zugelassen sind. Diese Einschränkungen können umgangen werden, indem Variablen dieses Typs zuvor in eine Struktur, z.B. in ein Tupel eingebettet werden[8].

[7] Aufgrund eines Fehlers im Compiler funktioniert in der derzeitigen Version des *P-Quest*-Systems die explizite Angabe von Subtypbeziehungen nicht.

[8] Ein Beispiel hierfür findet sich bei der Definition des modifizierbaren Kellerspeichers in Abschnitt 7.

6.2 Sequenzen und Schleifen

Eine Sequenz ist eine Zusammenfassung von Ausdrücken. Der Typ einer Sequenz ist der Typ des letzten Ausdrucks in der Sequenz. Sequenzen werden durch die Schlüsselworte *begin* und *end* zu Blöcken zusammengefaßt.

Schleifen fassen Folgen von Ausdrücken zum Zweck der Iteration zusammen. Der Typ einer solchen Schleife ist *Ok*. Die allgemeinste Form ist die *loop*-Schleife. Zusätzlich gibt es noch zwei speziellere Formen von Schleifen, die *while*-Schleifen und die *for*-Schleifen. Als Beispiel für Sequenzen und Schleifen sei hier eine Funktion angegeben, die den größten gemeinsamen Teiler von zwei ganzen Zahlen bestimmt.

```
let gcd(n,m :Int) :Int =
  begin
    let var vn = n
    and var vm = m
    while vn isnot vm do
      if vn > vm then vn := vn - vm end
      if vn < vm then vm := vm - vn end
    end
    vn
  end
```

Typ und Wert der Sequenz ergeben sich aus ihrem letzen Ausdruck, in diesem Fall *vn*.

6.3 Der Datentyp *Array*

Ein Feld (*Array*) ist eine durch nicht negative ganze Zahlen indizierte Sequenz von Elementen gleichen Typs. Die Größe des Felds wird bei der Erzeugung festgelegt und kann danach nicht mehr geändert werden. Die Elemente des Felds hingegen stellen modifizierbare Werte dar.

```
let a:Array(Int) = array of 0 1 2 3 4 5 end
let b:Array(Bool) = array of (5 false)
b[0] := {a[0] is 0}
```

Es gibt zwei Möglichkeiten, ein Feld zu initialisieren: Beim sechselementigen Feld *a* sind die Elemente einzeln aufgeführt und in *array of* und *end* eingeschlossen. Die zweite Möglichkeit ist die Angabe der Anzahl der Elemente und des Initialisierungswertes, so geschehen beim Feld *b*. Die einzelnen Elemente können, wie aus anderen Programmiersprachen bekannt, durch einem Wert in eckigen Klammern indiziert werden.

Mit Hilfe der *for*-Schleife kann eine Summenfunktion für beliebig große Felder ganzer Zahlen definiert werden.

```
let sum(a :Array(Int)) :Int =
  begin
    let var total = 0
    for i = 0 upto extent(a) - 1 do
      total := total + a[i]
    end
    total
  end
```

Mit dem Operator *extent* kann die Größe von Feldern ermittelt werden. Die Schleifenvariable *i* muß nicht deklariert werden.

Sind Felder Parameter von Funktionen, so kann eine abkürzende Schreibweise benutzt werden.

sum **of** *1 2 3 4* **end**	statt *sum*(**array of** *1 2 3 4* **end**)
sum **of** *(5 2)*	statt *sum*(**array of** *(5 2)*)

7 Module und Schnittstellen

Die Modularisierung ist nach den Funktionen die wichtigste Strukturierungsmöglichkeit moderner Programmiersprachen. *P-Quest* bietet ähnlich wie Modula-2 [Wir85] die Möglichkeit, große Programme in Schnittstellenmodule (*interfaces*) und die eigentlichen Implementationsmodule (*modules*) zu unterteilen. In den Schnittstellenmodulen werden die Bezeichner und Typen der Variablen, die Bezeichner und Kinds der abstrakten Datentypen und die Bezeichner und Signaturen der Funktionen und Typoperatoren, die von den Modulen implementiert und exportiert werden, deklariert.

Zusätzlich kann ein Schnittstellenmodul Typdefinitionen enthalten. Diese werden nicht wie normale Typdefinitionen mit *Let*, sondern mit dem Schlüsselwort *Def* eingeleitet. Die Definition dieser Typen ist, anders als die abstrakten Datentypen, für alle Benutzer der Schnittstelle sichtbar. Werden solche Typen von anderen Modulen benutzt, so ist als Name <Schnittstellenname>_<Typname> anstatt <Modulname>.<Typname> zu verwenden.

In *P-Quest* können zu einem Schnittstellenmodul im Gegensatz zu Modula-2 beliebig viele Implementationsmodule existieren. Module sind, wie Funktionen, Objekte *erster Klasse*, was bedeutet, daß sie an Bezeichner gebunden und als Parameter an Funktionen übergeben werden können.

Schnittstellen- und Implementationsmodule müssen auf dem *Top-Level* der interaktiven Programmierumgebung definiert werden, wobei dadurch implizit externe Dateien, die den Schnittstellen- beziehungsweise Modulcode enthalten, erzeugt werden.

Beide Modularten können ihrerseits von anderen Modulen importieren. Dies geschieht mittels des Schlüsselwortes *import*. Als Beispiel wird noch einmal ein Kellerspeicher betrachtet. Im Gegensatz zu dem in Abschnitt 4.4 vorgestellten wurde hier aber ein modifizierbarer Kellerspeicher gewählt.

```
interface Stack
export
   T(E ::TYPE) ::TYPE
   new(E ::TYPE) :T(E)
   empty(E ::TYPE stack :T(E)) :Bool
   push(E ::TYPE element :E stack :T(E)) :Ok
   pop(E ::TYPE stack :T(E)) :Ok
   top(E ::TYPE stack :T(E)) :E
end
```

Ein zugehöriges Implementationsmodul, das wie die in Abschnitt 4.4 vorgestellte Implementation eine Realisierung des Kellerspeichers basierend auf Listen verwendet, könnte dann wie folgt aussehen:

```
module stack :Stack
import list:List
export
  Let T(E ::TYPE) ::TYPE = Tuple var l :list.T(E) end
  let new(E ::TYPE) :T(E) =
    tuple let var l = list.new(:E) end
  let empty(E ::TYPE stack :T(E)) :Bool = list.empty(stack.l)
  let push(E ::TYPE element :E stack :T(E)) :Ok =
    stack.l := list.cons(element stack.l)
  let pop(E ::TYPE stack :T(E)) :Ok =
    stack.l := list.tail(stack.l)
  let top(E ::TYPE stack :T(E)):E = list.head(stack.l)
end
```

Das Modul kann nun seinerseits von anderen Modulen importiert werden. Nach einem Import in ein Programm oder Modul kann mit der Punktnotation auf die Modulkomponenten zugegriffen werden.

```
module main
import stack :Stack print :Print
  let s = stack.new(:Int)
  stack.push(7 s)
  if not stack.empty(s) then
    print.Int(stack.top(s))
  end
end;
```

8 Ausnahmebehandlung

Wie die Modularisierung, dient auch die Ausnahmebehandlung (*Exception handling*) der Strukturierung großer Programme. Sie ist ein Kontrollflußmechanismus, der sich in *P-Quest* aber problemlos in das Typkonzept einfügt.

Die Ausnahmebehandlung ist ein wichtiges Konzept in einer Programmiersprache, weil bereits einfache Funktionen, wie z.B das Teilen durch Null, Fehlersituationen erzeugen, die abgefangen werden müssen. Neben diesen Standardausnahmen unterstützt *P-Quest* auch benutzerdefinierte Ausnahmen.

Dafür gibt es einen speziellen Typoperator *Exception*, mit dem man verschiedene Typen für Ausnahmen definieren kann. Eine Ausnahme vom Typ *Exception(T)* kann, wenn sie ausgelöst wird, einen Wert vom Typ T als Parameter erhalten.

Ähnlich wie beim Typ *Tuple* werden Werte vom Typ *Exception(T)* zwischen zwei Schlüsselworten eingeschlossen, und zwar *exception* und *end*:

```
let exc1 :Exception(Int) =
  exception integerException :Int end
let exc2 :Exception(String) =
  exception stringException :String end
```

Eine Ausnahme wird mit dem *raise*-Befehl ausgelöst. Dabei kann, wie oben bereits erwähnt, ein Wert vom entsprechenden Typ als Parameter übergeben werden.

```
raise exc1 with 3 end
```

Weiterhin bietet *P-Quest* die Möglichkeit Ausnahmen abzufangen. Dazu wird das *try*-Konstrukt verwendet:

```
try
  ...
  raise exc1 with 55 end
  ...
  when exc1 with x then x + 5
  when exc2 then 1
  else 0
end
```

Die einzelnen Zweige des *try*-Konstrukts dienen dazu, spezielle Ausnahmen, hier *exc1* und *exc2*, zu behandeln. Für die jeweilige Ausnahme wird die genannte Operation ausgeführt. Der mit der Ausnahme erzeugte Wert kann hierbei verwendet werden. Ausnahmen, die in keinem Zweig auftreten, werden vom *else*-Zweig behandelt.

Falls kein solcher existiert, wird die Ausnahme weiter propagiert, bis sie auf ein passendes *try*-Konstrukt trifft oder der Top-Level erreicht ist. In diesem Fall wird die Ausnahme nach außen hin sichtbar.

9 Persistenz

Es gibt in der Sprache *P-Quest* zwei Ansätze, um Daten persistent zu machen.

Dynamische Datentypen — Das Modul *Dynamic* bietet Funktionen an, mit denen Daten samt ihrer Typinformation auf Dateien geschrieben werden können.

Objektspeicher — Beim Übergang von *Quest* zu *P-Quest* wurde das System um das Konzept eines Objektspeicher erweitert, der es ermöglicht, Daten vollständig transparent persistent zu speichern.

9.1 Dynamische Datentypen

Die *P-Quest* Standardbibliothek enthält ein Modul names *Dynamic*, das den abstrakten Datentyp *dynamic.T* sowie eine Menge von Funktionen auf diesem Datentyp zur Verfügung stellt. Der Typ *dynamic.T* und die dazugehörigen Operationen können dazu verwendet werden, Daten ohne Verlust von Typinformation persistent zu speichern.

Der Typ *dynamic.T* kann als ein Tupel interpretiert werden, bei dem in der ersten Komponente der Typ eines Datums und in der zweiten der Wert gespeichert wird. Nachfolgendes Beispiel demonstriert, wie Daten in eine externe Datei exportiert werden können. (Die Funktionen der Module *writer* und *reader* werden dabei zum Erzeugen bzw. Öffnen einer Datei benötigt.)

```
let wr = writer.file("Test.qst")
let dyn = dynamic.new(:Int 3)
dynamic.extern(wr dyn dynamic.portable)
writer.close(wr)
```

Der Parameter *dynamic.portable* legt die Formatierung der exportierten Daten fest.

In einer nachfolgenden Sitzung kann dieses Datum wie folgt in die interaktive Umgebung importiert und dort verwendet werden.

```
let addone(x :Int) = x + 1
let rd = reader.file("Test.qst")
let dyn = dynamic.intern(rd dynamic.portable)
```

```
reader.close(rd)
addone(dynamic.be(:Int dyn))
```

Über die Funktion *dynamic.be* wird eine Typüberprüfung initiiert. Stimmt der Typ, der der Funktion *dynamic.be* als erster Parameter übergeben wird, nicht mit dem in dem Bezeichner *rd* gespeicherten Typ überein, kommt es zu einem Laufzeitfehler. Im anderen Fall wird der Wert des Bezeichners *dyn* an den Aufrufer übergeben[9].

Darüberhinaus können diese Funktionen auch dazu benutzt werden, übersetzte Funktionen oder *P-Quest*-Programme persistent zu speichern, was im nachfolgenden Beispiel demonstriert wird.

```
let wr = writer.file("sqr.qst")
let hello() :Ok = print("Hello world")
dynamic.extern(wr dynamic.new(:All() :Ok hello) dynamic.portable)
writer.close(wr)
```

9.2 Der Objektspeicher von *P-Quest*

In *P-Quest* gibt es keine Unterscheidung zwischen persistenten und temporären Daten. Jedes Objekt kann persistent gemacht werde. Das Kriterium für die Persistenz ist die Erreichbarkeit vom Hauptprogramm aus. Bei diesem Hauptprogramm handelt es sich meistens um die interaktive Compilerumgebung (Es sind jedoch auch Applikationsprogramme möglich.). Wenn das Hauptprogramm die Compilerumgebung ist, können alle benannten Top-Level-Objekte und alle von diesen aus (auch transitiv) erreichbaren Objekte persistent gemacht werden. Mögliche Top-Level-Objekte sind Werte, Typen, Kinds, aber auch ganze Schnittstellen und Module.

Um diese Objekte persistent zu machen, muß eine explizite Stabilisierung des Objektspeichers durchgeführt werden. Die Vorgehensweise beim Anlegen und Stabilisieren eines persistenten Speichers wird im folgenden beschrieben.

9.2.1 Das Anlegen eines persistenten Speichers

Jeder Benutzer legt zunächst seinen privaten persistenten Speicher an. Dies geschieht über den Befehl *PQFormat*, dem die gewünschte Größe des Speichers als Parameter übergeben wird. Auf weitere Parameter und die genaue Funktionsweise des Befehls wird hier nicht eingegangen.

Nach dem Anlegen eines privaten, persistenten Speichers muß dieser durch den Befehl

```
PQInit
```

initialisiert werden. Dieser Befehl lädt eine Reihe von vordefinierten *P-Quest*-Objekten (wie z.B. die eingebauten Ausnahmen) in den persistenten Speicher.

Durch den Befehl

```
PQuest.NewQuest.qm
```

wird das interaktive *P-Quest*-System in den persistenten Speicher geladen. Außerdem werden alle in der Datei *Library.qst* aufgeführten Schnittstellen und Module importiert.

[9] Diese Typüberprüfung findet in *P-Quest* (Version 14) nicht statt.

Man befindet sich nach Ausführung der obigen Befehle in der interaktiven *P-Quest*-Umgebung. Der Zustand des persistenten Speichers kann jetzt durch die folgenden Befehle[10] persistent gemacht werden.

import *store*
store.stabilise()

Diese Schritte müssen nur beim Anlegen, also nur einmal für jeden persistenten Speicher durchgeführt werden.

9.2.2 Persistente Speicher für Applikationen

Wie oben bereits erwähnt, kann man nicht nur für die interaktive Compiler-Umgebung, sondern auch für andere Programme einen persistenten Speicher anlegen. Dazu muß das gewünschte Programm (in Form einer parameterlosen Funktion mit Rückgabewert vom Typ *Ok*) zunächst explizit in eine Datei geschrieben werden. Das Modul *Dynamic* bietet dafür einen speziellen Befehl *bootFile* an.

Die Ausführung der Befehle

dynamic.bootFile(prog "prog.qm" dynamic.portable)

schreibt ein Programm *prog* in eine Datei *prog.qm*[11].

Das Programm kann dann durch den Befehl

PQuest . prog.qm

in seinem persistenten Speicher ausgeführt werden.

9.2.3 Stabilisieren des Speichers und Setzen von Sicherungspunkten

Die Persistenz beruht wie oben bereits erläutert, auf der Erreichbarkeit von Objekten vom Hauptprogramm aus. Um solche Werte, Typen, Kinds usw. persistent zu machen, muß der Speicher explizit *stabilisiert* werden. Dazu stehen zwei Befehle des Standardmoduls *store* zur Verfügung.

store.stabilise()

stabilisiert den Speicherinhalt und

store.halt()

stabilisiert den Speicherinhalt und beendet sofort die Programmausführung.

Die Befehle können auch dazu benutzt werden, um in Programmen Sicherungspunkte (checkpoints) anzulegen. Sie bewirken, daß die vom Programm aus zum Zeitpunkt des Aufrufs erreichbaren Objekte persistent gemacht werden. Wenn das *P-Quest*-System durch Control-D verlassen wird oder es zu einem Absturz des Systems kommt, werden alle Änderungen, die seit dem letzten Stabilisieren an Objekten des persistenten Speicher durchgeführt worden sind, rückgängig gemacht.

Der Aufruf des System durch den Befehl *PQuestRecover* bewirkt dann eine Wiederaufnahme der Ausführung des Programms beim nächstem Befehl hinter dem letztem *store.stabilise* bzw. *store.halt*. Bei einen Aufruf von *PQuest* wird das Hauptprogramm neu gestartet.

[10] Die Befehle werden weiter unten genauer erläutert.

[11] Die übergebene Funktion wird dabei so modifiziert, daß sie mit einer Ausnahme endet. Dies ist aus internen Gründen erforderlich.

9.2.4 Ein Beispiel

Das folgende Beispiel soll zur Veranschaulichung der vorgestellten Konzepte dienen.

```
import
  list :List
  print :Print
  fmt :Fmt
  store :Store
  dynamic :Dynamic

let db = tuple let var l = list.new(:Int) end

let printList(l :list.T(Int)) :Ok =
  print.iter(list.elements(l) fmt.int)

let listProg() :Ok =
  begin
    db.l := list.cons(1 db.l)
    db.l := list.cons(2 db.l)
    db.l := list.cons(3 db.l)
    printList(db.l)
    store.halt()
    db.l := list.cons(4 db.l)
    printList(db.l)
  end
```

Die Variable *db* wird an eine modifizierbare Liste gebunden. Die Funktion *listProg* fügt drei Elemente in diese Liste ein, führt ein *store.halt* aus und fügt danach noch ein Element in die Liste ein. Vor dem Aufruf von *store.halt* und am Ende der Funktion wird der aktuelle Inhalt der Liste durch einen Aufruf der Funktion *printList* ausgegeben.

Der folgende Aufruf schreibt das Programm *prog*, wie oben bereits erläutert, in die Datei *prog.qm*.

```
dynamic.bootFile(listProg "prog.qm" dynamic.portable)
```

Ein Start des Programms *prog* durch den Befehl *PQuest . prog.qm* führt zur Ausgabe der Listenelemente:

```
[3, 2, 1]Exception:
```

Der Aufruf von *store.halt* im Programm macht den aktuellen Inhalt der Liste zum Zeitpunkt des Aufrufs persistent und führt zur Beendigung des Programms (deshalb die Ausnahme).

Eine Wiederaufnahme des Programms durch den Befehl *PQuestRecover* liefert das Ergebnis:

```
[4, 3, 2, 1]..ok : Ok
```

Das *ok :Ok* zeigt, daß das Progamm bis zum Ende durchgelaufen ist. Das Element 4 ist neu in die Liste eingefügt worden, während die Elemente 1, 2 und 3 noch vom vorherigen Programmlauf in der Struktur enthalten sind. Der Aufruf von *store.halt* hat sie, wie bereits erwähnt, persistent gemacht.

Wenn man das Programm stattdessen durch den Befehl *PQuest* wieder startet, erhält man das Ergebnis:

[3, 2, 1, 3, 2, 1]Exception:

Da das Programm wieder von vorne gestartet wird, werden noch einmal die gleichen Elemente in die Liste eingefügt. Wie im vorigen Fall sieht man auch hier, daß die Elemente des ersten Programmlaufs noch vorhanden sind.

9.2.5 Löschen durch *Garbage Collection*

Es existiert keine Operation für das explizite Löschen von Objekten aus dem persistenten Speicher. Stattdessen gibt es eine Funktion, die alle Objekte ermittelt, die nicht mehr erreichbar sind, und den von ihnen belegten Speicherplatz wieder freigibt (*Garbage Collection*)

Dieser Vorgang kann zum einen zu jedem Zeitpunkt vom Benutzer durch einen expliziten Aufruf folgender Funktion initialisiert werden.

store.garbageCollect()

Außerdem wird der Vorgang automatisch gestartet, wenn es zu einem Überlauf des persistenten Speichers kommt.

Literatur

[Car89] L. Cardelli. Typeful Programming. Digital Systems Research Center Reports 45, DEC SRC Palo Alto, May 1989.

[Car90] L. Cardelli. The Quest Language and System (Tracking Draft). Digital systems research center, DEC SRC Palo Alto, 1990. (shipped as part of the Quest V.12 system distribution).

[CDG+88] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 Report. Technical Report ORC-1, Olivetti Research Center, 2882 Sand Hill Road, Memlo Park, California, 1988.

[Mat91] F. Matthes. Tycoon Library Manual (*P-Quest* Version). DBIS Tycoon Report 102-91, Fachbereich Informatik, Universität Hamburg, West Germany, October 1991.

[Mü91] Rainer Müller. Sprachprozessoren und Objektspeicher: Schnittstellenentwurf und -implementierung. Master's thesis, Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt, West Germany, November 1991.

[Wir85] N. Wirth. *Programming in Modula-2.* Springer-Verlag, 1985.