Referat

zum Proseminar: Programmiersysteme

Thema:

Serialisierung

von
Benedict Fehringer

Betreuer: Guido Tack
Lehrstuhl: Prof. Dr. Gert Smolka

Inhaltsverzeichnis:

1. Einführung

Während ein Programm läuft werden Daten (Informationsträger) miteinander verbunden, indem sie z.B. auf einander verweisen; dadurch entsteht ein Datengraph. (genaue Definition: siehe unten)
Serialisierung bedeutet, dass ein Datengraph in eine eindimensionale (lineare) – Form gebracht wird, die eindeutig in den Ursprungs-Datengraphen umgewandelt werden kann.

1.1 Anwendung

Serialisierung wird für unterschiedliche Zwecke beansprucht:
Serialisierung wird beim Speichern von Daten benötigt. Der Sinn der Serialisierung besteht darin, dass die Daten (bzw. der Datengraph) in eine speicherungsfähige Form gebracht werden. Außerdem macht das Transferieren von Daten (beispielsweise der Austausch über ein Netzwerk) es zwingend erforderlich, dass die vorhanden Datengraphen zuvor linearisiert werden, weil sie nur dann „transportfähig“ sind. In manchen Programmiersprachen (z.B. Alice, Mozart/Oz (Vorgänger von Alice)) spielt Serialisierung auch beim Kompilieren eine Rolle. Dabei wird der Quelltext durch den Compiler in eine Zwischensprache, eine Art abstrakter Syntaxbaum übersetzt. Dieser wird dann mit Hilfe von Serialisierung abgespeichert.

1.2 Umsetzung der Serialisierung in verschiedenen Sprachen

- CLU (eine von Pascal abgeleitete Sprache) [11]: In CLU wurde schon frühzeitig ein allgemeiner Mechanismus zum Enkodieren von Objekten entwickelt, der dazu dienen sollte, dass diese Objekte über das Netzwerk transferiert werden können.
- Java [2]: Der Pickling-Prozess bei Java dient in erster Linie dazu Daten (Objekte) abzuspeichern bzw. sie (z.B. über ein Netzwerk) zu transferieren. Bei der Ausführung des Picklings gibt es die Möglichkeit des Pruning, d.h. man kann selbst bestimmen, welche Teile eines Objektes gepickelt werden sollen, und welche nicht.
- Microsoft`s .NET Framework [6]: Dieser Serialisierungsmechanismus ist dem von Java sehr ähnlich. Im Gegensatz zu Java bezieht sich aber die .NET Spezifikation ausschließlich auf Interfaces und nicht auf den Mechanismus selbst. Das .NET System bietet Serialisierung in verschiedenen Formaten an: ein unspezifisches binäres Format, das „Simpel Object Access Protocol“ (SOAP, ein XML Format, das für die Inter-Prozess Kommunikation gebraucht wird) und einfach XML.
- Python [8]: In dieser Sprache gibt es zwei Mechanismen: der eine kann zwar einen Code pickeln, aber keine zyklischen Strukturen, und der andere kann mit Zyklen umgehen, aber nicht mit Codes.
- Ruby [9]: R. spezifiziert nichts für einen Serialisierungs-Mechanismus, oder ein Pickle-Format; wobei sehr wohl die Möglichkeit von Pickling besteht.
- SML/NJ [10]: Übersetzte SML-Module werden (ähnlich wie in Alice) als Pickles gespeichert. Allerdings steht der Pickling-Mechanismus nur dem Compiler zur Verfügung, nicht aber dem normalen Programmierer.
- Alice [3]: Noch viel ausgeprägter als bei NJ und SML besteht bei Alice die Möglichkeit eines separaten Kompilierens; darüber hinaus wird bei Alice das Kompilieren zum Speichern und Transferieren benötigt.
- Ocaml [7]: Genauso wie in Python und Ruby, nur, dass es nicht möglich ist Codes zu pickeln.
- Mozart/Oz [4]: M. benutzt Pickling zum Speichern und Transferieren von Daten.

2. Datengraphen und abstrakter Speicher

Datengraphen sind Modelle für Objekte, die ein Programm erzeugt, während es läuft. Die folgenden Definitionen sind so allgemein, dass sie platform-unabhängig sind.

2.1 Definition eines gerichteten Graphen

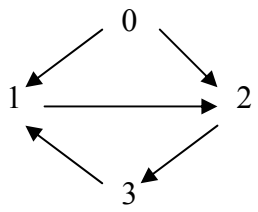
Ein gerichteter Graph ist eine Funktion g, so dass gilt:

$$Ran(g) \phi Dom(g)^*$$

Wenn die Funktion g endlich ist, so wird der Graph ebenfalls als endlich bezeichnet.

Die Elemente von Dom(g) werden Knoten genannt, denen ein Tupel – das ihre Nachfolger beinhaltet – zugeordnet wird:

v	g(v)
0	(1,2)
1	(2)
2	(3)
3	(1)



In diesem Beispiel ist Dom(g) = {0,1,2,3}.

Für einen Datengraph sind mehr Strukturen notwendig. Es werden noch „labels“ (Lab) und „strings“ (Str) benötigt. So kann er wie folgt definiert werden:

2.2 Definition eines Datengraphen

Ein Datengraph ist eine endliche Funktion g, so dass gilt:

$$Ran(g) \phi Lab \times (Str \propto Dom(g)^*)$$

Als Beispiel in Java seien folgende Spezifikationen von Klassen gegeben:

```
class A
{
public string s;
public int i;
public B b;
}
```

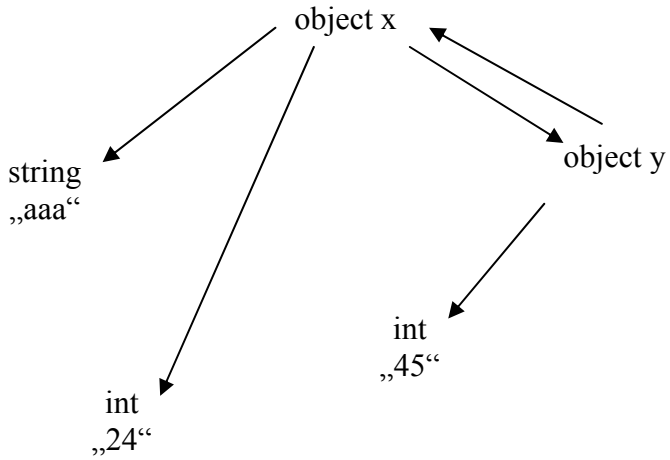
```
class B
{
public int j;
public A a;
}
```

Desweiteren seien die Objekte x, y wie folgt definiert:

```
A x = new A ();
B y = new B ();
x.s = „aaa“;
x.i = 24;
x.b = y;
y.j = 45;
y.a = x;
```

Daraus ergibt sich der nachstehende Datengraph:

Adressen	Label	Inhalt
0	object x	1 | 2 | 3
1	int	„24“
2	string	„aaa“
3	object y	0 | 4
4	int	„45“



2.3 Abstrakter Speicher

Bei der Umsetzung auf der konkreteren Ebene kann durch eine bestimmte Implementierung eine Optimierung erreicht werden. Beispielsweise können spezielle Datenstrukturen durch spezielle Repräsentation dargestellt werden (Zahlen sollten im computer-eigenen Format geschrieben werden, damit das Programm nicht unnötig langsam wird). Außerdem sollte darauf geachtet werden welche einzelnen Informationen für die Darstellung der Knoten von Bedeutung sind. (So sollten auf jeden Fall das entsprechende Label und die einzelnen Nachfolger angegeben werden)

3. Pickles

3.1 Definitionen

Ein *Pickle* ist eine lineare, externale, platform-unabhänige Repräsentation eines Datengraphen. Ein Pickle ist ein String, der genügend Informationen zur Rekonstruktion des originalen Datengraphen enthält.

Pickling bezeichnet den Vorgang, bei dem ein Datengraph in einen Pickle umgewandelt wird.

Unpickling bezeichnet den Vorgang, bei dem aus einem Pickle ein Datengraph konstruiert wird.

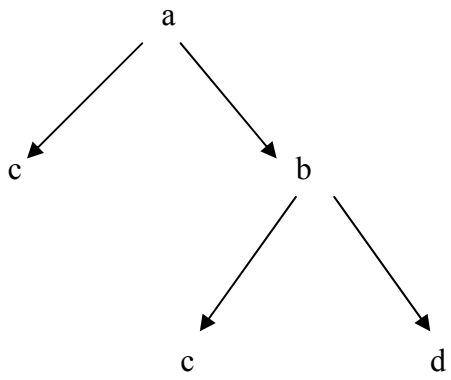
3.2 Unpickling

Im Allgemeinen ist es so, dass der Vorgang des Unpickling sehr häufig angewendet wird, auch ohne in direktem Zusammenhang mit einem vorherigen Pickling zu stehen. Dies liegt daran, dass es Sinn macht, einen Datengraphen aus bestimmten Instruktionsvorschriften zu konstruieren. Dagegen macht es wenig Sinn, einen Datengraphen zu pickeln, ohne dass er später jemals wieder unpickelt wird. Aus diesem Grund soll zunächst hauptsächlich der Vorgang des Unpicklings beschrieben, und erst danach auf das Pickling eingegangen werden.
Für beide Vorgänge (Pickling und Unpickling) gibt es grundsätzlich zwei verschiedene Arten der Durchführung: Bottom-up und Top-down. Im folgenden wird zunächst nur der Bottom-up-Mechanismus beschreiben. Im Anschluß an die Darstellung wird kurz auf den Top-down-Mechanismus eingegangen.

3.2.1 Konstruktion eines Baums

Der gewöhnliche Weg der Implementierung ist ein stack-basierter Interpreter, der wie folgt funktioniert: Seine Kommandos bestehen aus einem Paar (l,n), das ein Label und eine Zahl beinhaltet. Das Label ist das Label des Knotens und die Zahl gibt die Anzahl der Nachfolger des Knotens an. Wenn nun ein bestimmtes Kommando (bezogen auf einen Knoten a) gegeben wird, so werden n Knoten (die Kinder von a), die auf dem Stack liegen, „gepoppt“ und der Knoten a oben auf den Stack gelegt. Am Ende der Konstruktion enthält der Stack nur noch den Wurzel-Knoten.
Wenn Knoten hinzu genommen werden sollen, die Strings beinhalten, so kann dies einfach realisiert werden, indem die Kommandos um die Paare (l,s) erweitert werden, bei denen l das Label des Knotens ist und s der String, den der Knoten enthält. Der Knoten selbst wird dann einfach auf den Stack „gepusht“.

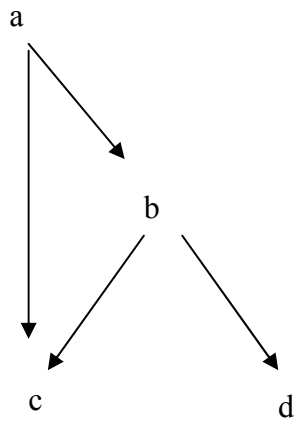
Instruktion	# Nachfolger
c	0
c	0
d	0
b	2
a	2



3.2.2 Konstruktion eines azyklischen Graphen

Die Konstruktion eines azyklischen Graphen funktioniert genauso, wie die eines Baumes nur muss der zusätzliche Fall berücksichtigt werden, dass ein Knoten ein shared-Knoten ist. (Ein Knoten wird als shared-Knoten bezeichnet, wenn er mehr als einen Vorgänger besitzt, oder wenn er die Wurzel ist und mindestens einen Vorgänger besitzt.) Wenn dies eintritt, so wird der entsprechende Knoten mit Hilfe eines STORE-Befehls in ein Register gespeichert, und dann, wenn er einem anderen Vorgänger zugewiesen wird mit Hilfe eines LOAD-Befehls aus dem Register wieder auf den Stack „gepusht".

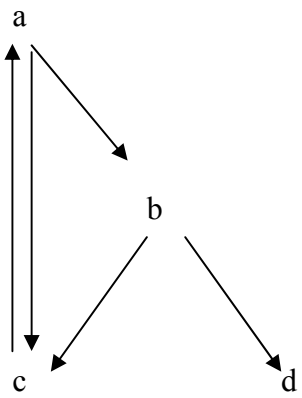
Instruktion	# Nachfolger
c	0
STORE 0	-
LOAD 0	-
d	0
b	2
a	2



3.2.3 Konstruktion eines zyklischen Graphen

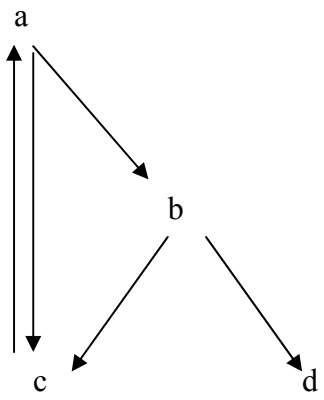
Bei dieser Konstruktion werden die gleichen Interpreter-Befehle benutzt wie beim azyklischen Graphen. Sie werden jedoch um zwei zusätzliche Befehle erweitert, um einen Zyklus darstellen zu können. Ein Knoten des Zyklus wird durch einen PROMISE-Befehl erzeugt und in einem Register abgespeichert, wobei der – durch diesen Befehl – erzeugte Knoten zwar das gleiche Label und die gleiche Anzahl von Nachfolgern besitzt, jedoch noch keine Verbindung zu den Kindern hergestellt ist. Dies geschieht erst durch den FULFIL-Befehl, der aus dem Register den Knoten wieder auf den Stack „pusht", wenn die Kinder des Knotens – in geordneter Reihenfolge – zu oberst auf dem Stack liegen.

Instruktion	# Nachfolger
PROMISE 0 a	2
c	0
STORE 1	-
LOAD 1	-
d	0
b	2
FULFIL 0	2

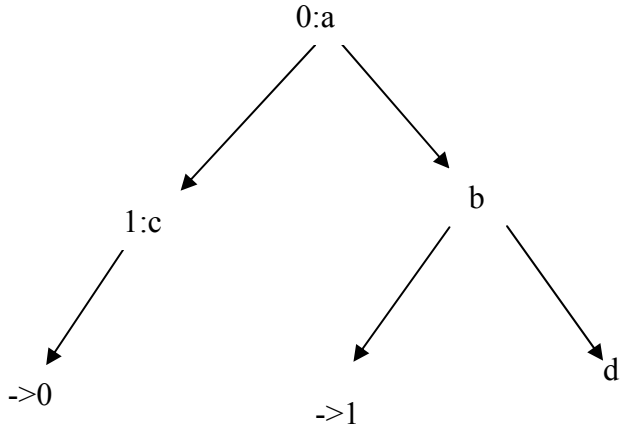


3.3 Pickling eines Datengraphen

Der erste Schritt beim Pickling geschieht durch eine Umformung des Datengraphen in einen „Pickle-Baum“. Hierbei werden alle shared-Knoten v_i in einen index-Knoten, der das Label i:l (wenn l das Label von v_i ist) und in eine Menge von kinderlosen reference-Knoten, die das Label ->i besitzen aufgeteilt. Der index-Knoten wird ein Kind eines (beliebigen) Vorgängers von v_i, die reference-Knoten werden jeweils ein Kind der restlichen Vorgänger.

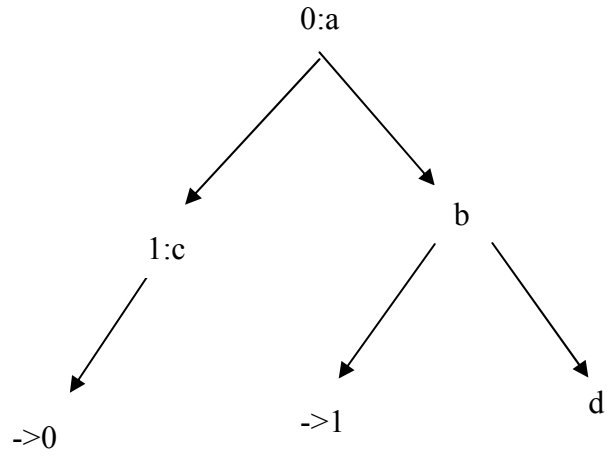


Zyklischer Graph



Pickle-Baum

Im nächsten Schritt wird der nun gewonnene „Pickle-Baum“ in eine Postorder-Linearisierung umgewandelt:



Pickle-Baum

Instruktion	# Nachfolger
-> 0	-
1:c	1
-> 1	-
d	0
b	2
0:a	2

Postorder-Linearisierung

Im letzten Schritt wird die Postorder-Linearisierung des Pickle-Baums wie folgt in einen Pickle umgewandelt:
Alle Knoten, bei denen der index-Knoten vor den reference-Knoten auf den Stack gelegt wird, können durch STORE- und LOAD-Instruktionen dargestellt werden. Alle Knoten, bei denen der index-Knoten nach dem ersten reference-Knoten auf den Stack gelegt wird, können nur durch PROMISE- und FULFIL-Instruktionen dargestellt werden. Alle anderen Knoten, die nur einmal im Stack vorkommen, können einfach übernommen werden.

Instruktion	# Nachfolger
-> 0	-
1:c	1
-> 1	-
d	0
b	2
0:a	2

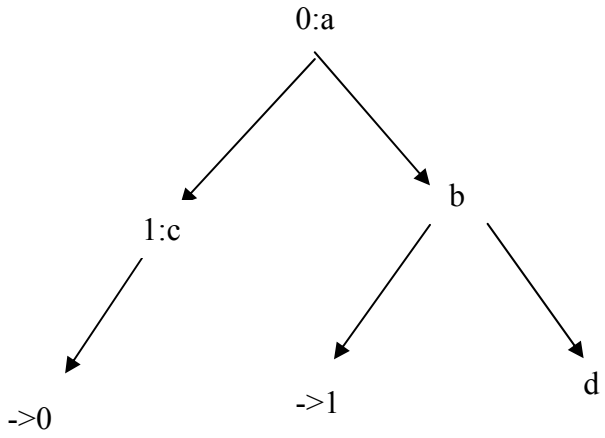
Postorder-Linearisierung des Pickle-Baums

Knoten	# Nachfolger
PROMISE 0 a	2
c	1
STORE 1	-
LOAD 1	-
d	0
b	2
FULFIL 0	2

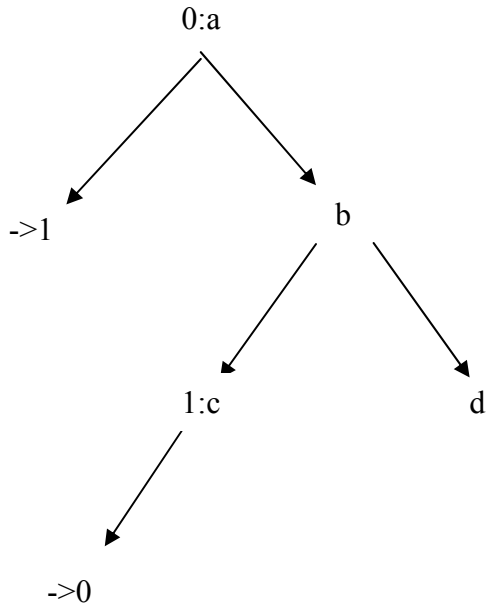
Der resultierte Bottom-up-Pickle

Eigenschaft der Präorder-Linearisierung:

Bei der Präorder-Linearisierung wird aus dem resultierenden Pickle im Gegensatz zum Pickle, der durch Postorder-Linearisierung erzeugt wird, der Datengraph durch den Top-down-Mechanismus konstruiert. Durch Vertauschen der index-Knoten und reference-Knoten (siehe unten) eines Pickle-Baumes, kann immer eine Präorder-Linearisierung gefunden werden, so dass man ohne PROMISE- und FULFIL-Instruktionen auskommen kann, also nur STORE und LOAD benötigt.



=



3.4 Implementier-Details

Die Implementierung eines Algorithmus zum Pickling bzw. Unpickling kann mit Hilfe der Depth First Search (DFS) realisiert werden. Zum einen können dadurch alle shared Knoten eines Graphen gefunden werden, um den Pickle-Baum zu erzeugen, und zum anderen können alle Knoten des Pickle-Baums systematisch in den Pickel überführt werden. Somit kann der

Graph in einem Schritt gepickelt werden. Außerdem kann mit Hilfe des folgenden Algorithmus die maximale Stack-Höhe ermittelt werden, damit nicht unnötig Speicherplatz blockiert werden muß.

$$\text{height}(v) = \begin{cases} 1, \text{ wenn v ein Blatt ist} \\ \max\{\text{height}(v_0) + 0, \ldots, \text{height}(v_n) + n\}, \text{ wenn v die Kinder } (v_0, .. v_n) \text{ besitzt} \end{cases}$$

4. Realisierung in Java und Alice

4.1 Das Java-Objekt-Modell

In Java gibt es verschiedene Komponenten, die bei den nachfolgenden Beispielen von Bedeutung sind:

Klassen:

Eine Klasse beschreibt die Eigenschaften der Objekte und gibt somit den Bauplan an. Jedes Objekt ist ein Exemplar einer Klasse.

Eine Klasse definiert:

- Felder (Attribute, Variablen)
- Methoden (Operationen, die Funktionen einer Klasse)
- Weitere Klassen (innere Klassen)

Objekte:

O. beschreiben die Daten. O. beziehen sich auf eine bestimmte Klasse und sind genau durch die in der Klasse definierten Eigenschaften festgelegt. Dies bringt den Vorteil mit sich, dass Änderungen gut möglich sind, da die Eigenschaften von den Objekten abgekapselt sind.

4.2 Bsp für Pickling in Java [5]

Der einzige Unterschied, der zwischen der Erzeugung einer „normalen“ Klasse und einer Klasse, deren Objekte gepickelt werden sollen besteht, ist der, dass bei der letzteren das `java.io.Serializable` Interface implementiert werden muß (Zeile 20).

```
10 import java.io.Serializable;
20 public class A implements Serializable
30 {
40 public int i;
50 public string s;
60 }
```

Um nun den eigentlichen Pickling-Vorgang durchzuführen, muß zunächst die `ObjectOutputStream` Klasse geladen werden, die das Protokoll erledigt. Die eigentliche Arbeit wird in Zeile 80 gemacht, in der die `ObjectOutputStream.writeObject()` Methode den Serialisierungsvorgang anstößt und somit das Objekt linearisiert wird und unter „aa.ser“ gespeichert wird.

```
10 public class FlattenA
20 {
30 public static void main(String [] args)
40 {
50 A a = new A();
60 FileOutputStream fos = new FileOutputStream(“aa.ser“);
70 ObjectOutputStream out = new ObjectOutputStream(fos);
80 out.writeObject(a);
90 out.close();
100 }
110 }
```

Das Unpickling erfolgt mit dem Aufruf der `ObjectInputStream.readObject()` Methode in Zeile 120. Mit dieser Methode wird ein neues Objekt erzeugt, das der gleichen Klasse angehört wie das ursprüngliche und die gleichen Werte in den Feldern stehen hat. (dies wird sicher gestellt, indem die Klasse (in diesem Fall Klasse A) überprüft wird). Wobei zu Beachten ist, dass beim Pickling nur der Status des Objektes, aber nicht die Objekt-Klasse oder deren Methoden gespeichert werden.

```
10 public class InflateA
20 {
30 public static void main(String [] args)
40 {
50 A a = null;
60 FileInputStream fis = null;
70 ObjectInputStream in = null;
80 try
90 {
100 fis = new FileInputStream("aa.ser");
110 in = new ObjectInputStream(fis);
120 a = (A)in.readObject();
130 in.close();
140 }
150 catch(IOException ex) {ERROR!!!}
160 catch(ClassNotFoundException ex) {ERROR!!!}
170 }
180 }
```

Außerdem hat man bei Java noch die Möglichkeit selbst zu bestimmen, was gepickelt werden soll und was nicht. Entsprechende Klassen, Methoden oder Attribute können mit Hilfe des Schlüsselwortes „`transient`“ markiert werden, damit sie nicht mit gepickelt werden. Diesen Vorgang bezeichnet man als Pruning.

```
10 import java.io.Serializable;
20 public class A implements Serializable
30 {
40 transient public int i;
50 public string s;
60 }
```

4.3 Bsp. für Pickling in Alice [3]

Als Beispiel sei folgende Signatur und Struktur gegeben:

```
signature NUM =
sig
    type t
    fun fromInt : int -> t
    fun toInt   : t -> int
    fun add     : t * t -> t
end
structure Num :> NUM =
struct
    type t = int
    fun toInt n   = n
    fun fromInt n = n
    val add       = op+
end
```

Pickling der Struktur:

```
Pickle.save: string * package -> unit

Pickle.save ("Num." ^ Pickle.extension, pack Num :> NUM)
val it : unit = ()
```

Die Funktion „pack“ wandelt die Struktur „Num“ in ein package (Dieses enthält den Pickle der Struktur und Informationen über die Signatur) um, das durch „Pickle.save“ unter dem Filename „"Num." ^ Pickle.extension“ abgespeichert wird.

Unpickling der Struktur:

```
Pickle.load: string -> package

structure Num' = unpack Pickle.load ("Num." ^ Pickle.extension) : NUM
structure Num' : NUM
```

Hier wird zunächst durch die Funktion „Pickle.load“ das package geladen und dann mit der Signatur „NUM“ verglichen; wenn der Signaturtyp der „gepackten“ Struktur damit übereinstimmt, wird die Struktur mit Hilfe von „unpack“ entpackt.

Wobei zu beachten ist, dass Num‘ als eine andere Struktur als Num interpretiert wird, und somit auf Type-Validität geachtet werden muß:

```
Num'.add (Num.fromInt 4, Num.fromInt 5)
1.0-1.39: argument type mismatch:
   t * t
does not match argument type
   Num'.t * Num'.t
because type
   Num.t
does not unify with
   Num'.t
```

5. Literaturverzeichnis

1. Guido Tack, *Linearisation, Minimisation and Transformation of Data Graphs with Transients*. Diplomarbeit, Saarbrücken, Mai 2003
2. Roger Riggs, Jim Waldo, Ann Wollrath Sun Microsystems, Inc., *Pickling State in the Java™ System*, Toronto, Ontario, canada, June 1996
3. The Alice Project. Available from http://www.ps.uni-sb.de/alice, 2003. Homepage at the Programming Systems Lab, Universität des Saarlandes, Saarbrücken.
4. The Mozart Consortium. The Mozart programming system. Available from http://www.mozart-oz.org, 2003.
5. Java Object Serialization Specification. Available from http://java.sun.com/j2se/1.4/docs/guide/serialization/, 2001.
6. Microsoft. Microsoft .NET. Available from http://www.microsoft.com/net, 2003.
7. The OCaml programming system. Available from http://www.ocaml.org, 2003.
8. The Python programming language. Available from http://www.pyhton.org, 2003.
9. The Ruby programming language. Available from http://www.ruby-lang.org, 2003.
10. A. W. Appel and D. B. MacQueen. Separate compilation for Standard ML. In *Proceedings of the ACM SIGPLAN '94 conference on Programming language design and implementation*, pages 13–23. ACM Press, 1994.
11. B. Liskov and St. Zilles. Programming with abstract data types. In *Proceedings of the ACM SIGPLAN symposium on Very high level languages*, pages 50–59. ACM Press, 1974.