Informatik, Elektrotechnik und Informationstechnik

Bachelorarbeit Nr. 339

Anbindung von SKilL an Haskell

Rafael Harth

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. Erhard Plöderer
Betreuer/in:	Timm Felden
Beginn am:	30. Mai 2016
Beendet am:	29. November 2016
CR-Nummer:	D.3.0, D.3.2, D.3.3

Kurzfassung

Die Sprache SKilL definiert ein Format, in dem große Mengen an Daten hochgradig effizient sowie sprach- und plattenunabhängig serialisiert und deserialisiert werden können. Um dies zu realisieren, müssen Anbindungen in Zielsprachen geschrieben werden, die sowohl die Serialisierung implementieren, als auch eine Schnittstelle zur Bearbeitung der Daten bereitstellen. Die hiesige Arbeit befasst sich mit dem Problem, eine solche Anbindung für Haskell, eine rein funktionale Sprache, zu entwickeln.

Inhaltsverzeichnis

1 Einleitung

Mit dem Ziel, große Datenmengen zeit- und speichereffizient sowie sprach- und plattenunabhängig zu serialisieren, wurde 2013 die Sprache SKilL[1] von Timm Felden, wissenschaftlichem Mitarbeiter an der Universität Stuttgart, im Rahmen seiner Doktorarbeit entwickelt [Tim].[2] Da SKilL explizite Unterstützung für den Umgang mit den erzeugten Daten vorsieht, sind explizite Anbindungen zu Zielsprachen notwendig. Bislang auf diese Weise unterstützt werden Java [Tim16b], C [Fab14] und Ada [Tim16b].

1.1 Terminologie

Im Rahmen dieses Dokuments und des entwickelten Produkts erweist es sich als notwendig, eine Reihe von ähnlichen Elementen zu referenzieren. Um Verwechslungen vorzubeugen, folgt hier eine Liste aller für diesen Zweck definierten Begriffe. Jedes Auftauchen eines solchen ist im weiteren Verlauf des Dokuments *kurisv* hervorgehoben.

- *Generator:* das entwickelte Produkt, welches die Anbindung an Haskell realisiert
- *Generierte Anbindung:* das durch Ausführung des Generators resultierende Werkzeug
- *Serialisierungs-Werkzeug:* der Teil der *Generierten Anbindung*, der für die Serialisierung und Deserialisierung zuständig ist, oder der betreffende Teil des *Generators*
- *Schnittstelle:* der restliche Teil der *Generierten Anbindung*, durch den der Benutzer die *Generierte Anbindung* bedienen kann
- *Spezifikationsdatei:* eine Datei mit in SKilL geschriebenem Code.
- *Binärdatei:* eine Datei mit zu Bytecode serialisierten Objekten, die in der Regel den Definitionen einer (unter Umständen nicht mehr existierenden) *Spezifikationsdatei* entsprechen.
- *Mindestanforderungen:* die in der Aufgabenstellung der Bachelorarbeit gestellten Aufgaben

[1] *Serialization Killer Language*

[2] Die aktuelle Version von SKilL ist momentan unveröffentlicht, weshalb im weiteren Verlauf des Dokuments zumeist eine ältere Version als Quelle angeführt wird.

Weiterhin kursiv hervorgehoben sind mit der Serialisierung zusammenhängende Attribute.

Die Verwendung englischer Begriffe impliziert meist einen Eigennamen, in der Regel entweder aus [Tim] oder allgemein in der Informatik gebräuchlich.

1.2 Aufgabe

Ziel der mit diesem Dokument verbundenen Bachelor-Arbeit ist es, die Anbindung an Haskell [Len90] zu programmieren. Verbunden damit ist eine Demonstration, dass eine solche Anbindung auch in einer rein funktionalen Sprache möglich ist, sowie dass sich das Konzept der Lazy Evaluation für diese Aufgabe nutzen lässt. Dabei wird die bestehende Infrastruktur [Tim13a] ausgenutzt, die unter anderem ein Frontend durch eine Zwischendarstellung der *Spezifikationsdateien* in Form von Javacode realisiert, wodurch das Parsen derselben als Aufgabe entfällt. Optional sind Vorschläge zu Sprachanpassungen oder Erweiterungen [Tim16a].

1.2.1 Konkrete Schritte

Eingabe für den zu erstellenden Generator ist eine *Spezifikationsdatei*. Darauf basierend soll Haskell-Code erzeugt werden, der eine möglichst einfache Realisierung folgender Aufgaben ermöglicht:

- Auslesen von Daten aus *Binärdateien*
- Modifikation gelesener Daten
- Schreiben von (modifizierten) Daten in *Binärdateien*[3]

Da ein großer Teil des dafür nötigen Codes von der *Spezifikationsdatei* unabhängig ist, lässt sich die Gesamtaufgabe grob aufteilen in den Teil des *Generators*, der die *Schnittstelle* erzeugt und den, der das *Serialisierungs-Werkzeug* erzeugt. Für letzteren werden bei der Ausführung des *Generators* lediglich eine Reihe von Haskell-Modulen unverändert kopiert. Diese beinhalten etwa zwei Drittel vom gesamten Code des *Generators*.

[3] Die so entstandenen *Binärdateien* müssen gegebenen syntaktischen Richtlinien entsprechen und von anderen *Serialisierungs-Werkzeugen* gelesen werden können, allerdings auch bei gleichen Daten nicht identisch zu eingelesenen *Binärdateien* sein.

2 Grundlagen von SKilL

2.1 Type-System

Dieser Abschnitt bietet einen kurzen Überblick über das Type-System von SKilL [Tim, §4]. Die Darstellung jedes dieser Typen in der *Generierten Anbindung* durch eine äquivalente Datenstruktur in Haskell ist Teil der *Mindestanforderungen*.

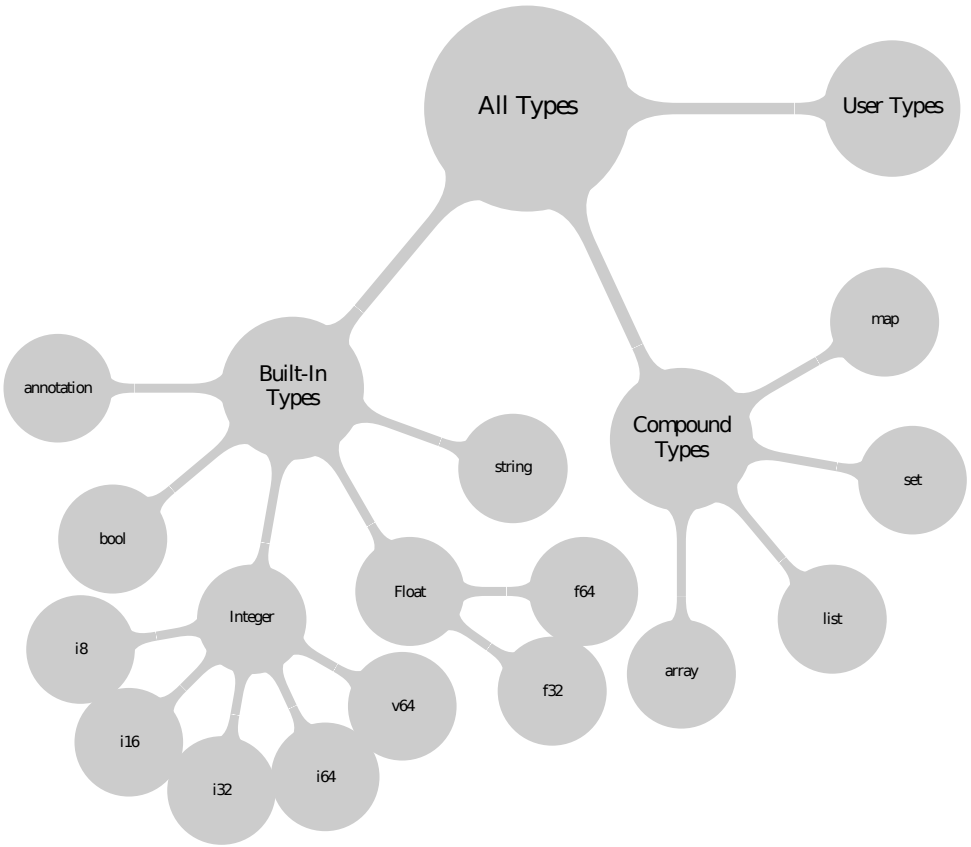

Abbildung 2.1: Das Type-System von SKilL [Tim13b, §4]

Die Integer Types sind signierte ganze Zahlen mit 8, 16, 32 und 64-bit Länge, die jeweils eine direkte Repräsentation in Haskell besitzen. Der v64 Typ ist ein Integer mit variabler Länge,[1] kann aber in Haskell durchgehend mit 64-bit Integern repräsentiert werden. Analog sind Floats Gleitkommazahlen mit einfacher und doppelter Genauigkeit, die ebenfalls direkte Gegenstücke in Haskell besitzen. Ebenso direkt repräsentierbar sind Strings und Booleans.

Lists, Variable Length Arrays und Sets werden identisch behandelt und als Listen dargestellt. Duplikate bei Sets werden durch die *Generierte Anbindung* nicht verhindert. Fixed Length Arrays unterscheiden sich während der Deserialisierung, besitzen danach aber die gleiche Darstellung. Für Maps existiert ein Paket [Unb16].

User Types sind die in der *Spezifikationsdatei* definierten Konstrukte, die jeweils eine beliebige Anzahl (auch 0) an Attributen beliebiger Typen vereinen. User Types können voneinander erben. Mehrfachvererbung ist nicht möglich. Jeder bestehende User Type erweitert das Type-System um einen Typ, den jedes Feld annehmen kann. Solche Felder speichern stets eine Referenz auf den jeweiligen User Type (oder `null`), niemals dessen Inhalt.

Genauso sind auch Annotations Referenzen auf Instanzen von User Types. Ihr Unterschied besteht darin, dass die Identität des User Types nicht Teil der Typinformationen ist, sondern Teil der Daten des jeweiligen Feldes.

Zur Darstellung von Referenzen in Haskell siehe Kapitel 3.

2.2 Aufbau einer Binärdatei

In diesem Abschnitt wird der allgemeine Aufbau einer syntaktisch korrekten *Binärdatei* erläutert [Tim, §6].

In ihrer einfachsten Form besteht eine *Binärdatei* aus einem String-Block, gefolgt von einem Type-Block (Vergleich Abbildung 2.2). Der String-Block beinhaltet eine einfache Auflistung aller in der *Binärdatei* verwendeten Strings. Der Type-Block ist eine vollständige Beschreibung der vorhandenen Felddaten, gefolgt von den Daten selbst. In ihrer allgemeineren Form besteht die *Binärdatei* nicht zwangsläufig aus nur einem Paar, sondern aus einer beliebig langen Abfolge von jeweils einem String-Block und einem Type-Block. Dieser Umstand verändert die Menge der abbildbaren Zustände nicht, erlaubt es aber, bestehende *Binärdateien* effizienter zu erweitern.

Für jegliche Strings, die außerhalb der String-Blocks auftauchen, wird lediglich ein Index codiert.

[1] siehe [Tim13b, §Appendix A]

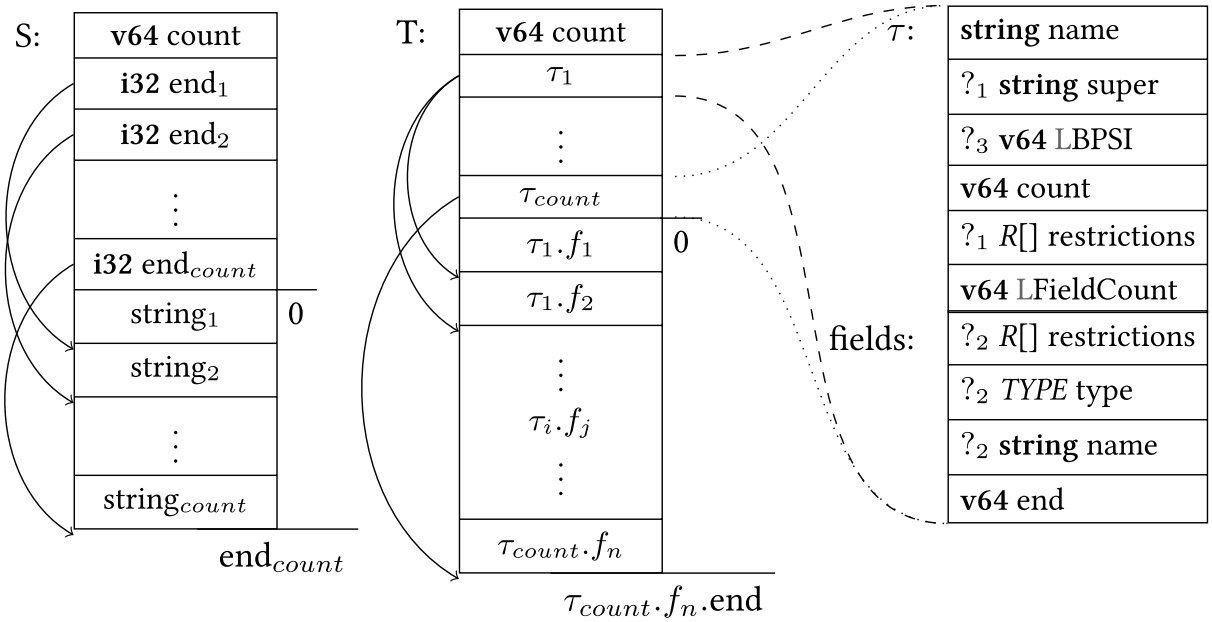

Abbildung 2.2: Illustration des Aufbaus einer *Binärdatei* [Tim]

In den folgenden Abschnitten wird der Aufbau durch Auflisten aller Elemente detailliert erklärt.[2]

2.2.1 Aufbau eines String-Blocks

- Ein Attribut *count*, dessen Wert der Anzahl der *strings* entspricht
- *count* viele *offsets*, die jeweils die Endposition (in Bytes) eines *strings* angeben
- *count* viele utf8-codierte *strings*

2.2.2 Aufbau eines Type-Blocks

- Ein Attribut *count*, dessen Wert der Anzahl der *Type-Descriptors* entspricht
- *count* viele *Type-Descriptors*
- Für jeden *Type-Descriptor* t_i:
  - t_i.*LFieldCount* viele *Field-Descriptors*

[2]Dieser weist leichte Abweichungen mit Abbildung 2.2 auf, da diese nicht der aktuellsten (noch nicht veröffentlichten) Version entspricht.

- Für jeden *Type-Descriptor* t_i:
  - Für jeden *Field-Descriptor* f_j des *Type-Descriptors* t_i:
    - t_i.*count* viele Instanzen vom Feld des *Field-Descriptors* f_j

2.2.3 Aufbau eines Type-Descriptors t_i

- Ein Attribut *name*, das dem Index des Namens des von t_i definierten User-Types entspricht
- Ein Attribut *count*, das der Anzahl der Instanzen des von t_i definierten User-Types entspricht
- Falls t_i der erste *Type-Descriptor* ist, der seinen User-Type beschreibt:[3]
  - Eine Liste von *Restrictions* (Teil der Zusatzanforderungen und von der *Generierten Anbindung* nicht unterstützt).
  - Ein Attribut *superID*, welches dem Index des Supertypen entspricht oder 0, falls keiner vorhanden ist. Der Index ist implizit gegeben durch die Position in der *Binärdatei*; der n-te definierte User-Type[4] hat den Index n.
- Falls ein Supertyp von t_i existiert und $t_i.count > 0$:
  - Ein Attribut *LBPO*, das anzeigt, welche Instanzen des Supertyps zu t_i gehören[5]
- Ein Attribut *LFieldCount*, das der Anzahl der Felder des von t_i definierten User-Types entspricht

2.2.4 Aufbau eines Field-Descriptors f_j von einem Type-Descriptor t_i

- Ein Attribut *fieldID*, eine Zahl zur eindeutigen Identifikation[6]
- Falls f_j der erste *Field-Descriptor* ist, der sein Feld beschreibt:
  - Ein Attribut *name*, das der ID des Namens des von f_j definierten Felds entspricht
  - Ein Attribut *type*, das dem Typ des von f_j definierten Felds entspricht

[3] Weitere *Type-Descriptors* in späteren Type-Blocks können bestehende User-Types erweitern, indem sie Felder und/oder Instanzen hinzufügen.

[4] Die Indizierung läuft über User-Types, nicht *Type-Descriptors*, das heißt *Type-Descriptors*, die bestehende User-Types erweitern, erhalten keinen eigenen Index.

[5] siehe Kapitel 2.3

[6] Field IDs sind notwendig, um es *Type-Descriptors* zu ermöglichen, nur zu manchen Feldern Instanzen hinzuzufügen.

  - Eine Liste von *Restrictions* (ebenfalls nicht unterstützt)
- Ein Attribut *end*, welches die Endposition in Bytes des von f_j definierten Felds relativ zum Anfangspunkt der Daten im *Type-Block* entspricht

2.3 Vererbung und Local Base Pool Offset

SKilL erlaubt die Definition von Vererbungshierarchien. Ein Typ erbt die Felder aller Typen, die in seiner Vererbungslinie aufwärts liegen. In der *Binärdatei* werden Felder jeweils dem Typ zugeordnet, der sie definiert. Um festzulegen, welche Instanzen eines geerbten Felds zu einem Typ gehören, dient das Local Base Pool Offset (*LBPO*). Für eine Illustration dieses Konzepts siehe [Tim13b, §6.3].

Die Vererbung muss ebenfalls in der *Generierten Anbindung* nachgebildet werden, unter anderem weil Referenzen mit bestimmtem Zieltyp auch auf Subtypen dieses Zieltypen zeigen können.

3 Beschreibung der Schnittstelle

Die *Schnittstelle* umfasst all den Code der *Generierten Anbindung*, welcher von der eingelesenen *Spezifikationsdatei* abhängig ist. Die *Schnittstelle* beinhaltet Funktionalität zum lesenden und schreibenden Zugriff auf Daten und bietet Funktionen zur Bedienung des *Serialisierungs-Werkzeugs*.

3.1 Interface

Im Folgenden werden die Signaturen aller für den Benutzer relevanten Methoden der *Schnittstelle* erläutert.

3.1.1 Bedienung, Zustand

Das Design der *Schnittstelle* erlaubt es, sie sowohl interaktiv, durch Eingabe und Auswertung von Ausdrücken mittels GHCI [Has90], als auch durch Schreiben und Ausführen der Befehle mittels einer Kopfprozedur zu bedienen. Für beides kann die generierte Datei "Interface.hs" benutzt werden. In ihr werden alle benötigten Module importiert, und lassen sich daher interaktiv benutzen oder durch geschriebenen Code aufrufen.

Der gesamte Inhalt einer *Binärdatei* entspricht in der Darstellung der *Generierten Anbindung* einer Instanz des Typs `State`. Um das Einlesen und Speichern mehrerer *Binärdateien* zu ermöglichen, werden diese intern durch eine Liste repräsentiert.

```
states :: [State]
```

Mit einem einfachen Methodenaufruf wird die Deserialisierung angestoßen, die *Binärdatei* an dem übergebenen Pfad gelesen und der aus ihr erzeugte Zustand an den Anfang der Liste gesetzt.

```
deserialize :: FilePath -> IO ()
```

Zugriffe auf eingelesene Zustände sind per Index möglich.

```
getState :: Int -> State
```

Da neue Zustände vorne zu der Liste hinzugefügt werden, hat der als letztes eingelesene stets den Index 0. Das Überschreiben von bestehenden Zuständen verläuft analog.

```
1 deserialize  " binaryFile .sf"
2 let  state  =  getState  0
3 visualize  state
4 serialize  state  "G:/ externalPath / binaryFile .sf"
```

Listing 3.1: Über GHCI: eine Binärdatei wird gelesen, ihr Zustand textuell beschrieben und unverändert in eine neue Binärdatei kopiert.

```
writeState :: Int -> State -> IO ()
```

Schließlich lässt sich ein bestehender Zustand ebenfalls wieder serialisieren und in eine neue *Binärdatei* schreiben.

```
serialize :: State -> FilePath -> IO ()
```

Zusätzliche Funktionen erlauben Zugriffe auf die Strings, User-Types oder Type-Descriptors eines Zustands, sowie eine textuelle Beschreibung auf der Konsole.[1]

```
getStrings :: State -> [String]
getUserTypes :: State -> [UserType]
getTypeDescriptors :: State -> [TD]
visualize :: State -> IO ()
```

3.1.2 Datenzugriff

Auf Objekte mit dem Namen `objectName` und dem Typ `ObjectType`[2] kann bei Übergabe des zugehörige Zustands und des Indexes des gewünschten Objekts zugegriffen werden.

```
getObjectName :: State -> Int -> ObjectType
```

Ebenfalls kann auf eine Liste aller Objekte eines Typs direkt zugegriffen werden.

```
getObjectNames :: State -> [ObjectType]
```

Zugriffe auf Felder werden ebenfalls unterstützt.

```
getObjectName_FieldName :: ObjectType -> FieldType
getObjectName_FieldNames ::  State -> [FieldType]
```

Hierbei muss im ersten Fall das Objekt des gewünschten Felds explizit übergeben werden, im zweiten Fall wird aus jedem Objekt des Typs innerhalb des übergebenen Zustands eine Liste vom Feldern zusammengestellt.

[1] Vergleich Abbildung 6.1 auf Seite 32.

[2] Ausdrücke der *Generierten Anbindung* sind in camelCase geschrieben, daher startet der Name des Objekts im Namen des Ausdrucks zwangsläufig mit einem Großbuchstaben.

```
deserialize " somethingGraphical .sf"
let (x,y) = getPunkt (getState 0) 2
setPosition 0 2 (y,x) -- Koordinaten des dritten Punktes werden vertauscht
deletePosition 0 3 -- Vierter Punkt wird verworfen
makePosition 0 (10, 10) -- Neuer Punkt (10, 10) wird erzeugt
serialize (getState 0) " somethingGraphicalModified .sf"
```

Listing 3.2: Mögliche Zugriffe auf ein Objekt Punkt = (Int32, Int32)

Bei schreibenden Zugriffen wird jeweils nur der Index des zu verändernden Zustands übergeben.

```
setObjectName :: Int -> Int -> ObjectType -> IO ()
```

Analog dazu verläuft das Erzeugen und Löschen von Objekten.

```
makeObjectName :: Int -> ObjectType -> IO ()
deleteObjectName :: Int -> Int -> IO ()
```

Im letzteren Fall wird noch der Index des zu löschenden Objekts übergeben.

3.1.3 Zugriffe in IO Monade, Abkürzungen

Wie obige Beispiele zeigen, fällt bei der Benutzung durch bisherige Methoden einiges an Trivialität an, die unter anderem zu wiederholten Aufrufen von `getState 0` führt. Um Zugriffe zu vereinfachen, gibt es daher für die meisten der im letzten Abschnitt aufgelisteten Methoden noch zwei Alternativen.

Die erste ist durch einen Apostrophen am Ende ihres Namens gekennzeichnet, bei ihr fehlt jeweils der erste Parameter und es wird implizit der Zustand an der Stelle 0 referenziert. Die zweite ist durch zwei Apostrophen gekennzeichnet. Hier wird der Zustand, auf den sich bezogen wird, erst noch erzeugt und an den Anfang der Liste `states` gesetzt. Der erste Parameter wird dafür jeweils durch einen `FilePath` ersetzt.

Weiterhin existieren für einige lesende Methoden mit dem Präfix `get` Alternativen mit dem Präfix `read`. Sie unterscheiden sich darin, dass bei letzteren der zurückgegebene Wert nicht aus der `IO`-Monade herausgenommen wird.

An einigen Stellen führt die Variante mit einem Apostroph zu unären, das heißt parameterlosen Funktionen, wie etwa `getState' :: State`, wenn die Funktion nach obigen Vorschriften funktioniert. Solche Funktionen werden von Haskell aber als statische Ausdrücke interpretiert und nur einmalig ausgewertet, daher werden sie in die Schnittstelle nicht aufgenommen. Für sie existiert stets die `read` Variante.

```
1  procedureA  = do  deserialize  "C:/ filepath / age.sf"
2                    state <-  readState  0
3                    let  ages = getAges  state
4                    print  ages
5
6  procedureB  = do  deserialize  "C:/ filepath / age.sf"
7                    let  ages = getAges '
8                    print  ages
9
10 procedureC  = getAges '' "C:/ filepath / age.sf" >>= print
```

Listing 3.3: Drei Prozeduren mit identischem Effekt für ein Objekt mit Typnamen Age

3.1.4 Objekte und Referenzen

Objekte werden in der *Generierten Anbindung* durch Tupel all ihrer Felder dargestellt. Hat ein Objekt nur ein Feld, so ist seine eigene Darstellung und die seines Feldes identisch. Referenzen werden in Haskell nicht unterstützt;[3] eine Referenz in SKilL hat in der *Generierten Anbindung* daher folgenden Typ:

```
type Ref = (Int, Int)
```

Hierbei referenziert der erste Index den Typ, der zweite das Objekt (beides nur implizit). Es ist unmöglich, den Typ `Ref` komplett zu vermeiden und direkt durch den Typ zu ersetzen, auf den die Referenz zeigt, denn dafür wäre es notwendig, transitiv auch jedem Feld vom Typ `Ref` im Zielobjekt zu folgen, was zu geschlossenen Schleifen führen kann. Ein Beispiel hierfür entsteht bei einem Objekt vom Typ A mit einem Feld a vom Typ Ref, welches auf Objekte vom Typ A zeigt. Es existieren auch zyklische Beispiele ohne direkte Selbstrekursion.

Weiterhin können Referenzen in SKilL mit `null` belegt sein, Werte in Haskell aber nicht. Um Referenzen trotzdem bedingt zu unterstützen, wird beim Lesen von Feldern vom Typ `Ref` der Referenz einmalig gefolgt.

Der Typ des zurückgegebenen Werts einer Methode `M`, die ein Feld `a` vom Typ `Ref` liest, dass auf ein Objekt vom Typ `T` ohne Subtypen zeigt, ist somit `Maybe B`. Für Methoden, die ganze Objekte lesen, wird das selbe Prinzip auf jedes Feld angewandt. Das `Maybe` ist notwendig, um den Fall von `null`-Referenzen zu behandeln; in dem Fall ist der Rückgabewert `Nothing`.[4] Führt dies zu weiteren Referenzen, so werden diese ohne weitere Interpretation als `Ref = (Int, Int)` zurückgegeben.

Besitzt `B` transitiv die Subtypen `T_1` bis `T_n`, $n \geq 1$, so besitzt die Datenstruktur der *Schnittstelle* einen weiteren Typ folgender Gestalt:

[3] Eine Ausnahme hierzu wird in Kapitel 7, Fußnote 3 erläutert.

[4] siehe https://hackage.haskell.org/package/base-4.9.0.0/docs/Data-Maybe.html oder https://wiki.haskell.org/Maybe

```
type A = (Ref) -- Ein Attribut: Referenz auf B
type B = (Ref) -- Ein Attribut: Referenz auf C
type C = (Int16, Int8) -- Zwei Attribute: 16- und 8-bit Integer
getAs' :: [Maybe B]
getBs' :: [Maybe C]
getCs' :: [Maybe (Int16, Int8)]
```

Listing 3.4: Generierter Code für drei Typen A B und C

```
data B' = B'B B | B'T_1 T_1 | B'T_2 T_2 | ... | B'T_n T_n
```

In diesem Fall ist `Maybe B'` der Rückgabewert von `M`. Die `T_1` bis `T_n` in der obigen Definition sind die transitiven Subtypen von `B` und können beliebig komplexe Formen annehmen. Die `B'T_1` bis `B'T_n` sowie `B'B` sind Datenkonstruktoren. Der Typ des Werts, auf den eine Referenz zeigt, kann somit sowohl der Zieltyp der Referenz selbst sein als auch ein beliebiger der Typen, welche in der Vererbungshierarchie unter ihm liegen.

Da die Ziele von Referenzen vom jeweiligen Zustand abhängig sind, bei den betroffenen Methoden der Zustand aber nicht immer als Parameter vorhanden ist, wird vom als letztes eingelesenen Zustand ausgegangen. Dieses Verhalten wird nicht als wünschenswert betrachtet, sondern wurde aus Zeitgründen implementiert.

Follow

Das einmalige Folgen von Referenzen wird intern durch den Aufruf Zielobjekt-spezifischer `Follow`-Methoden verwirklicht. Abbildung 3.4 demonstriert jedoch, dass durchaus Situationen existieren, in denen ein zwei- oder mehrfaches Folgen sinnvoll ist, weshalb die Methoden auch für den Benutzer von Bedeutung sind. Basierend auf dem obigen Beispiel haben die `Follow`-Methoden folgende Gestalt (im Fall von Typen mit Supertypen analog wie oben beschrieben):

```
followB :: Ref -> Maybe B
followC :: Ref -> Maybe C
```

3.1.5 unsafePerformIO

In Haskell wird eine zeitliche Abfolge von Aktionen durch die sogenannte `IO`-Monade realisiert.[5] `IO` bildet unter anderem für jeden bestehenden Typ `TypeName` einen neuen Typ `IO TypeName`. Außerhalb von Ausdrücken, die schlussendlich selbst `IO` Werte abbilden, besteht

[5] siehe vorzugsweise https://en.wikibooks.org/wiki/Haskell/Understanding_monads

die einzige Möglichkeit, einen Typ außerhalb des Kontexts einer "`IO`-Methode" aus seiner Monade herauszunehmen, aus einem Aufruf der `unsafePeformIO` Methode.

Da die gesamte Deserialisierung deterministisch abläuft, wird die Nutzung dieser Methode bei der Beschaffung des Zustands trotz ihres Namens als ungefährlich eingestuft. Trotzdem ist es möglich, die Nutzung von `unsafePerformIO` komplett zu umgehen (mit Ausnahme eines einmaligen Aufrufs zur Konfigurierung des Startzustands von `states` als leere Liste). Dafür kann die Schnittstelle innerhalb einer selbst geschriebenen Methode mit Typ `IO ()` bedient werden, wobei für Zugriffe über die `read`-Methoden ablaufen (Vergleich Listing 3.3).

3.2 Datenstruktur

Die von der Schnittstelle benutzte Datenstruktur hat folgende Gestalt:[6]

```
State     = ([String], [UserType], [TypeDesc])
TypeDesc  = (TypeID, Name, [FieldDesc], [TypeDesc], Restrictions)
FieldDesc = (Name, [Something], FieldRecord)
```

`[UserType]` ist eine Liste der Namen aller in der *Binärdatei* definierten Klassen, die jedoch momentan noch nicht weiter verwendet werden.[7] `Restrictions` und `FieldRecord` beinhalten Daten, die von der Schnittstelle nicht benutzt, aber bei der Serialisierung wieder in eine *Binärdatei* geschrieben werden.

3.2.1 Subtypen

Um Vererbung in der Datenstruktur darzustellen, wurde sie hierarchisch aufgebaut. Das Attribut *typeDescs* von `State` besteht aus genau den Type-Descriptors, deren zugehörige User-Types keinen Supertyp besitzen. Die restlichen befinden sich transitiv in dem Attribut *subTypes* des Type-Descriptors ihres jeweiligen Supertypen.

Auf die mit dieser Darstellung verbundenen Aufgaben und Probleme wird in den Abschnitten 2.3, 3.1.4, 4.1.2, 5.1 und 7 eingegangen. Insbesondere erfordern jegliche Operationen auf der hierarchischen Struktur die Implementierung einer Tiefensuche.

[6] Es ist TypeID = Int, Name = String.
[7] siehe Kapitel 7

```
1  data  Something  =  GRef     Ref
2  |  GBool     Bool
3  |  GInt8     Int8
4  |  GInt16    Int16
5  |  GInt32    Int32
6  |  GInt64    Int64
7  |  GV64      Int64
8  |  GFloat    Float
9  |  GDouble   Double
10 |  GString   String
11 |  GList     [ Something ]
```

Listing 3.5: Ausschnitt der Deklaration des Something Typs

3.2.2 Something

Um die Interpretation der Daten logisch schon während der Deserialisierung abzuschließen, wird ein generischer Datentyp benötigt, der alle möglichen Typen abdeckt. In Haskell kann eine solche Struktur mit dem `data` Schlüsselwort (gegebenenfalls rekursiv) definiert werden. Die tatsächliche Interpretation geschieht aufgrund von Haskell's Lazy Evaluation trotzdem erst auf Anfrage.[8]

Zur Nutzung der Daten müssen die Felder dann lediglich zu ihren Zieltypen gecasted werden. Die benötigten Informationen dafür werden der *Spezifikationsdatei*, beziehungsweise der Zwischendarstellung in Java entnommen und hart im Code der *Schnittstelle* verankert.

[8] siehe Kapitel 4.1

4 Deserialisierung

Die Deserialisierung ist eine Hälfte des von der *Spezifikationsdatei* unabhängigen Teils der *Generierten Anbindung*, der für das Erzeugen der Zwischendarstellung aus einer *Binärdatei* zuständig ist.

4.1 Technischer Ansatz

Zum Parsen von binären Daten stellt Haskell die `Get`-Monade [Len16] zur Verfügung. Ausdrücke innerhalb der Monade sind von dem Typ `Get(AnyType)`. Die Semantik eines solchen `Get`-Ausdrucks ist eine Abfolge von Befehlen, in der Regel das Lesen einer Anzahl von Bytes und die Vereinigung derselben zu einem Wert des umschlossenen Typs (`AnyType`). Diese Ausdrucke werden zunächst unabhängig von einer Datenquelle definiert und können dann auf einem Element das Typs `ByteString` [Don16] ausgeführt werden, welches einen Vektor binärer Daten darstellt.

Beim Programmieren eines neuen `Get`-Ausdrucks G dient stets der Aufruf bestehender `Get`-Ausdrücke als Mittel, um lesende Befehle zu definieren. Dabei werden die Werte innerhalb der aufgerufenen `Get`-Ausdrücke (logisch betrachtet) aus ihren `Get`-Monaden herausgenommen, sodass sie frei gehandhabt und zu einem neuen Wert kombiniert werden können, dessen Typ dem von G umschlossenen entsprechen muss, und der am Ende der Prozedur stets wieder in die `Get`-Monade hinein gesetzt werden muss, zum Beispiel durch einen Aufruf von `return`. Der so entstandene `Get`-Ausdruck kann wiederum von anderen `Get`-Ausdrücken auf die gleiche Weise benutzt werden, wodurch sich hierarchisch beliebig komplexe Abfolgen aufbauen lassen. Die Basis für all dies sind eine Reihe von vom Paket bereitgestellten Primitiva, die in der Regel einfache Werte einlesen.

Für ein Beispiel siehe Listing 4.1. Hier werden mit Hilfe des Primitivums `getInt8`, (Typ `Get Int8`), welches ein einzelnes Byte einliest und als signierten Integer zurückgibt, ein neuer `Get`-Ausdruck definiert, der zwei Bytes nacheinander einliest und als Paar von signierten Integern zurück gibt. Nun kann im Kontext anderer `Get`-Ausdrücke `getIntPair` auf die gleiche Weise wie hier das Primitivum `getInt8` benutzt werden.

Um die Deserialisierung der gesamten *Binärdatei* anzustoßen, wird zunächst ihr Inhalt als einzelner `ByteString` eingelesen, dann wird der `Get`-Ausdruck an der Spitze der definierten Hierarchie auf selbigem ausgeführt.

```
1 getIntPair  :: Get (Int8, Int8)
2 getIntPair  = do i1 <-  getInt8
3                  i2 <-  getInt8
4                  return (i1, i2)
```

Listing 4.1: Beispiel eines selbst programmierten Get-Ausdrucks

Lazy Evaluation wird bei diesem Ansatz auf ganzer Linie unterstützt. Hierfür muss lediglich die "Lazy Version" des `ByteString` Pakets benutzt werden. Die Deserialisierung eines Type-Descriptors ist stets von allen vorherigen abhängig, daher ist Lazy Evaluation für Metadaten[1] nur sehr bedingt effektiv, und ist in der konkreten Implementierung der *Generierten Anbindung* völlig ohne Effekt, was bedeutet, dass die Metadaten zwar lazy ausgewertet werden, jedoch trotzdem garantiert vor jeglichem Einlesen von Felddaten. Bei großen *Binärdateien* stellen die Metadaten jedoch nur einen relativ geringen Teil dar und für die Felddaten ist Lazy Evaluation effektiv.

4.2 Strategischer Ansatz

Weitestgehend wurden folgende Richtlinien befolgt:

- Beginne das Einlesen chronologisch; speichere alle benötigten Informationen.
- Strebe die für die *Schnittstelle* bestmögliche Datenstruktur an.[2]
- Nehme Modifizierungen der momentanen Datenstruktur, welche Schritte hin zur endgültigen Struktur darstellen, jeweils zum frühest möglichen Zeitpunkt vor.

Einige Optimierungen, wie etwa das Verwerfen der Offsets eines String-Blocks, sind unmittelbar nach ihrem Gebrauch möglich. Andere, wie etwa die Zuweisung der Daten-Blöcke zu ihren jeweiligen Field-Descriptors können erst zu einem späteren Zeitpunkt vorgenommen werden.

4.2.1 Transformationen

Da die Datenstruktur der Schnittstelle sich von der Darstellung in der *Binärdatei* unterscheidet, haben diese Prinzipien einige nicht triviale Implikationen. Insbesondere haben sie die Implementierung folgender Transformationen der Datenstruktur nach vollständigem Parsen der *Binärdatei* motiviert:

[1] Mit Metadaten sind hier sämtliche Inhalte des Type-Blocks abzüglich der Felddaten gemeint. Bei kleinen *Binärdateien* bilden sie häufig noch über die Hälfte des gesamten Inhalts.

[2] siehe Kapitel 3.2

1. Type-Descriptors, die bestehende User-Types erweitern, werden mit dem den Typ ursprünglich definierenden Type-Descriptors vereinigt. Insbesondere werden hierbei auch Field-Descriptors gleicher Felder sowie deren zugehörige Daten vereinigt. Hierdurch wird eine 1-zu-1 Beziehung zwischen Type-Descriptors und User-Types hergestellt.
2. Die zu diesem Zeitpunkt bereits korrekt zugewiesenen Daten der Field-Descriptors werden durch das Ausführen des im letzten Abschnitt beschriebenen `Get`-Ausdrucks interpretiert (dies ändert nichts an der Lazy-Evalution desselben).
3. Die lineare Struktur wird zu einer hierarchischen transformiert, indem jedem Typ eine Liste seiner Subtypen zugewiesen wird. Die betroffenen Subtypen werden dabei aus der oberen Ebene entfernt. Zusätzlich wird jedes Feld f_i jenem Type-Descriptor zugewiesen, der den Typ des Objekts beschreibt, zu welchem f_i gehört, anstatt wie ursprünglich jenem, der f_i erstmals definiert.

Bewertung

Unter dem Aspekt einer möglichst einfachen Implementierung der geforderten *Mindestanforderungen* werden die oben beschriebenen Optimierungen rückblickend als Fehler eingestuft. Der benötigte Aufwand erwies sich als höher als ursprünglich erwartet, zumal für die Serialisierung die ursprüngliche Struktur auch wieder hergestellt werden muss.[3] Bei der Implementierung von Referenzen[4] erwies sich die veränderte Struktur sogar als hinderlich. Vereinfacht wird dagegen das Hinzufügen, Löschen und Ändern von Objekten.

Eine weitere positive Auswirkung besteht in der Verschmelzung der verschiedenen String-Blocks und Type-Blocks der *Binärdatei* (falls vorhanden). Diese verleiht der Anbindung die einzigartige Eigenschaft, eingelesene und wieder geschriebene *Binärdateien* zu "komprimieren". Insbesondere wird garantiert, dass nur jeweils ein String-Block und Type-Block vorhanden ist sowie dass für jeden definierten Typ nur genau ein Type-Descriptor vorhanden ist.

4.3 Interpretation von Felddaten

Die benötigten Informationen zur Interpretation sämtlicher Felddaten sind in dem Attribut *type* eines Field-Descriptors angelegt.[5] Diese stehen im Deserialisierungs-Prozess ausnahmslos vor den tatsächlichen Daten. Da Felddaten selbst als `ByteStrings` vorliegen, legt dies den Ansatz nahe, die Informationen zur Interpretation der Felddaten wiederum in einen `Get` Ausdruck umzuwandeln. Im Idealfall ist in dieser Form keine weitere Logik mehr notwendig,

[3] siehe Kapitel 5.1
[4] siehe Kapitel 3.1.4
[5] siehe Kapitel 2.2.4

um die Interpretation der Daten zu einem späteren Zeitpunkt tatsächlich vorzunehmen. Um das Problem des unbekannten Datentyps zu lösen, wurde der generische Typ `Something` eingeführt.[6]

Der beschrieben Ansatz wirft nun das Problem auf, dass die folgenden zwei ähnlichen Semantiken unterschieden werden müssen.

1. Lese die Typ-Informationen eines Feldes ein. Lese weiterhin den Wert des Feldes nach den beschriebenen Informationen ein. Produziere einen `Get` Ausdruck, der bei Ausführung den eingelesenen Wert zurückgibt.
2. Lese die Typ-Informationen eines Feldes ein. Produziere einen `Get` Ausdruck, der bei Ausführung den Wert des Ausdrucks nach den beschriebenen Informationen liest und zurückgibt.

Ihr Unterschied besteht darin, dass bei der ersten Semantik der Wert des Ausdrucks direkt auf die Typ-Informationen folgend eingelesen wird, während bei der zweiten der Wert erst als Teil der Ausführung des erzeugten `Get` Ausdrucks gelesen wird. Das erste Verhalten ist für Konstanten gewünscht, das zweite für sonstige Typen. Beide Semantiken führen zu Ausdrücken des Typs `Get (Get Something)`, es ist daher nicht unmittelbar ersichtlich, welches Verhalten eine Implementierung haben wird.

Tatsächlich ist es möglich, beide Verhalten korrekt zu implementieren, ohne dass dabei bei Ausführung des resultierenden `Get` Ausdrucks eine Fallunterscheidung vorgenommen werden muss. Die folgenden Codeausschnitte produzieren aus einem ein Byte langen signierten Integer jeweils einen Ausdruck des Typs `Get (Get Something)` mit der ersten bzw. zweiten Semantik.

```
(return . GInt8) `fmap` getInt8
return (GInt8 `fmap` getInt8)
```

`getInt8` ist ein Primitivum der `Get`-Monade, das ein einzelnes Byte liest und als signierten Integer interpretiert. `getInt8` hat den Typ `Get(Int8)`.

`GInt8` ist der Daten-Konstruktor, der einen Ausdruck des Typs `Int8` zu einem des Typs `Something` verarbeitet. `GInt8` hat den Typ `Int8 -> Something`.

`fmap` ist eine Methode, die einen Wert w innerhalb einer Monade M (Notation hier $M(w)$), sowie eine Funktion f als Paremeter nimmt, und $M\Big(f(w)\Big)$ zurückgibt. Illustrativ nimmt `fmap` den Wert aus der Monade heraus, wendet die Funktion auf ihn an und setzt das Ergebnis wieder in die Monade hinein. `fmap` hat den Typ `m a -> (a -> b) -> m b`.

`return` ist eine Methode, die im Kontext einer Monade M einen Wert w auf $M(w)$ abbildet. `return` hat den `Typ a -> M a`.

[6] siehe Kapitel 3.2.1

Die ``-Notation wird benutzt, um eine Methode mit zwei Parametern mittig zu schreiben.

Es ist `ausdruck1 ‘function‘ ausdruck2 = function ausdruck1 ausdruck2`.

Der Punkt schließlich ist die Hintereinanderausführung (auch: Konkatenation) zweier Methoden.

Der Unterschied in der Semantik steckt nun innerhalb des `Get`. Sowohl `getInt8` als auch `return 4` sind (hier) vom Typ `Get Int8`, aber `getInt8` hat die Semantik, ein Byte aus dem `ByteString` zu verarbeiten, auf dem sie ausgeführt wird, während die Funktion `return 4` lediglich einen konstanten Wert in eine `Get`-Monade steckt, unabhängig von dem `ByteString`, auf dem sie ausgeführt wird. Analog dazu hat auch in unserem Beispiel `getInt8` die lesende und `return` die triviale Semantik. Da beim erstmaligen Auswerten des zu bestimmenden Ausdrucks stets das äußere `Get` ausgeführt wird, ist entscheidend, welches der beiden `Gets` im Ausdruck `Get (Get Something)` außen und welches innen steht.

Befehlsabfolge des ersten Ausdrucks `(return . GInt8) ‘fmap‘ getInt8`:

1. Der Wert wird aus dem lesenden `Get` herausgenommen (`fmap`).
2. Der Wert wird nacheinander zuerst zu einem `Something` verarbeitet (`GInt8`), dann in ein triviales `Get` gesetzt (`return`).
3. Der Wert wird wieder in das lesende `Get` gesetzt (`fmap`).

Befehlsabfolge des zweiten Ausdrucks `return (GInt8 ‘fmap‘ getInt8)`:

1. Der Wert wird aus dem lesenden `Get` herausgenommen (`fmap`).
2. Der Wert wird zu einem `Something` verarbeitet.
3. Der Wert wird in das lesende `Get` gesetzt (`fmap`).
4. Der Wert wird in ein triviales `Get` gesetzt (`return`).

Daher wird beim erstmaligen Ausführen des ersten Ausdrucks das triviale `Get` ausgeführt und das lesende bleibt bestehen, während beim Ausführen des zweiten das lesende ausgeführt und das triviale bestehen bleibt. Mit dem gleichen Prinzip lassen sich beide Semantiken für Felddaten implementieren.

5 Serialisierung

Die Serialisierung ist für die Erzeugung einer *Binärdatei* aus einem Zustand der Zwischendarstellung zuständig. Sie ist nach der Deserialisierung die zweite Hälfte des *Serialisierungs-Werkzeugs*.

5.1 Rücktransformationen

Um dem Format der *Binärdatei*[1] zu entsprechen, müssen einige der nach dem Parsen vorgenommenen Transformationen (beschrieben in Kapitel 4.2.1) wieder rückgängig gemacht werden. Dies beinhaltet unter anderem die Redistributierung der Field-Descriptors, sowie das Berechnen von neuen *LBPOs*[2]. Ebenfalls müssen IDs, die Supertypen referenzieren, angepasst werden, da während der Transformationen Typen in die hierarchische Struktur umgeordnet werden, ihre ID in der Binärdatei aber gerade durch ihre implizite Position gegeben ist. Da die Datenstruktur nur logisch betrachtet hierarchisch und in Haskell durch einfache Listen repräsentiert ist, muss selbst die Anzahl der Type-Descriptors explizit berechnet werden.[3]

5.2 In der Schnittstelle fehlende Informationen

Felddaten, Typinformationen und Restrictions sind für die Funktionalität der Schnittstelle nicht relevant, müssen aber trotzdem serialisiert werden. Dies legt die Ansätze “Neuberechnung” und “Aufbewahrung” nahe.

Da sich Typinformationen durch Nutzung der Schnittstelle niemals ändern, wurde für diese letzterer Ansatz verfolgt und implementiert, das heißt, es werden die relevanten Abschnitte aus der *Binärdatei* während der Deserialisierung kopiert und unverändert aufbewahrt. Dies stellt die Anforderung doppelter Verwendung von Abschnitten der *Binärdatei* an die `Get`-Monade sowie zusätzliche Ansprüche an die interpretierende Funktion. Ihr Rückgabewert ändert sich dabei von `Get (Get Something)` zu `Get (ByteString, (Get Something))`. Beides ist möglich

[1] siehe Kapitel 2.2

[2] siehe Kapitel 2.2.3

[3] Dies ist ein Symptom eines weiterreichenden Problems, das ebenfalls zu Schwierigkeiten mit der korrekten Implementierung von Zeigern auf Typen innerhalb einer Vererbungshierarchie geführt hat (siehe Kapitel 7).

```
parseTypeDescription   :: [ String ] -> Get ( ByteString , Get  Something )
parseTypeDescription   s = do b1            <-  lookAhead  g' v64ByteString
                              num           <- g'v64
                              (b2, getter ) <- go s num
                              return (b1 `a` b2, getter )
where go _ 0  = do b <-  lookAhead   getLazyByteString  1
                   v <-  getInt8
                   return (b, ( return ( CInt8 v)))
```

Listing 5.1: Ausschnitt der endgültigen Prozedur zur Verarbeitung von Typinformationen

(siehe Listing 5.1).[4] Bei Restrictions ist der Ansatz der Aufbewahrung alternativlos, da sich diese aus den Felddaten nicht ergeben, wirft allerdings auch keine besonderen Probleme auf.

Für Felddaten wurde eine Neuberechnung implementiert.[5] Dazu wird als Teil der Rücktransformationen die Binärdarstellung sämtlicher Felddaten berechnet und diese anschließend verworfen. Ununterscheidbare Daten mit unterschiedlicher Binärdarstellung (beispielsweise 50 als 8-bit und 16-bit Integer) können mit Hilfe des Datenkonstruktors des `Something` Typs[6] bequem unterschieden werden.

Während der Serialisierung selbst müssen dann in allen drei Fällen lediglich die kopierten beziehungsweise berechneten Abschnitte unverändert eingefügt werden.

[4] Tatsächlich scheinen auch geschachtelte *Get*-Ausdrücke mit dem richtigen Code praktisch beliebig manipulierbar zu sein.

[5] Zunächst wurde auch hier der Ansatz "Aufbewahrung" verfolgt. Da sich Daten durch Nutzen der Schnittstelle durchaus ändern können, wurde angestrebt, die entsprechende Binärdarstellung immer parallel zu synchronisieren. Die daraus resultierende Methode besaß bereits bedeutend größeren Umfang und Komplexität als die die jetzige Implementierung, bevor sie aufgrund von unlösbar scheinenden Problemen verworfen wurde. Rückblickend wäre eine Neuberechnung auch für Typinformationen der bessere Ansatz gewesen.

[6] siehe Kapitel 3.2.2 beziehungsweise Listing 3.5

6 Tests

Um systematisches Testen in Haskell zu ermöglichen, wurde das `HUnit` Paket verwendet [Dea16]. Es bietet die Möglichkeit, Testfälle als Reihe von `IO`-Befehlen zu definieren. Die in ähnlichen Frameworks gewöhnlichen `Assert`-Anweisungen haben ebenfalls den Typ `IO ()`, wodurch sie sich beliebig zwischensetzen lassen. Ausgeführt werden können sowohl einzelne Testfälle als auch mehrere, die als Liste zusammengefasst wurden. Jeder Testfall endet mit dem Resultat "Error" im Falle eines solchen an einem beliebigen Punkt in der Befehlskette, "Failure" im Falle einer Assertion mit unerwartetem Wert oder "Success" in allen übrigen Fällen.

Da es offensichtliche Vorteile bringt, Tests automatisch zu generieren, wurden die durchgeführten Tests in zwei Kategorien unterteilt. Die Funktion des *Serialisierungs-Werkzeugs* ist nicht von einer *Spezifikationsdatei* abhängig, daher wurden Tests hier automatisiert. Für die Schnittstelle dagegen sind spezifische Zugriffe auf bestimmte, in der *Spezifikationsdatei* definierte Felder und Objekte notwendig, daher wurden diese von Hand geschrieben.

6.1 Automatisierte Tests

Zur Erzeugung von Tests für die Deserialisierung wurde ein Modul "TestGenerator.hs" erstellt. Seine Schnittstelle beschränkt sich auf eine einzelne Methode, die für zwei gegebene Pfade im Verzeichnis des zweiten einen Test für jede *Binärdatei* im Verzeichnis des ersten erstellt, sowie eine Kopfprozedur. Die einzelnen Tests definieren jeweils einen Testfall, die Kopfprozedur importiert alle anderen erstellten Dateien als Module, fasst sie zu einer Liste von Testfällen zusammen und führt diese aus. Auf die Verwendung von Assertions wurde aus einer Reihe von Gründen komplett verzichtet:

- Fehler im Code der Deserialisierung führen weit öfter zur irrtümlichen Terminierung des Prozesses als zu fehlerhaften Daten, ein Fall, der als auch ohne die Festlegung von Assertions als Error abgefangen wird.
- Allgemein sinnvolle Assertions sind schwer festzulegen. Insbesondere ist die Existenz eines beliebigen Elements niemals eine universell wünschenswerte Eigenschaft, da selbst eine *Binärdatei*, die nur aus leeren String-Blocks und Type-Blocks besteht, syntaktisch korrekt ist.
- Der Verzicht vermeidet die Notwendigkeit von zur *Binärdatei* passenden *Spezifikationsdateien*.

```
 ---------------
-- Test date --
 ---------------
### Type 0 : date ###

> Field date: [1, 255, 255, 255, 255, 255, 255, 255, 255, 255] -> [GV64 1,
GV64 (-1)]

Number of Subtypes: 0


Cases: 11   Tried: 7   Errors: 0   Failures: 0

 --------------------
-- Test crossNodes --
 --------------------
### Type 0 : node ###

> Field node: [2, 1] -> [GRef (0,1), GRef (0,0)]

Number of Subtypes: 0


Cases: 11   Tried: 8   Errors: 0   Failures: 0
```

Listing 6.1: Ausschnitt der Konsolenausgabe für generierte Tests aus src/test/resources/genbinary/auto/accept

Stattdessen wurde eine Prozedur zur übersichtlichen Darstellung der eingelesenen Daten erstellt, die stets nach beendeter Deserialisierung bei Ausführung eines Tests aufgerufen wird. Dieser Ansatz profitiert von der Verwendung des Pakets noch primär dadurch, dass die Terminierung des Einlesevorgangs einer bestimmten *Binärdatei* nicht zur Terminierung des Gesamtprozesses führt.

6.2 Selbst geschriebene Tests

Um die Funktionalität der *Schnittstelle* sicherzustellen, wurden in dem üblichen Verfahren Tests von Hand geschrieben (mit Assertions). Ihre Funktionsweise unterscheidet sich in keiner bedeutenden Weise von der in anderen Sprachen.

Als Beispiel folgt ein Test, der eine *Binärdatei* liest, erweitert und als eine neue *Binärdatei* speichert, diese dann erneut liest und ihre Elemente überprüft.

```
filePath = "sf/age.sf"

testAppend = TestCase $
   do deserialize filePath
      makeAge' 10
      makeAge' 20
      serialize' "sf/ageAppend.sf"
      deserialize "sf/ageAppend.sf"
      ages <- readAges'
      assertEqual "test all ages after appending" [1,28,10,20] ages
```

6.3 Ergebnisse

Für die Deserialisierung wurden die *Binärdateien* aller genbinary/.../accept Verzeichnisse herangezogen, abzüglich der als "age16.sf" betitelten, da diese zwar funktionieren, aber aufgrund ihrer großen Datenmenge lange Zeit in Anspruch nehmen und die Konsolenausgabe unleserlich machen. Von den übrigen 73 Dateien erzeugen die drei "localBasePoolOffset.sf" genannten jeweils einen Error,[1] die restlichen werden fehlerlos abgehandelt.

Für die Schnittstelle wurde das Lesen, Schreiben, Hinzufügen und Löschen von Dateien erfolgreich getestet sowie die Serialisierung und erneute Deserialisierung einer *Binärdatei* nach Modifikation der Daten (wie im obigen Beispiel). Weiterhin wurden Datenzugriffe auf Dateien mit komplexeren Vererbungshierarchien erfolgreich getestet, das Verfolgen von Referenzen führt jedoch erwartungsgemäß zu einem Error.[2]

Schließlich wurden noch erfolgreiche Tests zu Container-Typen sowie Strings mit Unicode Zeichen durchgeführt.

[1] Der Grund hierfür scheint mit *maps* und/oder Konstanten zusammenzuhängen.
[2] siehe Kapitel 7

7 Nicht erfüllte Mindestanforderungen

Von den in der Aufgabenstellung enthaltenen *Mindestanforderungen* wurde insbesondere die Handhabung unbekannter Feldtypen nicht korrekt implementiert. Das gewünschte Verhalten sieht vor, dass in der *Binärdatei* vorkommende Feldtypen, welche in der *Spezifikationsdatei*, auf deren Basis die *Generierte Anbindung* erstellt wurde, nicht definiert sind, ebenfalls unterstützt werden. Hierfür müsste die Datenstruktur durch das Hinzufügen entsprechender generischer Typen erweitert werden.

Stattdessen setzt die bestehende Anbindung die exakte Übereinstimmung der gelesenen *Binärdateien* mit der Spezifikationsdatei voraus, was unter anderem unerwünscht ist, da sich Spezifikationen jederzeit ändern können. Grund für diesen Mangel war in erster Linie fehlende Zeit.[1] Eine Erweiterung der Datenstruktur sollte ohne über die Aufgabe hinausgehende strukturelle Änderungen möglich sein. Insbesondere wird eine benötigte Liste aller User-Types aus der *Binärdatei* bereits extrahiert.

Weitere Schwachpunkte des *Generators* bestehen vor allem im Umgang mit fehlerhaften Dateien. Momentan werden keinerlei Checks während der Deserialisierung durchgeführt, die sicherstellen könnten, dass die eingelesenen Dateien Sinn ergeben, daher führen Fehler in der Regel entweder zu einer "Verschiebung" der Interpretation auf falsche Daten, was quasizufällig entweder zu einem Fehler während dem Parsen führt (sowie etwa ein vorschnelles Ende der Datei), oder zum Einlesen falscher Daten, die wiederum zu späteren Zeitpunkten schwer nachvollziehbare Fehler hervorrufen können. Von den 21 fehlerhaften *Binärdateien* im genbinary-Verzeichnis werden im Moment 14 ohne Widerspruch eingelesen.

Hier wird der bestehende Code als einfach erweiterbar eingestuft. Die Interpretation logisch zusammenhängender Teile ist im Generator zu `Get`-Ausdrücken zusammengefasst,[2] die jeweils

[1] Diese wiederum geht, neben fehlerhaftem Zeitmanagement und den vorgenommenen Transformationen (siehe Kapitel 4.4), primär auf Schwierigkeiten bei der Benutzung bestehender Infrastruktur zurück, insbesondere der Einrichtung einer funktionierenden Entwicklungsumgebung für Haskell. EclipseFP [JP 09] ließ sich trotz zahlreicher Versuche nicht einrichten; HaskForce [car13] funktionierte ursprünglich, jedoch nicht mehr, nachdem eine erneute Aufsetzung notwendig wurde, weswegen schlussendlich auf Leksah [Lek07] ausgewichen wurde. Weitere Schwierigkeiten sind in Verbindung mit dem Package-Management von cabal [Git06] entstanden. Schließlich lässt sich auch unabhängig von der Aufsetzung ohne Zweifel feststellen, dass eine Entwicklungsumgebung für Haskell, welche eine mit Eclipse vergleichbare Mächtigkeit besitzt, erhebliche Zeitersparnisse eingebracht hätte.

[2] siehe Kapitel 3.1

innerhalb einer Methode definiert sind, in der Regel in einem `do`-Block. Entsprechende Abfragen können problemlos an den richtigen Stellen eingefügt werden.

Ein davon unabhängiger Mangel hat mit dem Support von Referenzen zu tun. Referenzen werden in Haskell nicht brauchbar unterstützt,[3] daher werden diese in der Schnittstelle durch Integer Paare simuliert,[4] wobei der erste Integer den Typ, der zweite die Instanz referenziert.

Die in der *Binärdatei* nur implizit vorhandenen Indexe von Typen werden bei der Deserialisierung explizit abgespeichert, daher kann dem ersten Index korrekt gefolgt werden. Da jedoch bei Typen mit Supertypen die Zuordnung der Felder geändert wird,[5] werden beim Verfolgen von Referenzen in betroffenen Fällen falsche Instanzen richtiger Typen erreicht. Eine Lösung, die aus Zeitgründen nicht mehr implementiert wurde, besteht darin, bei der Implementierung der Transformationen jedem Type-Descriptor zusätzlich eine Liste von 8-bit Integern zuzuordnen, welche alle Instanzen auflistet, die diesem zugeordnet sind. Beim Folgen von Referenzen müssen dann alle Type-Descriptors einer Vererbungshierarchie durchlaufen und der mit dem gewünschten Eintrag ausgewählt werden.

Schließlich existiert noch ein bislang unergründetes Problem mit maps und/oder Konstanten.[6]

[3] Es existiert ein Typkonstruktor `IORef`, der eine Referenz mit modifizierbarem Zustand implementiert. Tatsächlich wurde genau dieser Typ auch benutzt, um die globalen Zustände der Zwischendarstellung zu speichern; der Ausdruck `states` hat den Typ `IORef [State]`. Für kleinere, oft verwendete Referenzen ist dieser Ansatz jedoch völlig ungeeignet.

[4] Vergleich Kapitel 3.1.4

[5] siehe Kapitel 4.2.1

[6] siehe Kapitel 6.3

Literaturverzeichnis

[car13] carymrobbins. *HaskForce*. 2013. URL: http://haskforce.com (zitiert auf S. 35).

[Dea16] Dean Herington. *The HUnit package*. 2016. URL: https://hackage.haskell.org/package/HUnit-1.5.0.0 (zitiert auf S. 31).

[Don16] Don Stewart, Duncan Coutts. *The bytestring package*. 2016. URL: https://hackage.haskell.org/package/bytestring-0.10.8.1 (zitiert auf S. 23).

[Fab14] Fabian Harth. „Plattform- und sprachunabhängige Serialisierung mit SKilL“. German. Diploma Thesis. University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, Nov. 2014, S. 55. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-3665&engl=1 (zitiert auf S. 7).

[Git06] Github. *cabal*. 2006. URL: %5Curl%7Bhttps://github.com/haskell/cabal/graphs/contributors%7D (zitiert auf S. 35).

[Has90] Haskell. *Glasgow Haskell Compiler Interactive*. 1990. URL: https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/ghci.html (zitiert auf S. 15).

[JP 09] JP Moresmau. *EclipseFP*. 2009. URL: http://eclipsefp.github.io (zitiert auf S. 35).

[Lek07] Leksah Community. *Leksah*. 2007. URL: http://leksah.org (zitiert auf S. 35).

[Len16] Lennart Kolmodin. *The binary package*. 2016. URL: https://hackage.haskell.org/package/binary-0.8.4.1 (zitiert auf S. 23).

[Len90] Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, John Hughes, Thomas Johnsson, Mark Jones, Simon Peyton Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, Philip Wadler. *Haskell*. 1990. URL: https://www.haskell.org (zitiert auf S. 8).

[Tim] Timm Felden. „The SKilL Language“. Eine weniger aktuelle Version ist unter ftp://ftp.informatik.uni-stuttgart.de/pub/library/ncstrl.ustuttgart_fi/TR-2013-06/TR-2013-06.pdf veröffentlicht (zitiert auf S. 7–11).

[Tim13a] Timm Felden. *SKilL*. 2013. URL: %5Curl%7Bhttps://github.com/skill-lang/skill%7D (zitiert auf S. 8).

[Tim13b] Timm Felden. *The SKilL Language*. 2013. URL: ftp://ftp.informatik.uni-stuttgart.de/pub/library/ncstrl.ustuttgart_fi/TR-2013-06/TR-2013-06.pdf (zitiert auf S. 9, 10, 13).

[Tim16a] Timm Felden. *Anbindung von SKilL an Haskell*. 2016. URL: %5Curl%7Bhttp://www.iste.uni-stuttgart.de/fileadmin/user_upload/iste/ps/Lehre/dasa/akt_themen/anbindung_skill_an_haskell.pdf%7D (zitiert auf S. 8).

[Tim16b] Timm Felden and Martin Wittiger. „Migrating Bauhaus from IML to SKilL“. In: *Softwaretechnik-Trends*. Bd. 36:2. 2016 (zitiert auf S. 7).

[Unb16] Unbekannt. *The containers package*. 2016. URL: https://hackage.haskell.org/package/containers-0.5.8.1 (zitiert auf S. 10).

Alle URLs wurden zuletzt am 26. 11. 2016 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift