Kompakte Repräsentation von Datentermen

Lasse Züngel

Bachelorarbeit
04/2024

Programmiersprachen und Übersetzerkonstruktion
Institut für Informatik
Christian-Albrechts-Universität zu Kiel

Betreut durch
Prof. Dr. Michael Hanus und M. Sc. Kai Prott

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Weiterhin erkläre ich, dass die digitale Fassung dieser Arbeit, die dem Prüfungsamt per E-Mail zugegangen ist, der vorliegenden schriftlichen Fassung entspricht.

Kiel,

Zusammenfassung

Die deklarative Programmiersprache Curry arbeitet mit Termstrukturen, um zur Laufzeit Daten zu repräsentieren. Häufig ist es notwendig, diese Datenterme persistent zu speichern und zu einem späteren Zeitpunkt wieder einzulesen oder diese Daten einem anderen Programm zur Verfügung zu stellen. Beispielsweise generiert das Front-End eines Compilers aus dem Quelltext eines Programmes eine Zwischendarstellung, die über eine Schnittstelle an weitere Programmkomponenten (partielle Auswerter, Analysetools, Backends, etc.) übergeben wird. Diese Daten können mitunter sehr groß werden, was das Speichern und Einlesen erschwert.

In dieser Arbeit wird eine kompakte Repräsentation von Datentermen in Curry vorgestellt, mit der es ermöglicht wird, große Datenterme effizient zu speichern und das Einlesen deutlich zu beschleunigen. Weiterhin wird ein Werkzeug entwickelt, womit für Typdefinitionen eines Curry-Moduls automatisch Operationen zum Schreiben und Lesen generiert werden. Anschließend wird die Implementierung evaluiert und generell sowie im Kontext von FlatCurry-Programmen mit der aktuellen Ausgangslage verglichen.

Inhaltsverzeichnis

Kapitel 1

Einführung

1.1. Motivation

Im digitalen Zeitalter sind Daten ein zentraler Bestandteil zahlreicher Anwendungen und Systeme. Damit unterschiedliche Programme und Programmkomponenten reibungslos Daten austauschen können, müssen gemeinsame Datenformate bereitstehen, um Daten darzustellen, zu speichern und zu lesen. Gängige Datenaustauschformate wie JSON und XML haben sich etabliert, um strukturiert und lesbar Daten zu repräsentieren und auszutauschen. Da komplexe Software mitunter erhebliche Datenmengen generiert und verarbeitet, ist es erstrebenswert, Datenformate zu entwickeln, die kompakt sind und eine effiziente Verarbeitung ermöglichen.

Liegt ein Term in Curry in Normalform vor, d.h. ist er nicht weiter reduzierbar, dann lässt er sich durch eine Baumstruktur darstellen.

Betrachten wir als Beispiel die Typdefinition

```
data List a = Nil | Cons a (List a)
```

und die dreielementige Liste

```
myList :: List Int
myList = Cons 1 (Cons 2 (Cons 3 Nil))
```

Dieser Term lässt sich auf viele verschiedene Arten textuell darstellen. Mittels XML erhalten wir beispielsweise folgende Darstellung:

```
<Cons>
  <Int> 1 </Int>
  <Cons>
    <Int> 2 </Int>
    <Cons>
      <Int> 3 </Int>
      <Nil/>
    </Cons>
</Cons>
```

Mit dem Datenaustauschformat JSON lässt sich der Term wie folgt darstellen:

```
{
  "Cons": {
    "Int": 1,
    "Cons": {
      "Int": 2,
      "Cons": {
        "Int": 3,
        "Nil": {}
      }
    }
  }
}
```

Die Datenaustauschformate JSON und XML haben die Zielsetzung, hierarchische, menschlich lesbare und maschinenlesbare Daten zu erzeugen.

In Curry werden Terme üblicherweise mittels ihrer `Show`-Instanz als String dargestellt. Dabei werden die Konstruktoren und die Argumente durch Leerzeichen getrennt und die Argumente bei Bedarf geklammert. Es ergibt sich

```
show (Cons 1 (Cons 2 (Cons 3 Nil))) = "Cons 1 (Cons 2 (Cons 3 Nil))"
```

Alle diese Darstellungen haben gemeinsam, dass sie den Term für den Menschen lesbar und strukturiert darstellen. Daraus ergibt sich jedoch, dass unnötige Informationen enthalten sind, die für die rein maschinelle Verarbeitung nicht notwendig sind. Vor allem bei großen Datenmengen kostet die Verarbeitung dieser Daten unnötig Zeit. Möchte man sehr große Datenmengen zwischen Programmkomponenten austauschen, dann ist Lesbarkeit für Menschen nicht relevant. In diesem Zusammenhang wird in dieser Bachelorarbeit eine kompakte Datenrepräsentation entwickelt und vorgestellt. Diese Repräsentation wird verwendet, um Curry-Pakete wie `flatcurry`[1] zu optimieren und darauf aufbauende Programme und Werkzeuge zu beschleunigen. Um diese Darstellung einfach in Curry verwenden zu können, wird ein Werkzeug entwickelt, das automatisch Operationen zum Schreiben und Lesen von Daten in dieser kompakten Darstellung generiert.

1.2. Gliederung

Kapitel 2 beschreibt die Grundlagen von Curry und die für diese Arbeit relevanten Konzepte und Techniken. In Kapitel 4 wird die kompakte Repräsentation von Daten in Curry vorgestellt. Dabei werden unterschiedliche Konzepte vorgestellt, die nachfolgend implementiert und

[1] Informationen unter `https://www-ps.informatik.uni-kiel.de/~cpm/pkgs/flatcurry.html`

getestet werden. In Kapitel 5 wird die Implementierung der kompakten Darstellung sowie das entwickelte Werkzeug vorgestellt und in Kapitel 6 wird das Ergebnis ausgewertet und die Anwendbarkeit der kompakten Repräsentation anhand von Benchmarks evaluiert. Dabei wird insbesondere das Curry-Paket `flatcurry` mit der neuen Datenrepräsentation getestet und mit den Curry-Systemen PAKCS[2] und KiCS2[3] erprobt. Abschließend werden in Kapitel 7 die Ergebnisse zusammengefasst und ein Ausblick auf mögliche Erweiterungen und zukünftige Arbeiten gegeben.

[2] `https://www.informatik.uni-kiel.de/~pakcs/`

[3] `https://www-ps.informatik.uni-kiel.de/kics2/`

Kapitel 2

Grundlagen

In diesem Kapitel die für das Verständnis der Arbeit notwendigen Grundlagen vorgestellt. Zunächst wird ein Überblick über die Programmiersprache Curry gegeben. Insbesondere wird dabei auf das Typsystem eingegangen, welches für die Struktur von Ausdrücken sowie Datentermen und somit auch für die Arbeit essenziell ist. Anschließend wird Metaprogrammierung im Allgemeinen sowie die in Curry verfügbaren Techniken zur Metaprogrammierung vorgestellt. In diesem Zusammenhang wird auch das Curry-Paket `FlatCurry` vorgestellt. Anschließend wird ein Überblick über die Curry-Systeme PAKCS und KiCS2 sowie den CPP gegeben.

2.1. Curry

Curry ist eine syntaktisch auf Haskell basierende, funktional-logische Programmiersprache. Im Folgenden werden relevante Aspekte kurz vorgestellt. Eine ausführliche Einführung in diese und weitere Themen findet sich im Curry-Tutorial[1].

2.1.1. Funktionale und logische Programmierung

In Curry wird zwischen zwei Arten von Berechnungen unterschieden. Deterministische Berechnungen sind solche, die für eine gegebene Eingabe immer genau ein Ergebnis als Wert liefern (oder fehlschlagen). Betrachten wir die folgende Funktion:

```
fromMaybe :: a -> Maybe a -> a
fromMaybe _ (Just x) = x
fromMaybe d Nothing  = d
```

Die Funktion `fromMaybe` nimmt einen Defaultwert vom Typ `a` und ein `Maybe a` und liefert den Defaultwert zurück, falls die Eingabe ein `Nothing` ist. Andernfalls wird der vom Typ `Maybe` verpackte Wert zurückgegeben. Die Funktion `fromMaybe` ist deterministisch, da sie für jede Eingabe genau ein Ergebnis liefert.

Nichtdeterministische Berechnungen sind solche, die für eine gegebene Eingabe mehrere Ergebnisse als Wert liefern können. Ein Beispiel für eine nichtdeterministische Funktion ist

[1] `https://www.informatik.uni-kiel.de/~curry/tutorial/`

`coin`:

```
coin :: Int
coin = 0
coin = 1
```

Wird der Ausdruck `coin` ausgewertet, so werden die beiden Ergebnisse `0` und `1` zurückgegeben, da sich die linken Regelseiten (trivial) überlappen. Bemerkenswert ist hierbei, dass keine Liste von Ergebnissen zurückgegeben wird. Stattdessen teilt sich die Berechnung in zwei Pfade auf, die jeweils ein Ergebnis liefern. Damit liefert der Ausdruck `coin + 2` die Ergebnisse `2` und `3`.

Da man mit unterschiedlichen Ergebnissen weiterarbeiten möchte, etwa um die Ergebnisse zu filtern oder zu vergleichen, ist es nötig, nichtdeterministische Berechnungen zu kapseln. Hierdurch lassen sich die unterschiedlichen Berechnungszweige zusammenfassen. Beispielweise lässt sich mit `allValues coin` eine Liste aller Ergebnisse von `coin` erzeugen. Da die Auswertungsreihenfolge aber nicht klar ist, könnte das Ergebnis entweder `[0,1]` oder `[1,0]` sein, was nicht im Sinne der deklarativen Programmierung ist. Diesem und anderen Problemen wird mit sogenannten *Set Functions* begegnet, die in [AH09] vorgestellt werden. Nichtdeterministische Operationen im Sinne der Berechnung mehrerer Ergebnisse werden in dieser Arbeit nicht weiter betrachtet, jedoch bietet diese Funktionalität auch im Zusammenhang mit deterministischen[2] Berechnungen Vorteile. Durch die eingebaute Unterscheidung von fehlschlagenden und nicht-fehlschlagenden Berechnungen ist es möglich, Fehlerbehandlung zu vereinfachen, indem die u.U. fehlschlagende Operation durch `oneValue :: a -> Maybe a` gekapselt wird. Liefert die Operation ein Ergebnis `x`, dann wird `Just x` zurückgegeben - ansonsten wird `Nothing` zurückgegeben. Beispielsweise lassen sich Berechnungen, die in der `Maybe`-Monade[3] laufen, dadurch deutlich handlicher aufschreiben. Im späteren Verlauf dieser Arbeit wird das Ausnutzen dieses Vorteils diskutiert.

2.1.2. Bedarfsgesteuerte Auswertung

Im Gegensatz zu vielen imperativen Programmiersprachen, die alle Ausdrücke grundsätzlich strikt auswerten, wird in Curry die Auswertung von Ausdrücken bedarfsgesteuert (auch faul oder *lazy*) durchgeführt. Das bedeutet, dass ein Ausdruck nur dann ausgewertet wird, wenn sein Wert tatsächlich benötigt wird. Diese Auswertungsstrategie ermöglicht es, effiziente Programme deutlich eleganter aufzuschreiben, als es in strikt auswertenden Sprachen möglich ist. Ein Beispiel hierfür ist die Funktion `all` aus der **`Prelude`**:

```
all :: (a -> Bool) -> [a] -> Bool
all p = and . map p

and :: [Bool] -> Bool
```

[2] Deterministisch in dem Sinne, dass die Berechnung nicht mehr als ein Ergebnis hat

[3] Mehr Informationen unter `https://www-ps.informatik.uni-kiel.de/~fhu/FP16/Monad.html`

```
and = foldr (&&) True

(&&) :: Bool -> Bool -> Bool
True  && x = x
False && _ = False

n = all even [1..]
```

Die Funktion n prüft, ob alle positiven Zahlen gerade sind. Die Auswertung von n terminiert bereits nach dem Überprüfen des ersten Elements, da `even 1 = False`. Strategien zur bedarfsgesteuerten Auswertung in Verknüpfung mit Nichtdeterminismus werden eingehend in [Han13] erläutert.

Möchte man die vollständige Auswertung einer Berechnung erzwingen, etwa um die Laufzeit zu messen, so bieten sich beispielsweise die Funktion `normalForm` an, die einen Ausdruck so lange auswertet, bis er keine Funktionsaufrufe mehr enthält. Weiterhin lässt sich die vollständige Auswertung eines Ausdrucks `e` durch die Berechnung von `e == e` erzwingen.

2.1.3. Datentypen

In Curry ist jeder Ausdruck getypt. Typen sind beispielsweise die Basistypen (`Int`, `Float`, `Char`, `Bool`) und Typen für Listen (`[a]`) und Tupel (`(a,b)`). Des Weiteren ist es in Curry möglich, algebraische Datentypen mittels sogenannter Typdeklarationen einzuführen:

```
data T τ1 ... τn = C1 e1,1 ... e1,m1
                 | ...
                 | Ck ek,1 ... ek,mk
```

Hierbei ist `T` der Name des Typs, τ_1 bis τ_n sind Typvariablen und die Typausdrücke $e_{i,j}$ sind die Argumente der Konstruktoren.

Für den einfachen Fall $n = 0$ spricht man von Monomorphen Datentypen. Ein Beispiel hierfür ist der Aufzählungstyp

```
data Bool = True | False
```

Im Falle von $n > 0$ Typvariablen spricht man von polymorphen Datentypen. Ein Beispiel hierfür ist der Typ `Pair`, der im Wesentlichen eine handgeschriebene Variante eines 2-Tupels ist:

```
data Pair a b = Pair a b
```

Polymorphe Typen zeichnen sich dadurch aus, dass sie bezüglich ihrer Typvariablen parametrisch sind. Das bedeutet, dass beliebige Typen für die Typvariablen `a` und `b` substituiert

werden können. Beispielsweise sind die folgenden Ausdrücke gültig:

```
e1 = Pair 1    2   :: Pair Int Int
e2 = Pair True 'a' :: Pair Bool Char
```

Weiterhin ist es erlaubt, Typkonstruktoren als sogenannte *Records* aufzuschreiben. Dabei handelt es sich um syntaktischen Zucker, um Konstruktorparameter mit Namen zu versehen. Betrachten wir eine Typdeklaration für eine Person, die einen Namen, ein Alter, eine Größe und ein Gewicht hat:

```
data Person = Person String Int Int Int
```

Das Problem liegt auf der Hand: Ohne weitere Informationen können wir die Konstruktorargumente gar nicht unterscheiden. Dies sorgt für eine schlechte Lesbarkeit und Wartbarkeit des Codes. Zudem müssen wir viele Hilfsfunktionen (*selectors*) schreiben, um auf die einzelnen Argumente zugreifen und den Typen sinnvoll nutzen zu können:

```
name :: Person -> String
name (Person n _ _ _) = n

age :: Person -> Int
age (Person _ a _ _) = a

...
```

Mit *Records* lässt sich die Typdeklaration deutlich eleganter aufschreiben. Darüber hinaus werden implizit vom Compiler die Hilfsfunktionen `name`, `age`, `height` und `weight` generiert und können direkt genutzt werden.

```
data Person = Person { name   :: String
                     , age    :: Int
                     , height :: Int
                     , weight :: Int }

isTaller :: Person -> Person -> Bool
isTaller p1 p2 = height p1 > height p2
```

Typen können auch rekursiv definiert werden. Ein Beispiel hierfür ist die Definition des Typs `List`:

```
data List a = Nil | Cons a (List a)
```

Grundlegende Informationen zu Typen und streng getypten Programmiersprachen finden sich in [CW85]. Weitere technische Details zu Datentypen und *Records* in Curry werden

ausführlich in [Han16] beschrieben.

2.1.4. Polymorphismus

Mittels *parametrischem Polymorphismus* ist es möglich, Funktionen zu schreiben, die für beliebige zulässige Typen (gleich) funktionieren. Ein Beispiel hierfür ist `head`:

```
-- Returns the first element of a list
head :: [a] -> a
head (x:_) = x
```

Die Funktion `head` berechnet unabhängig vom konkreten Typ, der `a` ersetzt, das erste Element einer Liste. Damit ist bspw. `head [1..] = 1` und `head "ABC" = 'A'`. Somit verhält sich eine parametrisch polymorphe Funktion für alle konkreten zulässigen Typen gleich.

Sogenannter *ad-hoc Polymorphismus* ermöglicht es, für verschiedene Typen unterschiedliche Implementierungen einer Funktion zu schreiben. Betrachten wir zuerst eine Funktion, die den kleinsten `Int` aus einer Liste berechnet:

```
minimum :: [Int] -> Int
minimum xs = foldr1 min xs
 where
  min x y | x <= y    = x
          | otherwise = y
```

Möchten wir die Funktion nun derart erweitern, dass sie grundsätzlich für alle Typen funktioniert, müssen wir für jeden Typen eine Funktion `min` zur Verfügung stellen:

```
minimum :: (a -> a -> a) -> [a] -> a
minimum min xs = foldr1 min xs
```

Der Nachteil hier ist, dass wir für jeden Typen, den wir unterstützen wollen, eine Funktion `min` in den Aufruf übergeben müssen. Dieses Problem wird durch sogenannte *Typklassen* gelöst, wodurch es ermöglicht wird, für verschiedene Typen unterschiedliche Implementierungen einer Funktion zu schreiben. Durch die prägnante Schreibweise wird der Code lesbarer und wartbarer. Ein Beispiel hierfür ist die Typklasse `Ord`[4], die hier stark verkürzt dargestellt ist:

```
class Ord a where
  ...
  min :: a -> a -> a

instance Ord Int where
  min x y | x <= y    = x
```

[4] https://www-ps.informatik.uni-kiel.de/~cpm/DOC/base-3.2.0/Prelude_curry.html#Ord

```
        | otherwise = y

instance Ord Bool where
  min = (&&)

minimum :: Ord a => [a] -> a
minimum xs = foldr1 min xs
```

Details zur Nutzung und Implementierung von Typklassen finden sich in [Tee16].

2.2. Metaprogrammierung

Als Metaprogrammierung bezeichnet man eine Programmiertechnik, Programme zur Laufzeit in Datenform zu laden, darzustellen, zu verändern und zu generieren. Darunter fällt zum Beispiel das Parsing von Quelltexten, bei dem eine lesbare Repräsentation eines Programms in eine Datenstruktur umgewandelt wird, die das Programm repräsentiert. Auch Compiler oder andere Programme, die Quelltext generieren, sind Beispiele für Metaprogramme. In Curry stehen als Pakete für Metaprogrammierung unter anderem[5] `AbstractCurry` und `FlatCurry` zur Verfügung.

2.2.1. `AbstractCurry`

`AbstractCurry` ist ein Paket, das genutzt werden kann, um Curry-Programme bzw. -Module in Curry als Terme darzustellen, d.h. mittels `AbstractCurry` kann auf einem abstrakten Syntaxbaum (*AST*) eines Curry-Programms gearbeitet werden. Dabei wird insbesondere Wert darauf gelegt, die Struktur des dargestellten Programms so detailliert wie möglich zu erhalten.

Im Modul `AbstractCurry.Types`[6] ist die Struktur des ASTs und damit die Darstellung von Curry-Programmen definiert. Die Darstellung eines Curry-Moduls, `CurryProg`, besteht dabei aus allen notwendigen Informationen, um das Modul zu repräsentieren. Dazu gehören unter anderem die Modulbezeichnung, die importierten Module, die Klassen-, Instanz- und Typdeklarationen sowie die Regeln. Jeder auftretende Bezeichner ist qualifiziert und damit eindeutig einem Modul zugeordnet. Funktionen haben eine Sichtbarkeit.

Die handliche Darstellung von Curry-Programmen in Form von Termen sowie nützliche Funktionalitäten zur Generierung und Analyse von Curry-Programmen machen `AbstractCurry` zu einem nützlichen Werkzeug für die Metaprogrammierung in Curry. Da das Paket einen *Pretty-Printer* enthält, lassen sich leicht textuelle Darstellungen der angefertigten Curry-Programme

[5] Im *CPM Repository* sind einige weitere Sprachrepräsentationen gelistet; beispielsweise für Prolog, Go und Javascript

[6] `https://git.ps.informatik.uni-kiel.de/curry-packages/abstract-curry/-/blob/master/src/AbstractCurry/Types.curry`

generieren.

Das Modul

```
module MyModule where

-- Returns the value wrapped in a Maybe
myFromJust :: Maybe a -> a
myFromJust (Just x) = x
myFromJust Nothing = error "myFromJust: Nothing"

-- Some constant
myNumber :: Int
myNumber = 4815162342

data MyData = MyCons Int String
  deriving (Show)
```

lässt sich beispielsweise in folgender Weise darstellen:

```
1  {- AbstractCurry 3.0 -}
2  CurryProg "MyModule" [] Nothing [] []
3   [CType ("MyModule","MyData") Public []
4     [CCons ("MyModule","MyCons") Public
5       [CTCons ("Prelude","Int"),CTCons ("Prelude","String")]] [("Prelude","Show")]]
6   [CFunc ("MyModule","myFromJust") 1 Public
7     (CQualType (CContext []) -- Maybe a -> a
8       (CFuncType (CTApply (CTCons ("Prelude","Maybe")) (CTVar (0,"a"))) (CTVar (0,"a"))))
9     [CRule [CPComb ("Prelude","Just") [CPVar (0,"x")]]
10       (CSimpleRhs (CVar (0,"x")) []),
11      CRule [CPComb ("Prelude","Nothing") []]
12        (CSimpleRhs (CApply (CSymbol ("Prelude","error"))
13          (CLit (CStringc "myFromJust: Nothing"))) [])],
14    CFunc ("MyModule","myNumber") 0 Public
15      (CQualType (CContext []) (CTCons ("Prelude","Int")))
16      [CRule [] (CSimpleRhs (CLit (CIntc 4815162342)) [])]]
17   []
```

Hieran lässt sich gut die Baumstruktur der abstrakten Syntax sowie die Eindeutigkeit aller Bezeichner erkennen. Zur automatisierten Generierung solcher `AbstractCurry`-Programme stehen einige Hilfestellungen zur Verfügung. Beispielsweise lässt sich der Typausdruck `Maybe a -> a`, dargestellt durch den recht komplizierten Term in Zeile 7, durch den folgenden Ausdruck aufbauen:

```
applyTC (pre "Maybe") [CTVar (0, "a")] ~> (CTVar (0, "a"))
```

Diese und weitere **Build**-Operationen erleichtern die Codegenerierung erheblich.

2.2.2. FlatCurry

`FlatCurry` ist eine Zwischensprache (*Intermediate Representation*, IR) von Curry, welche vom Curry-Frontend aus Curry-Quelltext generiert wird. Das Ziel dieser Zwischendarstellung ist es, Curry-Code möglichst kompakt und vereinfacht darzustellen, wobei *syntaktischer Zucker* entfernt wird. Funktionen, die in Curry mehrere Regeln haben, werden durch `FlatCurry` in eine einzige Regel umgewandelt, wobei *Pattern Matching* auf der linken Seite der Regeln durch *Case*-Ausdrücke ersetzt wird.

Da beispielsweise das Analysetool *call-analysis*[7] sowie Curry-Systeme wie *PAKCS* und *KiCS2* mit `FlatCurry`-Darstellungen arbeiten und diese demnach häufig benötigt werden, ist es erstrebenswert, das Lesen und Schreiben von `FlatCurry`-Dateien zu beschleunigen.

2.3. Curry-Systeme

Zwei der wichtigsten Curry-Systeme sind *PAKCS* und *KiCS2*. Die Systeme unterscheiden sich in ihrer Implementierung, weshalb sie näher betrachtet werden müssen. PAKCS kompiliert Curry-Programme nach Prolog, während KiCS2 Curry-Programme nach Haskell kompiliert. Curry unterstützt das Schlüsselwort `external` bei Funktionsdeklarationen. Dieses Schlüsselwort wird verwendet, um anzumerken, dass die jeweilige Funktion extern implementiert ist, wobei die Implementierung sich compilerabhängig unterscheiden kann. Betrachten wir das Beispiel **ReadShowTerm**[8]:

```
readUnqualifiedTerm :: [String] -> String -> _
readUnqualifiedTerm = ...

prim_readsUnqualifiedTerm :: [String] -> String -> [(_,String)]
prim_readsUnqualifiedTerm external
```

Mittels `external` können in der Zielsprache des Compilers von Hand geschriebene, optimierte Operationen direkt in Curry verwendet werden. Beispielsweise ist die Operation `prim_readsUnqualifiedTerm` ein PAKCS-seitig direkt in Prolog geschriebener Parser für Datenterme. Die beiden Operationen `showTerm` und `readUnqualifiedTerm` werden von Curry-Systemen und -Tools genutzt, um `FlatCurry`-Programme in Dateien zu schreiben bzw. aus Dateien zu lesen. Wegen der durch externe Implementierungen ermöglichten Performanceverbesserungen beschleunigen sie die Ausgabe und das Einlesen von Datentermen gegenüber

[7] https://cpm.curry-lang.org/pkgs/call-analysis.html

[8] https://www-ps.informatik.uni-kiel.de/~cpm/DOC/html-cgi-0.0.1/ReadShowTerm_curry.html

der von der **Prelude** zur Verfügung gestellten Funktionalitäten enorm, verwenden aber nach wie vor eine recht große Darstellung.

Curry-Systeme unterstützen durch ihren Präprozessor die bedingte Kompilierung von Quelltexten. Hierbei wird der Quelltext in Abhängigkeit von Compiler-Flags noch vor der Kompilierung verändert. Die Syntax orientiert sich an der des C-Präprozessors. Wie folgt können wir noch vor der Kompilierung des Quelltextes den verwendeten Compiler ermitteln und als String in den Quelltext einfügen:

```
#ifdef __KICS2__
compiler = "KiCS2"
#elif defined(__PAKCS__)
compiler = "PAKCS"
#else
compiler = "Unknown"
#endif

main = putStrLn ("This program was compiled with the compiler " ++ compiler)
```

Das Resultat der Ausführung dieses Programms hängt nun ausschließlich davon ab, welcher Compiler verwendet wurde. Die Identifikation des verwendeten Compilers kann bei der Entwicklung von Programmen hilfreich sein, weil Code genau auf die jeweiligen Compilersysteme zugeschnitten werden kann, wodurch die Vorteile unterschiedlicher Compiler ausgenutzt werden können.

Kapitel 3

Verwandte Arbeiten

Für Haskell ist das Paket `Data.Binary`[1] verfügbar, das die Serialisierung von Daten in Binärform ermöglicht. Um mit Binärdarstellungen arbeiten sowie diese effizient lesen und schreiben zu können, werden spezielle Sprachkonstrukte wie *lazy bytestrings*[2] verwendet. Da für Curry bislang keine derartigen Konstrukte verfügbar sind und auch das systemnahe und effiziente Lesen und Schreiben von Binärdateien nicht einfach möglich ist[3], wird in dieser Arbeit ein textuelles Format zum Darstellen von Daten entwickelt. Durch diese abstrahierte Darstellung von Daten ist auch eine gute Portabilität gewährleistet, insbesondere muss sich nicht mit technischen Problemen (bspw. *Endianness* oder systemabhängige Operationen) auseinandergesetzt werden. Grundsätzlich soll das in der Arbeit entwickelte Werkzeug allerdings mit geringem Aufwand modifiziert oder erweitert werden können, um in Zukunft die Darstellungen in Bezug auf neue Entwicklungen anzupassen.

[1] `https://hackage.haskell.org/package/binary-0.8.9.1/docs/Data-Binary.html`

[2] `https://hackage.haskell.org/package/bytestring-0.12.1.0/docs/Data-ByteString-Lazy.html`

[3] Das Modul `System.IO.BinaryFile` implementiert zwar eine Schnittstelle zum Lesen und Schreiben von Binärdateien, jedoch werden die einzelnen Bytes durch Integer kodiert, was zu einer ineffizienten Darstellung führt.

Kapitel 4

Entwurf der kompakten Darstellung

In diesem Kapitel wird der Entwurf der kompakten Darstellung von Datentermen in Curry beschrieben. Des Weiteren werden die Anforderungen an die Darstellung erläutert.

4.1. Genereller Ansatz

Betrachten wir zunächst ein Beispiel für die Darstellung von Listen in Curry.

```
data List a = Nil
            | Cons a (List a)
  deriving (Read, Show)

data Bool = True | False
  deriving (Read, Show)
```

Mittels des algebraischen Datentyps `List` lassen sich Listen von Werten eines beliebigen Typs `a` darstellen. Beispielsweise lässt sich die Liste `[True, False, True]` mithilfe dieses Datentyps durch den folgenden Term repräsentieren:

```
list = Cons True (Cons False (Cons True Nil))
```

Da für den Datentyp `List` eine `Show`-Instanz definiert ist, lässt sich der Wert von `list` als String in eben dieser Repräsentation darstellen:

```
> show term
"Cons True (Cons False (Cons True Nil))"
```

Wir können sehen, dass die Ausgabe groß ist. Parser müssen viel Aufwand betreiben, um die Klammerstrukturen sowie die Leerzeichen zu parsen und die Konstruktoren zu lesen. Eine grundlegende Eigenschaft unserer geplanten kompakten Darstellung ist die Abstraktion über die Struktur des Terms sowie die Namen der Konstruktoren. Die strukturierte Darstellung wird somit in eine *sequenzielle* Darstellung überführt.

Jedem Konstruktor wird ein (pro Datentyp) eindeutiges Zeichen aus einem Alphabet Σ zugeordnet. Dabei wird dem n-ten Konstruktor das n-te Zeichen aus Σ zugeordnet. Die

Argumente des Konstruktors werden in der Reihenfolge ihres Auftretens in der Konstruktordeklaration sequenziell dargestellt. Es ergibt sich für die Liste `[True, False, True]` die folgende Darstellung:

```
Cons True (Cons False (Cons True Nil))
1    0     1    1      1    0    0
```

Die textuelle Darstellung des Terms ist somit `"1011100"`. Diese Darstellung ist üblicherweise 50-75% kompakter und vor allem schneller lesbar als die ursprüngliche Darstellung.

Da der Parser stets weiß, wie viele Argumente ein Konstruktor hat, ist es nicht notwendig, die Argumente zu trennen. Die Argumente werden einfach sequenziell hintereinander geschrieben. Weiterhin ist für jedes Argument bekannt, welchen Typen es hat, weshalb die Konstruktoren lediglich pro Datentyp eindeutig identifiziert werden müssen. Hierdurch können im oberen Beispiel die beiden Konstruktoren `Cons` und `False` durch das Zeichen `'1'` ersetzt werden, ohne dass eine Ambiguität entsteht[1].

4.1.1. Allgemeine Darstellung von algebraischen Datentypen

Sei folgende Typdeklaration gegeben:

$$
\begin{aligned}
&\texttt{data T } \tau_1 \ \ldots \ \tau_n \texttt{ = } C_1 \ t_{1,1} \ \ldots \ t_{1,m_1} \\
&\qquad\qquad\qquad\quad \texttt{| } \ldots \\
&\qquad\qquad\qquad\quad \texttt{| } C_k \ t_{k,1} \ \ldots \ t_{k,m_k}
\end{aligned}
$$

Dabei ist T der Name des Datentyps, $\tau_1, ..., \tau_n$ die Typvariablen, $C_1, ..., C_k$ die Konstruktoren und $t_{i,j}$ Typausdrücke. Dann ist der schematische Aufbau einer Funktion `showRW` für den Datentyp T wie folgt:

$$
\begin{aligned}
&\texttt{showRW}_T \texttt{ :: } (\tau_1 \texttt{ -> String}) \texttt{ -> } \ldots \texttt{ -> } (\tau_n \texttt{ -> String}) \texttt{ -> T } \tau_1 \ \ldots \ \tau_n \texttt{ -> String} \\
&\texttt{showRW}_T \ \texttt{showRW}_{\tau_1} \ \ldots \ \texttt{showRW}_{\tau_n} \ (C_1 \ e_{1,1} \ \ldots \ e_{1,m_1}) \\
&\quad \texttt{= } \sigma_1 \texttt{ : showRW}_{\delta(e_{1,1})} \ e_{1,1} \texttt{ ++ } \ldots \texttt{ ++ showRW}_{\delta(e_{1,m_1})} \ e_{1,m_1} \\
&\ldots \\
&\texttt{showRW}_T \ \texttt{showRW}_{\tau_1} \ \ldots \ \texttt{showRW}_{\tau_n} \ (C_k \ e_{k,1} \ \ldots \ e_{k,m_k}) \\
&\quad \texttt{= } \sigma_k \texttt{ : showRW}_{\delta(e_{k,1})} \ e_{k,1} \texttt{ ++ } \ldots \texttt{ ++ showRW}_{\delta(e_{k,m_k})} \ e_{k,m_k}
\end{aligned}
$$

Dabei ist $\sigma_i \in \Sigma$ für $1 \leqslant i \leqslant k$ das Zeichen, das dem Konstruktor C_i zugeordnet ist. Weiter ist δ eine Funktion, die jedem Ausdruck textuell ihren Typen zuordnet. Die Funktionen $\texttt{showRW}_{\delta(e_x)}$

[1] Durch die fehlenden Konstruktorbezeichner kann es vorkommen, dass Daten bei der Wahl des falschen Parsers falsch interpretiert werden. Diesem Problem wird in Kapitel 4.2 begegnet.

sind derart definiert, dass sie die Argumente e_x in die kompakte Darstellung[2] überführen. Für jede polymorphe Typvariable τ_i muss weiterhin eine Funktion `showRW`$_{\tau_i}$ angegeben werden. Später werden wir sehen, wie wir Typklassen und Ad-hoc-Polymorphismus nutzen können, um solche Funktionsdefinitionen zu vereinfachen und eleganter aufzuschreiben.

Diese schematische Definition der Funktion `showRW` ist für jeden algebraischen Datentypen anwendbar, falls die Anzahl der Konstruktoren beschränkt ist durch die Mächtigkeit des Alphabets Σ. Verfügt man Beispielsweise nur über das Alphabet $\Sigma =$`"0123456789"` und möchte einen Datentypen mit mehr als zehn Konstruktoren kodieren, so ist die Darstellung mit nur einem Zeichen pro Konstruktor nicht möglich. In diesem Fall müssen mehr Zeichen verwendet werden, um die Konstruktoren voneinander unterscheiden zu können. Dies stellt aber grundsätzlich kein Problem dar, da sich die schematische Definition der Funktion `showRW` hierdurch nur geringfügig ändert. Darüber hinaus ist es praktisch unwahrscheinlich, dass die Größe des Alphabets der Anzahl der Konstruktoren eines algebraischen Datentyps unterliegt, sofern ein hinreichend großes Alphabet gewählt wird. Da die Darstellung textuell ist und somit beispielsweise auf ASCII basiert, stehen viele darstellbare Zeichen zur Verfügung.

Weiterhin tritt bei einer Typdeklaration häufig der Fall $k = 1$ auf, d.h. der Typ weist genau einen Konstruktor auf. In diesem Fall ist es nicht notwendig, den Konstruktor explizit darzustellen, da die Wahl des Konstruktors eindeutig ist. Beispielsweise ist dann für

```
data T a = C a
```

die Funktion `showRW` wie folgt definiert:

```
showRW_T :: (a -> String) -> T a -> String
showRW_T showRW_a (C e) = showRW_a e
```

Aus diesen Regeln ergibt sich dann auch, dass im einfachsten Fall $k = 1$ und $m_1 = 0$ (genau ein Konstruktor mit genau null Argumenten) gar nichts dargestellt werden muss. Für den *Einheitstypen* (`data () = ()`) ist die Funktion `showRW`$_{()}$ daher wie folgt definiert:

```
showRW_() :: () -> String
showRW_() () = ""
```

4.1.2. Darstellung von Zahlen

Betrachten wir zunächst ein Beispiel für die Darstellung von Termen in Curry.

```
data Expr = Const Int
          | Add Expr Expr
```

[2] Zur Veranschaulichung ist hier der Rückgabetyp von `showRW`$_T$ ein `String`. Es gibt bessere Ansätze wie `ShowS` aus dem Modul `Text.Show`, wie wir später sehen werden.

```
        | Multiply Expr Expr
  deriving (Read, Show, Eq)
```

Der algebraische Datentyp `Expr` stellt arithmetische Ausdrücke dar, bestehend aus Ganzzahlen als Operanden sowie den Operationen Addition und Multiplikation. Für den Ausdruck `3 * (1 + 5)` ist dann

```
term = Multiply (Const 30) (Add (Const 1) (Const 5))
```

Würden wir die Zahlen in der kompakten Darstellung direkt darstellen, so ergäbe sich für den Term `term` die folgende Darstellung:

```
> showRW_Expr term
"203010105"
   ^^  ^ ^
```

Da ohne beliebig weites Vorausschauen nicht ersichtlich ist, ob die Zeichen `"30"` für die Zahl `30` stehen oder für die Zahl `3` sowie einen darauf folgenden Konstruktor, ist es notwendig, die Zahlen eindeutig darzustellen. Dafür stehen grundsätzlich zwei Möglichkeiten zur Verfügung:

1. **Darstellung mit bekannter Länge:** Die Zahlen werden in einer festen Länge dargestellt. Beispielsweise bieten sich mehrere Zeichen (Bytes) an, um die Zahlen darzustellen. Ein Problem bei diesem Ansatz liegt darin, dass die interne Darstellung von Zahlen in Curry nicht an eine feste Länge gebunden ist. Grundsätzlich können beispielsweise `Int`-Zahlen beliebig groß sein. Darüber hinaus muss mit Binärarithmetik gearbeitet werden, um die Zahlen in die kompakte Darstellung zu überführen und aus dieser zu lesen.

2. **Darstellung mit Trennzeichen:** Die Zahlen werden durch Trennzeichen voneinander getrennt. Somit kann die Ganzzahl `30` als `"30;"` dargestellt werden. Hierbei müssen keine Annahmen über die maximale Größe der Zahlen getroffen werden, allerdings muss die Zahl ebenfalls mittels arithmetischer Operationen gelesen werden (Einzelne Ziffern müssen von der textuellen Darstellung in eine Ganzzahldarstellung überführt werden, mit der dann weitergerechnet wird).

Da die erste Möglichkeit aufgrund der Komplexität und der Unflexibilität nicht praktikabel ist, wird die zweite Möglichkeit gewählt. Die Darstellung von Zahlen in der kompakten Darstellung erfolgt durch Trennzeichen. In praktischen Tests hat sich gezeigt, dass die Darstellung mit Trennzeichen nicht nur einfacher zu implementieren ist, sondern üblicherweise auch schneller zu parsen ist als das Arbeiten mit arithmetischen Operationen auf Bit- und Byteebene.

4.1.3. Darstellung von Listen und Strings

Die Darstellung von einzelnen Symbolen (Chars) ist trivial. Jedes Symbol wird durch sich selbst dargestellt[3]. Die oben genannte Darstellung von Listen lässt sich exakt auf die eingebaute Liste in Curry (`data [a] = [] | a : [a]`) übertragen. Wegen `type String = [Char]` ergibt sich sofort die Darstellung von Strings. Somit lässt sich der String `"Hello, World!"` als `"0H0e0l0l0o0,0 0w0o0r0l0d1"` darstellen.

Eine alternative Darstellung sieht vor, dass die Länge der Liste vorangestellt wird. Dies hat den Vorteil, dass die Länge der Liste sofort ersichtlich ist und die Konstruktoren der Liste nicht explizit dargestellt werden müssen. Hierdurch ließe sich der String `"Hello, World!"` als `"13;Hello, World!"` darstellen. Praktisch zeigt sich insbesondere bei kleineren Listen, dass der zusätzliche Overhead beim Lesen von Zahlen das Lesen der Liste insgesamt verlangsamen kann.

Typische Daten, die in Programmen verarbeitet sowie von Programmen geschrieben werden, enthalten häufig viele Strings. Relevante Beispiele hierfür sind im Zusammenhand mit Curry beispielsweise `FlatCurry` oder `AbstractCurry`. Gerade bei solchen Datenmengen ist es wichtig, dass die Darstellung von Strings effizient ist. Betrachten wir abermals die Darstellung eines Programms in `AbstractCurry`:

```
{- AbstractCurry 3.0 -}
CurryProg "MyModule" [] Nothing [] []
 [CType ("MyModule","MyData") Public []
   [CCons ("MyModule","MyCons") Public
     [CTCons ("Prelude","Int"),CTCons ("Prelude","String")]] [("Prelude","Show")]]
 [CFunc ("MyModule","myFromJust") 1 Public
   (CQualType (CContext [])
     (CFuncType (CTApply (CTCons ("Prelude","Maybe")) (CTVar (0,"a"))) (CTVar (0,"a"))))
   [CRule [CPComb ("Prelude","Just") [CPVar (0,"x")]]
      (CSimpleRhs (CVar (0,"x")) []),
    CRule [CPComb ("Prelude","Nothing") []]
      (CSimpleRhs (CApply (CSymbol ("Prelude","error"))
        (CLit (CStringc "myFromJust: Nothing"))) [])],
  CFunc ("MyModule","myNumber") 0 Public
    (CQualType (CContext []) (CTCons ("Prelude","Int")))
    [CRule [] (CSimpleRhs (CLit (CIntc 4815162342)) [])]]
 []
```

Bereits in diesem kleinen Beispiel taucht der String `"Prelude"` achtmal auf, der String `"MyModule"` taucht fünfmal auf. Da das Lesen von Strings viel Zeit in Anspruch nehmen kann und das mehrfache Darstellen dieser Strings eine unnötig große Ausgabe erzeugt, bietet

[3] Je nach Format der Ausgabe ist ggf. eine Ersatzdarstellung (*escaping*) von Sonderzeichen notwendig

es sich an, diese nur ein einziges Mal darzustellen und dann mittels IDs in der kompakten Darstellung zu repräsentieren. In der Ausgabe lassen sich die Strings dann beispielsweise Zeilenweise als Liste darstellen, die kompakte Repräsentation enthält dann nur noch Referenzen auf diese Strings.

4.1.4. String-Extraktion

Wegen der zeilenweisen Darstellung aller auftretenden Strings bietet es sich offensichtlich an, diese anhand ihrer Indexe zu adressieren, wobei jeder String durch einen Integer adressiert wird. Betrachten wir dafür das Beispiel der Liste

```
["Hello, World!", "map", "Prelude", "myFromJust", "foldr",
 "myFromJust: Nothing", "Prelude", "Prelude"]
```

Die Darstellung dieser Liste in der kompakten Darstellung ist dann

```
1  0;1;2;3;4;5;2;2;0
2  Hello, World!
3  map
4  Prelude
5  myFromJust
6  foldr
7  myFromJust: Nothing
```

Dabei ergeben die ausgegrauten Bestandteile der Kodierung (Zeile 1) die Kodierung der Liste selbst, während die blau gefärbten Integerkodierungen die Adressierung der Strings darstellt, welche in der Liste enthalten sind. Man beachte, dass der String `"Prelude"` dreimal adressiert wird und lediglich ein einziges Mal direkt repräsentiert wird. Bei diesem Ansatz fallen zwei Probleme auf:

1. Es müssen Integer geschrieben und gelesen werden. Das Lesen von Integern ist üblicherweise langsam.

2. Falls die interne Darstellung der Strings beim Lesen eine Liste von Strings ist, so muss bei jeder Stringadressierung ein Index dieser Liste nachgeschlagen werden. Da Listen in Curry wie einfach verkettete Listen implementiert sind, ist das Nachschlagen des n-ten Elements in einer Liste in $\mathcal{O}(n)$. Insbesondere bei großen Listen kann dies zu einer erheblichen Laufzeitverzögerung führen.

Wie wir in der Implementierungsphase sehen werden, existieren elegantere Lösungen dafür, die String-IDs darzustellen und Strings in einer effizienten Datenstruktur zu speichern sowie nachzuschlagen. Da die konkrete Implementierung des String-*Lookups* im Wesentlichen von der kompakten Repräsentation losgelöst ist und Strings geordnet und zeilenweise angeordnet dargestellt werden, bleibt vorerst offen, wie Strings tatsächlich extrahiert und dargestellt

werden. Unterschiedliche Möglichkeiten werden in Kapitel 5 vorgestellt und evaluiert sowie in Kapitel 6 anhand von konkreten Tests (*Benchmarks*) erprobt.

Strings, welche sehr kurz sind oder nur sehr selten vorkommen, können in der kompakten Darstellung auch direkt dargestellt werden. Ist ein String selbst beispielsweise nicht länger als der Verweis auf ihn, der bei einer Extraktion notwendig wäre, so kann es sinnvoll sein, den String direkt darzustellen.

4.1.5. Escaping

Da bisher jeder String in der kompakten Darstellung durch ein Trennzeichen (*Newline*, `'\n'`) abgeschlossen wird, ist es notwendig, dieses Trennzeichen anders darzustellen, falls es im String selbst vorkommt. Wir unterscheiden zwei Fälle beim Schreiben eines Strings:

1. Der String enthält ein *Newline*: Schreibe die Länge des Strings, gefolgt von einem `';'`, gefolgt vom String selbst.

2. Ansonsten: Schreibe ein `';'`, gefolgt vom String selbst. Schreibe anschließend ein `'\n'`.

Für die Liste `["New\nline", "map", "Hello, \tWorld!", "Another \tnew\nline!"]` ergibt sich dann die folgende Darstellung[4]:

```
10;11;12;13;0
8;New
line;map
;Hello, \tWorld!
18;Another \tnew
line!
```

Beim Lesen der extrahierten Strings muss nun wie folgt vorgegangen werden:

1. Beginnt die Zeile mit einer Zahl, so wird die Zahl n mit zugehörigem `';'` gelesen. Anschließend ergeben die nächsten n Zeichen den String.

2. Ansonsten wird die Zeile bis zum ersten `'\n'` gelesen.

Diese Vorgehensweise sorgt dafür, dass Strings, die keine *Newlines* enthalten, in der kompakten Darstellung nur minimal länger sind als in der ursprünglichen Darstellung. Da die meisten Strings keine *Newlines* enthalten, wird viel Zeit dadurch gespart, dass kein *Unescaping* notwendig ist. Insbesondere muss nicht bei jedem Zeichen geprüft werden, ob es Teil einer *Escaping*-Sequenz ist. Gegenüber der üblichen Vorgehensweise, jedes `'\n'` in `"\\n"` zu überführen (und umgekehrt), kann somit viel Rechenzeit gespart werden.

[4] Tabulatoren werden in der Ausgabe durch `\t` dargestellt und sind ein Zeichen lang.

4.1.6. Funktionen

Da das Typsystem von Curry *Funktionen höherer Ordnung* unterstützt, dürfen Funktionstypen grundsätzlich überall dort auftreten, wo auch andere Typen auftreten dürfen (Funktionen als Argumente und Ergebnisse von Funktionen, in Typdeklarationen, etc.). Beispielsweise sind also Typdeklarationen wie die folgenden gültig:

```
data T a = C a (a -> a)
```

Da Funktionen aber nicht Daten im herkömmlichen Sinne sind und mit üblichen Programmiertechniken auch kein *Pattern-Matching* auf ihnen möglich ist, gibt es keine praktisch sinnvolle Möglichkeit, sie mit den vorgestellten Mitteln textuell darzustellen. Somit ist es nötig, Funktionen in der kompakten Darstellung zu ignorieren. Das in dieser Arbeit vorgestellte Werkzeug wird bei der Codegenerierung eine Warnung ausgeben. Es wird eine `ReadWrite`-Instanz für `T` generiert, allerdings wird eine Instanz für `Int -> Int -> Int` fehlen, weshalb manuelles Eingreifen des Anwenders notwendig wird. Alternativ bietet es sich an, nur Operationen für Konstruktoren zu generieren, die keine Funktionstypen als Argumente besitzen. Praktisch handelt es sich hierbei kaum um eine Einschränkung, da man Funktionen in der Regel nicht textuell darstellen und speichern möchte.

4.2. Typidentifikation

Betrachten wir die Definition

```
myExpr = [Just True, Nothing, Just False]
```

Das Schreiben und anschließende Laden von `myExpr` mittels `Prelude.Read` und `Prelude.Show` lässt sich wie folgt realisieren:

```
1 main :: IO ()
2 main = do
3   writeFile "myExpr.txt" (show myExpr)
4   -- ...
5   fileContent <- readFile "myExpr.txt"
6   let myExpr' = read fileContent :: [Maybe Bool]
7   print (myExpr == myExpr')
8   let myExpr'' = read fileContent :: [(Bool, Bool, Bool)]
9   print myExpr''
```

Da die Funktion `read` die Typsignatur `Read a => String -> a` hat, ist es notwendig, den Zieltyp der geladenen Daten explizit anzugeben, denn der Compiler kann nicht raten, welchen Typ die geladenen Daten haben. In diesem Fall ist der Zieltyp `[Maybe Bool]`, weshalb der Aufruf in Zeile 3 mit dem Typ `[Maybe Bool]` annotiert wird. Entsprechend wird in Zeile 5

`True` ausgegeben, da `myExpr` und `myExpr'` gleich sind. In Zeile 7 schlägt die Berechnung fehl, weil `read` die geladenen Daten nicht in den Typ `[Maybe Int]` umwandeln kann. Dies ist ein gewünschtes Verhalten, da die geladenen Daten nicht in den Typ `[Maybe Int]` umgewandelt werden können, und der Tatsache geschuldet, dass die gelesenen Konstruktoren nicht mit den erwarteten Konstruktoren übereinstimmen.

Sei nun `readData :: String -> a` eine Operation, die Daten in kompakter Darstellung in den Typ `a` umwandelt. Wir betrachten ein analoges Beispiel:

```
main' :: IO()
main' = do
  writeFile "myExpr.txt" (showRW[Maybe Bool] myExpr)
  -- ...
  fileContent <- readFile "myExpr.txt"
  let myExpr' = readData fileContent :: [Maybe Bool]
  print (myExpr == myExpr')
  let myExpr'' = readData fileContent :: [(Bool, Bool, Bool)]
  print myExpr''
```

In diesem Beispiel schlägt die Ausführung der Zeile 7 nicht fehl! Betrachten wir dafür noch einmal die Definition von `myExpr`:

```
myExpr = [Just True, Nothing, Just False]
```

Die Repräsentation von `myExpr` in der kompakten Darstellung ist `"101111000"`. Wird dieser String mit `readData :: String -> [Maybe Bool]` gelesen, so wird der String in die Liste `[Just True, Nothing, Just False]` umgewandelt:

```
1   01            1   1        1   00          0
(:) (Just True)   ((:) Nothing ((:) (Just False) []))
```

Wird der String mit `readData :: String -> [(Bool, Bool, Bool)]` gelesen, so wird der String in die Liste `[(False, True, True), (True, False, False)]` umgewandelt:

```
1   011                 1   100                 0
(:) (False, True, True) ((:) (True, False, False) [])
```

Das Programmverhalten von der gewählten Typannotation abhängig zu machen, kann in Fällen wie diesem zu unerwünschten Ergebnissen führen. Lädt ein Programm beispielsweise irrtümlich Daten aus einer falschen Datei, so kann es vorkommen, dass das Programm weiterhin korrekt läuft, obwohl die geladenen Daten nicht den erwarteten Daten entsprechen. Durch das unerwartete Programmverhalten wird es mitunter schwierig, diese Fehler mittels *Debugging* zu finden und zu beheben, denn es kommt im schlechtesten Fall nicht zu Laufzeitfehlern. Eine Lösung für dieses Problem ist, die Daten beim Schreiben in Bezug auf

ihren Typen eindeutig zu identifizieren. Dies lässt sich beispielsweise dadurch lösen, dass beim Schreiben der Ausgabe zusätzlich der Typ der Daten geschrieben wird (bspw. in der ersten Zeile). Beim Lesen der Daten wird nun zuerst der Typ gelesen und geprüft, ob die Funktion `readData` mit dem korrekten Zieltypen aufgerufen wurde. Anschließend werden die Daten in den erwarteten Typ umgewandelt. Hierdurch ändert sich der Typ von `readData` zu `readData :: String -> Maybe a`, wobei `a` der Typ der Daten ist und `Nothing` zurückgegeben wird, falls die Eingabe nicht in den erwarteten Typ umgewandelt werden können. Für `writeData (1 :: Int)` ist das Ergebnis dann:

```
Int
1;
```

Für polymorphe Typen ist es zusätzlich nötig, die Parameter herauszuschreiben. Beispielsweise ist das Ergebnis von `writeData myExpr` dann:

```
[Maybe Bool]
101111000
```

Auf diese Weise wird sichergestellt, dass beim Lesen Daten nur Daten vom erwarteten Typ gelesen werden können. Eine bekannte Einschränkung dieser Methode ist, dass keine Informationen über die Struktur und Reihenfolgen der Konstruktoren gespeichert werden. Wird nach dem Schreiben der Daten die Struktur des Datentyps geändert, so kann es vorkommen, dass die geladenen Daten nicht mehr korrekt gelesen werden können.

Kapitel 5

Implementierung des Tools

In diesem Kapitel wird die Implementierung des Pakets `rw-data` vorgestellt, das die in Kapitel 4 vorgestellten Konzepte in Curry umsetzt. Darüber hinaus wird die Implementierung des enthaltenen Tools zur automatischen Generierung von Operationen zum Lesen und Schreiben von Daten in dieser Darstellung vorgestellt.

5.1. Die Typklasse `ReadWrite`

Jeder Typ, der in der kompakten Repräsentation dargestellt werden soll, muss eine Instanz der Typklasse `ReadWrite` besitzen. Diese Typklasse definiert die Operationen, die es ermöglichen, Werte des Typs aus der kompakten Repräsentation zu lesen und in diese zu schreiben. Die Klasse mit ihren grundlegenden Operationen definieren wir zunächst wie folgt:

```
class ReadWrite a where
  showRW :: a -> ShowS
  writeRW :: Handle -> a -> IO ()
  readRW :: String -> (a, String)
```

Die Operation `readRW` liest einen Wert des Typs `a` aus einer Zeichenkette, welche die kompakte Repräsentation des Wertes darstellt. Neben dem gelesenen Element gibt die Funktion den Rest der Eingabe zurück.

Sei folgende Typdeklaration gegeben:

```
data T τ1 ... τn = C1 t1,1 ... t1,m1
                 | ...
                 | Ck tk,1 ... tk,mk
```

Dabei ist T der Name des Datentyps, $\tau_1, \ldots, \tau_n$ die Typvariablen, $C_1, \ldots, C_k$ die Konstruktoren und $t_{i,j}$ Typausdrücke. Betrachten wir zunächst die Operation `showRW`. Die Strings $\sigma_1, \sigma_2, \ldots, \sigma_k \in \Sigma^*$ sind dabei wie folgt gewählt:

1. `length` σ_1 = `length` σ_2 = ... = `length` σ_k

2. alle σ_i sind paarweise verschieden

Hierdurch ergibt sich, dass jeder Konstruktor eindeutig kodiert und gelesen werden kann, auch wenn die Anzahl der Konstruktoren die Anzahl verfügbarer Zeichen im Alphabet Σ übersteigt. Für die Instanz der Typklasse `ReadWrite` für den Datentyp T ergibt sich folgende Implementierung:

```
instance (ReadWrite τ1, ..., ReadWrite τn) => ReadWrite (T τ1 ... τn) where
  ...
  showRW (C1 e1,1 ... e1,m1) = showString σ1 . showRW e1,1
                               . ... . showRW e1,m1
  ...
  showRW (Ck ek,1 ... ek,mk) = showString σk . showRW ek,1
                               . ... . showRW ek,mk
```

Die allgemeine Form der Implementierung der Operation `showRW` für einen Typ T orientiert sich damit stark am in Kapitel 4.1.1 vorgestellten Schema. Durch die Verwendung von *Ad-Hoc*-Polymorphismus können wir die Funktion `showRW` für jeden Typen aufrufen, der eine Instanz der Typklasse `ReadWrite` besitzt. Dies ermöglichen wir durch die Verwendung eines Kontextes, der die Typvariablen $\tau_1, ..., \tau_n$ entsprechend einschränkt (*class constraint*).

Der `(++)`-Operator für Stringkonkatenation hat eine lineare Laufzeitkomplexität in Bezug auf die Länge des linken Strings. Wenn viele Strings miteinander verkettet werden, kann dies zu einer ineffizienten Ausführung führen. Wir verwenden den Operator `(.)`, um die einzelnen Bestandteile zu kombinieren (Funktionskomposition). Die Funktion `showRW` für den Typ T ist also eine Verkettung von Funktionen, welche die einzelnen Bestandteile des Wertes in die kompakte Repräsentation umwandeln, was schon durch den Rückgabetyp von `showRW` ersichtlich wird (`type ShowS = String -> String`). Eine Funktion vom Typ `ShowS` ist eine Funktion, die eine Zeichenkette einer anderen Zeichenkette voranstellt. Durch die Funktionskomposition wird eine Auswertung von rechts nach links erzwungen, wodurch die rechte Seite der Konkatenation den String aufbaut und kein Overhead durch wachsende linke Seiten entsteht.

Möchten wir die kompakte Repräsentation eines Wertes des Typs T (direkt) in eine Datei schreiben, so bietet es sich an, sogenannte *Handle*-Operationen zu verwenden. Ein *Handle* ist ein Objekt, welches Operationen zum Lesen und Schreiben von Daten in einen Datenstrom (*stream*) - in diesem Fall eine Datei - bereitstellt. Die Operation `writeRW` schreibt einen Wert des Typs T in einen Datenstrom. Die Implementierung der Operation `writeRW` für den Typ T orientiert sich an der Implementierung der Operation `showRW`:

```
instance (ReadWrite τ1, ..., ReadWrite τn) => ReadWrite (T τ1 ... τn) where
...
writeRW h (C1 e1,1 ... e1,m1) = hPutStr h σ1 >> writeRW h e1,1 >> ... >> writeRW h e1,m1
```

```
...
writeRW h (C_k e_{k,1} ... e_{k,m_k}) = hPutStr h σ_k >> writeRW h e_{k,1} >> ... >> writeRW h e_{k,m_k}
```

Der Vorteil hierbei liegt darin, dass zur Laufzeit keine großen Funktionen entstehen, die den gesamten Wert in die kompakte Repräsentation umwandeln. Stattdessen wird die Darstellung während der Laufzeit in die Datei geschrieben, was zu einer geringeren Ausführungszeit führen kann.

Die Operation `readRW` liest einen Wert des Typs T aus einer Zeichenkette, welche die kompakte Repräsentation des Wertes darstellt. Die Implementierung der Operation `readRW` für den Typ T lautet wie folgt:

```
instance (ReadWrite τ_1, ..., ReadWrite τ_n) => ReadWrite (T τ_1 ... τ_n) where
  ...
  readRW s = (C_1 e_{1,1} ... e_{1,m_1}, r_{m_1})
    where (e_{1,1}, r_1) = readRW s_1
          ...
          (e_{1,m_1}, r_{m_1}) = readRW s_{m_1}
  ...
  readRW s = (C_k e_{k,1} ... e_{k,m_k}, r_{m_k})
    where (e_{k,1}, r_1) = readRW s_1
          ...
          (e_{k,m_k}, r_{m_k}) = readRW s_{m_k}
```

Ein konkretes Beispiel hierfür ist die Instanz der Typklasse `ReadWrite` für die Liste `[a]`:

```
instance ReadWrite a => ReadWrite [a] where
  showRW []     = showString "0"
  showRW (x:xs) = showString "1" . showRW x . showRW xs

  writeRW h []     = hPutStr h "0"
  writeRW h (x:xs) = hPutStr h "1" >> writeRW h x >> writeRW h xs

  readRW ('0':s) = ([], s)
  readRW ('1':s) = (x:xs, r2)
    where (x, r1) = readRW s
          (xs, r2) = readRW r1
```

5.1.1. Darstellung von Strings

Wenn wir nun Instanzen für `[a]` und `Char` definieren, dann können wir bereits Strings (`type String = [Char]`) in der kompakten Repräsentation darstellen. Möchten wir aber die in Kapitel 4.1.4 vorgestellten Verbesserungen implementieren, so müssen wir `[Char]` als Spe-

zialfall einer Liste behandeln, welche einer besonderen Implementierung bedarf. Für Instanzen der Form `instance (ReadWrite` u_1`,...,ReadWrite` u_n`) => ReadWrite (T` u_1 `...` u_n`) where ...` besteht in Curry die Einschränkung, dass $u_1, \ldots, u_n$ Typvariablen sein müssen. Somit lässt sich nicht direkt eine Instanz für `String` definieren, da `String` ein Synonym für `[Char]` ist. Um dies zu umgehen, fügen wir der Typklasse `ReadWrite` Operationen zum Lesen und Schreiben von Listen hinzu:

```
class ReadWrite a where
  ...
  showListRW :: [a] -> ShowS
  showListRW []     = showString "0"
  showListRW (x:xs) = showString "1" . showRW x . showListRW xs

  writeListRW :: Handle -> [a] -> IO ()
  writeListRW h []     = hPutStr h "0"
  writeListRW h (x:xs) = hPutStr h "1" >> writeRW h x >> writeListRW h xs

  readListRW :: String -> ([a], String)
  readListRW ('0':s) = ([], s)
  readListRW ('1':s) = (x:xs, r2)
    where (x, r1) = readRW s
          (xs, r2) = readListRW r1
```

Zu beachten ist hier, dass die Operationen `showListRW`, `writeListRW` und `readListRW` in der Klassendefinition bereits implementiert sind (*default implementation*). Für jeden Typen `a`, für den eine Instanz von `ReadWrite` existiert, lassen sich diese Funktionen nun nutzen, um Ausdrücke vom Typ `[a]` zu lesen und schreiben. Die Instanz für `[a]` lässt sich dann stark vereinfachen:

```
instance ReadWrite a => ReadWrite [a] where
  showRW = showListRW
  writeRW = writeListRW
  readRW = readListRW
```

Diese Umstrukturierung ermöglicht uns jetzt, für die Instanz von `Char` die Listenoperationen zu überschreiben und somit spezielle Operationen für Strings zu definieren.

```
instance ReadWrite Char where
  ...
  showListRW    = <spezielle Implementierung>
  writeListRW h = <spezielle Implementierung>
  readListRW s  = <spezielle Implementierung>
```

Jetzt können wir speziell für Chars Operationen definieren, welche Listen von Chars (also Strings) auf eine spezifische Art und Weise lesen und schreiben. Konkret bedeutet dies, dass wir nun die String-Extraktion implementieren können, wie sie in Kapitel 4.1.4 vorgestellt wurde. Um Strings beim Schreiben (`show(List)RW` sowie `write(List)RW`) extrahieren und ihr auftreten durch Referenzen ersetzen zu können, müssen wir die Typen der Schreiboperationen anpassen. Um den Nebeneffekt, während des Schreibens Strings herauszuziehen und durch Referenzen zu ersetzen, zu ermöglichen, muss beispielsweise die Schreiboperation `show(List)RW` wie folgt geändert werden[1]:

```
type StringMap = [(String, String)]
showRW :: StringMap -> a -> (StringMap, ShowS)
showListRW :: StringMap -> [a] -> (StringMap, [ShowS])
```

`StringMap` ist ein Typ, der eine Abbildung zwischen Strings darstellt. Das linke Element stellt hierbei den String dar, der auftritt, während der rechte String die String-ID ist, die anstelle des Strings in der kompakten Repräsentation eingesetzt wird. Das Hinein- und Herausreichen der `StringMap` erlaubt es uns, die Abbildung zu aktualisieren (Nebeneffekt), während wir die kompakte Repräsentation aufbauen. Beim Schreiben eines Strings gehen wir nun wie folgt vor:

- ▻ Falls der String bereits in der `StringMap` enthalten ist, so geben wir die entsprechende String-ID zurück.
- ▻ Ansonsten ermitteln wir eine neue String-ID, fügen den String zusammen mit der ID in die `StringMap` ein und geben die ID zurück.

Konkret lässt sich dieses Verhalten wie folgt in Curry definieren:

```
instance ReadWrite Char where
  ...
  showListRW params strs str = (strs',index)
  where
    (strs',index) = writeString params strs str
...
writeString :: StringMap -> String -> (StringMap, ShowS)
writeString strs s
  = case lookup s strs of
      Just i  -> (strs, showString i . showChar ';')
      Nothing ->
        let coding = intToASCII (length strs)
        in (insert s coding strs, showString coding . showChar ';')
```

[1] Analog hierzu muss auch die Schreiboperation `write(List)RW` angepasst werden.

Die Funktion `intToASCII :: Int -> String` assoziiert einer Zahl eine möglichst kurze, eindeutige Zeichenkette aus einem Alphabet Σ. Bei der Wahl des Alphabets Σ ist zu beachten, dass resultierende Zeichenketten möglichst kurz sind, um die Länge der String-IDs zu minimieren.

Die Operation `read(List)RW` muss nun auch die **`StringMap`** als Argument erhalten, um die Referenzen wieder in die ursprünglichen Strings umwandeln zu können.

```
readRW :: StringMap -> String -> (a, String)
readListRW :: StringMap -> String -> ([a], String)
```

Da beim Lesen der kompakten Darstellung keine Strings mehr vorliegen, sondern String-IDs, ist die **`StringMap`** keine Abbildung zwischen Strings und String-IDs, sondern zwischen String-IDs und Strings. Beim Lesen wird nun die gelesene String-ID in der **`StringMap`** nachgeschlagen und der zugehörige String zurückgegeben.

Nicht-Extraktion von kurzen Strings

Bei sehr kurzen Strings erweist sich eine Extraktion üblicherweise als unnötig, da durch die Verwendung einer String-ID die Kompaktheit kaum verbessert wird. Darüber hinaus entsteht durch die Extraktion ein Overhead, denn beim Lesevorgang muss die String-ID gelesen sowie in der **`StringMap`** nachgeschlagen werden. Um dies zu vermeiden, können wir eine minimale Länge für Strings festlegen, ab der eine Extraktion stattfindet. Implementieren lässt sich dies wie folgt:

```
minStringLength :: Int
minStringLength = 5

writeString :: StringMap -> String -> (StringMap, ShowS)
writeString strs s
  | isStub s = (strs, showChar ';' . showString s . showChar '"')
  | otherwise
  = case lookup s strs of
      Just i -> (strs, showString i . showChar ';')
      Nothing ->
        let coding = intToASCII aLen (length strs)
        in (insert s coding strs, showString coding . showChar ';')
 where
  isStub str =
    length str < minStringLength && not (elem '"' str) && not (containsNewline str)
```

Wird nun also ein kurzer String gelesen, so wird dieser direkt in die kompakte Repräsentation geschrieben. Da dieser String terminiert werden muss, wird ein Terminierungssymbol `"` angehängt. Somit ist eine wichtige Eigenschaft eines nicht-extrahierten Strings, dass er

das Terminierungssymbol selbst nicht enthält. Als Beispiel betrachten wir die kompakte Darstellung von `["longFunctionName", "x"]`:

```
1a;1;x"0
;longFunctionName
```

Das erste Element wird hierbei durch `a;` dargestellt, wobei `a` die String-ID ist. Das zweite Element wird durch `;x"` direkt dargestellt, da es kürzer als `minStringLength` ist.

Beim Lesen des Strings wird nun also wie folgt vorgegangen:

- ▷ Wird eine nicht-leere Zeichenkette vor dem nächsten Semikolon gelesen, so wird diese als String-ID interpretiert.
- ▷ Wird stattdessen direkt ein Semikolon gelesen, so wird die nächste Zeichenkette (mit `"` als Terminierungssymbol) als nicht-extrahierter String interpretiert.

Durch diese Implementierung wird die Laufzeit in bestimmten Fällen verbessert, da das Nachschlagen der kurzen Strings erspart bleibt. Dabei sind nur kleine Verschlechterungen der Kompaktheit zu erwarten. Praktisch anwendbar ist dies insbesondere bei der Darstellung von Variablennamen (z.B. in FlatCurry-Programmen), die in der Regel kurz sind.

`Trie` als alternative Datenstruktur für `StringMap`

Bisher verwenden wir eine einfache Liste von Paaren als **`StringMap`**. Dies hat den Nachteil, dass die Suche nach einer String-ID in der Liste in $\mathcal{O}(n)$ erfolgt, wobei n die Anzahl der Einträge in der Liste ist, da jeder Eintrag überprüft werden muss. Um die Laufzeit der Suche zu verbessern, bietet es sich an, baumbasierte Datenstrukturen zu verwenden. Eine in Curry verfügbare Datenstruktur, die als Binärbaum implementiert ist, wird vom Modul **`Data.Map`** bereitgestellt. Praktisch verbessert sich die Laufzeit beim Verwenden einer **`Map`** gegenüber einer Liste von Paaren lediglich in sehr großen **`StringMaps`**. Zwei Gründe sprechen insbesondere im Hinblick auf den Aufbau der Schlüssel (Zeichenketten) gegen die Verwendung von Binärbäumen:

1. Für sehr kleine Listen lohnt sich eine binäre Suche nicht, da der Aufbau des Baumes einen Overhead verursacht. Darüber hinaus ist die Speichereffiizienz von Listen besser als die von Binärbäumen.
2. Normale Binärbäume sind für die Suche nach Strings nicht geeignet, da sie nur auf der Basis von Ordnungsrelationen arbeiten. Vergleiche von Strings sind grundsätzlich teuer. Die Struktur eines Strings wird völlig außer Acht gelassen und wichtige Optimierungsmöglichkeiten bleiben ungenutzt.

Gesucht ist nun also eine Datenstruktur, die ein effizientes Nachschlagen von String-IDs

ermöglicht und dabei die Geschwindigkeitsvorteile unterschiedlicher Ansätze kombiniert. Hierfür bietet sich eine Datenstruktur namens *Trie* an. Ein *Trie* ist ein Suchbaum, der eine effiziente Abbildung zwischen Zeichenketten und Informationen ermöglicht. Betrachten wir beispielsweise einen Trie, der die Strings `"x"`, `"xs"`, `"map"` und `"max"` enthält. Abbildung 5.1 stellt diesen Trie dar.

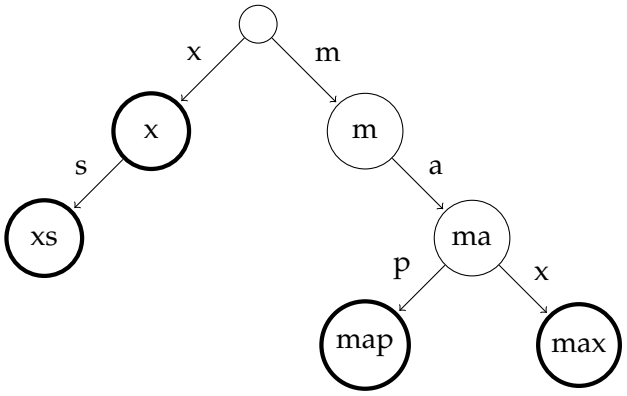


Abbildung 5.1. Beispiel eines Tries

Jeder Knoten im Trie hat für ein Alphabet Σ von Symbolen bis zu $|\Sigma|$ ausgehende Kanten, welche mit unterschiedlichen Symbolen beschriftet sind. Ein Pfad von der Wurzel zu einem markierten Knoten repräsentiert einen Schlüssel. Bei einer einfachen Implementierung von Tries, bei der die ausgehenden Kanten der Knoten in Listen gespeichert werden, erfolgt die Suche nach einem Schlüssel s in einem Trie in $\mathcal{O}(|\Sigma| \cdot |s|)$. Der Trie aus dem oberen Beispiel unterscheidet lediglich zwischen markierten und unmarkierten Knoten, weshalb er sich eignet, um schnell zu prüfen, ob ein String in diesem Baum enthalten ist. Es ist weiterhin möglich, die Trie-Struktur so anzupassen, dass nicht zwischen markierten und unmarkierten Knoten unterschieden wird, sondern stattdessen Informationen an den Positionen im Trie gespeichert werden, welche durch die Schlüssel erreicht werden.

Abbildung 5.2 stellt einen Trie dar, der String-IDs auf Strings abbildet. Konkret handelt es nun also um eine alternative Darstellung der `StringMap`, die wir in der Implementierung von `ReadWrite` verwenden. Dargestellt ist ein Trie, welcher der `StringMap [("a", "foldr"), ("b", "foldl"), ("aa", "map"), ("ab", "zipWith")]` mit Σ `= "ab"` entspricht.

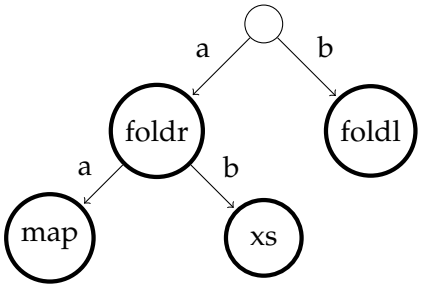
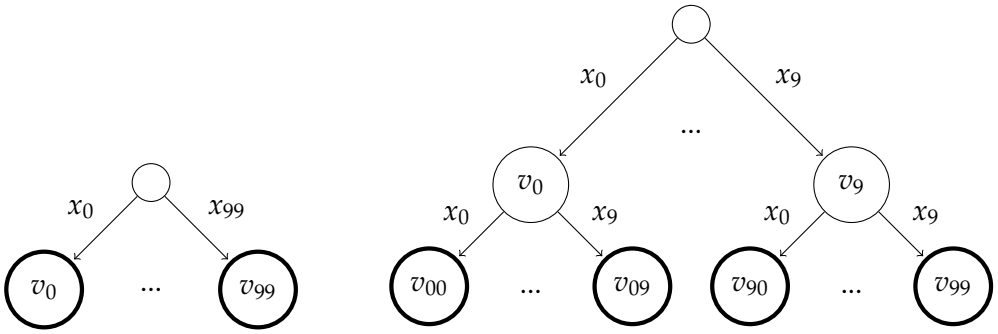


Abbildung 5.2. Abbildung von String-IDs auf Strings

Die Implementierung der Trie-Datenstruktur in Curry erweist sich als unkompliziert. Die Datenstruktur lässt sich in Currys Typsystem wie folgt definieren:

```
data Trie a = Trie (Maybe a) [(Char, Trie a)]
```

Wichtig ist die Wahl des Alphabets Σ, welches die Menge der Symbole darstellt, die in den Schlüsseln des Tries vorkommen. Wird das Alphabet zu groß gewählt, so steigt die Anzahl der ausgehenden Kanten der Knoten, was zu einer ineffizienten Suche führen kann (horizontale Ausdehnung). Wird das Alphabet zu klein gewählt, so steigt die Tiefe des Baums, was zu einer ineffizienten Suche führen kann. Abbildung 5.3 zeigt zwei Tries, die unterschiedliche Alphabete verwenden. Der linke Trie verwendet ein Alphabet Σ mit $|\Sigma| = 100$, wobei die Wurzel direkt mit 100 Knoten (Blättern) verbunden ist. Dies ist ein typisches Beispiel für ein schlecht gewähltes Alphabet, welches zu einer erheblichen horizontalen Ausdehnung des Baums führt. Der rechte Trie verwendet ein Alphabet Σ mit $|\Sigma| = 10$, wobei die Wurzel mit zehn Knoten verbunden ist, die wiederum mit je zehn Knoten verbunden sind. Möchte man zum Knoten mit der Beschriftung v_{90} gelangen, so sind im linken Beispiel 91 Kanten zu überprüfen (x_0 bis x_{90}), während im rechten Beispiel lediglich zehn Kanten geprüft werden müssen (x_0 bis x_9 von der Wurzel ausgehend sowie x_0 von v_9 ausgehend).



(a) $\Sigma = \{x_0, \ldots, x_{99}\}$ **(b)** $\Sigma = \{x_0, \ldots, x_9\}$

Abbildung 5.3. Tries mit unterschiedlichen Alphabeten

Praktisch zeigt sich, dass die Implementierung von Tries in Curry sehr effizient ist. Hierdurch lässt sich das Nachschlagen von String-IDs in der `StringMap` beschleunigen, was insbesondere bei Daten mit vielen Strings zu einer Verbesserung der Laufzeit führt. Nicht nur zum Nachschlagen von String-IDs beim Lesen der kompakten Daten ist ein Trie unübertroffen effizient, sondern auch beim Schreiben, da auch das Nachschlagen von arbiträren Strings in der `StringMap` beschleunigt wird. Insbesondere können Tries auch dann sehr effizient sein, wenn sich die Schlüssel lange Präfixe teilen. Hierdurch wird insbesondere das Schreiben von FlatCurry-Datentermen beschleunigt, da viele Daten lange, automatisch generierte Bezeichner enthalten, die sich in der Regel lange Präfixe teilen.

Parametrisierung der String-Extraktion

In den letzten Abschnitten wurden einige Verfeinerungen der String-Extraktion vorgestellt. Jedoch ist weiterhin unklar, wie nun Größen wie `minStringLength` oder die Länge des Alphabets Σ gewählt werden. Grundsätzlich ist es für einen Anwender erstrebenswert, Kontrolle über diese Funktionen zu haben, um für konkrete Anwendungsfelder optimale Werte zu wählen. Um dies zu ermöglichen, werden die Schreibeoperationen parametrisiert. Beispielsweise die Operationen `showRW` und `writeRW` werden nun also angepasst, um zusätzliche Parameter zu akzeptieren:

```
--- Writing parameters.
data Parameters = Parameters
  { minStrLen   :: Int
  , alphabetLen :: Int
  }

class ReadWrite a where
  ...
  showRW :: Parameters -> Trie String -> a -> (Trie String, ShowS)
  writeRW :: Parameters -> System.IO.Handle -> a -> Trie String -> IO (Trie String)
```

Das Schreiben von kompakten Daten ist nun also parametrisiert. Da beim Lesen von kompakten Daten die korrekten String-IDs aus der Liste vorhandener Strings ermittelt werden müssen, muss zusätzlich die Alphabetgröße geschrieben werden. Der Parameter `minStrLen`, der die minimale Länge extrahierter Strings angibt, beeinflusst nur den Schreibvorgang, weshalb er nur in der Schreiboperation vorkommt und nicht geschrieben werden muss.

Möchte man nun mit `minStrLn=6` und `alphabetLen=26` den Term `["longFunctionName", "x"]` kompakt repräsentieren, dann ist das Ergebnis wie folgt:

```
26
1a;1;x"0
;longFunctionName
```

Vor der eigentlichen kompakten Repräsentation wird nun der Parameter `alphabetLen` geschrieben. Dieser Wert gibt die Länge des Alphabets an, welches zur Kodierung der String-IDs verwendet wird. Anschließend wird die kompakte Repräsentation geschrieben.

Weiterhin bietet es sich an, den verwendeten Container der **`StringMap`** für den Lesevorgang zu parametrisieren. In den bisherigen Beispielen wurde ein **`Trie`** verwendet, um die **`StringMap`** zu repräsentieren. Jedoch ist es durchaus denkbar, dass für bestimmte Anwendungsfälle eine andere Datenstruktur besser geeignet ist. Um dies zu ermöglichen, lässt sich beispielsweise die Operation `readRW` derart anpassen, dass sie einen beliebigen Container für die **`StringMap`** akzeptiert:

```
class ReadWrite a where
  ...
  readRW :: Container c => c -> String -> (a,String)
```

Ein **`Container`** ist hierbei ein abstrakter Typ, der Operationen zum Einfügen, Löschen und Suchen von Elementen bereitstellt. Die konkrete Implementierung des Containers ist hierbei nicht von Bedeutung, solange die Operationen zur Verfügung stehen. Durch Parametrisierung lassen sich flexibel verschiedene Container für die **`StringMap`** verwenden, ohne dass die Implementierung von **`ReadWrite`** angepasst werden muss. Dies ist insbesondere praktisch, um schnell und unkompliziert die Auswirkungen des vorgestellten Tries auf die Laufzeit zu testen.

5.1.2. Implementierung der Typidentifikation

Um Typen konstruieren und darstellen zu können, benötigen wir zunächst eine Repräsentation für Typen. Der konkrete Typ eines Ausdrucks kann in Curry durch

```
data RWType = RWType String [RWType]
 deriving Eq
```

dargestellt werden.

Dabei ist das erste Argument des Konstruktors der Name des Typs, während das zweite Argument eine Liste von Typen ist, welche die Typvariablen des Typs ersetzen. Insbesondere gilt also für monomorphe Typen, dass das zweite Argument die leere Liste ist. Ein paar Beispiele für Typen und ihre Repräsentation sind:

```
Int          →  RWType "Int" []
Maybe Float  →  RWType "Maybe" [RWType "Float" []]
[Char]       →  RWType "[]" [RWType "Char" []]
```

Da jeder Ausdruck seinem Typen zugeordnet werden soll, erweitern wir die Typklasse **`ReadWrite`** um eine Operation, die den Typ eines Ausdrucks berechnet:

```
class ReadWrite a where
  ...
  typeOf :: a -> RWType
```

Das Argument vom Typen `a` ist hierbei der Ausdruck, dessen Typ berechnet werden soll. Da der Wert des Ausdrucks nicht von Bedeutung ist, sondern nur der Typ, wird er niemals ausgewertet.

Für monomorphe Typen ist die Implementierung von `typeOf` trivial, da der Typ des Ausdrucks bereits bekannt ist. Beispielsweise ist

```
instance ReadWrite Int where
  ...
  typeOf _ = RWType "Int" []
```

Für polymorphe Typen ist die Implementierung von `typeOf` etwas komplexer, da der Typ des Ausdrucks von den Typen der Subausdrücke abhängt. Versuchen wir beispielsweise, `typeOf` für `[a]` zu implementieren:

```
instance ReadWrite a => ReadWrite [a] where
  ...
  typeOf (x:_) = RWType "[]" [typeOf x]
```

Die Implementierung von `typeOf` für `[a]` ist hierbei nicht vollständig, da sie nur für nicht-leere Listen definiert ist. Weiterhin wird das Argument von `typeOf` ausgewertet, was gegen die Voraussetzung verstößt und die Implementierung von `typeOf` unflexibel macht. Eine Implementierung, die für alle Listen funktioniert, ist jedoch notwendig. Für Listen können wir beispielsweise wie folgt vorgehen:

```
instance ReadWrite a => ReadWrite [a] where
  ...
  typeOf xs = RWType "[]" [typeOf n]
   where (n:_) = failed:xs
```

Hierdurch erzwingen wir `n :: a`. Da `typeOf` für `a` definiert ist, können wir `typeOf n` aufrufen, um den Typ von `n` zu erhalten.

Nun stellt sich die Frage, wie sich `typeOf` für komplexere polymorphe Typen implementieren lässt. Betrachten wir beispielsweise den Typ `Maybe a`. Um an den Typen `a` zu gelangen, muss der Typ des ersten Arguments des Konstruktors `Maybe` berechnet werden:

```
instance ReadWrite a => ReadWrite (Maybe a) where
  ...
  typeOf x = RWType "Maybe" [typeOf n]
```

```
    where [Just n,_] = [Just failed, x]
```

Hierbei wird abermals `n :: a` erzwungen, um `typeOf n` aufrufen zu können. Bei deutlich komplexeren Typen wie `data MyType a = MyCons (Mabye [a])` wird die Implementierung von `typeOf` schnell unübersichtlich und schwer wartbar. Insbesondere wird die automatische, schematische Generierung von `typeOf` für komplexe Typen sehr komplex, weil aufwendige Analysen notwendig werden, um die konkreten Typen zu ermitteln, welche die Typvariablen ersetzen. Abgesehen davon ist die Implementierung von `typeOf` für einige Typen auf diese Weise sogar unmöglich. Betrachten wir die polymorphe Typdeklaration für den eigentlich monomorphen Typen `data MyType2 a = Cons`. Die Typdeklaration `MyType2` enthält zwar eine Typvariable, jedoch taucht diese niemals als Typausdruck in den Konstruktordeklarationen auf. Ausdrücke wie `Cons :: [(Maybe Int, [Either Char Float])]` sind also gültig, obwohl `a` in der Typdeklaration von `MyType2` nicht vorkommt. Intuitiv lässt sich eine Implementierung von `typeOf` für `MyType2` wie folgt aufschreiben:

```
instance ReadWrite a => ReadWrite (MyType2 a) where
  ...
  typeOf _ = RWType "MyType2" [typeOf (failed :: a)]
```

Leider ist ein solcher Aufschrieb in Curry nicht erlaubt, weil Typvariablen von Instanzen nicht in Typannotationen vorkommen dürfen. Wir können dies aber umgehen, indem wir lokale Funktionen verwenden, welche die Typen der Subausdrücke berechnen. Die Implementierung von `typeOf` für `MyType2` könnte also wie folgt aussehen:

```
instance ReadWrite a => ReadWrite (MyType2 a) where
  ...
  typeOf n = RWType "MyType2" [typeOf (get_a n)]
   where
    get_a :: MyType2 a' -> a'
    get_a _ = failed
```

Dieser Ansatz lässt sich sehr einfach auf beliebig komplexe Typen anwenden, da die Typen der Subausdrücke in lokalen Funktionen berechnet werden, weshalb die automatische Generierung der Funktion `typeOf` mit AbstractCurry sehr einfach ist. Im Kapitel 5.2 wird auf die automatische Generierung von Code für Funktionen wie `typeOf` eingegangen.

Mittels einer Funktion, welche den Datentypen `RWType` textuell darstellt (*pretty printer*), lässt sich der Typ eines Ausdrucks nun leicht in einen String überführen. Dieser String kann dann zusätzlich zu den kompakten Daten geschrieben werden, um den Typen der dargestellten Daten zu speichern. Die Ausgabe für `showData (1 :: Int)` sieht dann beispielsweise wie folgt aus:

```
26
Int
```

```
1;
```

Die erste Zeile gibt die parametrisierte Länge des Alphabets an, wie in Kapitel 5.1.1 beschrieben wurde, während die zweite Zeile den Typen angibt. Die dritte Zeile stellt die kompakte Repräsentation des Ausdrucks dar. Der folgende Code zeigt Teile der Implementierung der Funktion `readData`, welche beim Lesen die Typprüfung übernimmt:

```
readData :: ReadWrite a => String -> Maybe a
readData ls =
  let n@(sLen,t,_,_) = parseInput ls
      result         = calculate n
  in if ppType (typeOf (fst result)) == t
      then Just $ fst result
      else Nothing
 where
  calculate = ...
```

Hierbei ist `t` der gelesene Typ und `result :: a` das Ergebnis. Sind die zu lesenden Daten nicht mit dem Typen des Parsers kompatibel, so wird `Nothing` zurückgegeben. Ansonsten wird der gelesene Ausdruck zurückgegeben. Die Funktion `ppType` ist ein *pretty printer* für den Datentyp `RWType`, welcher den Typen in einen String überführt. Durch die bedarfsgesteuerte Auswertung von `result` wird die kompakte Darstellung grundsätzlich nicht ausgewertet, wenn der Typ nicht mit dem gelesenen Typ übereinstimmt.

Neben der Typkorrektheit ist ebenfalls die Korrektheit der kompakten Darstellung relevant. Beispielsweise durch eine unzulässige Bearbeitung gespeicherter Daten in kompakter Form kann die kompakte Darstellung ungültig werden. Um Programmabstürze beim Laden ungültiger Daten zu vermeiden, bietet Curry Mechanismen an, um Laufzeitfehler zu erkennen. Da Laufzeitfehler in Curry als Fehlschläge behandelt werden, können sie durch Kapselung von Nichtdeterminismus erkannt werden. Die Operation `oneValue :: a -> Maybe a` wertet ihren Argumenten aus und gibt das Ergebnis im Falle einer erfolgreichen Ausführung zurück - ansonsten wird `Nothing` zurückgegeben. Die Kapselungsoperation ist jedoch mit einer Verschlechterungen der Laufzeit verbunden, weshalb sie sich für performancekritische Berechnunge weniger eignet. Stattdessen bietet sich die `IO`-Operation `catch :: IO a -> (IOError -> IO a) -> IO a` an, wodurch Ausnahmebehandlungen in der `IO`-Monade ohne nennenswerten Overhead ermöglicht werden. Wir ergänzen die Operation `loadData`, welche die kompakten Daten direkt aus einer Datei liest:

```
loadData :: ReadWrite a => String -> IO (Maybe a)
loadData file = do
  dt <- readFile file
  catch (return $ readData dt)
        (\_ -> return Nothing)
```

Im Falle eines Fehlschlages wird nun `Nothing` zurückgegeben (Zeile vier). Die Operation `loadData` ist also eine sichere `IO`-Aktion, um kompakte Daten zu laden, ohne Laufzeitfehler aufgrund ungültiger Daten zu erzeugen.

5.2. Das Tool

Um zu vermeiden, dass Entwickler für jeden selbstdefinierten Datentypen die Operationen zum Lesen und Schreiben der Daten in kompakter Darstellung sowie die Typidentifikation manuell implementieren müssen, wurde im Rahmen dieser Arbeit ein Tool entwickelt, welches diese Operationen automatisch generiert. Das Tool ist als *Command Line Interface* (CLI) konzipiert und ermöglicht es, Module mit Typdeklarationen zu analysieren und daraus die Operationen zum Lesen und Schreiben der Daten in kompakter Darstellung sowie die Typidentifikation zu generieren. Durch die Nutzung des Tools wird die Implementierung von Operationen zum Lesen und Schreiben von Daten in kompakter Darstellung sowie zur Typidentifikation automatisiert, was die Entwicklung von Programmen in Curry erleichtert.

Ablauf des Tools

Als typisches Beispiel für ein solches Werkzeug zum Generieren zum Quelltext betrachten wir zunächst `xmldata`. Das Paket `xmldata` enthält ein Tool, das es ermöglicht, aus Modulen mit Typdeklarationen automatisch Operationen zum Lesen und Schreiben der Daten in XML-Darstellung zu generieren. Das Tool ist als *Command Line Interface* (CLI) konzipiert. Zur Analyse der Module sowie zur Generierung neuer Module wird das Paket `AbstractCurry` genutzt, womit es sich bei `xmldata` um ein typisches Metaprogramm handelt. In ähnlicher Weise ist auch das Tool, das in dieser Arbeit entwickelt wurde, als Metaprogramm konzipiert.

Das Tool erwartet als Eingabe die Namen oder Pfade der Module, für welche die Instanzen von `ReadWrite` generiert werden sollen. Dabei geht das Tool pro Modul wie folgt vor:

1. Lesen und Analyse des Moduls

Das Modul `FlatCurry.Files` wird genutzt, um für das gewünschte Modul die Typdeklarationen zu lesen. Anschließend wird das Modul hinsichtlich der Typdeklarationen analysiert. Wichtig ist hierbei, dass lediglich Typdeklarationen berücksichtigt werden, die mittels `data` oder `newtype` definiert sind. Typsynonyme werden nicht berücksichtigt, da sie keine neuen Typen definieren. Weiterhin werden alle Typen, welche in Konstruktordeklarationen auftauchen, gesammelt. Alle Typen, welche in Konstruktordeklarationen auftauchen, nicht aber im selben Modul definiert sind, werden als externe Typen betrachtet. Da für externe Typen keine Instanzen von `ReadWrite` generiert werden können, wird ein Hinweis ausgegeben, dass die Instanzen für diese Typen unter Umständen zusätzlich durch den Nutzer unter Verwendung des Tools generiert werden müssen.

Grundsätzlich könnte das Tool alle Abhängigkeiten laden und für diese Module ebenfalls alle notwendigen Instanzen generieren. Dies ist aber üblicherweise nicht erwünscht, weil die Anzahl der generierten Instanzen schnell sehr groß werden kann. Das Tool für jedes notwendige Modul aufzurufen, sorgt dafür, dass keine Instanzen mehr als ein mal generiert werden.

2. Generierung der Instanzen von `ReadWrite`

Um Metaprogrammierung zu betreiben und Quelltext für die Klasseninstanzen zu generieren, wird das Paket **`AbstractCurry`** genutzt. Das Modul **`AbstractCurry.Types`** stellt ein ausführliches Typsystem bereit, welches es ermöglicht, Curry-Quelltext zu repräsentieren und zu generieren. Für jede auftretende Typdeklaration wird eine Instanz von `ReadWrite` generiert. Um die schematische Übersetzung von Typdeklarationen zu Instanzen von `ReadWrite` zu ermöglichen, wie sie in Kapitel 5.1 vorgestellt wurde, werden Funktionsgeneratoren verwendet:

```
type FunctionGenerator = CTypeDecl -> RWM [CRule]
```

Ein **`FunctionGenerator`** ist also eine Funktion, welche eine Typdeklaration in eine **`AbstractCurry`**-Repräsentation von Funktionsregeln übersetzt. Die Berechnung (Codegenerierung) wird dabei in die `ReadWrite`-Monade **`RWM`** gehoben, um die Flexibilität zu erhöhen und Nebeneffekte zu ermöglichen. Pro Instanzfunktion für **`ReadWrite`** (d.h. `showRW`, `writeRW`, `readRW` sowie `typeOf`) ist ein **`FunctionGenerator`** definiert, der die jeweilige Funktion generiert.

Beispielhaft ist im Folgenden der **`FunctionGenerator`** für die Funktion `typeOf` skizziert:

```
1  --- typeOf generator implementation
2  ---
3  --- For a data definition
4  ---  data T a b ... = ...
5  --- this function generates the following code:
6  ---  typeOf :: T a b ... -> RWType
7  ---  typeOf n = RWType "T" [typeOf (get_a n), typeOf (get_b n), ...]
8  ---    where
9  ---      get_a :: T a b ... -> a
10 ---      get_a (T a b ...) = failed
11 ---      ...
12 generatorTypeOf :: FunctionGenerator
13 generatorTypeOf typedecl = do
14   if isMonomorphic typedecl
15     then return [CRule [anonPattern] (CSimpleRhs (applyF ("ReadWriteBase", "mono")
16                                        [string2ac (snd $ typeName typedecl)]) [])]
17     else return ...
```

Für monomorphe Typen ist die Generierung des Quelltexte trivial, da der Typ des Ausdrucks bereits bekannt ist. Eine Ausgabe für die Typdeklaration `data T = Cons` könnte also wie folgt aussehen:

```
typeOf _ = mono "T"
```

Dabei ist `mono :: String -> RWType` eine Funktion, die einen monomorphen Typen zurückgibt. Die kompliziertere Implementierung von `typeOf` für polymorphe Typen lässt sich leicht durch die Verwendung von `Build`-Funktionen umsetzen, welche im Modul `AbstractCurry.Build`[2] sowie im in dieser Arbeit implementierten Modul `RW.Build`[3] definiert sind.

Pro Typdeklaration wird eine Klasseninstanz für `ReadWrite` generiert, welche dann mit Funktionen gefüllt wird, die von den vier Funktionsgeneratoren generiert werden.

3. Generierung der Ausgabe

Das Modul `AbstractCurry.Pretty` wird genutzt, um das generierte Modul in Quelltext zu überführen. Der Quelltext wird dann in eine Datei geschrieben, die den Namen des ursprünglichen Moduls mit dem Suffix `RW` trägt. Wurde dabei ein Pfad zu einem Modul angegeben, so wird die Datei im selben Verzeichnis wie das ursprüngliche Modul gespeichert. Wurde hingegen ein Modulname angegeben, so wird die Datei im Verzeichnis, in dem das Tool ausgeführt wurde, gespeichert.

Um in jedem Fall die semantische Korrektheit des generierten Quelltexts zu gewährleisten, muss die Option `showLocalSigs` des PrettyPrinters auf `True` gesetzt werden. Diese Option sorgt dafür, dass die Signaturen lokaler Funktionsdefinitionen im generierten Quelltext angezeigt werden. Dies ist bei der Generierung der Funktion `typeOf` für polymorphe Typen notwendig:

```
1 typeOf n = RWType "MyType2" [typeOf (get_a n)]
2   where
3     get_a :: MyType2 a' -> a'
4     get_a _ = failed
```

Würde der Quelltext in Zeile drei fehlen, so wäre der generierte Quelltext nicht semantisch korrekt, da der Typ der Funktion `get_a` nicht bekannt wäre.

5.3. Parametrisierung

Um die in Kapitel 5.1 vorgestellten Parametrisierung des Schreib- und Lesevorgangs durch das Tool zu ermöglichen, wird die Eingabe des Tools um zusätzliche Parameter erweitert.

[2] https://git.ps.informatik.uni-kiel.de/curry-packages/abstract-curry/-/blob/master/src/AbstractCurry/Build.curry
[3] https://git.ps.informatik.uni-kiel.de/theses/2023/2023-lzuengel-ba/-/blob/main/rw-data/src/RW/Build.curry

Um die Eingabeargumente einfach verarbeiten und mit einfachen Mitteln ein leistungsstarkes *Command Line Interface* erstellen zu können, wird das Modul `System.Console.GetOpt` genutzt.

Anhang D beschreibt die Verwendung des Tools und die möglichen Eingabeparameter. Die Eingabeparameter ermöglichen es, die Größe des Alphabets, die minimale Länge von Strings, die zu kompakten Daten extrahiert werden, sowie die Pfade oder Namen der Module, für welche die Instanzen von `ReadWrite` generiert werden sollen, zu spezifizieren.

Kapitel 6

Auswertung

In diesem Kapitel werden die Ergebnisse dieser Arbeit vorgestellt und diskutiert. Hierbei werden die Geschwindigkeitsvorteile gegenüber der bestehenden Lösung in PAKCS und KiCS2 aufgezeigt und die Ergebnisse der Benchmarks analysiert. Ein Hauptaugenmerk liegt dabei auf der Laufzeit des Lesens von `FlatCurry`-Termen aus Dateien, da es sich hierbei um eine übliche und aufwändige Operation handelt. Da viele Tools und Bibliotheken vom schnelleren Einlesen von `FlatCurry`-Daten profitieren können, ist eine Untersuchung der Laufzeit von besonderem Interesse. Insbesondere wird untersucht, inwieweit das in dieser Arbeit vorgestellte Tool von der kompakten Repräsentation profitiert und wie stark die Laufzeit durch die kompakte Repräsentation verbessert werden kann. Somit werden die Ergebnisse dieser Arbeit auf das in dieser Arbeit entwickelte Tool selbst angewendet.

6.1. Übersicht allgemeiner Benchmarks

Im Folgenden wird ein einfaches Beispiel für algebraische Datentypen, wie sie in Curry auftreten könnten, betrachtet. Hierbei handelt es sich um eine einfache Repräsentation von Peano-Zahlen. Die Peano-Zahlen sind eine Möglichkeit, natürliche Zahlen rekursiv darzustellen. Eine Peano-Zahl ist entweder die Null oder der Nachfolger einer Peano-Zahl. Definiert ist der Datentyp induktiv wie folgt:

```
data Nat = Zero | Successor Nat
```

Abbildung 6.1 zeigt die Laufzeit[1] des Lesens einer Peano-Zahl der Größe Tausend in PAKCS sowie Einhunderttausend in KiCS2. Konkret wird hierbei die entwickelte kompakte Darstellung (`rw-data`) mit den bestehenden Lösungen (`read`) und `ReadShowTerm` verglichen. Gegenüber der in Curry implementierten `read`-Operationen ist der Geschwindigkeitsunterschied in PAKCS und KiCS2 deutlich. In PAKCS ist die Laufzeit des Lesens einer Peano-Zahl der Größe Tausend mit der kompakten Darstellung etwa 100-mal schneller als die bestehende Lösung. In KiCS2 ist die Laufzeit des Lesens einer Peano-Zahl der Größe einer 100.000 mit der kompakten Darstellung bis zu zweihundertmal so schnell wie die bestehende Lösung. Vergleicht man die Laufzeit der kompakten Darstellung mit der `ReadShowTerm`-Operation, so

[1] Alle Benchmarks wurden unter Ubuntu 22.04.3 LTS (VirtualBox 7.0.10) bei Verwendung eines Intel Core i7 10700K (4.7GHz) mit 3733MHz-DDR4-RAM (CL16) durchgeführt.

ist die kompakte Darstellung in PAKCS etwa viermal so schnell und in KiCS2 etwa siebzigmal so schnell. Dieser Geschwindigkeitsunterschied lässt sich dadurch erklären, dass die in Prolog geschriebene Implementierung der Leseoperation des Moduls `ReadShowTerm` sehr viel stärker optimiert ist als der von Curry nach Prolog kompilierte Code von `rw-data`. Relativ gesehen ist der Geschwindigkeitsunterschied in KiCS2 sehr viel größer als in PAKCS, da der von Curry nach Haskell kompilierte Code der kompakten Darstellung sehr viel näher an die Geschwindigkeit nativen Haskellcodes heranreicht als bei PAKCS und Prolog. Hieraus ergibt sich, dass die Nutzung der kompakten Darstellung in KiCS2 besonders lohnenswert ist. Da KiCS2 im Allgemeinen schnellere Programme erzeugt als PAKCS und damit die erste Wahl für die Erzeugung von schnellen Programmen ist, bietet sich der Geschwindigkeitsvorteil der kompakten Darstellung besonders an.

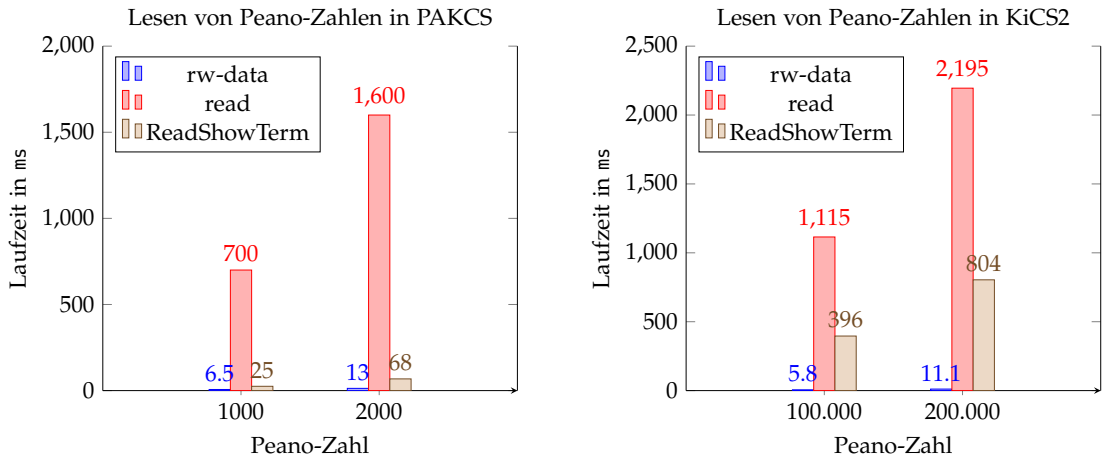


Abbildung 6.1. Laufzeit des Lesens von Peano-Zahlen

Zur automatisierten Generierung von Benchmarkergebnissen wurde das Modul **`Benchmark`**[2] entwickelt, welches die Laufzeit verschiedener Operationen misst und die Ergebnisse strukturiert ausgibt. Auch bei anderen algebraischen Datentypen wie Binärbäumen zeichnet sich ein ähnliches Bild ab. Für die praktische Anwendung sind Daten mit vielen Strings interessanter, weshalb im Folgenden die Laufzeit des Lesens von **`FlatCurry`**-Daten betrachtet wird.

6.2. Anwendung bei `FlatCurry`

Um die Vorteile der kompakten Darstellung direkt mit **`FlatCurry`** testen zu können, wurde eine lokale Kopie des **`FlatCurry`**-Repositoriums[3] erstellt und derart angepasst, dass die kompakte

[2] `https://git.ps.informatik.uni-kiel.de/theses/2023/2023-lzuengel-ba/-/blob/main/rw-data/tests/Benchmark.curry`

[3] `https://git.ps.informatik.uni-kiel.de/curry-packages/flatcurry`

Darstellung verwendet wird. Konkret wurde hierbei das Modul **FlatCurry.Files** angepasst, um beim Lesen und Schreiben von Dateien die kompakte Darstellung zu verwenden.

Der Curry-Paketmanager *CPM* erlaubt es, mittels `cypm link flatcurry` die modifizierte Variante von **FlatCurry** im lokalen System verfügbar zu machen. Hierdurch wird die nahtlose Integration der kompakten Darstellung in bestehende Projekte ermöglicht, wie im folgenden Beispiel gezeigt wird.

Selbstanwendung

Ziel dieser Arbeit ist es, eine kompakte Repräsentation von Datentermen zu entwickeln, um das Lesen von Termen aus Dateisystemen zu beschleunigen. Insbesondere sollen Werkzeuge, Programme und Bibliotheken, die viele Daten lesen und schreiben müssen, von der kompakten Repräsentation profitieren. Ein Beispiel für ein Tool, welches **FlatCurry**-Terme liest und verarbeitet, ist das in dieser Arbeit vorgestellte Tool. Es liest **FlatCurry**-Darstellungen von Curry-Modulen aus Dateien, analysiert diese und arbeitet mit diesen Daten weiter. Da ein nicht unwesentlicher Anteil der Laufzeit des Tools durch das Lesen der Daten bestimmt wird, kann es für die Gesamtlaufzeit sinnvoll sein, die Lesevorgänge zu beschleunigen. Somit bietet es sich an, die vorgestellten Techniken auf dieses Tool selbst anzuwenden und zu überprüfen, wie stark die Laufzeit durch die kompakte Repräsentation verbessert werden kann.

Es stellt sich heraus, dass **FlatCurry**-Daten, welche in kompakter Form vorliegen, deutlich schneller gelesen werden können als in der bestehenden Lösung. Üblicherweise sind die Daten in kompakter Form etwa 70% kleiner als in der bestehenden Lösung und können mit KiCS2 zehnmal so schnell gelesen werden.

In einem Beispiel wurde ein Modul (`FlatCurry.Types`) mit dem Tool verarbeitet. Die Ausführung des Tools mittels KiCS2 beim Einlesen der 568.7kB großen Daten `Types.fcy` dauerte 0.6 Sekunden. Lagen die Daten jedoch in kompakter Form (174kB) vor, so dauerte die Ausführung nur 0.13 Sekunden. Insgesamt konnte die Laufzeit des Tools in diesem Beispiel also um über 75% reduziert werden. Dieser Geschwindigkeitsvorteil ist insbesondere bei großen **FlatCurry**-Dateien von Vorteil, da hier die Laufzeit des Tools stark von der Lesezeit abhängt. Der überwiegende Rest der Laufzeit des Tools wird durch das *pretty printing* und Ausgeben des Zielcodes bestimmt. Die eigentliche Analyse und Codegenerierung mittels **AbstractCurry** stellt nur einen unwesentlichen Anteil der Laufzeit dar.

Parametrisierung

Getestet wurde die Parametrisierung mit einem **FlatCurry**-Term, welcher das Modul `ReadWriteBase` repräsentiert. In diesem Modul, welches grundlegende Funktionalitäten der kompakten Darstellung bereitstellt, werden viele Funktionen, Instanzen und Datentypen definiert. Durch den für Curry-Module typischen Aufbau eignet sich dieses Modul besonders gut für die Untersuchung der Laufzeit des Lesens von **FlatCurry**-Daten. Die Daten enthalten

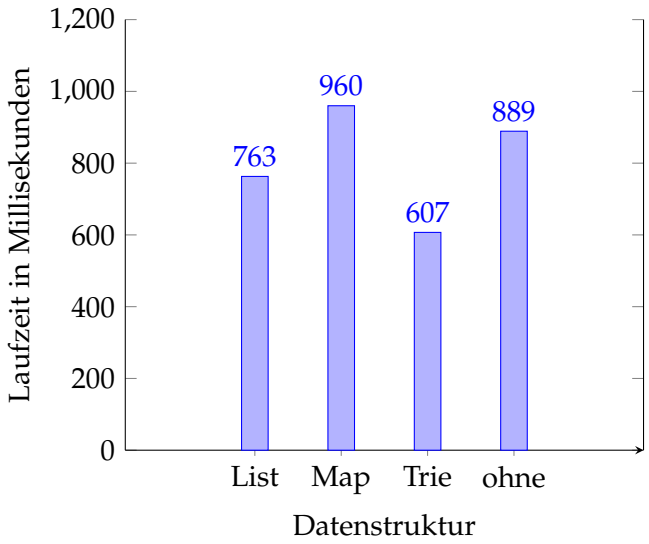


Abbildung 6.2. Laufzeit des Lesens von `ReadWriteBase.fcy` in PAKCS

235 unterschiedliche Strings.

Abbildung 6.2 zeigt die Lesedauer der **`FlatCurry`**-Datei unter der Verwendung verschiedener Datenstrukturen zum Abbilden zwischen String-IDs und Strings. Wie zu sehen ist, ist die in dieser Arbeit entwickelte Trie-Datenstruktur die schnellste Methode, um Strings nachzuschlagen. Gänzlich ohne Stringextraktion ist nicht nur die Kompaktheit am schlechtesten und die Ausgabe zu groß, auch ist die Laufzeit erheblich schlechter als bei der Extraktion von Strings mittels Tries.

Die Wahl eines Alphabets für die String-IDs hat ebenfalls einen Einfluss auf die Laufzeit. Sehr große Alphabete führen zu einer langsameren Laufzeit, da die lineare Komponente der Suche in Tries sehr groß wird (horizontale Ausdehnung). Insbesondere KiCS2 profitiert von kleineren Alphabeten. Hier hat sich herausgestellt, dass eine Alphabetgröße von zehn Zeichen eine gute Ausgangsgröße ist. Insgesamt zeigt sich, dass eine Wahl zwischen zehn und 26 Symbolen die besten Laufzeitergebnisse bei hinreichend guter Kompaktheit ergibt.

Die minimale Länge für extrahierte Strings hat ebenfalls einen Einfluss auf die Laufzeit, wobei sich die Wahl einer Länge von fünf Zeichen als praktisch sinnvoll erwiesen hat. Die durchschnittliche Laufzeitverbesserung bei **`FlatCurry`**-Daten liegt üblicherweise bei $0 - 5\%$.

Kapitel 7

Fazit

In diesem Kapitel wird die Arbeit zusammengefasst und ein Ausblick auf mögliche zukünftige Entwicklungen und sinnvolle Erweiterungen gegeben.

7.1. Zusammenfassung

Im Rahmen dieser Arbeit wurde eine Technik der kompakten Darstellung von Datentermen in Curry vorgestellt, welche die Laufzeit des Lesens von Termen aus Dateien beschleunigen soll. Hierfür wurde eine neue Repräsentation von Datentermen entwickelt, die es ermöglicht, Daten kompakter zu schreiben und damit effizienter zu lesen. Die Ergebnisse der Benchmarks zeigen, dass die kompakte Repräsentation die Laufzeit des Lesens von `FlatCurry`-Termen aus Dateien signifikant verbessern kann, wodurch die Gesamtlaufzeit von Programmen, welche viele `FlatCurry`-Daten laden und verarbeiten, verbessert werden kann. Der Effekt der Laufzeitverbesserung wurde hierbei am Beispiel des in dieser Arbeit vorgestellten Tools gezeigt, welches `FlatCurry`-Daten aus Dateien liest und analysiert. Die kompakte Repräsentation konnte den Lesevorgang von Eingabedaten sowie die Laufzeit des Tools signifikant verbessern. Das in dieser Arbeit vorgestellte Tool automatisiert die Erzeugung von `ReadWrite`-Instanzen, wodurch die kompakte Repräsentation von Datentermen leicht in bestehende Programme integriert werden kann. Die Ergebnisse dieser Arbeit zeigen, dass die kompakte Repräsentation von Datentermen in Curry eine effiziente Möglichkeit darstellt, um die Laufzeit des Lesens von Daten aus Dateien zu verbessern und somit die Gesamtlaufzeit von Programmen zu optimieren.

7.2. Ausblick

Ein wünschenswertes Ziel eines jeden Compilers ist es, dem Programmierer Arbeit abzunehmen. Aus diesem Grund bietet es sich an, die automatische Generierung von `ReadWrite`-Instanzen in das Curry-Frontend zu integrieren. Beispielsweise soll sich für eine Typdeklaration mit der Annotation `deriving ReadWrite` automatisch eine Instanz für `ReadWrite` generieren lassen. Dies würde die Arbeit des Programmierers erleichtern und die Verwendung der kompakten Repräsentation von Datentermen in Curry fördern. Dies ließe sich beispielsweise durch die Verwendung des Curry-Präprozessors realisieren.

Weiterhin kann es sinnvoll sein, nativen Support für Binärdateien in Curry zu implementieren und Möglichkeiten zu schaffen, mit Bytevektoren zu arbeiten, wie sie in Haskell durch `Data.ByteString`[1] bereitgestellt werden. Durch die Verwendung von Bytevektoren könnte das Darstellen von Daten und Lesen von Binärdateien weiter beschleunigt werden, was sich auf die vorgestellte kompakte Repräsentation von Datentermen übertragen ließe.

Die Auswertung hat ergeben, dass Code, welcher direkt in einer Zielsprache (etwa Haskell oder Prolog) implementiert ist, unter Umständen schneller ausgeführt wird als Code, der in Curry geschrieben ist. Aus diesem Grund bietet es sich an, die Implementierung der kompakten Darstellung in den Zielsprachen der Curry-Compiler vorzunehmen und in Curry mittels `external` lediglich die notwendigen Schnittstellen bereitzustellen.

Die in dieser Arbeit vorgestellte Implementierung eines Tries verbessert die Laufzeit des Lesens von **`FlatCurry`**-Daten signifikant, indem sie das Nachschlagen von Schlüsseln (Zeichenketten) in einer großen Menge von Einträgen beschleunigt. Es bietet sich an, die Implementierung des Tries zu erweitern, um insbesondere die Speichernutzung weiter zu verbessern. Hierbei könnten verschiedene Techniken zur Optimierung von Tries, wie etwa die Verwendung von Patricia-Tries [Knu98], in Betracht gezogen werden. Auch losgelöst von dieser Arbeit finden sich in der Praxis viele Anwendungsfälle für Tries, sodass sich eine allgemeine Implementierung eines Tries in Curry als nützlich erweisen kann.

[1] `https://hackage.haskell.org/package/bytestring-0.12.1.0/docs/Data-ByteString.html`

Anhang A

Quellcode

Der Quellcode dieser Arbeit ist im GitLab-Repository unter `https://git.ps.informatik.uni-kiel.de/theses/2023/2023-lzuengel-ba` im Branch "`main`" zu finden. Unter `rw-data/tests/` ist eine umfassende Sammlung von Benchmarks und Tests zu finden.

Im Branch "`feature-cli-containers`" ist eine Variante des Tools zu finden, welche die zum Extrahieren von Strings verwendete Datenstruktur parametrisiert.

Anhang B

Installation

Zur Nutzung des Werkzeugs ist PAKCS oder KiCS2 erforderlich.

Um das Tool zu installieren, muss zunächst das Repository [1] geklont werden. Anschließend muss in das Verzeichnis `rw/` gewechselt werden und das Tool kann mittels `cypm install` kompiliert und installiert werden.

Nun kann das Tool mittels `rw-data` aufgerufen werden. Eine Beispielanwendung ist in Anhang C zu finden. Weitere Anwendungen und Optionen des Tools sind in Anhang D beschrieben.

Um das Paket lokal nutzen zu können, muss das Tool mittels `cypm add --package rw-data` lokal dem CPM hinzugefügt werden.

[1] `https://git.ps.informatik.uni-kiel.de/theses/2023/2023-lzuengel-ba`

Anhang C

Beispiel

Sei das Modul `Peano` gegeben, welches eine Darstellung für natürliche Zahlen bereitstellt:

```haskell
module Peano where

--- Representation of natural numbers using Peano axioms
data Nat = Zero | Successor Nat
  deriving (Show, Eq, Read)

--- Int to Nat
toNat :: Int -> Nat
toNat n | n < 0     = error "toNat: negative number"
        | n == 0    = Zero
        | otherwise = Successor (toNat (n - 1))

--- Nat to Int
fromNat :: Nat -> Int
fromNat Zero          = 0
fromNat (Successor n) = 1 + fromNat n
```

Die Anwendung des Tools im gleichen Verzeichnis mittels `rw-data Peano` liefert ein Modul `PeanoRW`, welches nun beispielsweise in einem neuen Modul importiert und genutzt werden kann:

```haskell
module MyPeanoTest where

import PeanoRW
import Peano
import RW.ReadWriteBase

--- Example usage
main :: IO ()
main = do
  let str        = showData $ toNat 10000
```

```
11   let (Just nat) = readData str
12   print $ fromNat nat
13
14 -- output: 10000
```

Anhang D

Nutzung des Tools

Nach erfolgreicher Installation des Tools kann dieses mittels `rw-data -h` aufgerufen werden. Die Ausgabe des Tools zeigt die möglichen Optionen und deren Verwendung:

```
-----------------------------------------
RW - ReadWrite instance generator for Curry
-----------------------------------------
Usage: rw [options] <path1> <path2> ...
  -h, -?   --help        Print this help message.
  -t TYPE  --type=TYPE   0: Generate only the RW-instances (default).
                         1: Generate RW-instances and a module containing
                            parametrized read and write operations (see below).
                            The generated module will be placed in the same directory
                            as the first input file.
  -s SLEN  --sl=SLEN,    Minimum length of extracted strings. Default is 5.
  -a ALEN  --al=ALEN,    Alphabet length. Default is 26.
                         Alphabet length must be within [1..94].
```

Mittels `rw-data <module> -t 1 -s 6 -a 10` kann das Tool aufgerufen werden. Neben dem Modul, welches die **`ReadWrite`**-Instanzen für die Datentypen enthält, wird ein weiteres Modul `RWOps.curry` generiert, welches parametrisierte Operationen zum Schreiben von Daten in der kompakten Darstellung bereitstellt. In diesem konkreten Beispiel wird die minimale Länge von extrahierten Strings auf sechs und die Alphabetlänge auf zehn gesetzt.

Literatur

[AH09] Sergio Antoy und Michael Hanus. „Set functions for functional logic programming“. In: *Proceedings of the 11th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*. PPDP '09. Coimbra, Portugal: Association for Computing Machinery, 2009, S. 73–82. ISBN: 9781605585680. DOI: 10.1145/1599410.1599420. URL: https://doi.org/10.1145/1599410.1599420.

[CW85] Luca Cardelli und Peter Wegner. „On understanding types, data abstraction, and polymorphism“. In: *ACM Comput. Surv.* 17.4 (Dez. 1985), S. 471–523. ISSN: 0360-0300. DOI: 10.1145/6041.6042. URL: https://doi.org/10.1145/6041.6042.

[Han13] Michael Hanus. „Functional Logic Programming: From Theory to Curry“. In: *Programming Logics: Essays in Memory of Harald Ganzinger*. Berlin, Heidelberg: Springer, 2013, S. 123–168. DOI: 10.1007/978-3-642-37651-1_6.

[Han16] Michael Hanus. *Curry: An Integrated Functional Logic Language*. 2016. URL: https://www.curry-language.org.

[Knu98] Donald E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. USA: Addison Wesley Longman Publishing Co., Inc., 1998. ISBN: 0201896850.

[Tee16] Finn Teegen. „Erweiterung von Curry um Typklassen und Typkonstruktorklassen“. Masterarbeit. Christian-Albrechts-Universität zu Kiel, 2016.