Fachbereich 12 Informatik und Mathematik
Institut für Informatik

Bachelorarbeit zum Thema:

Implementierung einer graphischen Benutzerschnittstelle für ein Programm zum automatischen Korrektheitsnachweis von Programmtransformationen in der funktionalen Programmiersprache Haskell

Tommaso Castrovillari

03. April 2013

eingereicht bei
Prof. Dr. Manfred Schmidt-Schauß
Professur für Künstliche Intelligenz und Softwaretechnologie

Erklärung gemäß Bachelor-Ordnung Informatik 2007 § 24 Abs. 11

Hiermit bestätige ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen Quellen oder Hilfsmittel als die in dieser Arbeit angegebenen verwendet habe.

Frankfurt am Main, den 03. April 2013

Tommaso Castrovillari

Zusammenfassung

Die deutsche Forschungsgemeinschaft (DFG) unterstützt aktuell das Forschungsprojekt "Automatic correctness proofs of program transformations" der Professur für Künstliche Intelligenz und Softwaretechnologie an der Goethe Universität Frankfurt. Dieses Forschungsprojekt befasst sich mit automatischen Korrektheitsnachweisen von Programmtransformationen und entwickelt ein Programm, dass Programmtransformationen automatisch beweisen kann. Das Programm wird mit der funktionalen Programmiersprache Haskell implementiert. Diese Arbeit befasst sich mit dem Entwickeln einer graphischen Benutzerschnittstelle für dieses Programm. Diese Benutzerschnittstelle wird ebenfalls in Haskell implementiert.

Inhaltsverzeichnis

1 Motivation

Haskell ist eine der bekanntesten funktionalen Programmiersprachen an der weltweit gearbeitet wird. Funktionale Programmiersprachen haben den Vorteil, dass Speicheränderungen des Systems gar nicht, oder nur gekapselt, ermöglicht werden. Durch die getrennte Behandlung von Speicheränderungen sind so gut wie alle Interaktionen bekannt und kontrollierbar. Eine große Anzahl von unvorhergesehenen Aktionen am Speicher werden direkt ausgeschlossen. Der mit funktionalen Programmiersprachen erzeugte Code ist damit weniger anfällig für solche Fehler und zuverlässiger. Funktionale Programmiersprachen zeichnen sich durch kurzen und wartungsarmen Code aus. Durch die Aufteilung von Programmfragmenten in verschiedene Funktionen kann modular und parallel programmiert werden. In der Veröffentlichung [ERER2001] der Firma *Ericsson* wird gezeigt, dass die Produktivität durch die Nutzung einer funktionalen Programmiersprache mehr als verdoppelt wird. In diesem Fall wurde die Programmiersprache *Erlang* benutzt, die Haskell sehr ähnelt und von der Firma *Ericsson* selbst entwickelt wurde. Wenn Unternehmen eine Software entwickeln, wird immer mehr Zeit für die Wartung und Anpassung der Software an Kundenwünsche verwendet. Da funktionale Programmiersprachen genau hier einen Vorteil aufweisen, können diese in Zukunft an Bedeutung gewinnen. Wenn es um die Entwicklung von Software geht, wird immer mehr Fokus auf das Thema Benutzerfreundlichkeit gelegt. Das Implementieren von graphischen Benutzeroberflächen, als einfache Schnittstelle zwischen einem komplexen Programm und dem Benutzer, ist elementarer Bestandteil einer an Benutzerfreundlichkeit orientierten Programmierung. Auch in diesem Bereich können funktionale Programmiersprachen Mittel und Lösungen bereitstellen. Haskell als Open-Source Gemeinschaft erkennt die neuen Bedürfnisse die an Programmiersprachen gestellt werden und reagiert dementsprechend. Der Vorteil hierbei ist, dass Open-Source Gemeinschaften ihre Lösungen frei zugänglich halten und die Qualität dieser ständig durch ein großes Publikum verifiziert wird.

In dieser Arbeit wird mit der Programmiersprache Haskell und ihren Mitteln eine graphische Benutzerschnittstelle für ein Programm zum automatischen Korrektheitsnachweis von Programmtransformationen entwickelt. Bei Programmtransformationen handelt es sich um Transformationen, die ein Programm in ein syntaktisch verändertes Programm überführen. Eine solche Transformation kann dann als korrekt bezeichnet werden, wenn das ursprüngliche Programm und das resultierende Programm semantisch gleich sind. Das ist der Fall, wenn sich beide Programme in einem größeren Programm gleich verhalten. In einem Compiler wird Programmcode optimiert und übersetzt. Programmtransformationen und dessen Korrektheit sind elementare Werkzeuge eines Compilers. Durch die Automatisierung von Korrektheitsnachweisen für Programmtransformationen könnten Compiler in Zukunft die Korrektheit von Programmtransformationen zur Compilezeit nachweisen.

2 Haskell

Haskell ist eine *funktionale Programmiersprache*. Funktionale Programmiersprachen zeichnen sich dadurch aus, dass sie *deklarativ* sind. Deklarative Programmierung legt den Fokus auf das Beschreiben des Problems. Im Gegensatz zur imperativen Programmierung, die ein Problem mittels Folge von Befehlen löst. Die Beschreibung und Lösung eines Problems mit einer funktionalen Programmiersprache erfolgt durch Funktionen. Ein funktionales Programm kann als eine einzige Funktion betrachtet werden, die als Resultat einen Wert liefert.

„Haskell ist eine pure funktionale Programmiersprache, die Funktionen höherer Ordnung, nicht-strikte Auswertung, statisch polymorphe Typisierung, Benutzerdefinierte Datentypen, Pattern Matching, Listenerzeugung durch list comprehension, Modulsysteme, monadische Ein- und Ausgabesysteme und eine Reihe von elementaren Datentypen unterstützt und bereitstellt.“([HR2010], Kapitel 1)

Der oben zitierte Satz entspringt dem *Haskell 2010 language report*. In diesem wird die aktuelle Semantik und Syntax von Haskell beschrieben. Der bekannteste Haskell Compiler ist der *Glasgow Haskell Compiler* (*GHC*). Dieser unterstützt zusätzliche Erweiterungen und wurde für diese Ausarbeitung genutzt. In den nächsten Abschnitten werden die Besonderheiten von Haskell als funktionale Programmiersprache erläutert. Dabei wird auf die Themen eingegangen, die im Entwicklungsprozess der Benutzerschnittstelle eine fundamentale Rolle gespielt haben. Dabei gibt dieser Teil grundlegende Erklärungen aus [EFP] wieder.

2.1 Haskell als pure funktionale Programmiersprache

Als *pur* wird eine funktionale Programmiersprache bezeichnet, die keine *Seiteneffekte* erlaubt. Seiteneffekte entstehen unter anderem dort, wo Programmiersprachen Variablen und Wertzuweisungen erlauben und der Zustand des Speichers verändert werden kann. Bei nicht puren Programmiersprachen spielen Seiteneffekte, wie Wertzuweisungen, eine fundamentale Rolle. In Haskell können keine Werte direkt zugewiesen werden. Konstrukte wie globale Variablen und direkte Speicherzuweisungen können mit puren funktionalen Programmiersprachen, somit auch mit Haskell, nicht umgesetzt werden. In Haskell, wie auch in anderen puren funktionalen Programmiersprachen, gilt das Prinzip der *referentiellen Transparenz*:

„Das Ergebnis einer Anwendung einer Funktion auf Argumente, hängt ausschließlich von den Argumenten ab, oder umgekehrt: Die Anwendung einer gleichen Funktion auf gleiche Argumente liefert stets das gleiche Resultat. Variablen in funktionalen Programmiersprachen bezeichnen keine Speicherplätze, sondern stehen für (unveränderliche) Werte.“([EFP], S. 3)

Wie Haskell dennoch zustandsabhängige Operationen ermöglicht, wird in Abschnitt **2.6** erläutert.

2.2 Nicht-strikte Auswertung

Als *nicht-strikt* wird eine Auswertung bezeichnet, in der die Auswertung der Argumente erst dann erfolgt, wenn sie zur Bestimmung des Endwertes benötigt werden. Diese Art der Auswertung wird auch *lazy-evaluation* genannt. Haskell unterstützt Funktionen höherer Ordnung. Diese Eigenschaft erlaubt es Funktionen zu definieren die nicht nur Werte als Argumente akzeptieren, sondern auch Funktionen. Die nicht-strikte Auswertung ermöglicht es die Auswertung der Funktionen in den Argumenten so weit zu verzögern, bis sie wirklich zur Bestimmung des Endwertes benötigt werden.

2.3 Typsystem

Haskell ist eine *polymorphe*, *statisch* und *stark* typisierte Programmiersprache. Polymorphe Typsysteme erlauben *schematische* Typen und die Beschreibung von Typen mittels *Typvariablen*. Programmiersprachen mit einem statischen Typsystem zeichnen sich dadurch aus, dass während der Laufzeit keine Typberechnungen mehr vorgenommen werden. Vielmehr werden alle Typenüberprüfunegn zur *Compilezeit* vorgenommen und der endgültige Typ des Programms steht nach dem Compilieren fest. Als stark wird ein Typsystem bezeichnet, dass nur ordentlich getypte Programme zur Compilierung erlaubt. Alle Ausdrücke und Funktionen in einem Programm müssen einen korrekten Typ benutzen. Haskell erlaubt den Verzicht auf Angabe von Typen im Programmquellcode. Diese Eigenschaft wird als *Typinferenz* bezeichnet. Hier kann auf Angabe von Typen verzichtet werden, wenn der Typ selbst berechnet werden kann. Die Angabe des Typs kann dennoch ratsam sein. Durch die Angabe des Typs wird der Quellcode besonders leserfreundlich. Da der Compiler nur den allgemeinsten Typ berechnet, kann die Angabe des Typs dazu genutzt werden, um den Typ genauer zu definieren. Der Typ in Haskell wird mit `::` beschrieben.

Die Syntax in Haskell kann Anhand von Beispielen kurz erläutert werden:

```
beliebigeZahlen :: [Int]
beliebigeZahlen = [1,5,32,4,2,4]
```

In diesem Beispiel wird eine Funktion definiert, die zwei elementare Datentypen nutzt. Die Funktion hat kein Argument und gibt einen festen Wert zurück. Der Typ der Funktion ist in der ersten Zeile definiert. Auf die erste Zeile kann auf Grund der Typinferenz verzichtet werden. Der resultierende Typ hat die Form einer Liste. Die Liste selbst enthält hierbei Elemente des Datentyps `Int`. Beide Datentypen, sowohl der Typkonstruktor `Int`, als auch der Listenkonstruktor der leeren Liste `[]`, sind vordefinierte Datentypen und werden bereitgestellt.

```haskell
type MeinNeuerTyp = [Int]
type ListeVonListen = [MeinNeuerTyp]

eineNeueListe :: ListeVonListen
eineNeueListe = [[1,2,3],[2,31,32,321],[1]]

data Baum = Blatt Int | Knoten Int Baum Baum
```

Man kann auch Typsynonyme und Datentypen selbst definieren, wie man an den oben aufgeführten Beispielen erkennen kann. Das erste Typsynonym das definiert wird ist eine Zusammensetzung von zwei bekannten Datentypen. Das heißt man fasst Typen zusammen und verwendet sie unter einem neuen Typnamen. Dabei können selbstdefinierte Typsynonyme wiederum Bestandteil eines anderen Typsynonyms sein. Diese werden mit dem Ausdruck `type` erzeugt. Mit dem Ausdruck `data` werden Datentypen beschrieben, die wie `Baum` auch rekursiv sein können. Die Funktion `eineNeueListe` ist nun vom selbstdefinierten Typ `ListeVonListen`. Dieser Typname repräsentiert nichts anderes, als die Zusammensetzung der elementaren Datentypen und kann somit vom Compiler als diese erkannt werden.

2.3.1 Listen als Datentyp

Listen in Haskell sind ein Datentyp und werden wie folgt definiert:

```haskell
data [a] = [] | a : [a]
```

Listen können mit Hilfe von zwei verschiedenen Konstruktoren erstellt werden.

`[]` vom Typ `[a]`

`(:)` vom Typ `a → [a] → [a]`

Der erste Konstruktor hat keine Argumente. Der zweite hat zwei Argumente, das erste Element einer Liste, dass den Typ `a` hat und den Rest der Liste, mit Elementen des gleichen Typs. Beide Konstruktoren sind beim *Pattern Matching* nützlich, auf das später eingegangen wird. Die nächsten drei Beispiele sollen die Nutzung von beiden Konstruktoren veranschaulichen.

```haskell
"aaaa"
```

Das ist ein Sonderfall und gilt nur für den Typ `String`. In diesem Fall ist das eine Liste von Elementen des Datentyp `Char`. Ein `String` in Haskell ist somit eine Liste von Zeichen des Typs `Char` und wird wie folgt definiert:

```
type String = [Char]
```

Strings in Haskell werden von Anführungszeichen umklammert.

```
['a','a','a','a']
```

Das ist eine Liste mit Elementen des Datentyps `Char`, die durch den Konstruktor `[]` dargestellt wird. Dabei handelt es sich um eine vereinfachte Darstellung für die folgende Schreibweise. In Abschnitt **2.5.1** und **2.6** wird auf diese Thematik eingegangen.

```
('a':('a':('a':('a':[]))))
```

Das ist die gleiche Liste, die mit dem zweiten Konstruktor erstellt wird.

2.4 Module in Haskell

Mit Haskell ist es möglich Programmteile auszulagern und zu kapseln. Folgendes Beispiel soll dieses Prinzip erläutern:

```
module MeinModul(funktion1,funktion2) where

--Liste der Importe
import Data.List
import qualified MeinModul2 as M

{-
Auskommentieter Bereich
-}

--Funktionen und Typdefinitionen
funktion1 x = x+1
funktion2 x = x+2
funktion3 x = M.funktionAusM x
funktion4 x = Data.List.map (funktion1) x
```

Das *Modul* in Haskell besteht aus drei Elementen. Dem *Modulnamen*, der *Exportliste* und dem *Modulrumpf*. Der Modulname sollte per Konvention mit einem Großbuchstaben beginnen und muss den gleichen Namen haben wie die Datei des Quelltextes. Haskellquellcode hat überwiegend die Dateiendung *.hs*. Die Exportliste ist optional und enthält die Funktionen und Typen, die beim Import dieses Moduls zur Verfügung stehen sollen. In diesem Beispiel werden die Funktionen `funktion1` und `funktion2` exportiert. Das Weglassen der Exportliste führt dazu, dass alle Funktionen und Typdefinitionen exportiert werden. Der erste Teil des Modulrumpfes besteht aus Importen anderer Module. Ein Modul kann allgemein importiert werden oder mit dem Konstrukt `qualified`, mit dem man dem importierten Modul einen selbst definierten Namen zuordnen kann. In dem oben beschriebenen Beispiel werden zwei Funktionen genutzt, die importiert werden. Die Funktion `funktionAusM`, die aus dem Modul `MeinModul2` importiert wird und die Funktion `map`, die aus dem Modul `Data.List` importiert wird. Wenn die Funktionen eindeutig ihren Modulen zugeordnet werden können, dann muss das Modul nicht zum Funktionsnamen geschrieben werden. Das heißt, wenn die Funktion namens `map` in keinem anderen importierten Modul vorkommt und auch nicht im aktuellen Modulrumpf, dann reicht eine solche Beschreibung:

```
funktion4 x = map (funktion1) x
```

Es muss in jedem Fall sichergestellt werden, dass der Compiler erkennen kann, welche Funktion gemeint ist. Der zweite Teil besteht aus den Funktionsdefinitionen und Typdefinitionen. Mit zwei `-` lässt sich eine Zeile auskommentieren. Mit `{-` gefolgt von `-}` lassen sich komplette Bereiche auskommentieren.

2.5 Kontrollstrukturen in Haskell

2.5.1 Case Expressions und Pattern Matching

Ein unverzichtbares Konstrukt in Haskell sind *Case Expressions*, mit denen man Fälle eines Wertes unterschiedlich behandeln kann.

```
funktion :: Int -> Int -> Int
funktion x y = case x of
                 0 -> 0
                 1 -> 1
                 x -> x+y
```

In diesem Fall verhält sich `funktion` wie folgt. Wenn das erste Argument `x` den Wert `0` hat, dann gibt die Funktion den Wert `0` zurück. Analog `1`, wenn das erste Argument den

Wert `1` hat. Sollte das erste Argument weder den Wert `0`, noch `1` haben, dann gibt die Funktion den resultierenden Wert folgender Operation zurück: `x+y`.

Hierbei ist es wichtig zu erwähnen, dass man in Haskell sowohl mit Einrückungen, als auch mit Klammerungen arbeiten kann.

```
funktion :: Int -> Int -> Int
funktion x y = case x of {0 -> 0;1 -> 1;x -> x+y}
```

Dieses Prinzip wird beim *Pattern Matching* deutlich. Beim Pattern Matching handelt es sich um die Fähigkeit, eine Fallunterscheidung über den Konstruktor abzuarbeiten. Ein Beispiel hierfür soll die folgende Funktion darstellen:

```
nimmZwei :: [a] -> [a]
nimmZwei [] = []
nimmZwei (x:[]) = []
nimmZwei (x:y:xs) = xs

nimmZwei x = case x of
                [] -> []
                (x:[]) -> []
                (x:y:xs) -> xs
```

In diesem Fall erwartet die Funktion eine Liste, dann wird versucht die ersten zwei Elemente der Liste zu entfernen und die restliche Liste zurückzugeben. Die erste Definition fängt den Fall ab, dass die Liste leer ist. Die zweite Definition fängt den Fall ab, dass die Liste nur ein Element enthält. In beiden Fällen wird eine leere Liste zurückgegeben. Die letzte Definition erkennt eine Liste mit dem ersten Element `x`, dem zweiten Element `y` und gibt die restliche Liste `xs` zurück. Die Funktion kann natürlich auch mit `case` umgesetzt werden, wie die zweite Darstellung zeigt. Die Eigenschaft Quellcode für Menschen einfacher zu Strukturieren und leserfreundlicher zu machen wird als *syntaktischer Zucker* bezeichnet. Ein weiteres Beispiel dafür befindet sich im Abschnitt **2.6**.

2.5.2 As-Patterns

Mit sogenannten *As-Patterns* stellt Haskell ein weiteres Konstrukt zur Verfügung, mit dem der Code vereinfacht werden kann.

```
funktionAs [] = ([],[])
funktionAs y@(x:xs) = ([x],y)
```

Mit As-Patterns kann man Argumenten einer Funktion einen Variablennamen zuweisen und das Argument auf der rechten Seite der Funktion mit diesem Variablennamen nutzen. Die oben beschriebene Funktion soll dies veranschaulichen. Die Funktion `funktionAs` erwartet eine Liste und erzeugt aus dieser ein Tupel. Wenn die Liste leer ist, dann wird ein Tupel mit zwei leeren Listen zurückgegeben. In der zweiten Definition, wird die Liste als `y` definiert, um sie dann im Rumpf der Funktion als `y` zu verwenden. Diese Zuweisung erfolgt mit `@`. Sollte die Liste nicht leer sein, wird ein Tupel mit dem ersten Element der Liste und der ganzen Liste zurückgegeben.

2.5.3 Let Expressions

Mit *Let Expressions* kann man in Haskell einen lokalen Namensraum definieren in dem bestimmte Variablen einen definierten Wert haben. Ein Beispiel hierfür ist:

```
let x = 1
    y = 2
    in funktion x y
```

Im `let` Abschnitt können Variablen deklariert und zugewiesen werden, deren Gültigkeitsbereich den `let` und den `in` Abschnitt umfasst. Dabei können die Variablen selbst das Resultat einer weiteren Funktion sein:

```
let x = funktion1 2
    y = funktion2 2
    in funktion x y
```

Das heißt in diesem Fall, `x` ist der Wert den `funktion1` mit Argument `2` zurückgibt und `y` ist der Wert den `funktion2` mit Argument `2` zurückgibt. Wenn man das gleiche ohne Let Expression erzeugen möchten, würde das so aussehen:

```
funktion (funktion1 2) (funktion2 2)
```

2.5.4 If-then-else Expressions

if-then-else Kontrollstrukturen werden auch in Haskell unterstützt und sehen beispielsweise wie folgt aus:

```
funktion x = if (x<2) then funktion3 x
             else funktion4 x
```

In diesem Fall prüft die Struktur, ob `x<2` ist. Wenn das der Fall ist, wird der Wert, der aus der Ausführung von `funktion3 x` resultiert, zurückgegeben. Gilt `x>=2`, so wird der resultierende Wert von `funktion4 x` zurückgegeben. Dabei kann sowohl der `then`, als auch der `else` Bereich erneut aus Kontrollstrukturen bestehen. Wichtig hierbei ist es die Einrückungen zu beachten.

2.6 Monaden

„Monadisches Programmieren ist (aus Programmierersicht) eine bestimmte Strukturierungsmethode, um sequentiell ablaufende Programme in einer funktionalen Programmiersprache zu implementieren. Der Begriff Monade stammt aus dem Teilgebiet der Kategorientheorie der Mathematik. Ein Typkonstruktor ist eine Monade, wenn er etwas verpackt und bestimmte Operationen auf dem Datentyp zulässt, wobei die Operationen die sog. monadischen Gesetze erfüllen müssen.“([EFP], S. 175)

Mit dem Prinzip der *Monaden* in Haskell können Operationen, die den Zustand des Speichers verändern, realisiert werden. Dazu gehören Ein- und Ausgabeoperationen, sowie Lese- und Schreibprozesse. Die Typklasse *Monad* ist eine *Kontruktorklasse* und wie folgt definiert:

```
class Monad m where
    (>>=) :: m a -> (a -> m b) -> m b
    (>>) :: m a -> m b -> m b
    return :: a -> m a
    fail :: String -> m a
    m >> k = m >>= \_ -> k
    fail s = error s
```

Typklassen in Haskell dienen dazu, *ad hoc Polymorphismus* zu ermöglichen. Der Begriff wird wie folgt beschrieben:

„Ad hoc Polymorphismus tritt auf, wenn eine Funktion (bzw. Funktionsname) mehrfach für verschiedene Typen definiert ist, wobei sie sich die Implementierungen für verschiedene Typen völlig anders verhalten können. Ein Beispiel ist der Additionsoperator +, der z.B. für Integer- aber auch für Double-Werte implementiert ist und verwendet werden kann. Besser bekannt ist ad hoc Polymorphismus als Überladung.“([EFP], S.115)

Die oben definierten Operatoren werden dazu benötigt, um zwei *monadische Aktionen* sequentiell zu einer zu verbinden. Eine monadische Aktion ist hierbei ein Ausdruck vom Typ `m a`. Das erste Element kann mit Hilfe des Operators `>>=` von der zweiten Aktion verwendet werden, ohne die Monade selbst zu verlassen. Der Operator `>>`, auch *then* genannt, verhält

sich ähnlich zum Operator `>>=`, der auch als *bind* bezeichnet wird. Der Unterschied liegt darin, dass der *then* Operator das Ergebnis der ersten Aktion nicht verwendet. Beim `return` handelt es sich um einen Operator, der aus einem funktionalen Ausdruck `a` eine monadische Aktion vom Typ `m a` macht.

2.6.1 Maybe

Beim Datentyp *Maybe* handelt es sich um eine *Instanz* der Typklasse Monad. Mit diesem Datentyp lassen sich eine Folge von Berechnungen durchführen, dessen Ausgang definiert ist. Ein Gelingen einer solchen Folge resultiert als `Just a`. Ein Fehlschlagen resultiert als `Nothing`. Dieses Resultat steht für einen nicht definierten Ausgang. Die Beschreibung dieser Instanz sieht man im folgenden Ausschnitt:

```
instance Monad Maybe where
    return = Just
    fail = Nothing
    Nothing >>= f = Nothing
    (Just x) >>= f = f x
```

Die Definition des bind Operators ist fundamental, um Fehlschläge während der Folge als Fehlschlag des gesamten Ausdrucks zu behandeln.

2.6.2 Do-Notation

Die *do-Notation* ermöglicht es die schlecht lesbaren Ausdrücke, die mit monadischen Operatoren implementiert wurden, in eine für den Leser einfachere Struktur zu bringen. Die Definition der do-Notation sieht wie folgt aus und wird auch als syntaktischer Zucker bezeichnet:

```
do { x <- e ; s } = e >>= \x -> do { s }
do { e; s } = e >> do { s }
do { e } = e
do { let binds; s } = let binds in do { s }
```

Alle Ausdrücke die mit `do` umgesetzt wurden, können eins zu eins mit monadischen Operatoren umgesetzt werden.

2.6.3 IO Monade

Mit der *monadischen Ein- und Ausgabe* kann man in Haskell Lese- und Schreiboperationen vom funktionalen Teil des Programms trennen, um die Eigenschaft der *Seiteneffektfreiheit* nicht zu verletzten. Dazu wurde der Datentyp `IO a` implementiert. Dieser Datentyp stellt eine sogenannte *IO-Aktion* dar. Diese Aktion selbst führt keine Ein- und Ausgabeaktionen aus, sondern diese werden gekapselt und erst außerhalb der funktionalen Sprache und während der Laufzeit ausgeführt. Ähnlich zu `Maybe` ist auch `IO` eine Instanz der Klasse `Monad`. Man kann die Instanz `IO` in dieser Form veranschaulichen:

```
type IO a = Welt -> (a,Welt)
```

Die Welt ist hierbei beispielsweise der Speicherraum der adressiert werden kann. Dieser und andere IO-Ressourcen werden in Form einer Welt dargestellt. Haskell behandelt Ausdrücke des Typs `IO a` als festen Wert der nicht ausgewertet werden muss. Die Welt kann nicht als Argument von Haskell mitgegeben werden und wird erst während der Laufzeit als erlaubter Seiteneffekt ermittelt. Somit kann man sich die Welt als Eingabe vorstellen, dessen Zustand kontrolliert verändert wird. Die Rückgabe ist ein Wert vom Typ `a` und die veränderte Welt. Die folgenden Beispiele sollen die Funktionsweise von IO-Aktionen veranschaulichen und stammen aus ([EFP],S. 187 u. S. 188):

```
getChar :: IO Char
```

Mit dieser Funktion lässt sich ein Zeichen des Typs `Char` lesen. Als IO-Aktion betrachtet verändert diese Funktion zur Laufzeit den Zustand der Welt und liefert einen Wert des Typs `Char`.

```
putChar :: Char -> IO ()
```

Diese Funktion schreibt ein Zeichen, aber im Gegensatz zu der ersten Funktion liefert diese nur die veränderte Welt zurück. Das heißt die Funktion bekommt einen Ausdruck des Typs `Char`, eine Welt und liefert als Rückgabe nur die veränderte Welt zurück. Das Verhalten der beiden Funktionen ist einfach zu erläutern. Wenn ein Zeichen gelesen werden soll, dann wird ein Zeichen als Rückgabe erwartet. Soll ein Zeichen geschrieben werden, erwartet man kein Zeichen als Rückgabe. Das Besondere an der `IO` Monade ist, dass diese in Form eines Arguments mitgeliefert wird. Diese muss beim Verarbeiten der Ausdrücke berücksichtigt und mittels monadische Operatoren verwaltet werden. Zur Visualisierung der monadischen Operatoren, die bereits erwähnt wurden, hilft dieses Beispiel:

```
echo :: IO ()
echo = getChar >>= putChar
```

Mit Verwendung der do-Notation:

```
echo :: IO()
echo = do
    c <- getChar
    putChar c
```

Beide Umsetzungen führen die gleiche Aktion aus. Die Funktion `echo` ist eine IO Aktion. Es wird ein Zeichen gelesen und direkt zurückgeschrieben. Da diese Aktionen gekapselt stattfinden und innerhalb einer `IO` Monade ablaufen, muss diese mittels monadische Operatoren verarbeitet werden. Da `getChar` eine Welt erwartet, und einen Wert und eine Welt zurückliefert, während `putChar` einen Wert und eine Welt erwartet, und eine Welt zurückliefert, lassen sich diese beiden Operationen mittels bind miteinander verbinden. Dabei wird bei der do-Notation der Umweg über die Variable c gegangen, um einen Wert ohne Monade an die Funktion `putChar` zu übergeben. Dieser Umweg ist beim direkten bind nicht nötig, da dieser die Monade bereits beachtet. Dieses Bild veranschaulicht beide Funktionen und ihre Ein- und Rückgabewerte ([EFP], S. 188):

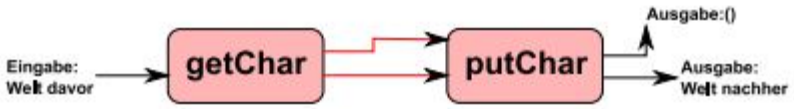


Abb. 2.6.3: Veranschaulichung von monadischen Operatoren am Beispiel der Funktion echo

3 Automatischer Korrektheitsnachweis von Programmtransformationen

In diesem Abschnitt wird das Programm zum automatischen Korrektheitsnachweis von Programmtransformationen und der theoretische Rahmen, auf das sich die graphische Benutzerschnittstelle stützt, näher erläutert. Dafür ist es notwendig auf die Grundlage von Programmtransformationen und dessen Korrektheit einzugehen. Dabei hält sich dieser Abschnitt an Definitionen und Beschreibungen aus [AKVP]. Das Programm kann dem Projekt [DFGP] entnommen werden.

3.1 Korrektheit von Programmtransformationen

Programmiersprachen können durch *Kernsprachen* beschrieben werden. So kann auch Haskell als funktionale Programmiersprache durch eine solche Kernsprache beschrieben werden. Diese Kernsprachen werden in der Regel in Form eines *Programmkalküls* beschrieben. Ein solcher Kalkül, der Haskell beschreibt, ist der *Lambda-Kalkül LR* aus [SNSA2008]. In Programmkalkülen wird die Auswertung von Ausdrücken durch eine Semantik definiert. Diese Semantik ist *operational*. Sie besteht aus *Reduktionsregeln* und einer *Normalordnungsreduktion*. Die Normalordnungsreduktion ist eine standardisierte Form der Auswertung. Bei *Programmtransformationen* handelt es sich um *Relationen*, die ein Programm zu einem anderen in Beziehung setzen. Die Korrektheit einer solchen Transformation kann mittels *kontextueller Gleichheit* geprüft werden. Das Prinzip der kontextuellen Gleichheit basiert auf der operationalen Semantik.

Zusammenfassung *kontextuelle Gleichheit von Programmen***:**

„Zwei Programme sind kontextuell Gleich, wenn diese in einem größeren Programmumfeld (auch Kontext genannt) ein nicht unterscheidbares Verhalten zeigen.“([AKVP], S. 2)

Mit der obigen Beschreibung kann auch die Korrektheit einer Programmtransformation beschrieben werden:

Zusammenfassung *Korrektheit einer Programmtransformation***:**

Eine Programmtransformation ist korrekt, wenn das Programm vor der Transformation *T* und das Programm nach der Transformation *T* kontextuell gleich sind.

Ein fundamentales Problem der kontextuellen Gleichheit liegt in der Anzahl der *Kontexte*. Als Kontext wird das größere Programmumfeld bezeichnet, in das man die Programme einsetzt, um sie auf kontextuelle Gleichheit zu prüfen. Die Klasse der Kontexte kann hierbei durch ein *Kontextlemma* reduziert werden. Dieses Kontextlemma erlaubt das Testen der Terminierung in

einer kleineren Klasse von Kontexten. Die Existenz einer endlichen Folge von Reduktionen kann dann *induktiv* mit Hilfe *vollständiger Mengen von Diagrammen* gezeigt werden. Diese Methode zum Prüfen der Korrektheit von Programmtransformationen wird *diagram-based method* genannt, also eine auf Diagrammen basierende Methode und wird im nachfolgenden Abschnitt genauer erläutert.

3.2 Diagramm basierte Methode für den Korrektheitsbeweis von Programmtransformationen

Bevor auf die Entstehung der Diagramme eingegangen wird, wird kurz die Symbolik erläutert:

Die *kontextuelle Gleichheit* von zwei *Programmen P1,P2* wird wie folgt dargestellt:

$$P1 \sim_c P2$$

Die *Transformation T* eines *Programms P1* in ein zweites *Programm P2* wird auf diese Weise notiert:

$$P1 \xrightarrow{T} P2$$

Die nächste Symbolik bedeutet, dass ein *Programm P* in Kontext *C* betrachtet wird:

$$C[P]$$

Für den Korrektheitsbeweis von Programmtransformationen werden zwei unterschiedliche Arten von Diagrammen benötigt. Bei der ersten Variante wird folgender Fall betrachtet:

1. Wenn *C[P1]* terminiert und die Terminierung von *C[P2]* gezeigt werden soll, dann wird dies als Gabel dargestellt:

$$t \xleftarrow{no,*} C[P_1] \xrightarrow{T} C[P_2].$$

   Das Symbol *t* steht für eine Normalform. Über eine Folge von Normalordnungsreduktionen

$$\xleftarrow{no,*}$$

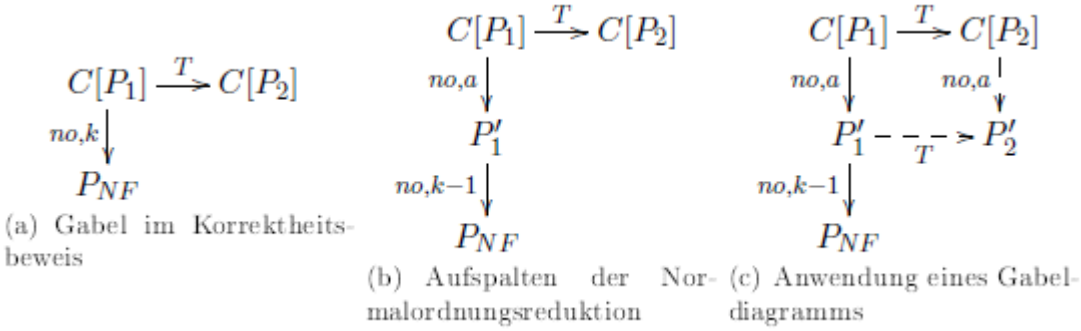
   wird diese von *C[P1]* aus erreicht. Außerdem transformiert *C[P1]* mittels *t* zu *C[P2]*. Gesucht wird dabei eine endliche Folge von Normalordnungsreduktionen von *C[P2]* zu einer Normalform *t'*. Diese gesuchte Folge wird mit Hilfe von *Gabeldiagrammen* konstruiert.

2. Wenn *C[P2]* terminiert und die Terminierung von *C[P1]* gezeigt werden soll, ergibt dies folgenden Fall:

$$C[P_1] \xrightarrow{T} C[P_2] \xrightarrow{no,*} t'.$$

Bei diesem Fall werden während der Induktion *Vertauschungsdiagramme* verwendet.

Die Anwendung eines Gabeldiagramms im Fall **1.** wird in folgender Abbildung veranschaulicht:



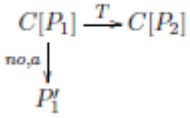
(a) Gabel im Korrektheitsbeweis

(b) Aufspalten der Normalordnungsreduktion

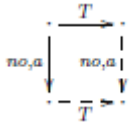
(c) Anwendung eines Gabeldiagramms

Abb. 3.2: Induktive Nutzung eines Gabeldiagramms für einen Korrektheitsnachweis einer Programmtransformation

In dieser Abbildung wird deutlich, wie ein Gabeldiagramm induktiv für den Korrektheitsnachweis einer Transformation *T* verwendet wird. Teil **(a)** der Abbildung zeigt eine Situation in der eine Folge von Normalordnungsreduktionen der Länge *k* zu einer Normalform P_{NF} führt. Es wird dabei von *C[P1]* ausgegangen. In Teil **(b)** der Abbildung entsteht die Gabel



in dem die Folge von Normalordnungsreduktionen aus Teil **(a)** aufgeteilt wird. In Teil **(c)** wird dann ein Diagramm der Form



genutzt. Dieses Diagramm wird benötigt um zu gewährleisten, dass eine so entstandene neue Gabel wieder mit Hilfe eines Diagramms induktiv bearbeitet werden kann. Mit einer solchen

Methode kann die Existenz einer endlichen Folge von Normalordnungsreduktionen gezeigt werden. Dabei wird von einer endlichen Folge für *C[P1]* ausgegangen, um dies für *C[P2]* nachzuweisen. Die Anzahl an zu prüfenden Kontexten kann hier reduziert werden, in dem ein passendes Kontextlemma genutzt wird, dass beispielsweise im *Lambda-Kalkül LR* beschrieben wird. Somit muss nicht die allgemeine Klasse *C* betrachtet werden, sondern es kann auf eine reduzierte Klasse zurückgegriffen werden.

3.2.1 Automatisierung der Induktion in Korrektheitsnachweisen

Die Existenz eines vollständigen Satzes von den oben erwähnten Diagrammen reicht für den Korrektheitsnachweis einer Transformation nicht aus. Für die Vervollständigung des Korrektheitsnachweises muss sichergestellt werden, dass aus Normalordnungsreduktionen, sowohl fehlschlagende als auch erfolgreiche, erfolgreiche und fehlschlagende Reduktionen konstruiert werden können. Die Konstruktion solcher Reduktionen wird mit Hilfe eines Induktionsbeweises realisiert. Ein solcher Induktionsbeweis hat folgende Struktur:

Die Induktionsbasis macht sich die Eigenschaft der Programmtransformationen zu nutze, dass diese *Endwerte der Standartreduktion*, also *Werte* oder *nicht-terminierende Programme* erhält. Diese Induktionsbasis, die sich auf die oben erwähnte Eigenschaft stützt, lässt sich einfach Nachweisen. Anhand der syntaktischen Struktur der Transformationsregel lässt sich bereits erkennen, ob diese Endwerte erhält. Eine solche Aussage für eine Transformation *T* lässt sich wie folgt beschreiben:

Zusammenfassung *Aussage für Transformation T in der Induktionsbasis*:

Wenn $s \xrightarrow{T} t$, dann ist *s* in Normalform genau dann wenn *t* in Normalform ist.

Bei der im Kalkül LR erwähnten Normalform handelt es sich um die *Weak Head Normal Form*. In dieser Arbeit wird nicht auf Kernsprachen, Grammatiken und deren Eigenschaften eingegangen. Grundlagen zu diesem Thema können in ([EFP], Kapitel 2.2) nachgelesen werden.

Der Induktionsschritt besteht im allgemeinen aus der Verwendung der Gabel- und Vertauschungsdiagramme. Diese werden nacheinander genutzt, um aus den gegebenen Reduktionsfolgen neue zu konstruieren. Für manche Fälle lässt sich die Induktion automatisieren. Die einfachsten Fälle liegen vor, wenn die Transformation *T* die *Aussage für Transformation T in der Induktionsbasis* erfüllt.

Die vollständige Menge aus Diagrammen zu *T* besteht aus Diagrammen, die als *Termersetzungsregeln* verstanden werden. Diese Termersetzungsregeln ersetzen eine Folge von Reduktionen durch eine andere. Zusammengefasst bildet diese Menge von Diagrammen, beziehungsweise Termersetzungsregeln, ein *Termersetzungssystem* das auch als *TRS* abgekürzt werden kann. Diese Abkürzung steht für *Term Reduction System* und wird in [TRAT1998] erläutert. Wenn dieses System für eine Transformation *T* terminiert, dann kann der induktive Korrektheitsnachweis automatisiert werden. Dies kann sichergestellt werden, weil die Anwendung von Termersetzungsregel keine unendliche Folge von Reduktionen erzeugt. Ein vollständiger Satz solcher Termersetzungsregeln kann beispielsweise wie folgt aussehen:

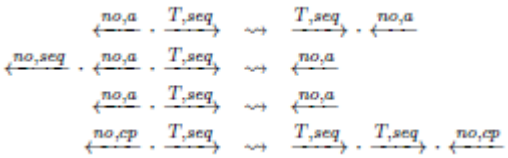
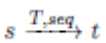
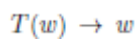


Abb. 3.2.1: Vollständiger Satz von Termersetzungsregeln / Diagrammen für die Transformation seq im LR-Kalkül

Die *Aussage für Transformation T in der Induktionsbasis* gilt für die Transformation *seq*:



Im Termreduktionssystem wird dies wie folgt kodiert:



Die Konstante *w* steht für einen Term in *WHNF*. Ein Werkzeug, dass für TRS die Terminierung nachweisen kann, wird in [ATPD2006] beschrieben und wird hier [APR] bereitgestellt. Dieses Werkzeug namens *AProVE* kann Diagramme wie in **Abb. 3.2.1** wie folgt darstellen:

```
(VAR x a)
(RULES
  T(seq,no(a,x)) -> no(a,T(seq,x))
  T(seq,no(a,no(seq,x))) -> no(a,x)
  T(seq,no(a,x)) -> no(a,x)
  T(seq,no(cp,x)) -> no(cp,T(seq,T(seq,x)))
  T(seq,w) -> w
)
```

Abb. 3.2.2: Darstellung von Abb. 3.2.1 als TRS

Man kann an dieser Darstellung erkennen, dass die Namen der Reduktionen auch kodiert wurden. Zudem wurden die Diagramme so kodiert, dass die Diagramme auf die Reduktionsfolge

$$s'_{WHNF} \xleftarrow{no,*} s \xrightarrow{T} t$$

angewendet werden können. Der Induktionsschritt kann mit *AProVE* automatisiert werden, solange sich in den Folgen von Reduktionen keine Termersetzungsregeln mit beliebiger Länge befinden. Das ganze Verfahren funktioniert nur, wenn die Eigenschaft der Transformation für die Induktionsbasis gezeigt werden kann. Dies muss separat erfolgen.

3.3 Berechnung von Diagrammen

Aus den in Abschnitt **3.2** vorgestellten Ideen ist ein Programm entstanden, dass in der Lage ist automatische Korrektheitsnachweise durchzuführen. Das Programm dient insbesondere zur Berechnung von Diagrammen. In diesem Abschnitt wird auf die Besonderheiten dieses Programms eingegangen, die für den Entwicklungsprozess der Benutzerschnittstelle notwendig sind.

3.3.1 Programmkalkül

Ein Programmkalkül besteht aus SORTS, LABELS, CLASSES, CONTEXTS , REDS und NOREDS. Die Kontexte des Programmkalküls werden im CONTEXTS Abschnitt definiert. Die Klassen des Kalküls im CLASSES Bereich. Unter SORTS wird die Grammatik des Kalküls beschrieben. Bei REDS handelt es sich um die Transformationen des Programmkalküls. Bei NOREDS um die Standartreduktionen. Mit Hilfe der LABELS können mehrere Transformationen unter einem Namen zusammengefasst werden. Der folgende Ausschnitt aus *LR.calc* soll dies veranschaulichen:

```
(LABELS
-- case    = [case-c,    case-e,    case-in],
-- choice  = [choice-l,  choice-r],
-- cp      = [cp-e,      cp-in],
-- llet    = [llet-e,    llet-in],
-- seq     = [seq-c,     seq-e,     seq-in],
-- ll      = [lapp,      lcase,     lseq],
no-case    = [no-case-c   ,no-case-e  ,no-case-in],
no-choice  = [no-choice-l,no-choice-r],
no-cp      = [no-cp-e     ,no-cp-in],
no-llet    = [no-llet-e   ,no-llet-in],
no-seq     = [no-seq-c    ,no-seq-e   ,no-seq-in],
no-ll      = [no-lapp     ,no-lcase   ,no-lseq]
)
```

Abb. 3.3.1: LABELS aus LR.calc

Ein Programmkalkül wird durch einen eigenen Datentyp namens `Calculus` dargestellt und wird wie folgt in Haskell definiert:

```
data Calculus = Calc { sortDecs :: [(String, ManySorted SSort)],
                       cclasses :: [ConClass],
                       cDecs    :: [ContextDef],
                       cLabels  :: [(String,[String])],
                       noReds   :: [Rule],
                       reds     :: [Rule]
                     }
                deriving(Show,Eq)
```

Eine wichtige Funktion, die aus einem Programmkalkül in Form eines Textes ein Element des Typs `Calculus` macht ist folgende Funktion:

```
loadCalc :: FilePath -> StateT Calculus StateFM Calculus
```

Diese Funktion erwartet einen Ausdruck vom Typ `FilePath` und liefert ein Objekt des Typs `StateT Calculus StateFM Calculus`. Der Programmkalkül befindet sich innerhalb von Monaden und kann mit der Funktion

```
execIt :: StateT Calculus StateFM a -> a
```

extrahiert werden. Der Typ `FilePath` steht für einen Pfad zu einer Datei. Die Programmtransformationen sind im Datentyp `Calculus` als Liste von Elementen des Datentyps `Rule` dargestellt und werden `reds` genannt. Der Datentyp `Rule` wird wie folgt beschrieben:

```
data Rule = Rule { ruleName :: Diag.Item,
                   ruleLhs  :: STerm,
                   ruleRhs  :: STerm}
          deriving(Show,Eq)
```

Der Name einer Transformation ist ein Element des Datentyps `Item`. Ein Item wird wie folgt definiert:

```
data Item     = SR Label
              | Trans Label
              | Answer
              | SingleName String
              | DoubleName String IntSym
              | Exception String
  deriving(Show,Eq,Ord)
```

In diesem Fall werden nur Elemente betrachtet, die für eine Transformation stehen. Diese werden mit dem Konstruktor `Trans Label` bezeichnet. Ein Element des Datentyps `Label` wird wie folgt definiert:

```
data Label     = Rule String
               | Tau
               | Var String
               | Plus Label
               | Star Label
  deriving(Show,Eq,Ord)
```

In diesem Fall wird sich auf den ersten Fall `Rule String` konzentriert. Hierbei ist es wichtig zu erwähnen, dass der Bezeichner `Rule` in diesem Fall nichts mit den vorher kennengelernten Datentyp `Rule` zu tun hat, da sich beide in verschiedenen Modulen befinden. Zusammenfassend kann erkannt werden, dass eine Transformation, beziehungsweise auch eine Standardreduktion, durch folgenden Typ repräsentiert wird:

Transformation:

`Trans (Rule String)`

Standardreduktion:

`SR (Rule String)`

Zudem werden die Transformationen und Standartreduktionen eines Kalküls durch eine Liste von Elementen des Typs `Rule` dargestellt.

3.3.2 Diagramme/Termersetzungsregeln

Diagramme werden durch den Datentyp `Diagram` repräsentiert:

```
data Diagram = [Item] :~~> [Item]
  deriving(Show,Eq,Ord)
```

Bei dem Datentyp `Item` handelt es sich um den gleichen Datentyp der im Abschnitt **Programmkalkül** erwähnt wird. Eine Funktion, die für ein Programmkalkül und eine Transformation eine Menge an Diagrammen erzeugt, wird mit

```
diagramsFor
  :: FilePath
    -> String
    -> StateT Calculus StateFM [Set.Set Diag.Diagram]
```

bereitgestellt. Der resultierende Typ muss ebenfalls von den Monaden befreit werden. Dafür kann man erneut die Funktion `execIt` verwenden. Die Anzahl der Diagramme kann mit Hilfe einer weiteren Funktion reduziert werden.

```
generalizedDiagramsFor
  :: FilePath
    -> String
    -> StateT Calculus StateFM [Set.Set Diag.Diagram]
```

Die Diagramme werden in diesem Fall anhand einer festgelegten Liste zusammengefasst.

3.3.3 ITRS und Textausgabe

Mit der Funktion

```
toITRS :: [Set.Set Diag.Diagram] -> String
```

können die Diagramme in Form eines *TRS* ausgegeben werden. Diese werden von Programmen wie *AProVE* unterstützt. Mit der Funktion

```
printDiagrams :: [Set.Set Diag.Diagram] -> String
```

können die Diagramme in Form einer Darstellung, die **Abb. 3.2.1** ähnelt, repräsentiert werden. Diese Form der Ausgabe wird in dieser Arbeit *Textausgabe* genannt.

4. Implementierung

In diesem Teil der Arbeit wird der Entwicklungsprozess der graphischen Benutzerschnittstelle erläutert. Das Programm aus Abschnitt **3.3** wird im folgenden Teil PT-Programm genannt. PT steht hier für Programmtransformation. Ausschnitte aus dem Programmcode dienen nur der Veranschaulichung und sind teilweise unvollständig. Der vollständige Code kann dem Anhang entnommen werden.

4.1 Zielsetzung

Bevor mit der Entwicklung einer graphischen Benutzerschnittstelle begonnen werden kann, muss das Aussehen und die Funktionalität der Benutzerschnittstelle klar definiert werden.

4.1.1 Funktionalität

Die graphische Benutzerschnittstelle muss folgende Funktionalitäten bereitstellen:

Funktionen eines Texteditors:

- Eine neue Textdatei erstellen
- Eine Textdatei öffnen
- Textdateien editieren (Text verändern, Text Ausschneiden/Kopieren/Einfügen)
- Textdateien speichern (Speichern unter, Speichern)

Funktionen zum Anzeigen von Diagrammgraphiken (.png):

- Eine Bilddatei öffnen
- Bilddateien speichern (Speichern unter)

Schnittstelle zum PT-Programm:

- Aus einem Programmkalkül im Form einer Textdatei (.txt oder .calc) sollen die Programmtransformationen extrahiert werden. Dabei soll das Programmkalkül in den Typ `Calculs` überführt werden. Die extrahierten Programmtransformationen sollen für den Benutzer einsehbar und auswählbar sein, um
- aus einer ausgewählten Programmtransformation und dem dazugehörigen Programmkalkül die dazugehörigen Termersetzungsregeln, beziehungsweise Diagramme, zu erzeugen. Diese sollen in drei verschiedenen Darstellungsformen erzeugt werden. Die erste Darstellungsform ist die sogenannte Textausgabe. Die Diagramme werden in Form eines Strings in eine Textdatei (.txt) geschrieben. Die zweite ist in Form eines TRS und wird ebenfalls als String in eine Textdatei (.txt) geschrieben. Bei beiden Ausgabeformen werden die reduzierte und die vollständige Anzahl der Diagramme in jeweils zwei Textdateien ausgegeben. Die letzte Darstellungsform sind Diagrammgraphiken. Hierbei wird nur die reduzierte Anzahl der Diagramme gezeichnet und als Bilddatei (.png) gespeichert.
- Der Nutzer soll vor dem Berechnen der Diagramme die Möglichkeit haben, aus den drei Darstellungsformen zu wählen. Dabei gilt die Regel, dass mindestens eine Darstellungsform ausgewählt werden muss.
- Die Ausgabe des Programms wird in einem gesonderten Ordner abgelegt. Für jede Ausgabeform existiert ein Unterordner in dem die Dateien abgelegt werden. Die Ausgabedateien und die Ordner unterliegen einer klar definierten Namensgebung.
- Sollte der Nutzer die Darstellungsform der Diagramme in Form einer Bilddatei (.png) gewählt haben, dann soll die generierte Bilddatei automatisch geöffnet und dem

Nutzer mit Hilfe der Bildbetrachtungsfunktionen präsentiert werden.

4.1.2 Graphisches Design der Benutzerschnittstelle

Beim Entwurf des graphischen Teils der Benutzerschnittstelle wird primär Wert auf die Erfüllung der geforderten Funktionalitäten gelegt. Diese bilden das Fundament der graphischen Elemente. Aus jeder geforderten Funktionalität folgen demnach graphische Elemente, die diese repräsentieren. Die Darstellungsformen werden dabei durch zwei Kriterien eingeschränkt. Die erste Einschränkung bildet das Thema Benutzerfreundlichkeit. Dem Nutzer soll die Nutzung der graphischen Benutzerschnittstelle so einfach wie möglich gestaltet werden. Dazu gehört es Darstellungsformen zu verwenden, die dem Nutzer schon bekannt sind. Eine Darstellungsform ist dem Nutzer mit großer Wahrscheinlichkeit bekannt, wenn diese weit verbreitet ist. Das zweite Einschränkungskriterium bilden die verwendeten Werkzeuge zur Softwareentwicklung, bei denen nur Darstellungsformen verwendet werden können, die von diesen Werkzeugen unterstützt werden. Mit Hilfe dieser Kriterien lassen sich die geeigneten Darstellungsformen für den graphischen Teil der Benutzerschnittstelle finden. Die folgenden Abbildungen werden die grundlegenden Darstellungsformen, die zu den geforderten Funktionalitäten passen, veranschaulichen.Die gängigsten Texteditoren zeichnen sich durch zwei Elemente aus. Ein Menü, in dem die grundlegenden Funktionen ausgewählt werden können und einer Fläche, die den Text einer Datei darstellt und auf der man den Text bearbeiten kann. Das Hauptfenster in Abbildung **4.1.1** soll dies veranschaulichen:

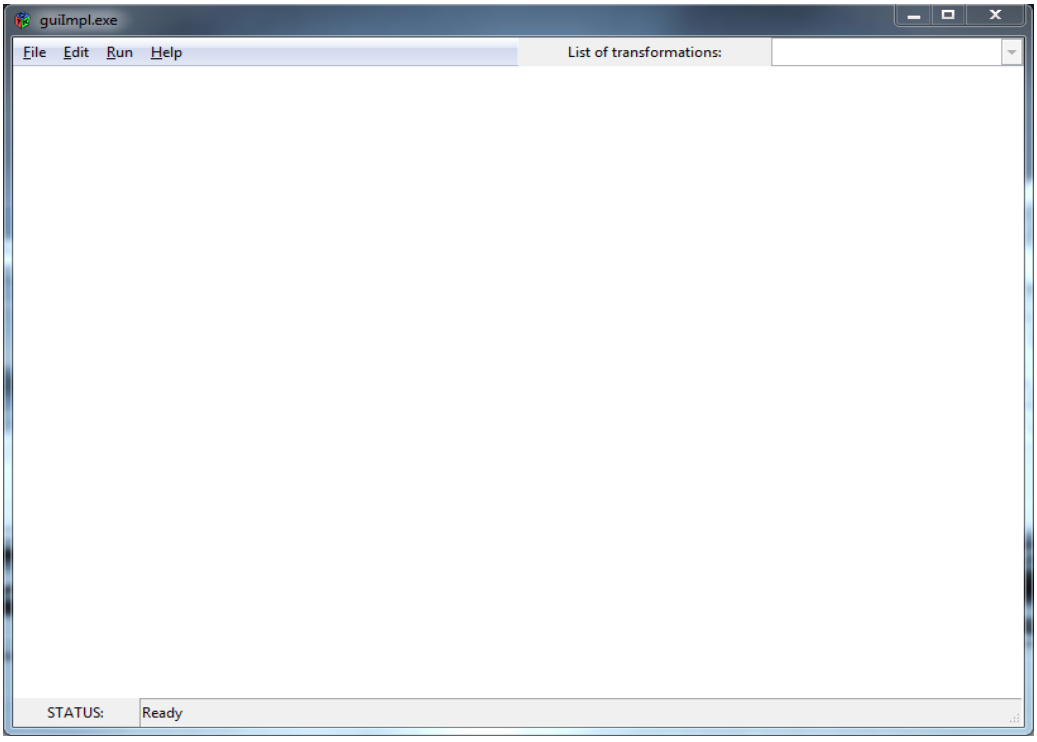


Abb. 4.1.1: Grunddesign des Hauptfensters

Jede Überschrift des Menüs kann eine freie Anzahl von Untermenüs haben, in denen die Funktionalitäten aufgerufen werden können. Diese Struktur ermöglicht eine nach Thematik geordnete Darstellung der Funktionalitäten. Das Öffnen und Speichern von Dateien wird mit Hilfe eines *File Manager* ermöglicht. Auch hier kann man programmunabhängige Merkmale erkennen, die in der Regel bei jedem File Manager zu finden sind. Ein grundlegendes Element des File Managers ist die Auswahl eines Verzeichnisses des aktuellen Systems. Das zweite Element ist die Eingabe eines Dateinamen mit Dateiendung. Dieses ist nur beim Speichern zwingend notwendig. Abbildung **4.1.2** zeigt einen solchen File Manager:

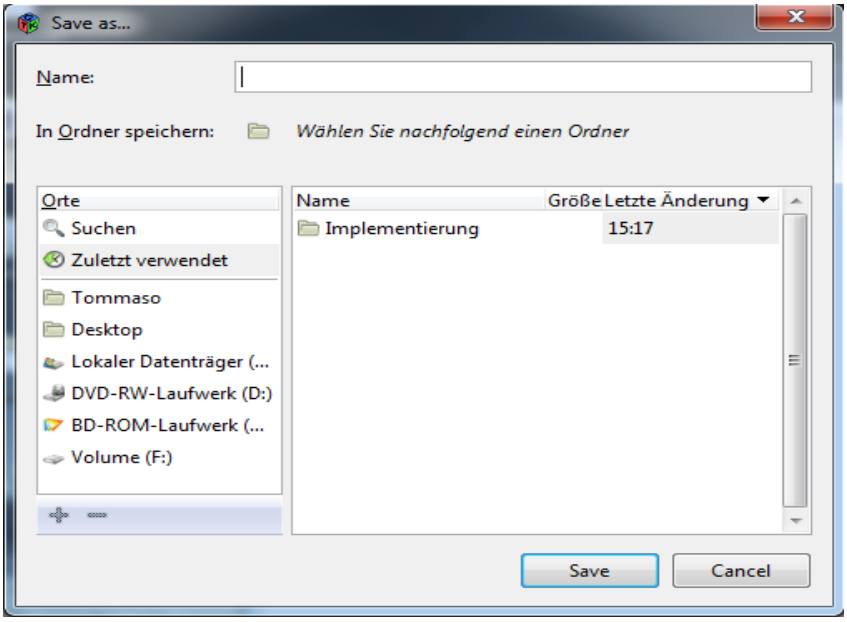


Abb. 4.1.2: File Manager

Das Öffnen einer Bilddatei erfolgt in einem separaten Fenster. Das Fenster präsentiert das Bild und stellt eine eigene Speicherfunktion für Bilder zur Verfügung. Die Unterscheidung ist nötig, da Bilddateien nicht nur in Form eines Textes repräsentiert werden sollen, sondern auch als Graphik. Siehe Abbildung **4.1.3**.

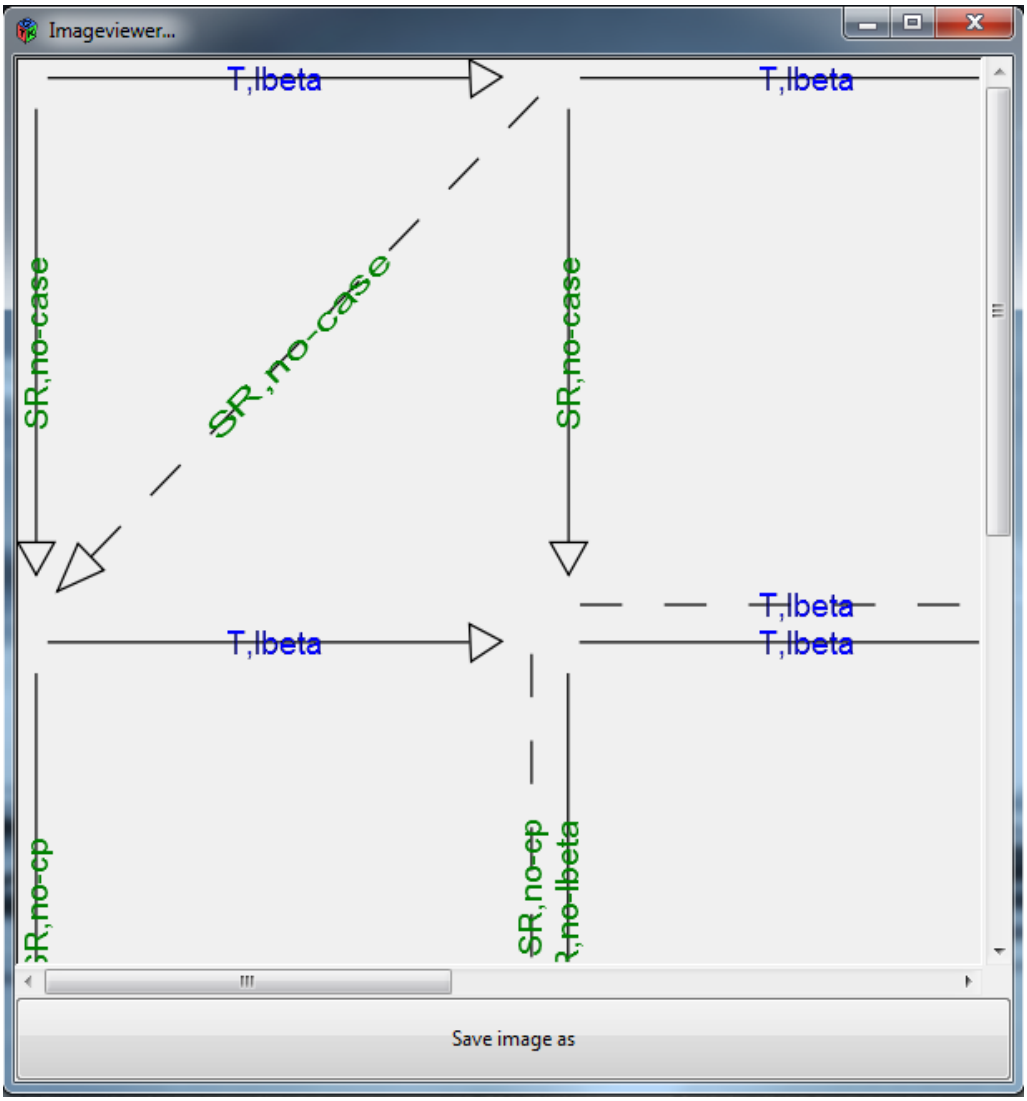


Abb. 4.1.3: Bildbetrachtungsfenster

Die Funktionen zum Auslesen der Programmtransformationen aus einem Programmkalkül und zum Erstellen der dazugehörigen Ausgabedateien werden unter einem der Menüpunkte zur Verfügung gestellt. Die auswählbaren Programmtransformationen werden rechts vom Menü präsentiert, wie Abbildung **4.1.4** veranschaulicht. Die Auswahl der Ausgabeform der Diagramme wird in einem weiteren Fenster abgefragt, dass in Abbildung **4.1.5** zu sehen ist.

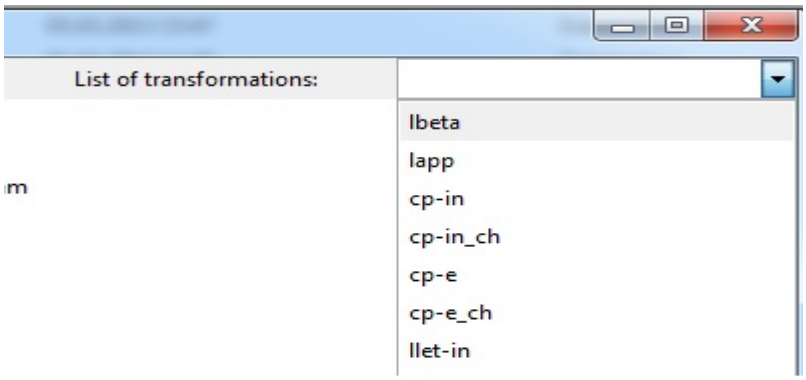


Abb. 4.1.4: Liste der Programmtransformationen

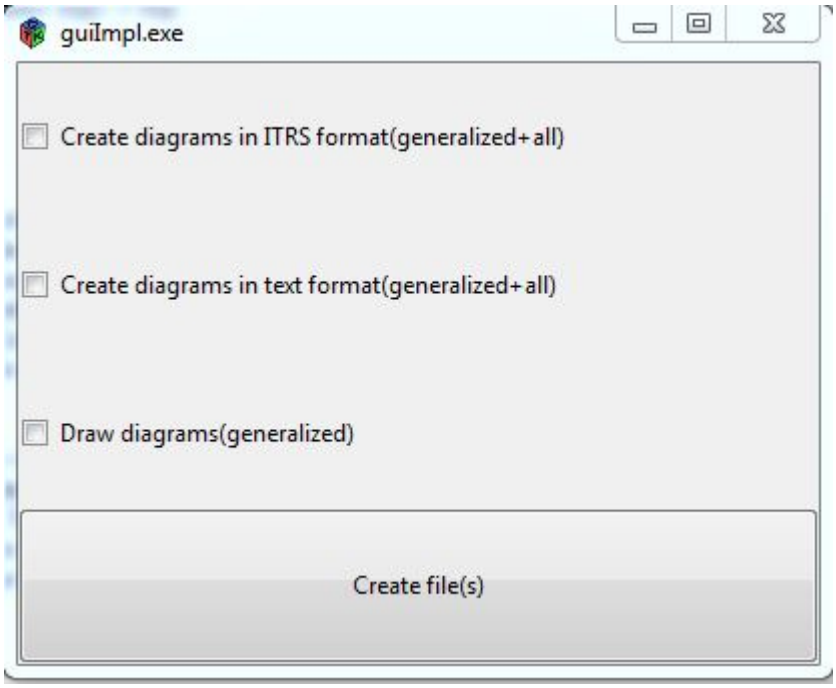


Abb. 4.1.5: Auswahl der Ausgabearten

Die letzten graphischen Elemente sollen dem Nutzer dienen und sind nicht direkt aus den geforderten Funktionalitäten ableitbar. Hierzu gehört die im unteren Teil des Hauptfensters liegende Statusanzeige, die den Nutzer über den aktuellen Programmstatus informieren soll und das *Hilfe-Fenster*, dass Informationen zum Programm bereitstellt. Die Statusanzeige ist in Abbildung **4.1.6** zu sehen.

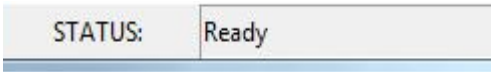


Abb. 4.1.6: Statusanzeige

4.2 Strukturüberblick

Die graphische Benutzerschnittstelle besteht aus drei Teilen. Der ersten Teil ist das Modul *PTI*, dass die Hauptschnittstelle zwischen der graphischen Benutzeroberfläche und dem PT-Programm darstellt. PT steht erneut für Programmtransformation und das I steht für *Interface*, also Schnittstelle. Der zweite und größte Teil ist die graphische Oberfläche und dessen Implementierung. Der dritte und letzte Teil ist das Modul *DrawEngine*, dass die Funktionen zum Zeichnen der Diagramme aus Abschnitt **3.3** und zum Erzeugen von Bilddateien bereitstellt. Um die Funktionen des Texteditors zu gewährleisten, muss das Programm jederzeit wissen welche Datei gerade bearbeitet wird. Da Haskell keine direkten Speicherzuweisungen erlaubt, wurde dieses Problem mit Hilfe der sich im Ordner co*nfig* befindenden Textdatei *Currentfilepath.txt* gelöst. In dieser wird der aktuelle Dateipfad abgespeichert. Die Datei *unsafed.txt* dient dazu, den Inhalt einer neu erstellten Textdatei solange zu speichern, bis der Nutzer aufgefordert wird, einen neuen Speicherplatz für den Inhalt zu wählen. Der Ordner *DiagramFiles* enthält Unterordner, in denen die jeweiligen Ausgabedateien des Nutzer beim Erstellen der Diagramme abgelegt werden. *DiagramImages* enthält die Bilddateien, *ITRS* die Ausgabedateien in TRS Form und *Textformat* die Dateien in Form einer Textausgabe.

4.3 Implementierung der Schnittstelle

Um den Anforderungen an die graphische Benutzerschnittstelle gerecht zu werden, werden Funktionen aus dem PT-Programm benötigt. In Abschnitt **3.3** wurden bereits die elementaren Funktionen erwähnt, die hierfür benötigt werden. Das PT-Programm ist kein einzelnes Modul, sondern besteht aus mehreren Modulen. Die Funktionen, die für die Implementierung der Benutzerschnittstelle benötigt werden, müssen also aus verschiedenen Modulen importiert werden. Das Modul PTI dient dazu, diese Importe zu verwalten. Es ist das einzige Modul, dass die Funktionen aus dem PT-Programm importiert. Eine Ausnahme dabei bildet das Modul DrawEngine, dass direkt ein Modul des PT-Programms importiert. Diese Art der Organisation der Importe vereinfacht es den Überblick über die genutzten Funktionen aus dem PT-Programm zu behalten. Folgende Abbildung soll die Idee veranschaulichen:

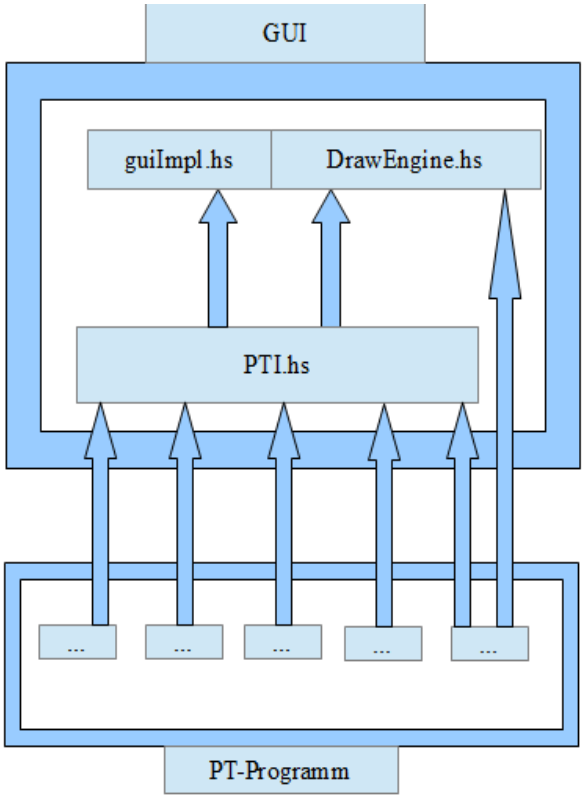


Abb. 4.3.1: Veranschaulichung der Importstruktur

Die erste Funktion die vom Modul PTI bereitgestellt wird ist `createCalc`. Die Eingabe ist hier ein Dateipfad vom Typ `FilePath`, der ein Programmkalkül beschreibt. Aus diesem wird ein Programmkalkül des Typs `Calculus`.

```
createCalc :: FilePath -> Calculus
createCalc x = execIt (loadCalc x)
```

Neben dem direkten Import von Funktionen aus dem PT-Programm, stellt das Modul PTI eine Funktion zur Verfügung, die Programmtransformationen aus einem Programmkalkül des Typs `Calculus` extrahiert. Diese wird dafür benötigt, um dem Nutzer die Programmtransformationen des Programmkalküls zu präsentieren. Siehe Abbildung **4.1.4**. Dieser kann dann für eine dieser Programmtransformationen Diagramme erzeugen. Als Darstellungsform wurde eine Liste gewählt, da diese einfach zu verarbeiten ist.

```
getReds :: Calculus -> [String]
getReds x = createLabelList (createRulenameList (extractReds x))

createLabelList :: [Item] -> [String]
createLabelList [] = []
createLabelList (ruleName:xs) = (createLabelList xs)++[extractRulenameString ruleName]

extractRulenameString :: Item -> String
extractRulenameString (Diag.Trans (Diag.Rule x)) = x

createRulenameList
  :: [Rule] -> [Item]
createRulenameList [] = []
createRulenameList ((Rule ruleName ruleLhs ruleRhs):xs) = (createRulenameList xs)++[ruleName]

extractReds
  :: Calculus -> [Rule]
extractReds (Calc sortDecs cclasses cDecs cLabels noReds reds) = reds
```

Die Funktion `getReds` erwartet ein Programmkalkül des Typs `Calculus` und liefert eine Liste von Strings als Rückgabe. Die Strings sind die im Programmkalkül definierten Namen der Programmtransformationen. Hierfür werden zu Beginn die Programmtransformationen des Programmkalküls mit Hilfe der Funktion `extractReds` extrahiert. Da diese im Typ `Calculus` als `reds` benannt sind, können diese einfach über die Variable entnommen werden. Da der Nutzer nicht den gesamten Typ der Programmtransformationen sehen soll und die am Ende resultierende Liste nicht aus Elementen des Typs `Rule` bestehen soll, müssen aus dem Typ `Rule` die Namen der Programmtransformationen in Form eines Strings entnommen werden. Dafür wird in `createRulenameList` die Liste durchlaufen und für jeden Eintrag den Typs `Rule` der Name extrahiert, der in Form eines Elements des Typs `ruleName`, beziehungsweise des Typs `Item`, vorliegt. Die dadurch erzeugte Liste besteht nun aus Elementen des Typs `Item`. Die Funktion `createLabelList` durchläuft die Liste erneut und bei jedem Element wird der Name der Programmtransformation in Form eines Strings extrahiert. Da es sich um Programmtransformationen handelt, bestehen alle Elemente des Typs `Rule` aus Elementen der Form `(Trans (Rule x)`. Das *x* steht hier für den Namen der Programmtransformation in Form eines Strings. Die Funktion `extractRulenameString` extrahiert genau diesen Namen. In der folgenden Abbildung wird die Typstruktur einer Programmtransformation im PT-Programm und die Vorgehensweise der Funktion `getReds` dargestellt:

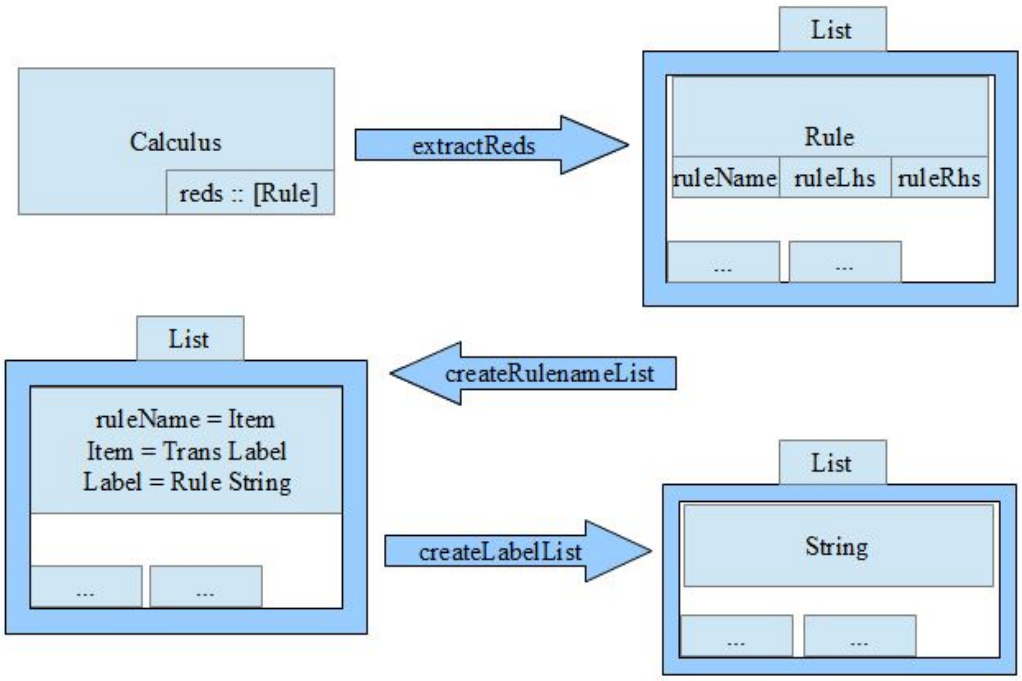


Abb. 4.3.2: Veranschaulichung der Funktion getReds

Die Funktionen `itrsOutput` und `textOutput` sind erneut direkte Importe aus dem PT-Programm.

```
itrsOutput :: [Diagram] -> String
itrsOutput x = toITRS x

textOutput :: [Diagram] -> String
textOutput x = printDiagrams x
```

Beide Funktionen erwarten als Eingabe eine Liste von Diagrammen. Diese Diagramme werden dann in Strings verwandelt, die einmal die Diagramme als Textausgabe und in Form eines TRS darstellen. Die letzten Funktionen `calc2Diag` und `calc2GenDiag`, die nachfolgend aufgeführt werden, erwarten als Eingabe einen Pfad vom Typ `FilePath` zu einer Datei, die ein Programmkalkül darstellt und eine Programmtransformation vom Typ `String`. Aus diesen Eingaben erzeugt die Funktion `diagramsFor` eine Liste mit Diagrammen und `generalizedDiagramsFor` eine reduzierte Liste von Diagrammen. In beiden Funktionen müssen die Listen von Diagrammen aus Monaden entnommen werden. Hierfür wird erneut die Funktion `execIt` verwendet.

```
calc2Diag
  :: FilePath -> String -> [Diagram]
calc2Diag fpath transf = execIt (diagramsFor fpath transf)

calc2GenDiag
  :: FilePath -> String -> [Diagram]
calc2GenDiag fpath transf = execIt (generalizedDiagramsFor fpath transf)
```

4.4 Graphische Oberfläche

Die graphische Oberfläche wurde mit Hilfe folgender Werkzeuge erstellt:

- *Glade* [GLADE]
- *GTK+* [GTK]
- *Gtk2hs* [GTK2HS]

Glade ist ein Tool zum Entwerfen von graphischen Benutzerschnittstellen, die in Form von XML-Dateien abgespeichert werden. Diese können dann mittels GTK+ gesteuert und genutzt werden. Bei GTK+ handelt es sich um ein von mehreren Plattformen unterstütztes Werkzeug zum erstellen und steuern von graphischen Benutzeroberflächen. GTK+ unterstützt eine große Bandbreite an Programmiersprachen unter Anderem auch Haskell. Bei Gtk2Hs handelt es sich um eine auf GTK+ basierte Bibliothek für Haskell. Mit dieser kann man funktional graphische Benutzeroberflächen steuern und implementieren.

4.4.1 Oberflächendesign mittels Glade

Die graphischen Oberflächen und Elemente der Benutzerschnittstelle können sowohl dynamisch und direkt mittels GTK+ erstellt werden, als auch statisch in Form einer XML-Datei. Die Erzeugung der Oberflächen mittels Glade hat den Vorteil, dass man einfach und anschaulich per *Drag and Drop*-Verfahren Fenster und graphischen Elemente erzeugen und organisieren kann. Die resultierende XML-Datei kann dann im Code mit Hilfe der Werkzeuge direkt angesprochen und genutzt werden.

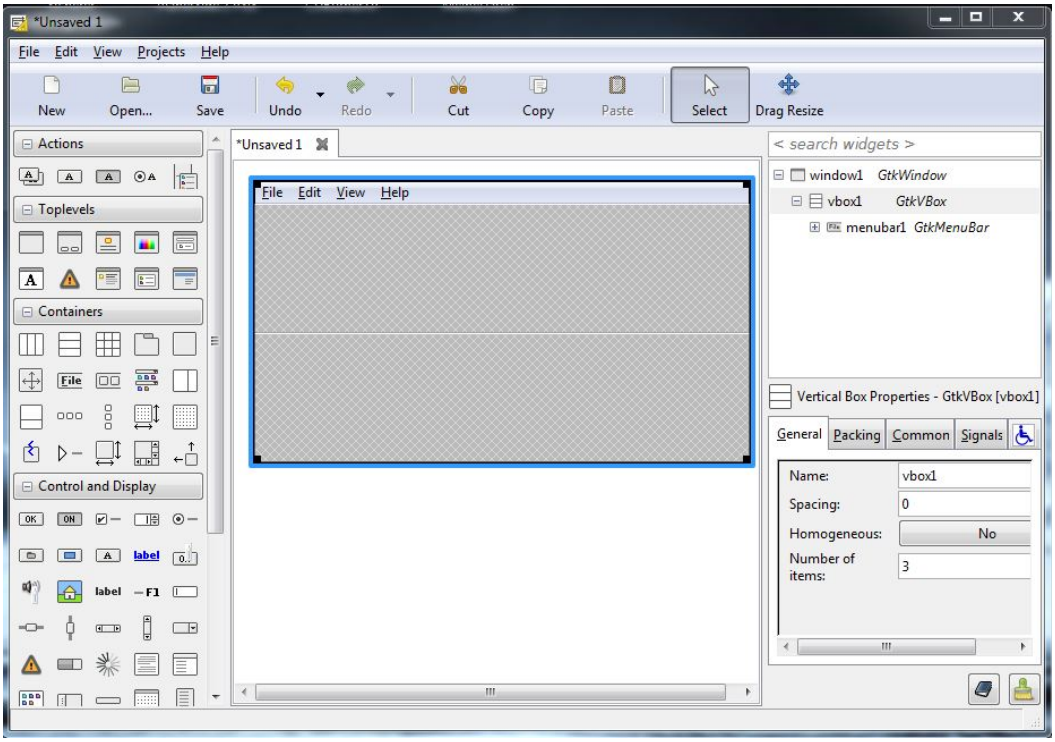


Abb. 4.4.1: Glade

Die mit Glade, siehe Abbildung **4.4.1**, erzeugten XML-Dateien haben die Endung *.glade* und sind wie folgt strukturiert:

```
<?xml version="1.0" encoding="UTF-8"?>
<interface>
  <requires lib="gtk+" version="2.24"/>
  <!-- interface-naming-policy project-wide -->
  <object class="GtkWindow" id="window1">
    <property name="can_focus">False</property>
    <child>
      <object class="GtkVBox" id="vbox1">
        <property name="visible">True</property>
        <property name="can_focus">False</property>
        <child>
          <object class="GtkMenuBar" id="menubar1">
            <property name="visible">True</property>
            <property name="can_focus">False</property>
            <child>
              <object class="GtkMenuItem" id="menuitem1">
                <property name="visible">True</property>
                <property name="can_focus">False</property>
                <property name="label" translatable="yes">_File</property>
                <property name="use_underline">True</property>
                <child type="submenu">
                  <object class="GtkMenu" id="menu1">
                    <property name="visible">True</property>
                    <property name="can_focus">False</property>
```

Abb. 4.4.2: Ausschnitt aus einer GLADE-Datei

Alle Elemente der graphischen Oberfläche haben eine Identifikation (*id*) in Form eines Namens mit der sie eindeutig angesprochen werden können. Zudem hat jedes Objekt bestimmte Attribute, die sowohl in Glade, also auch später im Code direkt verändert werden können. Jedes Attribut hat einen Namen und einen bestimmten Wert. Attribute die nicht in der GLADE-Datei angesprochen werden haben einen *default* Wert, also einen festgelegten Standartwert. Die Hierarchie der Oberfläche ist klar definiert. Sie wird in Form von Eltern-Kind-Beziehungen dargestellt. In den Abbildungen **Abb. 4.4.2** und **4.4.3** kann man diese Hierarchie erkennen. Das Hauptfenster namens *window1* bildet das Fundament für die gesamte Struktur. An diesem Objekt orientieren sich nun alle folgenden Elemente. Das nächste Element dient dazu, vertikale Bereiche abzutrennen in denen Platz für weitere Objekte ist. In diesem Fall wurde das Hauptfenster in drei vertikale Bereiche geteilt. Dieses Objekt wird als *GtkVBox* bezeichnet und hat die *id vbox1*. Das letzte Objekt das hinzugefügt wurde ist eine *GtkMenueBar*, die ein Menü repräsentiert. Man kann erkennen das dieses Menü wiederum aus weiteren inneren Objekten besteht, die automatisch angelegt wurden. Die Hierarchie der Objekte kann man verdeutlichen, wenn man sich mit den Breiten- und Höhenanforderungen der Objekte auseinandersetzt. Wenn das Hauptfenster eine beschränkte Abmessung zugeteilt bekommt, müssen sich alle Kind-Objekte an diese Abmessung halten. Das gleiche gilt auch für innere Eltern-Kind-Beziehungen. Die Abmessungen für Objekte können hier selbst statisch angegeben werden oder dynamisch berechnet werden.

4.4.2 Graphische Oberfläche der Benutzerschnittstelle

Die graphische Oberfläche der Benutzerschnittstelle wird durch vier GLADE-Dateien beschrieben. Das Hauptfenster wird in der Datei *gui.glade* beschrieben und besteht aus folgenden Elementen:

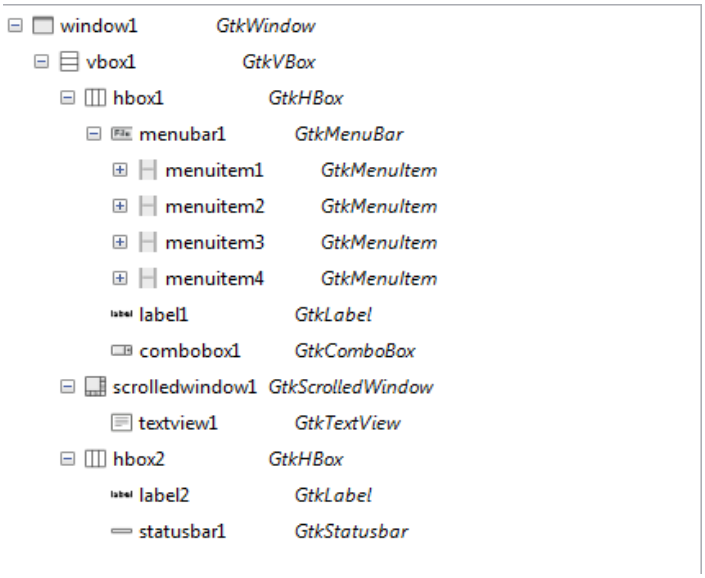


Abb. 4.4.3: Struktur des Hauptfensers

Das Hauptfenster aus **Abb. 4.1.1** besteht aus einem Fenster *window1*. Dieses wird in drei Bereiche unterteilt mittels *vbox1*. Der erste Bereich der *vbox1* enthält eine *GtkHBox*. Diese verhält sich analog zu der *GtkVBox*, in dem sie die Bereiche horizontal teilt. Im ersten Teil der drei horizontalen Bereiche befindet sich eine *GtkMenuBar*. Diese besteht aus den vier Menüpunkten die im Hauptfenster als *File*, *Edit*, *Run* und *Help* bekannt sind. Die zwei restlichen Bereiche sind ein Texttitel und eine *GtkComboBox*. Mit dieser können Listen dargestellt werden, dessen Listenelemente ausgewählt werden können. In **Abb. 4.1.4** wird dies veranschaulicht. Der zweite Bereiche enthält ein *GtkScrolledWindow*. Dieses Fenster kann bestimmte Inhalte anzeigen und ist mit *Scrollbars* ausgestattet. Der Inhalt des *scrolledwindow1* ist ein *GtkTextView*. Dieses Objekt kann Strings repräsentieren. Der Nutzer kann auf der Fläche des Objekts diese Strings verändern. Der letzte vertikale Bereich enthält erneut eine horizontale Aufteilung. Der erste Bereiche der *hbox2* enthält einen Texttitel und der zweite eine *GtkStatusbar*. Mit dieser lassen sich Strings repräsentieren. Die Strings werden mit einem Stack verwaltet.

Das Fenster zum Betrachten von Bilder wird in *imageviewer.glade* beschrieben und besteht aus folgenden Objekten:

Abb. 4.4.4: Struktur des Fensters zur Bildbetrachtung

Das Fenster aus **Abb. 4.1.3** besteht ebenfalls aus einem Hauptfenster *window1*. Dieses Fenster wird in zwei vertikale Bereiche unterteilt. In dem ersten Bereich befindet sich ein *GtkScrolledWindow* und im zweiten Bereich ein *GtkButton*. Dieses Objekt repräsentiert einen Button. Im *scrolledwindow1* wird später die Bilddatei dynamisch eingesetzt.

Das Fenster aus **Abb 4.1.5** hat folgende Struktur und wird in *outputdialog.glade* beschrieben:

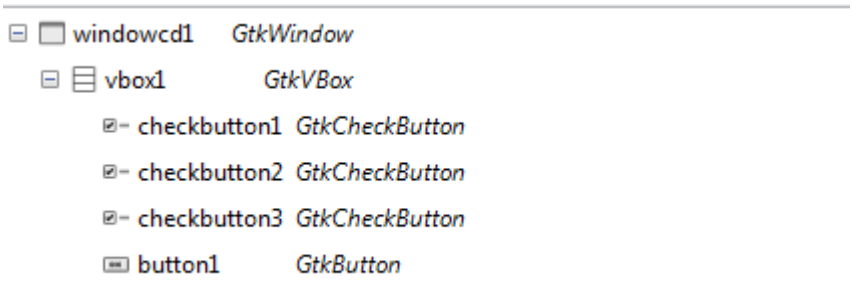


Abb. 4.4.5: Struktur des Outputdialog Fensters

Das Fenster *windowcd1* wird hier in vier Bereiche eingeteilt. Die ersten drei bestehen aus *GtkCheckButton*, die eine Auswahlmöglichkeit darstellen. Der letzte besteht aus einem Button. Das *Hilfe*-Fenster wird in *about.glade* beschreiben.

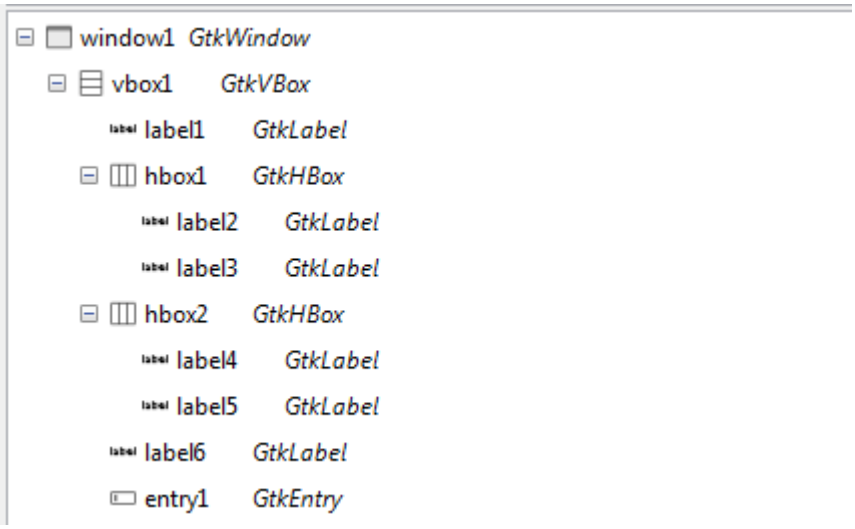
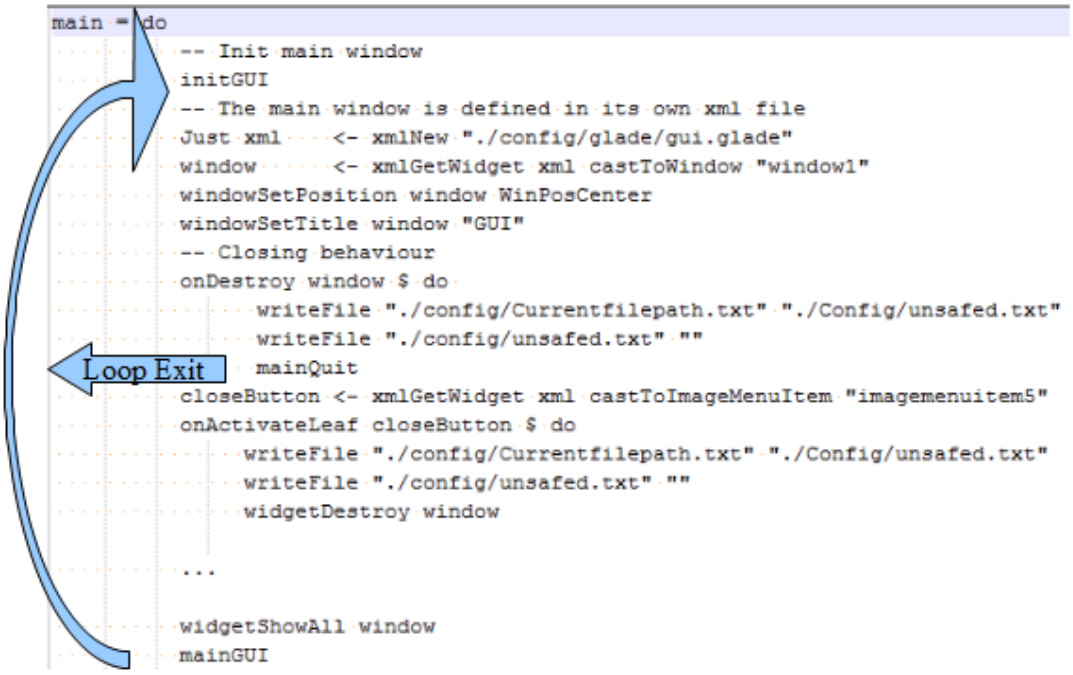


Abb. 4.4.6: Struktur des Hilfe-Fensters

4.4.3 Programmierung der Oberfläche mittels Haskell

Mit Hilfe von GTK+ und der Gtk2Hs Bibliothek lassen sich die mit Glade beschriebenen statischen Objekte direkt ansprechen oder direkt dynamisch erzeugen, ohne statische GLADE-Datei als Vorlage. Den Zugriff auf die Funktionen, die zum Erstellen und Verändern der Objekte benötigt werden, erhält man durch den Import des Pakets *Graphics.UI.Gtk*. Dieses Paket und dessen Unterpakete bilden die Gtk2Hs Bibliothek zur Nutzung von GTK+ mittels Haskell. Eine Hauptfunktion die eine graphische Oberfläche initialisiert ist `initGUI`. Diese muss vor jeder Funktion aus der Gtk2Hs Bibliothek aufgerufen werden und hat den Typ `IO [String]`. Die graphische Benutzerschnittstelle wird mittels Monaden realisiert. Dies ist nötig um Seiteneffekte zu ermöglichen. Um die Veränderung auf der Oberfläche zu registrieren und verarbeiten zu können, muss die Aktivität auf der Oberfläche regelmäßig geprüft werden. Dies wird durch die Funktion `mainGUI` gewährleistet. Die beiden Funktionen bilden eine Schleife, die mittels Funktion `mainQuit` beendet werden kann. **Abb. 4.4.7** soll die drei Funktionen veranschaulichen.




```
main = do
    -- Init main window
    initGUI
    -- The main window is defined in its own xml file
    Just xml    <- xmlNew "./config/glade/gui.glade"
    window      <- xmlGetWidget xml castToWindow "window1"
    windowSetPosition window WinPosCenter
    windowSetTitle window "GUI"
    -- Closing behaviour
    onDestroy window $ do
        writeFile "./config/Currentfilepath.txt" "./Config/unsafed.txt"
        writeFile "./config/unsafed.txt" ""
        mainQuit
    closeButton <- xmlGetWidget xml castToImageMenuItem "imagemenuitem5"
    onActivateLeaf closeButton $ do
        writeFile "./config/Currentfilepath.txt" "./Config/unsafed.txt"
        writeFile "./config/unsafed.txt" ""
        widgetDestroy window

    ...

    widgetShowAll window
    mainGUI
```

Abb. 4.4.7: GTK+ main event loop

Bevor näher auf den Code eingegangen wird ist es notwendig, die Hierarchie der Objekte in GTK+ und der Gtk2Hs Bibliothek zu erläutern. In folgendem Ausschnitt aus [GTKHie] kann

man die Hierarchie der Objekte erkennen:

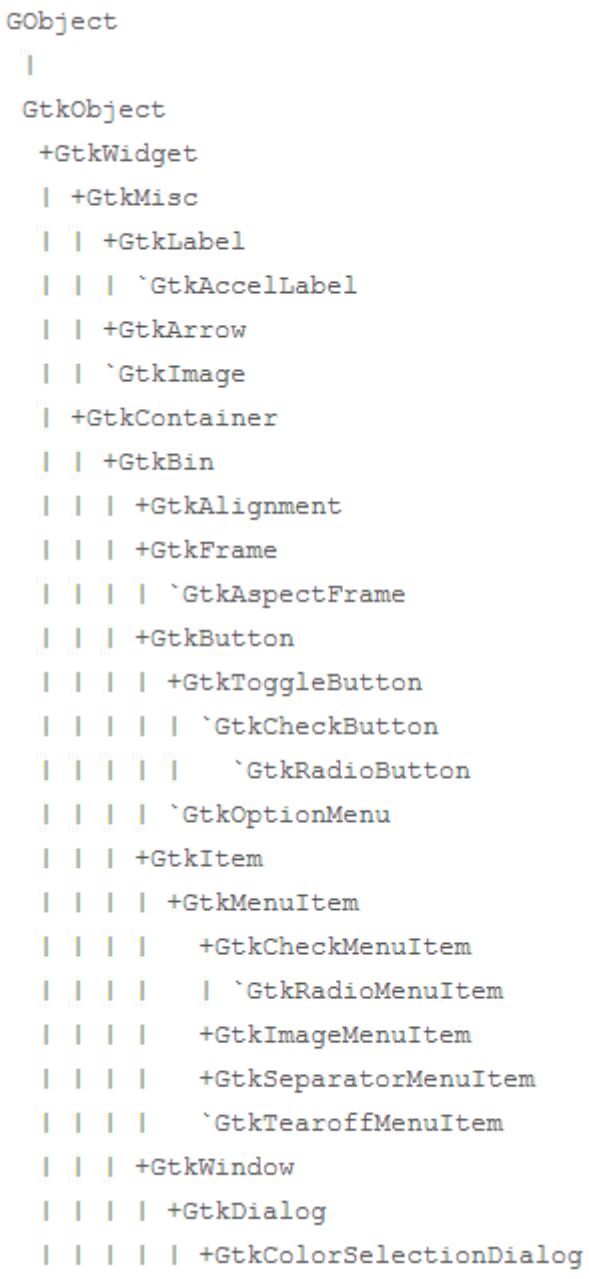


Abb. 4.4.8: Ausschnitt der Objekthierarchie in GTK+

Um die GLADE-Dateien zu verwenden bietet die Gtk2Hs Bibliothek die Funktion `xmlNew` an. Diese nimmt einen `FilePath`, der zu einer XML-Datei führt und macht aus dieser ein XML-Objekt. Aus diesem Objekt lassen sich mit der Funktion `xmlGetWidget` alle *GObjekte* beziehen, die in der Datei beschrieben werden. Die Funktion bekommt als Eingabe das XML-Objekt. Als zweites Argument wird eine Funktion angegeben, die aus einem GObjekt ein explizites *Widget* macht. Das explizite Widget kann zum Beispiel ein Fenster (*Window)* sein. Als letztes bekommt die Funktion die *id* des Objektes in der XML-Datei. Wie schon anfangs erwähnt, kann man Objekte auch ohne XML-Dateien erzeugen. Dazu bietet Gtk2Hs eigene Funktionen an. Ein Beispiel hierfür ist die Funktion `windowNew`, die ein Fenster erzeugt. Ein Fenster kann also mit diesem Ausdruck und ohne XML-Datei erzeugt werden:

```
window <- windowNew
```

Die Erzeugung der Objekte im Code gilt analog für alle GObjekte. Für jedes dieser Objekte bietet Gtk2Hs eine Reihe von Funktionen an, mit denen man diese Objekte erzeugen, deren Attribute verändern und Aktionen am Objekt erkennen kann. Dabei gilt das Prinzip der Vererbung der Attribute und Funktionen an Objekte die hierarchisch untergeordnet sind.

4.4.4 Implementierung der graphischen Oberfläche und der Anforderungen

Die Funktion `main` implementiert das Hauptfenster und bildet das zentrale Steuerelement aus dem alle anderen graphischen Objekte, Fenster und Funktionen ausgeführt werden. Im ersten Abschnitt der Funktion wird das Hauptfenster initialisiert, dass aus der Datei *gui.glade* gelesen wird. Alle Objekte des Hauptfensters werden mit Hilfe der Funktion `xmlGetWidget` und den dazugehörigen Argumenten initialisiert und implementiert. Der Abschnitt in der die Menüpunkte für das *File* Menü initialisiert werden, soll dies kurz veranschaulichen:

```
filenewButton      <- xmlGetWidget xml castToImageMenuItem "imagemenuitem1"
fileopenButton     <- xmlGetWidget xml castToImageMenuItem "imagemenuitem2"
fileopenimageButton <- xmlGetWidget xml castToImageMenuItem "imagemenuitem2_1"
filesaveButton     <- xmlGetWidget xml castToImageMenuItem "imagemenuitem3"
filesaveasButton   <- xmlGetWidget xml castToImageMenuItem "imagemenuitem4"
```

Je nach Art des Objekts, können Aktionen an diesem Objekt durch eine Funktion erkannt werden. Ein Aktion an einem Objetk kann zum Beispiel das Schließen dieses Objekts durch das Betätigen des *Schließen* Symbols sein:

```
onDestroy window $ do
    writeFile "./config/Currentfilepath.txt" "./Config/unsafed.txt"
    writeFile "./config/unsafed.txt" ""
    mainQuit
```

Dabei ist `window` der Name des Objektes das behandelt wird. In diesem Fall das Hauptfenster. Analog gibt es Funktionen, die das Auswählen eines der Menüpunkte registrieren können. In der `main` Funktion werden alle Aktionen des Nutzer abgefangen und in Unterfunktionen bearbeitet. Diese Aktionen werden nun einzeln erläutert:

Schließen der Anwendung:

Dem Nutzer ist es erlaubt die Anwendung durch Auswahl des *Exit* Menüpunktes im *File* Menü, oder durch Schließen des Fensters mittels des *Schließen* Symbols zu beenden. Hierbei wird in beiden Fällen der aktuelle Dateipfad auf *./config/unsafed.txt* gesetzt und der Inhalt der *unsafed.txt* gelöscht. Dies ist nötig, um beim erneuten Start der Anwendung eine leere und neue Textdatei vorzufinden, die als aktuelle Datei vom Programm erkannt wird. Das Schließen der Anwendung erfolgt durch das Beenden der Hauptschleife mittels `mainQuit`.

```
onDestroy window $ do
    writeFile "./config/Currentfilepath.txt" "./Config/unsafed.txt"
    writeFile "./config/unsafed.txt" ""
    mainQuit
closeButton <- xmlGetWidget xml castToImageMenuItem "imagemenuitem5"
onActivateLeaf closeButton $ do
    writeFile "./config/Currentfilepath.txt" "./Config/unsafed.txt"
    writeFile "./config/unsafed.txt" ""
    mainQuit
```

Neue Textdatei:

Wenn der Nutzer eine neue Textdatei erstellt, wird zunächst die Liste der Programm-transformationen gelöscht, in dem der Inhalt der *Combobox* zurückgesetzt wird. Das Objekt in dem der Text dargestellt wird ist ein *TextView.* Dieses besteht aus einem Objekt das *Textbuffer* heißt und Strings speichern kann. Das TextView Objekt ist in diesem Fall ein Behälter, in dem ein Textbuffer angezeigt und verändert werden kann. Der Textbuffer des TextView wird in diesem Fall abgerufen, gelöscht und erneut dem TextView zugewiesen. Der aktuelle Dateipfad wird aktualisiert und der Inhalt der Textdatei ebenfalls gelöscht. In der Statusanzeige und in der Konsole wird angezeigt das eine neue Textdatei erstellt wurde.

```
onActivateLeaf filenewButton $ do
    comboBoxSetModelText combobox
    textbuffer <- textViewGetBuffer textview
    textBufferSetText textbuffer ""
    textViewSetBuffer textview textbuffer
    writeFile "./config/Currentfilepath.txt" "./Config/unsafed.txt"
    writeFile "./config/unsafed.txt" ""
    setStatusbar statusbar 0 "New file"
    putStrLn "New file"
```

Textdatei öffnen:

Wenn eine neue Textdatei geöffnet wird, wird ebenfalls die Liste der Programmtransformationen zurückgesetzt. Zudem wird die Funktion `openDialog` ausgeführt. Diese öffnet den File Manager zum Öffnen einer Textdatei und wird später erläutert.

```
onActivateLeaf fileopenButton $ do
    comboBoxSetModelText combobox
    openDialog window textview statusbar
```

Bilddatei öffnen:

Wenn eine neue Bilddatei geöffnet wird, wird die Funktion `openImageDialog` aufgerufen. Diese Öffnet den File Manager zum Öffnen von Bilddateien und wird ebenfalls später genauer erläutert.

```
onActivateLeaf fileopenimageButton $ do
    openImageDialog window statusbar
```

Textdatei speichern:

Wenn die aktuelle Textdatei unter dem aktuellen Dateipfad gespeichert werden soll, wird der aktuelle Textbuffer extrahiert. Der Textbuffer hat ein Start- und Endwert, mit dem man den aktuellen String des Textbuffers in seiner vollen Länge ansprechen kann. Diese werden mittels einer entsprechender Funktion ermittelt. Der String des Textbuffers kann nun mit Hilfe des Start- und Endwertes entnommen und weiterverarbeitet werden. Sollte es sich bei dem aktuellen Dateipfad um den Standartpfad einer neuen Textdatei *./config/unsafed.txt* handeln, dann wird der File Manager zum Speichern von Textdateien aufgerufen mittels einer Funktion

`saveAsDialog`. Ansonsten wird in die aktuelle Datei der Text aus dem Textbuffer geschrieben. Auch hier wird in der Konsole und in der Statusanzeige die Aktion angezeigt. Um Fehler im Verlauf des Programmabschnitts abzufangen, wird mit der Funktion `catch` gearbeitet. Diese kann Fehler des Datentyps `SomeExeption` erkennen und einen separaten Programmablauf starten. Bei dem Datentyp `SomeException` handelt es sich um den allgemeinsten Fehlertyp in der Hierarchie der Fehlermeldungen. Sowohl die Funktion `catch`, also auch die Hierarchie der Fehlerdatentypen können dem Modul *Control.Exception* entnommen werden. Im Falle eines Fehlers, wird in der Statusanzeige und in der Konsole eine Fehlermeldung angezeigt.

```
onActivateLeaf filesaveButton $ do
    CE.catch (do    comboBoxSetModelText combobox
                    textbuffer <- textViewGetBuffer textview
                    startIt <- textBufferGetStartIter textbuffer
                    endIt <- textBufferGetEndIter textbuffer
                    text <- textBufferGetText textbuffer startIt endIt True
                    fpath <- readFile "./config/Currentfilepath.txt"
                    if (takeFileName fpath) == "unsafed.txt"
                    then do
                        saveAsDialog window textview statusbar
                        putStrLn "Save as dialog"
                    else do
                        writeFile fpath text
                        setStatusbar statusbar 1 "File saved"
                        putStrLn "File saved")

             (\(CE.SomeException err) -> do
                                        setStatusbar statusbar 1 "Saving file failed."
                                        putStrLn "Saving file failed.")
```

Textdatei speichern unter:

In diesem Fall wird die Funktion ausgeführt, die den File Manager zum Speichern von Textdateien aufruft.

```
onActivateLeaf filesaveasButton $ do
    saveAsDialog window textview statusbar
```

Text Kopieren, Ausschneiden und Einfügen:

Der Textbuffer bietet für alle drei Aktionen entsprechende Funktionen an. In allen drei Fällen wird zunächst der Textbuffer des TextView extrahiert. Danach wird mittels der Funktion `atomNew` ein Index in einer globalen Stringtabelle angelegt. Dies ist nötig, um den Text global auf dem System nutzen zu können. Mit Hilfe der Funktion `clipboardGet` kann der globale Index lokal genutzt werden. Je nach Aktion wird nun die entsprechende Funktion des Textbuffers ausgeführt und der veränderte Textbuffer dem TextView zurückgegeben. Die entsprechende Aktion wird der Konsole angezeigt. Zur Veranschaulichung wird nur die Implementierung der *Kopieren* Aktion angezeigt.

```
onActivateLeaf editcopyButton $ do
    textbuffer <- textViewGetBuffer textview
    cbatom <- atomNew "GDK_SELECTION_CLIPBOARD"
    cb <- clipboardGet cbatom
    textBufferCopyClipboard textbuffer cb
    textViewSetBuffer textview textbuffer
    putStrLn "Copy"
```

Programmkalkül auslesen:

In diesem Fall wird die Liste der Programmtransformationen gelöscht, um sie mit den neuen Programmtransformationen aus dem aktuellen Programmkalkül zu füllen. Hierfür wird der aktuelle Dateipfad der Textdatei mit dem Programmkalkül ermittelt. Dieser wird an die Funktion `getReds` aus dem Modul PTI weitergeleitet. Diese erstellt die Liste der Programmtransformationen, die direkt an die Funktion `fillCombobox` weitergeleitet wird. Diese Funktion nimmt die Liste der Programmtransformationen und füllt damit die Combobox. Sollte es zu Fehlern kommen, dann wird die Statusanzeige und die Konsole mit einer entsprechenden Fehlermeldung ausgestattet.

```
onActivateLeaf runparseButton $ do
    putStrLn "Parse Calculus"
    CE.catch (do
                -- The current content of the textview gets passed over to the Ca
                fpath <- readFile "./config/Currentfilepath.txt"
                -- The extracted transformations get written into the combobox
                comboBoxSetModelText combobox
                fillCombobox combobox (getReds (createCalc fpath))
                putStrLn "Calculus created"
                setStatusbar statusbar 0 ("Calculus parsed into the type Calculus.
                putStrLn ("Calculus parsed into the type Calculus. You see the ext
             (\(CE.SomeException err) -> (do  setStatusbar statusbar 1 "Unable to pa
                                              putStrLn "Unable to parse Calculus."))
```

Diagramme berechnen:

Zuerst wird der aktuelle Dateipfad der Textdatei und die ausgewählte Programmtransformation aus der Combobox ermittelt. Sollte keine Programmtransformation ausgewählt worden sein, so wird eine Fehlermeldung ausgegeben. Wenn alle nötigen Informationen vorhanden sind, werden diese an die Funktion `createDiagramDialog` weitergegeben. Diese Funktion öffnet das Fenster in dem der Nutzer die Ausgabeform auswählen kann. Sollte es im gesamten Verlauf zu einem Fehler kommen, dann wird eine Fehlermeldung ausgegeben.

```
onActivateLeaf runcreateButton $ do
    CE.catch (do
                -- The current file and the chosen transformation get passed over to the create
                fpath <- readFile "./config/Currentfilepath.txt"
                transf <- comboBoxGetActiveText combobox
                putStrLn "Creating diagrams... THIS MAY TAKE A WHILE DEPENDING ON YOUR SYSTEM RE
                setStatusbar statusbar 0 ("Creating diagrams... THIS MAY TAKE A WHILE DEPENDING
                case transf of
                    Just a -> do
                                createDiagramsDialog window (fromJust transf) statusbar fpath
                    Nothing -> do
                                putStrLn "Failed to create diagrams. Please choose a transfor
                                setStatusbar statusbar 0 ("Failed to create diagrams. Please
            (\(CE.SomeException err) -> do
                                putStrLn "Failed to create diagrams"
                                setStatusbar statusbar 0 ("Failed to create diagrams"))
```

Hilfe anzeigen:

Die letzte mögliche Aktion des Nutzer ist das Anzeigen des Hilfe-Fensters. In diesem Fall wird die Funktion `showInfo` ausgeführt.

```
onActivateLeaf helpaboutButton $ do
    putStrLn "Information"
    showInfo window
```

Im folgenden wird die Funktionsweise der File Manager anhand der Funktion `openImageDialog` erläutert. Die File Manager werden mit Hilfe der Funktionen `fileChooserWidgetNew` oder `fileChooserDialogNew` erzeugt. Der Unterschied der beiden Funktionen liegt in der Hierarchie der Objekte und kann in der Gtk2Hs Bibliothek nachgelesen werden. Zu Beginn wird ein Fenster erzeugt, dass als Behältnis für den File Manager dienen soll und das Hauptfenster wird ausgegraut. Danach wird mittels der oben genannten ersten Funktion das *FileChooserWidget* erzeugt. Als Argument wird angegeben,

welche Aufgabe der File Manager ausführen soll. Hierbei handelt es sich um das Öffnen von Dateien. Das FileChooserWidget wird dann mittels entsprechender Funktion dem erzeugten Fenster zugewiesen. In den nächsten zwei Abschnitten werden zwei Filter definiert. Der erste Filter soll nur Dateien im PNG-Format anzeigen und der zweite alle Dateiformate.

```
openImageDialog window statusbar = do
    -- Filechooser
    initGUI
    widgetSetSensitive window False
    windowSetDeletable window False
    windowfc <- windowNew
    set windowfc [windowTitle := "Open image from explorer...",windowDefaultWidth := 600,
    filechooserwidget <- fileChooserWidgetNew FileChooserActionOpen
    containerAdd windowfc filechooserwidget
    windowSetPosition windowfc WinPosCenterOnParent

    jpgfilt <- fileFilterNew
    fileFilterAddPattern jpgfilt "*.png"
    fileFilterSetName jpgfilt ".png"
    fileChooserAddFilter filechooserwidget jpgfilt

    nofilt <- fileFilterNew
    fileFilterAddPattern nofilt "*.*"
    fileFilterSetName nofilt "All files"
    fileChooserAddFilter filechooserwidget nofilt
```

Die Aktionen im File Manager werden in folgendem Teil behandelt. Im Falle einer Aktion werden das ausgewählte Verzeichnis und die Datei abgerufen. Dabei wird das aktuelle Verzeichnis, falls eines ausgewählt wurde, angegeben. Sollte eine Datei erfolgreich ausgewählt worden sein, dann wird das Fenster des File Managers geschlossen und der Dateipfad an die Funktion `openImageView` zum Öffnen von Bilddateien weitergegeben. Im Falle eines Fehlers, wird eine Fehlermeldung ausgegeben. Sollte das Fenster vom Nutzer geschlossen werden, dann wird der Status des Programms auf *Ready* gesetzt und die Funktionen des Hauptfensters wieder reaktiviert. Dabei ist zu beachten, dass im Falle eines erfolgreichen Öffnens einer Bilddatei die Reaktivierung des Hauptfensters der Funktion `openImageView` überlassen wird.

```
onFileActivated filechooserwidget $ do
    dir <- fileChooserGetCurrentFolder filechooserwidget
    file <- fileChooserGetFilename filechooserwidget
    case dir of
        Just dpath -> putStrLn ("The current directory is: " ++ dpath)
        Nothing -> putStrLn "Nothing"

    case file of
        Just fpath -> do
                      -- Cast image to a gtk window
                      CE.catch (do widgetDestroy windowfc
                                   openImageView fpath window statusbar)
                               (\(CE.SomeException err) -> (do setStatusbar statu
                                                               putStrLn "Unable t
                                                               )
        Nothing -> putStrLn "Nothing"
widgetShowAll windowfc
onDestroy windowfc $ do
    setStatusbar statusbar 0 "Ready"
    widgetSetSensitive window True
    windowSetDeletable window True
    mainQuit
mainGUI
```

Das Fenster zum Betrachten und Speichern von Bilddateien wird durch die Funktion `openImageView` erzeugt. Diese deaktiviert das Hauptfenster und erzeugt das Bildbetrachtungsfenster mittels *imageviewer.glade*. Falls der Nutzer dieses Fenster schließt, wird das Hauptfenster reaktiviert und die Statusanzeige des Programms verändert. Bilddateien werden mit Hilfe des Objektes *Image* visualisiert. Dieses ist ähnlich strukturiert wie ein TextView. Statt eines Textbuffers zum Speichern von Strings wird in diesem Fall ein *Pixbuf* ,das Pixelinformationen speichert, verwendet. Dieser wird dann dem Image Objekt zugeteilt. Das Image Objekt wird dann in das vordefinierte *ScrolledWindow* Objekt aus der GLADE-Datei hineingesetzt. In der Statusanzeige und der Konsole wird dem Nutzer der aktuelle Vorgang mitgeteilt. Im letzten Abschnitt wird die *Speichern unter* Funktion des Fensters implementiert. In diesem Fall wird die Funktion `saveAsDialogImage` mit den passenden Informationen aufgerufen, die einen angepassten File Manager zum Speichern von Bilddateien anzeigt.

```
openImageView fpath window statusbar = do initGUI
                                          widgetSetSensitive window False
                                          windowSetDeletable window False
                                          -- The image view window is defined in its own xml file
                                          Just xml      <- xmlNew "./config/glade/imageviewer.glade"
                                          windowiv      <- xmlGetWidget xml castToWindow "window1"
                                          windowSetPosition windowiv WinPosCenter
                                          set windowiv [windowTitle := "Imageviewer"]
                                          onDestroy windowiv $ do
                                              widgetDestroy windowiv
                                              setStatusbar statusbar 0 "Ready"
                                              widgetSetSensitive window True
                                              windowSetDeletable window True
                                              mainQuit
                                          scrolledwindowImg <- xmlGetWidget xml castToScrolledWindow "scr
                                          pixbuf <- pixbufNewFromFile fpath
                                          image <- imageNewFromPixbuf pixbuf
                                          scrolledWindowAddWithViewport scrolledwindowImg image
                                          setStatusbar statusbar 0 ("Image: "++fpath++" opened")
                                          putStrLn ("Image: "++fpath++" opened")
                                          imagesaveasbutton <- xmlGetWidget xml castToButton "button1"
                                          -- User pressed the save image button, saveAsDialog for images
                                          onClicked imagesaveasbutton $ do
                                              saveAsDialogImage window windowiv statusbar fpath pixbuf
                                          widgetShowAll windowiv
                                          mainGUI
```

Die Funktion `createDiagramsDialog` implementiert das Fenster zur Auswahl der Ausgabeformen für die Diagramme, wie in Abbildung **4.1.5** zu sehen ist. Auch hier werden die Grundobjekte aus der GLADE-Datei geladen und initialisiert. Der Unterschied zu den anderen Fensterimplementierungen liegt in der Auswahlmöglichkeit des Nutzer durch die Benutzung von *CheckButton*. Die Auswahl des Nutzer muss aus den Objekten gelesen werden und gesondert behandelt werden. Sollte keine Auswahl getroffen worden sein, dann wird eine Fehlermeldung ausgegeben. Wurde mindestens eine Ausgabeform ausgewählt, so wird für jede Ausgabeform die entsprechende Funktion ausgeführt. Hierfür werden die Diagramme und die reduzierte Anzahl der Diagramme mittels entsprechender Funktion aus dem Modul PTI erzeugt. Die Namensgebung und das Zielverzeichnis sind klar strukturiert. Der Rückgabewert der entsprechenden Funktion aus dem Modul PTI wird in die Dateien geschrieben. Sollte die ITRS Ausgabeform gewählt worden sein, so ist das Zielverzeichnis der *ITRS* Ordner und der Dateiname lautet *ITRS*_`Programmtransformation`.*txt*. Anstelle von `Programmtransformation` steht der Name der ausgewählten Programmtransformation. Der Inhalt der Datei ist dann das Resultat der `itrsOutput` Funktion aus dem PTI Modul. Analog gilt das Gleiche für die Textausgabe. Wenn die Diagramme gezeichnet werden sollen, dann wird die entsprechende Funktion aus dem Modul *DrawEngine* aufgerufen. Zudem wird die erzeugte Bilddatei an die Funktion zum Anzeigen der Bilder weitergegeben, damit das Bild direkt nach dem Erstellen dem Nutzer angezeigt werden kann. In allen Fällen werden Fehler durch Fehlermeldungen abgefangen und der Nutzer wird mittels Konsolen- und Statusnachrichten über den aktuellen Stand informiert.

```
onClicked createDiagButton $ do
    itrs      <- (toggleButtonGetActive itrsCheckButton)
    textformat <- (toggleButtonGetActive textdiagCheckButton)
    drawdiag   <- (toggleButtonGetActive drawdiagCheckButton)

    -- Check which checkbutton is active
    if ((not itrs) && (not textformat) && (not drawdiag))
    then do
            setStatusbar statusbar 1 "Please choose at least one option in order to create diagrams."
            putStrLn "Please choose at least one option in order to create diagrams."

    else
        do
            let diag = calc2Diag fpath transf
                genDiag = calc2GenDiag fpath transf
            in do
                if itrs then do
                -- ITRS option was selected , create itrs output for generalized diagrams and all diagrams (two files)
                CE.catch (do writeFile ("./DiagramFiles/ITRS/ITRS_"++transf++".txt") (itrsOutput diag)
                             writeFile ("./DiagramFiles/ITRS/ITRS_generalized_"++transf++".txt") (itrsOutput genDiag))
                         (\(CE.SomeException err) -> do setStatusbar statusbar 1 ("Failed to create diagrams in ITRS.")
                                                        putStrLn "Failed to create ITRS.")
                else do putStrLn "ITRS inaktive"

                if textformat then do
                -- Text option was selected , create text output for generalized diagrams and all diagrams (two files)
                CE.catch (do writeFile ("./DiagramFiles/Textformat/TXT_"++transf++".txt") (textOutput diag)
                             writeFile ("./DiagramFiles/Textformat/TXT_generalized_"++transf++".txt") (textOutput genDiag))
                         (\(CE.SomeException err) -> do setStatusbar statusbar 1 ("Failed to create diagrams in textformat.")
                                                        putStrLn "Failed to create diagrams in textformat.")
                else do putStrLn "textformat inaktive"

                if drawdiag then do
                -- Draw diagrams option was selected , generalized diagrams get printed on a png via DrawEngine
                CE.catch (do DrawEngine.drawDiagrams genDiag transf
                             openImageView ("./DiagramFiles/DiagramImages/Transformation_"++transf++".png") window statusbar
                             putStrLn "Generalized diagrams saved under: ./DiagramFiles/DiagramImages/")
                         (\(CE.SomeException err) -> do setStatusbar statusbar 1 ("Failed to draw diagrams.")
                                                        putStrLn "Failed to draw diagrams.")
                else do putStrLn "drawing diagrams inaktive"

                setStatusbar statusbar 0 ("Diagram files created in ./DiagramFiles. Store them elsewhere to avoid overwriting!")
                putStrLn "Diagram files created in ./DiagramFiles. Store them elsewhere to avoid overwriting!"
                widgetDestroy windowcd
```

4.5 Graphische Darstellung von Diagrammen

Das Zeichnen von Diagrammen und das Erzeugen von Bilddateien im PNG-Format wird mit Hilfe des Moduls *Diagrams* [DIAG] realisiert. Dieses Modul ermöglicht es komplexe Diagramme durch einfach Komposition einzelner Diagramme zu erzeugen. Zudem unterstützt das Modul durch eine Reihe von sogenannten *Backends* das Erzeugen von verschiedenen Bilddateiformaten. Um Bilddateien im PNG-Format zu erzeugen, nutzt das Modul *Diagrams.Backend.Cairo* [BCairo] eine Schnittstelle der Bibliothek *Cairo* [Cairo]. Diese stellt Methoden für das Zeichnen von zweidimensionalen Grafiken und für die Ausgabe dieser in verschiedenen Formaten zur Verfügung. Die Hauptfunktion zum Erzeugen einer Bilddatei ist `renderDia`.

```
renderDia :: Cairo -> Options Cairo R2 -> QDiagram Cairo R2 m -> (IO (), Render ())
```

Diese Erstellt aus den *CairoOptions* und aus dem Diagramm ein Tupel, dass aus einem direkten Schreibbefehl der Bilddatei und einer Aktion besteht, die zum Zeichnen des Diagramms auf ein *GtkObjekt* verwendet werden kann. Da nur die Bilddatei erzeugt werden soll, muss mittels `fst` die erste Aktion des Tupels entnommen werden, damit diese das Resultat der gesamten Funktion bilden kann. Die Funktion `drawDiagrams`, mit der ausgewählten Programmtransformation und der Liste der Diagrammen aus dem PT-Programm, wird zum Erzeugen der Bilddatei genutzt. Ausschnitte der Funktion sehen wie folgt aus:

```
drawDiagrams diag transf = fst (renderDia Cairo (CairoOptions ("./Diagram
```

`Die` CairoOptions bestehen aus dem Dateipfad der Zieldatei, der Seitenlänge des Bildes in Pixel, dem Dateiformat PNG und einem booleschen Wert, der das Überspringen einer Korrekturfunktion für Diagramme erlaubt.

```
(CairoOptions ("./DiagramFiles/DiagramImages/Transformation_"++transf++".png") (Width 2000) PNG False)
```

Das Zeichnen der Diagramme wird mit Hilfe der Funktion `createDiagrams` realisiert. Durch Komposition werden die einzelnen Diagramme zu einem Diagramm zusammengesetzt. Der Typ des Diagramms ist `Diagram b R2`. Dieser Typ ist ein Typsynonym für `QDiagram b v Any`. Dabei steht das Argument `v` für den Vektorraum. In diesem Fall ist `R2` der zweidimensionale euklidische Vektorraum. Das `b` repräsentiert den Backendtyp. Die Komposition der Diagramme wird im Modul *Diagrams* mit *Monioden* ermöglicht. Bei Monoiden handelt es sich um eine Menge von Elementen, die assoziativ verknüpft werden können und aus einem neutralen Element bestehen. Jedes Diagramm besteht aus einer *query*, in der jeder Punkt einem Wert zugewiesen wird. Mit Punkte sind die Punkte des Bildes im `R2` gemeint. Bei dem Typ `Any` sind die zugewiesenen Werte vom Standarttyp `Any`, der den Monoid mit logischem Operator *or* repräsentiert. Das neutrale Element ist der boolsche Wert *False*. In diesem Fall werden die Punkte der Diagramme so behandelt, dass zwischen den Punkten die innerhalb eines Diagramms sind und den Punkten die außerhalb eines Diagramms sind unterschieden wird [DiagUM].

4.5.1 Struktur der Diagramme

Bevor auf den Algorithmus zum Zeichnen der Diagramme eingegangen wird, muss die Struktur der Diagramme erläutert werden. Die Diagramme können zwei mögliche Formen haben. Eine Form ist die eines Rechtecks, bei dem das Diagramm aus zwei Gabeln besteht. Jede Gabel wiederum besteht aus einer Transformation und einer Standartreduktion:

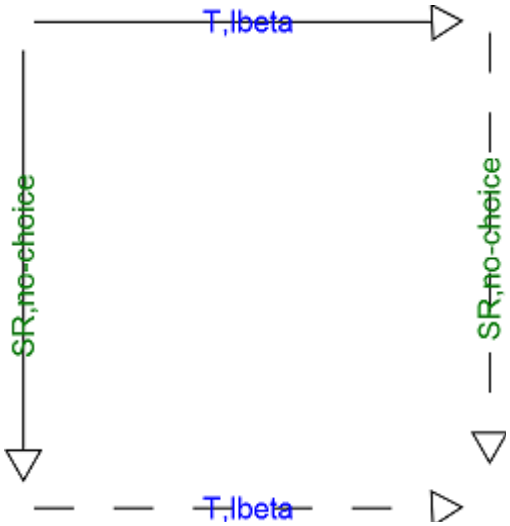


Abb. 4.5.1: Rechteckform eines Diagramms

Die zweite Form ist die ein Dreieck. In diesem Fall besteht der rechte Teil des Diagramms nur aus Standartreduktionen und schließt die linke Gabel des Diagramms direkt zu einem Dreieck:

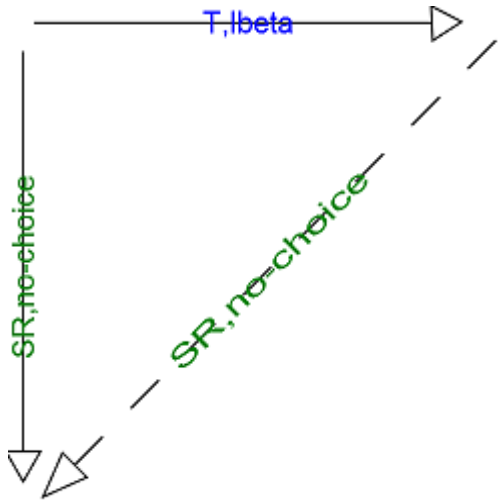


Abb. 4.5.2: Dreieckform eines Diagramms

Der Typ *Diagram* aus dem PT-Programm hat folgende Form:

```
data Diagram = [Item] :~~> [Item]
```

Der rechte und linke Teils des Diagramms sind in separaten Listen vorhanden. Dabei ist der linke Teil die linke Gabel und der rechte Teil eine schließende Folge von Standartreduktionen, wie im folgenden Beispiel zu sehen ist:

```
[SR (Rule "no-choice-1_1"),Trans (Rule "lbeta")] :~~> [Trans (Rule "lbeta"),SR (Rule "no-choice-1_1")]]
```

linker Teil:

```
[SR (Rule "no-choice-1_1"),Trans (Rule "lbeta")]
```

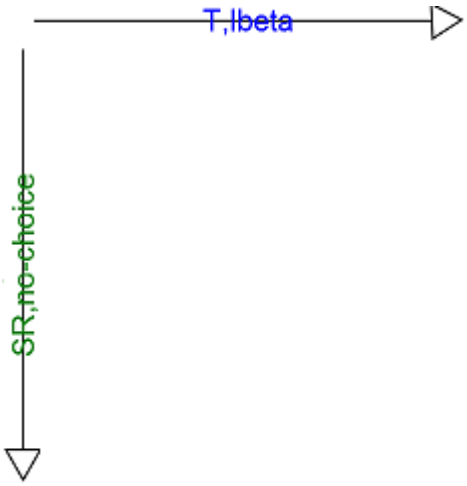


Abb. 4.5.3: Gabel eines Diagramms

rechter Teil:

```
[Trans (Rule "lbeta"),SR (Rule "no-choice-1_1")]]
```

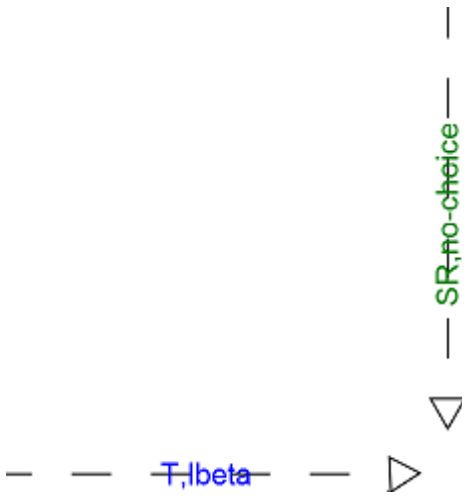


Abb. 4.5.4: Schließende Reduktionsfolge eines Diagramms

Sollte der rechte Teil des Diagramms nur aus Standartreduktionen bestehen, so entsteht eine Dreiecksform:

```
[SR (Rule "no-choice-l_1"),Trans (Rule "lbeta")] :~~> [SR (Rule "no-choice-l_1")]
```

linker Teil:

```
[SR (Rule "no-choice-l_1"),Trans (Rule "lbeta")]
```

rechter Teil:

```
[SR (Rule "no-choice-l_1")]
```

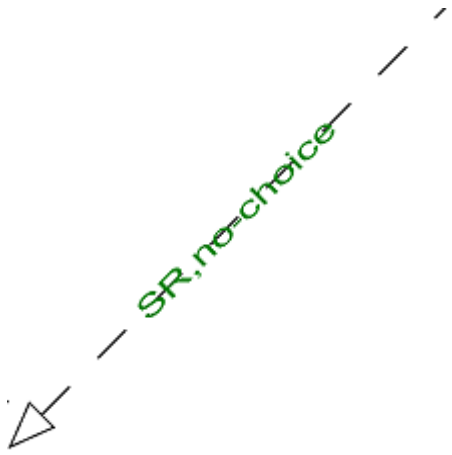


Abb. 4.5.5: Schießender Pfeil eines Diagramms

Dadurch das die Diagramme eine feste Struktur aufweisen, kann nun ein Algorithmus beschrieben werden, der diese Diagramme des Typs *Diagram* aus dem PT-Programm zeichnet. In den oben erwähnten Beispielen können die festen Strukturen erkannt werden. Der linke und rechte Teils des Diagramms muss zum Erstellen der Zeichnung nochmals getrennt betrachtet werden. Der linke Teil muss in zwei Unterlisten bestehend aus den Transformationen und Standartreduktionen geteilt werden. Das Gleiche muss mit dem rechten Teil des Diagramms gemacht werden. Dabei müssen die Unterlisten mit Standartreduktionen aus symmetrischen Gründen umgekehrt werden, da diese sonst eine falsche Reihenfolge aufweisen würden.:

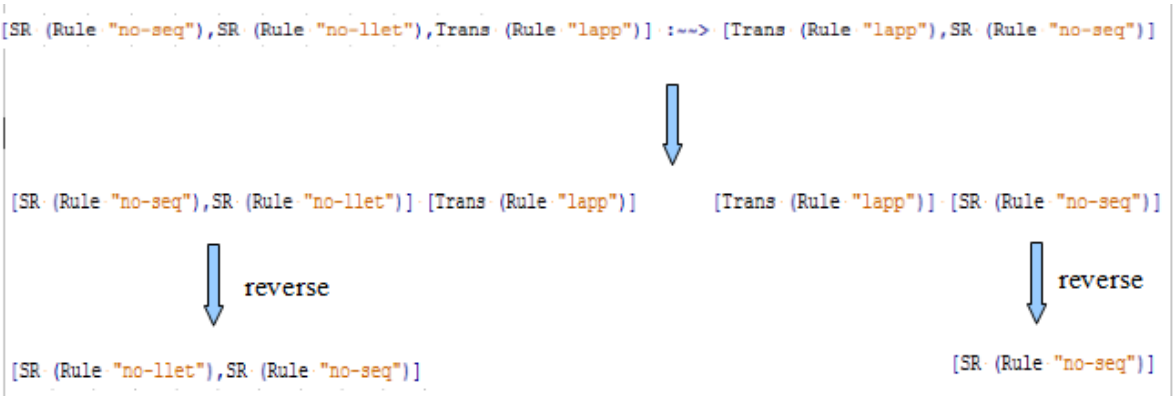


Abb. 4.5.6: Unterlisten des Diagramms

4.5.2 Zeichnen der Diagramme

Nachdem die einzelnen Komponenten des Diagramms extrahiert wurden, kann nun mit Hilfe der Funktionen aus *Diagrams*, jedes einzelne Element gezeichnet und zu einem einzigen großen Diagramm zusammengesetzt werden.

Zeichnen der Pfeile:

Die Pfeile mit Beschriftung die durchgezogen oder gestrichelt sind, werden von den Funktionen `trans`, `sr`, `transDashed` und `srDashed` gezeichnet. Dabei wird zwischen den Transformationen und Standartreduktionen unterschieden. Bei Transformationen handelt es sich um Pfeile, die horizontal sind. Bei Standartreduktionen um vertikale. Alle Pfeile bestehen aus einem Text, einem Strich und einem Dreieck. Der Strich und das Dreieck bilden zusammen den Pfeil. Der Text ist die Beschriftung.

```
sr name = ((text name # fontSize 0.3 # scale 0.5 # rotateBy (1/4) # fc green) |||
          centerY
            ((vrule 2 # lc black) ===
             (polygon with { polyType = PolyRegular 3 0.1} # rotateBy (1/6))))
```

Den Elementen kann man Attribute mitgeben, wie zum Beispiel die Textgröße bei einem Text oder die Rotation bei dem Polygon. Mit `centerY` kann man den aktuellen Ursprung des Diagramms an der `y`-Achse zentrieren. Mit den Funktionen `|||` und `===` kann man zwei Diagramme aneinander hängen. Im ersten Fall horizontal nebeneinander nach rechts und im zweiten Fall vertikal untereinander. Die Attribute die man einem Objekt geben kann, kann man in den meisten Fällen auch einem zusammengesetztem Objekt geben. Dies gilt unter anderem auch für die Rotation oder die Skalierung. Mit entsprechender Klammerung kann man beeinflussen, wie die Objekte zusammengesetzt werden sollen und in welcher Reihenfolge. Mit folgenden Funktionen werden die Unterlisten erstellt, die zum Zeichnen der Diagramme benötigt werden:

```
splitAtTrans itemList = splitAtTrans' itemList []
splitAtTrans' trs@((Diag.Trans _): is) srs = (trs,srs)
splitAtTrans' (i:is) srs = splitAtTrans' is (srs++[i])
```

Die Funktion `splitAtTrans` teil die Liste an dem Punkt, an dem zum ersten Mal eine Transformation gelesen wird. Die ersten Elemente bilden die Liste der Standartreduktionen und die Restlichen die Transformationen.

```
splitAtSR itemList = splitAtSR' itemList []
splitAtSR' srs@((Diag.SR _): is) trs = (srs,trs)
splitAtSR' (i:is) trs = splitAtSR' is (trs++[i])
```

Die Funktion `splitAtSR` teil die Liste an dem Punkt, an dem zum ersten Mal eine Standartreduktion gelesen wird. Der erste Teil ist die Liste der Transformationen und der Rest die Liste der Standartreduktionen. In beiden Funktionen werden die beiden Unterlisten in Form eines Tupels zurückgegeben. Hierbei gilt es zu beachten, dass die Listen im Tupel verdreht sind. Die Liste `srs` ist die Liste der Standartreduktionen und `trs` die Liste der Transformationen. Um eine ordentliche Beschriftung der Pfeile zu erhalten, muss der Name der jeweiligen Transformation oder Standartreduktion extrahiert und angepasst werden. Hierzu dienen die Funktionen:

```
fixTransName (Diag.Trans (Diag.Rule x)) = ("T,"++x)
fixSRName (Diag.SR (Diag.Rule x)) = ("SR,"++x)
```

Der Algorithmus sieht wie folgt aus und besteht aus den unten aufgeführten Funktionen. Dabei wird im ersten Teil des Algorithmus die Liste von Diagrammen in eine Liste von Unterlisten der Länge vier geteilt. Da der Typ der Diagramme `[Set.Set Diagram]` ist, muss dieser für die Verarbeitung angepasst werden. Hierfür wird das erste Element der Liste vom Typ `Set.Set Diagram` extrahiert. Das äußerste `Set` wird dann zu einer Liste gemacht. Die resultierende Liste `[[Set Diagrams]]` wird dann in Unterlisten der Länge vier geteilt.

```
(createDiagrams (Data.List.Split.Internals.chunksOf 4 (toList (head diag))) transf)
```

Dies ist notwendig, da die Diagramme Reihenweise untereinander gezeichnet werden sollen. Dabei besteht eine Reihe aus maximal vier Diagrammen.

```
createDiagrams [] transf = text ""
createDiagrams (x:xs) transf = createDiagRow x transf === createDiagrams xs transf

createDiagRow [] transf = text ""
createDiagRow (x:xs) transf = extract x transf ||| createDiagRow xs transf
```

Das Erzeugen einer Reihe von Diagrammen wird von der Funktion `createDiagRow` übernommen. Jeweils das erste Element der äußersten Liste wird an diese Funktion weitergegeben. Dabei handelt es sich um eine Liste der maximalen Länge vier, die eine Menge von Diagrammen enthält. Rekursiv werden dann diese Reihen der Länge vier mittels der `|||` Funktion untereinander gezeichnet. Die Funktion zum Zeichnen der Reihen ist ebenfalls rekursiv aufgebaut. Dabei wird statt vertikal, horizontal zusammengesetzt. In beiden Funktionen wird am Ende ein leeres Element gezeichnet, um den richtigen Rückgabetyp der Funktion zu erhalten. Dabei wird in beiden Fällen ein leeres Textfeld genutzt. Mit der Funktion `extract` werden die Diagramme des Typs `[Item] :~~> [Item]` verarbeitet.

```
extract (x Diag.:~~> y) transf =
  let l = (Data.List.map (fixTransName) (snd (splitAtSR y)))
      lt = Data.List.null l
      scaletr1 = (1/(fromIntegral (length (fst (splitAtTrans x)))))
      scaletr2 = (1/(fromIntegral (length (snd (splitAtSR y)))))
      scalesr1 = (1/(fromIntegral (length (snd (splitAtTrans x)))))
      scalesr2 = (1/(fromIntegral (length (fst (splitAtSR y)))))
  in createDiag
      (Data.List.map (trans) (Data.List.map (fixTransName) (fst (splitAtTrans x))))
      (Data.List.map (sr) (Data.List.reverse (Data.List.map (fixSRName) (snd (splitAtTrans x)))))
      (Data.List.map (transDashed) l)
      (Data.List.map (srDashed) (Data.List.reverse (Data.List.map (fixSRName) (fst (splitAtSR y)))))
      lt
      scaletr1
      scaletr2
      scalesr1
      scalesr2
```

Dabei werden die Unterlisten der linken und rechten Seite der Diagramme mit Hilfe der Funktionen `splitAtTrans` und `splitAtSr` erstellt. Zudem werden die Skalierungsfaktoren der Pfeile berechnet. Diese ergeben sich aus den jeweiligen Längen der Unterlisten. Dadurch wird gewährleistet, dass die Diagramme eine einheitliche Größe haben:

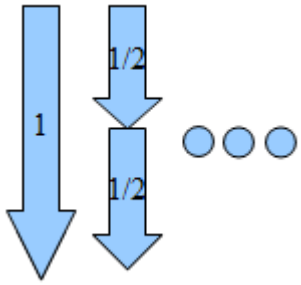


Abb. 4.5.7: Skalierung der Pfeile

Um die richtigen Argumente für die Funktion `createDiag` zu erzeugen, müssen die Elemente der Unterlisten angepasst werden. Dafür werden zu Beginn die Namen der Standartreduktionen und Transformationen extrahiert und angepasst. Danach wird die Reihenfolge der Listen mit Standartreduktionen umgekehrt, da diese sonst in der falschen Reihenfolge gezeichnet werden würden. Im Anschluss werden aus den angepassten und extrahierten Transformationsnamen und Standartreduktionsnamen die passenden Pfeilobjekte. Neben den Skalierungsfaktoren und den Listen mit Pfeilobjekten erwartet die Funktion `createDiag` einen boolschen Wert, der den Inhalt der Transformationen der rechten Seite des Diagramms prüft. Dies ist notwendig, weil es auch zu Dreiecksformen kommen kann. Siehe **Abb. 4.5.2**.

```
createDiag trans1List sr1List trans2List sr2List False scaletr1 scaletr2 scalesr1 scalesr2 =

((hcat $ Data.List.map (scaleX scaletr1) (Data.List.map (padX 1.05) trans1List)) # alignL
===
(vcat $ Data.List.map (scaleY scalesr1) (Data.List.map (padY 1.05) sr1List))
===
(hcat $ Data.List.map (scaleX scaletr2) (Data.List.map (padX 1.05) trans2List)) # alignL)
|||
(vcat $ Data.List.map (scaleY scalesr2) (Data.List.map (padY 1.05) sr2List)) # alignT
```

Wenn es sich um eine Rechteckform handelt, dann werden alle vier Listen verarbeitet. Dazu werden die Funktionen `padX` und `padY` verwendet. Mit diesen kann man die Grenzen eines Elements erweitern oder schrumpfen. Mit den Skalierungsfunktionen und den vorher berechneten Werten werden die Transformationen horizontal und die Standartreduktionen vertikal skaliert. Anschließend werden die Pfeilelemente mit Hilfe der Funktionen `hcat` und `vcat` horizontal und vertikal zu einem Element verbunden. Zum Schluss wird aus den resultierenden Elementen das vollständige Diagramm erzeugt. Hierfür wird die linke Reihe der Standartreduktionen unter die erste Reihe der Transformationen gezeichnet. Unter dieser wird dann die zweite Reiher der Transformationen gezeichnet. Die letzte Reihe der Standartreduktionen wird dann rechts an das entstandene Diagramm gesetzt. Mit Hilfe der `Align` Funktion können die Ursprünge der Objekte an andere Positionen der Elemente gesetzt werden. Zum Beispiel an das linke Ende eine Objekts. Im Falle einer Dreiecksform werden nur drei Reihen gezeichnet. Dabei wird die linke Gabel bestehend aus den ersten Transformationen und Standartreduktionen gebildet. Diese wird dann rotiert, so dass die offene Seite nach unten zeigt. Unter dieses Objekt wird dann die schließende Folge von Standartreduktionen gezeichnet. Da die Standartreduktionen aus vertikalen Pfeilen bestehen, müssen diese ebenfalls rotiert werden. Das resultierende Diagramm wird dann in seine ursprüngliche Lage rotiert. Dies ist beispielhaft in den folgenden Abbildungen zu sehen:

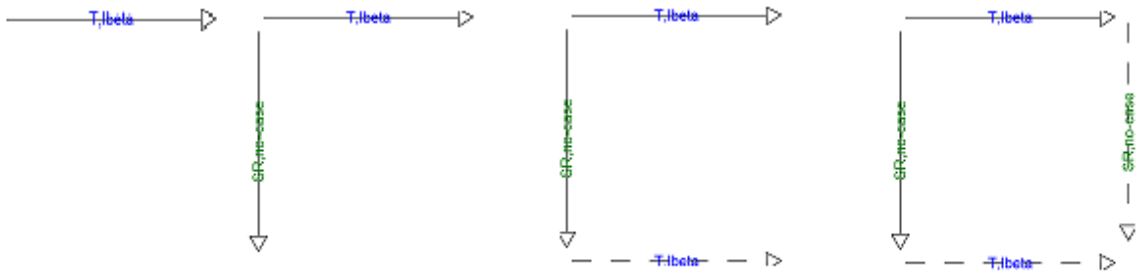


Abb. 4.5.8: Komposition eines Diagramms in Rechteckform

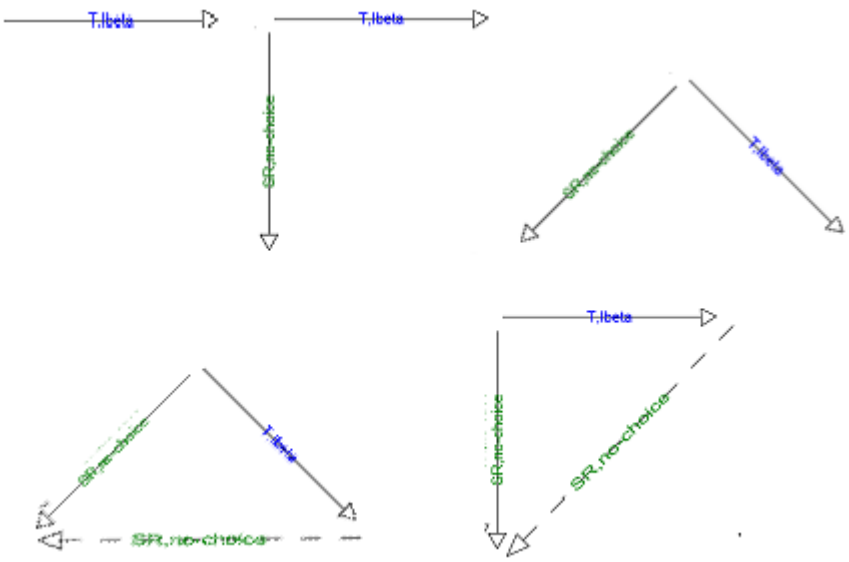


Abb. 4.5.9: Komposition eines Diagramms in Dreieckform

5. Fazit

Haskell und die in dieser Arbeit vorgestellten Werkzeuge ermöglichen es einen sehr überschaubaren und einfachen Code zu erzeugen. Sowohl Anpassungen als auch Änderungen des Codes konnten jeder Zeit einfach und präzise durchgeführt werden. Diese Eigenschaft hat im Verlauf der Entwicklung eine wichtige Rolle gespielt, da sich das PT-Programm noch in der Testphase befindet und kurzfristige Änderungen möglich sind. Auch hier konnte man die Einfachheit des Codes ausnutzen. Der simple und wartungsarme Code hat neben den vielen Vorteilen auch klare Nachteile. Ein Programm besteht aus Funktionen und Funktionsaufrufen die extrem Komplex werden können. Ab einer bestimmten Tiefe von Funktionsaufrufen kann es sehr schwer werden die gesamten Funktionszusammenhänge zu verstehen. Wenn man sich die Funktionen und Zusammenhänge aufzeichnet, würde das entstehende Bild immer größer und unverständlicher werden. Es erfordert sehr viel Konzentration und Zeit die Zusammenhänge zu verstehen. Ein weiterer Nachteil ist die Performance. Die sogenannte „faule Auswertung“ hat auch ihren Preis. Durch das Verschieben der Argumentauswertung bis zum Zeitpunkt an dem der Wert benötigt wird, leidet die Codegeschwindigkeit. Zusammenfassend lässt sich sagen, dass in diesem Fall die positiven Eigenschaften von Haskell die negativen Eigenschaften ganz klar in den Schatten gestellt haben. Für die Implementierung der Benutzerschnittstelle war es fundamental, dass der Code leicht angepasst werden kann. Der Performance Nachteil, der in zeitkritischen Programmen eine große Rolle spielt, war in diesem Fall vernachlässigbar. Es bleibt in diesem Fall die Komplexität der Funktionszusammenhänge, die in dieser Arbeit ausführlich erklärt wurden. Vor allem wenn man noch keine Erfahrung im Bereich der funktionalen Programmierung hat, kann die Umstellung der Art und Weise wie man Lösungsansätze formuliert viel Zeit beanspruchen. Diese Investition zahlt sich jedoch aus, da einem eine neues Werkzeug zur Verfügung gestellt wird, mit dem man eine große Anzahl an Problemen besser lösen kann.

6. Anhang

Im Anhang dieser Arbeit befindet sich eine CD/DVD mit dem Quellcode und dem Programm. Das Programm wird mittels *guiImpl.exe* gestartet. Eine kleine Anleitung zum Programm kann der *README.txt* entnommen werden.

Literaturverzeichnis

[APR] http://aprove.informatik.rwth-aachen.de, abgerufen am 12. April 2013

[AKVP] RAU C. , SABEL D. , SCHMIDT-SCHAUSS M.: *Correctness of program transformations as a termination problem* In GRAMLICH B. , MILLER D. ,SATTLER U.: *Automated Reasoning - Proceedings of the 6th International Joint Conference* - IJCAR 2012 - Manchester - UK, 26-29. Juni 2012 - *Volume 7364 of Lecture Notes in Computer Science*, Seiten 462-476. Springer Berlin / Heidelberg Juni 2012, Springer Verlag, http://www.ki.informatik.uni-frankfurt.de/papers/rau/auto-induct.pdf ,abgerufen am 15. April 2013

[ATPD2006] GIESL, J. , SCHNEIDER-KAMP, P. , THIEMANN, R.: *Automatic termination proofs in the dependency pair framework* - IJCAR 2006 *Volume 4130* von LNCS, Springer

[BCairo] http://projects.haskell.org/diagrams/doc/Diagrams-Backend-Cairo.html, abgerufen am 25. April 2013

[Cairo] http://cairographics.org/, abgerufen am 25. April 2013

[DFGP] http://www.ki.informatik.uni-frankfurt.de/research/dfg-diagram/en/, abgerufen am 15. April 2013

[DIAG] http://projects.haskell.org/diagrams/, abgerufen am 25. April 2013

[DiagUM] http://projects.haskell.org/diagrams/manual/diagrams-manual.html, abggerufen am 25. April 2013

[EFP] SCHMIDT-SCHAUSS, M. , SABEL D.: *Einführung in die Funktionale Programmierung (Vorlesungsskript)* Dezember 2011, http://www.ki.informatik.uni-frankfurt.de/lehre/WS2011/EFP/skript/skript.pdf, abgerufen am 10. April 2013

[ERER2001] WIGER U.: *Four-fold Increase in Productivity and Quality – Industrial -Strength Fuctional Programming in Telecom-Class Products* / FemSYS 2001 Development on distributed architectures Revision C, http://www.erlang.se/publications/Ulf_Wiger.pdf, abgerufen am 22. April 2013

[GLADE] http://glade.gnome.org/, abgerufen am 24. April 2013

[GTK] http://www.gtk.org/, abgerufen am 24. April 2013

[GTKHie] https://developer.gnome.org/gtk-tutorial/2.90/x477.html, abgerufen am 24. April 2013

[GTK2HS] http://projects.haskell.org/gtk2hs/, abgerufen am 24. April 2013

[HR2010] MARLOW, S. (editor): *Haskell 2010 language report*, http://www.haskell.org/onlinereport/haskell2010/, abgerufen am 15. April 2013

[SNSA2008] SCHMIDT-SCHAUSS M. , SCHÜTZ M. , SABEL D.: *Safety of Nöcker's strictness analysis – J. Funct. Programming*, 18(04):501-553, 2008 , Cambrige University Press, http://www.ki.informatik.uni-frankfurt.de/papers/schauss/JFP_Schmidt-Schauss.pdf, abgerufen am 15. April 2013

[TRAT1998] BAADER F. , NIPKOW T.: *Term Rewriting and All That*, ISBN-13: 978-0521779203 Cambridge University Press, New York, NY, USA, 1998.