Vom Nutzen funktionaler Datenstrukturen zum Entwurf ausführbarer Netzlistenbeschreibungen

Dissertation

der Mathematisch-Naturwissenschaftlichen Fakultät
der Eberhard Karls Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

vorgelegt von
Matthias Brettschneider
aus Haltern am See

Tübingen
2015

Gedruckt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der Eberhard Karls Universität Tübingen.

Tag der mündlichen Qualifikation:	16.02.2016
Dekan:	Prof. Dr. Wolfgang Rosenstiel
1. Berichterstatter:	Prof. Dr. Wolfgang Rosenstiel
2. Berichterstatter:	Prof. Dr. Tobias Häberlein

Ein altes Gleichnis besagt, das wir Zwerge auf den Schultern von Riesen seien. Ich möchte mich hiermit bei all Jenen bedanken, die mir mit Ihren sprichwörtlichen Schultern diese Arbeit ermöglicht haben. Ganz besonders danke ich Prof. Häberlein für die jahrelangen wissenschaftlichen Inspirationen. Vor allem aber danke ich meiner tollen Frau Mona und meiner lieben Mutter Heike für die Unterstützung, die Kritik, das Lob und das Verständnis über die Jahre.

Vom Nutzen funktionaler Datenstrukturen zum Entwurf ausführbarer Netzlistenbeschreibungen

Inhaltsverzeichnis

1. Einleitung

1.1. Motivation und Vorgehen

Der Fokus dieser Arbeit liegt auf Netzlisten, die im Späteren eingeführt werden. Bisherige Ansätze haben versucht, eine funktionale Hardwarebeschreibungssprache zu entwerfen (Wirtssprache), die eine bestehende Sprache (Gastsprache) erzeugt. Die Gastsprache entspricht der Hardwarebeschreibungssprache. Beim Abbilden einer Gastsprache, die ebenso ausdrucksstark ist wie ihre Wirtssprache, muss bei der Einbettung immer ein Kompromiss zwischen tiefer und seichter Einbettung getroffen werden. Wird die Gastsprache tief Eingebettet, so wird ihre Ausdrucksstärke auf jene der Wirtssprache eingeschränkt. Dies rührt daher, da für jedes Konzept der Gastsprache eine Entsprechung in der Wirtssprache existieren muss. Im Falle sichter Einbettung wird die Ausdrucksstärke der Gastsprache nicht durch fehlende Konzepte beschränkt, sondern durch die Schwierigkeit, diese Konzepte der Gastsprache allein mit den Mitteln der Wirtssprache zu beschreiben. Die Probleme der Einbettung bestehen auch dann noch, wenn beide Sprachen für sich genommen gleich ausdrucksstark sind.

In der Arbeit wird gezeigt, dass bei der funktionalen Beschreibung von Hardware die Funktion eingeschränkt wird, um so den Features eines Hardwarebausteines zu entsprechen. Dies wird in dieser Arbeit durch Anwendung des Konzeptes der Arrows geschehen. Arrows [50, 32] führen das Prinzip der Abstraktion fort, indem sie Funktionen abstrahieren. Arrows sind die Umsetzung „premonoidaler Effekt-Kategorien“ [51, 7] in der funktionalen Sprache Haskell, die auch unter dem Namen „Freyd Kategorien“[22] das Prinzip ausführbarer Semantiken innerhalb der Kategorientheorie [65] abbilden. Dieses Prinzip wird als higher-Order-Typklasse [28] realisiert, die den Funktionspfeil `→` abstrahiert. Dieser Funktionspfeil stellt die Abbildung einer Eingabe vom Typ `b` auf die durch die Funktion beschriebene Ausgabe vom Typ `c` dar: `b → c`. Mithilfe der Arrows wird der Funktionspfeil zu `a` abstrahiert:

`b ‘a‘ c`, bzw. prefix: `a b c`.

Die Arrow-Typklasse ist für alle `a`, die eine Kategorie sind, definiert. Eine Kategorie definiert die Identität `id` sowie die Funktionskomposition `>>>`. Der Quelltext spiegelt diese Abhängigkeit in der ersten Zeile wider, die Ausdrücke vor dem `=>`-Pfeil geben an, zu welchen Typklassen `a` gehören muss, bevor `a` zum Arrow werden kann.

```
1 class (Category a) => Arrow a where
2   arr    :: (b -> c) -> a b c
3   first  :: a b c   -> a (b, d) (c, d)
4   second :: a b c   -> a (d, b) (d, c)
5   (***)  :: a b1 c1 -> a b2 c2 -> a (b1, b2) (c1, c2)
6   (&&&)  :: a b c1  -> a b c2  -> a b (c1, c2)
```

Der Typparameter `a` stellt die Abstraktion dar und kann durch Instanziierung festgelegt werden. Im Fall, dass `a` mit dem Funktionspfeil instanziiert wird, entstehen Funktionen mit einer Datenflusssicht. Eine Stream-Instanziierung lässt die Funktion auf Datenströmen agieren und kann dabei für die Berechnung vorhergehende bzw. nachfolgende Daten betrachten. Letztlich stellt der Arrow hier die Möglichkeit der Metaprogrammierung durch geschickte Instanzbildung dar. Metaprogrammierung ist eine Methode, bei der in einer Sprache Zugriff auf den eigenen abstrakten Syntaxbaum besteht. Damit kann die Sprache genutzt werden, um ihren eigenen Syntaxbaum zu verändern. Die erzeugten Konstrukte sind nicht notwendigerweise gleichmächtig wie jene, die sie erzeugen. In diesem Fall spricht man auch von Multi-Level-Sprachen [42], da hier eine Hierarchie zwischen den beiden Sprachen besteht.

Die in dieser Arbeit entworfene Semantik der Netzlistenbeschreibungen ist an die formale Semantik von Programmiersprachen-Beschreibungen angelehnt. Diese abstrakte Sicht ist gerade für den funktionalen Entwurf von Netzlistenbeschreibungen nützlich, da Funktionen Hardwarebausteine nicht direkt abbilden können; Funktionen stellen im Gegensatz zu Hardwarebausteinen ein sehr viel mächtigeres Konzept dar. Es ist daher notwendig, die Funktion derart einzuschränken, dass sie genau die Features eines Hardwarebausteines abbilden kann.

Der einfache Gedanke, Hardware mit funktionalen Mitteln zu beschreiben, ist schon seit den 1980er Jahren vorhanden und reicht von seicht eingebetteten Lösungen wie Lava [5, 61, 62, 63, 3] hin zu tief eingebetteten Lösungen wie ForSyDe [54, 53]. Dabei haben sich bisherige Lösungen darauf beschränkt, Funktionen als Mittel zur

Erzeugung von Hardwarebeschreibungscodes einzusetzen. Der Ansatz, Hardware direkt über Funktionen zu beschreiben, führt dabei zu Problemen wie dem Sharing-Problem [17]. Eine Funktion ähnelt einem Hardwarebaustein nur bis zu einem Punkt. Darüber hinaus besitzen gerade die Funktionen des funktionalen Paradigmas eine ganze Reihe von Eigenschaften, die sich nicht auf den Hardwarebaustein übertragen lassen. Zu diesem Feature-Space gehört das Currying, bei dem eine Funktion mit Parametern unterversorgt wird. Eine solche Funktion liefert keinen Ergebniswert, sondern eine Funktion, welche die restlichen Parameter erwartet. Dies zeigt auch gleich ein weiteres Feature funktionaler Funktionen, das keine Entsprechung in der Hardware hat. Funktionen sind „First Class Citizens“ und können damit als Parameter einer Funktion erwartet werden oder als Rückgabewert zurückgeliefert werden.

In dieser Arbeit wird gezeigt, dass der Entwurf von Hardware mit abstrahierten Funktionen prinzipiell möglich ist und dabei ohne die Probleme vorhergehender Lösungen auskommt. Denn mithilfe einer zusätzlichen Abstraktionsebene lässt sich der Feature-Space von Funktionen derart einschränken, dass er direkt dem der Hardwarebausteine entspricht. Die zusätzliche Abstraktionsebene wird durch den Einsatz von higher-order-Typklassen bzw. konkret durch Arrows erzielt. Beispiel: Der Arrow `aXor >>> aNot` gibt zunächst nur die Verknüpfung des „exklusive-oder“-Gliedes mit dem „not“-Glied an. Die Bedeutung der Verknüpfung wird über die Instanziierung der higher-order-Typklasse erzielt. Diese kann dabei als Funktion, als Stream-Processor oder als Diagramm instanziiert werden.

Die Trennung von Beschreibung und Bedeutung lässt zusätzlich auch den notwendigen Raum, um einen guten Kompromiss zwischen tiefer und seichter Einbettung zu erreichen. Die vorliegende Arbeit lotet den Designspace aus und zeigt, wie weit diese Methode betrieben werden kann. Außerdem zeigt die vorliegende Arbeit, dass die Arrow-Kombinatoren eine direkte Entsprechung in den impliziten Kombinatoren von Schaltplänen haben. Die Schreibweise `aXor >>> aNot` ist dabei nichts anderes als die textuelle Darstellung der im Diagramm 1.1 gezeigten Schaltung.

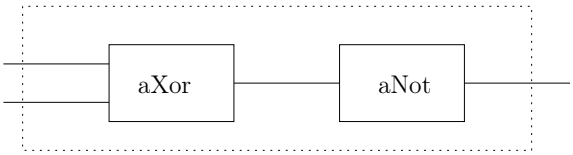

Abbildung 1.1.: Kombination von „xor“ und „not“

Der Begriff Hardware beschreibt in der vorliegenden Arbeit eine Schaltung im elektronischen Sinne. Es wird in der Arbeit nur Hardware betrachtet, die als Netzliste darstellbar ist. Der Begriff Netzlisten wird im Abschnitt 2 eingeführt und detailliert beschrieben. Synonym zum Begriff Hardware werden in der vorliegenden Arbeit die Begriffe Baustein oder Komponente verwendet.

Anhand eines digitalen Filters wird der Entwurf einer Netzliste exemplarisch mit Claessens [5] Lava-Ansatz gezeigt.

Rein funktionale Datenkonstrukte besitzen die Eigenschaft der Seiteneffektfreiheit, auf die im Abschnitt 5.1 genauer eingegangen wird. Der Nutzen des Erhalts der Seiteneffektfreiheit wird besonders deutlich an Algorithmen, die zu ihrer Beschreibung keine Seiteneffekte benötigen. Am Beispiel eines kryptographischen Kerns wurde im Paper „Crypto-Core Design using Functional Programming“ [27] ein Ansatz mit der Methode Sanders' [53] dargestellt.

An dieser Stelle unterscheidet sich der vorliegende Ansatz von den bisherigen Arbeiten. Durch die Beschränkung auf die Erzeugung von Netzlisten wird sichergestellt, dass die Gastsprache eine Untermenge der Wirtssprache ist. Außerdem erlaubt der in Kapitel 5 vorgestellte Datentyp einen wählbaren Mix aus tiefer und seichter Einbettung der Gastsprache in die Wirtssprache. Dies wird erreicht, indem mithilfe der hier vorgestellten Lösung Bausteine beliebiger Granularität definiert werden können. Trotzdem wird garantiert, dass diese Bausteine algebraisch miteinander kombinierbar bleiben. Kapitel 2 konkretisiert die unterschiedlichen Einbettungsformen.

Eine Konsequenz aus dem hier vorgestelltem Ansatz ist, dass verwandte Methoden zur Netzlistenbeschreibung, wie Diagramme, herangezogen werden können. Kapitel 2 erläutert diese Zusammenhänge genauer. Hier existiert ebenfalls ein Unterschied zu schon bestehenden Lösungen, denn der Entwurf einer Netzliste mit Arrows erlaubt es, diese Netzliste zu nutzen, um weitere Darstellungsformen zu erzeugen. Exemplarisch wurden diese am Beispiel von UML-Aktivitätendiagrammen in dem Paper [12] dargestellt. Neben Diagrammen können auch weitere HDLs als Darstellungsform für eine Netzliste herangezogen werden. Dies wurde in einem Paper [13] anhand des Entwurfes eines kryptographischen Algorithmus dargelegt. Das Paper zeigt die Grenzen der vorgestellten Lösung auf und gibt Hinweise, wie diesen Grenzen begegnet werden kann. Im Kapitel 8 wird dieser Zusammenhang detailliert beschrieben.

1.2. Bisherige Arbeiten zur funktionalen Hardwaremodellierung

Die Idee, Hardware mit funktionalen Mitteln zu entwickeln, kursiert in der funktionalen Community schon länger. Bis heute gibt es die unterschiedlichsten Ansätze. Die Anfänge sind in den frühen 1980er Jahren zu finden. Mary Sheeran entwarf damals die funktionale Hardwarebeschreibungssprache muFP [59]. Während derselben Dekade präsentierte John O'Donnell seinen Ansatz, HDRE [44, 47], ebenfalls eine funktionale Hardwarebeschreibungssprache. Beide Hardwarebeschreibungssprachen legten den Grundstein für zahlreiche weitere Ansätze. Davon basieren einige direkt auf den genannten, andere sind davon inspiriert.

Studenten um Mary Sheeran, darunter Coen Claessen, haben 1998 eine monadische Lösung, „Lava“ [5], vorgestellt. Dabei handelt es sich um seichte Einbettung (siehe Abschnitt 5.3) der Sprache. Lava ist in die Wirtssprache Haskell eingebettet und lässt sich zum Generieren von VHDL-Code verwenden. Die Probleme von Lava liegen in der mehrfachen Verwendung von Bauteilen, dem sogenannten Sharing-Problem [17].

Im Jahr 2001 stellte O'Donnell einen weiteren Ansatz namens Hydra [48, 46, 45] vor. Diese Sprache ist ebenfalls in Haskell eingebettet. Hydra wird nicht mehr weiterentwickelt und die Dokumentation ist unfertig.

Ingo Sanders [54, 53] stellte im Jahr 2003 den Ansatz ForSyDe vor. Mit dieser Methode baut er auf der Metaprogrammiertechnik Template Haskell [58] auf. ForSyDe hat gezeigt, dass der Metaprogrammieransatz mit Template Haskell möglich ist. Abschnitt 5.3 dieser Arbeit zeigt, dass der Arrow-Ansatz ebenfalls einen Metaprogrammieransatz darstellt. Der Abschnitt verdeutlicht außerdem, warum Metaprogrammierung für den Hardwareentwurf von Nutzen ist.

Es existieren funktionale Sprachen, die ausschließlich für den Zweck des Hardwareentwurfs definiert wurden. Dazu zählt SAFL [1] [56, 41]. Um aus SAFL Code eine Netzliste zu generieren, wird der FLaSH Compiler [40] benötigt. Dieser Compiler ist auf definierte Klassen von Schaltkreisen beschränkt. Auch haben die Autoren von SAFL/FLaSH angemerkt, dass SAFL keine ideale Entwicklungssprache ist. Zudem haben sie angegeben, dass es starke Probleme mit dem Ein- und Ausgabesystem gibt [57].

[1] Statically Allocated Functional Language

Ein weiterer Kandidat einer funktionalen Sprache, die auf Metaprogrammiertechniken setzt, ist *reFLect*[25]. Diese Sprache wird bei Intel entwickelt. *reFLect* ist auf Hardwaredesign und Theorembeweise spezialisiert.

Cryptol [23] ist ebenfalls eine funktionale Hardwarebeschreibungssprache. Cryptol fällt aus dem Rahmen, da sie speziell zum Definieren kryptographischer Hardware gedacht ist. Die Sprache wurde ursprünglich bei Galois [2] entworfen, um kryptographische Algorithmen zu testen und daraus FPGA-Implementierungen [24] zu generieren.

2011 wurde von Adam Megacz die Nähe zwischen Arrows und Metaprogrammiertechniken - im speziellen zu Multi-Level-Sprachen - dargestellt [39]. In dem Paper stellte Megacz „generalisierte Arrows“ vor und zeigt, dass Generalized-Arrows Multi-Level-Sprachen entsprechen.

1.3. Wissenschaftliche Beiträge

Das Spektrum der Möglichkeiten, Hardware funktional zu beschreiben, wurde in zahlreichen Papern aufgespannt. Dabei wurden auf der einen Seite Techniken der seichten Einbettung und auf der anderen Seite Techniken der tiefen Einbettung anhand von Beispielen untersucht.

In einem ersten Paper „Von funktionalen Programmen zur Hardwarebeschreibung digitaler Filter“ [11] wurde der Hardwareentwurf mit der seicht eingebetteten Methode Lava [5, 61, 62] durchgeführt. Am Beispiel digitaler Filter wurde diese Methode untersucht, sie wird im Abschnitt 4 vorgestellt. Schon anhand eines einfachen Filters lassen sich die Möglichkeiten, aber auch die Grenzen dieser Technik darlegen.

Im nächsten Paper, „Crypto-Core Design Using Functional Programming“ [27], wurde dann erörtert, wie mithilfe von Meta-Programmierung Hardware beschrieben werden kann. Es wurden kryptographische Algorithmen mit ForSyDe [54, 53], einer Technik tiefer Einbettung, beschrieben. Auch hier lassen sich anhand der Beispiele Möglichkeiten und Probleme dieser Technik zeigen.

Das Paper „Functional Abstractions for UML Activity Diagrams“ [12] zeigt auf, dass sich Arrows allgemein eignen, um UML-Aktivitätsdiagramme funktional dar-

[2] galois Corporate – http://corp.galois.com

zustellen. UML-Aktivitätsdiagramme wurden bereits an anderer Stelle [21] zur Hardwarebeschreibung eingesetzt. Hier wurde außerdem gezeigt, warum es hilfreich ist, seiteneffektfreie Teile im Entwurfsprozess gesondert zu modellieren. Es ist sinnvoll, wenn der seiteneffektfreie Anteil auch in einer seiteneffektfreien Sprache modelliert wird, da so die Erhaltung dieser Eigenschaft garantiert wird.

Da allgemeine Funktionen zu mächtig sind, um Hardwarebausteine beschreiben zu können, müssen diese beschränkt werden. Diese Einschränkung lässt sich durch Arrows erzielen und wurde im Paper „Using Haskell Arrows as Executable Hardware Specification Language“ [13] dargelegt. Das Paper bedient sich erneut kryptographischer Algorithmen als Modellierungsbeispiele. Die vorgestellte Methode wird dazu gebraucht, Netzlistenbeschreibungen aus den Modellen generieren zu können.

Wie Netzlistenbeschreibungen aus formalen Arrow-Beschreibungen generiert werden, zeigt das Paper „From Arrows to Netlists Describing Hardware“ [14]. In diesem Paper wird der Fokus auf die Überführung gewöhnlicher Arrows hin zu Arrows mit Vektoren als strukturierenden Datentyp gelegt. Dabei wird auch ausgebreitet, wie eine solche Beschreibung genutzt werden kann, um formal mit Hardware umgehen zu können.

1.4. Vorgehen

Für diese Arbeit werden Begriffe spezifiziert, da Begriffe wie „Hardware“ unterschiedliche Bedeutungen besitzen. So wird Missverständnissen aufgrund unscharfer Begriffe vorgebeugt. Im nächsten Schritt wird definiert, was als Netzliste verstanden wird und warum Hardware aus der Netzlistenperspektive gesehen wird. Es wird beschrieben, warum die funktionale Programmierung als Werkzeug verwendet wird. Es werden Eigenschaften von Algorithmen erläutert, die mithilfe der funktionalen Programmierung erhalten werden können. Da der Gedanke, funktionale Programmierung und Hardwareentwurf zusammenzubringen, in der Geschichte schon vorgekommen ist, werden die einschlägigen Arbeiten beschrieben, die diesen Gedanken verfolgt haben. Die unterschiedlichen Ansätze der Vergangenheit werden erläutert. Es wird die Idee des jeweiligen Ansatzes erklärt und die Meinung des Autors begründet, warum der Ansatz nicht weiter verfolgt werden kann.

Kapitel 3 geht vertiefender auf die funktionale Programmierung ein. Es werden die

Begriffe festgelegt, die Anwendung finden, und es wird beschrieben, was unter funktionaler Programmierung zu verstehen ist. Im Weiteren wird beschrieben, was die funktionale Programmiersprache leisten muss, damit sie für die Lösung von Nutzen ist. Im Kapitel 3 werden die funktionalen Vorgehensweisen und Methodiken anhand kurzer Beispiele erläutert. Dabei werden in dem Kapitel jene Methodiken beschrieben, die später bei der Lösung Einsatz finden.

Um eine geeignete Datenstrukur verwenden zu können, werden in Kapitel 2 unterschiedliche Betrachtungsweisen der Hardware beschrieben. Daraufhin werden diese unter allgemeineren Gesichtspunkten betrachtet. Daraus wird auf eine fundamentale, unterliegende Struktur geschlossen. Diese Struktur muss sich in den Datenstrukuren widerspiegeln.

Es wird dargestellt, dass sich eine zugrundeliegende Datenstruktur angeben lässt. Anhand eines Beispieles wird erleutert, wie sich die unterliegende Datenstruktur auch in der physikalischen Realität der Hardware wiederfinden lässt.

Kapitel 5 gibt an, mit welcher funktionalen Datenstruktur die Struktur der Hardware abgebildet wird. Es wird eine Methode vorgestellt, die Anwendung bei der Erzeugung dieser Struktur findet. Die Datenstruktur wird eingeführt und definiert, orientiert an der Geschichte der Struktur. Darauf folgt die Einordnung, wie diese Struktur genutzt wird, um die vorgestellten Lösungen zu unterstützen. Gezeigt wird ebenfalls, wie diese Datenstruktur der Anforderung „Ausführung“ nachkommen kann. Es werden unterschiedliche Schreibweisen der Datenstruktur vorgestellt und an Beispielen erläutert. Die Datenstruktur wird auf Probleme abgeklopft, die dann beschrieben werden. Es wird gezeigt, wie sich die Probleme auswirken und wie trotz dieser Probleme gearbeitet werden kann.

Im nächsten Schritt wird dargestellt, wie die vorgestellte Datenstruktur konkret zur Hardwarebeschreibung angewandt wird. Dabei werden mit den vorgestellten Methoden Beispielanwendungen modelliert. Es werden Erweiterungsmöglichkeiten vorgestellt und beispielhaft eingebaut. Außerdem wird anhand der Beispiele erläutert, ob die Struktur geeignet ist, die gestellten Probleme der Ausgabe und der Ausführbarkeit zu lösen. Es wird gezeigt, wie eine Bibliothek zur Hardwarebeschreibung mithilfe der funktionalen Datenstruktur aufgebaut wird. Außerdem werden konkrete Optimierungsmöglichkeiten der erzeugten Strukturen vorgestellt.

Kapitel 8 gibt Verbesserungsmöglichkeiten der vorgestellten Lösung an. Dabei wird

gezeigt, wie eine Verbesserung aussehen sollte. Es werden verschiedene Lösungsansätze vorgestellt. Es wird geprüft, ob diese Lösungen praktikabel sind.

Kapitel 9 zieht das Fazit aus der vorgestellten Arbeit und regt sinnvollen weiteren Forschungsbedarf an.

2. Unterschiedliche Betrachtungen von Netzlisten

Die im Folgenden dargestellten Diagrammarten wurden exemplarisch ausgewählt, da sie Anwendung beim Hardwareentwurf finden. Der Hardwareentwurf stützt sich nicht zwangsläufig auf diese Diagramme, aber die Möglichkeit dazu besteht. Es wurden jene Diagrammarten gewählt, die eine gemeinsame allgemeine Struktur teilen. Diese wird in Abschnitt 2.4 identifiziert.

Eine Netzliste ist eine allgemeine Darstellung von Hardware. Die Definition neuer Hardwarekomponenten wird aus bestehenden Komponenten aufgebaut. Das liegt daran, dass Hardware aus digitalen Bausteinen aufgebaut wird und die Anzahl der Komponenten endlich ist. Eine Netzliste beschreibt, aus welchen Hardwarebausteinen die neue Hardwarekomponente aufgebaut ist. Ein Hauptbestandteil einer Netzliste ist die Beschreibung der Verknüpfungen innerhalb der Netzliste. So wird aus bestehenden Elementen eine neue Hardwarestruktur erzeugt.

2.1. Schaltpläne

Eine Schaltung ist eine Zeichnung, die darstellt, wie eine Elektronik aufgebaut ist. Es gibt unterschiedliche Bauteile. Bauteile besitzen Schaltzeichen. Die Anzahl der verschiedenen Bauteile ist nicht begrenzt. Es gibt Zeichen für grundlegende Bauteile wie Transistoren, Widerstände oder Dioden. Es ist nicht möglich, allen Bauteilen ein eigenes Schaltzeichen zuzuordnen. Für integrierte Schaltungen werden einfache Rechtecke verwendet, in denen textuell die Bezeichnung des Bauteiles steht. Die einzelnen Schaltzeichen sind für die Schaltung letztlich nicht von Bedeutung. Ein Schaltplan lässt sich auch ausschließlich aus bezeichneten Rechtecken aufbauen. Wichtig sind außerdem die Verbindungen der Bauteile untereinander. Jeder Schalt-

plan ist aus Bauteilen und ihren Verknüpfungen aufgebaut. Das ist nur möglich, wenn jedes Bauteil eine definierte Anzahl an Ein- und Ausgängen besitzt. Nur die Ein- und Ausgänge werden über Linien miteinander verbunden.

Ein Schaltplan ist eine abstrakte Darstellung aller elektronischen Komponenten und ihrer Verbindungen. Diese Darstellung lässt sich direkt physikalisch nachbauen.

In Abbildung 2.1 können die Bausteine einer einfachen Schaltung identifiziert werden. Die Bauteile in der Abbildung sind anhand der Symbole zu erkennen. Außerdem sind sie benannt. Symbole können auch vollständig durch beschriftete Rechtecke ersetzt werden. Auf die durch Linien symbolisierten Verbindungen kann nicht verzichtet werden. Da Bauteile verschiedene Punkte kennen, an denen Verbindungen anknüpfen können, sind auch die Punkte Bestandteil des Schaltplanes. Hier wird zwischen Ein- oder Ausgabepunkten eines Bauteiles unterschieden.

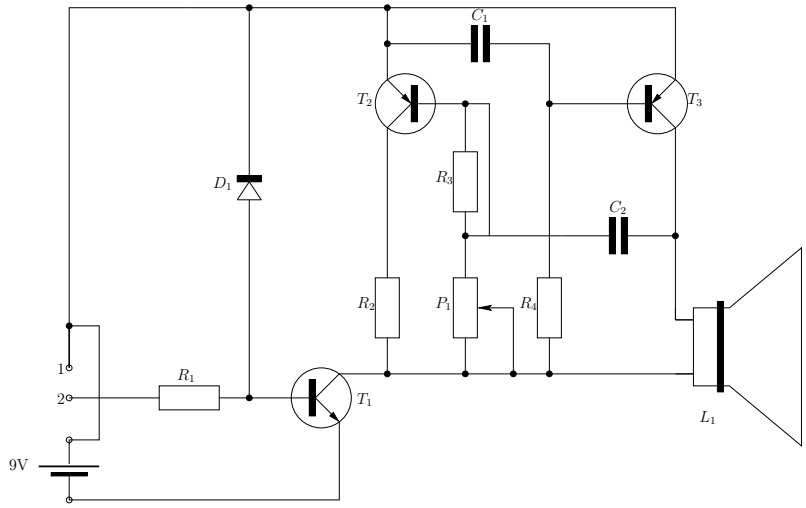

Abbildung 2.1.: Symbolbild einer einfachen Schaltung

2.2. Datenflussdiagramme

Hardware lässt sich auch aus Datenfluss-Sicht betrachten. Dabei gibt es Eingaben in die Schaltung hinein, sogenannte Quellen, und Ausgaben in sogenannte Senken. Ein Reset-Knopf stellt eine Eingabe dar; die Darstellung mit Strom/kein Strom ist eine Ausgabe.

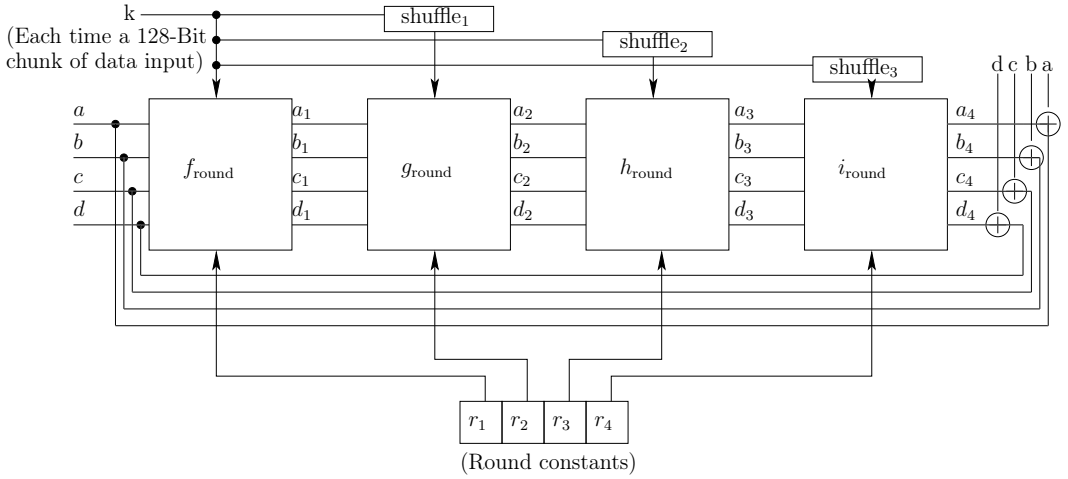

Abbildung 2.2.: Datenfluss einer MD5-Runde

Hier werden nicht elektronische Signale verfolgt, sondern Daten. Dabei ist das Datenflussdiagramm eine Abstraktion eines Schaltplanes. Daten, die fließen, sind nicht unbedingt identisch mit den Verknüpfungen, über die sie übermittelt werden. Allerdings sind Verbindungen letztlich auch elektrisch und damit in einem Schaltplan ausdrückbar. Dort, wo im Schaltplan mit den Verbindungen der Elektronenfluss dargestellt wurde, fließen Daten. Allerdings gibt dieses Diagramm keine klare Auskunft darüber, was unter Daten verstanden wird. Es kann sich um elektrische Verbindungen handeln oder um komplexere Verbindungen wie Zahlen.

Datenflussdiagramme haben die in Abbildung 2.2 dargestellte Form. Unterschiedliche Blöcke sind miteinander verbunden. Die Bauteile können unterschiedlich viele Ein- oder Ausgabepins besitzen. Außerhalb des Diagramms finden sich in Abbildung 2.2 links die Eingabepins und rechts die Ausgabepins. Das ist unabhängig von den Typen der Signale, die über die Pins übertragen werden. Dieselben Pins werden bei der Betrachtung von innen als Quelle und von außen als Senke bezeichnet.

Hier gibt es, wie im vorhergehenden Abschnitt, beschriftete Elemente. Sie werden über Rechtecke symbolisiert und stehen stellvertretend für die Funktion, welche die Daten verändert. Wichtig ist hier ebenfalls die Anzahl der Anknüpfungstellen. Komponenten können nicht beliebig viele Daten bearbeiten. Die Komponenten sind miteinander verknüpft, was durch einfache Linien symbolisiert wird. Diese haben, im Gegensatz zur elektronischen Schaltung, eine Richtung. Die Richtung des Datenflusses wird entweder durch Pfeile angegeben oder ergibt sich aus dem Kontext der

Beschriftungen. Datenflussdiagramme werden gerne für die Veranschaulichung von kryptographischen Algorithmen herangezogen. Beispielsweise stellt die Abbildung 2.2 den Fluss der Daten (a, b, c und d) innerhalb der MD5-Runden dar. Die Runden sind in den Boxen $[f, g, h, i]$round angegeben.

Das Diagramm ist aus Linien und Rechtecken aufgebaut, wobei die Rechtecke für Elemente des Algorithmus stehen. Sie sind über Verknüpfungslinien miteinander verbunden. Das Prinzip ähnelt dem der elektronischen Schaltungen.

2.3. Aktivitätsdiagramme

Unter Aktivitätsdiagrammen werden Diagramme verstanden, die innerhalb des UML-Standards als solche beschrieben wurden. Verglichen mit Datenflussdiagrammen, sind Aktivitätsdiagramme noch allgemeiner. Die Betonung liegt auf der Aktivität. Diese wird wie in den beiden vorhergehenden Darstellungsformen mit anderen Aktivitäten verknüpft. Die Verknüpfungen haben ebenfalls Richtungsangaben und transportieren Objekte. Im Unterschied zur Sicht des Datenflusses ist in dieser Diagrammform die Zahl der tatsächlichen Verknüpfungen nicht festgeschrieben. Werden Daten zwischen zwei Aktivitäten ausgetauscht, sind diese über einen Pfeil verbunden. Der Pfeil kann für ein Datum stehen oder für eine Vielzahl von Daten. In Aktivitätsdiagrammen gibt es besondere Aktivitäten, wie zum Beispiel Entscheidungsknoten, die als auf einer Spitze stehende Raute symbolisiert werden.

Aktivitätsdiagramme kennen neben Objektflüssen auch Kontrollflüsse. Es handelt sich um Verbindungen von Aktivitäten, die nicht auf Datenaustausch basieren, sondern die Reihenfolge der Auswertung festlegen. In Aktivitätsdiagrammen lassen sich die Elemente und deren Verknüpfungen identifizieren. Das war bereits in den beiden vorhergehenden Diagrammen der Fall.

2.4. Wiederkehrende Muster

Die bisher betrachteten Ansichten sind sich ähnlich. Sie unterscheiden sich nur marginal voneinander. Die angegebenen Darstellungen lassen sich jeweils zu einem Graphen umformen. Sei G ein Graph, bestehend aus dem Tupel (V, E) mit der Menge

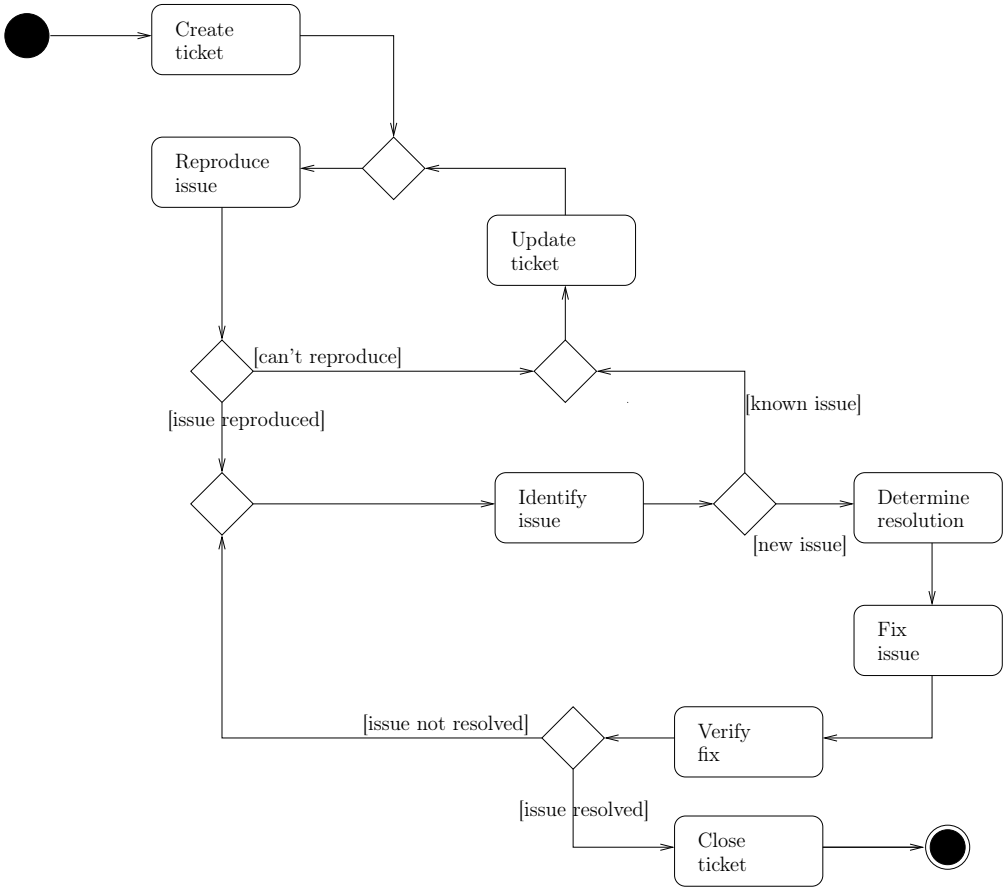

Abbildung 2.3.: Symbolbild Aktivitäten-Diagramm

der Knoten [1] V und mit der Menge der Kanten [2] E, lassen sich Morphismen von den einzelnen Diagrammen zu einem Graphen angeben.

Der Morphismus f_e formt einen Schaltplan zu einem Graphen um. Die Symbole der Bauteile werden zu den Knoten umgeformt, die Verbindungen zu Kanten des Graphen. Für ein Datenflussdiagramm lässt sich ein Morphismus f_d angeben. Für die Aktivitätsdiagramme existiert ebenfalls ein solcher Morphismus f_a. Die Aktionen, aber auch die Verzweigungen (Rauten), werden zu Knoten.

Sei $\mathcal{C}$ die Kategorie der Diagramme, wäre der Graph $G \in \mathcal{C}$, so wäre dieser damit das kategorientheoretische Limit der Kategorie $\mathcal{C}$.

$$\text{action chart} : a \in \mathcal{C}$$

[1] Vertices
[2] Edges

,

$$\text{electronic network} : e \in \mathcal{C}$$

,

$$\text{dataflow diagram} : d \in \mathcal{C}$$

,

$$\text{graph} : g \in \mathcal{C}$$

.

Also existiert zu jedem dieser Diagramme der oben erwähnte Morphismus, der das Diagramm in einen Graphen überführt.

$$f_a : a \in \mathcal{C} \to g \in \mathcal{C}, f_e : e \in \mathcal{C} \to g \in \mathcal{C}, f_d : d \in \mathcal{C} \to g \in \mathcal{C}$$

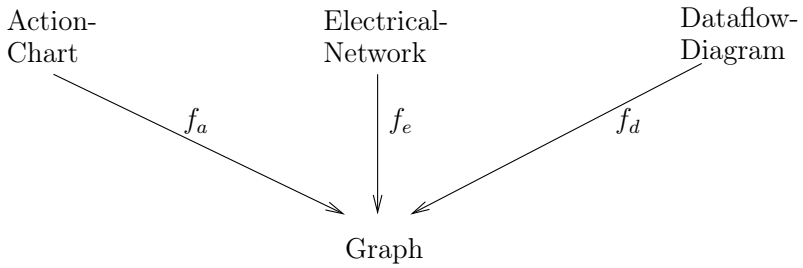

Abbildung 2.4.: Graph ist Limit

Da es diese Morphismen f_a, f_e, f_d geben muss, können die Diagramme als Graphen betrachtet werden. Eine funktionale Datenstruktur muss somit diese Eigenschaft in der Software widerspiegeln.

Aus der Kategorientheorie ist das Konzept der erweiterten Freyd-Kategorien [1] bekannt, eben jenen Graphen, die Beschreibungen zuordnen. Dieses Konstrukt wird in Haskell Arrows genannt. Ein Arrow stellt eine Berechnung abstrakt dar. Die Berechnung wird in einer Art „Black-Box“ dargestellt. Durch Haskells Typsystem wird abgesichert, dass diese „Black-Boxen“ nur typkorrekt mit anderen „Black-Boxen“ kombiniert werden können.

Mit Arrows lässt sich die Kombination von Hardwarekomponenten softwaretechnisch abbilden. Eine Darstellung mithilfe von Arrows ist so abstrakt, dass sie eine algebraische Interpretation eines Schaltkreises zuläßt. Deshalb erlaubt eine solche

Interpretation das „Rechnen“ mit den Hardwarekomponenten. Beispielsweise ist das Ausfiltern eines Bauteils oder das Ersetzen eines weiteren Bauteils möglich.

Bei den aktuell verwendeten Hardwarebeschreibungssprachen existieren Schaltungen nur als Text, der einer gewissen Semantik folgt. Suchen und Ersetzen können innerhalb dieser textuellen Beschreibung geschehen. Darüber hinaus ist kein maschinelles Arbeiten mit der Beschreibung möglich.

3. Funktionale Entwicklung

Die Voraussetzungen zur funktionalen Beschreibung von Netzlisten sind ebenfalls funktionale Mittel. Im folgenden Kapitel werden eben diese funktionalen Methoden und Herangehensweisen beschrieben. Damit wird das Vokabular der restlichen Arbeit angelegt. Darüber hinaus lädt dieses Kapitel zu einem Exkurs in die Methoden der funktionalen Entwicklung ein.

Viele Programmiersprachen beanspruchen das Label „funktionale Programmiersprache“. Haskell ist im Gegensatz zu anderen Sprachen ausschließlich funktional, also eine rein funktionale Sprache. Kein Konzept in Haskell widerspricht dem funktionalen Gedanken. Dies ist ein Alleinstellungsmerkmal. In anderen Sprachen steht das Label „funktionale Sprache“ vor allem für das Vorhandensein funktionaler Features.

Eine weite Verbreitung einer Sprache ist vorteilhaft. Um dies zu erreichen, darf die Hürde, sich mit der Sprache auseinanderzusetzen, nicht zu hoch sein.

Die beste Sprache für einen Programmierer wäre eine Sprache, die von jedem verstanden wird. Zusätzlich sollte ein Compiler existieren, der den Programmierer maximal unterstützt. Eine Sprache mit einem solchen Umfeld gibt es nicht.

Sprachen mit offenem Standard haben hier einen Vorteil. Eine hohe Verbreitung garantiert, dass sich viele Programmierer mit dieser Sprache auskennen. Das führt potentiell zu besseren Entwicklungswerkzeugen. Sind diese quelloffen, wird der Effekt noch verstärkt. Eine hohe Verbreitung wirkt sich damit potentiell auch auf Qualität und Quantität der Dokumentation der Sprache aus.

Als Programmiersprache und Repräsentant des funktionalen Paradigmas wird Haskell gewählt. Die Wahl erfolgt aufgrund der folgenden Punkte: 1: Haskell ist die funktionale Sprache mit der größten „Community“; 2: Entwicklern steht das Hindley-Milner-Typsystem, eines der besten existierenden Typsysteme, zur Verfügung; 3: Haskell ist keine Multiparadigmen-Sprache, sondern eine rein funktionale Sprache

mit den geschilderten Vorteilen.

Ein gutes Typsystem kann Programmierern in vielen Situationen mit fundierten Fehlermeldungen zur Verfügung stehen. Dabei entfaltet ein streng und striktes Typsystem seinen Nutzen schon während der Kompilierungszeit. Haskell besitzt ein solches Typsystem und kennt daher keine Laufzeitfehler.

Der dritte Punkt hilft, eine funktionale Lösung zu finden. Mit jeder Multiparadigmensprache besteht die Gefahr, den Algorithmus unbemerkt imperativ zu entwickeln. Mit Haskell ist das unmöglich.

David Welton hat auf der Seite langpop [1] versucht, die Popularität vieler verschiedener Programmiersprachen zu messen. Es ist nicht klar, wie Welton zu seinen Ergebnissen kommt und inwieweit die Ergebnisse belastbar sind. Nach Welton zählt Haskell zu den Sprachen, die häufig dort erwähnt werden, wo über Programmiersprachen diskutiert wird. In dem Chart zu del.icio.us [2] rangiert Haskell auf Platz 13. In der Auflistung zu programming.reddit.com [3] findet sich Haskell auf Platz 8 wieder und im Chart zur Erwähnung der Sprachen im IRC wird Haskell noch vor Java und C++ auf Platz 2 der meist diskutierten Sprachen genannt. Die Daten wurden im August 2013 abgerufen.

3.1. Funktionseigenschaften

Funktionen haben innerhalb des funktionalen Paradigmas eine besondere Stellung. Im Folgenden werden die Eigenschaften, die beim Erstellen einer Bibliothek verwendet werden, genauer beleuchtet. Diese Eigenschaften sind nicht zwingend nur in der funktionalen Programmierung zu finden. Es spricht nichts dagegen, ähnliche Funktionalitäten in imperative Sprachen zu integrieren.

First Class Functions bedeutet, dass Funktionen in Haskell weder positiv noch negativ diskriminiert werden. Funktionen können, wie andere „Elemente“ der Sprache, spontan einem Bezeichner zugewiesen werden. Sie können Rückgabewerte einer anderen Funktion sein oder als Parameter einer Funktion übergeben werden.

[1] http://www.langpop.com
[2] eine Plattform zum sozialen Sammeln von Links
[3] eine soziale Diskussionsplattform

Funktionskomposition ist ein Akt, der durch Kombination aus bestehenden Funktionen neue Funktionen erschafft. Es wird auf bestehende Funktionalitäten zurückgegriffen, was die modulare Gestaltung des Quellcodes fördert.

Operatoren werden in Haskell jene Funktionen genannt, die infix geschrieben werden. Das sind zum einen ausgewählte Funktionen wie `+`, `-` oder `*`. Mittels „backticks“ (‘) lässt sich jede Funktion in einen Operator umwandeln. `div 3 4` lässt sich als `3 ‘div‘ 4` schreiben. Der gegenteilige Fall ist, einen Operator in Präfix-Schreibweise anzugeben. Das geschieht durch Schreiben des Operators in runden Klammern: `(+) 3 4` entspricht `3 + 4`.

Sections erweitern die eben erwähnte Klammerung. Einem Operator kann ein Argument schon in der Klammer mitgegeben werden. Die Operatoren (Increment- und Decrement) aus C [4] sind in Haskell durch die Sections `(+1)` bzw. `(-1)` definiert, aber nicht auf den Wert 1 beschränkt.

3.2. Datentypen

Stellen Funktionen aufgrund des funktionalen Paradigmas die Einlage der „Suppe“ dar, so sind die Datentypen das sprichwörtliche Salz. Haskell ist eine streng und strikt getypte Programmiersprache und damit gut gewürzt. Dieses Prinzip steht im Kontrast zu „ducktyped“ [5] Sprachen wie Python, Javascript oder Ruby. Streng getypt heißt: In Haskell besitzt jeder Ausdruck einen festen Typ. Strikt getypt heißt: Der Typ eines Ausdrucks kann nicht verändert werden. Der Typchecker kann also durch “casts” nicht ausgehebelt werden. Haskells Typsystem ist statisch, was bedeutet, dass alle Typen zur Kompilierungszeit bekannt sind. Diese Eigenschaften erzwingen die Typkorrektheit jedes Programms.

Obwohl jeder Ausdruck in Haskell einen Typ besitzt, ist es nicht notwendig, jeden Term mit einer Typdefinition zu versehen. Haskells Typsystem bedient sich des Mechanismus der „Type Inference“. Das ist eine Methode, die den Typ zu einem Ausdruck ableitet, sofern dies möglich ist.

Obwohl die Typen in Haskell nicht immer angegeben werden, hilft das Typsystem dem Programmierer enorm. Definiert man zunächst den Typ einer Funktion und

[4] ++ und −−

[5] dynamisch typisierten

implementiert die Funktion daraufhin, entspricht dieses Vorgehen dem Entwickeln mit „Unit Tests“. Die Verbindung von Typ und Programm hat auch mathematisch eine Bedeutung. Wie Haskell Curry [19, 20] und William Howard [31] gezeigt haben, gibt es eine direkte Beziehung zwischen Programmen und Beweisen. Der Curry-Howard-Isomorphismus [60, 52] besagt, dass ein Programm einen Beweis für seinen Typ darstellt. Häufig sind jedoch Vorüberlegungen zum Typ einer Funktion einfach. Kann man eine Funktion definieren, die dem Typ entspricht, bleibt wenig Raum für Fehler. Je spezieller man den Typ angibt, umso bessere Ergebnisse werden erreicht.

Um spezielle Typen für Funktionen angeben zu können, ist ein aussagekräftiges Typsystem notwendig. Es ermöglicht dem Programmierer, die Typen nach eigenen Vorstellungen zu definieren. Im Folgenden werden einzelne Methoden, Typen zu definieren, vorgestellt. Die Auswahl beschränkt sich auf jene, die bei der Erstellung der Bibliothek angewandt wurden:

Aufzählungstypen stellen die einfachste Form einer Typdefinition dar. Hierzu zählen alle Typen, die sich durch reines Aufzählen der möglichen Werte beschreiben lassen. Beispiel für einen Aufzählungstyp ist `Bool`. Dieser Typ besteht aus `True` und `False`.

Zusammengesetzte Datentypen sind jene Typen, die aus existierenden Typen erstellt werden.

Parametrisierte Typen besitzen einen oder mehrere Parameter, die den exakten Typ festlegen. Der Listentyp ist beispielsweise ein parametrisierter Datentyp. Es gibt Ganzzahl-Listen `[Int]` und Chracter-Listen `[Char]`.

3.3. Typklassen

Polymorphie wird in Haskell über Typklassen realisiert. Datentypen können einer Typklasse zugeordnet werden, wenn sie den definierten Anforderungen genügen. Darauf aufbauend lassen sich Funktionen erstellen, die sich auf die Anforderung stützen. Die Typklasse Eq gibt beispielsweise an, ob die Gleichheit zweier Werte gleichen Typs ermittelt werden kann. Erfüllen beide Werte diese Eigenschaft (sind beide in der Typklasse), so können sie auf Gleichheit überprüft werden.

Die Funktion `lookup` ist eine Funktion, die in einer Liste nach einem Element

sucht und dieses nur zurückliefert, wenn es in der Liste vorkommt. Der Typ dieser Funktion sieht so aus:

```
1 lookup :: (Eq a) => a -> [a] -> Maybe a
```

Der Teil vor dem `=>` legt fest, welche Bedingungen `a` erfüllen muss. Die Schreibweise ist angelehnt an die mathematische „forall“-Darstellung:

$$\text{lookup} : \forall \alpha \in \text{Eq},\ \alpha \times P(\alpha) \to \alpha \,\dot{\vee}\, \text{Nothing}$$

Der Datentyp in Haskell kann auch so notiert werden:

```
1 lookup :: forall (a) . (Eq a) => a -> [a] -> Maybe a
```

Eine Typklasse besteht aus den Teilen Klassendefinition und Klasseninstanz. Die Klassendefinition legt fest, welche Funktionen definiert sein müssen, damit ein Typ Teil jener Typklasse wird. Bei `Eq` sieht die Klassendefinition so aus:

```
1 class Eq (a) where
2   (==) :: a -> a -> Bool
```

Für einen Datentyp, der Instanz der Typklasse `Eq` ist, muss definiert sein, wie die Funktion `(==)` für diesen Typ arbeitet. Die Instanzdefinition von Eq für Wahrheitswerte würde in etwa so aussehen:

```
1 instance Eq (Bool) where
2   (==) True  True  = True
3   (==) False False = True
4   (==) _     _     = False
```

Hier wird der Typ `Bool` in die Typklasse `Eq` aufgenommen. Sind beide Parameter gleich, ist das Ergebnis „wahr“. Die Unterstrich-Schreibweise `_` ist syntaktischer Zucker und definiert, dass in jedem anderen Fall das Ergebnis „falsch“ ist. Da die Anweisungen in der Reihenfolge ausgewertet werden, in der sie aufgelistet sind, kann die letzte Zeile als „catch-all“ verstanden werden.

3.4. Listen

Listen sind in funktionalen Programmiersprachen die zentrale Datenstruktur. Die Gründe dafür sind vielschichtig. Zunächst basiert die erste funktionale Sprache, Lisp, auf Listen. Dies spiegelt sich schon im Namen wider, Lisp steht für List Processor. Auch die Einfachheit einer Liste erklärt den Ursprung der zentralen Bedeutung in der funktionalen Welt. Eine Liste wird entweder als die leere Liste (`[]`) definiert oder als Zusammenschluss eines Elementes mit einer Restliste (`x:[xs]`). In Haskell wird dies so dargestellt:

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

Diese Datenstruktur spiegelt den rekursiven Ansatz vieler funktionaler Sprachen wider. Eine Funktion, die auf der gesamten Liste arbeitet, wird definiert, indem zwei Fälle berücksichtigt werden: Die Eingabe ist die leere Liste oder die Eingabe ist ein Element in Verbindung mit einer Restliste.

Imperative Sprachen verwenden häufig Arrays, funktionale Sprachen dagegen selten. In der Regel versteht man unter Arrays eine veränderbare Datenstruktur mit konstanter Zugriffszeit. Dies ist in funktionalen Sprachen nicht ohne Hilfskonstruktionen wie Monaden zu lösen. Nicht veränderbare Arrays lassen sich schwieriger erzeugen als Listen. Beim Einfügen eines Elementes in ein Array muss das gesamte Array kopiert werden, was zu einer quadratischen Laufzeit führt.

Da Listen in den meisten funktionalen Sprachen eine besondere Bedeutung zukommt, gibt es in Haskell viele syntaktische Vereinfachungen, um mit Listen zu arbeiten. Im Folgenden werden einige dieser Vereinfachungen vorgestellt.

Schreibweise : In Haskell wird für Listen eine einfachere Schreibweise angewandt als die oben erwähnte. Das `Nil` wird als `[]` geschrieben und `Cons` schreibt sich in Haskell als `:`. Vorteil ist, dass sich eine Liste kompakt schreiben lässt: `1:(2:(3:[]))`, im Gegensatz zu `(Cons 1 (Cons 2 (Cons 3 Nil)))`.

Konkatenation : Listen lassen sich mittels des Operators `(++)` zusammenfügen. `[1,2] ++ [3,4,5]` wird zu `[1,2,3,4,5]`.

Listenkomprehension : Haskell bietet die Möglichkeit, Listen über eine „besondere“ Schreibweise zu definieren. So lässt sich beispielsweise die Liste der Quadrat-

zahlen durch folgende Listenkomprehension schreiben: `[x*x | x <- [1..]]`. Hierbei handelt es sich um eine unendliche Liste.

3.5. Morphismen

Beim Programmieren einfacher Aufgaben stößt man immer wieder auf die Anforderung, wiederkehrende Probleme zu lösen. Eine Aufgabe, die gern nach einem gelösten „Hello World"-Problem gestellt wird, ist: „99 bottles of beer". Dabei soll ein Text 99 Mal ausgegeben werden. In jeder Iteration wird die Anzahl der „bottles of beer" um 1 verringert. An dieser Stelle werden in den meisten Programmiersprachen Schleifen vorgestellt. Schleifen, im Sinne von prozeduralen/imperativen Sprachen, benötigen Variablen, die bis zu einem bestimmten Wert verändert werden. Bis dieser Wert erreicht ist, wird der Inhalt der Schleife wiederholt ausgeführt.

Eine Folge der strikten Umsetzung des funktionalen Paradigmas ist die Abwesenheit von Variablen. Im λ-Kalkül existieren nur Funktionen. Variablen benötigen veränderbaren Speicher. Dieser Speicher besteht unabhängig von der Parameterliste einer Funktion (Heap). Dieser Seitenkanal existiert in rein funktionalen Sprachen nicht. Werte können nur über die Parameterliste in eine Funktion gelangen. Haskell kennt folglich keine Variablen im engeren Sinne. Die Funktionalität einer Schleife wird in Haskell über Rekursion, also sich selbst aufrufende Funktionen, erreicht. Auch hier ist die Orientierung nahe am mathematischen Vorbild, denn auch in mathematischen Beschreibungen von Strukturen wird man `for`-Schleifen vergebens suchen.

Haskell geht einen Schritt über einfache Rekursion hinaus. Rekursive Aufrufe von Funktionen werden in sogenannten „Rekursionsschemata" zusammengefasst. Das sind Lösungsmuster, die den „Design Pattern" der objektorientierten Programmierung ähneln.

Es existieren unterschiedliche Arten der Rekursion [6]. Die Unterscheidung wird aufgrund der Veränderung der Strukturen vorgenommen, die mit der Rekursion bearbeitet werden. Ein solches Rekursionsschema fasst die Art der Rekursion in einer Funktion zusammen. Es gibt strukturerhaltende Rekursionen, strukturerzeugende Rekursionen und strukturabbauende Rekursionen. Zur Veranschaulichung werden diese drei Morphismen vorgestellt:

[6] Hier ist nicht die Endrekursion gemeint.

3.5.1. Homomorphismen

Ein Homomorphismus ist in Haskell als `map` bekannt.

```
1 map :: (a -> b) -> [a] -> [b]
```

Der Typ zeigt an, dass `map` ein strukturerhaltender Morphismus ist. Die übergebene Struktur von Daten, auf denen der Morphismus arbeitet, bleibt erhalten. Lediglich innerhalb der Struktur wird geändert: aus Typ a wird Typ b.

Die Ausführung von `map` hängt nur von der aktuellen Zelle der Liste ab, nicht von der Gesamtliste. Daher ist es möglich, Aufrufe von `map` zu parallelisieren. Der Compiler kann selber entscheiden, ob die Ausführung auf mehreren Prozessoren geschehen soll.

3.5.2. Katamorphismen

Unter Katamorphismen wird in Haskell die Familie der folds verstanden. Es gibt eine ganze Reihe folds in Haskell, da es unterschiedliche Möglichkeiten gibt, zum Ergebnis zu gelangen. Als Beispiel wird die Typsignatur von `foldl` gezeigt:

```
1 foldl :: (b -> a -> b) -> b -> [a] -> b
```

Die letzten beiden Elemente dieser Typsignatur zeigen auf, dass ein fold ein strukturabbauender Morphismus ist. Strukturabbauend bedeutet, dass aus der Liste `[a]` ein einzelner Wert `b` berechnet wird. Es kann sich um einfache Berechnungen wie die Summe oder die Länge einer Liste handeln. Ein fold hat inhärent die Eigenschaft einer Reihenfolge. Das bedeutet, dass das Ergebnis der vorhergehenden Berechnung mit dem nächsten Element der Liste „verrechnet“ wird. Einem fold wird zusätzlich zur Funktion und zur Liste ein „Startwert“ z übergeben, der die initiale Eingabe der Berechnung ist. z verhält sich zur Funktion f als neutrales Element.

fold lässt sich im Gegensatz zu `map` nicht parallelisieren, da für einen Schritt das Ergebnis des vorhergehenden Schrittes benötigt wird. Um mit einem fold die Summe einer Liste zu berechnen, verwendet man die Addition als Funktion, mit dem neutralen Element 0 als „Start-Wert“: `foldl (+) 0`

3.5.3. Anamorphismen

Neben Katamorphismen und Homomorphismen gibt es Anamorphismen. Ein Anamorphismus, auch „unfold“ genannt, ist ein strukturerzeugender Morphismus. Darunter wird ein Rekursionsschema verstanden, das eine Datenstruktur, z. B. eine Liste, aus einem Startwert [7] erzeugt.

```
1 unfoldr :: (a -> Maybe (b, a)) -> a -> [b]
```

Die Typsignatur verdeutlicht, dass die Funktion zusammen mit dem Startwert eine Liste erzeugt. Der Rückgabetyp der Funktion muss so beschaffen sein, dass er ein Ergebnis liefern kann und das Ende der Rekursion anzeigt. Im Ergebnisfall muss dieses zweiteilig sein: Der erste Teil fließt in die Ergebnisliste ein, der zweite Teil ist die neue Eingabe für die Rekursion.

[7] auch „Seed“-Value genannt

4. Hardware Design Algebren

4.1. Lava Hardware Design Algebra

Lava ist ein Werkzeug für Hardware-Entwickler [6] und besteht aus einer Ansammlung von Funktionen und Datentypen. Ein Schaltkreis in Lava wird von einer Funktion in Haskell erzeugt, die auf `Signal` definiert ist.

Ein Halbaddierer besteht aus den Gattern „and“ und „xor“. Wird an beide Gatter dasselbe Signal angelegt, liegt am Ausgang des „xor“-Gatters die Summe `sum` und am Ausgang des „and“-Gatters der Übertrag `carry` an. Die Implementierung des Halbaddierers in Lava ist im folgenden Listing gezeigt [18, page 6].

```
1 halfAdd :: (Signal Bool, Signal Bool) -> (Signal Bool, Signal Bool)
2 halfAdd (a, b)
3   = (sum, carry)
4   where
5     sum   = xor2 (a, b)
6     carry = and2 (a, b)
```

Die Funktionen `xor2` und `and2` werden von Lava bereitgestellt. Alle weiteren Bausteine Lavas lassen sich dem Tutorial [18, page 73 ff.] entnehmen. Das Suffix „2“ an den Funktionsnamen zeigt eine Schwierigkeit Lavas auf: Sämtliche Funktionen lassen sich nicht allgemein definieren, sondern beziehen sich immer auf einen Datentyp, hier `(Signal Bool, Signal Bool)`. Der Signal-Typ ist der zentrale Datentyp Lavas. Er kommt in zwei Ausprägungen vor, nämlich `Signal Bool` und `Signal Int`.

Sämtliche Funktionen in Lava sind auf diesen Typ ausgelegt. Ein `Signal Bool` ist ein einzelner Draht, ein `Signal Int` ist eine vereinfachte Schreibweise mehrerer boolscher Signale.

In Lava definierte Schaltkreise können auf unterschiedliche Weisen interpretiert wer-

den. Durch die Interpretation des Modells lässt sich Hardware-Code erzeugen; das Modell kann simuliert oder verifiziert werden.

4.1.1. Simulation

Mit der Simulation der modellierten Schaltkreise kann das Modell gegen die Vorstellungen geprüft werden. Hierfür liefert Lava passende Werkzeuge. Simulation bedeutet in Lava, einen Schaltkreis mit definierten Eingabewerten zu überprüfen. Da Lava seicht eingebettet ist, kann ein Schaltkreis in Lava nicht direkt ausgeführt werden, sondern muss von der Simulationsfunktion `simulateSeq` [18] aufgerufen werden. Erst dadurch kann das funktionale Modell beobachtet werden.

Neben der händischen Definition der Eingabewerte bietet Lava eine Möglichkeit, den Schaltkreis indirekt mit allen möglichen Eingabewerten zu bestücken. Der dazu benötigte Eingabewert ist die Liste `domain`. Derselbe Mechanismus ist mittlerweile als Quick-Check [16] bekannt.

4.1.2. Verifikation

In Lava definierte Schaltkreise lassen sich verifizieren. Verifikation setzt voraus, dass der Schaltung bestimmte Eigenschaften zugeordnet werden können. Diese Eigenschaften müssen in eine Funktion eingepackt werden. Die Schaltung erfüllt diese Eigenschaften dann, wenn diese Funktion alle Parameter `true` zurückliefert. Lava bietet die Möglichkeit, dies zu verifizieren. Dazu wird die Funktion formalisiert, sodass sie mit einem „Theorem Prover“ verifiziert werden kann. Die Verifikation geschieht hier nur indirekt durch Lava, da ein externer „Theorem Prover“ eingesetzt wird.

Je komplexer eine Schaltung, desto komplexer wird auch die mathematische Beschreibung derselben. Es können daher mit dieser Methode sinnvoll nur einfache Beweisfunktionen und somit nur einfache Schaltungen verifiziert werden.

Ein einfaches Beispiel einer Verifikation wurde dem „Lava Tutorial“ [18] entnommen:

```
1 prop_HalfAddOutNeverBothTrue :: (Signal Bool, Signal Bool) -> Bool
2 prop_HalfAddOutNeverBothTrue (a, b)
3   = ok
```

```
4   where
5     (sum, carry) = halfAdd (a, b)
6     ok           = inv (and2 (sum, carry))
```

`and2` kann nur wahr werden, wenn der Halbaddierer defekt ist. Anders ausgedrückt, der Halbaddierer muss an beiden Ausgängen `true` ausgeben, damit die Beweisfunktion `false` ausgibt. Der Beweis wird von einem „Theorem Prover“ erbracht, der die Korrektheit der Funktion `verify prop_HalfAddOutNeverBothTrue` zeigt. Das Ergebnis der Prüfung der formalen Darstellung Lavas wird an Lava zurückgemeldet und der Schaltkreis als „valid“ gekennzeichnet.

4.1.3. Funktionaler Filter mit endlicher Impulsantwort

Ein FIR-Filter (Filter mit endlicher Impulsantwort) ist ein nicht rekursiver Filter. Mathematisch wird ein FIR-Filter mit einer Gleichung beschrieben, in der die Eingaben $x \in X$ mit b_q gewichtet werden:

$$y(n) = \sum_{q=0}^{Q} b_q * x(n-q) \tag{4.1}$$

Die Definition eines FIR-Filters geschieht in Lava über die Definition einer Funktion. Diese wird über ihre Typsignatur auf den Lava-Datentyp `Signal ..` beschränkt.

Ein FIR-Filter ist im funktionalen Modell Lavas eine Abbildung von Filterkoeffizienten und ein Eingangssignal auf ein Ausgangssignal.

```
1  conv :: Coefficients -> Values -> Values
2  conv bs xs = conv' n bs' xs'
3    where bs' = (replicate (length xs - 1) 0) ++ bs
4          xs' = (reverse xs) ++ (replicate (length bs - 1) 0)
5          n   = length xs'
6
7  conv' ::Int -> Coefficients -> Values -> Values
8  conv' 0 _ _ = []
9  conv' n bs xs = y : (conv' (n-1) bs xs')
10   where y   = lsum (ltimes bs xs)
11         xs' = (delayInt (int 0) (head xs)):xs_rst
12         xs_rst = (reverse.tail.reverse) xs
```

`Coefficients` und `Values` sind jeweils Typaliase für Listen von `Signal Int`-Werten.

```
type Values       = [Signal Int]
type Coefficients = Values
```

Der `Signal a`-Datentyp stellt eine eingehende, eine ausgehende oder eine interne Verbindung dar [15, S. 56]. Zusätzlich wird dieser Datentyp zur symbolischen Auswertung von Schaltungen verwendet. Auch die seichte Einbettung von Lava zeigt sich an diesem Datentyp, da bei der Auswertung der Schaltung diese in Form von Strings repräsentiert wird. Eine Folge dieser Auswertung ist das sogenannte Sharing-Problem [17], bei dem eine Schleife in eine unendliche Folge immer gleicher Bauteile übersetzt wird. Bei der seichten Einbettung besitzt die Gastsprache nicht die Komplexität der Wirtssprache. Eben diese Darstellung (hier in Form eines Strings) führt dazu, dass das Typsystem nicht erkennen kann, dass beispielsweise die folgenden Schaltungen (`circ1` und `circ2`) gleich sind.

```
circ1 = let output = latch output in output
circ2 = let output = latch (latch output) in output
```

[17, Section 3]

Das Entwicklerteam um Lava begegnet diesem Problem mit der Einführung einer nicht konservativen Spracherweiterung [17]. Diese Spracherweiterung - Observable Sharing - ermöglicht eine Variante des Signal-Typs, bei der das Sharing direkt im Datentyp sichtbar (observable) gemacht wird.

Der Filter `conv` ist durch `conv'` rekursiv definiert. Die Rekursionsbremse wird durch den zusätzlichen ganzzahligen Parameter `n` von `conv'` gegeben. Bei jeder Iteration wird von `conv'` ein `y` als die Summe (`lsum`) aller paarweise `ltimes`-verknüpften Werte aus `bs` und `xs'` berechnet. `xs'` stellt das um eine Einheit verzögerte (`delayInt (int 0) (head xs)`) Eingangssignal dar.

Mit `ltimes` wird die strukturerhaltende Multiplikation der Filtergleichung 4.1 abgebildet, `lsum` stellt das Äquivalent zum katamorphen Summenoperator dar. Beide Funktionen klammern links. Die eigentliche Multiplikation ist ein Operator Lavas und bildet zwei Eingangssignale auf ein Ausgangssignal ab.

Die Typsignatur `conv :: Coefficients -> Values -> Values` des FIR-Filters zeigt an, dass der Filter von beliebiger Ordnung ist. Das liegt daran, das `conv` auch von den `Coefficients` abhängt. Die Definition der Koeffizienten legt die Ordnung des Filters fest.

Die Korrektheit des Filters lässt sich über einen externen „Theorem Prover“ zeigen. Hierzu wird eine Eigenschaft des Filters - nämlich dass ein FIR-Filter seine Koeffizienten ausgibt, wenn er auf einen Dirac-Impuls ($\delta(n)$) angewandt wird - als Funktion definiert.

```
1 prop_ConvByOneBringsCoefs
2   :: (Coefficients -> Values -> Values)
3   -> Coefficients
4   -> Bool
5 prop_ConvByOneBringsCoefs f bs
6   =  bs == (f bs [1])
```

Diese Funktion ist nur dann wahr, wenn der Filter und seine Koeffizienten als Parameter der Funktion übergeben werden. `verify prop_ConvByOneBringsCoefs` veranlasst den „Theorem Prover“ die definierte Eigenschaft des Filters zu beweisen. Die Ausgabe `valid` besagt, dass sich der Filter wie erwartet verhält.

4.2. ForSyDe Hardware Design Algebra

ForSyDe ist eine Haskell-Bibliothek, mit der die Komponenten durch sogenannte Prozesse simuliert und synthetisiert werden. Prozesse werden mit Prozesskonstruktoren erzeugt. Dabei zeigt das „SY“-Suffix an, dass dieser Konstruktor einen synchronen Prozess erzeugt.

$$
\begin{aligned}
process \rightarrow\ & zipSY \\
\mid\ & unzipSY \\
\mid\ & mapSY\ procFun^1 \\
\mid\ & delaySY\ procFun^1 \\
\mid\ & zipWithSY\ procFun^2 \\
\mid\ & zipWith3SY\ procFun^3 \\
\mid\ & \ldots
\end{aligned}
$$

Prozesse bilden Signale auf weitere Signale ab. Ein Signal enthält eine Liste von Events, wobei jedes Event aus einem Wert und einem Zeitstempel besteht. Zur Vereinfachung können Signale als (auch unendliche) Werteliste betrachtet werden, bei der jeder Wert einem definierten Zeitpunkt entspricht.

Am Prozesskonstruktor `mapSY` lässt sich das verdeutlichen. Der Konstruktor überführt eine Funktion zusammen mit einem Identifikator in einen Prozess, der dann von `Signal a` nach `Signal b` abbildet. Der Typ des `mapSY`-Konstruktors wird so angegeben:

```
1 mapSY :: ProcId
2       -> ProcFun (a -> b)
3       -> Signal a
4       -> Signal b
```

Ein vielgenutzter Konstrukter ist `delaySY`. Dieser ermöglicht die Erstellung von sequentiellen Systemen. Er zeigt das Verhalten von Speicher-Flip-Flops, die ein Signal ebenfalls um eine Zeiteinheit verzögern. Der Konstruktor wird mit einem initialen Wert sowie einem Signal beschickt und erzeugt daraus das verzögerte Signal.

```
1 delaySY :: ProcId
2         -> a
3         -> Signal a
4         -> Signal a
```

Für die Implementation im nächsten Abschnitt sind der Prozesskonstruktor `zipSY` und dessen Umkehrfunktion `unzipSY` von Bedeutung. Der `zipSY`-Konstruktor bündelt Signale und `unzipSY` zerlegt ein gebündeltes Signal zurück in die Ausgangsstränge.

```
1 zipSY   :: ProcId
2         -> Signal a
3         -> Signal b
4         -> Signal (a, b)
5
6 unZipSY :: ProcId
7         -> Signal (a, b)
8         -> (Signal a, Signal b)
```

Ein System wird in ForSyDe als SysDef bezeichnet. Ein solches System kann ebenfalls mit einem Prozess spezifiziert werden. Es sind auch SysDef-Systeme, welche in VHDL überführt werden oder aber simuliert werden. Eine Systemdefinition ist in `newSysDef` dargestellt.

```
newSysDef :: (SysFun f)
          => f
          -> SysId
          -> [ProcId]
          -> [ProcId]
          -> SysDef f
```

Das Konstrukt `SysFun f` ist ein Prädikat und Haskells Weg, Polymorphie zu beschränken. Konkret wird `f` auf Prozesse eingeschränkt.

Mit dem `instantiate`-Konstruktor können Vervielfältigungen oder Instanziierungen der `SysFun` erzeugt werden. Dabei werden Prozesse erzeugt, die dann in eine Silizium-Entsprechung überführt werden können.

```
instantiate :: (SysFun f)
            => ProcId
            -> SysDef f
            -> f
```

Es wird hier als Beispiel gezeigt, wie mit dieser Methode ein kombinatorisches System erstellt werden kann. `funcf` beschreibt den internen Teil eines MD5 f-Blocks:

```
funcf :: ProcFun (Int32 -> Int32 -> Int32 -> Int32)
funcf
  = ... \x y z -> (x .\. y) .|. ((complement x) .\. z) ...
```

(Der Teil der Syntax, der hier zur Erzeugung des abstrakten Syntaxbaumes benötigt wird, ist nicht intuitiv nachvollziehbar und wurde daher hier zunächst weggelassen.) `ProcFun` selbst ist eine gewöhnliche Haskell-Funktion in Verbindung mit ihrem abstrakten Syntax-Baum.

Der Konstruktor `zipWith3SY` wird genutzt, um `funcf` in einen Prozess zu erheben (lift).

```
funcfProc :: Signal Int32
          -> Signal Int32
          -> Signal Int32
          -> Signal Int32
funcfProc
  = zipWith3SY "funcfProc" funcf
```

Zuletzt wird mit dem Ausdruck

```
newSysDef funcfProc "f"
  ["x","y","z"] ["x","y","z"]
```

eine neue Hardwarekomponente der obigen Systemdefinition erstellt. Diese ist in ForSyDe derart verankert, dass sie synthetisiert, simuliert oder instanziiert werden kann.

5. Funktionale Datenstrukturen zur Netzlistenbeschreibung

Dieses Kapitel beschreibt eine funktionale Datenstruktur. Sie ist aus Sicht des Autors ideal zur Netzlistenbeschreibung. Diese Datenstruktur ist deswegen ideal, weil sie einen Kompromiss aus den unterschiedlichen Einbettungstiefen zulässt. Die Struktur kann genutzt werden, um Netzlistenbeschreibungen oberflächlich in die Wirtssprache einzubetten. Sie kann aber auch zur tiefen Einbettung verwendet werden. Mit seichter Einbettung ist hier gemeint, dass Sprachkonstrukte der Wirtssprache genutzt werden, um ein Ergebniss in der Gastsprache zu erzielen. Tiefe Einbettung bedeutet, dass das Typsystem der Wirtssprache angewandt wird, um Elemente der Gastsprache typsicher zu beschreiben.

Die hier verwendete Datenstruktur wird Arrow genannt. Arrows sind als Verallgemeinerung von Monaden entstanden. Daher werden Arrows über den „Umweg“ von Monaden erläutert.

5.1. Von Monaden ...

Um einen Arrow zu beschreiben, wird das Konzept einer Monade und ihrer Anwendung in der funktionalen Sprache vorgestellt.

Im Abschnitt 3.5 wird der Unterschied einer Funktion im funktionalen Paradigma zu einer Funktion im prozeduralen/imperativen Paradigma erläutert. Danach sind Monaden notwendig, um Funktionalitäten wie z. B. Ein- und Ausgabe oder Zufälligkeiten zu implementieren.

Da von Funktionen des funktionalen Paradigmas ausgegangen wird, hängt das Resultat einer Funktion ausschließlich von ihren Parametern ab. Die Funktion ist frei

von Seiteneffekten. Als Seiteneffekt wird jeder Effekt bezeichnet, den eine Funktion neben der Berechnung des Ergebnisses ausübt. Dies ist in prozeduralen/imperativen Programmiersprachen oft der einzige Nutzen einer Funktion. Als Beispiel können hier etwa die `void`-Funktionen aus `C`/`C++` genannt werden.

Um in funktionalen Sprachen typische Seiteneffekte wie Ein- und Ausgabe zu realisieren, muss ein Hilfsmechanismus verwendet werden. Mithilfe eines zusätzlichen Parameters zur Funktion können Seiteneffekte auch in funktionalen Sprachen realisiert werden. Eine Funktion, die eine Ausgabe auf dem Bildschirm produziert, bekommt als zusätzlichen Parameter eine Datenstruktur übergeben, die den Inhalt des Bildschirms vor dem Aufruf enthält. P könnte beispielsweise eine solche Datenstruktur sein. Im Ergebnis liefert die Funktion nun, neben ihrem „eigentlichen“ Resultat, P mit den geänderten Inhalten zurück. Aus $inc : \mathbb{N} \to \mathbb{N}$ wird $incIO : P \times \mathbb{N} \to P \times \mathbb{N}$. Die neue Funktion ist aber noch keine Monade. Das Problem von $incIO$ ist, dass diese Funktion nicht mehr einfach durch Hilfe des $\circ$-Operators mit anderen Funktionen komponiert werden kann. Das Problem verdeutlicht sich, wenn man von einem gecurryten Typ ausgeht: $incIO : P \to \mathbb{N} \to P \times \mathbb{N}$. Benötigt wird an dieser Stelle ein Operator zur Funktionskomposition, der den hinzugefügten Parameter berücksichtigt. Dieser wird *bind* genannt. Um vorhandene Funktionen auch mit den neuen verwenden zu können, wird ein Operator benötigt. Dieser Operator wird unit genannt. Er sorgt dafür, dass ein nicht monadischer Wert monadisch wird. Monaden werden in Haskell über Typklassen realisiert. Diese verlangen die folgenden Definitionen:

```
1 class Monad m where
2   (>>=) :: m a -> (a -> m b) -> m b
3   return :: a -> m a
```

Eine Monade ist ein Tripel aus dem Parameter P, dem Verknüpfungsoperator $bind : P \times \mathbb{N} \to (\mathbb{N} \to P \times \mathbb{N}) \to P \times \mathbb{N}$ sowie dem Operator $unit : \mathbb{N} \to P \times \mathbb{N}$.

Das Tripel muss die Monadengesetze [29] befolgen:

- linke Identität: $bind\,(unit\,x)\,f \equiv f\,x$
- rechte Identität: $bind\,x\,unit \equiv x$
- Assoziativität: $bind\,(bind\,x\,f)\,g \equiv bind\,x\,(bind\,(\lambda y{-}{>}\,f\,x)\,g)$

Dieses einfache Konstrukt ermöglicht das Verknüpfen mehrerer Funktionen. Die Verknüpfung kann in unterschiedlichen Ausprägungen erscheinen. Ein Kontrollfluss kann in ein funktionales Programm eingewoben werden. Auch das Festhalten von Werten auf unterschiedlichsten Ebenen ist mit Monaden modellierbar. Monaden werden häufig für Ein- und Ausgaben eingesetzt, für das Vermitteln von Zuständen oder für das Logging in Dateien.

Dabei werden die sich wiederholenden Teile der Definition hinter syntaktisch cleveren Konstruktionen verborgen. Der Entwickler kann sich auf die Aufgabe konzentrieren. Er wird nicht von der Aufgabe der korrekten Verknüpfung der Parameter abgelenkt.

5.2. ... zu Arrows

Monaden machen es möglich, Werte miteinander zu verweben. Das Ergebnis ist jedoch, wie für Werte üblich, starr. Wenn zwei Funktionen mithilfe einer Monade in eine Reihenfolge gebracht werden, so ist nur diese Reihenfolge gültig. Die Eigenschaft passt nicht auf das Verhalten von Hardware. Es ist nur ein Weg möglich, auf dem ein Signal von einem Baustein zu einem anderen Baustein gelangt.

Hardware besitzt neben einem definierten Kontrolldatenfluss einen modularen Charakter. Ein Signal kann zu jeder Zeit aufgesplittet und an unterschiedliche Bauteile geleitet werden. Dies wird z. B. genutzt, um in einer Schaltung die Spannung zwischen zwei Punkten zu messen. Mit Monaden ist das nicht abbildbar, denn die Eingabe des `(>>=)`-Operators ist ein Wert $m\,a$ und eine Abbildung $a \to m\,b$. Damit wird mindestens ein Parameter als Wert behandelt. Es kann nicht auf die Berechnung, die zu diesem Wert geführt hat, zurückgegriffen werden.

Ein Arrow hat diese Eigenschaft nicht. In einem Arrow werden ausschließlich Berechnungen als Parameter verwendet. Hier muss genauer differenziert werden. Die Definition eines Arrows baut auf der Definition einer Kategorie auf. Als Kategorie wird alles bezeichnet, was die Funktionalität einer Identität (id) und einer Komposition ($\circ$) besitzt:

$$id : a\,b\,b$$

sowie

$$\circ : a\,c\,d \to a\,b\,c \to a\,b\,d$$

Schon die Identität ist eine Berechnung. Der Typklassenparameter a beschreibt die Form der Berechnung. Die Komposition wird aus zwei Berechnungen aufgebaut. Lediglich die Form der Berechnung (a) muss in allen Berechnungen gleich sein.

Ein Arrow fügt dieser Funktionalität einer Kategorie weitere Funktionen hinzu. Neben arr, der unit-Funktion der Arrows, wird die parallele Kombination von Berechnungen hinzugefügt.

Wie bei Monaden ist nicht viel erforderlich, um einen Arrow zu erzeugen. Die Kombination eines Typs mit einer Identitätsfunktion, einer Kombinationsfunktion, einer Unit-Funktion sowie der parallelen Verknüpfung ist bereits ein Arrow.

$$arr : (b \to c) \to a\, b\, c$$
$$(* * *) : a\, b\, c \to a\, b'\, c' \to a\ b \times b'\ c \times c'$$

Zusätzlich ist das Konzept eines Arrows allgemeiner als das einer Monade. Es gibt mehr „Dinge“, die Arrow sind, als es „Dinge“ gibt, die Monaden sind. Zu jeder Monade kann ein Arrow angegeben werden, ein sogenannter Kleisli-Arrow [33, 32].

Abschnitt 5.1 verdeutlicht, dass es mit Monaden schwieriger ist, Parallelitäten zu beschreiben, denn die Monadenklasse sieht keine Funktion für parallele Verknüpfungen vor. Die wirkliche Schwierigkeit von Monaden ergibt sich aus ihrem Charakter, mit berechneten Werten zu hantieren. Der erste Parameter wird immer als Wert übergeben. Also müsste die Berechnung dieses Wertes abgeschlossen sein, um die nächste Methode aufzurufen. Wollte man parallele Verarbeitung mit Monaden gestalten, wäre es notwendig, Monaden mehrschichtig zu gestalten. Der berechnete Wert müsste entweder in einer weiteren Monade verarbeitet werden, die es ermöglicht, parallel Werte zu verarbeiten, oder der Wert müsste in einer weiteren Monade zwischengespeichert werden, sodass die parallele Verarbeitung durch sequentielle Verarbeitung simuliert würde.

Arrows hingegen beschreiben Berechnungen, nicht Werte. Das führt dazu, dass Werte, die später innerhalb der Arrows verwendet werden, als Berechnung eingefügt werden müssen [1].

iDie Abbildung 5.1 zeigt, wie die Eingabe x durch den Arrow $\text{arr}(\lambda x \to (x, x))$ in zwei Ströme aufgeteilt wird. Dabei wird mit $\text{first}\, f$ die Funktion f auf den ersten

[1] z. B. als `const 4`

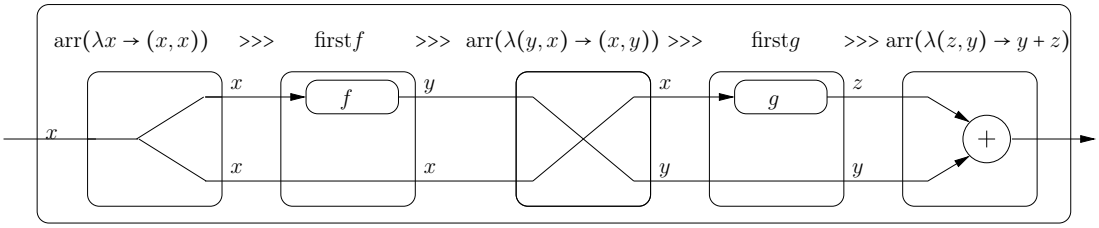

Abbildung 5.1.: Symbolbild eines Datenflusses mithilfe von Arrows

der beiden Ströme angewandt. Das Resultat y wird mit dem zweiten Datenstrom vertauscht und mit first g die Funktion g ebenfalls auf x ausgeführt. Dieses Resultat, z, fließt zusammen mit y in den letzten Arrow $\text{arr}(\lambda(z, y) \to y + z)$ ein. Hier werden beide Werte addiert.

Um einen Arrow spezifizieren zu können, muss zunächst eine Kategorie definiert werden. Dabei ist eine Kategorie einfach jeder Datentyp, für den eine Identitätsfunktion und eine Funktionskomposition angegeben werden können:

```
1 class Category cat where
2   id :: cat a a
3   (.) :: cat b c -> cat a b -> cat a c
```

Ein Arrow ist jeder Datentyp, der die folgende Spezifikation erfüllt:

```
1 class Category a => Arrow a
2   arr :: (b -> c) -> a b c
3   (***) :: a b c -> a b' c' -> a (b, b') (c, c')
4   (&&&) :: a b c -> a b c' -> a b (c, c')
5   first :: a b c -> a (b, d) (c, d)
6   second :: a b c -> a (d, b) (d, c)
```

Ein minimaler Arrow ist schon durch die Spezifikation der Funktionen `arr` und `first` definiert. Alle weiteren Funktionen lassen sich aus diesen beiden ableiten. Die Funktion `arr` ermöglicht es, eine beliebige Funktion in einem Arrow zu befördern. Die Funktion `first` erzeugt aus einem Arrow einen weiteren, der zwei Werte als Eingabe annimmt. Der übergebene Arrow wird aber nur auf den ersten Wert angewandt, der zweite Wert wird von dem Arrow nicht angetastet.

Die Funktion `(***)` stellt die parallele Auswertung zur Verfügung. Der Funktion

werden zwei Arrows übergeben. Daraus wird ein neuer Arrow erzeugt, der auf zwei Eingaben arbeitet. Im Diagramm 5.1 wäre dies durch einen Rahmen symbolisiert, der nicht nur eine, sondern zwei Funktionen enthält.

5.3. Arrows: zwischen „Deep“- und „Shallow“-Einbettung

Bei der Verwendung einer Programmiersprache zur Beschreibung von Hardware gibt es zwei verschiedene Systeme. Zum einen existiert die Programmiersprache, in der die Hardware ausgedrückt wird. Zum anderen ist da die Hardware, die in einer zweiten Sprache beschrieben wird. Diese zweite Sprache kann eine Netzliste, eine HDL wie VHDL oder aber ein Diagramm sein.

Letztlich handelt es sich immer um eine Form der Metaprogrammierung. Metaprogrammierung ist der Prozess, ein Programm zu entwerfen, welches selber wieder ein Programm als Resultat liefert. Befinden sich beide Programme in derselben Sprache, spricht man von homogener Metaprogrammierung; sind Wirts- und Gastsprache unterschiedlich, spricht man von heterogener Metaprogrammierung [30].

Bei der Hardwarebeschreibung fungiert die Programmiersprache als Wirtssprache, mit der die Gastsprache, also die Hardware, beschrieben wird. Die Beschreibung der Gastsprache kann auf zwei unterschiedliche Weisen erreicht werden. Zum einen kann das Wirtssystem verwendet werden, um Ausdrücke des Gastsystemes zu erzeugen. In diesem Fall spricht man vom seichter Einbettung. Zum anderen können die Ausdrücke des Gastsystems, z. B. als Datentypen, im Wirtssystem integriert werden. Hier spricht man von tiefer Einbettung.

Wird eine Gastsprache seicht in die Wirtssprache eingebettet, wird an die Wirtssprache keine Anforderung hinsichtlich ihrer Ausdrucksstärke gestellt. Somit kann eine noch so simple Sprache als Wirtssprache fungieren. Lediglich praktische Gesichtspunkte beschränken die Wahl der geeigneten Wirtssprache. Der Nachteil dieser Form der Einbettung ist, dass es keine Möglichkeit gibt, die Korrektheit der erzeugten Ausdrücke in der Gastsprache zu verifizieren. Beispielsweise können Ausdrücke als reiner Text in der Software gespeichert sein. Es kommt vor, dass Datenbank-Abfragen als Zeichenketten in der Software abgelegt sind und diese zur Laufzeit verändert werden. In der Wirtssprache kann nicht sichergestellt werden, dass die neue Abfrage

weiterhin korrekt ist.[2]

Demgegenüber steht die Methode, die Gastsprache tief in die Wirtssprache einzubinden. Hierzu müssen Datentypen im Wirtssystem definiert werden, die Ausdrücke des Gastsystems beschreiben können. Dieser Ansatz verlangt, dass die Wirtssprache mindestens so ausdrucksstark ist wie die Gastsprache. Sämtliche Mittel des Wirtssystems können genutzt werden, um Ausdrücke im Gastsystem zu erzeugen, da diese für die Wirtssprache ebenfalls „nur“ Daten sind. Zusätzlich kann in diesem Fall sichergestellt werden, dass die erzeugten Ausdrücke auch korrekt im Gastsystem sind. Die Schwierigkeit liegt darin, dass das Wirtssystem das Gastsystem umfänglich beschreiben muss. Das kann erheblichen Aufwand im Wirtssystem nach sich ziehen.

Arrows bieten die Möglichkeit, einen Mix der beiden Einbettungsformen tiefer und seichter Einbettung zu verwenden. Mit Arrows wird eine Gastsprache seicht in die Wirtssprache eingebettet, da z. B. Konstrukte der Gastsprache im Arrow abgelegt werden können. Daneben wird dieselbe Methode (Arrows) ebenfalls verwendet, um Daten bestehender Typen zu erzeugen. Diese Typen enthalten dann aber die Eigenschaften des zu beschreibenden Systems, der Hardware. Auf dieser Ebene werden Arrows angewandt, um tiefe Einbettung zu erzielen. Die Schärfe des Hardware beschreibenden Datentyps legt hierbei fest, wie tief der Entwurf in die Wirtssprache eingebettet wird. Es ist auch das Maß der seichten Einbettung bestimmbar, denn beim Entwurf ist dem Entwickler überlassen, wie präzise er Arrows nutzen möchte, um zu den Hardware beschreibenden Datentypen zu gelangen.

5.4. Argumentfreie Schreibweise

In diesem Abschnitt wird beschrieben, wie die Essenz zur Arrow-Kombination erarbeitet wird. Es wird die direkte, *argumentfreie* [3] Schreibweise vorgestellt. Daneben existiert auch eine syntaktisch bildlichere Schreibweise, die *proc*-Notation. Obwohl die argumentfreie Darstellung von Arrows verwirrend wirken kann, wird sie hier vorgestellt, da sie notwendig ist, um die Mechanik der Arrows zu verstehen.

Üblicherweise werden Funktionen miteinander kombiniert, indem eine Funktion mit Argumenten aufgerufen wird. Die Rückgabe wird abgelegt und steht so dem nächs-

[2] Gemeint sind hier z. B. SQL-Injection-Angriffe [64]
[3] point free

ten Funktionsaufruf zur Verfügung. Dieses Vorgehen hat seinen Nutzen vor allem darin, dass die Rückgabe in Einzelteile zerlegt werden kann. Wenn eine ganze Reihe von Funktionen miteinander kombiniert werden soll, bei denen sich die Aus- und die Eingaben nicht unterscheiden, ist dieses Vorgehen bestenfalls mühsam. Es müssen immer wieder die Zwischenwerte temporär abgelegt werden, nur damit sie für den nächsten Funktionsaufruf zur Verfügung stehen. Einfacher ist es, die Ausgabe einer Funktion direkt mit der Eingabe einer weiteren Funktion zu verknüpfen. Mit der argumentfreien Schreibweise lässt sich so dieser Zwischenschritt überspringen. Teilen also zwei Funktionen ein gemeinsames Interface, so lassen sich beide einfach kombinieren. Ein gemeinsames Interface haben zwei Funktionen dann, wenn eine Funktion die Ausgabe erzeugt, die von der zweiten Funktion als Eingabe erwartet wird. Dies ist üblich bei der Kombination mathematischer Funktionen. Es wird hier auch von Funktionskompostion gesprochen.

Als Beispiel wird hier die Umrechnung von zwei Temperatureinheiten aufgeführt. Bei der Umrechnung von Celsius nach Fahrenheit ($T_{fahrenheit} = T_{celsius} * \frac{9}{5} + 32$) wird implizit die argumentfreie Schreibweise genutzt. Die Formel lässt sich in drei Einzelfunktionen aufteilen, mit: $f_1(x) = x * 9$, $f_2(x) = \frac{x}{5}$ sowie $f_3(x) = x + 32$. Damit ist sie die Verknüpfung folgender drei Funktionen: $T_{fahrenheit} = f_3 \circ f_2 \circ f_1$.

Sofern die Funktionen als Arrows vorliegen, lässt sich die Umrechnung argumentfrei angeben:

```
1 aCelsius2Fahrenheit :: Float -> Float
2 aCelsius2Fahrenheit
3   =     aF1 -- Multipliziere Input mit 9
4     >>> aF2 -- Dividiere Input mit 5
5     >>> aF3 -- Addiere 32 zum Input
```

Zu beachten ist hier, dass die Reihenfolge in der Arrow-Schreibweise genau umgekehrt der mathematische Schreibweise ist. Das liegt daran, dass bei der mathematischen Schreibweise von innen nach außen ausgewertet wird, wobei $f_3 \circ f_2 \circ f_1$ für $f_3(f_2(f_1(x)))$ steht. Bei der Arrow-Schreibweise wird von vorne nach hinten ausgewertet. Beide Berechnungen beginnen mit der f_1-Funktion und enden mit der f_3-Funktion. Es lässt sich festhalten, dass der Rückgabetyp von `aF1` gleich dem Eingabetyp von `aF2` ist; der Rückgabetyp von `aF2` entspricht wiederum dem Eingabetyp von `aF3`.

Die argumentfreie Darstellung ist jene Darstellungsform, auf welche die Arrows abgebildet werden. Es gibt eine weitere Darstellungsform, die proc-Notation. Diese wird im nächsten Abschnitt genauer erläutert. Letztlich wird die proc-Notation auf die hier gezeigte argumentfreie Schreibweise zurückgeführt. Das bedeutet, dass nur mithilfe der argumentfreien Darstellung nachvollzogen werden kann, was der Präprozessor des Kompilers aus einem angegebenen Arrow macht.

5.5. Beschreibung der proc-Notation

Neben der argumentfreien Darstellung von Arrows gibt es eine weitere Darstellungsform, die proc-Notation [49]. Die proc-Notation hat gegenüber der argumentfreien Darstellung den Vorteil, dass sie sehr bildlich darstellt, wie der Arrow aufgebaut ist. Zur Angabe komplexer Arrows ist die proc-Notation verständlicher.

In der proc-Notation steht ein Arrow – bildlich – zwischen Ein- und Ausgabe. Die Ausgabe wird links neben dem Arrow notiert, die Eingabe rechts vom selben. Anfang und Ende eines in proc-Notation dargestellten Arrows werden durch Einrückung oder durch Klammerung markiert. Ein proc-Notations-Block beginnt mit dem Schlüsselwort `proc`. Diesem Schlüsselwort folgt eine Variable [4], welche die Eingabe des Arrows entgegennimmt. Die erste Zeile wird mit dem Schlüsselwort `do` beendet. Darauf folgt die Darstellung des Arrows.

Der Beispielarrow, der dem Celsius-Wert einen Fahrenheit-Wert berechnet, wird in proc-Notation wie folgt angegeben:

```
1 aCelsius2Fahrenheit :: Float -> Float
2 aCelsius2Fahrenheit
3   = proc tF -> do
4       tmp1 <- aF1 -< tF     -- Multipliziere Input mit 9
5       tmp2 <- aF2 -< tmp1   -- Dividiere Input mit 5
6       tmp3 <- aF3 -< tmp2   -- Addiere 32 zum Input
7       returnA     -< tmp3   -- gebe Ergebnis zurueck
```

Betrachtet man ein Datenflussdiagramm (Abbildung 5.2) dieses Arrows, so zeigt sich, dass die proc-Notation den Datenfluss textuell direkt umsetzt.

[4] anstelle einer Variablen ist auch ein Muster gültig, welches die Eingabe aufnimmt

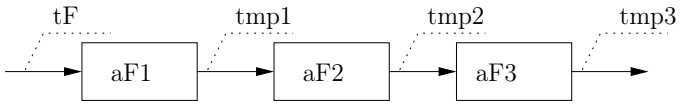

Abbildung 5.2.: Datenfluss des Arrows: aCelsius2Fahrenheit

Die proc-Notation ist ein syntaktischer Kniff. Ein in der proc-Notation dargestellter Arrow wird immer wieder in die argumentfreie Schreibweise überführt. Beide Schreibweisen sind damit untereinander austauschbar.

5.6. Tupellisten

Im Abschnitt 7.1 werden CRC-Bausteine definiert. Sie enthalten eine Reihe von Bits als Eingabe, welche sich idealerweise über eine listenähnliche Struktur abbilden lassen. Das Interface der Arrows basiert auf Tupeln, was dazu führt, dass binäre Operatoren bevorzugt abgebildet werden können.

Eine listenähnliche Struktur lässt sich mithilfe von Tupeln simulieren. Im Folgenden wird beschrieben, wie Operatoren aussehen, mit denen eine solche listenähnliche Struktur erzeugt wird. Es wird ebenfalls gezeigt, wie mit der Tupelliste gearbeitet wird.

Eine Liste ist die Kombination eines Elements mit einer Restliste (siehe Abschnitt 3.4). Restliste kann auch die leere Liste sein, wodurch die rekursive Struktur die Möglichkeit besitzt, beendet zu werden. Ein Tupel besteht aus zwei Elementen. Um daraus eine Tupelliste zu erzeugen, lässt sich ein Wert im ersten Element ablegen. In das zweite Element wird ein weiteres Tupel geschachtelt. Beenden lässt sich diese rekursive Struktur, indem an einer Stelle im zweiten Element des Tupels kein weiteres Tupel mehr ablegt wird, sondern der letzte Wert. Es wäre auch möglich, am Ende `()` abzulegen, um das Ende der Liste explizit zu kennzeichnen. Im Folgenden wird die erste Variante verwendet.

Eine Funktion auf Tupellisten ist das Verschieben eines Wertes innerhalb der Liste. Das Verschieben eines Wertes nach „rechts“ wird vom `aFloatR` übernommen. Schieben nach „rechts“ bedeutet, dass der Wert tiefer in die Tupellistenstruktur geschoben wird.

```
aFloatR :: (Arrow a)
  => Netlist a (my, (b, rst))
               (b, (my, rst))
aFloatR
  =  aDistr
  >>> first aSnd
```

Zur Definition von `aFloatR` wird auf zwei weitere Arrows zurückgegriffen: `aDistr` und `first aSnd`. `aDistr` verteilt den vorderen Teil des Tupels über das nächste Tupel. Aus $(x, (y, z))$ wird also $((x, y), (x, z))$. Mit `first aSnd` wird der erste Teil des vorderen Tupels verworfen. Aus $((x, y), (x, z))$ wird $(y, (x, z))$. x wird in der Tupelliste um eine Position weiter verschoben. Möchte man den Wert um mehr als eine Position verschieben, muss `aFloatR` erneut aufgerufen werden, allerdings nur von der restlichen Tupelliste. Dazu wurde `(>:>)` definiert. Dieser Operator führt den folgenden Arrow nur noch auf die innere Restliste aus. `(>:>)` ist ein Spezialfall des allgemeinen Arrow-Kompositionsoperators `(>>>)`.

```
aA >:> aB = aA >>> (second aB)
```

Beim Arbeiten mit Tupellisten kommt diese Art der Strukturveränderung häufig vor. Dazu müssen Elemente miteinander vertauscht, auf ihre Nachfolger verteilt oder die Klammerung der Elemente geändert werden. Für die Arbeit mit Tupellisten, aber auch für sonstige strukturverändernde Operationen auf Tupeln, wurden die folgenden Arrows definiert. Die Definitionen entsprechen jeweils direkt der Umsetzung der

angegebenen Datentypen.

$$\begin{aligned} \text{aAssocLeft} &: (a, (b, c)) \to ((a, b), c) \\ \text{aAssocRight} &: ((a, b), c) \to (a, (b, c)) \end{aligned}$$

$$\begin{aligned} \text{aDistr} &: (a, (b, c)) \to ((a, b), (a, c)) \\ \text{aDdistr} &: ((a, b), (a', c)) \to ((a, a'), (b, c)) \end{aligned}$$

$$\begin{aligned} \text{aSwapFst} &: (a, (b, c)) \to (b, (a, c)) \\ \text{aSwapSnd} &: ((a, b), c) \to ((a, c), b) \end{aligned}$$

$$\text{aFlip} : (a, b) \to (b, a)$$

6. Beschreibung von Netzlisten mittels Arrows

6.1. Der Netlist-Datentyp

Im vorhergehenden Kapitel 5 wurde skizziert, wie Hardware mithilfe von Arrows beschrieben werden kann. Arrows existieren zwar in unterschiedlichen Ausprägungen, für die Beschreibung von Hardware ist jedoch ein neuer Datentyp notwenig. Ein einfacher Zustands-Arrow (`ArrowState`) reicht dafür nicht. Das liegt daran, dass die Hardwarebeschreibung gleichberechtigt neben dem Arrow im Datentyp stehen muss. Der `ArrowState` führt intern einen Zustand mit sich. Dieser könnte zwar die Hardwarebeschreibung führen, allerdings wäre dann die Hardware in gewisser Weise für die Generierung ihrer eigenen Beschreibung mitverantwortlich.

Bei dem hier vorgestellten Datenyp steht die Beschreibung der Hardware gleichberechtigt neben der Funktionalität der Hardware. Der sogenannte Netlist-Arrow ist der vom Autor entworfene Arrow. Allgemein ist ein `Netlist`-Arrow ein abstrakter Datentyp, der aus einem Arrow (`a b c`) und seiner Beschreibung besteht. Die Beschreibung wird dabei vom Schaltkreisbeschreibungstyp `CircuitDescriptor` übernommen. Auch dieser ist vom Autor entworfen worden. Damit stellt der `Netlist`-Datentyp ein Produkt im Sinne der Kategorientheorie dar. In Haskell werden solche Produkte über Tupel abgebildet.

```
1 newtype Netlist a b c
2   = NL (a b c, CircuitDescriptor)
```

Für die Schaltkreisbeschreibung wird der Datentyp `CircuitDescriptor` verwendet. Dieser ist ein Graph-Datentyp, der zusätzliche Informationen enthält. Der Graph-Datentyp ist ansich ein verschachtelter Graph, denn die Knoten selbst sind wieder

`CircuitDescriptor`en. Damit können in der Beschreibung die Ebenen festgehalten werden, aus denen sich die Hardware zusammensetzt.

Als Beispiel für die Zusatzinformation ist hier `Area` aufgeführt. In `Area` soll die Fläche enthalten sein, die von dem Bauteil benötigt wird. Daraus lässt sich in der Beschreibung einer neu generierten Hardware (mit)berechnen, welche Fläche die neu entworfene Hardware abdecken wird. Diese Art der Information kann nach Belieben angepasst werden. Denkbar wären auch Informationen über die Kosten der Hardware oder Informationen über den Entwickler, der das Bauteil entworfen hat.

```
1 data CircuitDescriptor
2   = MkCombinatorial
3     { nodeDesc :: NodeDescriptor
4     , nodes    :: [CircuitDescriptor]
5     , edges    :: [Edge]
6     , cycles   :: Tick
7     , space    :: Area
8     }
```

Der Bezeichner `NodeDescriptor` enthält die Informationen, die für unterschiedliche Komponenten von gleicher Bedeutung sind. Dazu zählen der Name der Komponente, ihre Kompoenten-ID sowie die Listen der Ein- und Ausgabepins.

```
1 data NodeDescriptor
2     = MkNode
3       { label   :: String
4       , nodeId  :: ID
5       , sinks   :: Pins
6       , sources :: Pins
7       }
```

6.2. . . . eine Kategorie

Bevor der Typ `Netlist a` zu einer Arrow-Instanz werden kann, muss `Netlist a` zur Kategorie werden. Dies geht aus dem Kontext der Klassendefinition der Arrows hervor:

```haskell
class (Category a) => Arrow a where
```

Kategorien sind jene Datentypen, für die sich ein Verknüpfungsoperator (∘) sowie eine Identität angeben lassen. Die Definition der `Netlist a`-Kategorie ist daher überschaubar:

```haskell
instance (Category a) => Category (Netlist a) where
  id = id

  NL (f, cd_f) . NL (g, cd_g)
     = NL (f . g, cd_g `connect` cd_f)
```

Beim genauen Betrachten fällt auf, dass der Typ-Parameter `a` durch einen Kontext auf `a`s eingeschränkt wird, die selber schon Kategorien sind. Dies ist für die Kombination von zwei `Netlist`s von Bedeutung, also bei der Definition des ∘-Operators. Dies liegt daran, dass die Kombination in die `Netlist` hineinreicht. Für die Kombination der Schaltkreisbeschreibungen lässt sich dies angeben.

Für die Funktionalität innerhalb der `Netlist` gilt, dass diese selber kombinierbar sein muss. Diese Eigenschaft kann nicht allgemein bei der Definition des `Netlist`-Typs entschieden werden. Aus diesem Grund wird die Kombination der Funktionalitäten einer `Netlist` durch den Datentyp erzwungen.

Sollen zwei Netlists miteinander ∘-verknüpft werden und heißen die Arrows aus beiden Parametern `f` und `g`, so ist die resultierende Kategorie `f . g`. Diese Verknüpfung ist möglich, da `f` und `g` selber Kategorien sind. Sind die Schaltkreisbeschreibungen $\texttt{cd}_f$ und $\texttt{cd}_g$, so berechnet die Funktion `connect` daraus eine neue Schaltkreisbeschreibung. `connect` ermöglicht das sequentielle Verbinden von Schaltkreisbeschreibungen. Diese Funktion wird in Abschnitt 6.6 definiert.

Im nächsten Schritt wird die Arrow-Instanz von `Netlist a` implementiert.

6.3. ... ein Arrow

Ein Arrow besteht aus den Funktionen `arr`, `first` und `second` sowie den Operatoren `(***)` und `(&&&)`. Eine minimale vollständige Definition ist schon durch

die Funktionen `arr` und `first` gegeben. Die weiteren Funktion lassen sich aus diesen beiden ableiten. Um aber Kontrolle über die Implementierung der Funktion zu behalten, ist es notwendig, die Implementierung aller Funktionen anzugeben.

Auch hier wird das Typsystem verwendet, um sicherzustellen, dass das erste Element der `Netlist`s ebenfalls ein Arrow ist. Hier wird über den Kontext eine stärkere Einschränkung vorgenommen, als von der Arrow-Klasse verlangt. Laut Arrow-Klasse wird lediglich erwartet, dass zum Arrow werden kann, was eine Kategorie ist. Das reicht für den `Netlist`-Typ nicht aus, da eine `Netlist` nur beschreibende Informationen zu einem bestehenden Arrow hinzufügt.

Da es sich bei dem ersten Parameter des Produktes schon um einen Arrow handelt, werden die Arrow-Funktionen hier an den inneren Arrow durchgereicht. Für den `CircuitDescriptor` werden eigene Funktion definiert. Die Arrow-Definition des `Netlist a`-Typs sieht wie folgt aus:

```
1  instance (Arrow a) => Arrow (Netlist a) where
2    arr f
3      = NL (arr f, error "no such function")
4
5    first (NL (f, cd_f))
6      = NL ( first f
7           , cd_f `combine` idCircuit
8           )
9
10   second (NL (g, cd_g))
11     = NL ( second g
12          , idCircuit `combine` cd_g
13          )
14
15   NL (f, cd_f) &&& NL (g, cd_g)
16     = NL ( f &&& g
17          , cd_f `combine` cd_g
18          )
19
20   NL (f, cd_f) *** NL (g, cd_g)
21     = NL ( f *** g
22          , cd_f `combine` cd_g
23          )
```

Anhand dieser Definition wird erkennbar, dass die Arrow Funktionalitäten an den

Arrow innerhalb des `Netlist`-Typs weitergereicht werden. Weiter werden die dem `CircuitDescriptor` entsprechenden Funktionen aufgerufen. Letztlich beschränkt sich das auf die `combine`- und die `connect`-Funktion, die im Abschnitt 6.7 beschrieben werden.

Durch die Angabe von Präzedenzen wird die Klammerung der Operatoren festgelegt, ohne dass der Programmierer diese Klammern angeben muss. Das dient vor allem der Lesbarkeit des Quellcodes. Die folgenden Präzedenzen sind den Standardbibliotheken entnommen:

```
1 infixr 3 ***
2 infixr 3 &&&
3 infixr 9 .
4 infixr 1 >>>, <<<
```

6.4. Das arr-Problem

Ein Arrow beinhaltet mit der `arr`-Methode einen Weg, um eine einfache Funktion zu einem Arrow umzuformen. Dies birgt ein Problem, denn Hardware wird spontan erzeugt, sobald ein `arr`-Aufruf stattfindet. Lösungsansatz wäre eine Funktion, die den Typ der übergebenen Funktion zum Zeitpunkt der Typüberprüfung ermitteln kann. Falls das möglich wäre, könnte daraus eine fundamentale Schaltkreisbeschreibung abgeleitet werden. Eine solche Funktion müsste den folgenden Datentyp besitzen:

```
1 ... :: (b -> c) -> CircuitDescriptor
```

Aufgrund der Curry-Howard-Entsprechung [60] stellt der Typ eine Behauptung dar. Diese Behauptung kann für Hardware aber nicht bewiesen werden. Es wäre vielleicht denkbar, die Hardwarebeschreibungen derjenigen Funktionen ableiten zu lassen, die tatsächlich eine Entsprechung in Hardware besitzen. Es gibt mehr Funktionen in Software, als diese in Hardware gesetzt werden können. Das wird deutlich, wenn man überlegt, dass b und c beliebige Typvariablen sein können. Sie können damit selber wieder Funktionen darstellen. In Hardware würde das bedeuten, dass es Bauteile gäbe, die Bauteile als Parameter erhalten oder Hardware als Resultat zurückliefern. Das ist unmöglich.

Adam Megacz hat dasselbe Problem erkannt und erläutert, dass die `arr`-Funktion besagt, dass die Gastsprache eine Obermenge der Wirtssprache darstellen muss [39]. Er schlägt „generalisierte Arrows“ als Lösung des Problems vor. Für den in dieser Arbeit beschriebenen Sachverhalt sind generalisierte Arrows nicht die richtige Lösung. Für generalisierte Arrows muss das gesamte Arrow-Interface aufgegeben werden.

Zur Beschreibung von Hardware ist es nicht notwendig, jede beliebige Funktion spontan in Hardware übersetzen zu können. Auch in anderen HDLs ist dies nicht gegeben. In VHDL beispielsweise wird Hardware aus gültigen Bausteinen aufgebaut, die in Bibliotheken zu finden sind. Dazu gehören z. B. **IEEE_STD_LOGIC_1164** oder **IEEE_NUMERIC_STD**. Auch die vom Autor vorgestellte Lösung favorisiert einen Bibliothek-Ansatz.

6.4.1. Bibliotheken versus spontan erzeugter Hardware

Bibliotheken verhalten sich etwa wie Lego-Bausteine. Dem Entwickler steht ein fester Satz an unterschiedlichen Bausteinen zur Verfügung. Aus diesen kann er die neue Hardware definieren. Außerdem ist es untersagt, neue Bauteile zu definieren, oder dies wird nur eingeschränkt ermöglicht. So wird unterbunden, dass die „neuen“ Bauteile nicht zueinander passen. Übertragen auf Netzlisten-Arrows bedeutet das, dass nur Arrows verwendet werden können, die eine gültige Beschreibung besitzen. Für den Bibliothek-Ansatz wird lediglich die Verwendung der `arr`-Funktion während der Defintion einer Hardware-Komponente verboten. Daraus folgt, dass ein Entwickler nur aus existierenden Bauteilen ein neues erzeugen kann.

Neue Bauteile werden durch die Funktion `augment` erzeugt, die im nächsten Abschnitt genauer beschrieben wird. Das Verbot der `arr`-Funktion zieht nach sich, dass die proc-Notation nur noch sehr eingeschränkt verwendet werden kann. Diese Notation setzt oft unnötigerweise auf die spontane Erzeugung von Hardware. Der folgende proc-notierte Arrow

```
1 proc x -> do
2   y <-  f -< (x+1)
3         g -< (2*y)
4   let z = x + y
5   t <-  h -< (x*z)
6   returnA -< (t+z)
```

wird übersetzt zu:

```
1     arr (\x -> (x,x))
2  >>> first (arr (\x -> (x+1)) >>> f)
3  >>> arr (\(x, y) -> (y, (x, y)))
4  >>> first (arr (\y -> (2*y))   >>> g)
5  >>> arr snd
6  >>> arr (\(x,y) -> let z = x + y in ((x, z), z))
7  >>> first (arr (\(x, z) -> (x*z)) >>> h)
8  >>> arr (\(t, z) -> (t+z)
9  >>> returnA
```

An dem Beispiel lässt sich erkennen, dass der Aufruf von `arr` in Zeile 1 unnötig ist. Dieser Aufruf kann direkt durch einen Aufruf des `aDup`licate-Arrows ersetzt werden. Die `arr`-Aufrufe in den Zeilen 2, 4, 6, 7 und 8 basieren auf den verwendeten Berechnungen der rechten Seite der proc-Notation. Eine Einschränkung dieser Berechnungen könnte die benötigten `arr`-Statements drastisch verringern. Der Aufruf in Zeile 3 lässt sich vollständig durch den Arrow `aDup >>> first aFst` ersetzen. Der Aufruf in Zeile 5 kann durch `aSnd` ersetzt werden.

Leider ist proc-Notation fest in den GHC-Kompiler eingebaut, sodass keine einfache Ersetzung durch überarbeitete Präprozessordirektiven möglich ist. Dies wäre nur mit erheblichem Aufwand möglich. Aus diesem Grund werden in der weiteren Ausführung jeweils ein proc-notierter Arrow und einer in der argumentfreien Schreibweise angegeben.

6.5. Vom Arrow zum Netlist-Arrow

Ein Element des Typs `Netlist a` wird als Bauteil bezeichnet und besteht aus einer Funktion und einer strukturierten Beschreibung der Funktion.

Um von einer Funktion zu einem Bauteil zu gelangen, muss die Funktion mit Zusatzinformationen angereichert werden. Die Zusatzinformationen befinden sich in einem `CircuitDescriptor`. Dieser wird zusammen mit der Funktion in einen `Netlist`-Arrow überführt. Diese Aufgabe wird von der Funktion `augment` übernommen.

```
1  augment :: (Arrow a) => CircuitDescriptor -> a b c -> Netlist a b c
```

```
augment cd f = NL (f, cd)
```

Der Quellcode zeigt, dass `augment` der Typkonstruktor des `Netlist`-Typs ist. Korrekterweise handelt es sich um den „gecurryten“ und „geflippten“ `NL`-Typkonstruktor. Da dieser Typkonstruktor keine Funktion, sondern einen Arrow als einen Parameter erwartet, erwartet dies auch die `augment`-Funktion.

Man sieht, dass an dieser Stelle der `arr`-Operator nicht benötigt wird. Aus den übergebenen Parametern wird direkt ein `Netlist`-Arrow erzeugt. So wird mithilfe der `augment`-Funktion dem Arrow eine Schaltungsbeschreibung zugeordnet. Dies ist anders als bei der schlichten Verwendung der `arr`-Funktion des `Netlist`-Arrows.

6.6. Verdrahtungsschemata

Im vorhergehenden Abschnitt 6.5 wurde gezeigt, dass `Netlist` ein Arrow ist. Es wurde vom Typsystem gefordert, dass der Funktionsteil des `Netlist`-Tupels schon ein Arrow sein muss. Für den beschreibenden Teil wurde auf die Funktionen `connect` und `combine` zurückgegriffen. Beide Funktionen sind im Modul `Circuit.Splice` definiert, dem Modul, das sich um das Verbinden von Einzelstücken kümmert. Die Bezeichnung „spleißen“ deutet darauf hin, dass wie beim physischen Vorbild Einzelteile zu einem neuen Ganzen zusammengefügt werden.

`connect` steht für die sequentielle Verbindung von zwei Schaltungen; `combine` stellt die Funktion für die parallele Kombination zweier Schaltungen dar. Hinter beiden Funktionen verbergen sich die Verdrahtungsschemata von Schaltungsbeschreibungen.

Das Modul `Circuit.Splice` stellt verschiedene Funktionen zur Verfügung, um Schaltungen zusammenzufügen. Das Prinzip des Zusammenfügens ist in der Funktion `splice` festgehalten. Der Datentyp von `splice` besagt, dass eine neue Schaltkreisbeschreibung erzeugt wird.

```
splice
  :: ( ( CircuitDescriptor -> CircuitDescriptor
       -> ([Edge], (Pins, Pins))
       )
     , String
```

```
6       )
7     -> CircuitDescriptor -> CircuitDescriptor
8     -> CircuitDescriptor
```

Zur Erzeugung der neuen Schaltungsbeschreibung werden die Beschreibungen der Ausgangsschaltungen benötigt, orthogonal dazu eine weitere Funktion und ein Bezeichner. Die weitere Funktion wird im Folgenden als Verdrahtungsschema bezeichnet; sie heißt in der Implementierung `rewire`. Die Bezeichnung wird zur Erzeugung einer Debug-Darstellung verwendet.

`splice` überprüft die Eingabedaten darauf, ob eine der beiden eingehenden Beschreibungen keine Beschreibung (`NoDescriptor`) ist. In diesem Fall verhält sich `splice` als Identitätsfunktion gegenüber der anderen Beschreibung.

```
1 splice _ sg NoDescriptor = sg
2 splice _ NoDescriptor sg = sg
```

Wurden tatsächlich zwei Schaltkreisbeschreibungen an `splice` übergeben, so werden diese mit dem übergebenen Tupel an die nächste Funktion `splice'` weitergereicht. Diese erwartet, genau wie von `splice` übergeben, das Verdrahtungsschema `rewire` und beide Schaltungsbeschreibungen.

Bei den Beschreibungen von Schaltungen handelt es sich um eine Graphstruktur. Somit erzeugt das Kombinieren beider Graphen neue Kanten. Zusätzlich zu der einfachen Graphstruktur besitzen die Hardwarekomponenten Pins, an denen Kanten „andocken“ können. Bei der Kombination dieser Schaltungsbeschreibungen wird berechnet, welche Pins nach der Verdrahtung noch nicht verbunden wurden. Die Pins stellen dann nach außen hin das neue Hardwareinterface dar.

`rewire` ist eine Funktion, die aus zwei Schaltungsbeschreibungen die neuen Kanten sowie die übrigen Pins berechnet. Die unterschiedlichen Varianten, Schaltungen zu verbinden, werden im Abschnitt 6.7 genauer beschrieben.

`splice` fügt die erzeugten Kanten dem neuen Graphen hinzu und setzt dabei die Pins, die bisher nicht verdrahtet sind, an die entsprechende Stelle im Datentyp.

Jede Schaltung wird durch eine Komponentennummer (`CompID`) eindeutig identifiziert. Beim Zusammenfügen von zwei Schaltungen muss darauf geachtet werden, dass in der Nummerierung keine Zweideutigkeiten entstehen.

Dazu werden die Subkomponenten mithilfe der Funktion `alterCompIDs` mit neuen Komponentennummern versehen. Die Komponenten eines Schaltkreises werden, angefangen mit 1, durchnummeriert. Für den zweiten Schaltkreis wird die Nummerierung mit der Anzahl der Komponenten (`maxCompID`) des ersten Schaltkreises +1 begonnen. Die so veränderten Beschreibungen `cd_f'` und `cd_g'` werden an die nächste Funktion (`rewire`) übergeben. Daraus ergeben sich die neuen Kanten `es` sowie die übrigen Pins der Quellen `srcs` und die übrigen Pins der Senken `snks`.

Diese lokalen Definitionen fließen in die Erzeugung der neuen Schaltkreisbeschreibung ein.

```
1  splice' (rewire, s) cd_f cd_g
2    = MkCombinatorial
3      { nodeDesc
4          = MkNode
5            { label   =    (label.nodeDesc $ cd_f')
6                        ++ s
7                        ++ (label.nodeDesc $ cd_g')
8            , nodeId  = 0
9            , sinks   = srcs
10           , sources = snks
11           }
12       , nodes   = cd_f': cd_g' : []
13       , edges   = es
14       , cycles  = (cycles cd_f) + (cycles cd_g)
15       , space   = (space  cd_f) + (space  cd_g)
16     }
17   where
18     cd_f' = alterCompIDs 1 cd_f
19     cd_g' = alterCompIDs (maxCompID cd_f' +1) cd_g
20     (es, (srcs, snks)) = rewire cd_f' cd_g'
```

Als Verdrahtungsschemata werden Funktionen bezeichnet, die festlegen, wie Hardwarebausteine miteinander verbunden werden. An dieser Stelle zeigt sich die tiefe Einbettung. Die Eigenschaften der Hardwarebausteine, die notwendig sind, um miteinander verbunden zu werden, sind in einem Datentyp `CircuitDescriptor` gebündelt. Die Verdrahtungsschemata bilden den Prozess, Bausteine physikalisch miteinander zu verbinden, ab. Dieses Vorgehen wurde in Abschnitt 5.3 als tiefe Einbettung bezeichnet.

Als `rewire`-Funktionen stehen `seqRewire` zur sequentiellen und `parRewire` zur parallelen Verdrahtung zur Verfügung. Die übergebenen Schaltkreisbeschreibungen werden zu einer Zwischendarstellung überführt. Diese besteht aus einer Liste der neuen Kanten: `[Edge]` zusammen mit den unverbundenen Eingangspins und Ausgangspins.

```
1 ...Rewire
2   :: CircuitDescriptor -> CircuitDescriptor
3   -> ([Edge], (Pins, Pins))
```

Alle `rewire`-Funktionen bauen auf einer weiteren Funktion, nämlich `wire`, auf. Das Verbinden von Komponenten mit Drähten ist unabhängig davon, ob sequentiell oder parallel verbunden werden soll. Die Typdefinition der `wire`-Funktion zeigt an, dass hier aus den beiden `CompID`s und den Ein- und Ausgangspins die angedeutete Zwischendarstellung generiert wird.

```
1 wire :: Maybe CompID -> Maybe CompID
2      -> Pins         -> Pins
3      -> ([Edge], (Pins, Pins))
```

Da `wire` zum Verdrahten an den unterschiedlichen Stellen in der Schaltungsbeschreibung verwendet werden soll, muss mit `Maybe CompID`s gearbeitet werden. Drähte, die aus einer Schaltung hinauszeigen (wie z. B. die Füßchen eines ICs), sind noch mit keiner Komponente verbunden, was sich im Datentyp widerspiegelt.

Die Zwischendarstellung ist ein Tupel aus neuen Kanten und neuen Pins. Die Pins werden durch ein Tupel der Ein- und Ausgabepins gebildet. Innerhalb von `wire` werden zunächst die Anknüpfungspunkte der zu generierenden Kanten berechnet. Ein solcher Anknüpfungpunkt besteht aus einem Tupel der Komponenten-ID und der Pin-ID. Daraus ergeben sich die Listen der Anknüpfungpunkte der linken und der rechten Seite. Beiden Listen werden mittels `zip` zu einer neuen Liste verbunden. Zuletzt wird über diese Liste der `MkEdge`-Konstruktor gezipt, sodass die Liste der neuen Kanten entsteht.

Die Anzahl der Kanten gibt vor, wie viele Anknüpfungpunkte aus den Listen der rechten und linken Seite weggelassen werden können, um die unverbundenen Anknüpfungpunkte zu erhalten.

```
wire cid_l cid_r pins_l pins_r
  = (edges, (drop cnt pins_l, drop cnt pins_r))
  where points_l = map ((,) (cid_l)) pins_l
        points_r = map ((,) (cid_r)) pins_r
        edges    = map (uncurry MkEdge)
                       (zip points_l points_r)
        cnt      = length edges
```

Bei der Verdrahtung sind fünf Fälle zu berücksichtigen, aus denen neue Kanten entstehen können. Zum einen können Kanten von außen an die linke oder an die rechte Komponente gebunden werden, (`fromOuterToL` und `fromOuterToR`). Zum anderen können von der rechten oder der linken Komponente Kanten nach außen gezogen werden (`fromLToOuter` und `fromRToOuter`). Hinzu kommen alle Kanten zwischen der rechten und der linken Komponente `edgs`. Damit sind alle möglichen Kanten definiert.

Die Ein- und Ausgabepins (`super_srcs` und `super_snks`) ergeben sich aus der Anzahl der Pins der linken Komponente `length.sinks.nodeDes` plus der Anzahl der übrigen Pins der rechten Komponente `length snks_r'`. Dies gilt, nur anders herum, auch für die Ausgangspins. −1 wird gerechnet, da von 0 an gezählt wurde.

```
seqRewire cd_l cd_r
  = (    fromOuterToL
      ++ fromOuterToR
      ++ edgs
      ++ fromRToOuter
      ++ fromLToOuter
    , (super_srcs, super_snks)
    )
  where
    (edgs, (srcs_l', snks_r'))
      =  wire (Just $ nodeId.nodeDesc $ cd_l)
              (Just $ nodeId.nodeDesc $ cd_r)
              (sources.nodeDesc $ cd_l)
              (sinks.nodeDesc   $ cd_r)
    super_srcs
      = [0..(length.sinks.nodeDesc   $ cd_l) + length snks_r' -1]
    super_snks
      = [0..(length.sources.nodeDesc $ cd_r) + length srcs_l' -1]
    ( fromOuterToL, (super_srcs', _))
```

```
20        = wire Nothing
21              (Just $ nodeId.nodeDesc $ cd_l)
22              super_srcs
23              (sinks.nodeDesc $ cd_l)
24      ( fromOuterToR, _)
25        = wire Nothing
26              (Just $ nodeId.nodeDesc $ cd_r)
27              super_srcs'
28              (drop (length fromOuterToL) $ sinks.nodeDesc $ cd_r)
29      ( fromRToOuter, (_, super_snks'))
30        = wire (Just $ nodeId.nodeDesc $ cd_r)
31              Nothing
32              (sources.nodeDesc $ cd_r)
33              super_snks
34      ( fromLToOuter, _)
35        = wire (Just $ nodeId.nodeDesc $ cd_l)
36              Nothing
37              (drop (length fromRToOuter) $ sources.nodeDesc $ cd_l)
38              super_snks'
```

Für die parallele Verdrahtung (`parRewire`) werden beide Bausteine vertikal angeordnet. Die Eingänge beider Komponenten sowie deren Ausgänge werden parallel geschaltet. Hier kann eine vereinfachte Funktion (`wire_`) verwendet werden, die lediglich eine Kurzschreibweise für `fst . wire` darstellt.

Da nicht sequentiell verdrahtet werden muss, werden lediglich die eingehenden Kanten (`goingIn_edges`) sowie die ausgehenden Kanten (`goingOut_edges`) berechnet. Die Anzahl der Ein- und Ausgangspins der neuen Komponente ergibt sich aus der Summe der Pins der inneren Komponenten.

```
1  parRewire cd_u cd_d
2    = (    goingIn_edges
3        ++ goingOut_edges
4      , (super_srcs, super_snks)
5      )
6      where
7        super_srcs
8          = [0..(length $  (sinks.nodeDesc $ cd_u)
9                         ++ (sinks.nodeDesc $ cd_d)) -1]
10       super_snks
11         = [0..(length $  (sources.nodeDesc $ cd_u)
12                        ++ (sources.nodeDesc $ cd_d)) -1]
```

```
13      goingIn_edges
14        =     (wire_ Nothing
15                     (Just $ nodeId.nodeDesc $ cd_u)
16                     (super_srcs)
17                     (sinks.nodeDesc $ cd_u))
18            ++ (wire_ Nothing
19                     (Just $ nodeId.nodeDesc $ cd_d)
20                     (drop (length.sinks.nodeDesc $ cd_u) super_srcs)
21                     (sinks.nodeDesc $ cd_d))
22      goingOut_edges
23        =     (wire_ (Just $ nodeId.nodeDesc $ cd_u)
24                     Nothing
25                     (sources.nodeDesc $ cd_u)
26                     (super_snks))
27            ++ (wire_ (Just $ nodeId.nodeDesc $ cd_d)
28                     Nothing
29                     (sources.nodeDesc $ cd_d)
30                     (drop (length.sources.nodeDesc $ cd_u) super_snks))
```

Es wurde nun dargestellt, wie die Verdrahtungsschemata funktionieren und wie diese zum Zusammenspleißen von Schaltungen verwendet werden. Daraus lassen sich einfache Funktionen erzeugen, die zur Definition der Arrow-Klassen verwendet werden. Dies sind die Funktionen `connect` und `combine`.

6.7. Implementieren der Schemata durch Verbinden und Kombinieren

6.7.1. `connect`

Die Funktion `connect` dient der sequentiellen Verdrahtung der Bauteile. Der Bezeichner `connect` zeigt an, dass hier die Bausteine miteinander verbunden werden. Es werden Drähte zwischen den Bausteinen erzeugt. Der String `(>>>)` fließt in die `connect`-Funktion mit ein, um anzuzeigen, welcher Arrow-Operator für diese Verbindung verwendet wurde.

Es folgt die Definition von `connect`:

```
1  connect :: CircuitDescriptor -> CircuitDescriptor
```

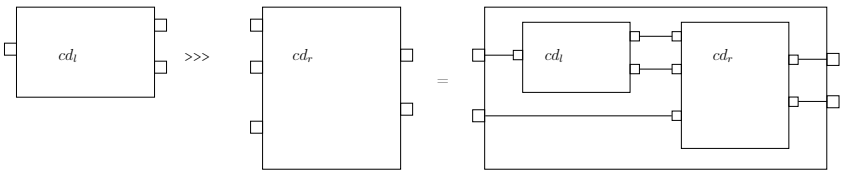

Abbildung 6.1.: Sequentielles Verdrahtungsschema

```
2         -> CircuitDescriptor
3 connect = splice (seqRewire, ">>>")
```

6.7.2. `combine`

Für die Funktion zur parallelen Verdrahtung wird der Bezeichner `combine` gewählt. Der Name soll andeuten, dass zwei Bausteine miteinander kombiniert werden, ohne sich gegenseitig zu beeinflussen. Auch die Definition ist in den folgenden Zeilen gegeben:

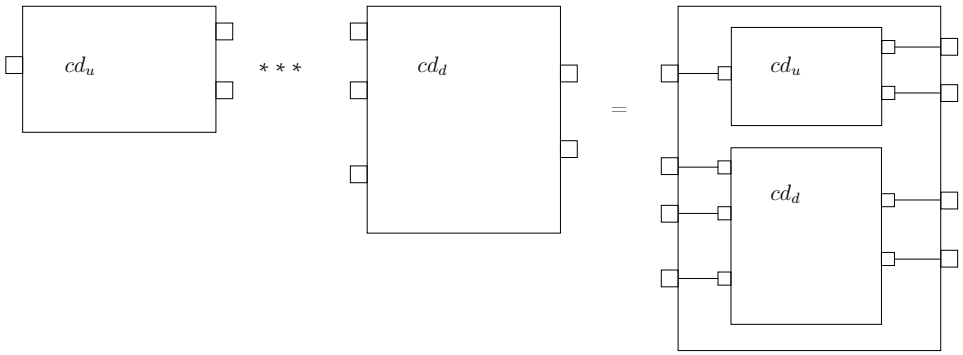

Abbildung 6.2.: Paralleles Verdrahtungsschema

```
1 combine :: CircuitDescriptor -> CircuitDescriptor
2         -> CircuitDescriptor
3 combine = splice (parRewire, "&&&")
```

6.8. Optimieren der generierten Graphstruktur

Die aus der Arrowbeschreibung erzeugte Struktur ist ein `CircuitDescriptor`, der einen Graphen darstellt. Mit einem `CircuitDescriptor` lassen sich neue Strukturen aus schon bestehenden Teilstrukturen erzeugen. Hier spiegelt sich der rekursive Aufbau in der Graphstruktur wider. Ein solcher Graph besteht aus Knoten, die durch einen eigenen Graphen beschrieben werden.

Dieser Abschnitt beschäftigt sich mit dem Bearbeiten rekursiver Graphstrukturen. Ein Graph, der aus mehreren Untergraphen besteht, lässt sich auf einen einzelnen Graphen reduzieren. Dieses Vorgehen wird hier als Glätten des Graphen bezeichnet.

6.8.1. Verschachtelte Graphen

Ein Graph ist eine Struktur, die aus Kanten und Knoten besteht. Bei der Hardwarebeschreibung durch Graphen stellen Knoten die jeweiligen Bauteile dar. Kanten stehen für die Verbindungen der Bauteile untereinander.

Der Graph als solcher enthält keinerlei Strukturelemente, die rekursiv definierbar sind. Ein Schaltkreis hingegen besteht zum einen aus der Graphstruktur der zusammenhängenden Bauteile. Daneben lassen sich Schaltkreise als Einheiten betrachten, die wiederum in andere Schaltkreise integriert werden können, sogenannte integrierte Schaltungen (ICs). Auch ein IC kann wieder aus ICs aufgebaut sein. Das lässt sich bis zur Ebene der Logik-Gatter weiterführen. Dieser Schaltkreis im Schaltkreis wird als verschachtelt bezeichnet.

Wie im Abschnitt 6.1 dargestellt, spiegelt der Datentyp `CircuitDescriptor` die Struktureigenschaft geschachtelter Graphen wider.

Als Beispiel sei die Funktion `aShiftL4XorKey` gezeigt, die Teil eines Kryptoalgorithmus ist. Der Name deutet die Funktionalität an. Es handelt sich um einen Baustein, der zwei Werte übergeben bekommt, den ersten um 4 nach links verschiebt und dann den zweiten mit `xor` verknüpft.

```
1 aShiftL4XorKey :: (Arrow a) => Netlist a (Int, Int) Int
2 aShiftL4XorKey
3   =   first (   aDup
4             >>> (aId *** (aConst 4))
```

```
5               >>> aShiftL
6               )
7   >>> aXor
```

Beim Erzeugen des Graphen aus einem Arrow entsteht eine Schaltung, die viele Verschachtelungen aufweist. Eine sequentielle Verbindung eines Bauteils mit dem Rest der Schaltung erzeugt einen neuen Graphen, der das neue Bauteil und die restliche Schaltung enthält. Diese Teilgraphen werden dann zu einem neuen Graphen zusammenführt.

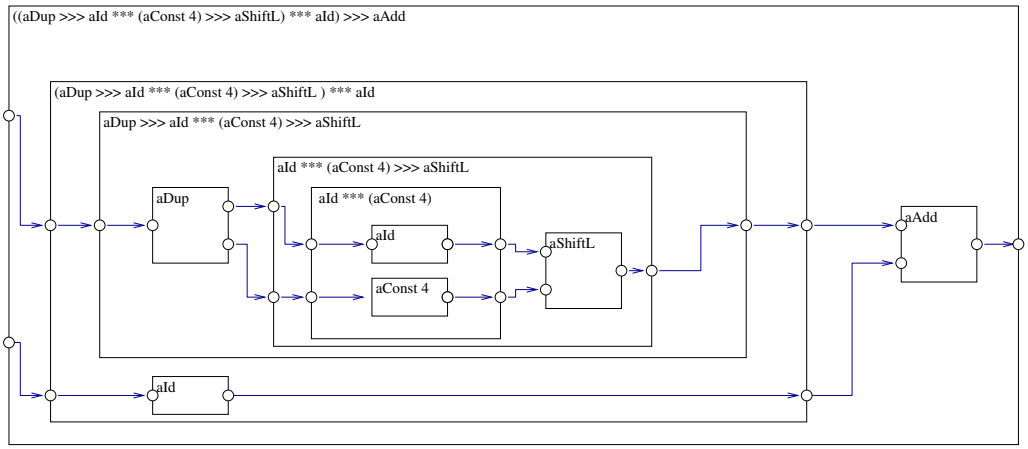

Abbildung 6.3.: Verschachtelter Graph

Da ein Graph keine Form des Verschachtelns kennt, lässt sich ein geschachtelter `CircuitDescriptor` auch ohne Schachtelung darstellen. Dem verschachtelten Graphen unterliegt eine einfachere Struktur, die denselben Graphen ausdrückt und dabei ohne die Schachtelung auskommt. Wie ein beliebiger Graph in diese „einfachste“ Struktur überführt wird, zeigt die Funktion `flatten`.

6.8.2. Optimierte Graphen

Die Funktion `flatten` stellt eine Surjektion zwischen unterschiedlichen Darstellungen desselben Schaltkreises dar. Bei der Erzeugung der Graphen aus der Arrowdarstellung wird zunächst eine Darstellungsform wie in Abbildung 6.3 generiert. Diese Darstellung lässt sich hinsichtlich der Komponentenanzahl sowie der Kantenanzahl vereinfachen. Die Funktion `flatten` übernimmt diese Aufgabe.

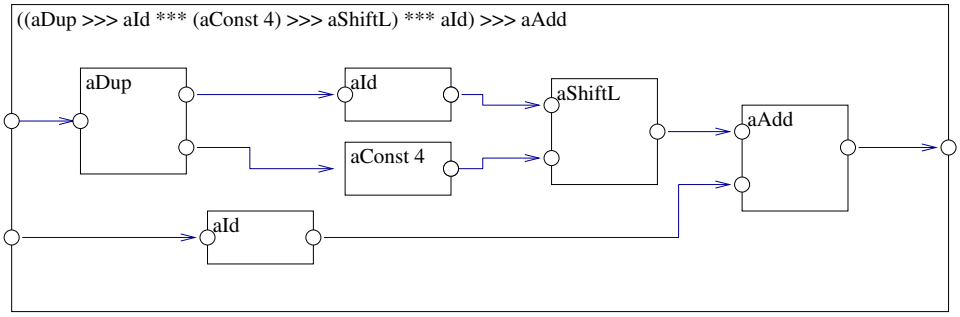

Abbildung 6.4.: Geglätteter Graph

Im Überblick wird aus der geschachtelten Darstellung 6.3 die flache Darstellung 6.4 generiert. Dies geschieht, indem alle nicht weiter teilbaren Bauteile (Atome) identifiziert werden. Es werden die aus diesen Atomen ausgehenden Kanten gesucht und so lange verfolgt, bis die Kante ein weiteres Atom erreicht hat. Die durchlaufenen Kanten lassen sich durch eine einzelne Kante ersetzen. Alle Komponenten, die nicht atomar sind, können weggelassen werden. Sie sind für die Funktionalität der Schaltung nicht von Bedeutung.

Technisch generiert man die Darstellung 6.4, indem man zunächst die Atome identifiziert. Die Liste der Komponenten aller Atome (`atomCIDs`) ist definiert durch

```
1 atomCIDs = filter isAtomic $ allCircuits g
```

wobei `g` den Ursprungsgraphen darstellt. Ein Graph ist dann atomar, wenn er keine weiteren Untergraphen enthält, also nicht weiter aufgeteilt werden kann.

Um den geglätteten Graphen zu berechnen, werden die Knoten und Kanten des `CircuitDescriptor`s `g` neu berechnet. Die Knoten des geglätteten Graphen entsprechen exakt den atomaren Knoten.

Die Kanten des geglätteten Graphen sind die abgehenden Kanten eines atomaren Knotens. Es muss darauf geachtet werden, dass diese Kanten auf Komponenten zeigen können, die im geglätteten Graphen nicht mehr vorkommen. Dazu kommen jene Kanten außerhalb des Graphen, die ebenso auf nicht mehr existierende Komponenten zeigen.

Betrachtet man zunächst nur die Kanten, die sich „innerhalb“ des geglätteten Graphen befinden, so lassen sich die notwendigen Kanten wie folgt berechnen.

Es werden alle abgehenden Kanten der atomaren Komponenten so weit verfolgt, bis sie auf ein weiteres Atom zeigen. Dazu wird die Funktion `nextAtomic` verwendet.

Die Funktion `nextAtomic` berechnet das Ziel einer übergebenen Kante `e` als ein Tupel aus Komponenten-ID und Pin-ID. Dabei gehört die Komponenten-ID des Ziels zu einer atomaren Komponente. Um diese Berechnung durchführen zu können, benötigt `nextAtomic` den „Kontext“, in dem die Berechnung durchgeführt wird, den gesamt Graphen `g`. Abhängig von der Ausgangskomponente der betrachteten Kanten `e` werden die Knoten der über- und untergeordneten „Rahmen“ abgeleitet. Dies sind: `sub`, `super` und `supersuper`

Bei der Berechnung der nächsten atomaren Komponenten-ID gibt es vier verschiedene Fälle, die betrachtet werden müssen:

(1) Die Kante zeigt bereits auf eine atomare Komponente: In diesem Fall können einfach die Komponenten-ID und die Pin-ID aus der Senke der betrachteten Kanten `e` übernommen werden.

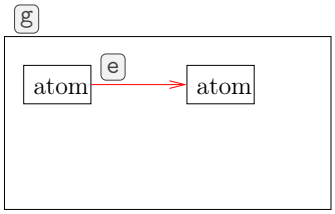

Abbildung 6.5.: Kante befindet sich zwischen zwei Atomen

(2) Die Kante zeigt aus dem Graphen `g` heraus: In diesem Fall kann als ID der nächsten atomaren Komponente die `mainID` genommen werden. Der Pin ist derselbe Pin, der in der Senke der Kante `e` angegeben wurde.

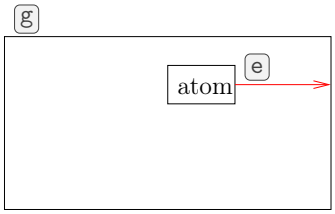

Abbildung 6.6.: Kante zeigt nach außen

(3) Die Kante zeigt aus einem Untergraphen hinaus: Anhand der Pin-ID wird

die Kante identifiziert, die von dem übergeordneten Graphen abgeht. Mit dieser Kante wird `nextAtomic` erneut aufgerufen.

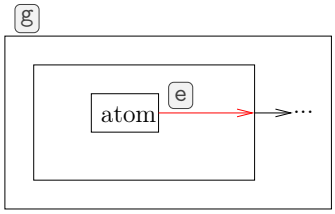

Abbildung 6.7.: Kante zeigt aus einem Subgraphen

(4) Die Kante zeigt auf eine nicht atomare Komponente: Hier wird ebenfalls anhand der Pin-ID jene Kante ermittelt, die innerhalb der nichtatomaren Komponente weiterführt. Auch hier wird `nextAtomic` mit dieser Kante erneut aufgerufen.

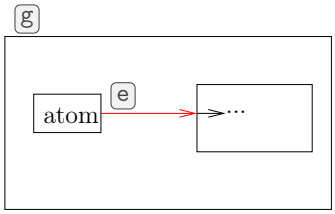

Abbildung 6.8.: Kante zeigt in einen Subgraphen

Entsprechen Quelle und Ziel einer atomaren Komponente, so wird eine neue Kante erzeugt, die alle nichtatomaren Zwischenkomponenten „überspringt“. Dies wird in einer Zwischendarstellung festgehalten. Aus der so gewonnenen Information lässt sich eine neue Kante generieren. Diese geht zwangsläufig von einem atomaren Bauteil aus und verbindet dieses mit einem weiteren Atom.

Die Funktion `nextAtomic` enthält mit Fall 2 den Sonderfall, bei dem eine Kante aus dem Graphen `g` hinauszeigt. Verfolgt werden zusätzlich zu den abgehenden Kanten der Atome noch die abgehenden Kanten der äußersten Pins des Gesamtgraphen `g`. Dadurch erhält man alle Kanten zwischen Atomen, plus der Kanten, die von außen kommen oder nach außen gehen. Diese Kanten werden als `esBetweenAtoms` bezeichnet.

Im letzten Schritt wird der äußerste Rahmen von Kanten und Knoten befreit und mit den neu berechneten Kanten und Knoten überschrieben.

```
flatten :: CircuitDescriptor -> CircuitDescriptor
flatten g
  = g { nodes = atomCIDs
      , edges = esBetweenAtoms
      }
```

6.9. Simulieren des Arrows

Da der `Netlist`-Arrow aus zwei Teilen besteht, nämlich dem Arrow sowie seiner Beschreibung, ist die Umformung zur Simulation simpel. Es wurde dazu die Funktion `toFunctionModel` definiert:

```
toFunctionModel
  :: Netlist (->) b c
  -> (b -> c)
toFunctionModel
  = runNetlist
```

`toFunctionModel` baut auf die Funktion `runNetlist` auf, die den Arrow aus dem Tupel auspackt. Diese Funktion erzeugt einen Arrow, der allein ohne seine Beschreibung zurückgeliefert wird.

```
runNetlist
  :: (Arrow a) => Netlist a b c
  -> a b c
runNetlist (NL (f, _))
  = f
```

Aufgrund der Funktion `toFunctionModel` wird vom Datentyp gefordert, dass der übergebene Arrow mit dem Funktionsanwendungsoperator `(->)` instanziiert wurde. Somit wird für den Typparameter `a` des Arrows der Funktionspfeil eingesetzt. Aus dem Arrow wird wieder jene Funktion, die einst bei der Erzeugung des Arrows hineingesteckt wurde. Da die `Netlist`-Arrows bei ihrer Erstellung jeweils auf einer tatsächlichen Funktion basieren, ist diese Rückübersetztung möglich.

6.10. Darstellung der Ausgabemechanismen

Weil der Netzlisten-Arrow einen Tupel beinhaltet und neben dem internen Arrow eine Beschreibung desselben steht, ist es möglich, aus dem Netzlisten-Arrow seine Beschreibung zu extrahieren. Der Datentyp `CircuitDescriptor` enthält diese Beschreibung. In den folgenden Unterabschnitten wird gezeigt, wie aus einer solchen Schaltkreisbeschreibung eine andere Darstellung produziert werden kann. Dabei wird der `CircuitDescriptor` selbst aus dem Netzlisten-Arrow durch die Funktion `toDescriptionModel` extrahiert:

```
1 toDescriptionModel
2   :: Netlist a b c
3   -> CircuitDescriptor
4 toDescriptionModel (NL (_, cd))
5   = cd
```

Die Funktion `toDescriptionModel` extrahiert ausschließlich den hinteren Wert des Tupels und verwirft den vorderen. Die Rückgabe, ein `CircuitDescriptor`, kann zu den unterschiedlichen Ausgabeformen überführt werden.

6.10.1. Von der Graphstruktur zur `.dot`-Notation

Der folgende Abschnitt zeigt, wie aus einem `CircuitDescriptor`-Arrow eine .dot-Datei erstellt wird. Eine .dot-Datei ist eine Datei, die eine Graphenbeschreibung nach der Dot-Spezifikation [2] enthält.

Die Syntax einer `.dot`-Datei kann der oben genannten Spezifikation entnommen werden. Dabei werden Graphen durch einzelne Knoten beschrieben. Diese können mithilfe von Kanten verbunden werden. Aus der Datei heraus können unterschiedliche Formate, wie z. B. `.ps`, `.png` oder `.svg`, generiert werden. Dies ist in der oben genannten Spezifikation detailiert beschrieben.

Die grundlegende Struktur dieser Ausgabe ist in der Funktion `showCircuit` dargestellt. Es wird eine Reihe von Strings erzeugt, die in die entsprechenden Dateien geschrieben werden. Diese `showCircuit`-Funktion erzeugt den String, der in eine `.dot`-Datei geschrieben werden kann, indem eine Liste der einzelnen Strings erzeugt wird. Diese Liste wird mit Zeilenumbrüchen versehen und zu einem vereinigten

String verbunden.

```
showCircuit :: CircuitDescriptor -> String
showCircuit g
   = concat $ map break
   [ dot_config
   , dot_outer_nodes g
   , dot_components  g
   , dot_connections g
   , "}"
   ]
```

Hinter der Funktion `dot_config` verbirgt sich ein Konfigurationsstring. Die Funktion `dot_outer_nodes` wird mit dem `CircuitDescriptor` als Parameter aufgerufen. Aus dem Descriptor werden die Anzahl der eingehenden und der ausgehenden Kanten des gesamten Bausteines ausgelesen und die jeweiligen Knoten erzeugt.

Die Funktion `dot_components` bildet für jeden Knoten im Descriptor einen Eintrag. Ebenso erzeugt die Funktion `dot_connections` für jede Kante einen entsprechenden Eintrag.

Exemplarisch sei die Funktion zur Erzeugung der Einträge für die Kanten herausgegriffen. `showEdge` formt eine Kante des Graphen in einen String um, der wiederum eine Kante symbolisiert. Es handelt sich dabei um eine benannte Kante in der Form `nodeId3:op0 -> nodeId6:ip0`. Eingehende Pins werden mit `ip` abgekürzt, ausgehende Pins mit `op`. Den Kürzeln folgt jeweils die Pin-Identifikationsnummer.

`showEdge` muss drei Fälle berücksichtigen. Dazu gehören die Fälle, in denen es eine Kante von außen oder nach außen gibt, sowie der Fall, bei dem die Kanten sich innerhalb des Graphen befinden. Für die ein- und ausgehenden Kanten fehlt jeweils die Komponentennummer der Komponente, von der die ein- bzw. die ausgehende Kante kommt. Dies wird durch einen entsprechenden `Nothing`-Wert gekennzeichnet. In diesen Fällen werden die eingehenden Kanten von den Pins des Knotens `xSTART` gezogen. Für die ausgehenden Kanten gehen diese zu den Pins des Knotens `xEND`. Diese zwei Komponenten werden von der Funktion `dot_outer_nodes` erzeugt. Ein Beispiel kann der Abbildung 7.5 entnommen werden.

Bei einer „normalen“ Kante existieren beide Komponentennummern. Die Kanten werden zwischen der Komponenten-ID `nodeId` und der Pin-ID, die der Pin-Bezeichnung

`ip` bzw. `op` folgt, verbunden.

```
showEdge :: Edge -> String
showEdge (MkEdge (Nothing, pid)          (Just snk_cid, snk_pid))
  =  "xSTART"                 ++ (':':"op") ++ show pid
  ++ " -> "
  ++ "nodeId" ++ show snk_cid ++ (':':"ip") ++ show snk_pid

showEdge (MkEdge (Just src_cid, src_pid) (Nothing, pid))
  =  "nodeId" ++ show src_cid ++ (':':"op") ++ show src_pid
  ++ " -> "
  ++ "xEND"                   ++ (':':"ip") ++ show pid

showEdge e
  =  "nodeId" ++ show src_cid ++ (':':"op") ++ show src_pid
  ++ " -> "
  ++ "nodeId" ++ show snk_cid ++ (':':"ip") ++ show snk_pid
  where
    (Just src_cid, src_pid) = sourceInfo e
    (Just snk_cid, snk_pid) = sinkInfo   e
```

Mit der Funktion `dot_outer_nodes` werden aus einem `CircuitDescriptor` die Knoten `xSTART` sowie der Knoten `xEND` erzeugt. Die Knoten bilden die Kanten des Graphen ab, welche von außen kommen (`xSTART`) bzw. die Kanten, die nach außen gehen (`xEND`). Benötigt werden diese Knoten, weil es in der Dot-Syntax nicht möglich ist, Kanten ohne Ursprung bzw. Kanten ohne Ziel anzugeben. Der `CircuitDescriptor` wird verwendet, um die Anzahl der ein- und ausgehenden Kanten auszulesen. So kann die benötigte Anzahl der Pins beider Knoten generiert werden.

```
dot_outer_nodes :: CircuitDescriptor -> String
dot_outer_nodes g
   = concat $ map break
   [ ""
   , "xSTART ["
   , "    " ++ dot_outer_label "op" (sinks.nodeDesc $ g)
   , "    " ++ "shape = \"record\""
   , "]"
   , ""
   , "xEND ["
   , "    " ++ dot_outer_label "ip" (sources.nodeDesc $ g)
   , "    " ++ "shape = \"record\""
```

```
13    , "]"
14    ]
```

`dot_components` erzeugt für jeden Baustein, der im `CircuitDescriptor` angegeben ist, eine Komponente. Zunächst wird das Label `nodeId` mit der zugehörigen Komponentennummer ausgegeben. Dem folgt die Beschreibung des Knotens, die von der Funktion `dot_label` aus den Pins, den Namen und der Komponentennummer erzeugt wird.

```
1  dot_components :: CircuitDescriptor -> String
2  dot_components  g
3    = concat $ nub $ map f (nodes g)
4    where f g'
5      =   concat
6        $ map break
7          [ ""
8          ,    "nodeId"
9            ++ show (nodeId.nodeDesc $ g')
10           ++ " ["
11         ,    "    "
12           ++ dot_label
13                (sinks.nodeDesc $ g')
14                (map   (\x -> if x == '>'
15                              then '-'
16                              else x)
17                     $ label.nodeDesc $ g')
18                (nodeId.nodeDesc  $ g')
19                (sources.nodeDesc $ g')
20         ,    "    "
21           ++ "shape = \"record\""
22         , "]"
23         ]
```

Die Funktion `dot_outer_label` erzeugt den beschreibenden String, der zwei Knoten darstellt, die „außen“ symbolisieren sollen. Diese Funktion unterscheidet sich von der Funktion `dot_label` insofern, als dass sie jeweils nur eine Anzahl an Pins übergeben bekommt. Die Bezeichnung ist identisch mit der Bezeichnung in `dot_label`.

```
1  dot_outer_label :: String -> Pins -> String
2  dot_outer_label s ps
3       =  "label = \" {{ | "
```

```
      ++ (concat $ map (f s) ps)
      ++ "}}\""
      where f :: String -> Int -> String
            f s x = "<" ++ s ++ show x ++ "> (" ++ show x ++ ") | "
```

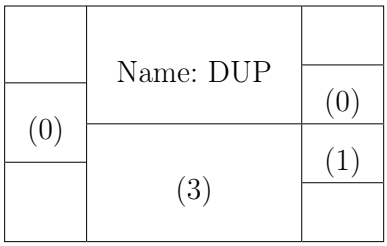

Abbildung 6.9.: Einfacher Dot-Knoten

Durch die Funktion `dot_label` wird der String erzeugt, der das Aussehen des Dot-Knotens bestimmt. Diese Funktion bekommt die Anzahl der eingehenden Pins, die Bezeichnung der Komponente, deren Komponentennummer sowie die Anzahl der ausgehenden Pins übergeben. Daraus wird der Knoten so aufgebaut, wie in Abbildung 6.9 dargestellt.

```
dot_label :: Pins -> String -> CompID -> Pins -> String
dot_label ips nme cid ops
      =  "label = \" {{ | "
      ++ (concat $ map (f "ip") ips)
      ++ "} | { Name: " ++ nme ++ " | (" ++ show cid ++ ")} | { | "
      ++ (concat $ map (f "op") ops)
      ++ "}}\""
      where f :: String -> Int -> String
            f s x = "<" ++ s ++ show x ++ "> (" ++ show x ++ ") | "
```

Durch `dot_connections` wird für jede Kante, die im `CircuitDescriptor` abgelegt ist, die zugehörige `Show`-Funktion aufgerufen. Diese `Show`-Funktion ist in `showEdge` beschrieben.

```
dot_connections :: CircuitDescriptor -> String
dot_connections g
      = concat $ map (\x -> showEdge x ++ "\n") (edges g)
```

6.10.2. Von der Graphstruktur zur `.vhdl`-Notation

Nachdem Abschnitt 6.10.1 verdeutlicht hat, wie aus dem `CircuitDescriptor` eine Darstellung im Dot-Format erzeugt werden kann, zeigt dieser Abschnitt, dass auch ein Export nach VHDL möglich ist. Es wird dargestellt, wie ein kombinatorisches Bauteil aus einem `CircuitDescriptor`-VHDL erzeugt wird. In kombinatorischen Bauteilen findet sich die Eigenschaft einer Funktion im funktionalen Sinne wieder. Die Ausgabe des Bauteiles ist von der Eingabe in das Bauteil abhängig [26, S. 122].

Die Beschreibung der VHDL-Extraktion lehnt sich an den Aufbau einer VHDL-Datei an. Zunächst wird gezeigt, wie der Kopf der Datei entsteht. Dem folgt die Beschreibung der „Entitäten“, auf die wiederum die Beschreibung der Architektur folgt. Dann wird dargestellt, wie die Signale und die einzelnen Entitäten miteinander verbunden werden. Zuletzt folgt die Beschreibung der „Portmaps“. Die Beschreibung ist mit Quelltextauszügen untermalt. Der erzeugte VHDL-Quelltext baut auf der Annahme auf, dass Bauteile vorhanden sind und verwendet werden können. Es wird davon ausgegangen, dass die IEEE.STD_LOGIC_1164.ALL-Bibliothek vorhanden ist. Dies stellt keine prinzipielle Beschränkung auf die genannte Bibliothek dar. Andere Bibliotheken können ebenso genutzt werden. Voraussetzung wäre ein Anpassen der Kopfbeschreibung.

Die Extraktion des `CircuitDescriptor`s erfolgt hier jeweils in einen String. Besser wäre es, mit der Ausgabe einen weiteren abstrakten Datentypen zu erzeugen, der die Eigenschaften von VHDL fassen kann. Dafür könnte ein Datentyp wie in der von Baaij [4] vorgestellten VHDL-Bibliothek verwendet werden. Leider ist diese Bibliothek in einem zu frühen Stadium [1]. Die Möglichkeit, von Strings bis zu abstrakten VHDL-Datentypen wählen zu können, veranschaulicht den Mix aus tiefer und seichter Einbettung erneut.

Die gesamte VHDL-Extraktion verbirgt sich hinter der Funktion `showCircuit`. Diese Funktion erzeugt die `.vhdl`-Datei, indem eine Liste der einzelnen Strings erzeugt wird. Diese Liste wird mit Zeilenumbrüchen versehen und zu einem neuen String verbunden.

```
1 showCircuit :: CircuitDescriptor -> String
2 showCircuit g
3   = concat $ map break
```

[1] Version 0.1.x, seit 4 Jahren keine Änderungen

```
  [ ""
  , vhdl_header
  , vhdl_entity       g namedComps
  , vhdl_architecture g
  , vhdl_components   g namedComps
  , vhdl_signals      g namedEdges
  , vhdl_portmaps     g namedComps namedEdges
  ]
  where
    namedEdges = generateNamedEdges g
    namedComps = generateNamedComps g
```

Kopf

Hinter dem Namen `vhdl_header` verbirgt sich ein gewünschter Kopf einer `.vhdl`-Datei. An dieser Stelle werden die verwendeten Bibliotheken eingebunden.

Entitäten

Entitäten beschreiben das Bauteil bezüglich der Ein- und Ausgabepins. Hier wird das Interface des Bauteiles definiert. Die Pins erhalten Namen, auf die später in der Architekturbeschreibung und in den Portmaps zurückgeriffen wird. Aufgrund der Festlegung von Arrows auf binäre Signale zur Datenübertragung werden für das Interface nur Pins verwendet, die den gleichen Datentyp besitzen. Dieser verbirgt sich hinter der Funktion `pinType`, die in diesem Fall `std_logic` zurückgibt. Hier kann die Implementation weitere Signaltypen einführen. Wie von Makinwa [37] vorgeschlagen und im Abschnitt 8 erwähnt, ist es sinnvoll, sich früh auf einen digitalen Typ festzulegen.

```
vhdl_entity :: CircuitDescriptor -> [NamedComp] -> String
vhdl_entity g namedComps
     = concat $ map break
     [ "ENTITY " ++ (label.nodeDesc) g ++ " IS"
     , "PORT ("
     , (sepBy "\n" $ map (\x -> x ++ " : IN  " ++ pinType x ++ ";") $ snks)
     , (sepBy "\n" $ map (\x -> x ++ " : OUT " ++ pinType x ++ ";") $ srcs)
     , ");"
     , "END " ++ (label.nodeDesc) g ++ ";"
```

```
10      ]
11      where snks = getInPinNames  namedComps (nodeId.nodeDesc $ g)
12            srcs = getOutPinNames namedComps (nodeId.nodeDesc $ g)
```

Dem Listing kann entnommen werden, dass zusätzlich zum `CircuitDescriptor` eine Liste von Komponenten mit deren Namen übergeben wird. Eine `NamedComp` wird aus einem Tupel gebildet. Das erste Element dieses Tupels ist die Komponenten-ID. Das zweite Element des Tupels ist ein weiteres Tupel, das aus `[NamedPin]`, ein- und ausgehenden Pins besteht. Ein `NamedPin` ist die Kombination einer `PinID` mit einem String.

```
1  type NamedComp = (CompID, (InNames, OutNames))
2  type InNames   = [NamedPin]
3  type OutNames  = [NamedPin]
4  type NamedPin  = (PinID, String)
```

Architektur

Die Architektur ist mit zwei Funktionen zu beschreiben. `vhdl_architektur` erzeugt den Kopf der Architekturbeschreibung; die Funktion `vhdl_components` generiert die einzelnen Rümpfe.

Von der zweiten Funktion wird das grundsätzliche Interface der Komponenten erzeugt, das bei der Definition verwendet wird. Es handelt sich um die Komponenten des obersten Graphen. Nachdem die Funktion `flatten` aus Abschnitt 6.8 auf den `CircuitDescriptor` angewandt wurde, existiert nur noch diese Ebene in dem Graphen. An dieser Stelle wird zusätzlich zum `CircuitDescriptor` die Liste `NamedComp`s übergeben.

```
1  vhdl_components :: CircuitDescriptor -> [NamedComp] -> String
2  vhdl_components g namedComps
3    = concat $ nub $ map f (nodes g)
4    where f g'
5      = concat $ map break
6        [ ""
7        , "COMPONENT " ++ (label.nodeDesc $ g') ++ "Comp"
8        , "PORT ("
9        , (sepBy "\n" $ map (\x -> x ++ " : IN  " ++ pinType x ++ ";") $ snks)
```

```
10        , (sepBy "\n" $ map (\x -> x ++ " : OUT " ++ pinType x ++ " ") $ srcs)
11        , ");"
12        , "END COMPONENT " ++ (label.nodeDesc $ g') ++ "Comp;"
13        ]
14        where
15          snks = getInPinNames  namedComps (nodeId.nodeDesc $ g')
16          srcs = getOutPinNames namedComps (nodeId.nodeDesc $ g')
```

Dem Listing lässt sich entnehmen, dass auch hier die Liste der Strings erzeugt wird, die später in die `.vhdl`-Datei einfließen.

Signale

Signale geben die internen Drähte der Komponte wieder. Diese können aus dem `CircuitDescriptor` und einer weiteren temporären Datenstruktur generiert werden. Es handelt sich bei der zweiten Struktur um eine Liste, die aus dem Tupel der Andockpunkte und einem String besteht. Es sind die Kanten, die mit ihrem Namen kombiniert werden.

```
1  vhdl_signals :: CircuitDescriptor -> [([Anchor], String)] -> String
2  vhdl_signals _ [] = ""
3  vhdl_signals g namedEdges
4       = "SIGNAL " ++ sepBy ", " signals ++ ": " ++ pinType_ ++ ";"
5       where signals = map snd namedEdges
```

Portmaps

Die Erstellung von Portmaps stellt den letzten Schritt zur Generierung einer `.vhdl`-Datei dar. Es handelt sich um die Verbindung der Pins mit den internen Drähten. Der Funktion werden die Parameter mit übergeben. `vhdl_portmaps` erhält einen `CircuitDescriptor` zusammen mit der Liste der benannten Komponenten und der Liste der benannten Kanten. Die Funktion `vhdl_portmaps` ist durch mehrere Aufrufe der Funktion `vhdl_portmap` definiert. Der Singular drückt aus, dass diese Funktion ausschließlich eine Portmap erzeugt. Der Name der aufrufenden Funktion steht im Plural, da potentiell an dieser Stelle mehrere Portmaps entstehen können. Alle Portmaps werden von demselben `BEGIN` - `END`-Block umschlossen.

```
vhdl_portmaps :: CircuitDescriptor -> [NamedComp] -> [([Anchor], String)]
              -> String
vhdl_portmaps g namedComps namedEdges
     = concat $ map break
     [ "BEGIN"
     , concat $ map (vhdl_portmap g namedComps namedEdges) $ nodes g
     , "END;"
     ]
```

Die Portmapfunktion erhält dieselben Parameter wie die aufrufende Funktion. Zusätzlich wird der `CircuitDescriptor` die Komponente, für welche die Portmap erzeugt wird, übergeben. Innerhalb der Funktion werden diese `CircuitDescriptor`en als `superG` sowie als `g` bezeichnet.

```
vhdl_portmap :: CircuitDescriptor -> [NamedComp] -> [([Anchor], String)]
             -> CircuitDescriptor -> String
vhdl_portmap superG namedComps namedEdges' g
  = concat $ map break
    [   (label.nodeDesc $ g)
     ++ "Inst"
     ++ (show.nodeId.nodeDesc $ g)
     ++ ": "
     ++ (label.nodeDesc $ g)
     ++ "Comp"
    ,    "PORT MAP ("
      ++ (sepBy ", " $ filter ((>0).length) [ incoming
                                            , signaling
                                            , outgoing])
      ++ ");"
    ]
  where
    relevantEdges = filter (isFromOrToComp $ nodeId.nodeDesc $ g)
                                            $ edges superG
    edge2inside   = filter (isFromOuter)    $ relevantEdges
    edge2outside  = filter (isToOuter)      $ relevantEdges
    pin2signal    = relevantEdges \\ (edge2outside ++ edge2inside)
    incoming      = sepBy ", " $ map (genPortMap
                                        namedComps namedEdges'
                                        (nodeId.nodeDesc $ g))
                                 $ edge2inside
    outgoing      = sepBy ", " $ map (genPortMap
```

```
                                        namedComps
                                        namedEdges'
                                        (nodeId.nodeDesc $ g))
                                $ edge2outside
    signaling     = sepBy ", " $ map (genPortMap
                                        namedComps
                                        namedEdges'
                                        (nodeId.nodeDesc $ g))
                                $ pin2signal
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY DUP>>>-ID-&&&CONST_4>>>SHIFTL&&&-ID->>>ADD IS
    PORT (
    inc0 : IN  std_logic;
    inc1 : IN  std_logic;
    out0 : OUT std_logic
);
END DUP>>>-ID-&&&CONST_4>>>SHIFTL&&&-ID->>>ADD;

ARCHITECTURE DUP>>>-ID-&&&CONST_4>>>SHIFTL&&&-ID->>>ADDStruct OF DUP>>>-ID-\
&&&CONST_4>>>SHIFTL&&&-ID->>>ADD IS

    COMPONENT DUP>>>-ID-&&&CONST_4>>>SHIFTL&&&-ID-Comp
        PORT (
        e0 : IN  std_logic;
        e1 : IN  std_logic;
        a0 : OUT std_logic
        a1 : OUT std_logic
    );
    END COMPONENT DUP>>>-ID-&&&CONST_4>>>SHIFTL&&&-ID-Comp;

    COMPONENT ADDComp
        PORT (
        e0 : IN  std_logic;
        e1 : IN  std_logic;
        a0 : OUT std_logic
    );
    END COMPONENT ADDComp;

    SIGNAL i0, i1: std_logic;
BEGIN
```

```
34      DUP>>>-ID-&&&CONST_4>>>SHIFTL&&&-ID-Inst1: DUP>>>-ID-&&&CONST_4>>>SHIFTL\
35  &&&-ID-Comp
36      PORT MAP (e0 => inc0, e1 => inc1);
37      ADDInst10: ADDComp
38      PORT MAP (e0 => i0, e1 => i1, a0 => out0);
39
40      END;
```

7. Zwei Fallbeispiele der Arrowmodellierung

In diesem Kapitel werden zwei konkrete Anwendungsfälle mithilfe der Arrow-Methode modelliert. Dazu wird die Anwendung eines CRC-Arrows modelliert. Als zweites Beispiel wird ein Algorithmus aus dem kryptographischen Umfeld gewählt.

7.1. CRC-Arrow

Es wird dargestellt, wie ein CRC-Algorithmus mit Arrows umgesetzt wird. Eine zyklische Redundanzprüfung (CRC [1]) ist eine Methode zur Prüfsummenberechnung von Daten. Die Berechnung des CRC-Wertes basiert auf der Polynomdivision. Der Rest dieser Division stellt den CRC-Wert dar. Der vorgestellte Ansatz berechnet den CRC mithilfe eines linear rückgekoppelten Schieberegisters, kurz LSFR [2].

Ein solches LSFR wird mithilfe eines Polynoms (z. B. $x^8 + x^4 + x^2 + 1$) beschrieben. Der höchste Exponent gibt die Anzahl der benötigten Register wieder. Die weiteren Exponenten zeigen an, nach welchen Registern das Signal rückgeführt wird. Im Beispielpolynom sind dies 8 Register, wobei nach dem vierten und dem zweiten Register das Zwischenergebnisssignal mit dem zurückgeführten Signal `xor` verknüpft wird. Weiter wird der Ausgang des LSFR mit dem Eingangsssignal `xor` verknüpft. Das oben gewählte Beispiel ist in Abbildung 7.1 dargestellt.

Das Shiftregister wird aus einer Liste sowie den Feedbackpositionen modelliert. Die Liste wird mithilfe von Tupeln simuliert, wie dies im Abschnitt 5.6 zu Tupellisten vorgestellt wurde. Der Feedbackwert wird durch einen Arrow (`aFloatR`) innerhalb der Liste weitergeschoben, um an der richtigen Stelle mit dem dort stehenden Wert

[1] cyclic redundancy check
[2] linear feedback shift register

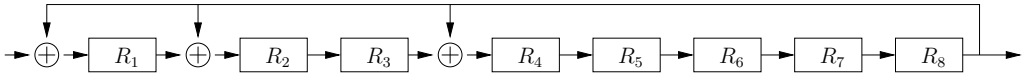

Abbildung 7.1.: Symbolisation eines galoisartigen LFSR

`xor`-verknüpft zu werden. Unterschiedliche Feedbackpositionen werden durch die Arrows `aMoveBy`$_n$ verarbeitet. Diese Arrows „schieben“ den Feedbackwert an die entsprechende Position innerhalb der Tupelliste.

```
1  aMoveBy5
2    :: Netlist a (x, (b, (c, (d, (e, (f, rst))))))
3                 (b, (c, (d, (e, (f, (x, rst))))))
4  aMoveBy5
5    =   aFloatR
6    >:> aFloatR
7    >:> aFloatR
8    >:> aFloatR
9    >:> aFlip
10   where
11     (>:>) aA aB
12       = aA >>> (second aB)
```

Resultat des `aMoveBy5`-Arrows ist eine Tupelliste, in der der erste Wert um 5 Stellen weiter in die Tupelliste geschoben wird.

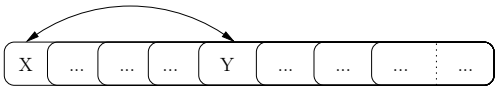

Abbildung 7.2.: Arrow verschiebt Wert um 5 Positionen in Tupelliste

Das Prinzip eines CRC-Bauteiles wird in einem CRC-Arrow festgehalten. Aufgrund des strikten Typsystems der Wirtssprache Haskell ist es notwendig, die Schachtelungstiefe und damit die Bitgenauigkeit des CRC-Bauteiles anzugeben. Als Beispiel wird eine 8-Bit-Variante des CRC-Bauteiles dargestellt:

```
1  aCRC8 polynom polySkip start reminder padding
2    =   (padding &&& aId)  >>> aMoveBy8
3
4    >>> (start &&& reminder)
5    >>> first polynom
```

```
6
7     >>> step
8     >>> step
9     >>> step
10    >>> step
11    >>> step
12    >>> step
13
14    >>> aFlip
15    >>> polySkip
16    >>> polynom
17
18    where step
19            =   aAssocLeft
20            >>> first
21                (   aFlip
22                >>> polySkip
23                >>> polynom
24                )
```

Mithilfe dieser abstrakten Darstellung lässt sich jeder CRC-Baustein, der 8 Bit breite Eingabedaten erwartet, formulieren. Zu den CRC-Eingabedaten werden die Füllbits (`padding`) getupelt. Diese werden mittels `aMoveBy8` an das rechte Ende der Tupelliste geschoben. Dies ist im folgenden Beispiel illustriert:

$$((p_0, p_1), (b_0, (b_1, (b_2, b_3)))) \rightarrow (b_0, (b_1, (b_2, (b_3, (p_0, p_1)))))$$

Danach stehen in der Tupelliste des `aCRC8`-Arrows die 8 Bits, die zur Berechnung benötigt werden, gefolgt von den 4-padding Bits.

Der Arrow `(start &&& reminder)` teilt die Tupelliste in zwei Teile. Im vorderen Teil stehen 5 Bits, im hinteren Teil bleiben die restlichen 7 Bits. Mithilfe des `first polynom`-Arrows wird der Arrow, der die Berechnung des Polynoms enthält, auf die ersten 5 Bits angewandt.

Diesem Arrow folgen 6 weitere Schritte, in denen dieselbe Berechnung durchgeführt wird. Danach folgt ein Schritt, der sich von den 6 vorhergehenden darin unterscheidet, dass nicht alle Teilarrows die letzten Schritte ausführen werden.

Ein Schritt bedeutet mithilfe von `aAssocLeft` den 4 Ergebnissbits ein weiteres Bit

hinzuzufügen.

$$((b_0,(b_1,(b_2,b_3))),(b_{next},(b_{rest})) \rightarrow (((b_0,(b_1,(b_2,b_3))),b_{next}),(b_{rest}))$$

Dieses neue Bit wird durch den `aFlip`-Arrow an die vordere Position geholt

$$(((b_0,(b_1,(b_2,b_3))),b_{next}),(b_{rest})) \rightarrow ((b_{next},(b_0,(b_1,(b_2,b_3)))),(b_{rest}))$$

und daraufhin durch `polySkip` an das Ende der vorderen Tupelliste geschoben.

$$((b_{next},(b_0,(b_1,(b_2,b_3)))),(b_{rest})) \rightarrow ((b_0,(b_1,(b_2,(b_3,b_{next})))),(b_{rest}))$$

Für diese Berechnung wird ein Arrow, der das Polynom symbolisiert, verwendet. Polynome sind immer auf ein spezielles Problem abgestimmt. Das vorgestellte Beispiel verwendet einen CRC-Code der ITU-T [3], den CRC-CCITT. Dieser CRC-Code wird für Xmodem-Kommunikation mit 4 Bits verwendet. Das Polynom lautet:

$$\text{CCITT} : x^4 + x^1 + 1$$

Dieses Polynom beschreibt, an welchen Stellen die Werte zurückgeführt und mit dem dortigen Wert **xor**-verknüpft werden. Bit_0 und Bit_1 werden mit Bit_4 **xor**-verknüpft.

Für den CCITT-CRC ist der höchste Exponent 4. Daraus folgt:

- `polySkip` wird durch `aMoveBy4` realisiert,
- es werden 4 `False`-Bits als Füllbits verwendet,
- die Tupelliste wird nach dem 5ten Bit geteilt.

Um das CRC-CCITT-Polynom zu definieren, wird ein Arrow benötigt, der eine Tupelliste der Länge 5 als Eingabe erhält. Die Ausgabe ist eine Tupelliste der Länge 4. Das Bit_4 wird um zwei Positionen nach rechts verschoben und dann mittels `aDistr` mit Bit_1 und Bit_0 vertupelt. `aXor` wird verwendet, um beide Bits mit Bit_4 **xor** zu verknüpfen.

Grafisch lässt sich die Berechnung eines Schrittes des CRC-CCITT-Polynomes wie folgt darstellen:

Diese Darstellung kann direkt in die proc-Notation übersetzt werden:

[3] ehemals: Comité Consultatif International Téléphonique et Télégraphique (CCITT)

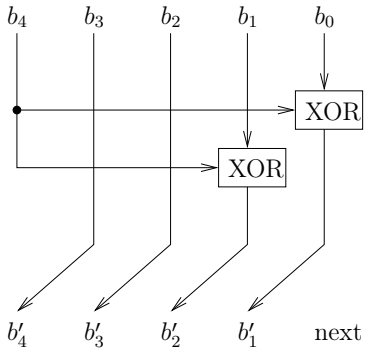

Abbildung 7.3.: CCITT CRC

```
1 crc_polynom_ccit
2   = proc (x4, (x3, (x2, (x1, x0)))) -> do
3       o1 <- aXor -< (x4, x0)
4       o2 <- aXor -< (x4, x1)
5       o3 <- aId  -< (x2)
6       o4 <- aId  -< (x3)
7       returnA    -< (o4, (o3, (o2, o1)))
```

Derselbe Arrow wird argumentfrei wie folgt geschrieben:

```
1 crc_polynom_ccit
2   = mvRight >:> mvRight >:>
3     (   aDistr
4     >>> (aXor *** aXor)
5     )
```

Der CRC-CCITT-Arrow `aCRC_CCITT_8` entsteht durch die Parametrisierung des abstrakten Arrows `aCRC8`:

```
1 aCRC_CCITT_8
2   = aCRC8
3       crc_polynom_ccit
4       aMoveBy4
5       ((second.second.second.second) aFst)
6       (aSnd >>> aSnd >>> aSnd >>> aSnd >>> aSnd)
7       (aConst (False, (False, (False, False))))
```

Es wird deutlich, dass mit der vorgestellten Methode jeder andere CRC-Algorithmus in einen Arrow und damit in einen Hardwarebaustein überführt werden kann. Das Hindley-Milner-Typsystem erfordert lediglich, dass die Anzahl der Bits in der Tupelliste vorher bekannt ist. Letztlich ist sogar diese Einschränkung nicht prinzipiell notwendig, sondern rührt von den Tupellisten her. Im Kapitel 8 wird dieses Problem tiefer betrachtet.

Mithilfe des CRC-Arrow kann eine Prüfsumme für beliebige 8-Bit-Eingaben errechnet werden. Der CCITT-CRC-Arrow wird mit einer 8-Bit-Tupelliste gefüttert und errechnet daraus die CRC-Prüfsumme. Diese wird als 4-Bit-Tupelliste zurückgegeben. Bei einer Eingabe von `0xAA` errechnet der Arrow `0x09` als Prüfsumme. Die Eingabe als Tupelliste erfolgt.

```
> toFunctionModel aCRC_CCITT_8 $
  (True, (False, (True, (False, (True, (False, (True, False)))))))
> (True, (False, (False, True))))
```

Die CRC-Prüfsumme kann als Padding-Bits genutzt werden, um einen Arrow zum Überprüfen zu erzeugen. Dieser sieht so aus:

```
1 aCRC_CCITT_8_check
2   = aCRC8
3       crc_polynom_ccit
4       aMoveBy4
5       ((second.second.second.second) aFst)
6       (aSnd >>> aSnd >>> aSnd >>> aSnd >>> aSnd)
7       (aConst (True, (False, (False, True))))
```

Wird dieser Arrow erneut mit den Daten `0xAA` gefüttert, muss er `0x00` liefern. Dies ist der Fall:

```
> toFunctionModel aCRC_CCITT_8_check $
  (True, (False, (True, (False, (True, (False, (True, False)))))))
> (False, (False, (False, False))))
```

Die vorgestellte Methode ist nicht auf den CCITT-CRC-Code beschränkt.

7.2. TEA-Arrow

Dieser Abschnitt beschreibt als weiteres Beispiel einen kryptographischen Algorithmus. Dazu wurde der TEA (Tiny Encryption Algorithm [4]) gewählt, da er die folgenden Eigenschaften gut miteinander verbindet. Am TEA lassen sich typische Elemente eines Verschlüsselungsalgorithmus, wie „Diffusion“ und „Konfusion“ [55], wiederfinden. „Konfusion“ bedeutet, dass kleine Änderungen am Klartext große Auswirkungen auf den verschlüsselten Text haben. „Diffusion“ steht für die Eigenschaft, dass es schwierig ist, auf den Schlüssel zu schließen, auch wenn große Mengen von Klartext und verschlüsseltem Text vorhanden sind. Beim TEA wird „Diffusion“ erreicht, indem ein Teil des Klartextes pro Feistelrunde mit dem verschleierten zweiten Teil des Klartextes sowie mit nur einem Teil des Schlüssels verknüpft. Die Verknüpfung von Teilen des Klartextes zusammen mit einer Konstanten wirkt als „Konfusion“, denn eine einzelne Änderung am Klartext führt in jeder Runde zu geänderten Ergebnissen. Diese propagieren sich in jeder Runde fort. Als Konstante wird $\frac{2^{32}}{\phi}$ gewählt. ϕ (der goldene Schnitt) stellt eine irrationale Zahl dar.

Mit XOR, ADD und SHIFT werden algebraische Operationen aus verschiedenen Gruppen verwendet. Der TEA besitzt mit einer rundenbasierenden Struktur ein typisches Verhalten von kryptographischen Algorithmen. Aus kryptographischer Sicht besitzt der TEA jedoch Schwächen.[35, 34]

Der TEA basiert auf Feistelrunden. In einer Feistelrunde gibt es zwei Klartextteile (V_0, V_1) und zwei Schlüsselteile. Zwei Feistelrunden bilden einen Zyklus. Pro Zyklus werden alle 4 Schlüsselteile (K_0, K_1, K_2, K_3) angewandt. Pro Runde wird V_1 auf drei Pfaden verändert. Er wird nach links und nach rechts geshiftet sowie mit unterschiedlichen Teilschlüsseln und einer konstanten XOR verknüpft. Die Ergebnisse werden verodert und wiederum mit V_0 verknüpft. Darauf tauschen die Ergebnisse ihre Position. Zwei dieser Feistelrunden sind ein Zyklus. Siehe Abbildung 7.4:

In der proc-Notation lässt sich eine Feistelrunde beinahe bildlich beschreiben: Die Zwischenergebnisse werden mit Namen versehen, sodass sie als Input weiterverwendet werden können. Die `OR`-Verknüpfung muss in zwei Schritte geteilt werden, da der verwendete Arrow (`aOr`) nur zwei Inputs verknüpfen kann.

```
1 aFeistel
```

[4] kleiner Verschlüsselungsalgorithmus

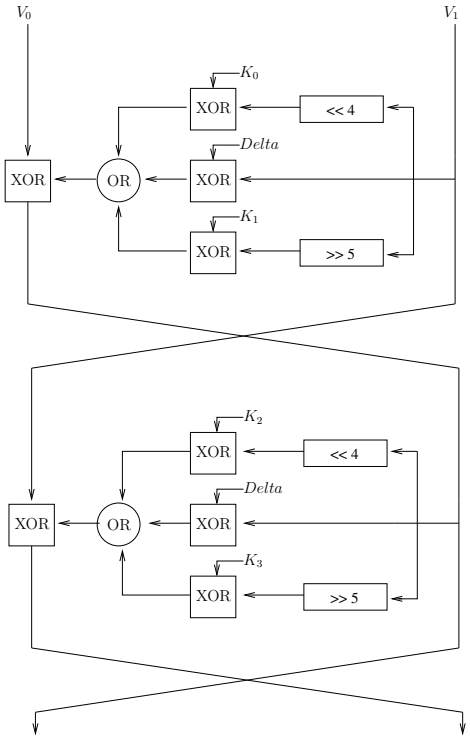

Abbildung 7.4.: 2 Feistelrunden im TEA

```
2   = proc ((v0, v1), (k0, k1)) -> do
3       vsl <- aShl -< (v1, 4)
4       vsr <- aShr -< (v1, 5)
5
6       t1  <- aXor -< (k0, vsl)
7       t2  <- aXor -< (d,  v1)
8       t3  <- aXor -< (k1, vsr)
9
10      t4  <- aOr -< (t1, t2)
11      t5  <- aOr -< (t4, t3)
12
13      v0' <- aId -< (v1)
14      v1' <- aXor -< (v0, t5)
15      returnA     -< (v0', v1')
```

Mithilfe des Arrows einer Feistelrunde kann ein Zyklus, der aus zwei Feistelrunden besteht, definiert werden:

```
1 aCycle
```

```
2   = proc ((v0, v1), ((k0, k1), (k2, k3))) -> do
3       (v0', v1')   <- aFeistel -< ((v0, v1),   (k0, k1))
4       (v0'', v1'') <- aFeistel -< ((v0', v1'), (k2, k3))
5       returnA                  -< (v0'', v1'')
```

Die Definition einer Feistelrunde in argumentfreier Schreibweise benötigt genaues Hinschauen. Es ist etwas Aufwand zu betreiben, um die eingehenden Tupel derart umzuformen, dass die Arrows `aXor` und `aOr` angewandt werden können. Für die argumentfreie Schreibweise ist ein modularer Ansatz von Vorteil. Im ersten Schritt werden die Arrows der drei einzelnen Pfade definiert. Für den ersten Pfad sieht der Arrow wie folgt aus:

```
1  aShiftL4_XorKey
2    =   first
3        (     aDup
4          >>> second (aConst 4)
5          >>> aShiftL
6        )
7    >>> aXor
```

Die zwei weiteren Pfade, `aShiftR5_AddKey` sowie `aAddDelta`, sind im Quelltext im Anhang zu finden. Der die Feistelrunde beschreibende Arrow erscheint kompliziert, ist aber logisch aufgebaut. Der Arrow der Feistelrunde erhält, wie im proc-notierten Arrow gezeigt, zwei Tupel. Das erste Tupel enthält die beiden Klartextteile (V_0, V_1), das zweite Tupel enthält die beiden Schlüsselteile (K_0, K_1).

```
1  aFeistel
2    = (aFst >>> aSnd) &&& (aFst >>> aFst) &&& (first aSnd)
3     >>> second
4         ( second
5           (   aDistr &&& aFst
6           >>> (aShiftL4_AddKey *** aShiftR5_AddKey) *** aAddDelta
7           >>> first aOr
8           >>> aOr
9           )
10        )
11    >>> second aXor
```

In der ersten Quelltextzeile (Zeile 2) wird die übergebene Struktur so geändert, dass eine Tupelliste entsteht. Das erste Element muss V_1 sein, da dies der erste Parameter

der zweiten Feistelrunde sein wird. Dem folgt V_0, da das Ergebnis der drei Pfade mit V_0 XOR verknüpft wird. Danach folgt zusammen mit den Schlüsselteilen V_1.

$$((V_0, V_1), (K_0, K_1)) \rightarrow (V_1, (V_0, [V_1(K_0, K_1)]))$$

Die eckige Klammer deutet den Bereich an, der durch die zwei `second`-Selektoren selektiert wird. Zeile 5 erzeugt die Eingaben der drei Pfade aus der Abbildung 7.4.

$$(V_1, (V_0, [V_1(K_0, K_1)])) \rightarrow (V_1, (V_0, [((V_1, K_0), (V_1, K_1)), V_1]))$$

Daraufhin können in Zeile 6 die drei Pfade ausgeführt werden. Die nächsten zwei Schritte verknüpfen die Zwischenergebnisse jeweils mit einem `aOr`.

$$(V_1, (V_0, [((V_1, K_0), (V_1, K_1)), V_1])) \rightarrow (V_1, (V_0, [(t_1, t_3), t_2]))$$
$$(V_1, (V_0, [(t_1, t_3), t_2])) \rightarrow (V_1, (V_0, [t_5]))$$

Im letzten Schritt wird das Ergebniss im hinteren Tupel mit dem ursprünglichen V_0-Wert `xor` verknüpft. Da dieses Ergniss nun schon an zweiter Stelle steht, muss hier keine Position getauscht werden. Das Ergebniss steht für eine weitere Feistelrunde zur Verfügung.

Ein Zyklus von zwei Feistelrunden lässt sich argumentfrei wie folgt ausdrücken:

```
1 aCycle
2   =    aAssocLeft
3    >>> first aFeistel
4    >>>       aFeistel
```

Nun werden die Vorteile einer argumentfreien Darstellung deutlich. Verglichen mit der Darstellung eines Zyklus in der proc-notation ist die argumentfreie Darstellung deutlich kürzer.

Aus dem Arrow lässt sich beispielsweise der Graph extrahieren, indem die Funktion `synthesize` mit dem entsprechenden Arrow aufgerufen wird. Ein ungeglätteter Graph würde lediglich aus zwei Knoten bestehen, die miteinander verbunden sind. Aus diesem Grund wird die Funktion `flatten` vorher auf den Graphen angewand. `flatten (synthesize aShiftL4_XorKey)` erzeugt den in Abbildung 7.5 dargestellten Graphen.[5]

[5]Es wird der Inhalt der `.dot`-Datei ausgegeben. Der Graph lässt sich mittels `dot <infile> -Tpng`

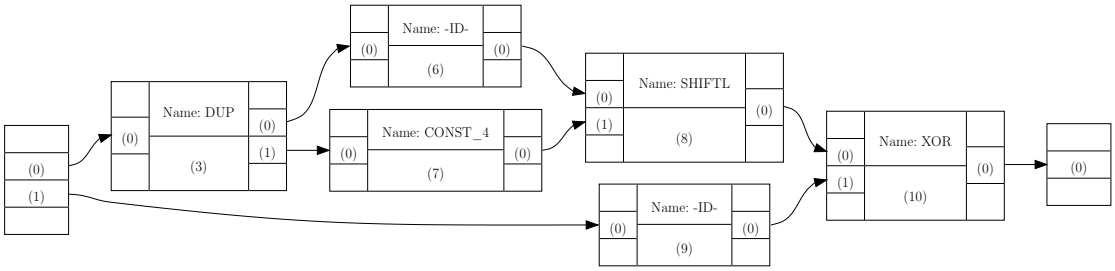

Abbildung 7.5.: Aus aShiftL4_XorKey generierter Graph

`-o <outfile>` in ein Bild konvertieren.

8. Das Tupelproblem

Aufgrund der Tupelstruktur lassen sich Arrows idealerweise auf binäre Schaltungen anwenden. Es wird dargestellt, wie der Problematik, auch n-näre ($n > 2$) Hardware zu beschreiben, begegnet werden kann.

Ein Tupel ermöglicht es, beliebige Typen abzubilden, ist aber in der Anzahl auf exakt zwei beschränkt. Das bedeutet, dass Tupel an der falschen Stelle Flexibilität liefern: im Typ der Elemete, nicht in ihrer Anzahl. Bei der digitalen Hardwarebeschreibung werden vor allem binäre Signale zwischen den Hardwarebausteinen ausgetauscht. Sämtliche Daten, die ausgetauscht werden, werden zunächst binär codiert und dann in dieser Form übertragen. Den Entwicklern von Sensoren wird sogar empfohlen, ihre Signale möglichst früh zu digitalisieren [37].

Arrows sind unnötigerweise auf Tupeln aufgebaut. Daher eigenen sich Arrows hervorragend, um binäre Schaltungen abzubilden. Für andere Schaltungen muss das Interface zur Hardware in einer Weise beschrieben werden, die der Generizität von Arrows widerspricht. Im Folgenden wird erläutert, wie dieser Unzulänglichkeit begenet werden kann.

8.1. Lösung mittels Vektoren

Arrows sind allgemein gehalten: `(Arrow a) => a b c`. Dies ist für Hardwarebeschreibung wenig hilfreich. Hardware überträgt vor allem binäre Signale. Zusätzlich hilft die Tupelstruktur nicht bei der Beschreibung des Hardwareinterfaces. Das Tupel `(a, (b, (c, d)))` beschreibt den gleichen Satz an Drähten wie das Tupel `(a, ((b, c), d))`. Die Klammerung lässt sich nicht auf Hardware übertragen bzw. bringt keinen Mehrwert. Zusätzlich nimmt die Anzahl an möglichen Klammerungen, die alle ein und dasselbe Hardwareinterface beschreiben, exponentiell mit der Anzahl der Drähte zu.

Das Hardwareinterface kann formal deutlich einfacher mithilfe von Vektoren beschrieben werden. Vektoren sind im Gegensatz zu Tupeln homogen im Typparameter, aber heterogen in ihrer Länge. Also müssen alle Elemente eines Vektors, genau wie bei Listen, vom selben Typ sein (Homogenität im Typparameter). Anders als bei Listen hat ein Vektor eine definierte Länge. Mithilfe dieser („Meta“)Informationen lassen sich die Rückgabetypen berechneter Vektoren auf Typebene berechnen. Ein Vektor Vec_n^b bildet eine Datenstruktur ab, die nur Daten des Typs b fassen kann (Homogenität im Typparameter). Außerdem sind Vektoren dynamisch in ihrer Länge. Der Vektor Vec_n^b hat eine definierte Länge n.

Die Verwendung von Vektoren ausschließlich der Länge 2 soll dazu führen, dass ein parametriesierter Arrow exakt dem Tupelarrow entspricht.

Es ist naheliegend, die Arrow-Typklasse mithilfe eines Vektor-Datentyps erneut zu definieren. Diese Typklasse sieht direkt umgesetzt so aus:

$$
\begin{aligned}
arr &:: (\mathrm{Vec}_n^b \to \mathrm{Vec}_m^b) &&\to a\ \mathrm{Vec}_n^b\,\mathrm{Vec}_m^b \\
firsts &:: a\ \mathrm{Vec}_n^b\,\mathrm{Vec}_m^b &&\to a\ \mathrm{Vec}_{(n+k)}^b\,\mathrm{Vec}_{(m+k)}^b \\
seconds &:: a\ \mathrm{Vec}_n^b\,\mathrm{Vec}_m^b &&\to a\ \mathrm{Vec}_{(k+n)}^b\,\mathrm{Vec}_{(k+m)}^b \\
(>>>) &:: a\ \mathrm{Vec}_n^b\,\mathrm{Vec}_m^b \to a\ \mathrm{Vec}_m^b\,\mathrm{Vec}_k^b &&\to a\ \mathrm{Vec}_n^b\,\mathrm{Vec}_k^b \\
(\&\&\&) &:: a\ \mathrm{Vec}_n^b\,\mathrm{Vec}_m^b \to a\ \mathrm{Vec}_n^b\,\mathrm{Vec}_k^b &&\to a\ \mathrm{Vec}_n^b\,\mathrm{Vec}_{(m+k)}^b \\
(***) &:: a\ \mathrm{Vec}_n^b\,\mathrm{Vec}_m^b \to a\ \mathrm{Vec}_k^b\,\mathrm{Vec}_j^b &&\to a\ \mathrm{Vec}_{(n+k)}^b\,\mathrm{Vec}_{(m+j)}^b
\end{aligned}
$$

Die Namen **firsts** und **seconds** erinnern an die ursprünglichen Namen der Arrowtypklasse. Der Plural verdeutlicht, dass im Gegensatz zum ursprünglichen Arrow hier potentiell mehr als zwei Elemente enthalten sein können. Anstelle von **seconds** wäre auch ein Bezeichner wie **lasts** denkbar.

Es werden unterschiedliche Ansätze vorgestellt, wie Vekor-Arrows innerhalb der Sprache Haskell umgesetzt werden können.

8.1.1. Vektoren mit Horn-Klauseln

Die Programmiersprache Haskell kann das Prinzip eines Vektors nicht im Typsystem fassen. Allerdings kennt Haskell Listen, die, wie bereits dargestellt, in vielen Teilen

Vektoren ähnlich sind. Eine Liste ist wie ein Vektor im Typparameter homogen. Im Gegensatz zu den Vektoren besitzt sie allerdings keine definierbare Länge.

Es ist daher naheliegend, eine Liste in der Länge einzuschränken, um so die Eigenschaften des Vektors abzubilden. Dazu wird im Typsystem ein Zähler benötigt. Dieser muss schon im Typsystem bekannt sein. Deswegen werden konkrete Zahlen schon im Typsystem benötigt. Zahlen lassen sich im Hindley-Millner-Typsystem z. B. durch Peano-Zahlen ausdrücken. Für eine Peano-Zahl wird lediglich ein Typ benötigt, der den („Null“)Wert beschreibt. Außerdem wird ein Konstrukt benötigt, das den Nachfolger einer schon existierenden Peano-Zahl abbilden kann.

Eine Peano-Zahl definiert in Haskell:

```haskell
data VNat
  = VZero
  | VSucc (VNat)
```

Der Typkonstruktor `VZero` konstruiert den leeren Typ, der für Null steht. `VZero` hat einen Kind von `*`. Daher kann `VZero` nicht auf andere Typen angwandt werden. Im Gegensatz dazu ist der beschriebene `VSucc`-Typkonstruktor derjenige, der eine nachfolgende Zahl beschreibt. Er ist damit auf die Übergabe einer bestehenden Peano-Zahl angewiesen. Dies spiegelt sich im Kind wider, der für `VSucc` gleich `* -> *` ist. `VSucc VZero` stellt die Beschreibung der konkreten Zahl 1 dar.

Ein solcher Vektor besteht aus den eigentlichen Werten sowie einer Peano-Zahl, die für die Größe des Vektors steht. Die Definition solcher Vektoren muss aus zwei Teilen bestehen. Ein Vektor kann entweder leer sein `T :: Vec VZero a` oder er ist eine Kombination aus einem Element und einem Vektor. Darstellung in Haskell:

```haskell
data Vec n a where
  T    :: Vec VZero a
  (:.) :: (VNat n)
       => a -> (Vec n a) -> (Vec (VSucc n) a)
```

Dieser Ansatz ist an die heterogenen Kollektionen von Kiselyov et al. [36] angelehnt. Die Arrowklasse für diesen Datentyp wird ähnlich wie die ursprüngliche Arrowklasse definiert. `firsts` wendet den übergebenen Arrow auf die ersten n Elemente des Vektors an. `seconds` geht ähnlich vor, mit dem Unterschied, dass der Arrow auf die hinteren Elemente des Vektors angewandt wird. Es ist problematisch, zu entscheiden,

wie viele „erste“ oder „zweite“ Elemente mit `firsts` und `seconds` angesprochen werden. Da Haskells Typsystem statisch ist, muss die Anzahl der Elemente schon vor der eigentlichen Berechnung im Datentyp festgelegt sein. Um die Aufteilung zwischen dem vorderen und dem hinteren Element vornehmen zu können, muss die Position dieser Teilung im Datentyp als Peano-Zahl dargestellt werden. Dazu werden Berechnungen auf der Ebene der Datentypen notwendig. Es ist möglich, Berechnungen im Typsystem durchzuführen. Allerdings ist Haskells Typsystem nicht darauf ausgelegt. Damit werden Berechnungen im Typstem sehr komplex und stehen letztlich dem praktischen Nutzen des Horn-Klausel-Vectors im Weg. Um im Typsystem zu rechnen, ist es notwendig, für jede Rechenoperation eine Typklasse zu erstellen. Exemplarisch ist die Definition der Addition angegeben. Dafür sieht die Typklasse wie folgt aus:

```
1 class (VNat a, VNat b, VNat ab)
2     => VAdd a b ab |  a ab -> b, a b -> ab
3     where vAdd :: a -> b -> ab
```

Die Typklasse `VAdd` besitzt die Abhängigkeiten dreier Zahlen, nämlich `a`, `b` sowie den Ergebniswert `ab`. Es ist zu erkennen, dass für das Ergebnis der Typ angegeben werden muss. Es ist dem Typsystem nicht möglich, den Ergebnistyp der Berechnung selbst zu berechnen. In der Instanzdefinition wird angegeben, dass die Addition von zwei Nullen wieder Null ergibt:

```
1 instance VAdd VZero VZero VZero
```

Im zweiten Fall wird Null zu einer anderen Zahl addiert:

```
1 instance (VNat x) => VAdd VZero x x
```

Zuletzt kann die rekursive Definition der Addition erfolgen. Hier wird in üblicher Peano-Arithmetik ein Nachfolgeelement von der vorderen Zahl „abgeschält“ und zur zweiten Zahl hinzugefügt:

```
1 instance (VNat n1, VNat x) => VAdd (VSucc n1) x (VAdd n1 (VSucc x) ())
```

Zur Anwendung müssen die Typklasse und sämtliche verwendeten Variablen im Kontext aufgeführt werden. Für den `(***)`-Operator sieht der Datentyp wie folgt

aus:

```
(***) :: ( Arrow a
         , VNat n, VNat m, VNat i, VNat j
         , VAdd n i ni, VAdd m j mj)
      => a (Vec n b)  (Vec m b) -> a (Vec i b) (Vec j b)
      -> a (Vec ni b) (Vec mj b)
```

Der mitgeführte Kontext, in dem die Berechnungen gemacht werden, erinnert an das Programmieren mit Prologs Horn-Klauseln. Daher leitet sich der Name dieses Ansatzes ab. Es ist mit dieser Methode möglich, die Trennung des vorderen Teils von dem hinteren Teil zu erreichen. Allerdings wird die Typdefinition schon für einfache Berechnungen sehr komplex.

Für die Berechnung des Datentyps eines **xor** Elementes sieht die Definition des Vektors Horn-Klauseln wie folgt aus:

```
aXor :: (Arrow a)
     => a (Vec (VSucc (VSucc VZero)) Bool) (Vec (VSucc VZero) Bool)

firsts aXor
  :: ( VNat k
     , VAdd (VSucc (VSucc VZero)) k nk, VAdd (VSucc VZero) k mk)
  => a (Vec nk Bool) (Vec mk Bool)
```

Diese Darstellung ist der Hauptnachteil der Horn-Klausel-Vektoren. Es ist immer erforderlich und meist sehr langatmig, die Typdefinition anzugeben. Zusätzlich ist es dem Typsystem nicht möglich, diese Darstellung aus vorhandenem Wissen abzuleiten. Es lassen sich keine Funktionen angeben. Lediglich der Ableitungsmechanismus des Typsystems wird verwendet, um zu zeigen, dass $2+3=5$ ist. Der Programmierer muss immer wieder die Typen selber angeben. Für einfache Arrows mag dies „nur“ störend sein; für komplexe Arrows ist dies eine zusätzliche, enorme Fehlerquelle.

8.1.2. Vektoren mit Typfamilien

Eine weitere Möglichkeit, Vektoren als Datenypen zu realisieren, ist die Verwendung von Typfamilien. Um die Lösung gleich vorwegzunehmen: Typfamilien ermöglichen es, Funktionen auf der Typebene zu definieren. Haskell kennt eine Spracherweite-

rung, die der Sprache Typfamilien hinzufügt [66]. Eine Typfamilie ist ein Weg, den Typkonstruktor eines Datentyps zu berechnen. Dazu wird eine Funktion auf der Ebene des Typsystems eingeführt.

Dies ist gegenüber Horn-Klausel-Vektoren von Vorteil, da bei Horn-Klauseln keinerlei Berechnungen von der Sprache übernommen werden können. Die Länge eines Vektors wird über Peano-Zahlen im Datentyp angegeben. Peano-Zahlen sind widerum eine Anhäufung von Typkonstruktoren, sodass Typfamilien genutzt werden können, um sie auszurechnen. Die Beschreibung im Haskell-Wiki bringt die Erklärung einer Typfamilie mit einem Vergleich auf den Punkt:

> Eine Typfamilie verhält sich zu einem Datentyp, wie die Methode einer Klasse sich zu einer Funktion verhält.

Unter Angabe einer Typfamilie lässt sich die Addition (hier der Typkonstruktor `:+`) auf Peano-Zahlen wie folgt definieren:

```
1 type family   (n :: VNat) :+ (m :: VNat) :: VNat
2 type instance VZero       :+ m           =  m
3 type instance (VSucc n)   :+ m           =  VSucc (n :+ m)
```

Die erste Zeile entspricht der Typdefinition bei normalen Funktionen. Sie erfüllt denselben Zweck. `:+` arbeitet mit zwei Typen: `n` und `m`, die beide den Typ `VNat` besitzen müssen. Die beiden weiteren Zeilen geben die Definition der Typfunktion an und stellen eine direkte Umsetzung der Addition dar.

Mithilfe der Typfamilie kann eine Arrowklasse für Vektoren deutlich einfacher definiert werden, als es mit Horn-Klauseln möglich war. Exemplarisch ist der Typ des `(***)`-Operators unter strikter Anwendung der Typfamilie angegeben:

```
1 (***) :: a (Vec n b)  (Vec m b) -> a (Vec i b) (Vec j b)
2       -> a (Vec (n :+ i) b) (Vec (m :+ j) b)
```

Diese Darstellung ist eine Verbesserung gegenüber der Horn-Klausel-Schreibweise. Bei der Entwicklung muss der Entwickler die Peano-Zahlen nicht mehr direkt angeben, sondern kann sie mithilfe der Typfamilie berechnen lassen. Auch die Infixschreibweise des Typoperators verbessert die Lesbarkeit und damit auch die Verständlichkeit eines solchen Ausdrucks.

Problematisch wird der Typfamilienansatz bei der Aufsplittung eines Vektors in zwei Teile. Im nächsten Schritt wird dieses Problem genauer erläutert. Betrachtet man die folgenden zwei Definitionen, so lässt sich ein Unterschied zwischen der Definiton des `firsts`-Operators und der des `seconds`-Operators feststellen.

```
1 firsts  :: a (Vec n b)         (Vec m b)
2         -> a (Vec (n :+ k) b)  (Vec (m :+ k) b)
3
4 seconds :: ((k :+ m) ~ (m :+ k))
5         => a (Vec n b)         (Vec m b)
6         -> a (Vec (k :+ n) b)  (Vec (k :+ m) b)
```

Die Definition von `firsts` sieht einen Arrow als Parameter vor. Dieser bildet einen Vektor der Länge n auf einem Vektor der Länge m ab. `firsts` soll erreichen, dass ein Arrow nur auf einen Teilvektor angewandt wird. Deswegen erwartet das Resultat von `firsts` einen Vektor, der genau k Elemente länger ist ($n + k \to m + k$).

Dieses Verhalten wird auch vom `seconds`-Operator erwartet. Es ist vom Typsystem nicht leistbar, aufgrund der Art der Definition der Additionsfunktion innerhalb der Typfamilie. Dort ist festgelegt, dass die Addition vom ersten Parameter ein `VSucc` abtrennt und rekursiv dem Endergebnis hinzufügt. Damit wird der erste Parameter irgendwann zu `VZero` und die Rekursion endet. Im `seconds`-Fall wird der zweite Paramter vor dem ersten Parameter auf ein `VZero` treffen. Diese Situation wird von der Typfamilie nicht abgedeckt. Überlappende Definitionen sind in Typfamilien nicht möglich, weshalb dieser Fall in einer Typklasse nicht abgedeckt werden kann.

Als Workaround ist „nur“ darauf zu achten, dass der erste Parameter der Addition der kleinere ist. Ist dies nicht der Fall, kann man sich der Kommutativität der Addition bedienen und beide Parameter tauschen. Das Typsystem kann dies nicht automatisch ableiten. Aus diesem Grund muss der Entwickler dem Typsystem einen „Tipp“ geben. Im Falle von `seconds` wird dieser Hinweis mit `((k :+ m)  (m :+ k))` in der Kontextdefinition gegeben. Der `~`-Operator Haskells ist der Typgleichheits-Operator.

Auch an dieser Stelle lässt sich feststellen, dass es dem Typsystem nicht möglich ist, einfache Berechnungen oder Umformungen abzuleiten. Und wieder gilt, wie bei Horn-Klauseln: Diese Hinweise können für einfache Ableitungen vom Programmierer angegeben werden, bei komplexen Arrows ist diese Praxis ein Garant für Fehler.

8.1.3. Vektoren benötigen abhängige Typisierung

Die beiden vorhergehenden Abschnitte beschreiben Lösungen, um dieselbe Schwäche Haskells auszugleichen. Das Problem besteht darin, dass das Typsystem Haskells nicht ausdrucksstark genug ist, um Vektoren fassen zu können. Anders gesagt, mit Haskells Typsystem lassen sich Vektoren angeben, das Arbeiten mit selben ist jedoch kompliziert.

Eine Programmiersprache, in der die Datentypen von Berechnungen abhängen, wird „abhängig typisiert“ [1] genannt. Eine solche Programmiersprache ermöglicht es, den Typ eines Ausdruckes der Sprache im Typsystem berechnen zu lassen. Es existieren mehrere Programmiersprachen, die ein Typsystem besitzen, das die Angabe von „abhängigen Typen“ unterstützt. Eine dieser Programmiersprachen - Agda [38][43] - hat eine starke Verbindung zu Haskell. Agda ist auf der Syntax Haskells aufgebaut. Es existiert ein Compiler-Backend, das nach Haskell kompilieren kann.

Der Lösungsansatz, eine Sprache mit einem abhängigen Typsystem vorzustellen, soll illustrieren, dass es kein grundsätzliches Problem ist, hardwarebeschreibende Arrows mit funktionalen Mitteln zu definieren. Die Mächtigkeit des Typsystems entscheidet darüber, wie einfach diese Form der Arrows ausgedrückt werden kann.

Die Spezifikation der Arrowtypklasse in Agda:

```
record VArrow (_=>_ : Set -> Set -> Set) : Set1 where
  field
    arr
     : forall {B n m}
     -> (Vec B n -> Vec B m)
     -> (Vec B n) => (Vec B m)
    firsts
     : forall {B n m k}
     -> Vec B n => Vec B m
     -> Vec B (n + k) => Vec B (m + k)
    seconds
     : forall {B n m k}
     -> Vec B n => Vec B m
     -> Vec B (n + k) => Vec B (m + k)
    _>>>_
     : forall {B n m k}
```

[1] dependently typed

```
17      -> Vec B n => Vec B m
18      -> Vec B m => Vec B k
19      -> Vec B n => Vec B k
20     _***_
21      : forall {B n m k j}
22      -> Vec B n => Vec B m
23      -> Vec B k => Vec B j
24      -> Vec B (n + k) => Vec B (m + j)
25     _&&&_
26      : forall {B n m k}
27      -> Vec B n => Vec B m
28      -> Vec B n => Vec B k
29      -> Vec B n => Vec B (m + k)
```

Dieser Agda-Record definiert einen Funktionstypen `_=>_`. Das `forall`-Statement gibt einen Kontext, ähnlich wie der Kontext eine Haskellfunktion, an. Bei genauem Betrachten der zwei Funktionen `firsts` und `seconds` fällt die Typgleichheit auf. Auch hier fehlt dem Compiler die notwendige Information, die Gleichheit von $n + k$ und $k + n$ abzuleiten. Anders als in Haskell ist es in Agda möglich, diesen Beweis zu führen. Dieser kann dann vom Compiler verwendet werden. Ist dies erstmals bewiesen, so kann Agda dies in Zukunft selber ableiten.

9. Fazit

9.1. Zusammenfassung

In der Arbeit wird ausgeführt, wie eine ausführbare Netzlistenbeschreibungen erzeugt werden kann. Als notwendige Vorbedingung wird erläutert, wie funktionale Datenstrukturen, im Speziellen Arrows, dafür eingesetzt werden.

Kapitel 1: Die Ziele und Hintergründe der wissenschaftlichen Auseinandersetzung werden aufgezeigt. Sie knüpfen an bereits bestehende Ansätze der Hardwarebeschreibung in der funktionalen Programmierung an, erweitern diese jedoch um ein wesentliches Element, die Ausführbarkeit. Es werden relevante Arbeiten anderer Forscherteams und ihre Besonderheiten dargestellt.

Kapitel 2: Im zweiten Kapitel wird ein kurzer Exkurs in funktionale Programmiertechniken unternommen. Es werden jene Themen beschrieben, die in den späteren Implementierungen in Kapitel 5 eingesetzt werden. Es werden die Schwerpunkte auf für die vorliegende Arbeit relevante funktionale Techniken gelegt.

Kapitel 3: Im dritten Kapitel werden ausgewählte Betrachtungsweisen von Hardware dargestellt. Dazu werden Diagrammtypen erläutert, mit denen Hardwaresysteme entworfen oder visualisiert werden. Es werden Gemeinsamkeiten in Diagrammen aufgezeigt sowie die Möglichkeit, diese Diagramme auf einfache Graphstrukturen herunterzubrechen. Die Darstellung in Form von Graphen verdeutlicht, dass Arrows die richtige funktionale Datenstruktur darstellen, um ausführbare Hardwarestrukturen zu erzeugen.

Kapitel 4 gibt eine Einführung in die Arrowdatenstruktur. Orientierung ist hier eine historische Betrachtung von Arrows, die stark an die der Monaden geknüpft ist. Es wird aufgezeigt, inwiefern Arrows eine Kombination aus tiefer und seichter Einbettung darstellen.

Kapitel 5: Die konkrete Umsetzung von Netzlistenarrows wird anhand von Quelltext erläutert. Es werden die Algorithmen zur Grapherzeugung vorgestellt. Dazu werden die Verdrahtungsschemata erläutert und deren Umsetzung mit der Arrowdatenstruktur wird dargelegt. Weiter werden Optimierungsalgorithmen dargestellt, die aus einer sehr verschachtelten Struktur eine minimale Struktur errechnen. Das Kapitel führt aus, wie die aufgebaute Graphstruktur in unterschiedliche Darstellungsformen überführt wird. Es werden beispielhaft ein Export in die Dot-Beschreibungssprache sowie ein zweiter Export in die VHDL-Form beschrieben.

Kapitel 6: Es werden zwei unterschiedliche Anwendungsfälle analysiert: Es wird ein CRC-Algorithmus mithilfe von Shift-Registern in der Arrowdatenstruktur dargestellt. Zudem wird ein Kryptoalgorithmus erläutert. Dieser wird mit der Arrowmethode modelliert. Es wird an den Fallbeispielen aufgezeigt, wie aus Arrows Graphen exportiert werden. An einem Beispiel wird exemplarisch verdeutlicht, wie VHDL-Quelltext erzeugt werden kann. Zuletzt wird die Ausführung der entworfenen Netzlistenstruktur dargestellt.

In Kapitel 7 werden prinzipieller Probleme dargestellt, die in der Entwicklung entstehen können, sowie Lösungen dargelegt. Es wird vor allem erläutert, warum eine Lösung auf abhängige Typsysteme zurückgreifen soll. Es werden Lösungsansätze geboten, die in der verwendeten Programmiersprache eingesetzt werden können.

9.1.1. Evaluation

Ziel der wissenschaftlichen Auseinandersetzung war der Entwurf ausführbarer Netzlistenbeschreibungen. Das Ziel wird mit der im Kapitel 6 dargelegten Implementierung erreicht. Der Weg zur Erreichung des Ziels wird durch die Paper [11, 27, 12, 13, 14] und durch die vorliegende Arbeit wissenschaftlich dokumentiert. Die während der Arbeit entstandene Software ist in die Forschungsergebnisse des MERSES-Projektes [8] eingeflossen und kann dort der Downloadsektion [1] entnommen werden. Des Weiteren ist die Software über den Haskell-Paketserver „hackage“ [9] auf dem branchenüblichen Weg erhältlich. Mit dem Befehl `cabal install arrowVHDL` kann sie heruntergeladen und installiert werden. Der Quelltext ist im Github-Repository [10] zu finden. Der Software liegt ein Usermanual bei, das auch im AnhangA.1 zu finden ist.

[1] `http://www.merses.de/downloads.html`

Die Netzlistenbeschreibung ist beschreibender Natur, ohne dabei einschränkend in der Ausführbarkeit zu sein. Dieses Ziel wird mit den Ausführungen des Kapitels 5 erreicht. Kapitel 6 verdeutlicht dies anhand konkreter Beispiele.

Der Erhalt der Seiteneffektfreiheit ist als weiteres Ziel erreicht, da keine Funktion angewandt wird, die zusätzliche Seiteneffekte enthält. In Kapitel 3 wird erläutert, dass in der gewählten Programmiersprache – Haskell – Funktionen nur dann mit Seiteneffekten behaftet sein können, wenn sich dies im Datentyp der Funktion widerspiegelt. Dem Quelltext kann entnommen werden, dass keine Funktion Seiteneffekte verwendet, um die Netzlistenstruktur zu erzeugen, sie zu bearbeiten oder aus ihr weitere Strukturen zu erzeugen. Das Simulieren von Netzlisten ist möglich, ohne dass auf seiteneffektbehaftete Funktionen zurückgegriffen werden muss. Damit wird auch das Ziel „Seiteneffektfreiheit“ erreicht.

Es muss darüber hinaus festgehalten werden, dass beim Entwurf der Datenstrukturen Probleme aufgetaucht sind. Diese wurde zusätzlich und unabhängig von der Zieldefinition behandelt und es wurden Lösungsvorschläge vorgestellt. Da die Probleme nicht prinzipieller Natur sind, sondern Grenzen der eingesetzten funktionalen Sprache aufzeigen, sind die Lösungsvorschläge von Nutzen. Die Arbeit hält an vielen Stellen Lösungen für einen flexiblen Entwurf von Netzlistenbeschreibungen bereit. Deshalb wird angeregt, diese Arbeit als Ausgangspunkt weiterer Forschung auf diesem Gebiet zu nutzen.

A. Anhang

A.1. Usermanual

Ziel dieser Software ist es, das Entwickeln von elektronischen Hardwarebausteinen mithilfe funktionaler Arrows zu unterstützen. Ein Baustein wird durch die Darstellung als Arrow beschrieben. Eine solche Darstellung entspricht einer Netzliste und lässt sich in unterschiedliche andere Formate transformieren. Mit dieser Software lassen sich drei unterschiedliche Darstellungweisen erzeugen.

- Zum einen lässt sich ein Arrow in eine einfache textuelle Darstellung überführen, die einen schnellen Überblick über die Schaltung ermöglicht.
- Zum anderen wird die Erzeugung einer VHDL-Darstellung aus einem Arrow ermöglicht.
- Zuletzt lassen sich die Arrows in die Dot-Notation der „graphviz"-Bibliothek überführen, um daraus wiederum eine graphische Darstellung ableiten zu können.

A.1.1. Setup

Die Software ist an mehreren Stellen erhältlich. Der Quelltext lässt sich als gezippter Tarball von Haskells Paketserver „hackage" [1] beziehen.

Das Git-Repository ist über den öffentlichen Dienst „github" abrufbar. Eine Kopie lässt sich mit folgendem Befehl beziehen:

```
> git clone https://github.com/frosch03/arrowVHDL.git
```

[1] `http://hackage.haskell.org/package/ArrowVHDL-1.1/ArrowVHDL-1.1.tar.gz`

Die Software ist auch direkt über das Haskell-Paketverwaltungssystem „cabal“ installierbar. Diese Methode wird empfohlen, da „cabal“ sämtliche Paketabhängigkeiten auflöst und die entsprechenden Pakete mit installiert. Die Installation über „cabal“ wird mit folgendem Befehl eingeleitet:

```
> cabal update
> cabal install ArrowVHDL
```

Da die Software für eine interaktive Nutzung aus dem Haskellcompiler „ghci“ heraus gedacht ist, wird keine weitere Installation notwendig.

A.1.2. Grundlegende Funktionalität

Der Funktionskanon zur Erzeugung der Darstellungen aus der Arrowbeschreibung wird im Folgenden dargestellt.

- `flatten` - Mit „flatten“ lassen sich sämtliche atomaren Teile einer Netzliste aus einer generierten Darstellung erzeugen. Diese Funktion wird für alle komplexeren Schaltungen benötigt, um eine sinnvolle Netzliste als Ergebnis zu erhalten.
- `synthesize` - Diese Funktion generiert aus der Arrowdarstellung die Darstellung in eine der ausgewählten Beschreibungssprachen. Das Ergebnis für komplexere Schaltungen sollte dann als Eingabe der Funktion `flatten` zugeführt werden, um eine sinnvolle Darstellung zu erzeugen.
- `simulate` - Mit „simulate“ lässt sich der ausführbare Teil der Schaltung extrahieren. Um eine Schaltung zu simulieren, muss das Ergebnis dieser Funktion als Eingabe der Funktion `runStream` übergeben werden.
- `runStream` - Diese Funktion wird verwendet, um den ausführbaren Teil der Arrowdarstellung auf eine Liste von Eingabewerten anzuwenden.

A.1.3. Beispiel

Zum interaktiven Arbeiten mit der Arrowdarstellung muss die Quelltextdatei (hier `Beispiel.hs`) in die „ghci“-Session geladen werden. Dies geschieht in der Kommandozeile mit dem Befehl:

```
ghci Beispiel.hs
Ok, modules loaded: ...
Beispiel>
```

Die Arrowdarstellungen aus `Beispiel.hs` sind nun in der „ghci“-Session verfügbar.

Data Types

Zunächst lässt sich der Datentyp einer Arrowdarstellung mit dem Kommando `:t` abrufen:

```
Beispiel> :t aTest0
aTest0 :: (Arrow a, Num c) => Grid a c c
```

Der Schaltkreis `aTest0` stellt eine Verdopplung dar. Der Datentyp spiegelt dies wider. Jede Zahl `c` kann in eine Zahl `c` überführt werden. Die Schaltung erreicht dies, indem die Eingabe verdoppelt wird und dies als Input an ein Additionsbauteil weitergeleitet wird.

Simulation

Die Simulation einer Schaltung erfolgt, indem der Schaltung eine Liste mit Eingangswerten überreicht wird. Die Schaltung wird sukzessive auf die Eingabe der Liste angewandt und das Ergebnis der Ergebnisliste angehängt.

Dies wird erreicht, indem `simulate` auf aTest0 angewandt wird. Das Ergebnis daraus wird der Funktion `runStream` übergeben. `runStream` liefert eine weitere Funktion zurück, die auf eine Liste von Werten angewandt werden kann.

```
Beispiel> (runStream (simulate aTest0)) [1..10]
[2,4,6,8,10,12,14,16,18,20]
```

Da Haskell den Umgang mit unendlichen Listen unterstützt, lässt sich eine Schaltung auch auf unendliche Listen anwenden.

```
Beispiel> (runStream (simulate aTest0)) (repeat 23)
[46,46,46,46,46,46,46,46,46,46,46,46,46,46,46,...]
```

Code Generation

Um Code aus der Arrowdarstellung zu erzeugen, muss die Funktion `synthesize` verwendet werden. Diese Funktion überführt einen Arrow in eine der weiteren Darstellungen. Die aktuell unterstützten Formen sind graphvizs dot-Darstellung wie auch VHDL-Code. Der Prozess der Quelltexterstellung wird eingeleitet mit:

```
Beispiel> (synthesize aShiftR5_XorKey)
```

Die interne Umsetzung der Erzeugung von Netzlisten aus Arrowbeschreibungen würde bei dem oben genannten Aufruf lediglich eine Netzliste mit zwei Bauteilen erzeugen. Um eine aussagekräftigere Netzliste zu erhalten, sollte das Ergebnis noch der Funktion `flatten` übergeben werden.

```
Beispiel> (flatten (synthesize aShiftR5_XorKey))
```

Dieser Aufruf erzeugt die vollständige Netzliste und gibt diese in der ausgewählten Sprache zurück.

Umstellen der Beschreibungsmethode

Die Auswahl der Sprache, die aus der Arrowbeschreibung erstellt werden kann, wird direkt im Quelltext der ArrowVHDL-Bibliothek eingestellt. Hierzu wird im **Show**-Modul die entsprechende Darstellungsbibliothek inkludiert. Um dies zu verändern, muss die Datei `Show.hs` im Ordner `./System/ArrowVHDL/Circuit/` geöffnet werden.

In dieser Datei sind drei Importanweisungen vorhanden. Nur eine ist tatsächlich aktiv, die beiden anderen sollten auskommentiert sein. Zum Ändern muss nun lediglich die gewünschte Variante auskommentiert werden, die anderen beiden Varianten sollten als Kommentar in der Quelldatei verbleiben. Der entsprechende Teil des **Show**-Moduls könnte dabei wie folgt aussehen:

```
-- import System.ArrowVHDL.Circuit.Show.Simple
import System.ArrowVHDL.Circuit.Show.VHDL
-- import System.ArrowVHDL.Circuit.Show.Dot
```

Nach diesem Schritt müssen alle laufenden „ghci“-Sessions beendet werden. Nach einem erneuten Einlesen der „ArrowVHDL“-Bibliothek ist diese wieder verwendbar.

Sofern mehr als eine Darstellungsform im **Show**modul auskommentiert bleibt, wird sich der Compiler mit einem `Ambiguous occurrence` Fehler beenden.

```
System/ArrowVHDL/Circuit/Show.hs:42:12:
Ambiguous occurrence ‘showCircuit’
It could refer to either ‘Simple.showCircuit’,
imported from ‘System.ArrowVHDL.Circuit.Show....
```

A.2. ALU

```haskell
module ALU
where

import Prelude hiding (id, (.))

import Control.Category -- here we get >>> ...

import Circuit
import Circuit.Arrow -- here we get first and second
import Circuit.Defaults

type Input   = (Bool, Bool)
type Output  = Bool
type Cin     = Bool
type Cout    = Bool
type Opt1Bit = Bool
type Opt2Bit = (Opt1Bit, Opt1Bit)
type Opt3Bit = (Opt1Bit, Opt2Bit)
type Opt4Bit = (Opt1Bit, Opt3Bit)

type In2Bit  = (Input,  (Input))
type Out2Bit = (Output, (Output))
```

```
23
24 type In4Bit  = (Input,  (Input,  (Input,  (Input))))
25 type Out4Bit = (Output, (Output, (Output, (Output))))
26
27 type In8Bit  = (Input,  (Input,  (Input,  (Input,
28                (Input,  (Input,  (Input,  (Input)))))))
29 type Out8Bit = (Output, (Output, (Output, (Output,
30                (Output, (Output, (Output, (Output)))))))
31
32
33 -- aFst
34 -- aSnd
35 -- aXor
36 -- aShiftL
37 -- aShiftR
38 -- aAdd
39
40
41 aDdistr_new :: (Arrow a) => Grid a ((b, c), (b, e)) (b, (c, e))
42 aDdistr_new
43     = proc ((x1, y), (x2, z)) -> do
44         returnA -< (x1, (y, z))
45
46
47 -- |With the 'noC' operator, one can shortwire the CarryIn bit direct to the
48 -- CarryOut bit
49 noC :: (Arrow a) => (Grid a Input Output) -> Grid a (Cin, Input) (Output, Cout)
50 noC arrow
51     =   second arrow
52     >>> aFlip
53
54
55 -- |'aFullAdd' is the full adder implementation
56 aFullAdd :: (Arrow a) => Grid a (Cin, Input) (Output, Cout)
57 aFullAdd
58     =   second (aXor &&& aAnd)
59     >>> a_aBC2ABc
60     >>> first  (aXor &&& aAnd)
61     >>> a_ABc2aBC
62     >>> second (aOr)
63
64 -- Multiplexer --
65
66 -- | With 'aMux' a 1 Bit multiplexer is defined
```

```
67  aMux :: (Arrow a) => Grid a (Opt1Bit, (Output, Output))  Output
68  aMux
69      =   aDistr
70      >>> first (first aNot)
71      >>> aAnd *** aAnd
72      >>> aOr
73
74  aMux2Bit :: (Arrow a) --         00      01         10      11
75           => Grid a (Opt2Bit, ((Output, Output), (Output, Output)))  Output
76  aMux2Bit
77      =   a_ABc2aBC
78      >>> second aDistr
79      >>> second (aMux *** aMux)
80      >>> aMux
81
82  aMux3Bit :: (Arrow a) --          000     001        010     011
83           => Grid a (Opt3Bit, (((Output, Output), (Output, Output)),
84                   --      100     101        110     111
85                      ((Output, Output), (Output, Output))))  Output
86  aMux3Bit
87      =   a_ABc2aBC
88      >>> second aDistr
89      >>> second (aMux2Bit *** aMux2Bit)
90      >>> aMux
91
92  aMux4Bit :: (Arrow a)
93           => Grid a (Opt4Bit, ( (((Output, Output), (Output, Output)),
94                                  ((Output, Output), (Output, Output)))
95                               , (((Output, Output), (Output, Output)),
96                                  ((Output, Output), (Output, Output)))))  Output
97  aMux4Bit
98      =   a_ABc2aBC
99      >>> second aDistr
100     >>> second (aMux3Bit *** aMux3Bit)
101     >>> aMux
102
103 -- |'anXum' is the Multiplexer where the last input-pin is the s-line
104 -- |it is generated out of one of the mux'es
105 anXum mux =   aFlip
106           >>> mux
107
108 aXum  = anXum aMux
109 a2Xum = anXum aMux2Bit
110 a3Xum = anXum aMux3Bit
```

```
111 a4Xum = anXum aMux4Bit
112
113 --_____________--
114
115 --eval :: (Arrow a) => Grid a (Cin, (Opt1Bit, Input)) (Output, Cout)
116   -> Grid a (Cin, (Opt1Bit, (Input, rest))) (Output, (Cout, (Opt1Bit, rest)))
117 -- |The 'eval' function takes a single bit ALU and evaluates the first bit of
118 -- a multiBit input so with 'eval' one can define the steps of n-Bit ALU
119 eval aALU
120     =   second aDistr
121     >>> a_aBC2ABc
122     >>> first aALU
123     >>> a_ABc2aBC
124
125 -- n-Bit ALU's --
126
127 --a1BitALU :: (Arrow a) => Grid a (Cin, (Opt1Bit, Input))   (Output, Cout)
128 --                       -> Grid a (Cin, (Opt1Bit, (Input))) (Output, (Cout))
129 mk1BitALU = id
130
131
132 --a2BitALU :: (Arrow a) => Grid a (Cin, (Opt1Bit, Input)) (Output, Cout)
133 --                    -> Grid a (Cin, (Opt1Bit, (Input,  (Input))))
134 --                                              (Output, (Output, (Cout)))
135 mk2BitALU aALU
136     = eval aALU >>> next aALU
137     where next = second
138
139
140 --a4BitALU :: (Arrow a) => Grid a (Cin, (Opt1Bit, Input)) (Output, Cout)
141 --     -> Grid a (Cin, (Opt1Bit, (Input,  (Input,  (Input,  (Input))))))
142 --                                (Output, (Output, (Output, (Output, (Cout)))))
143 mk4BitALU aALU
144     = eval aALU >>> next
145     ( eval aALU >>> next
146     ( eval aALU >>> next aALU))
147     where next = second
148
149
150 --a8BitALU :: Arrow a => Grid a (Cin, (Opt1Bit, Input)) (Output, Cout)
151 --    -> Grid a (Cin, (Opt1Bit, (Input,  (Input,  (Input,  (Input,
152 --                               (Input,  (Input,  (Input,  Input)))))))))
153 --                               (Output, (Output, (Output, (Output, (Output,
154 --                               (Output, (Output, (Output, Cout)))))))))
```

```
155 mk8BitALU aALU
156     = eval aALU >>> next
157     ( eval aALU >>> next
158     ( eval aALU >>> next
159     ( eval aALU >>> next
160     ( eval aALU >>> next
161     ( eval aALU >>> next
162     ( eval aALU >>> next aALU))))))
163     where next = second
164
165
166 -- n-Bit ALU's --
167 --_____________--
168
169 -- n-Bit Operators --
170
171 aOpt1Bit :: (Arrow a)
172          => Grid a (Cin, Input) (Cout, Output)
173          -> Grid a (Cin, Input) (Cout, Output)
174          -> Grid a (Cin, (Opt1Bit, Input)) (Output, Cout)
175 aOpt1Bit aOp0 aOp1
176     =   a_aBC2ABc
177     >>> first aFlip
178     >>> a_ABc2aBC
179     >>> second
180         (   aOp0 &&& aOp1
181         >>> aDdistr
182         )
183     >>> aDistr
184     >>> aMux *** aMux
185
186
187 aOpt2Bit :: (Arrow a)
188          => Grid a (Cin, Input) (Cout, Output)
189          -> Grid a (Cin, Input) (Cout, Output)
190          -> Grid a (Cin, Input) (Cout, Output)
191          -> Grid a (Cin, Input) (Cout, Output)
192          -> Grid a (Cin, (Opt2Bit, Input)) (Output, Cout)
193 aOpt2Bit aOp00 aOp01 aOp10 aOp11
194     =   a_aBC2ABc
195     >>> first aFlip
196     >>> a_ABc2aBC
197     >>> second
198         (   (aOp00 &&& aOp01) &&& (aOp10 &&& aOp11)
```

```
199         >>> aDdistr *** aDdistr
200         >>> aDdistr
201         )
202     >>> aDistr
203     >>> aMux2Bit *** aMux2Bit
204
205
206 aOpt3Bit :: (Arrow a)
207          => Grid a (Cin, Input) (Cout, Output)
208          -> Grid a (Cin, Input) (Cout, Output)
209          -> Grid a (Cin, Input) (Cout, Output)
210          -> Grid a (Cin, Input) (Cout, Output)
211          -> Grid a (Cin, Input) (Cout, Output)
212          -> Grid a (Cin, Input) (Cout, Output)
213          -> Grid a (Cin, Input) (Cout, Output)
214          -> Grid a (Cin, Input) (Cout, Output)
215          -> Grid a (Cin, (Opt3Bit, Input)) (Output, Cout)
216 aOpt3Bit aOp000 aOp001 aOp010 aOp011 aOp100 aOp101 aOp110 aOp111
217     =   a_aBC2ABc
218     >>> first aFlip
219     >>> a_ABc2aBC
220     >>> second
221         (    ((aOp000 &&& aOp001) &&& (aOp010 &&& aOp011))
222          &&& ((aOp100 &&& aOp101) &&& (aOp110 &&& aOp111))
223         >>> (aDdistr *** aDdistr) *** (aDdistr *** aDdistr)
224         >>> aDdistr *** aDdistr
225         >>> aDdistr
226         )
227     >>> aDistr
228     >>> aMux3Bit *** aMux3Bit
229
230
231
232 aOpt4Bit :: (Arrow a)
233          => Grid a (Cin, Input) (Cout, Output)
234          -> Grid a (Cin, Input) (Cout, Output)
235          -> Grid a (Cin, Input) (Cout, Output)
236          -> Grid a (Cin, Input) (Cout, Output)
237          -> Grid a (Cin, Input) (Cout, Output)
238          -> Grid a (Cin, Input) (Cout, Output)
239          -> Grid a (Cin, Input) (Cout, Output)
240          -> Grid a (Cin, Input) (Cout, Output)
241          -> Grid a (Cin, Input) (Cout, Output)
242          -> Grid a (Cin, Input) (Cout, Output)
```

```
243           -> Grid a (Cin, Input) (Cout, Output)
244           -> Grid a (Cin, Input) (Cout, Output)
245           -> Grid a (Cin, Input) (Cout, Output)
246           -> Grid a (Cin, Input) (Cout, Output)
247           -> Grid a (Cin, Input) (Cout, Output)
248           -> Grid a (Cin, Input) (Cout, Output)
249           -> Grid a (Cin, (Opt4Bit, Input)) (Output, Cout)
250 aOpt4Bit aOp0000 aOp0001 aOp0010 aOp0011 aOp0100 aOp0101 aOp0110 aOp0111
251 aOp1000  aOp1001 aOp1010 aOp1011 aOp1100 aOp1101 aOp1110 aOp1111
252     =   a_aBC2ABc
253     >>> first aFlip
254     >>> a_ABc2aBC
255     >>> second
256         (       (((aOp0000 &&& aOp0001) &&& (aOp0010 &&& aOp0011))
257          &&&    ((aOp0100 &&& aOp0101) &&& (aOp0110 &&& aOp0111)))
258             &&& (((aOp1000 &&& aOp1001) &&& (aOp1010 &&& aOp1011))
259          &&&    ((aOp1100 &&& aOp1101) &&& (aOp1110 &&& aOp1111)))
260         >>> ((aDdistr *** aDdistr) *** (aDdistr *** aDdistr))
261         *** ((aDdistr *** aDdistr) *** (aDdistr *** aDdistr))
262         >>> (aDdistr *** aDdistr) *** (aDdistr *** aDdistr)
263         >>> aDdistr *** aDdistr
264         >>> aDdistr
265         )
266     >>> aDistr
267     >>> aMux4Bit *** aMux4Bit
268
269
270 -- n-Bit Operators --
271 --------------------
272
273 aFAD_XOR_AND_OR = aOpt2Bit aFullAdd (noC aXor) (noC aAnd) (noC aOr)
274
275 -- Auxillary --
276
277 t8b :: Int -> (Bool, (Bool, (Bool, (Bool, (Bool, (Bool, (Bool, (Bool))))))))
278 t8b x
279     = (bit7, (bit6, (bit5, (bit4, (bit3, (bit2, (bit1, (bit0))))))))
280     where (bit7: bit6: bit5: bit4: bit3: bit2: bit1: bit0: []) = i2t8b [] x
281
282 i2t8b :: [Bool] -> Int -> [Bool]
283 i2t8b list x
284     | x == 0
285     = list ++ (replicate (8 - (length list)) False)
286
```

```
287     | x == 1
288     = (list ++ [True]) ++ (replicate (8 - (length list) - 1) False)
289
290     | x `mod` 2  == 0
291     = i2t8b (list ++ [False]) (x `div` 2)
292
293     | x `mod` 2 == 1
294     = i2t8b (list ++ [True]) ( (x-1) `div` 2)
295
296 inp8b :: Int -> Int -> In8Bit
297 inp8b x1 x2
298     = ((bit07, bit17), ((bit06, bit16), ((bit05, bit15), ((bit04, bit14),
299       ((bit03, bit13), ((bit02, bit12), ((bit01, bit11), ((bit00, bit10)))))))))
300 where (bit07: bit06: bit05: bit04: bit03: bit02: bit01: bit00: []) = i2t8b [] x1
301       (bit17: bit16: bit15: bit14: bit13: bit12: bit11: bit10: []) = i2t8b [] x2
```

A.3. Beispiel

```
1  module Beispiel where
2
3  import Control.Category
4  import Prelude hiding (id, (.))
5
6  import Circuit.Arrow
7  import Circuit.Auxillary
8  import Circuit.Descriptor
9
10 import TEA
11 import Circuit.Workers
12
13 import Circuit
14 import Circuit.Defaults
15         ( aId
16         , aConst
17         , aDup
18         , aFlip
19         , aAdd
20         , aXor
21         , aShiftL, aShiftL4
22         , aShiftR, aShiftR5
```

```
23          , aXorMagic
24          , aFst, aSnd
25          )
26
27  aAdd1 :: (Arrow a) => a Int Int
28  aAdd1 = arr (\x -> x +1)
29
30  aSub1 :: (Arrow a) => a Int Int
31  aSub1 = arr (\x -> x -1)
32
33  aSub2 = aSub1 >>> aSub1
34
35
36
37  -- Beispiel 0
38  -------------
39  aTest0
40      =    aDup
41      >>> aAdd
42
43  netlist_Test0 :: CircuitDescriptor
44  netlist_Test0
45      = synthesize aTest0
46
47
48  -- Beispiel 1
49  -------------
50  aTest1
51      = proc (x) -> do
52          tmp <- aAdd -< (x, x)
53          returnA      -< tmp
54
55  _netlist_Test1
56      = synthesize aTest1
57
58
59  -- Beispiel 2
60  -------------
61  aTest2
62      = proc (x1, x2) -> do
63          tmp1 <- aDup -< x1
64          tmp2 <- aDup -< x2
65          tmp3 <- aAdd -< tmp1
66          tmp4 <- aAdd -< tmp2
```

```
67          tmp5 <- aAdd -< (tmp3, tmp4)
68          returnA      -< tmp5
69
70  _netlist_Test2
71      = synthesize aTest2
72
73
74  -- Beispiel 3
75  -------------
76  aTest2'
77      = ((first aDup >>> arr (\ (tmp1, x2)   -> (x2,   tmp1))) >>>
78         (first aDup >>> arr (\ (tmp2, tmp1) -> (tmp1, tmp2))) >>>
79         (first aAdd >>> arr (\ (tmp3, tmp2) -> (tmp2, tmp3))) >>>
80         (first aAdd >>> arr (\ (tmp4, tmp3) -> (tmp3, tmp4))) >>> aAdd)
81
82  netlist_Test2'
83      = synthesize aTest2'
84
85
86  -- Beispiel 4
87  -------------
88  aShiftL4_XorKey
89      =   first (   aDup
90                >>> second (aConst 4)
91                >>> aShiftL
92                )
93      >>> aXor
94
95  -- netlist_ShiftL4_XorKey
96  --      = synthesize aShiftL4_XorKey
97
98
99  -- Beispiel 5
100 -------------
101 -- delta = 2654435769
102 -- aXorDelta
103 --      =   second (aConst delta)
104 --      >>> aXor
105
106 -- netlist_XorDelta
107 --      = synthesize aXorDelta
108
109
110 -- Beispiel 6
```

```
111 -------------
112 aShiftR5_XorKey
113     =   first (   aDup
114               >>> second (aConst 5)
115               >>> aShiftR
116               )
117     >>> aXor
118
119 -- netlist_ShiftR5_XorKey
120 --      = synthesize aShiftR5_XorKey
121
122
123 counter :: (ArrowCircuit a) => a Int Int
124 counter = proc reset -> do
125             rec output <- (arr (+1)) -< reset
126                 next   <- delay 0    -< output
127             returnA -< output
128
129
130 aLoopBsp :: (ArrowLoop a) => Grid a Int Int
131 aLoopBsp
132     = loop (aAdd >>> (aId &&& (aConst 4)))
```

A.4. CRC

```
1  module CRC
2  where
3
4  import Prelude
5
6  import Control.Category ((>>>))
7
8  import Circuit
9  import Circuit.Arrow
10 import Circuit.Auxillary
11 import Circuit.Defaults
12
13 import Data.Bits
14
15 infixr 0 >:>
16
```

```
17 (>:>) aA aB = aA >>> (second aB)
18
19 mvRight :: (Arrow a) => Grid a (my, (b, rst)) (b, (my, rst))
20 mvRight
21     =   aDistr
22     >>> first aSnd
23
24
25
26 -- crc_polynom_ccitt ::  (Arrow a, Bits b)
27 --                    => Grid a (b, (b, (b, (b, b))))  (b, (b, (b, b)))
28 crc_polynom_ccitt ::  (Arrow a) => Grid a (Bool, (Bool, (Bool, (Bool, Bool))))
29                                           (Bool, (Bool, (Bool, Bool)))
30 crc_polynom_ccitt
31     =   mvRight >:> mvRight >:>
32         (   aDistr
33         >>> (aXor *** aXor)
34         )
35
36 -- crc_polynom_ccitt
37 --  = proc (x4, (x3, (x2, (x1, x0)))) -> do
38 --      o1 <- aXor -< (x4, x0)
39 --      o2 <- aXor -< (x4, x1)
40 --      o3 <- aId  -< (x2)
41 --      o4 <- aId  -< (x3)
42 --      returnA    -< (o4, (o3, (o2, o1)))
43
44 -- crc_polynom_usb ::   (Arrow a, Bits b) => Grid a (b, (b, (b, (b, (b, b)))))
45 --                                                  (b, (b, (b, (b, b))))
46 crc_polynom_usb ::   (Arrow a)
47                 => Grid a (Bool, (Bool, (Bool, (Bool, (Bool, Bool)))))
48                           (Bool, (Bool, (Bool, (Bool, Bool))))
49 crc_polynom_usb
50     =   mvRight >:> mvRight >:>
51         (   aDistr
52         >>> aXor *** (mvRight >:> aXor)
53         )
54
55 -- crc_polynom_usb
56 --  = proc (x5, (x4, (x3, (x2, (x1, x0))))) -> do
57 --      o1 <- aXor -< (x5, x0)
58 --      o2 <- aId  -< (x1)
59 --      o3 <- aXor -< (x5, x2)
60 --      o4 <- aId  -< (x3)
```

```
61  --      o5 <- aId  -< (x4)
62  --      returnA    -< (o5, (o4, (o3, (o2, o1))))
63
64  -- crc_polynom_sdmmc :: (Arrow a, Bits b)
65                       => Grid a (b, (b, (b, (b, (b, (b, (b, b)))))))
66                                 (b, (b, (b, (b, (b, (b, b))))))
67  crc_polynom_sdmmc :: (Arrow a)
68            => Grid a (Bool, (Bool, (Bool, (Bool, (Bool, (Bool, (Bool, Bool)))))))
69                      (Bool, (Bool, (Bool, (Bool, (Bool, (Bool, Bool))))))
70  crc_polynom_sdmmc
71      =   mvRight >:> mvRight >:> mvRight >:>
72          (   aDistr
73          >>> aXor *** (mvRight >:> mvRight >:> aXor)
74          )
75
76  -- crc_polynom_sdmmc
77  --  = proc (x7, (x6, (x5, (x4, (x3, (x2, (x1, x0)))))))) -> do
78  --      o1 <- aXor -< (x7, x0)
79  --      o2 <- aId  -< (x1)
80  --      o3 <- aId  -< (x2)
81  --      o4 <- aXor -< (x7, x3)
82  --      o5 <- aId  -< (x4)
83  --      o6 <- aId  -< (x5)
84  --      o7 <- aId  -< (x6)
85  --      returnA    -< (o7, (o6, (o5, (o4, (o3, (o2, o1))))))
86
87
88
89  --                                                3   2   1   0      2   1   0
90  -- inner_crc_3ord :: (Arrow a, Bits b) => Grid a (b, (b, (b, b)))  (b, (b, b))
91  inner_crc_3ord
92      =   aDistr
93      >>> aSnd *** aDistr
94      >>> second (aXor *** aXor)
95
96
97  toInner3
98      =   mvRight
99      >:> mvRight
100     >:> aFlip
101
102 toInner4
103     =   mvRight
104     >:> mvRight
```

```
105     >:> mvRight
106     >:> aFlip
107
108 toInner5
109     =   mvRight
110     >:> mvRight
111     >:> mvRight
112     >:> mvRight
113     >:> aFlip
114
115 toInner7
116     =   mvRight
117     >:> mvRight
118     >:> mvRight
119     >:> mvRight
120     >:> mvRight
121     >:> mvRight
122     >:> aFlip
123
124 toInner8
125     =   mvRight
126     >:> mvRight
127     >:> mvRight
128     >:> mvRight
129     >:> mvRight
130     >:> mvRight
131     >:> mvRight
132     >:> aFlip
133
134
135 -- crc_checksum_8 crc_polynom polySkip start rest padding
136 crc_checksum_8 crc_polynom polySkip start rest padding
137     =   (padding &&& aId)
138     >>> toInner8
139
140     >>> (start &&& rest)
141     >>> first (crc_polynom)
142
143     >>> step
144     >>> step
145     >>> step
146     >>> step
147     >>> step
148     >>> step
```

```
149
150     >>> aFlip
151     >>> polySkip
152     >>> crc_polynom
153
154     where step =   a_aBC2ABc
155                >>> first
156                     (   aFlip
157                     >>> polySkip
158                     >>> crc_polynom
159                     )
160
161
162 -- crc_test
163 --     = crc_checksum_8
164 --         inner_crc_3ord
165 --         toInner3
166 --         (second (second (second aFst)))
167 --         (aSnd >>> aSnd >>> aSnd >>> aSnd)
168 --         (aConst (False, (False, False)))
169
170
171 -- crc_checksum_ccitt_8 ::  (Arrow a, Bits b)
172 --                      => Grid a (b, (b, (b, (b, (b, (b, (b, b)))))))
173 --                                (b, (b, (b, b)))
174 crc_checksum_ccitt_8 ::  (Arrow a)
175           => Grid a (Bool, (Bool, (Bool, (Bool, (Bool, (Bool, (Bool, Bool)))))))
176                     (Bool, (Bool, (Bool, Bool)))
177 crc_checksum_ccitt_8
178     = crc_checksum_8
179         crc_polynom_ccitt
180         toInner4
181         (second.second.second.second $ aFst)
182         (aSnd >>> aSnd >>> aSnd >>> aSnd >>> aSnd)
183         (aConst (False, (False, (False, False))))
184
185 -- crc_checksum_usb_8 ::   (Arrow a, Bits b)
186 --                     => Grid a (b, (b, (b, (b, (b, (b, (b, b)))))))
187 --                               (b, (b, (b, (b, b))))
188 -- crc_checksum_usb_8
189 --     = crc_checksum_8
190 --         crc_polynom_usb
191 --         toInner5
192 --         (second.second.second.second.second $ aFst)
```

```
193 --          (aSnd >>> aSnd >>> aSnd >>> aSnd >>> aSnd >>> aSnd)
194 --          (aConst (False, (False, (False, (False, False)))))
195 
196 -- crc_checksum_sdmmc_8 :: (Arrow a, Bits b)
197 --                      => Grid a (b, (b, (b, (b, (b, (b, (b, b)))))))
198 --                                (b, (b, (b, (b, (b, (b, b))))))
199 -- crc_checksum_sdmmc_8
200 --     = crc_checksum_8
201 --          crc_polynom_sdmmc
202 --          toInner7
203 --          (second.second.second.second.second.second.second $ aFst)
204 --          (aSnd >>> aSnd >>> aSnd >>> aSnd >>> aSnd >>> aSnd >>> aSnd >>> aSnd)
205 --          (aConst (False, (False, (False, (False, (False, (False, False)))))))
```

A.5. TEA

```
1  module TEA where
2  
3  import Data.Char (digitToInt, ord, chr)
4  import qualified Data.Bits as B
5  import Data.Word (Word32(..))
6  import Prelude hiding (cycle)
7  
8  import Control.Category
9  import Circuit
10 import Circuit.Arrow
11 import Circuit.Defaults hiding (Value, Key)
12 
13 type Key     = (Word32, Word32, Word32, Word32)
14 type HalfKey = (Word32, Word32)
15 type Value   = (Word32, Word32)
16 type Round   = Int
17 
18 type TL4
19     = (Bool, (Bool, (Bool, (Bool))))
20 
21 type TL32
22     = (Bool, (Bool, (Bool, (Bool, (Bool, (Bool, (Bool, (Bool,
23       (Bool, (Bool, (Bool, (Bool, (Bool, (Bool, (Bool, (Bool,
24       (Bool, (Bool, (Bool, (Bool, (Bool, (Bool, (Bool, (Bool,
25       (Bool, (Bool, (Bool, (Bool, (Bool, (Bool, (Bool,  Bool
```

```
      ))))))))))))))))))))))))))))))))
type KeyTL     = (TL32, TL32, TL32, TL32)
type HalfKeyTL = (TL32, TL32)
type ValueTL   = (TL32, TL32)
type RoundTL   = TL32

-- aXorTL4
--   =     aXor
--     >:> aXor
--     >:> aXor
--     >:> aXor
--   where
--     aA >:> aB = ((aFst *** aFst) >>> aA) &&& ((aSnd *** aSnd) >>> aB)
--     infixr 4 >:>

-- aXorTL32
--   =     aXor >:> aXor >:> aXor >:> aXor >:> aXor >:> aXor >:> aXor >:> aXor
--     >:> aXor >:> aXor >:> aXor >:> aXor >:> aXor >:> aXor >:> aXor >:> aXor
--     >:> aXor >:> aXor >:> aXor >:> aXor >:> aXor >:> aXor >:> aXor >:> aXor
--     >:> aXor >:> aXor >:> aXor >:> aXor >:> aXor >:> aXor >:> aXor >:> aXor
--   where
--     aA >:> aB = ((aFst *** aFst) >>> aA) &&& ((aSnd *** aSnd) >>> aB)
--     infixr 4 >:>

-- aOrTL32
--   =     aOr >:> aOr >:> aOr >:> aOr >:> aOr >:> aOr >:> aOr >:> aOr
--     >:> aOr >:> aOr >:> aOr >:> aOr >:> aOr >:> aOr >:> aOr >:> aOr
--     >:> aOr >:> aOr >:> aOr >:> aOr >:> aOr >:> aOr >:> aOr >:> aOr
--     >:> aOr >:> aOr >:> aOr >:> aOr >:> aOr >:> aOr >:> aOr >:> aOr
--   where
--     aA >:> aB = ((aFst *** aFst) >>> aA) &&& ((aSnd *** aSnd) >>> aB)
--     infixr 4 >:>

magicConstant = 0x9e3779b9

-- |‘delta‘ is the function, that considering the round number,
-- calculates the magic number.
delta :: Int -> Word32
delta r = fromIntegral r * magicConstant


-- |Here comes a function ‘cycleN‘ that calculates the cypher of the
-- TEA. It therefore takes a round counter and calculates the TEA with
-- exact that amount of rounds.
```

```haskell
cycleN :: Int -> Key -> Value -> Value
cycleN n = cycleN' (1, n)

-- |Another more specific version of `cycleN` is `cycleN'`. This
-- function has a more complex round counter (Tuple of Int vs. single
-- Int). The first element of the tuple is the actual round counter;
-- the second element count's backward and acts as the recursion
-- break.
cycleN' :: (Round, Int) -> Key -> Value -> Value
cycleN' (_,0) _ v = v
cycleN' (r,n) k v = cycleN' (r+1, n-1) k (cycle r k v)

-- |`cycle` without any naming suffixes is the function that actually
-- calculates one round of the TEA.
cycle :: Round -> Key -> Value -> Value
cycle r (k0, k1, k2, k3) value
  = let value' = feistel1 value
    in           feistel2 value'
  where feistel1 = feistel r (k0, k1)
        feistel2 = feistel r (k2, k3)

-- |The `feistel` function contains the logic of the TEA. It is
-- similar to the depicted definition inside the wikipedia-article on
-- TEA: https://en.wikipedia.org/wiki/Tiny_Encryption_Algorithm
feistel :: Round -> HalfKey -> Value -> Value
feistel r (kA, kB) (vA, vB)
  = (vB, vA + tB)
  where t1 = kA + (vB `B.shift`    4)
        t2 = vB + delta r
        t3 = kB + (vB `B.shift` (-5))
        tB = t1 `B.xor` t2 `B.xor` t3

-- |The `feistel'functional` function contains the logic of the TEA. It is
-- similar to the depicted definition inside the wikipedia-article on
-- TEA: https://en.wikipedia.org/wiki/Tiny_Encryption_Algorithm
feistel'functional :: Round -> HalfKey -> Value -> Value
feistel'functional r (kA, kB) (vA, vB)
  = (vB, vA + tB)
  where t1 = kA + (vB `B.shift`    4)
        t2 = vB + delta r
        t3 = kB + (vB `B.shift` (-5))
        tB = t1 `B.xor` t2 `B.xor` t3

feistel'arrow :: Round -> HalfKey -> Value -> Value
```

```
114 feistel'arrow r key val
115   = runGrid aTEA $ (key, val)
116 -- ((K0, K1), (V0, V1))
117   where aTEA :: (Arrow a) => Grid a (HalfKey, Value) Value
118 -- (((K0, K1), V0), ((K0, K1), V1))
119         aTEA =   aDistr
120              >>> first
121 -- (V0, ((K0, K1), V1))
122                  (   aSnd
123                  )
124              >>> second
125 -- (V0, ((V1, (K0, K1)), V1))
126                  ( (     aFlip -- &&& aSnd
127 -- (V0, ((((V1, K0), (V1, K1)), V1), V1))
128                      >>> aDistr &&& aFst
129                      >>> (   ((first aSHL4)
130                          *** (first aSHR5))
131                          *** aXorDeltaR
132 -- (V0, ((((S1, K0), (S1, K1)), T3), V1))
133                          )
134 -- (V0, (((T1, T2), T3), V1))
135                      >>> first (aXor *** aXor)
136 -- (V0, ((T12, T3), V1))
137                      >>> first aXor
138 -- (V0, (TB, V1))
139                      >>> aXor
140                    )
141                    &&& aSnd
142                  )
143              >>> a_aBC2ABc
144              >>> first aXor
145              >>> aFlip
146         aSHL4 :: (Arrow a, B.Bits b) => Grid a b b
147         aSHL4 = (aId &&& (aConst (4 :: Int))) >>> aShiftL
148
149         aSHR5 :: (Arrow a, B.Bits b) => Grid a b b
150         aSHR5 = (aId &&& (aConst (5 :: Int))) >>> aShiftR
151
152         aXorDeltaR :: (Arrow a) => Grid a Word32 Word32
153         aXorDeltaR = (aId &&& (aConst (delta r))) >>> aXor
154
155
156 -- key :: Key
157 -- key   = (65, 65, 65, 65) -- AAAA
```

```
158
159 -- testMessage = "Hello World!"
160
161 -- msg2vals :: String -> [Value]
162 -- msg2vals []          = []
163 -- msg2vals (s0:[])     = msg2vals [s0, '\0']
164 -- msg2vals (s0:s1:ses)
165 --   = (fromIntegral $ ord s0, fromIntegral $ ord s1) : (msg2vals ses)
166
167 -- vals2msg :: [Value] -> String
168 -- vals2msg [] = ""
169 -- vals2msg ((v0,v1):vs)
170 --            = (chr (fromIntegral v0): (chr (fromIntegral v1) : (vals2msg vs)))
171
172 -- aFeistel
173 --   = proc ((v0, v1), (k0, k1)) -> do
174 --       vsl <- aShl -< (v1, 4)
175 --       vsr <- aShr -< (v1, 5)
176
177 --       t1  <- aXor -< (k0, vsl)
178 --       t2  <- aXor -< (d,  v1)
179 --       t3  <- aXor -< (k1, vsr)
180
181 --       t4  <- aOr  -< (t1, t2)
182 --       t5  <- aOr  -< (t4, t3)
183
184 --       v0' <- aId  -< (v1)
185 --       v1' <- aXor -< (v0, t5)
186 --       returnA     -< (v0', v1')
187 --   where
188 --     aShl = aShiftL
189 --     aShr = aShiftR
190 --     d    = (delta 0)
191
192 -- aCycle
193 --   = proc ((v0, v1), ((k0, k1), (k2, k3))) -> do
194 --       (v0', v1')   <- aFeistel -< ((v0, v1),   (k0, k1))
195 --       (v0'', v1'') <- aFeistel -< ((v0', v1'), (k2, k3))
196 --       returnA                  -< (v0'', v1'')
197
198
199 aShiftL4_XorKey
200   =   first
201       (     aDup
```

```
202         >>> second (aConst 4)
203         >>> aShiftL
204       )
205     >>> aAdd
206     -- >>> aXor
207
208 aShiftR5_XorKey
209   =   first
210       (     aDup
211         >>> second (aConst 5)
212         >>> aShiftR
213       )
214     >>> aAdd
215     -- >>> aXor
216
217 aXorDelta
218     =   aId &&& (aConst 2654435769)
219     >>> aAdd
220     -- >>> aXor
221
222 aFeistelTest
223   = (aFst >>> aSnd) &&& (aFst >>> aFst) &&& (first aSnd)
224    >>> second
225        ( second
226          (   aDistr &&& aFst
227          >>> (aShiftL4_XorKey *** aShiftR5_XorKey) *** aXorDelta
228          >>> first aOr
229          >>> aOr
230          )
231        )
232    >>> second aAdd
233
234 aCycleTest
235   =    a_aBC2ABc
236    >>> first aFeistelTest
237    >>> aFeistelTest
```

A.6. Circuit.Auxillary

```haskell
module Circuit.Auxillary
where

import Data.List (union, groupBy, isInfixOf)
import Data.Maybe
import Data.Either
import Control.Monad (msum)

import GHC.Exts (sortWith)

import Circuit.Grid
import Circuit.Stream

import Control.Category
import Circuit.Arrow

import Circuit.Grid
import Circuit.Stream

import Circuit.Descriptor
import Circuit.Show
import Circuit.Tests
import Circuit.Splice
import Circuit.Sensors
import Circuit.Workers

import Circuit.Graphs (emptyCircuit)



-- 'nextID' is a function that gets a list of component numbers and
-- generates the next valid one
nextID :: [CompID] -> CompID
nextID []    = 0
nextID [cid] = cid + 1
nextID cids  = nextID [foldl max 0 cids]



-- Die Funktion \hsSource{onlyInnerEdges} filtert aus einer Liste von
-- Kanten genau diese Kanten heraus, die die internen Kanten im
```

```
42 -- Schaltkreis darstellen. Die Ergebnismenge enthaelt keine Ein- und
43 -- Ausgehenden Kanten.
44 onlyInnerEdges :: [Edge] -> [Edge]
45 onlyInnerEdges es = es'
46     where es' = filter notIO $ es
47           notIO :: Edge -> Bool
48           notIO (MkEdge (Nothing, _) _) = False
49           notIO (MkEdge _ (Nothing, _)) = False
50           notIO _                       = True
51
52
53
54 -- Typischerweise verwendet man den Begriff \begriff{Synthese} in der
55 -- Hardware-Community fuer den Prozess, aus einer Modellhaften
56 -- Hardwarebeschreibung heraus tatsaechlichen Hardwarecode
57 -- (beispielsweise VHDL) zu erzeugen. Daneben ist auch die
58 -- \begriff{Simulation} von Hardwaremodellen notwendig, um die
59 -- entworfenen Modelle vor der Realisierung ueberpruefen zu koennen.
60 --
61 --
62 -- Die beiden Prozesse lassen sich auch auf das \hsSource{Grid}-Arrow
63 -- Modell uebertragen. So stellt \hsSource{synthesize} eine Funktion
64 -- dar, die aus einem gegebenen \hsSource{Grid} die fertige
65 -- Hardwarebeschreibung \footnote{in diesem Fall ausgeliefert in VHDL}
66 -- ausliest. Die Simulation wird mittels der Funktion
67 -- \hsSource{simulate} abgebildet. Diese Funktion liest nun aus einem
68 -- \hsSource{Grid} den Arrow heraus und ueberfuehrt diesen in einen
69 -- \hsSource{Stream}-Arrow, der dann mit einem kontinuierlichem
70 -- Datenstrom simuliert werden kann.
71
72
73
74 testPreSynth :: (Arrow a) => Grid a b c -> (a b c, CircuitDescriptor)
75 testPreSynth (GR tuple) = tuple
76
77 test2PreSynt = snd
78
79 --synthesize :: (Arrow a) => Grid a b c -> CircuitDescriptor
80 --synthesize (GR (_, cd)) = flatten
81 --with Looping-Stuff ...
82 synthesize :: Grid (->) b c -> CircuitDescriptor
83 -- synthesize :: Grid (->) b c -> CircuitDescriptor
84 synthesize (GR (_, cd)) = cd
85 --synthesize (GR x) = snd x
```

```haskell


simulate :: Grid (->) b c -> Stream b c
simulate f = arr (toFunctionModel f)



-- Um einen \hsSource{Grid}-Arrow kombinatorisch auszuwerten,
-- existiert die Hilfsfunktion \hsSource{toFunctionModel}, die ein
-- Synonym fuer \hsSource{runGrid} ist.


toFunctionModel :: Grid (->) b c -> (b -> c)
toFunctionModel = runGrid



-- Weitere Hilfsfunktionen werden notwendig, um schon bestehende
-- \hsSource{Grid}-Arrows mit Schaltkreis Beschreibungen anzureichern.


insert :: b -> (a, b) -> (a, b)
insert sg ~(x, _) = (x, sg)

insEmpty
  = insert emptyCircuit
    { nodeDesc =
      MkNode { label   = "eeeempty"
             , nodeId  = 0
             , sinks   = mkPins 1
             , sources = mkPins 3
             }
    }

augment :: (Arrow a) => CircuitDescriptor -> a b c -> Grid a b c
augment cd_f f = GR (f, cd_f)
```

A.7. Circuit.Defaults

```haskell
module Circuit.Defaults where

```

```haskell
import Control.Category
import Prelude hiding (id, (.))
import qualified Data.Bits as B -- (shiftL, shiftR, xor, (.&.))

import Circuit

import Circuit.Grid

import Circuit.Arrow

import Circuit.Auxillary
import Circuit.Descriptor
import Circuit.Graphs
import Circuit.Show


type KeyChunk = Int
type ValChunk = Int
type Key    = (KeyChunk, KeyChunk, KeyChunk, KeyChunk)
type KeyHalf = (KeyChunk, KeyChunk)
type Value = (ValChunk, ValChunk)

-- xor :: Bool -> Bool -> Bool
-- xor x y | x == True && y == False = True
--         | x == False && y == True = True
--         | otherwise = False

oneNodeCircuit :: String -> CircuitDescriptor
oneNodeCircuit s = emptyCircuit { nodeDesc = emptyNodeDesc { label = s } }

aId :: (Arrow a) => Grid a b b
aId
    = augment
        emptyCircuit
          { nodeDesc = emptyNodeDesc
            { label   = "ID"
            , sinks   = mkPins 1
            , sources = mkPins 1
            }
          , cycles  = 1
          , space   = 1
          }
    $ arr id

```

```
47
48 aConst :: (Arrow a, Show b) => b -> Grid a c b
49 aConst x
50     = augment
51         emptyCircuit
52           { nodeDesc = emptyNodeDesc
53             { label = "CONST_" ++ (show x)
54             , sinks   = mkPins 1 -- a sink is needed for the
55                                  -- rewire-function to work properly
56             , sources = mkPins 1
57             }
58           , cycles  = 0
59           , space   = 1
60           }
61     $ arr (const x)
62
63
64 (.&.) :: Bool -> Bool -> Bool
65 True .&. True = True
66 _    .&. _    = False
67
68 (.|.) :: Bool -> Bool -> Bool
69 False .|. False = False
70 _     .|. _     = True
71
72 xor :: Bool -> Bool -> Bool
73 xor True False = True
74 xor False True = True
75 xor _     _    = False
76
77 aAnd :: (Arrow a) => Grid a (Bool, Bool) (Bool)
78 aAnd
79     = augment
80         emptyCircuit
81           { nodeDesc = emptyNodeDesc
82             { label   = "AND"
83             , sinks   = mkPins 2
84             , sources = mkPins 1
85             }
86           , cycles  = 1
87           , space   = 4
88           }
89     $ arr (uncurry (.&.))
90
```

```haskell

aOr :: (Arrow a) => Grid a (Bool, Bool) (Bool)
aOr
    = augment
        emptyCircuit
          { nodeDesc = emptyNodeDesc
            { label   = "OR"
            , sinks   = mkPins 2
            , sources = mkPins 1
            }
          , cycles  = 1
          , space   = 4
          }
    $ arr (uncurry (.|.))


aNot :: (Arrow a) => Grid a (Bool) (Bool)
aNot
    = augment
        emptyCircuit
          { nodeDesc = emptyNodeDesc
            { label   = "NOT"
            , sinks   = mkPins 1
            , sources = mkPins 1
            }
          , cycles  = 1
          , space   = 2
          }
    $ arr (not)


aBXor :: (Arrow a, B.Bits b) => Grid a (b, b) (b)
aBXor
    = augment
        emptyCircuit
          { nodeDesc = emptyNodeDesc
            { label   = "XOR"
            , sinks   = mkPins 2
            , sources = mkPins 1
            }
          , cycles  = 1
          , space   = 4
          }
    $ arr (uncurry B.xor)
```

```
135
136 aXor :: (Arrow a) => Grid a (Bool, Bool) (Bool)
137 aXor
138     = augment
139         emptyCircuit
140           { nodeDesc = emptyNodeDesc
141             { label   = "XOR"
142             , sinks   = mkPins 2
143             , sources = mkPins 1
144             }
145           , cycles  = 1
146           , space   = 4
147           }
148     $ arr (uncurry xor)
149
150
151 -- aFst :: (Arrow a, Bits b) => Grid a (b, c) (b)
152 aFst :: (Arrow a) => Grid a (b, c) (b)
153 aFst
154     = augment
155         emptyCircuit
156           { nodeDesc = emptyNodeDesc
157             { label   = "FST"
158             , sinks   = mkPins 2
159             , sources = mkPins 1
160             }
161           , cycles  = 1
162           , space   = 4
163           }
164     $ arr (fst)
165
166
167 aSnd :: (Arrow a) => Grid a (b, c) (c)
168 aSnd
169     = augment
170         emptyCircuit
171           { nodeDesc = emptyNodeDesc
172             { label   = "SND"
173             , sinks   = mkPins 2
174             , sources = mkPins 1
175             }
176           , cycles  = 1
177           , space   = 4
178           }
```

```
179     $ arr (snd)
180
181
182 aShiftL :: (Arrow a, B.Bits b) => Grid a (b, Int) (b)
183 aShiftL
184     = augment
185         emptyCircuit
186           { nodeDesc = emptyNodeDesc
187             { label   = "SHIFTL"
188             , sinks   = mkPins 2
189             , sources = mkPins 1
190             }
191           , cycles  = 1
192           , space   = 6
193           }
194     $ arr (uncurry B.shiftL)
195
196 aShiftR :: (Arrow a, B.Bits b) => Grid a (b, Int) (b)
197 aShiftR
198     = augment
199         emptyCircuit
200           { nodeDesc = emptyNodeDesc
201             { label   = "SHIFTR"
202             , sinks   = mkPins 2
203             , sources = mkPins 1
204             }
205           , cycles  = 1
206           , space   = 6
207           }
208     $ arr (uncurry B.shiftR)
209
210 aAdd :: (Arrow a, Num b) => Grid a  (b, b) (b)
211 aAdd
212     = augment
213         emptyCircuit
214           { nodeDesc = emptyNodeDesc
215             { label   = "ADD"
216             , sinks   = mkPins 2
217             , sources = mkPins 1
218             }
219           , cycles  = 1
220           , space   = 4
221           }
222     $ arr (uncurry (+))
```

```
223
224 aFlip :: (Arrow a) => Grid a (b, c) (c, b)
225 aFlip
226     = augment
227         emptyCircuit
228           { nodeDesc = emptyNodeDesc
229             { label   = "FLIP"
230             , sinks   = mkPins 2
231             , sources = mkPins 2
232             }
233           , cycles  = 1
234           , space   = 4
235           }
236     $ arr (\(x, y) -> (y, x))
237
238 aSwapSnd :: (Arrow a) => Grid a ((b, c), d) ((b, d), c)
239 aSwapSnd
240     = augment
241         emptyCircuit
242           { nodeDesc = emptyNodeDesc
243             { label   = "SWPSND"
244             , sinks   = mkPins 2
245             , sources = mkPins 2
246             }
247           , cycles  = 1
248           , space   = 6
249           }
250     $ arr (\((x, y), z) -> ((x, z), y))
251
252 aAssocRight = a_ABc2aBC
253 aAssocLeft  = a_aBC2ABc
254
255 a_ABc2aBC :: (Arrow a) => Grid a ((b, c), d) (b, (c, d))
256 a_ABc2aBC
257     = augment
258         emptyCircuit
259           { nodeDesc = emptyNodeDesc
260             { label   = "ABc2aBC"
261             , sinks   = mkPins 2
262             , sources = mkPins 2
263             }
264           , cycles  = 1
265           , space   = 6
266           }
```

```
267     $ arr (\((x, y), z) -> (x, (y, z)))
268
269 a_aBC2ABc :: (Arrow a) => Grid a (b, (c, d)) ((b, c), d)
270 a_aBC2ABc
271     = augment
272         emptyCircuit
273           { nodeDesc = emptyNodeDesc
274             { label   = "aBC2ABc"
275             , sinks   = mkPins 2
276             , sources = mkPins 2
277             }
278           , cycles  = 1
279           , space   = 6
280           }
281     $ arr (\(x, (y, z)) -> ((x, y), z))
282
283
284 -- aDistr :: (Arrow a, Bits b, Bits c, Bits d)
285 --        => Grid a (b, (c, d)) ((b, c), (b, d))
286 aDistr :: (Arrow a) => Grid a (b, (c, d)) ((b, c), (b, d))
287 aDistr
288     =   aDup
289     >>> second aFst *** second aSnd
290
291 -- |'aDdistr' is the reverse operation to the Distr operation
292 aDdistr :: (Arrow a) => Grid a ((b, c), (d, e)) ((b, d), (c, e))
293 aDdistr
294     =   aSwapSnd
295     >>> a_aBC2ABc *** aId
296     >>> a_ABc2aBC
297     >>> aId *** aFlip
298
299 aShiftL4 :: (Arrow a, B.Bits b) => Grid a b b
300 aShiftL4
301     = augment
302         emptyCircuit
303           { nodeDesc = emptyNodeDesc
304             { label   = "SHIFTL4"
305             , sinks   = mkPins 1
306             , sources = mkPins 1
307             }
308           , cycles  = 1
309           , space   = 6
310           }
```

```
311     $ arr (flip B.shiftL 4)
312
313 aShiftR5 :: (Arrow a, B.Bits b) => Grid a b b
314 aShiftR5
315     = augment
316         emptyCircuit
317           { nodeDesc = emptyNodeDesc
318             { label   = "SHIFTR5"
319             , sinks   = mkPins 1
320             , sources = mkPins 1
321             }
322           , cycles  = 1
323           , space   = 6
324           }
325     $ arr (flip B.shiftR 5)
326
327 aXorMagic
328     = augment
329         emptyCircuit
330           { nodeDesc = emptyNodeDesc
331             { label   = "ADDMAGIC"
332             , sinks   = mkPins 1
333             , sources = mkPins 1
334             }
335           , cycles  = 1
336           , space   = 4
337           }
338     $ arr (\x -> (x, 2654435769)) >>> aBXor
339
340 aDup
341     = augment
342         emptyCircuit
343           { nodeDesc = emptyNodeDesc
344             { label   = "DUP"
345             , sinks   = mkPins 1
346             , sources = mkPins 2
347             }
348           , cycles  = 1
349           , space   = 4
350           }
351     $ arr (\(x) -> (x, x))
352
353 aRegister :: (Arrow a) => Grid a b b
354 aRegister
```

```
355     = augment
356         ( mkRegister $ emptyNodeDesc
357                        { sinks   = mkPins 1
358                        , sources = mkPins 1
359                        }
360         )
361     $ arr id
```

A.8. Circuit.Descriptor

```
1  module Circuit.Descriptor
2  where
3
4
5  -- \subsection{Typen mit neuen Namen}
6  -- Zunaechst werden grundlegende Typaliasse vergeben, die im gesamten
7  -- Quelltext Anwendung finden.
8
9
10 -- Jede Hardwarekomponente besitzt Pins verschiedener Art, darunter
11 -- ein- und ausgehende. Diese sind durchnummeriert und werden ueber
12 -- ihre Nummer, die als Integer abgebildet wird, identifiziert. Fuer
13 -- die Komponenten ID gilt dies ebenfalls.
14 --
15 --
16 -- Daneben wird der Typ fuer einen \begriff{Clock-Cycle} definiert. In
17 -- diesem Fall wird ein Integer dafuer verwendet. Auch die Flaeche wird
18 -- mit einem Integer beschrieben. Fuer die Flaeche besagt der Wert,
19 -- wieviele \begriff{Zellen} der Baustein belegt.
20
21 type PinID = Int
22 type Pins  = [PinID]
23
24 type ID     = Int
25 type CompID = ID
26
27 type Tick   = Int
28 type Area   = Int
29
30
```

```
31
32 -- Eine Kante ist eine Verbindung zwischen zwei \hsSource{Pins}, dabei
33 -- muessen die beiden \hsSource{Pins} nicht unbedingt zu
34 -- unterschiedlichen Komponenten gehoeren \footnote{Beispielsweise bei
35 -- zyklischen Schaltungen}. Das Tupel aus \hsSource{PinID} und
36 -- \hsSource{CompID} identifiziert einen Pin. Da auch Kanten
37 -- abgebildet werden sollen, die nach ausserhalb der aktuellen
38 -- Komponente gehen sollen, ist es notwendig, die \hsSource{CompID}
39 -- als \hsSource{Maybe} Typ zu beschreiben. Das Tupel wird im
40 -- folgenden mit \hsSource{Anchor} bezeichnet.
41
42 type Anchor       = (Maybe CompID, PinID)
43 type SinkAnchor   = Anchor
44 type SourceAnchor = Anchor
45
46
47 -- Ausserdem wurden noch zwei weitere Aliasse vergeben, die eingehende
48 -- (\hsSource{SinkAnchor}) und ausgehende (\hsSource{SourceAnchor})
49 -- Pins voneinander unterscheiden.
50 --
51 -- \subsection{Benannte Typen} Um die Graph-Struktur spaeter in
52 -- Sourcecode ueberfuehren zu koennen, benoetigt man Namen fuer
53 -- \hsSource{Anchor} und \hsSource{Edges} sowie Lookuptabellen, in
54 -- denen dann die Namen abgelegt werden. Die Typaliasse hierfuer werden
55 -- hier direkt definiert.
56
57 type NamedPins = [(String, Anchor)]
58 type NamedSigs = [(String, Edge)]
59 type NamedSnks = NamedPins
60 type NamedSrcs = NamedPins
61 type NamedIOs  = (NamedSnks, NamedSrcs)
62
63 nameSig = "i"
64 nameExI = "inc"
65 nameExO = "out"
66 nameInI = "e"
67 nameInO = "a"
68
69
70
71 -- \subsection{Schaltungsbeschreibung} Jede Komponente die durch einen
72 -- Arrow repraesentiert wird, hat zusaetzliche Attribute, die nicht in
73 -- dem Arrow selber stehen. Diese Attribute werden auch nicht fuer alle
74 -- Arrowklassen Instanzen benoetigt. Daher sind diese lose an den
```

```
75  -- Arrow gekoppelt. \footnote{Arrow und Attribute werden in einem
76  -- Tupel zusammengefasst}
77
78
79  -- \subsection{Pins} Zunaechst werden grundlegende Typaliasse vergeben,
80  -- die im gesamten Quelltext Anwendung finden.
81
82  -- Jede Hardware Komponente besitzt Pins verschiedener Art, darunter
83  -- Eingabe- und Ausgabepins.
84
85  -- \subsection{Schaltungsbeschreibung} Jede Komponente die durch einen
86  -- Arrow repraesentiert wird, hat zusaetzliche Attribute, die nicht in
87  -- dem Arrow selber stehen. Diese Attribute werden auch nicht fuer alle
88  -- Arrowklassen Instanzen benoetigt. Daher sind diese lose an den
89  -- Arrow gekoppelt. \footnote{Arrow und Attribute werden in einem
90  -- Tupel zusammengefasst}
91
92
93  -- Hier wird unterschieden zwischen kombinatorischen Schaltungen
94  -- \hsSource{MkCombinatorial}, Registern \hsSource{MkRegister} und
95  -- nicht vorhandenen Schaltungen \hsSource{NoDescriptor}.
96
97
98  -- Ein Register unterscheidet sich von einer kombinatorischen
99  -- Schaltung, teilweise gibt es Gemeinsamkeiten. Diese Gemeinsamkeiten
100 -- werden ueber den Datentyp \hsSource{NodeDescriptor} verkoerpert.
101
102
103
104 data NodeDescriptor
105     = MkNode
106       { label   :: String
107       , nodeId  :: ID
108       , sinks   :: Pins
109       , sources :: Pins
110       }
111     deriving (Eq)
112
113
114
115
116 -- Der \hsSource{NodeDescriptor} taucht in der Definition eines
117 -- Schaltkreises, aber auch in der Definition eines Registers, auf.
118
```

```
119
120 data CircuitDescriptor
121     = MkCombinatorial
122       { nodeDesc  :: NodeDescriptor
123       , nodes     :: [CircuitDescriptor]
124       , edges     :: [Edge]
125       , cycles    :: Tick
126       , space     :: Area
127       }
128
129     | MkRegister
130       { nodeDesc  :: NodeDescriptor
131       , bits      :: Int
132       }
133
134     | MkLoop
135       { nodeDesc  :: NodeDescriptor
136       , nodes     :: [CircuitDescriptor]
137       , edges     :: [Edge]
138       , space     :: Area
139       }
140
141     | MkComposite
142       { composite :: [CircuitDescriptor]
143       }
144
145     | NoDescriptor
146     deriving (Eq)
147
148 type Netlist = CircuitDescriptor
149
150
151 -- In Haskell lassen sich die Komponentenattribute ueber einen
152 -- Summentyp abgebildet. Dieser Datentyp ist ein fundamentaler
153 -- Datentyp, da er gleichberechtigt neben der eigentlichen
154 -- Funktionalitaet steht. Er besitzt eine Bezeichnung \hsSource{label},
155 -- sowie eine eindeutige ID \hsSource{nodeId}. Zusaetzlich sind die
156 -- ein- sowie die ausgehenden Pins aufgefuehrt und auch die
157 -- verbindenden Kanten (\hsSource{edges}).
158 --
159 --
160 -- Jede Komponente kann durch untergeordnete Komponenten beschrieben
161 -- werden. Dies wird im Datentyp ueber die \hsSource{nodes} Liste
162 -- abgebildet.  Ist die Komponenten atomar, so enthaelt dieses Datum
```

```haskell
-- die leere Liste. Der Konstruktor \hsSource{NoSG} ist analog zu
-- $\varnothing$.
--
-- Um zu verhindern, dass ungueltige Schaltungsbeschreibungen erzeugt
-- werden, koennen \begriff{smart constructor}s eingesetzt
-- werden. Hierbei handelt es sich um Funktionen, die analog zu den
-- Konstruktoren arbeiten. Diese unterscheiden sich darin, dass die
-- Konstruktoren jedes Datum erzeugen. Allerdings ist es haeufig nicht
-- gewuenscht, jedes Datum erzeugen zu koennen oder es ist gewuenscht,
-- dass der Benutzer beim Versuch ein falsches Datum zu erzeugen, mit
-- einer Fehlermeldung konfrontiert wird.
--
--
-- Es folgt der Quellcode des \begriff{smart constructor}s fuer ein
-- Register. Hierbei ist nur darauf zu achten, dass dieses Register
-- keine ID bekommt, die schon einmal vergeben wurde.
--


mkRegister :: NodeDescriptor -> CircuitDescriptor
mkRegister nd
    = MkRegister
      { nodeDesc = nd { label = "REG" ++ (show $ nodeId nd) }
      , bits     = length $ sinks nd
      }



-- Zuletzt fehlt jetzt lediglich die Definition einer Kante. Eine
-- Kante kennt einen Quellpin sowie einen Zielpin.

data Edge
    = MkEdge { sourceInfo :: SourceAnchor
             , sinkInfo   :: SinkAnchor
             }
    deriving (Eq)



-- \subsection{Smarte Konstruktoren} In Haskell lassen sich Typen sehr
-- fein granular definieren. Fuer die Liste der Pins einer Schaltung
-- passt eine Liste von Integerwerten sehr gut aber nicht perfekt. So
-- gibt es auch Integerlisten mit keinem Inhalt. Wollte man dies auf
-- der Typebene verhindern, so wuerde dies einen Overhead erfordern,
```

```
207 -- der nicht im Verhaeltnis zum Nutzen stehen wuerde. Eine einfache
208 -- aber nuetzliche Alternative sind \begriff{smart
209 -- constructor}. Hierbei handelt es sich um einen Funktion, die als
210 -- Parameter alle Werte bekommt, die benoetigt wird, um ein Datum des
211 -- gewuenschten Types zu erzeugen. Die Funktion erzeugt dann ein
212 -- solches Datum und kann sich daneben auch noch um
213 -- Fehlerbehandlungen kuemmern.
214 --
215 -- Bei \hsSource{mkPins} handelt es sich um so einen \begriff{smart
216 -- Constructor}. Dieser erhaelt die Anzahl der benoetigten Pins als
217 -- Parameter und erzeugt  eine entsprechende Liste.
218
219
220 mkPins :: Int -> Pins
221 mkPins 0 = error $ show "It is not possible to generate a component with 0 pins"
222 mkPins n = [0..n-1]
```

A.9. Circuit.EdgeTransit

```
1  module Circuit.EdgeTransit
2  where
3
4
5
6  -- Im spaeteren wird die Funktionen aus dem
7  -- \ref{mod:Circuit.Descriptor} Modul benoetigt, sowie die Funktion
8  -- \hsSource{isJust} aus \hsSource{Data.Maybe}.
9
10 import Data.Maybe (isJust)
11 import Circuit.Descriptor
12
13
14 -- Die Zwischenstruktur \hsSource{NamedEdge} wird ueber ein Typalias
15 -- definiert. Hierbei wird einem Namen eine Liste von Ankerpunkten
16 -- zugeordnet.
17
18 type NamedEdge = ([Anchor], String)
19
20 -- Die Funktion \hsSource{generateNamedEdges} filtert aus dem
21 -- uebergebenen \hsSource{CircuitDescriptor} die Kanten heraus. Von
```

```
22 -- allen Kanten werden die Kanten heraus gefiltert, die nach
23 -- \begriff{aussen} verbinden. Diese Kanten haben als Quell- oder
24 -- Zielkomponente \hsSource{Nothing} gesetzt. Die relevanten Kanten
25 -- werden durchnummeriert und in der Form einer \hsSource{NamedEdge}
26 -- zurueckgeliefert.
27
28 pre :: String
29 pre = nameSig
30
31 generateNamedEdges :: CircuitDescriptor -> [NamedEdge]
32 generateNamedEdges g
33     = map (\(i, e) -> (sourceInfo e : sinkInfo e : [], pre ++ show i))
34         $ zip [0..] relevantEdges
35         where relevantEdges = filter (\  (MkEdge (ci,_) (co,_))
36                                       -> isJust ci && isJust co)
37                             $ edges g
38
39
40
41 -- \hsSource{getAllEdgeNames} filtert lediglich das zweite Datum aus
42 -- dem Tupel heraus.
43
44 getAllEdgeNames :: [NamedEdge] -> [String]
45 getAllEdgeNames = map snd
46
47
48
49 -- Zum Ermitteln des Namens einer Kante, die an einem bestimmten
50 -- Ankerpunkt beginnt oder endet, wird eine Funktion
51 -- \hsSource{getNamedEdge} definiert. Hierzu wird ueberprueft, ob der
52 -- uebergebene Ankerpunkt in einer der benannten Kanten vorkommt. Ist
53 -- dies der Fall, wird das erste Element der Ergebnisliste
54 -- zurueckgeliefert.
55
56 getNamedEdge :: [NamedEdge] -> Anchor -> NamedEdge
57 getNamedEdge nedgs ap
58       = head
59       $ filter (\(aps, _) -> ap `elem` aps)
60       $ nedgs
61
62
63
64 -- Um nur den Namen einer Kante zu ermitteln, die einen bestimmten
65 -- Ankerpunkt verbindet, wird die Funktion \hsSource{getEdgeName}
```

```
66 -- definiert.  Diese filtert den zweiten Wert des Ergebnisstupels von
67 -- \hsSource{getNamedEdge} heraus und liefert ihn als Rueckgabe.
68
69 getEdgeName :: [NamedEdge] -> Anchor -> String
70 getEdgeName nedgs ap
71       = snd $ getNamedEdge nedgs ap
```

A.10. Circuit.Graphs

```
1  module Circuit.Graphs
2  where
3
4
5
6  -- Verwendet wird ausschliesslich das Modul \ref{mod:Circuit.Graphs}
7  -- (\hsSource{Circuit.Descriptor}).
8
9
10
11 import Circuit.Descriptor
12
13
14 -- \subsection{Leerer Schaltkreis}
15 -- Zunaechst wird ein leerer Schaltkreis definiert, also ein Datum des
16 -- Typs \hsSource{CircuitDescriptor}. Dieses Datum enthaelt zwar die
17 -- korrekte Struktur, aber keine Nutzdaten. \hsSource{String}s sind
18 -- leer \footnote{der \hsSource{String} ``...'' ist fuer das Debugging
19 -- bewusst nicht leer}, \hsSource{Integer} werden auf $0$ gesetzt und
20 -- Listen sind jeweils leere Listen.
21 --
22 --
23 -- Da ein \hsSource{CircuitDescriptor} auch aus einem
24 -- \hsSource{NodeDescriptor} aufgebaut ist, liegt es auf der Hand,
25 -- auch einen leeren \hsSource{NodeDescriptor} zu definieren.
26
27
28
29 emptyNodeDesc :: NodeDescriptor
30 emptyNodeDesc
31     = MkNode { label    = "..."
```

```
32              , nodeId  = 0
33              , sinks   = []
34              , sources = []
35              }
36
37
38
39
40  -- Der leere Schaltkreis laesst sich jetzt ueber den leeren
41  -- \hsSource{NodeDescriptor} beschreiben.
42
43
44  emptyCircuit :: CircuitDescriptor
45  emptyCircuit
46      = MkCombinatorial
47        { nodeDesc = emptyNodeDesc
48        , nodes    = []
49        , edges    = []
50        , cycles   = 0
51        , space    = 0
52        }
53
54
55  -- \subsection{Schaltkreis Modifikatoren}
56  -- Neben der leeren Schaltung werden weitere Schaltungen benoetigt, die
57  -- groesstenteils Unterschiede im Namen und in der Anzahl der Ein- und
58  -- Ausgaenge haben. Diese koennte man definieren, indem die
59  -- Beschreibung der leeren Schaltung an den entsprechenden Stellen
60  -- ueberschrieben wird.  Schoener ist es allerdings, wenn man hier auf
61  -- Modifikatoren zurueckgreifen kann, welche die Aufgabe
62  -- uebernehmen. Bei Aenderungen an der \hsSource{CircuitDescriptor}
63  -- Struktur muessen dann nur die Modifikatoren veraendert werden, nicht
64  -- saemtliche Schaltungsdefinitionen.
65
66
67
68  withLabel :: String -> CircuitDescriptor -> CircuitDescriptor
69  withLabel l cd   = cd { nodeDesc = nd { label = l } }
70      where nd = nodeDesc cd
71
72  sinkCount :: Int -> CircuitDescriptor -> CircuitDescriptor
73  sinkCount i cd   = cd { nodeDesc = nd { sinks = [0..(i-1)] } }
74      where nd = nodeDesc cd
75
```

```haskell
sourceCount :: Int -> CircuitDescriptor -> CircuitDescriptor
sourceCount i cd = cd { nodeDesc = nd { sources = [0..(i-1)] } }
    where nd = nodeDesc cd



-- Mithilfe der Modifikatoren lassen sich nun die weiteren
-- Schaltkreise definieren:


arrCircuit
    = withLabel "-ARR-" . sinkCount 1 . sourceCount 1 $ emptyCircuit

throughCircuit
    = withLabel "(-)"   . sinkCount 1 . sourceCount 1 $ emptyCircuit

idCircuit
    = withLabel "-ID-"  . sinkCount 1 . sourceCount 1 $ emptyCircuit

leftCircuit
    = withLabel "(L)"   $ emptyCircuit

rightCircuit
    = withLabel "(R)"   $ emptyCircuit
```

A.11. Circuit.PinTransit

```haskell
module Circuit.PinTransit
where




-- Benoetigt werden Moduldefinitionen aus \hsSource{Circuit}.


import Circuit.Descriptor


-- \subsection{Datenstruktur}
-- Zunaechst werden Typaliasse angelegt, welche die verwendeten
```

```haskell
-- Datentypen nach ihrer Aufgabe benennen.


type NamedPin  = (PinID, String)
type InNames   = [NamedPin]
type OutNames  = [NamedPin]

type NamedComp = (CompID, (InNames, OutNames))



-- \subsection{Funktionen}

-- Um die interne Datenstruktur des algebraischen Datentypes bedienen
-- zu koennen, werden folgende Funktionen definiert:

-- Die Funktion \hsSource{generateNamedComps} erzeugt die benannten
-- Komponentenliste. Hier werden nur die Pins benannt. Ein Pin kann
-- immer von zwei Seiten aus gesehen werden. Es gibt \begriff{externe}
-- Pins, also Pins, an die von aussen ein Draht angeschlossen
-- werden kann. Diese werden unter dem Namen
-- \hsSource{generateSuperNames} zusammengefasst. Daneben existieren
-- auch \begriff{interne} Pins, die man hinter dem Namen
-- \hsSource{generateSubNames} findet. Diese Unterscheidung muss
-- getroffen werden, da externe Pins mit den Praefixen
-- \hsSource{nameExI} und \hsSource{nameEx0} versehen werden. Interne
-- Pins werden mit \hsSource{nameInI} sowie \hsSource{NameIn0} benamt.
-- \hsSource{generateSuperNames} wird auf den uebergebenen
-- \hsSource{CircuitDescriptor} angewandt, die Funktion
-- \hsSource{generateSubNames} hingegen auf alle dem
-- \hsSource{CircuitDescriptor} untergeordneten
-- \hsSource{CircuitDescriptor}en.


generateNamedComps :: CircuitDescriptor -> [NamedComp]
generateNamedComps g = generateSuperNames g : (map generateSubNames $ nodes g)
      where generateSuperNames g = ( (nodeId.nodeDesc) g
                                   , ( namePins (sinks.nodeDesc)   nameExI g
                                     , namePins (sources.nodeDesc) nameEx0 g
                                     )
                                   )
            generateSubNames g   = ( (nodeId.nodeDesc) g
                                   , ( namePins (sinks.nodeDesc)   nameInI g
                                     , namePins (sources.nodeDesc) nameIn0 g
```

```
59                                      )
60                                     )
61
62  -- Die beiden Funktionen \hsSource{getNamedInPins} sowie
63  -- \hsSource{getInPinNames} holen jeweils aus einer Liste von
64  -- \hsSource{NamedComp} und einer \hsSource{CompID} den passenden
65  -- Datensatz heraus. Hierbei unterscheiden sie sich lediglich im
66  -- Rueckgabetyp voneinander. So liefert \hsSource{getNamedInPins} die
67  -- \hsSource{InNames}, also eine Liste benannter Pins,
68  -- zurueck. \hsSource{getInPinNames} liefert nur eine Liste der Namen.
69
70
71  getNamedInPins :: [NamedComp] -> CompID -> InNames
72  getNamedInPins = getPinNames fst
73
74  getInPinNames :: [NamedComp] -> CompID -> [String]
75  getInPinNames cname cid = map snd $ getNamedInPins cname cid
76
77
78
79
80  -- Die beiden naechsten Funktionen, \hsSource{getNamedOutPins} sowie
81  -- \hsSource{getOutPinNames} verhalten sich analog zu den beiden
82  -- \begriff{InPin} Varianten, mit dem Unterschied, dass diese beiden
83  -- Funktionen sich auf die Ausgabepins beziehen.
84
85
86  getNamedOutPins :: [NamedComp] -> CompID -> OutNames
87  getNamedOutPins = getPinNames snd
88
89  getOutPinNames :: [NamedComp] -> CompID -> [String]
90  getOutPinNames cname cid = map snd $ getNamedOutPins cname cid
91
92
93
94
95  -- Mit der Funktion \hsSource{getPinNames} wird die eigentliche Logik
96  -- beschrieben, die fuer das Herausfiltern der Pinnamen notwendig
97  -- ist. Aus der uebergebenen Liste der benannten Komponenten werden all
98  -- die Komponenten herausgefiltert, die der ebenfalls uebergebenen
99  -- Komponenten ID entsprechen. Im naechsten Schritt (\hsSource{map
100 -- snd}) wird die Komponenten ID verworfen. Die uebergeben Funktion
101 -- \hsSource{f} definiert nun, ob das Ergebnis eingehende oder
102 -- ausgehende Pins sind. Der letzte Schritt entfernt ueberfluessige
```

```haskell
-- Listenverschachtelungen.


getPinNames ::   (([NamedPin], [NamedPin]) -> [NamedPin])
              ->  [NamedComp] -> CompID -> [(PinID, String)]
getPinNames f cname cid
    = concat
    $ map f
    $ map snd
    $ filter (\(x, _) -> x == cid)
    $ cname




-- Die folgenden fuenf Funktionen arbeiten analog zu den vorhergehenden
-- fuenf. Sie unterscheiden sich darin, dass sie einen weiteren
-- Parameter erwarten, ein \hsSource{PinID}. Dieser Parameter schraenkt
-- die Ergebnismenge auf exakt ein Ergebnis ein, sie liefert also
-- einen benannten Pin.

getNamedInPin :: [NamedComp] -> CompID -> PinID -> NamedPin
getNamedInPin = getPinName getNamedInPins

getNamedOutPin :: [NamedComp] -> CompID -> PinID -> NamedPin
getNamedOutPin = getPinName getNamedOutPins

getInPinName :: [NamedComp] -> CompID -> PinID -> String
getInPinName cname cid pid = snd $ getNamedInPin cname cid pid

getOutPinName :: [NamedComp] -> CompID -> PinID -> String
getOutPinName cname cid pid = snd $ getNamedOutPin cname cid pid

getPinName ::   ([NamedComp] -> CompID -> [NamedPin])
             -> [NamedComp] -> CompID -> PinID -> NamedPin
getPinName f cname cid pid
    = head
    $ filter (\(x, _) -> x == pid)
    $ f cname cid




-- Die letzte Funktion in diesem Modul erzeugt aus einem
```

```
147 -- \hsSource{CircuitDescriptor} und einem Praefix eine Liste benannter
148 -- Pins. Diese Funktion wird in \hsSource{generateNamedComps}
149 -- verwendet, um interne und externe Pins mit Namen zu versehen. Je
150 -- nach uebergebenen \hsSource{f} produziert \hsSource{namePins}
151 -- benannte interne oder benannte externe Pinlisten.
152
153 namePins ::   (CircuitDescriptor -> Pins)
154             -> String -> CircuitDescriptor -> [NamedPin]
155 namePins f pre g
156     = map (\x -> (x, pre ++ (show x))) $ f g
```

A.12. Circuit.Sensors

```
1  module Circuit.Sensors
2  where
3
4
5
6  -- Es werden lediglich die Standarddefinitionen, sowie die Tests
7  -- benoetigt.
8
9
10 import Circuit.Descriptor
11 import Circuit.Tests
12
13
14
15 -- \subsection{Schaltungssensoren}
16 -- Die Funktion \hsSource{allCircuits} holt aus einer Schaltung alle
17 -- Bausteine heraus, aus denen diese Schaltung aufgebaut ist. Das
18 -- Ergebnis wird als Liste von Bausteinen zurueckgegeben.
19
20
21 allCircuits :: CircuitDescriptor -> [CircuitDescriptor]
22 allCircuits sg
23     = if (length next_sg == 0) then sg : []
24                                else sg : (concat $ map allCircuits next_sg)
25     where next_sg = nodes sg
26           cid     = nodeId.nodeDesc $ sg
27
28
```

```haskell


-- Mithilfe der Funktion \hsSource{maxCompID} laesst sich die maximale
-- Komponentennummer einer Schaltung anzeigen.


maxCompID :: CircuitDescriptor -> CompID
maxCompID sg
  = (nodeId.nodeDesc $ sg) `max` (foldl max 0 $ map maxCompID (nodes sg))




-- Mit der Funktion \hsSource{getComp} laesst sich ein definierter
-- Baustein aus einer Schaltung auslesen. Identifiziert wird der
-- auszulesende Bausteine ueber seine
-- Komponentennummer. \hsSource{getComp} ist hier die Funktion mit
-- Fehlerbehandlung, die Logik selber steckt in \hsSource{getComp'}.


getComp :: CircuitDescriptor -> CompID -> CircuitDescriptor
getComp g cid = if length output == 1
                then head output
                else error "getComp: there is no such circuit"
    where output = getComp' g cid

getComp' :: CircuitDescriptor -> CompID -> [CircuitDescriptor]
getComp' g cid
    | (nodeId.nodeDesc $ g) == cid
    = [g]
    | otherwise
    = concat $ map (flip getComp' cid) (nodes g)




-- Um den uebergeordneten Schaltkreis eines Bausteins zu erhalten, kann
-- man sich der Funktion \hsSource{superNode} bedienen. Diese Funktion
-- erwartet eine Schaltung sowie eine Komponentennummer. Sofern
-- es eine SuperNode \footnote{also eine Schaltung, die mindestens aus
-- der Komponente mit der uebergebenen Nummer besteht} gibt, wird diese
-- zurueckgeliefert. Andernfalls wird eine Fehlermeldung ausgegeben.


```

```
73  superNode :: CircuitDescriptor -> CompID -> CircuitDescriptor
74  superNode g cid
75      = if length output == 1
76          then head output
77          else error "superNode: there is no such supernode"
78      where output = superNode' g cid
79
80  superNode' :: CircuitDescriptor -> CompID -> [CircuitDescriptor]
81  superNode' g cid
82      | g `isSuperNodeOf` cid
83      = [g]
84      | otherwise
85      = concat $ map (flip superNode' cid) $ nodes g
86
87
88
89
90  -- Als atomar werden Schaltungen bezeichnet, die selbst aus keinen
91  -- weiteren Schaltungen aufgebaut sind. Diese Schaltungen koennen auch
92  -- \begriff{Bausteine} genannt werden. Die Funktion
93  -- \hsSource{nextAtomic} ermoeglicht es, aus einer gegebenen Schaltung
94  -- und einer Kante die naechste Komponente zu ermitteln, die atomar
95  -- ist. Ausserdem wird der Pin angegeben, ueber welchen
96  -- diese Komponente angeschlossen ist.
97
98
99  nextAtomic :: CircuitDescriptor -> Edge -> (CompID, PinID)
100 nextAtomic g e
101     | isToOuter e && (nodeId.nodeDesc $ super) == mainID
102     = (mainID, snkPin e)
103
104     | isToOuter e
105     =   nextAtomic g
106       $ head
107       $ filter (\x-> sourceInfo x == (Just $ nodeId.nodeDesc $ super, snkPin e))
108       $ edges supersuper
109
110     | not.isAtomic $ sub
111     =   nextAtomic g
112       $ head
113       $ filter (\x -> (isFromOuter x) && (snkPin e == srcPin x)) $ edges sub
114
115     | isAtomic sub
116     = (snkComp e, snkPin e)
```

```haskell
    where mainID     = nodeId.nodeDesc $ g
          sub        = getComp   g (snkComp e)
          super      = superNode g (srcComp e)
          supersuper = superNode g (nodeId.nodeDesc $ super)



-- \subsection{Kantensesoren}
--
-- Mit der Funktion \hsSource{fromCompEdges} koennen alle Kanten
-- ausgelesen werden, die von einer bestimmten Komponente her kommen.


fromCompEdges :: CircuitDescriptor -> CompID -> [Edge]
fromCompEdges g cid
    = filter (\x -> (not.isFromOuter $ x)
                 && (cid == (srcComp x) ) ) $ edges $ superNode g cid




-- Hilfreich ist auch die Funktion \hsSource{allEdges}, die wie der
-- Name vermuten laesst, alle Kanten die innerhalb einer
-- Schaltung verbaut, sammelt und als Ergebnisliste zurueckgibt.


allEdges :: CircuitDescriptor -> [Edge]
allEdges g = edges g ++ (concat $ map allEdges (nodes g))


-- \begriff{snk} und \begriff{src} sind Kurzschreibweisen fuer
-- \begriff{Sink} und \begriff{Source}. Mit den Funktionen
-- \hsSource{snkPin} und \hsSource{srcPin} lassen sich aus einer Kante
-- entweder der Quell- oder der Zielpin auslesen. \hsSource{snkComp}
-- und \hsSource{srcComp} sind fuer die Quell- sowie fuer die
-- Zielkomponente verantwortlich.


snkPin :: Edge -> PinID
snkPin (MkEdge (_, _) (_, pid)) = pid

srcPin :: Edge -> PinID
srcPin (MkEdge (_, pid) (_, _)) = pid

```

```
161 snkComp :: Edge -> CompID
162 snkComp (MkEdge (_, _) (Just cid, _)) = cid
163
164 srcComp :: Edge -> CompID
165 srcComp (MkEdge (Just cid, _) (_, _)) = cid
```

A.13. Circuit.Splice

```
1  module Circuit.Splice
2  where
3
4  -- Verwendet werden die Standarddefinitionen, sowie eine Sensor und
5  -- eine Workerfunktion.
6
7
8  import Data.List (nub)
9
10 import Circuit.Graphs
11
12 import Circuit.Descriptor
13 import Circuit.Workers (alterCompIDs)
14 import Circuit.Sensors (maxCompID)
15
16
17 -- Auch wenn hier tatsaechlich zwei Funktionen stehen, wird
18 -- \hsSource{splice} als eine Einheit
19 -- angesehen. \hsSource{splice'} enthaelt die Funktionalitaet,
20 -- \hsSource{splice} ist der oeffentliche Bezeichner, der obendrein
21 -- noch eine grundlegende Fehlerpruefung macht.
22
23 -- \hsSource{splice} wird eine \hsSource{rewire} Funktion
24 -- uebergeben. Diese Funktion enthaelt die Logik, nach der die
25 -- ``Verdrahtung'' der beiden Schaltkreise erfolgen wird. Hier ist es
26 -- dann moeglich, beispielsweise sequentiell oder parallel zu
27 -- verdrahten. Ausserdem erwartet \hsSource{splice} noch zwei
28 -- Schaltungen, die zusammengefuehrt werden sollen. Diese beiden werden
29 -- auf die gewaehlte Art miteinander verbunden. Die uebrigen
30 -- ``Draehte'' werden nach aussen gefuehrt, ein neuer Name wird erzeugt.
31 -- Dieser neue Schaltkreis wird dann zurueckgegeben.
32
```

```haskell
splice :: ( (CircuitDescriptor -> CircuitDescriptor -> ([Edge], (Pins, Pins)))
          , String)
       -> CircuitDescriptor -> CircuitDescriptor -> CircuitDescriptor
splice _           sg NoDescriptor = sg
splice _           NoDescriptor sg = sg
splice (rewire, s) cd_f cd_g       = splice' (rewire, s) cd_f cd_g


splice' :: ( (CircuitDescriptor -> CircuitDescriptor -> ([Edge], (Pins, Pins)))
           , String)
        -> CircuitDescriptor -> CircuitDescriptor -> CircuitDescriptor
splice' (rewire, s) cd_f cd_g
    = MkCombinatorial
        { nodeDesc = MkNode
            { label   = (label.nodeDesc $ cd_f') ++ s ++ (label.nodeDesc $ cd_g')
            , nodeId  = 0
            , sinks   = srcs
            , sources = snks
            }
        , nodes   = cd_f': cd_g' : []
        , edges   = es
        , cycles  = (cycles cd_f) + (cycles cd_g)
        , space   = (space  cd_f) + (space  cd_g)
        }
    where cd_f'              = alterCompIDs 1                    cd_f
          cd_g'              = alterCompIDs (maxCompID cd_f' +1) cd_g
          (es, (srcs, snks)) = rewire cd_f' cd_g'



-- \subsection{Verdrahtungsvarianten} Die Funktion \hsSource{splice}
-- aus dem Modul \ref{mod:Circuit.Splice} verwendet fuer das
-- ``Verdrahten'' eine der folgenden
-- \hsSource{rewire}-Funktionen. Daneben wird eine Zeichenkette
-- zugeordnet, um spaeter Debugausgaben erzeugen zu koennen.

-- Die \hsSource{connect} Funktion verbindet zwei Schaltkreise zu
-- einem neuen. Hierbei wird sequentiell verbunden, als Zeichenkette
-- wird derselbe Operator angegeben, der aus der
-- \hsSource{Arrow}-Schreibweise bekannt ist.

connect :: CircuitDescriptor -> CircuitDescriptor -> CircuitDescriptor
connect = splice (seqRewire, ">>>")

```

```haskell

-- Neben dem sequentiellen Verbinden lassen sich Schaltkreise auch
-- parallel verbinden. Dies ist mit der Funktion \hsSource{combine}
-- moeglich.  Als Zeichenkette wird auch hier das aus der
-- \hsSource{Arrow}-Schreibweise bekannte Operatorsymbol verwendet.

combine :: CircuitDescriptor -> CircuitDescriptor -> CircuitDescriptor
combine = splice (parRewire, "&&&")


-- Eine Variante der \hsSource{combine} Funktion ist die Funktion
-- \hsSource{dupCombine}. Hier werden die Eingaenge zunaechst dupliziert
-- und dann parallel weiterverbunden.

dupCombine :: CircuitDescriptor -> CircuitDescriptor -> CircuitDescriptor
dupCombine = splice (dupParRewire, ">2>")


-- Um eine Verzoegerung in Hardware zu realisieren, ist es notwendig,
-- die Daten zu speichern. Dies wird ueber ein Register erreicht. Nach
-- der gewuenschten Anzahl von Zyklen, kann das Datum aus dem
-- Register wieder ausgelesen werden und in der Schaltung weiter
-- verwendet werden.

delayByRegister :: CircuitDescriptor -> CircuitDescriptor
delayByRegister cd@(MkCombinatorial nd _ _ _ _)
    = MkComposite (cd : reg : [])
    where reg = mkRegister nd



-- Moechte man einen \begriff{Loop} erstellen, so wird dieser durch ein
-- Register gefuehrt, das eine Verzoegerung um einen Takt
-- ermoeglicht. Die Funktion, die ein Bauteil um eine Schleife mit
-- Register erweitert, nennt sich \hsSource{registerloopRewire}. Diese
-- Funktion laesst sich mittels \hsSource{splice} zu der nach aussen
-- verwendeten \hsSource{loopWithRegister} Funktion umbauen.


loopWithRegister :: CircuitDescriptor -> CircuitDescriptor
loopWithRegister cd
    = MkLoop
        { nodeDesc = MkNode
            { label    = "loop(" ++ (label.nodeDesc $ cd) ++ ")"
```

```
121             , nodeId  = 0
122             , sinks   = srcs
123             , sources = snks
124             }
125         , nodes   = [alterCompIDs 1 cd]
126         , edges   = es
127         , space   = space cd
128         }
129     where (es, (srcs, snks)) = registerLoopRewire cd
130
131
132 -- Unter \hsSource{rewire}-Funktionen sind Funktionen zu
133 -- verstehen, die eine Vorstufe fuer die eigentliche Verbindung (das
134 -- \begriff{splicen}) darstellen. Zwei Schaltkreise werden jeweils in
135 -- eine Zwischendarstellung ueberfuehrt. Die Zwischendarstellung besteht
136 -- aus einer Liste von neuen Kanten (\hsSource{[Edge]}), zusammen mit
137 -- den ueberbleibenden Ein- und Ausgangspins.
138 --
139 -- Alle \hsSource{rewire}-Funktionen nutzen eine Funktion, naemlich
140 -- \hsSource{wire}. Das Verbinden von Draehten mit Komponenten ist,
141 -- unabhaengig davon, ob sequentiell oder parallel verbunden werden
142 -- soll, immer gleich. Eingehende Parameter zu \hsSource{wire} sind
143 -- die beiden Komponentennummern, sowie die Pinlisten. Auch diese
144 -- Funktion erzeugt die schon beschriebene Zwischendarstellung.
145
146 wire :: Maybe CompID -> Maybe CompID -> Pins -> Pins -> ([Edge], (Pins, Pins))
147 wire cid_l cid_r pins_l pins_r
148     = (edges, (drop cnt pins_l, drop cnt pins_r))
149     where points_l = map ((,) (cid_l)) pins_l
150           points_r = map ((,) (cid_r)) pins_r
151           edges    = map (uncurry MkEdge) $ zip points_l points_r
152           cnt      = length edges
153
154
155 -- \hsSource{wire_} ist ein Synonym fuer \hsSource{fst . wire}.
156
157 wire_ :: Maybe CompID -> Maybe CompID -> Pins -> Pins -> [Edge]
158 wire_ cid_l cid_r pins_l pins_r = fst $ wire cid_l cid_r pins_l pins_r
159
160
161 -- Bei der Funktion \hsSource{seqRewire} werden die Verbindungen
162 -- sequentiell erstellt; uebrige Ein oder Ausgaenge werden zu den gesamt
163 -- Ein- und Ausgaengen hinzugefuegt.
164
```

```haskell
seqRewire :: CircuitDescriptor -> CircuitDescriptor -> ([Edge], (Pins, Pins))
seqRewire sg_l sg_r
  = ( fromOuterToL ++ fromOuterToR ++ edgs ++ fromRToOuter ++ fromLToOuter
    , (super_srcs, super_snks)
    )
    where
      (edgs, (srcs_l', snks_r'))
        =  wire (Just $ nodeId.nodeDesc $ sg_l)
                (Just $ nodeId.nodeDesc $ sg_r)
                (sources.nodeDesc $ sg_l) (sinks.nodeDesc $ sg_r)
      super_srcs
        =  [0..(length.sinks.nodeDesc   $ sg_l) + length snks_r' -1]
      super_snks
        =  [0..(length.sources.nodeDesc $ sg_r) + length srcs_l' -1]
      ( fromOuterToL, (super_srcs', _))
        =  wire Nothing
                (Just $ nodeId.nodeDesc $ sg_l)
                super_srcs
                (sinks.nodeDesc $ sg_l)
      ( fromOuterToR, (_          , _))
        =  wire Nothing
                (Just $ nodeId.nodeDesc $ sg_r)
                super_srcs'
                (drop (length fromOuterToL) $ sinks.nodeDesc $ sg_r)
      ( fromRToOuter, (_, super_snks'))
        =  wire (Just $ nodeId.nodeDesc $ sg_r)
                Nothing
                (sources.nodeDesc $ sg_r )
                super_snks
      ( fromLToOuter, (_, _))
        =  wire (Just $ nodeId.nodeDesc $ sg_l)
                Nothing
                (drop (length fromRToOuter) $ sources.nodeDesc $ sg_l)
                super_snks'


-- Bei der \hsSource{parRewire} Funktion werden beide Bausteine
-- ``uebereinander'' angeordnet. Die Eingaenge beider Komponenten, sowie
-- deren Ausgaenge werden parallel geschaltet.

parRewire :: CircuitDescriptor -> CircuitDescriptor -> ([Edge], (Pins, Pins))
parRewire sg_u sg_d
    = ( goingIn_edges ++ goingOut_edges
      , (super_srcs, super_snks)
```

```
209       )
210     where
211       super_srcs
212         = [0..(length $(sinks.nodeDesc $   sg_u) ++ (sinks.nodeDesc $   sg_d))-1]
213       super_snks
214         = [0..(length $(sources.nodeDesc $ sg_u) ++ (sources.nodeDesc $ sg_d))-1]
215       goingIn_edges
216         = (wire_ Nothing
217                  (Just $ nodeId.nodeDesc $ sg_u)
218                  (super_srcs)
219                  (sinks.nodeDesc $ sg_u)
220           ) ++
221           (wire_ Nothing
222                  (Just $ nodeId.nodeDesc $ sg_d)
223                  (drop (length.sinks.nodeDesc $ sg_u) super_srcs)
224                  (sinks.nodeDesc $ sg_d)
225           )
226       goingOut_edges
227         = (wire_ (Just $ nodeId.nodeDesc $ sg_u)
228                  Nothing
229                  (sources.nodeDesc $ sg_u)
230                  (super_snks)
231           ) ++
232           (wire_ (Just $ nodeId.nodeDesc $ sg_d)
233                  Nothing
234                  (sources.nodeDesc $ sg_d)
235                  (drop (length.sources.nodeDesc $ sg_u) super_snks)
236           )
237
238
239 -- Die Funktion \hsSource{dupParRewire} funktioniert dabei analog zur
240 -- Funktion \hsSource{parRewire}. Lediglich die Eingaenge werden
241 -- zunaechst dupliziert und dann auf beide Komponenten geschaltet.
242
243 dupParRewire :: CircuitDescriptor -> CircuitDescriptor -> ([Edge], (Pins, Pins))
244 dupParRewire sg_u sg_d
245     = ( goingIn_edges ++ goingOut_edges
246       , (super_srcs, super_snks)
247       )
248     where
249       super_srcs
250         = [0..(length.sinks.nodeDesc $ sg_u) -1]
251       super_snks
252         = [0..(length $(sources.nodeDesc $ sg_u) ++ (sources.nodeDesc $ sg_d))-1]
```

```
253       goingIn_edges
254         = (wire_ Nothing
255                  (Just $ nodeId.nodeDesc $ sg_u)
256                  super_srcs
257                  (sinks.nodeDesc $ sg_u)
258           ) ++
259           (wire_ Nothing
260                  (Just $ nodeId.nodeDesc $ sg_d)
261                  super_srcs
262                  (sinks.nodeDesc $ sg_d)
263           )
264       goingOut_edges
265         = (wire_ (Just $ nodeId.nodeDesc $ sg_u)
266                  Nothing
267                  (sources.nodeDesc $ sg_u)
268                  (super_snks)
269           ) ++
270           (wire_ (Just $ nodeId.nodeDesc $ sg_d)
271                  Nothing
272                  (sources.nodeDesc $ sg_d)
273                  (drop (length.sources.nodeDesc $ sg_u) super_snks))
274
275
276
277 registerLoopRewire :: CircuitDescriptor -> ([Edge], (Pins, Pins))
278 registerLoopRewire cd
279     = (es, (srcs, snks))
280     where
281       reg = mkRegister $ nodeDesc emptyCircuit
282       (es1, (srcs1, snks1)) = seqRewire cd reg
283       (es2, (srcs2, snks2)) = seqRewire reg cd
284       es   = es1 ++ es2
285       srcs =    nub $ filter (flip elem srcs2) srcs1
286              ++ filter (flip elem srcs1) srcs2
287       snks =    nub $ filter (flip elem snks2) snks1
288              ++ filter (flip elem snks1) snks2
```

A.14. Circuit.Tests

```
1 module Circuit.Tests
2 where
```

```haskell

-- Lediglich das Modul \hsSource{Circuit.Descriptor} wird benoetigt


import Circuit.Descriptor


-- Die Funktion \hsSource{isFromOrToComp} ermittelt ob eine uebergebene
-- Kante eine Verbindung zu einer Komponente, die ueber ihre
-- Komponenten ID identifiziert wird, darstellt.

isFromOrToComp :: CompID -> Edge -> Bool
isFromOrToComp cid (MkEdge (Nothing, pi) (Just co, po)) = cid == co
isFromOrToComp cid (MkEdge (Just ci, pi) (Nothing, po)) = cid == ci
isFromOrToComp cid (MkEdge (Just ci, pi) (Just co, po)) = cid == co


-- Jeder Schaltkreis besitzt eine innere Verschaltung und daneben eine
-- Verbindung zur Aussenwelt. Eine Kante kann von ``Aussen'' her kommen,
-- oder nach ``Aussen'' gehen. Genau diese Kanten sind die Pins eines
-- tatsaechlichen Chips.


-- Mit \hsSource{isToOuter} wird getestet, ob es sich um eine
-- abgehende Kante, also einen ``outgoing''-Pin handelt.

isToOuter :: Edge -> Bool
isToOuter (MkEdge (_, _) (Nothing, _)) = True
isToOuter _                            = False


-- \par Das Pendant zu abgehenden Kanten sind eingehende Kanten. Diese
-- werden auch als Inputs bezeichnet. \hsSource{isFromOuter} testet,
-- ob eine Kante eingehend ist.

isFromOuter :: Edge -> Bool
isFromOuter (MkEdge (Nothing, _) (_, _)) = True
isFromOuter _                            = False




-- Mit der Funktion \hsSource{hasLabel} wird ueberprueft, ob ein
-- Schaltkreis den uebergebenen Namen traegt.
```

```
47
48  hasLabel :: String -> CircuitDescriptor -> Bool
49  hasLabel s
50      = ((== s).label.nodeDesc)
51
52
53  -- \hsSource{isAtomic} ueberprueft, ob ein Baustein atomar ist, also ob
54  -- er aus weiteren Bausteinen zusammengesetzt ist, oder nicht.
55
56  isAtomic :: CircuitDescriptor -> Bool
57  isAtomic g
58      = if (length (nodes g) == 0) then True else False
59
60
61  -- Die Funktion \hsSource{isSuperNodeOf} testet, ob eine Komponente
62  -- eine bestimmte andere Komponente, identifiziert durch ihre
63  -- Komponenten ID, enthaelt.
64
65  isSuperNodeOf :: CircuitDescriptor -> CompID -> Bool
66  isSuperNodeOf g cid
67      = if length (filter (== cid) subNodes) > 0
68          then True
69          else False
70      where subNodes = map (nodeId.nodeDesc) $ nodes g
71
72
73  -- Mit der Funktion \hsSource{isGenerated} laesst sich herausfinden, ob
74  -- einen Komponente eine vom Entwickler entwickelte Komponente ist,
75  -- oder ob diese Komponente automatisch vom System erzeugt wurde.
76
77  isGenerated :: CircuitDescriptor -> Bool
78  isGenerated s =     ((== '|').head.label.nodeDesc) s
79                  && ((== '|').head.reverse.label.nodeDesc) s
80
81
82  isID :: CircuitDescriptor -> Bool
83  isID = hasLabel "-ID-"
84
85  -- In einem weiteren Test wird die Art der Schaltung
86  -- ueberprueft.
87
88  isCombinatorial :: CircuitDescriptor -> Bool
89  isCombinatorial (MkCombinatorial _ _ _ _ _) = True
90  isCombinatorial otherwise           = False
```

```
91
92 isRegister :: CircuitDescriptor -> Bool
93 isRegister (MkRegister _ _) = True
94 isRegister otherwise      = False
95
96 isLoop :: CircuitDescriptor -> Bool
97 isLoop (MkLoop _ _ _ _) = True
98 isLoop otherwise        = False
```

A.15. Circuit.Tools

```
1  module Circuit.Tools
2  where
3
4
5  -- Eingebunden werden fuer die Demo-Funktionen aus den
6  -- Systembibliotheken die folgenden Module:
7
8  import System.IO (writeFile)
9  import System.Cmd (system)
10
11 -- Daneben wird noch der \begriff{except}-Operator (\hsSource{(\\)})
12 -- aus dem Listenmodul benoetigt:
13
14 import Data.List ((\\))
15
16
17 -- Schliesslich werden eine ganze Reihe von Modulen aus der
18 -- \hsSource{Circuit}-Reihe verwendet:
19
20 import Circuit.Descriptor
21 import Circuit.Show
22 import Circuit.Graphs
23 import Circuit.Auxillary
24 import Circuit.Show.DOT
25 import Circuit.Workers (mergeEdges)
26 import Circuit.Tests
27
28
29 -- \subsection{Demo Funktionen} Mit \hsSource{write} und
30 -- \hsSource{genPicture} werden zwei Funktionen definiert, die auf das
```

```
31 -- Dateisystem zugreifen und letztlich dazu verwendet werden, eine
32 -- Grafik einer Schaltung zu erstellen. Dazu wird mittels
33 -- \hsSource{show} die \begriff{.dot}-Darstellung des uebergebenen
34 -- Schaltkreises erzeugt und im \hsSource{/tmp}-Folder abgespeichert.
35
36 -- \hsSource{genPicture} greift dann auf die erzeugte Datei zu,
37 -- und erzeugt aus dem \begriff{.dot}-File ein Bild mit der
38 -- Grafik. Hierbei wird vorausgesetzt, dass auf dem System die
39 -- \begriff{graphviz}-Umgebung vorinstalliert ist.
40
41 write x    = writeFile "/tmp/test.dot" (Circuit.Show.DOT.showCircuit x)
42 genPicture = system "dot /tmp/test.dot -Tjpg -o /tmp/test.jpg"
43
44
45
46 -- Mit der Funktion \hsSource{toCircuit} laesst sich aus einer minimal
47 -- Definition ein gueltiger Schaltkreis erzeugen. Die minimale
48 -- Definition muss wenigstens einen Namen, sowie die Anzahl der
49 -- eingehenden und ausgehenden Pins enthalten.
50
51 toCircuit :: (String, Int, Int) -> CircuitDescriptor
52 toCircuit (name, inPins, outPins)
53     = emptyCircuit { nodeDesc = nodedesc
54                    }
55       where nodedesc = MkNode { label   = name
56                               , nodeId  = 0
57                               , sinks   = [0..(inPins -1)]
58                               , sources = [0..(outPins -1)]
59                               }
60
61
62 -- Die Funktion \hsSource{filterByName} ist ein Beispiel fuer die
63 -- Maechtigkeit des neuartigen Ansatzes. \hsSource{filterByName} geht
64 -- hier ueber eine Schaltung hinweg und nimmt alle vorkommenden
65 -- Bausteine mit einem gewissen Namen heraus. So lassen sich zum
66 -- Testen sehr leicht von den enthaltenen Schaltungen diese
67 -- durch eine andere Version ersetzen.
68
69 filterByName :: CircuitDescriptor -> String -> CircuitDescriptor
70 filterByName s n
71     = if ((label . nodeDesc) s == n) && (length (nodes s) < 1)
72         then NoDescriptor
73         else s { nodes = (map (flip filterByName n) $ nodes s) }
74
```

```
75
76  -- Zur Anwendung wird \hsSource{filterByName} in der naechsten
77  -- Funktion, naemlich in \hsSource{replace} gebracht. Hier uebergibt man
78  -- die Schaltung in der man aendern moechte. Ausserdem uebergibt man ein
79  -- Tupel bestehendend aus einem \hsSource{from}-Baustein und einem
80  -- \hsSource{to}-Baustein, wobei nach dem \hsSource{from}-Baustein
81  -- gesucht wird, und dieser dann mit dem \hsSource{to}-Baustein
82  -- ersetzt wird. Als Ergebnis erhaelt man eine veraenderte Schaltung
83
84  replace :: CircuitDescriptor
85          -> (CircuitDescriptor, CircuitDescriptor)
86          -> CircuitDescriptor
87  replace s ft@(from, to)
88      | not $ isAtomic s
89      = s { nodes = map (flip replace $ ft) (nodes s) }
90
91      |  (label . nodeDesc) s           == (label . nodeDesc) from
92      && length (sinks   . nodeDesc $ s) == length (sinks   . nodeDesc $ from)
93      && length (sources . nodeDesc $ s) == length (sources . nodeDesc $ from)
94      && length (sinks   . nodeDesc $ s) == length (sinks   . nodeDesc $ to)
95      && length (sources . nodeDesc $ s) == length (sources . nodeDesc $ to)
96      = to { nodeDesc = (nodeDesc to) { nodeId = (nodeId . nodeDesc) s } }
97
98      | otherwise = s
99
100
101 -- Die Funktion \hsSource{bypass} ermoeglicht eine aehnliche
102 -- Funktionalitaet, wie schon \hsSource{filterByName} oder
103 -- \hsSource{replace}.  Allerdings nimmt \hsSource{bypass} nur die
104 -- gefundenen Bausteine aus der Schaltung (heraus).
105 --
106
107 bypass :: CircuitDescriptor -> CircuitDescriptor -> CircuitDescriptor
108 bypass s item
109     = s { nodes = ns
110         , edges = es
111         }
112     where (es, ns)
113       = foldl (rebuildIf (\x -> label (nodeDesc x) == label (nodeDesc item)))
114               (edges s, []) $ nodes s
115
116
117 rebuildIf :: (CircuitDescriptor -> Bool) ->  ([Edge], [CircuitDescriptor])
118           -> CircuitDescriptor -> ([Edge], [CircuitDescriptor])
```

```
119 rebuildIf isIt (super_es, new_ns) NoDescriptor = (super_es, new_ns)
120 rebuildIf isIt (super_es, new_ns) n
121     |  isIt n && length (sinks . nodeDesc $ n) == length (sources . nodeDesc $ n)
122     = (new_es , new_ns)
123
124     | otherwise
125     = (super_es, new_ns ++ [n'])
126
127     where new_es  = (super_es \\ (lws ++ rws)) ++ nws
128           lws     = leftWires super_es (nodeId . nodeDesc $ n)
129           rws     = rightWires super_es (nodeId . nodeDesc $ n)
130           nws     = zipWith MkEdge (map sourceInfo lws) (map sinkInfo rws)
131           (es,ns) = foldl (rebuildIf isIt) (edges n, []) $ nodes n
132           n'      = n { nodes = ns
133                       , edges = es
134                       }
135
136
137 grepWires :: (Edge -> Anchor) -> [Edge] -> CompID -> [Edge]
138 grepWires f es cid = filter (\e -> fst (f e) == Just cid) es
139
140 leftWires :: [Edge] -> CompID -> [Edge]
141 leftWires = grepWires sinkInfo
142
143 rightWires :: [Edge] -> CompID -> [Edge]
144 rightWires = grepWires sourceInfo
145
146 solderWires :: ([Edge], [Edge]) -> [Edge]
147 solderWires = mergeEdges
```

A.16. Circuit.Workers

```
1 -- In diesem Modul werden eine Reihe von \begriff{worker}-Funktionen
2 -- definiert die alle einen uebergebenen Wert veraendern und die
3 -- geaenderte Variante zurueckliefern.
4
5 module Circuit.Workers
6 where
7
8 -- Zur Funktionsdefinition werden Funktionen aus folgenden Modulen
9 -- benoetigt.
```

```
10
11  import Data.List (nub, (\\))
12
13  import GHC.Exts (sortWith)
14
15  import Circuit.Descriptor
16  import Circuit.Sensors
17  import Circuit.Tests
18
19
20
21  -- \subsection{CircuitDescriptor Funktionen}
22  -- Dieser Abschnitt befasst sich mit Funktionen, die auf
23  -- \hsSource{CircuitDescriptor}en arbeiten.
24
25  -- Mit der Funktion \hsSource{alterCompIDs} lassen sich alle
26  -- Komponenten IDs innerhalb eines \hsSource{CircuitDescriptor}s
27  -- veraendern. Der erste Parameter legt dabei die kleinst moegliche ID
28  -- fest.
29
30  alterCompIDs :: Int -> CircuitDescriptor -> CircuitDescriptor
31  alterCompIDs i sg
32      = sg { nodeDesc = nd { nodeId = nodeId nd + i }
33           , nodes  = map (alterCompIDs i) $ nodes sg
34           , edges  = map (\ (MkEdge (ci,pi) (co,po))
35                          -> (MkEdge (maybe ci (Just.(+i)) $ ci ,pi)
36                                     (maybe co (Just.(+i)) $ co ,po))
37                          ) $ edges sg
38           }
39       where nd = nodeDesc sg
40
41
42  -- Die Funktion \hsSource{dropCircuit} ermoeglicht es, einzelne
43  -- Schaltkreis-Beschreibungen aus dem \hsSource{CircuitDescriptor} zu
44  -- entfernen.  Hierzu wird eine match-Funktion als erster Parameter
45  -- erwartet.
46
47  dropCircuit :: (CircuitDescriptor -> Bool)
48              -> CircuitDescriptor -> CircuitDescriptor
49  dropCircuit f sg
50      = sg { nodes = newNodes
51           , edges = newEdges
52           }
53      where
```

```
54        specific
55          = filter f (nodes sg)
56        newEdges
57          = foldl (flip dropEdgesBordering)
58                  (edges sg)
59                  (map (nodeId.nodeDesc) specific)
60        newNodes
61          = map (dropCircuit f) $ nodes sg \\ specific
62
63
64  -- flatten} ist eine Funktion, welche die interne Struktur des
65  -- \hsSource{CircuitDescriptor}s \begriff{glaettet}. Jeder
66  -- CircuitDescriptor} der nicht atomar ist, enthaelt weitere
67  -- Unterstrukturen. Diese beschreiben, woraus dieser
68  -- CircuitDescriptor} aufgebaut wird. Enthaelt der
69  -- \hsSource{CircuitDescriptor} unnoetige Verschachtelungen, werden
70  -- diese mittels flatten entfernt.
71
72  -- Als Sonderfall gelten die Schaltungen, die Schleifen
73  -- darstellen. Hier gibt es keine ueberfluessigen Verschachtelungen,
74  -- mindestens aber muss der Algorithmus zum Erkennen solcher ein
75  -- anderer sein, sodass \hsSource{flatten} auf diese Teilbereiche
76  -- zunaechst nicht angewandt werden soll.
77  flatten :: CircuitDescriptor -> CircuitDescriptor
78  flatten g
79      | isLoop g
80      = g
81
82      | otherwise
83      = g { nodes = nub $ atomCIDs
84          , edges =       esBetweenAtoms
85          }
86      where atomCIDs
87              = filter isAtomic $ allCircuits g
88            esFromAtoms
89              =   concat
90                $ map (fromCompEdges g . nodeId . nodeDesc) atomCIDs
91            esFromOuter
92              = filter isFromOuter $ edges g
93            esBetweenAtoms
94              = zipWith MkEdge
95                        (map sourceInfo $ esFromOuter ++ esFromAtoms)
96                        (map nextAtomOrOut $ esFromOuter ++ esFromAtoms)
97            nextAtomOrOut
```

```
98              = (\e -> let (c, p) = nextAtomic g e
99                       in  if c == mainID then (Nothing, p) else (Just c, p))
100           mainID
101             = nodeId.nodeDesc $ g
102
103
104 -- Die Funktionen \hsSource{dropGenerated} sowie \hsSource{dropID}
105 -- stellen Spezialfaelle der \hsSource{dropCircuit} Funktion dar.
106 -- dropGenerated} loescht saemtliche \hsSource{CircuitDescriptor}en, die
107 -- automatisch generiert wurden. Ebenso loescht \hsSource{dropID}
108 -- CircuitDescriptor}en, die den \hsSource{isID}-Test
109 -- bestehen. \hsSource{isID} sowie \hsSource{isGenerated} sind im
110 -- Modul \ref{mod:Circuit.Tests} beschrieben.
111
112 dropGenerated :: CircuitDescriptor -> CircuitDescriptor
113 dropGenerated = dropCircuit isGenerated
114
115 dropID :: CircuitDescriptor -> CircuitDescriptor
116 dropID = dropCircuit isID
117
118
119
120 -- Diese Funktionen arbeiten auf \hsSource{CircuitDescriptor}en, und
121 -- erzeugen Kanten oder es handelt sich um Funktionen, die aus
122 -- bestehenden Kanten neue Kanten generieren.
123
124
125 -- Mit der Funktion \hsSource{connectCID} lassen sich zwei
126 -- \hsSource{CircuitDescriptor}en miteinander verbinden. Dabei werden
127 -- zwei \hsSource{CircuitDescriptor}en uebergeben, sowie die
128 -- Quellkomponenten ID und die Zielkomponenten ID zusammen mit einer
129 -- Zielpin ID. Erzeugt wird eine Kante, welche die Verbindung
130 -- zwischen beiden \hsSource{CircuitDescriptor}en darstellt. Von der
131 -- Quelle wird keine \hsSource{PinID} benoetigt, da hier auf den naechst
132 -- freien Pin zurueckgegriffen wird. Auf der Zielseite ist es
133 -- notwendig, einen Zielpin zu definieren.
134
135 connectCID :: CircuitDescriptor -> CircuitDescriptor -> CompID
136            -> (CompID, PinID) -> Edge
137 connectCID old_g g cidF (cidT,pidT)
138     = MkEdge (Just cidF, nextFpin) (Just cidT, pidT)
139     where nextFpin
140             = head $ drop cntEsFrom $ sources.nodeDesc $ getComp old_g cidF
141           cntEsFrom
```

```
142             =   length
143               $ filter (\x -> (not.isFromOuter $ x) && (srcComp x == cidF))
144               $ edges g
145
146
147 -- Zum Entfernen von Kanten, die an eine Komponente angrenzen, ist die
148 -- Funktion \hsSource{dropEdgesBordering} da. Uebergeben wird die ID
149 -- der Komponente, die herausgeloest werden soll, sowie die Liste mit
150 -- den betroffenen Kanten. Es wird dann eine neue Liste mit Kanten
151 -- erstellt, die nicht mehr zu der Komponente mit der besagten ID
152 -- fuehren. Alle Kanten, die nicht mehr an einer Komponente andocken,
153 -- werden zusammengefuegt.  Diese Funktion kann nur dann funktionieren,
154 -- wenn die zu loesende Komponente genausoviele eingehende Pins, wie
155 -- ausgehende Pins besitzt.
156
157 dropEdgesBordering :: CompID -> [Edge] -> [Edge]
158 dropEdgesBordering cid es
159     = (es ++ mergeEdges (toIt, fromIt)) \\ (toIt ++ fromIt)
160     where toIt   = filter ((== (Just cid)).fst.sinkInfo)   $ es
161           fromIt = filter ((== (Just cid)).fst.sourceInfo) $ es
162
163
164 -- \hsSource{mergeEdges} ist eine Funktion, die zwei Listen mit Kanten
165 -- entgegennimmt und diese zusammenfasst. Kanten, die auf einen
166 -- Pin enden und Kanten, die vom gleichen Pin starten, werden zu einer
167 -- Kante zusammengefasst.
168
169 mergeEdges :: ([Edge], [Edge]) -> [Edge]
170 mergeEdges (xs, ys)
171     = zipWith (\x y -> MkEdge (sourceInfo x) (sinkInfo y)) xs' ys'
172     where x_snkPins
173             = map snkPin xs
174           y_srcPins
175             = map srcPin ys
176           xs'
177             =   sortWith snkPin
178               $ filter (\edg -> (snkPin edg) `elem` y_srcPins) xs
179           ys'
180             =   sortWith srcPin
181               $ filter (\edg -> (srcPin edg) `elem` x_snkPins) ys
182
183
184 -- Mit der \hsSource{fillEdgeInfoCompID} Funktion, lassen sich die
185 -- Quell- und Zielkomponenten IDs in Kanten setzen, in denen bis
```

```
186 -- dahin \hsSource{Nothing} als Wert gespeichert ist. Dies ist
187 -- notwendig, wenn eine neue Komponente in eine bestehende Struktur
188 -- eingefuegt wird. Eine noch nicht integrierte
189 -- Komponente bekommt ihre Werte von einer unbekannte Komponente
190 -- (\hsSource{Nothing}) und liefert die Ergebnisse auch an
191 -- \hsSource{Nothing}. Wird sie nun eine Unterkomponente, kann das
192 -- \hsSource{Nothing} durch eine tatsaechliche Komponenten ID ersetzt
193 -- werden.
194
195 fillEdgeInfoCompID :: CompID -> Edge -> Edge
196 fillEdgeInfoCompID cid (MkEdge (Nothing, srcPid) (snkInfo))
197   = (MkEdge (Just cid, srcPid) (snkInfo))
198 fillEdgeInfoCompID cid (MkEdge (srcInfo) (Nothing, snkPid))
199   = (MkEdge (srcInfo) (Just cid, snkPid))
200 fillEdgeInfoCompID _   e
201   = e
202
203
204 -- Ein aehnliches Problem wie \hsSource{fillEdgeInfoCompID} wird auch
205 -- von den Funktionen \hsSource{fillSrcInfoCompID} und
206 -- \hsSource{fillSnkInfoCompID} geloest. Diese unterscheiden sich
207 -- lediglich darin, dass diese Funktionen jeweils nur die Quellpins
208 -- oder nur die Zielpins betreffen.
209
210 fillSrcInfoCompID :: CompID -> Edge -> Edge
211 fillSrcInfoCompID cid (MkEdge (Nothing, srcPid) (snkCid, snkPid))
212     = (MkEdge (Just cid, srcPid) (snkCid, snkPid))
213
214 fillSnkInfoCompID :: CompID -> Edge -> Edge
215 fillSnkInfoCompID cid (MkEdge (srcCid, srcPid) (Nothing, snkPid))
216     = (MkEdge (srcCid, srcPid) (Just cid, snkPid))
```

A.17. Circuit

```
1 module Circuit
2   ( Grid(..)
3   , runGrid
4
5   , Stream(..)
6
7   , Show(..)
```

```
  )
where

-- Modul: Circuit
-- stellt ein Wrapper-Modul nach dem Fassaden Entwurfsmuster dar.

import Circuit.Grid
import Circuit.Stream
import Circuit.Show
```

A.18. Circuit.Arrow.Class

```
module Circuit.Arrow.Class
    ( Arrow(..)
    , ArrowLoop(..)
    , ArrowCircuit(..)
    , ArrowChoice(..)
    )
where

-- Folgenden Module werden benoetigt, um die Arrows definieren zu
-- koennen:
import Prelude hiding (id, (.))
import qualified Prelude as Pr
import Control.Category
import Circuit.ShowType



-- Durch die Angabe von Praezedenzen lassen sich Operatoren ohne
-- Klammer schreiben. Dies hilft bei der Leserlichkeit von
-- Quellcode. Folgende Praezedenzen werden festgelegt:

infixr 3 ***
infixr 3 &&&


-- Zunaechst einmal muessen die Klassen definiert werden. Diese Vorgaben
-- muessen von allen Instanzdefinitionen befolgt werden.

-- Hier folgt die Klassendefinition eines Arrows. Anzumerken ist, dass
```

```
30 -- diese Klassendefinition geringfuegig von der Standard-Definition
31 -- abweicht. Die Arrow-Klasse wird hier in der \hsSource{arr} Funktion
32 -- soweit eingeschraenkt, dass nur Funktionen (\hsSource{(b -> c)}) von
33 -- \hsSource{arr} akzeptiert werden, die Mitglied der Typklasse
34 -- \hsSource{ShowType b c} sind.
35
36 class (Category a) => Arrow a where
37     arr    :: (ShowType b c) => (b -> c) -> a b c
38     first  :: a b c -> a (b, d) (c, d)
39     second :: a b c -> a (d, b) (d, c)
40     second f = arr swap >>> first f >>> arr swap
41               where swap :: (b, c) -> (c, b)
42                     swap ~(x, y)  = (y, x)
43     (***)  :: a b c -> a b' c' -> a (b, b') (c, c')
44     f *** g = first f >>> second g
45     (&&&)  :: a b c -> a b c'  -> a b (c, c')
46     f &&& g = arr (\b -> (b, b)) >>> f *** g
47
48 -- Weitere Arrow Instanzen sind \hsSource{ArrowLoop}. Hierbei handelt
49 -- es sich um Arrow, die sich selbst wieder aufrufen koennen, und somit
50 -- einen Fixpunkt in der Arrow-Funktion versuchen zu finden.
51
52 class (Arrow a) => ArrowLoop a where
53     loop :: a (b,d) (c,d) -> a b c
54
55
56 -- Mit \hsSource{ArrowCircuit} sind die Arrows gemeint, die sich als
57 -- getaktete Schaltung darstellen lassen. Definiert werden muss hier
58 -- lediglich, was ein Takt ist.
59
60 class (ArrowLoop a) => ArrowCircuit a where
61     delay :: b -> a b b
62
63 -- Eine weitere Typklasse ist \hsSource{ArrowChoice}. Mit Arrow-Choice
64 -- ist es Moeglich, Varianten der selben Loesung parallel nebeneinander
65 -- erzeugen zu lassen.
66
67 class (Arrow a) => ArrowChoice a where
68     left  :: a b c -> a (Either b d) (Either c d)
69     right :: a b c -> a (Either d b) (Either d c)
70     (+++) :: a b c -> a b' c' -> a (Either b b') (Either c c')
71     (|||) :: a b d -> a c  d  -> a (Either b c ) d
72
73     right f = arr mirror >>> left f >>> arr mirror
```

```
74          where mirror :: Either x y -> Either y x
75                mirror (Left x)  = (Right x)
76                mirror (Right x) = (Left x)
77
78      f +++ g = left f  >>> right g
79
80      f ||| g = f +++ g >>> arr untag
81          where untag (Left x)  = x
82                untag (Right x) = x
```

A.19. Circuit.Arrow.Instance

```
1   module Circuit.Arrow.Instance
2   where
3
4   -- Folgenden Module werden benoetigt, um die Arrows definieren zu
5   -- koennen:
6   import Circuit.Arrow.Class
7
8   -- Da die folgende Arrow-Instanz nicht zur Ausfuehrung benoetigt wird,
9   -- sondern damit das Typsystem an anderer Stelle den richtigen Typ
10  -- ableiten kann, ist es moeglich diese Instanz als \begriff{dummy
11  -- Instance} zu definieren. Dies bedeutet, dass die Instanz keine
12  -- definierten Methoden besitzt. Der Kompiler warnt die nicht
13  -- vorhandenen Methoden zwar an, es bleibt allerdings bei der Warnung.
14
15  instance Arrow (->) where
16      arr   f = f
17      first f = (\(x, y) -> (f x, y))
18
19  -- Im folgenden wird eine Instanz der \hsSource{ArrowLoop}-Klasse fuer
20  -- einfache Funktionsauswertung definiert. Auch hier reicht eine
21  -- \begriff{dummy}-Definition fuer das Typsystem.
22
23  instance ArrowLoop (->) where
24  --     loop f b = let (c, d) = f (b, d) in c
```

A.20. Circuit.Arrow.Helpers

```haskell
module Circuit.Arrow.Helpers
where

-- Folgenden Module werden benoetigt, um die Arrows definieren zu
-- koennen:

import Control.Category

import Circuit.Arrow
import Circuit.Descriptor
import Circuit.Grid
import Circuit.Stream
import Circuit.Graphs

-- Weitere Hilfsfunktionen werden notwendig, um schon bestehende
-- \hsSource{Grid}-Arrows mit Schaltkreis Beschreibungen anzureichern.

insert :: b -> (a, b) -> (a, b)
insert sg ~(x, _) = (x, sg)

insEmpty = insert emptyCircuit { nodeDesc = nodedesc }
    where nodedesc = MkNode
      { label = "eeeempty"
      , nodeId = 0
      , sinks = mkPins 1
      , sources = mkPins 3
      }

augment :: (Arrow a) => CircuitDescriptor -> a b c -> Grid a b c
augment sg f = GR (f, sg)


-- Zu guter letzt werden noch Funktionen benoetigt, die bei der
-- Umstrukturierung von Daten gebraucht werden.

movebrc :: ((a, b), c) -> (a, (b, c))
movebrc ~(~(x, y), sg) = (x, (y, sg))

backbrc :: (a, (b, c)) -> ((a, b), c)
backbrc ~(x, ~(y, sg)) = ((x, y), sg)

```

```
42 swapsnd :: ((a, b), c) -> ((a, c), b)
43 swapsnd ~(~(x, y), sg) = ((x, sg), y)
```

A.21. Circuit.Arrow

```
1  module Circuit.Arrow
2    ( Arrow(..)
3    , ArrowLoop(..)
4    , ArrowCircuit(..)
5    , ArrowChoice(..)
6    , returnA
7    , movebrc
8    , backbrc
9    , swapsnd
10   )
11 where
12
13
14 import Prelude (id)
15
16 import Circuit.Arrow.Class
17 import Circuit.Arrow.Instance
18
19 -- | 'returnA' is a standard arrow-function that is similar to return
20 -- in the monad context
21 returnA :: (Arrow a) => a b b
22 returnA = arr id
23
24 -- | 'movebrc', 'backbrc' and 'swapsnd' are functions that change the
25 -- order of tuples
26 movebrc :: ((a, b), c) -> (a, (b, c))
27 movebrc ~(~(x, y), sg) = (x, (y, sg))
28
29 backbrc :: (a, (b, c)) -> ((a, b), c)
30 backbrc ~(x, ~(y, sg)) = ((x, y), sg)
31
32 swapsnd :: ((a, b), c) -> ((a, c), b)
33 swapsnd ~(~(x, y), sg) = ((x, sg), y)
```

A.22. Circuit.Grid.Datatype

```
1  -- Das Modul \hsSource{Circuit.Arrowdefinition} beschreibt, wie die
2  -- Arrow-Klasse zu implementieren sind um damit spaeter Schaltkreise
3  -- beschreiben, bearbeiten oder benutzen zu koennen.
4
5  module Circuit.Grid.Datatype
6  where
7
8  -- Folgende Module werden benoetigt, um den Datentyp definieren zu koennen:
9
10 import Control.Arrow
11
12 import Circuit.Descriptor
13
14
15 -- Ein \hsSource{Grid} ist ein Datentyp, fuer den die Arrow-Klassen
16 -- implementiert werden. Man spricht dann von einem
17 -- \hsSource{Grid}-Arrow. Der Name \hsSource{Grid} soll dabei an ein
18 -- Steckbrett erinnern, auf denen man Bauteile anbringen und
19 -- miteinander verbinden kann. \hsSource{Grid} besitzt 3 Typvariablen
20 -- (\hsSource{a}, \hsSource{b}, \hsSource{c}), wobei \hsSource{a} die
21 -- jeweilige Arrowinstanz repraesentiert (z.B.  \hsSource{(->)}),
22 -- \hsSource{b} und \hsSource{c} stellen den Typ des Arrows dar. Die
23 -- Funktion \hsSource{(+1)} hat den Typ \hsSource{Int -> Int}. Dies
24 -- laesst sich auch in praefix-Notation schreiben \hsSource{(->) Int
25 -- Int} und ist damit analog zu den Typvariablen des \hsSource{Grid}
26 -- Types.
27
28 newtype Grid a b c = GR (a b c, CircuitDescriptor)
```

A.23. Circuit.Grid.Instance

```
1  -- Das Modul \hsSource{Circuit.Grid.Instance} beschreibt, wie die
2  -- Arrowinstanzen des Grid-Datentypes implementiert werden.
3
4  module Circuit.Grid.Instance
5  where
6
7
```

```haskell
-- Folgenden Module werden benoetigt, um die Arrows definieren zu
-- koennen:

import Circuit.Grid.Datatype

import Circuit.Descriptor
import Circuit.Graphs
import Circuit.Workers (flatten)
import Circuit.ShowType

import Prelude hiding (id, (.))
import qualified Prelude as Pr

import Control.Category

import Circuit.Arrow

import Circuit.Splice


-- Bevor fuer den Typ \hsSource{Grid} eine Arrowinstanz implementiert
-- werden kann, muss \hsSource{Grid} Mitglied der Typklasse
-- \hsSource{Category} sein.

instance (Category a) => Category (Grid a) where
    id
      = id

    GR (f, cd_f) . GR (g, cd_g)
      = GR (f . g, cd_g `connect` cd_f)


-- Im naechsten Schritt wird die Arrowinstanz von \hsSource{Grid}
-- implementiert. Laut Definition ist ein Arrow vollstaendig definiert
-- durch die Funktionen \hsSource{arr} und \hsSource{first}. Alle
-- weiteren Funktion lassen sich aus diesen beiden ableiten. Da hier
-- aber die Kontrolle ueber die Implementierung jeder Funktion behalten
-- werden soll, ist eine Implementierung fuer alle Einzelfunktionen
-- gegeben.

instance (Arrow a) => Arrow (Grid a) where
    arr   f
      = error "Can't construct arbitarly Hardware-Components"

```

```haskell
    first (GR (f, cd_f))
      = GR ( first f
           , cd_f `combine` idCircuit
           )

    second (GR (g, cd_g))
      = GR ( second g
           , idCircuit `combine` cd_g
           )

    GR (f, cd_f) &&& GR (g, cd_g)
      = GR ( f &&& g
           , cd_f `combine` cd_g
           )

    GR (f, cd_f) *** GR (g, cd_g)
      = GR ( f *** g
           , cd_f `combine` cd_g
           )

-- Die Definition von \hsSource{ArrowLoop} ist dann notwendig, wenn
-- Schleifen abgebildet werden sollen. Hierzu ist die Implementation
-- einer einzigen Funktion notwendig, naemlich der \hsSource{loop :: a
-- (b, d) (c, d) -> a b c} notwendig.

instance (ArrowLoop a) => ArrowLoop (Grid a) where
    loop (GR (f, cd_f)) = GR (loop f, loopWithRegister cd_f)


instance (ArrowCircuit a) => ArrowCircuit (Grid a) where
    delay x = GR (delay x, mkRegister (MkNode "" 0 (mkPins 1) (mkPins 1)))


-- Zu dem \hsSource{Grid}-Arrow gehoert ausserdem noch eine Funktion,
-- die den \hsSource{Grid}-Typ auspacken kann und dann ``ausfuehren''
-- kann.

runGrid :: (Arrow a) => Grid a b c -> a b c
runGrid (GR (f, _)) = f
```

A.24. Circuit.Grid

```haskell
-- Das Modul \hsSource{Circuit.Grid} stellt ein Wrapper-Modul nach dem
-- Fassaden Entwurfsmuster dar.

module Circuit.Grid
    ( Grid(..)
    , runGrid
    )
where


-- Folgenden Module werden benoe!tigt, um die Arrows definieren zu koennen:

import Circuit.Grid.Datatype
import Circuit.Grid.Instance
```

A.25. Circuit.Show.DOT

```haskell
module Circuit.Show.DOT
( showCircuit
, showEdge
)
where

import Data.Maybe ( isJust )
import Data.List ( nub
                 , (\\)
                 )

import Prelude hiding ( break )

import Circuit.Descriptor

import Circuit.PinTransit
import Circuit.EdgeTransit

import Circuit.Show.Tools


```

```
22 -- This function produces the edge-description as it is required by the
23 -- dot language... something like this:
24 --     nodeId3:op0 -nodeId6:ip0
25
26 showEdge :: Edge -> String
27 showEdge (MkEdge (Nothing, pid) (Just snk_cid, snk_pid))
28     =  "xSTART" ++ ':':"op" ++ show pid
29     ++ " -> "
30     ++ "nodeId" ++ show snk_cid ++ ':': "ip" ++ show snk_pid
31 showEdge (MkEdge (Just src_cid, src_pid) (Nothing, pid))
32     =  "nodeId" ++ show src_cid ++ ':': "op" ++ show src_pid
33     ++ " -> "
34     ++ "xEND" ++ ':':"ip" ++ show pid
35 showEdge e
36     =  "nodeId" ++ show src_cid ++ ':': "op" ++ show src_pid
37     ++ " -> "
38     ++ "nodeId" ++ show snk_cid ++ ':': "ip" ++ show snk_pid
39     where (Just src_cid, src_pid) = sourceInfo e
40           (Just snk_cid, snk_pid) = sinkInfo e
41
42
43 showCircuit :: CircuitDescriptor -> String
44 showCircuit g
45      = concat $ map break
46      [ ""
47      , "digraph G {"
48      , dot_config
49      , dot_outer_nodes g
50      , dot_components  g
51      , dot_connections g
52      , "}"
53      ]
54      where namedEdges = generateNamedEdges g
55            namedComps = generateNamedComps g
56
57 dot_config :: String
58 dot_config
59      = concat $ map break
60      [ ""
61      , "graph ["
62      , "    rankdir = \"LR\""
63      , "]"
64      ]
65
```

```
66  dot_outer_nodes :: CircuitDescriptor -> String
67  dot_outer_nodes g
68       = concat $ map break
69       [ ""
70       , "xSTART ["
71       , "    " ++ dot_outer_label "op" (sinks.nodeDesc $ g)
72       , "    " ++ "shape = \"record\""
73       , "]"
74       , ""
75       , "xEND ["
76       , "    " ++ dot_outer_label "ip" (sources.nodeDesc $ g)
77       , "    " ++ "shape = \"record\""
78       , "]"
79       ]
80
81  dot_components :: CircuitDescriptor -> String
82  dot_components  g
83       = concat $ nub $ map f (nodes g)
84       where f g' = concat $ map break
85                  [ ""
86                  , "nodeId" ++ show (nodeId.nodeDesc $ g') ++ " ["
87                  , "    " ++ dot_label (sinks.nodeDesc $ g')
88                                        (  map (\x -> if x == '>' then '-' else x)
89                                         $ label.nodeDesc $ g')
90                                        (nodeId.nodeDesc $ g')
91                                        (sources.nodeDesc $ g')
92                  , "    " ++ "shape = \"record\""
93                  , "]"
94                  ]
95
96  dot_outer_label :: String -> Pins -> String
97  dot_outer_label s ps
98       =  "label = \" {{ | "
99       ++ (concat $ map (f s) ps)
100      ++ "}}\""
101      where f :: String -> Int -> String
102            f s x = "<" ++ s ++ show x ++ "> (" ++ show x ++ ") | "
103
104 dot_label :: Pins -> String -> CompID -> Pins -> String
105 dot_label ips nme cid ops
106      =  "label = \" {{ | "
107      ++ (concat $ map (f "ip") ips)
108      ++ "} | { Name: " ++ nme ++ " | (" ++ show cid ++ ")} | { | "
109      ++ (concat $ map (f "op") ops)
```

```
110       ++ "}}\""
111       where f :: String -> Int -> String
112             f s x = "<" ++ s ++ show x ++ "> (" ++ show x ++ ") | "
113
114
115 dot_connections :: CircuitDescriptor -> String
116 dot_connections g
117       = concat $ map (\x -> showEdge x ++ "\n") (edges g)
```

A.26. Circuit.Show.Simple

```
1  module Circuit.Show.Simple
2  ( showCircuit
3  , showEdge
4  )
5  where
6
7  import Circuit.Descriptor
8
9  showCircuit :: CircuitDescriptor -> String
10 showCircuit g =  showCircuit' g
11               ++ "\n" ++ "Area: " ++ (show $ space g)
12               ++ "\n" ++ "Time: " ++ (show $ cycles g)
13
14 showCircuit' :: CircuitDescriptor -> String
15 showCircuit' g =  "\n"
16             ++ (show.nodeId.nodeDesc) g
17             ++ "(" ++ (show.label.nodeDesc) g ++ "): "
18             ++ (prtInOuts.sinks.nodeDesc) g ++ "] "
19             ++ (showEdges.edges) g
20             ++ " [" ++ (prtInOuts.sources.nodeDesc) g
21             ++ (showNode.nodes) g
22          where showNode [] = ""
23                showNode n  = concat $ map showCircuit' n
24                prtInOuts [] = "_"
25                prtInOuts x  = foldl1 (\x y -> x ++ ',':y) $ map show x
26
27
28 showEdge :: Edge -> String
29 showEdge ed
```

```
30    = (prtConnection.sourceInfo) ed ++ "->" ++ (prtConnection.sinkInfo) ed
31        where prtConnection (Just cid, pid) = show (cid, pid)
32              prtConnection (Nothing,  pid) = "(_," ++ show pid ++ ")"
33
34
35  showEdges :: [Edge] -> String
36  showEdges = concat . (map showEdge)
```

A.27. Circuit.Show.Tools

```
1   module Circuit.Show.Tools
2   where
3
4   -- break is a function, that appends a newline character to the
5   -- end of a string.
6
7   break :: String -> String
8   break = flip (++) "\n"
```

A.28. Circuit.Show.VHDL

```
1   module Circuit.Show.VHDL
2   ( showCircuit
3   , showEdge
4   )
5   where
6
7   import Data.Maybe ( isJust )
8   import Data.List ( nub
9                    , (\\)
10                   )
11
12  import Prelude hiding ( break )
13
14  import Circuit.Descriptor
15
16  import Circuit.PinTransit
17  import Circuit.EdgeTransit
```

```
18 import Circuit.Tests
19
20 import Circuit.Show.Tools
21
22
23 -- The showEdge function is only needed for the Show class.  While
24 -- VHDL-Code is generated, of this function is made no use ...
25
26 showEdge :: Edge -> String
27 showEdge ed
28   = (prtConnection.sourceInfo) ed ++ "->" ++ (prtConnection.sinkInfo) ed
29       where prtConnection (Just cid, pid) = show (cid, pid)
30             prtConnection (Nothing,  pid) = "(_," ++ show pid ++ ")"
31
32
33 -- In a VHDL-Sorce file, there are two main sections, that we need to
34 -- specify in order to get working VHDL-Source.
35 --
36 -- The function that starts the show-process is the
37 -- toVHDL-function. Here we define the basic structure of a
38 -- VHDL-SourceCode with an header, the entity-definition as well as
39 -- the component-definition.
40
41 showCircuit :: CircuitDescriptor -> String
42 showCircuit g
43      = concat $ map break
44      [ ""
45      , vhdl_header
46      , vhdl_entity       g namedComps
47      , vhdl_architecture g
48      , vhdl_components   g namedComps
49      , vhdl_signals      g namedEdges
50      , vhdl_portmaps     g namedComps namedEdges
51      ]
52      where namedEdges = generateNamedEdges g
53            namedComps = generateNamedComps g
54
55 nameEdges :: String -> CircuitDescriptor -> [([Anchor], String)]
56 nameEdges pre g
57     =   map (\(i, e) -> (sourceInfo e : sinkInfo e : [], pre ++ show i))
58       $ zip [0..] relevantEdges
59     where
60       relevantEdges
61         = filter (\(MkEdge (ci,_) (co,_)) -> isJust ci && isJust co) $ edges g
```

```
62
63  nameGraphPins :: CircuitDescriptor
64                -> [(CompID, ([(PinID, String)], [(PinID, String)]))]
65  nameGraphPins g = nameSuperPins g : (map nameSubPins $ nodes g)
66
67  nameSuperPins :: CircuitDescriptor
68                -> (CompID, ([(PinID, String)], [(PinID, String)]))
69  nameSuperPins g = (nodeId.nodeDesc $ g, (namedSinks, namedSources))
70      where namedSinks   = namePins' (sinks.nodeDesc)   nameExI g
71            namedSources = namePins' (sources.nodeDesc) nameExO g
72
73  nameSubPins :: CircuitDescriptor
74              -> (CompID, ([(PinID, String)], [(PinID, String)]))
75  nameSubPins g = (nodeId.nodeDesc $ g, (namedSinks, namedSources))
76      where namedSinks   = namePins' (sinks.nodeDesc)   nameInI g
77            namedSources = namePins' (sources.nodeDesc) nameInO g
78
79  namePins' :: (CircuitDescriptor -> Pins)
80            -> String -> CircuitDescriptor -> [(PinID, String)]
81  namePins' f pre g = map (\x -> (x, pre ++ show x)) $ f g
82
83  -- The VHDL-Header is just some boilerplate-code where library's are
84  -- imported
85
86  vhdl_header :: String
87  vhdl_header
88       = concat $ map break
89       [ "LIBRARY ieee;"
90       , "USE ieee.std_logic_1164.all;"
91       ]
92
93  -- A VHDL-Entity defines an "interface" to a hardware component. It
94  -- consists of a name and of some port-definitions (like what wires go
95  -- inside and come back out)
96
97  vhdl_entity :: CircuitDescriptor -> [NamedComp] -> String
98  vhdl_entity g namedComps
99       = concat $ map break
100      [ "ENTITY " ++ (label.nodeDesc) g ++ " IS"
101      , "PORT ("
102      , (sepBy "\n" $ map (\x -> x ++ " : IN  std_logic;") $ snks)
103      , (sepBy "\n" $ map (\x -> x ++ " : OUT std_logic ") $ srcs)
104      , ");"
105      , "END " ++ (label.nodeDesc) g ++ ";"
```

```
106       ]
107       where snks = getInPinNames  namedComps (nodeId.nodeDesc $ g)
108             srcs = getOutPinNames namedComps (nodeId.nodeDesc $ g)
109
110 vhdl_architecture :: CircuitDescriptor -> String
111 vhdl_architecture g
112     =     "ARCHITECTURE "
113       ++ (label.nodeDesc $ g)
114       ++ "Struct OF "
115       ++ (label.nodeDesc $ g)
116       ++ " IS"
117
118
119 -- The VHDL-Component definitions describe the basic interface to the components
120 -- that are used inside this new definition. We therefore pick the components
121 -- of which these new component consists. We call this components the level 1
122 -- components, because we descent only one step down in the graph.
123
124 vhdl_components :: CircuitDescriptor -> [NamedComp] -> String
125 vhdl_components g namedComps
126      = concat $ nub $ map f (nodes g)
127     where f g' = concat $ map break
128                  [ ""
129                  , "COMPONENT " ++ (label.nodeDesc $ g') ++ "Comp"
130                  , "PORT ("
131                  , (sepBy "\n" $ map (\x -> x ++ " : IN  std_logic;") $ snks)
132                  , (sepBy "\n" $ map (\x -> x ++ " : OUT std_logic ") $ srcs)
133                  , ");"
134                  , "END COMPONENT " ++ (label.nodeDesc $ g') ++ "Comp;"
135                  ]
136                  where snks = getInPinNames  namedComps (nodeId.nodeDesc $ g')
137                        srcs = getOutPinNames namedComps (nodeId.nodeDesc $ g')
138
139
140 -- The VHDL-Signals is the list of inner wires, that are used inside
141 -- the new component.
142
143 vhdl_signals :: CircuitDescriptor -> [([Anchor], String)] -> String
144 vhdl_signals _ [] = ""
145 vhdl_signals g namedEdges
146      = "SIGNAL " ++ sepBy ", " signals ++ ": std_logic;"
147      where signals = map snd namedEdges
148
149
```

```
150 vhdl_portmaps :: CircuitDescriptor
151               -> [NamedComp] -> [([Anchor], String)] -> String
152 vhdl_portmaps g namedComps namedEdges
153      = concat $ map break
154      [ "BEGIN"
155      , concat $ map (vhdl_portmap g namedComps namedEdges) $ nodes g
156      , "END;"
157      ]
158
159 vhdl_portmap :: CircuitDescriptor -> [NamedComp] -> [([Anchor], String)]
160              -> CircuitDescriptor -> String
161 vhdl_portmap superG namedComps namedEdges' g
162      = concat $ map break
163      [    (label.nodeDesc $ g)
164        ++ "Inst"
165        ++ (show.nodeId.nodeDesc $ g)
166        ++ ": "
167        ++ (label.nodeDesc $ g)
168        ++ "Comp"
169      , "PORT MAP ("
170      ++ (sepBy ", " $ filter ((>0).length) [incoming, signaling, outgoing])
171      ++ ");"
172      ]
173      where
174        relevantEdges
175          = filter (isFromOrToComp $ nodeId.nodeDesc $ g) $ edges superG
176        edge2inside
177          = filter (isFromOuter)   $ relevantEdges
178        edge2outside
179          = filter (isToOuter)     $ relevantEdges
180        pin2signal
181          = relevantEdges \\ (edge2outside ++ edge2inside)
182        incoming
183          =   sepBy ", "
184            $ map (genPortMap namedComps namedEdges' (nodeId.nodeDesc $ g))
185            $ edge2inside
186        outgoing
187          =   sepBy ", "
188            $ map (genPortMap namedComps namedEdges' (nodeId.nodeDesc $ g))
189            $ edge2outside
190        signaling
191          =   sepBy ", "
192            $ map (genPortMap namedComps namedEdges' (nodeId.nodeDesc $ g))
193            $ pin2signal
```

```haskell

genPortMap :: [NamedComp] -> [NamedEdge] -> CompID -> Edge -> String

-- From the inner component to the outside
--  : PORT MAP (a0 =out0, a1 => out1);
--               +--------+
--               |  pi = [0] -
--               |ci = 0  |
--               |  pi = [1] -
--               +--------+

genPortMap namedComps _ _ (MkEdge (Just ci, pi) (Nothing, po))
    = pinName ++ " => " ++ outName
    where pinName  = getOutPinName namedComps ci       pi
          outName  = getOutPinName namedComps superCid po
          superCid = fst . head $ namedComps


-- From the outside to the inner component
--  : PORT MAP (e0 =in0, e1 => in1);
--                 +--------+
--             -[0] = po  |
--                 |co = 0  |
--             -[1] = po  |
--                 +--------+

genPortMap namedComps _ _ (MkEdge (Nothing, pi) (Just co, po))
    = pinName ++ " => " ++ incName
    where pinName  = getInPinName namedComps co       po
          incName  = getInPinName namedComps superCid pi
          superCid = fst . head $ namedComps


-- From the inner component to an inner signal
--  : PORT MAP (a0 =i0, a0 => i1);
--   +--------+                       +--------+
--   |  pi = [0] - -----i0------ -> [0] = po  |
--   |ci = 0  |                       |co = 1  |
--   |  pi = [1] - ------i1----- -> [1] = po  |
--   +--------+                       +--------+

genPortMap namedComps namedEdges ownID (MkEdge ie@(Just ci, pi) oe@(Just co, po))
 | ownID == ci = iPinName ++ " => " ++ iSigName
 | ownID == co = oPinName ++ " => " ++ oSigName
```

```
238       where iPinName = getOutPinName  namedComps ci pi
239             oPinName = getInPinName namedComps co po
240             iSigName = getEdgeName namedEdges ie
241             oSigName = getEdgeName namedEdges oe
242
243
244
245
246 -- In the last genPortMap function there are some irregularities
247
248 -- The namePins function takes a function that extracts a list of
249 -- PinIDs out of an StructGraph.  (This could be the sinks or the
250 -- sources functions) It also takes a StructGraph (suprise :)) and a
251 -- String, that is prepended to the actual PinName.  This functions
252 -- returns a list, where every element is a tuple of the actual named
253 -- pin (a string) and a part, that identifies the name.
254
255 namePins :: (CircuitDescriptor -> Pins) -> String
256          -> CircuitDescriptor -> [(String, Anchor)]
257 namePins f pre g
258     = map (\x -> (pre ++ (show x), (Nothing, x))) $ f g
259
260
261 -- The nameEdges function is pretty similar to the namePins function
262 -- with some minor differences.  First of all, you don't need a
263 -- function that extracts the edges of a StructGraph. There is only
264 -- one field in the StructGraph that holds the edges.  And also the
265 -- return-type is a bit simpler, becaus an edge identifies itself, so
266 -- there is no need to do this once more.
267
268 sepBy :: String -> [String] -> String
269 sepBy sep []     = ""
270 sepBy sep (x:[]) = x
271 sepBy sep xs     = foldl1 (\x y -> x ++ sep ++ y) xs
```

A.29. Circuit.Show

```
1 -- Dieses Modul stellt die Schnittstelle fuer die Show-Funktionalitaet
2 -- dar. Wird \hsSource{Circuit.Show} in einem anderen Modul
3 -- eingebunden, so koennen \hsSource{CircuitDescriptor}en mittels
4 -- \hsSource{show} angezeigt werden.
```

```
5
6  module Circuit.Show
7  where
8
9  import Circuit.Descriptor
10 -- import Circuit.Show.Simple
11 -- import Circuit.Show.VHDL
12 import Circuit.Show.DOT
13 import qualified Circuit.Show.Simple as Simple
14 import qualified Circuit.Show.DOT as DOT
15 import qualified Circuit.Show.VHDL as VHDL
16
17 -- An externen Modulen wird hier lediglich die Kerndefinition
18 -- \hsSource{Circuit.Descriptor} verwendet. Darueber hinaus werden die
19 -- jeweiligen Anzeigemodule eingebunden. Folgende Formate koennen als
20 -- Ausgabeformat gewaehlt werden: \begin{itemize} \item Mit
21 -- \begriff{Simple} ist ein Ausgabeformat gemeint, welches sehr kurz
22 -- und praegnant alle verfuegbaren Informationen anzeigt. Dieses Format
23 -- eignet sich besonders gut, um damit debug-Informationen auszugeben.
24 -- \item Die \begriff{DOT}-Sprache ist eine Beschreibungssprache, die
25 -- verwendet wird, um Graphen abzubilden. Es gibt eine ganze Reihe von
26 -- Werkzeugen \footnote{Fuer Linux sei hier \begriff{graphviz}
27 -- genannt}, die dann aus einer \texttt{.dot}-Datei eine Grafik
28 -- erzeugen koennen, die den Graphen abbildet.  \item Auch das erzeugen
29 -- von \begriff{VHDL} ist eine Ausgabeform, die ueber Haskells
30 -- \hsSource{Show}-Methoden abgebildet wird.  \end{itemize}
31 --
32 -- Um Haskells \hsSource{Show}-Klasse verwenden zu koennen, muss der
33 -- Typ \hsSource{CircuitDescriptor} Mitglied der Klasse
34 -- \hsSource{Show} sein.  Da die eigentliche Definition der
35 -- \hsSource{Show}-Methode in einem Untermodul stattfindet, ist die
36 -- Instanzdefinition einfach. Es wird lediglich der
37 -- \hsSource{show}-Methode eine der vorhandenen Methoden zugeordnet.
38 --
39
40 instance Show (Edge) where
41     show = showEdge
42
43 instance Show (CircuitDescriptor) where
44     show = showCircuit
```

A.30. Circuit.ShowType.Class

```
1  -- Das Modul \hsSource{Circuit.ShowType.Class} beschreibt, wie die
2  -- Arrow-Klasse zu implementieren sind um damit spaeter Schaltkreise
3  -- beschreiben, bearbeiten oder benutzen zu koennen.
4
5  module Circuit.ShowType.Class
6  where
7
8
9  -- Folgenden Module werden benoetigt, um die Arrows definieren zu
10 -- koennen:
11
12 import Prelude hiding (id, (.))
13 import qualified Prelude as Pr
14
15 import Control.Category
16
17 import Circuit.Descriptor
18
19
20 -- Zunaechst einmal muessen die Klassen definiert werden. Diese Vorgaben
21 -- muessen von allen Instanzdefinitionen befolgt werden.
22
23 -- Die \hsSource{ShowType}-Klasse wird benoetigt, um zur Laufzeit
24 -- Typinformationen in den \hsSource{Arrow} zu bekommen. Dies wird
25 -- immer dann ausgenutzt, wenn mittels \hsSource{arr} eine Funktion in
26 -- einen Arrow geliftet wird. Wie man spaeter in der
27 -- \hsSource{Arrow}-Klassen Definition erkennen kann, ist
28 -- \hsSource{arr} nur fuer Funktionen definiert, die auch in
29 -- \hsSource{ShowType} sind.
30
31 class ShowType b c where
32     showType :: (b -> c) -> CircuitDescriptor
```

A.31. Circuit.ShowType.Instance

```
1  -- Das Modul \hsSource{Circuit.ShowType.Instance} beschreibt, wie die
2  -- Arrow-Klasse zu implementieren sind um damit spaeter Schaltkreise
3  -- beschreiben, bearbeiten oder benutzen zu koennen.
```

```haskell

module Circuit.ShowType.Instance
where


-- Folgenden Module werden benoetigt, um die Instanzen definieren zu
-- koennen:

import Circuit.ShowType.Class

import Prelude hiding (id, (.))
import qualified Prelude as Pr

import Control.Category

import Circuit.Graphs
import Circuit.Descriptor

-- Showtype wird hier fuer bestimmte Tupel-Typen beschrieben. Einzig
-- die Tupel-Information ist das, was mittels Showtype in
-- Pin-Informationen uebersetzt wird.


-- Die \hsSource{ShowType}-Instanz definiert die verschiedenen
-- Typenvarianten, welche abbildbar sind.

-- Mit dem Typ \hsSource{(b, c) -> (c, b)} stellt die folgende
-- Variante eine dar, in der die Ein- und Ausgaenge miteinander
-- vertauscht werden.

instance ShowType (b, c) (c, b) where
    showType _
        = emptyCircuit { nodeDesc = MkNode
                                  { label   = "|b,c>c,b|"
                                  , nodeId  = 0
                                  , sinks   = mkPins 2
                                  , sources = mkPins 2
                                  }
                       }
        where nd = nodeDesc emptyCircuit


-- Diese Variante mit dem Typ \hsSource{b -> (b, b)} beschreibt den
-- Fall, in dem ein Eingang verdoppelt wird.
```

```
48
49  instance ShowType b (b, b) where
50      showType _
51          = emptyCircuit { nodeDesc = MkNode
52                                 { label   = "|b>b,b|"
53                                 , nodeId  = 0
54                                 , sinks   = mkPins 1
55                                 , sources = mkPins 2
56                                 }
57                         }
58          where nd = nodeDesc emptyCircuit
59
60
61  -- Mit dem Typ \hsSource{(b, b) -> b} wir dann der Fall abgebildet,
62  -- der zwei Eingaenge auf einen zusammenfasst.
63
64  instance ShowType (b, b) b where
65      showType _
66          = emptyCircuit { nodeDesc = MkNode
67                                 { label   = "|b,b>b|"
68                                 , nodeId  = 0
69                                 , sinks   = mkPins 2
70                                 , sources = mkPins 1
71                                 }
72                         }
73          where nd = nodeDesc emptyCircuit
74
75  -- instance ShowType (b, c) (b', c') where
76  --   showType _ = emptyCircuit { label = "b,c>b',c'"
77  --                             , sinks = mkPins 1
78  --                             , sources = mkPins 1
79  --                             }
80  --
81  -- instance ShowType b b where
82  --  showType _ = emptyCircuit { label = "|b>b|"
83  --                            , sinks = mkPins 1
84  --                            , sources = mkPins 1
85  --                            }
86  --
87  -- instance ShowType (b -> (c, d)) where
88  --   showType _ = emptyCircuit { label = "b>c,d"
89  --                             , sinks = mkPins 1
90  --                             , sources = mkPins 1
91  --                             }
```

```haskell
--
-- instance ShowType ((b, c) -> d) where
--   showType _ = emptyCircuit { label = "b,c>d"
--                             , sinks = mkPins 1
--                             , sources = mkPins 1
--                             }

-- Letztlich bleibt noch der allgemeinste Fall der Moeglich ist. Diese
-- Varianten ist somit auch eine \begriff{CatchAll} Variante.

instance ShowType b c where
    showType _
        = emptyCircuit { nodeDesc = MkNode
                                  { label   = "|b>c|"
                                  , nodeId  = 0
                                  , sinks   = mkPins 1
                                  , sources = mkPins 1
                                  }
                       }
        where nd = nodeDesc emptyCircuit
```

A.32. Circuit.ShowType

```haskell
-- Das Modul \hsSource{Circuit.ShowType} ist ein strukturelles Modul
-- nach dem Fassaden-Entwurfsmuster.

module Circuit.ShowType (ShowType(..))
where


-- Folgenden Module werden benoetigt, um die Instanzen definieren zu
-- koennen:

import Circuit.ShowType.Class (ShowType(..))
import Circuit.ShowType.Instance
```

A.33. Circuit.Stream.Datatype

```
1  -- Das Modul \hsSource{Circuit.Stream.Datatype} beschreibt, was ein
2  -- Stream-Datentyp ist.
3
4  module Circuit.Stream.Datatype
5  where
6
7
8  -- In Schaltkreisen sind Schleifen nicht ueber den normalen
9  -- \hsSource{ArrowLoop} Ansatz realisierbar. Das Problem liegt darin,
10 -- dass zwischen zwei Berechnungen keine Verzoegerung
11 -- stattfindet. Wollte man das erreichen, so benoetigt man mindestens
12 -- \begriff{Register}, die einen Taktzyklus verzoegern. Damit ist dann
13 -- festgelegt, dass Schleifen in Hardware nur Sinn
14 -- ergeben, wenn ein kontinuierlicher Datenstrom verarbeitet werden
15 -- kann.
16
17 -- Auch fuer andere Ansaetze wird ein \hsSource{Stream}-Arrow
18 -- notwendig. Dies kann beispielsweise der Fall sein, wenn
19 -- Zwischenergebnisse innerhalb des Schaltkreises ermittelt werden
20 -- sollen.
21
22 -- Zunaechst wird der Datentyp definiert. \hsSource{Stream} hat einen
23 -- Typkonstruktor \hsSource{SF} und besteht aus einer Funktion, welche
24 -- aus einer Liste von b's (\hsSource{[b]}) eine Liste von c's
25 -- \hsSource{[c]}) erzeugt.
26
27 newtype Stream b c = SF { runStream :: ([b] -> [c]) }
```

A.34. Circuit.Stream.Instance

```
1  -- Das Modul \hsSource{Circuit.Stream.Instance} beschreibt, wie die
2  -- Arrow-Klasse zu implementieren sind, um damit spaeter Schaltkreise
3  -- beschreiben, bearbeiten oder benutzen zu koennen.
4
5  module Circuit.Stream.Instance
6  where
7
8
```

```haskell
-- Folgende Module werden benoetigt, um Arrows definieren zu koennen:

import Prelude hiding (id, (.))
import qualified Prelude as Pr

import Control.Category

import Circuit.Arrow

import Circuit.Descriptor
import Circuit.Graphs
import Circuit.Workers (flatten)

import Circuit.Stream.Datatype



-- Im naechsten Schritt wird \hsSource{Stream} zu einer Kategorie
-- ernannt, indem die \hsSource{Category}-Typklasse fuer
-- \hsSource{Stream} implementiert wird. Erst wenn \hsSource{Stream}
-- eine Kategorie ist, laesst sich \hsSource{Stream} in einen Arrow
-- befoerdern.

instance Category Stream where
    id
      = SF (id)

    (SF f) . (SF g)
      = SF (f . g)


-- Nachdem \hsSource{Stream} eine Kategorie ist, kann
-- \hsSource{Stream} als Arrow implementiert werden. AEhnlich wie bei
-- der Implementierung der Arrowinstanz von \hsSource{Grid} ist es
-- auch bei \hsSource{Stream} notwendig, alle Funktionsbeschreibungen
-- anzugeben. Die abgeleiteten Funktionen der Minimaldefinition,
-- reichen nicht aus.

instance Arrow Stream where
    arr f
      = SF $ map f

    first  (SF f)
      = SF $ (uncurry zip) . (\(bs, cs) -> (f bs, cs)) . unzip
```

```
53
54      second (SF g)
55        = SF $ (uncurry zip) . (\(bs, cs) -> (bs,  g cs)) . unzip
56
57      (SF f) *** (SF g)
58        = SF $ (uncurry zip) . (\(bs, cs) -> (f bs, g cs)) . unzip
59
60
61  -- Da \hsSource{Stream} ein Arrow ist und kontinuierliche Datenstroeme
62  -- fuer das Looping notwendig sind, kann fuer \hsSource{Stream} die
63  -- \hsSource{ArrowLoop} Instanz angegeben werden.
64
65  instance ArrowLoop Stream where
66      loop (SF f)
67        = SF $ (\bs ->
68                  let (cs, ds) = unzip . f $ zip bs (stream ds)
69                  in  cs
70                )
71        where stream ~(x:xs) = x:stream xs
72
73
74  -- Mit der \hsSource{Stream} Instanz von \hsSource{ArrowLoop} ist es
75  -- nun moeglich, die \hsSource{ArrowCircuit} Instanz zu
76  -- implementieren.  Diese ist eine direkte Umsetzung des
77  -- \hsSource{delay}'s auf die Listenfunktionalitaet.
78
79  instance ArrowCircuit Stream where
80      delay x = SF (x:)
```

A.35. Circuit.Stream

```
1   -- Das Modul \hsSource{Circuit.Stream} stellt ein Wrappermodul nach
2   -- dem Fassadenentwurfsmuster dar.
3
4   module Circuit.Stream
5       ( Stream(..)
6       )
7   where
8
9
10  -- Folgende Module werden benoetigt, um Arrows definieren zu koennen:
```

```
11
12 import Circuit.Stream.Datatype
13 import Circuit.Stream.Instance
```

Abbildungsverzeichnis

Literaturverzeichnis

[1] Robert Atkey. What is a Categorical Model of Arrows? In MSFP2008. ACM, 2008.

[2] AT&T. Graphviz – Graph Visualization Software.

[3] Emil Axelsson, Koen Classen, and Mary Sheeran. Using Lava and Wired for Design Exploration. In Designing Correct Circuits Workshop; A Satellite Event of ETAPS 2006, Wien, Maerz 2006.

[4] Christiaan Baaij. VHDL: VHDL AST and Pretty Printer. `http://hackage.haskell.org/package/vhdl-0.1.2.1`, June 2010.

[5] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware Design in Haskell. In In International Conference on Functional Programming, pages 174–184. ACM Press, 1998.

[6] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in haskell. ACM Press, 1998.

[7] Richard Blute, J.R.B. Cockett, and R.A.G. Seely. Categories for Computation in Context and Unified Logic. Journal of Pure and Applied Algebra, 116:49–98, 1997.

[8] Matthias Brettschneider. ArrowVHDL: A Library to Generate Netlist Code from Arrow Descriptions, April 2012.

[9] Matthias Brettschneider. ArrowVHDL: A Library to Generate Netlist Code from Arrow Descriptions. `http://hackage.haskell.org/package/ArrowVHDL-1.1`, April 2012.

[10] Matthias Brettschneider. ArrowVHDL: A Library to Generate Netlist Code from Arrow Descriptions, April 2012.

[11] Matthias Brettschneider and Tobias Häberlein. Von funktionalen Programmen zur Hardwarebeschreibung digitaler Filter. In Forum on specification & Design Languages, 2008.

[12] Matthias Brettschneider and Tobias Häberlein. Functional Abstractions for UML Activity Diagrams. In Forum on specification & Design Languages, 2010.

[13] Matthias Brettschneider and Tobias Häberlein. Using Haskell Arrows as Executable Hardware Specification Language. In Design of Circuits and Integrated Systems, 2011.

[14] Matthias Brettschneider and Tobias Häberlein. From Arrows to Netlists Describing Hardware. In Beniamino Murgante, Sanjay Misra, Maurizio Carlini, CarmeloM. Torre, Hong-Quang Nguyen, David Taniar, BernadyO. Apduhan, and Osvaldo Gervasi, editors, Computational Science and Its Applications – ICCSA 2013, volume 7973 of Lecture Notes in Computer Science, pages 128–143. Springer Berlin Heidelberg, 2013.

[15] Koen Claessen. An Embedded Language Approach to Hardware Description and Verification. Master's thesis, Göteborg University, 2000.

[16] Koen Claessen and John Hughes. Testing Monadic Code with QuickCheck. In In Proc. ACM SIGPLAN workshop on Haskell, pages 65–77, 2002.

[17] Koen Claessen and David Sands. Observable Sharing for Functional Circuit Description. In Proceedings of the 5th Asian Computing Science Conference on Advances in Computing Science, ASIAN '99, pages 62–73, London, UK, 1999. Springer-Verlag.

[18] Koen Claessen and Mary Sheeran. A Tutorial on Lava. Chalmers University of Technology.

[19] H. Curry. Functionality in Combinatorial Logic. In Proceedings of National Academy of Sciences, volume 20, pages 584–590, 1934.

[20] H. B. Curry and R. Feys. Combinatory Logic, Volume I. North-Holland, 1958. Second printing 1968.

[21] Zamira Daw and Marcus Vetter. Deterministic uml models for interconnected activities and state machines. In Andy Schürr and Bran Selic, editors, Model

Driven Engineering Languages and Systems, volume 5795 of Lecture Notes in Computer Science, pages 556–570. Springer Berlin Heidelberg, 2009.

[22] Jean-Guillaume Dumas, Dominique Duval, and Jean-Claude Reynaud. Cartesian Effect Categories are Freyd-Categories. J. Symb. Comput., 46(3):272–293, 2011.

[23] Galois, Inc., Portland, Oregon. Cryptol Programming Guide, October 2008. `http://www.galois.com/files/Cryptol/Cryptol_Programming_Guide.pdf`.

[24] Galois, Inc., Portland, Oregon. From Cryptol to FPGA, October 2008. `http://www.galois.com/files/Cryptol/Cryptol_Tutorial.pdf`.

[25] Jim Grundy, Tom Melham, and John O'Leary. A Reflective Functional Language for Hardware Design and Theorem Proving. Journal on Functional Programming, 16(2):157–196, 2006.

[26] Tobias Häberlein. Technische Informatik. Vieweg+Teubner, 2011.

[27] Tobias Häberlein and Matthias Brettschneider. Crypto-Core Design using Functional Programming. In Design of Circuits and Integrated Systems, Lanzarote, 2010.

[28] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type Classes in Haskell. ACM Trans. Program. Lang. Syst., 18(2):109–138, March 1996.

[29] Haskell Community. Monad Laws, 2012.

[30] Christoph Herrmann. Metaprogrammierung. University of Passau, 2005.

[31] W. A. Howard. The Formulae-as-Types Notion of Construction. In J. P. Seldin and J. R. Hindley, editors, To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, pages 479–490. Academic Press, Inc., New York, N.Y., 1980.

[32] John Hughes. Generalising Monads to Arrows. Science of Computer Programming, 37:67–111, May 2000.

[33] Bart Jacobs and Ichiro Hasuo. Freyd is Kleisli, for Arrows. In In C. McBride, T. Uustalu, Proc. of Wksh. on Mathematically Structured Programming, MSFP 2006, Electron. Wkshs. in Computing. BCS, 2006.

[34] John Kelsey, Bruce Schneier, and David Wagner. Key-Schedule Cryptanalysis of IDEA, G-DES, GOST, SAFER, and Triple-DES. In In Advances in Cryptology - CRYPTO '96, pages 23–7. Springer-Verlag, 1996.

[35] John Kelsey, Bruce Schneier, and David Wagner. Related-Key Cryptanalysis of 3-WAY, Biham-DES, CAST, DES-X, NewDES, RC2, and TEA. In DES, RC2, and TEA, Proceedings of the 1997 International Conference on Information and Communications Security, pages 233–246. Springer-Verlag, 1997.

[36] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly Typed Heterogeneous Collections. In Proceedings of the Haskell Workshop, Snowbird, Utah, 2004.

[37] Kofi Makinwa. Smart Sensors - No Signal to Small. In DCIS, 2011.

[38] Per Martin-Löf. Intuitionistic Type Theory. Bibliopolis, Napoli, 1984.

[39] Adam Megacz. Multi-Level Languages are Generalized Arrows, 2010.

[40] Alan Mycroft and Richard Sharp. The FLaSH Compiler: Efficient Ciruits from Functional Specifications. Technical report, University of Cambridge, Computing Laboratory, 2000.

[41] Alan Mycroft and Richard Sharp. Hardware/Software Co-Design using Functional Languages. In Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, pages 236–251, 2001.

[42] Flemming Nielson and Hanne Riis Nielson. Prescriptive Frameworks for Multi-Level Lambda-Calculi. In Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM '97, pages 193–202, New York, NY, USA, 1997. ACM.

[43] Ulf Norell. Towards a Practical Programming Language based on Dependent Type Theory, 2007.

[44] John O'Donnell. Hardware Description with Recursion Equations. In Proceedings of the IFIP 8th International Symposium on Computer Hardware

Description Languages and their Applications, pages 363–382, North-Holland, April 1987.

[45] John O'Donnell. Overview of Hydra: A Concurrent Language for Synchronous Digital Circuit Design. In IPDPS, 2002.

[46] John O'Donnell and Gudula Rünger. Functional Pearl: Derivation of a Carry Lookahead Addition Circuit. In R. Hinze, editor, Preliminary Proceedings of the ACM SIGPLAN Haskell Workshop (HW'2001), pages 1–31, New York, NY, USA, 2001. ACM.

[47] John T. O'Donnell. Hydra: Hardware Description in a Functional Language using Recursion Equations and High Order Combining Forms. In G.J. Milne, editor, The Fusion of Hardware Design and Verification, pages 309–328, Glasgow, Scotland, 1988. North-Holland.

[48] John T. O'Donnell. The Hydra Computer Hardware Description Language. University of Glasgow, 2003.

[49] Ross Paterson. A New Notation for Arrows. In ICFP, pages 229–240, 2001.

[50] Ross Paterson. Arrows and Computation. In Jeremy Gibbons and Oege de Moor, editors, The Fun of Programming, pages 201–222. Palgrave, 2003.

[51] John Power and Edmund Robinson. Premonoidal Categories and Notions of Computation. Mathematical Structures in Computer Science, 7(5):453–468, October 1997.

[52] Robert J. Irwin. Review of: "Derivation and Computation: Taking the Curry-Howard Correspondence Seriously by Harold Simmons", Cambridge University Press, 2000. SIGACT News, 39(2):42–44, June 2008.

[53] Ingo Sander. System Modeling and Design Refinement in ForSyDe. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, April 2003.

[54] Ingo Sander and Axel Jantsch. System modeling and transformational design refinement in ForSyDe. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 23(1):17–32, January 2004.

[55] Claude E. Shannon. Communication Theory of Secrecy Systems. Bell Systems Technical Journal, 28:656–715, 1949.

[56] Richard Sharp. Higher-Level Hardware Synthesis. Lecture Notes in Computer Science. Springer Verlag, 2004.

[57] Richard Sharp and Alan Mycroft. The FLaSH Compiler: Efficient Circuits from Functional Specifications. Technical report, AT&T Research Laboratories Cambridge, 2000.

[58] Tim Sheard and Simon Peyton Jones. Template Meta-Programming for Haskell. In Proceedings of the Haskell Workshop, Pittsburgh, 2002.

[59] Mary Sheeran. Mu FP-an Algebraic VLSI Design Language: Mary Sheeran. Oxford University Computing Laboratory, Programming Research Group, 1984.

[60] Harold Simmon. Derivation and Computation: Taking the Curry-Howard Correspondence Seriously. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, June 2000.

[61] Satnam Singh. The Lava Hardware Description Language. http://raintown.org/lava/.

[62] Satnam Singh. System Level Specification in Lava. In Proceedings of the conference on Design, Automation and Test in Europe - Volume 1, DATE '03, pages 10370–, Washington, DC, USA, 2003. IEEE Computer Society.

[63] Satnam Singh. Designing Reconfigurable Systems in Lava. In 17th International Conference on VLSI Design, Mumbai, India, January 2004. IEEE Computer Society.

[64] Zhendong Su and Gary Wassermann. The Essence of Command Injection Attacks in Web Applications. In POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 372–382, New York, NY, USA, 2006. ACM.

[65] Thomas Timmermann. Kategorientheorie. University Münster, April 2012.

[66] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving Haskell a Promotion. In Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation, TLDI '12, pages 53–66, New York, NY, USA, 2012. ACM.