Paradigmenübergreifende Quelltexttransformation von Haskell nach Ruby

Universität Stuttgart
Institut für Softwaretechnologie
Studienarbeit
2013

Jannik Zschiesche

Inhalt

1 Abkürzungsverzeichnis

AST	Abstracter Syntax Baum
GADT	Generalized Algebraic Data Type
GHC	Glasgow Haskell Compiler
OMG	Object Management Group
STG	Spineless Tagless G-Machine
UML	Unified Modeling Language

2 Abbildungsverzeichnis

3 Einleitung

In der Entwicklung der Programmiersprachen haben sich unterschiedliche Ideen über Programme und ihre verwendeten Stile entwickelt. Gewisse Stile und Konzepte werden zu grundlegenderen und größeren Programmierparadigmen zusammengefasst. Von den heute verwendeten Paradigmen sind das prozedurale, das funktionale und das objektorientierte Programmierparadigma die am weitesten verbreiteten.

Bei aktuellen Programmiersprachen gibt es den Trend, mehrere der Programmierparadigmen zu bündeln, so unterstützt beispielsweise C++ unter anderem die funktionale, imperative, objektorientierte, prozedurale, strukturierte und generische Programmierung. Der Compiler, und auch die Laufzeitumgebung müssen alle Konzepte dieser Paradigmen verstehen und unterstützen. Dies betrifft auch virtuelle Bytecode-Maschinen, wie die Java Virtual Machine, die mit Java eine objektorientierte und mit Clojure eine funktionale Programmiersprache unterstützt.

Diese Studienarbeit geht der Frage nach, inwiefern diese unterschiedlichen Konzepte mit einem Compiler[1] automatisiert ineinander überführt werden können, am Beispiel einer Kompilierung von der funktionalen Programmiersprache Haskell in das hauptsächlich objektorientierte Ruby. Auch wenn Ruby funktionale Programmierung unterstützt, werden für einige Haskell-Eigenschaften aufwändigere Transformationen benötigt.

Um diese Gemeinsamkeiten und Unterschiede zu finden werden zunächst von allen involvierten Programmiersprachen Metamodelle im für diese Studienarbeit vorgegebenen Umfang erstellt und dann aufeinander abgebildet. Anschließend wird auf einige Implementierungsdetails von dem im Zuge dieser Studienarbeit erstellten Compilers Haru[2] eingegangen.

Einige *Notationen* werden direkt aus der Sprache des GHC übernommen, um den Bruch beim Wechsel der Bezeichnungen möglichst gering zu halten:

- *Identifier*: ein Bezeichner, im Grunde ein Name mit einigen angehängten Daten.
- *Applikation*: eine Funktionsanwendung, ein Funktionsaufruf.
- *Lambda*: eine anonyme Funktion, also eine Funktion ohne Bezeichner.

4 Grundlagen

Bevor die Transformation und deren praktische Umsetzung in einem Programm genauer beschrieben werden, müssen Details zu den verwendeten Ansätzen, Sprachen und Technologien eingeführt werden.

4.1 Sprachkonzepte und Paradigmen[3]

Programmiersprachen entstehen als eine Sammlung von Konzepten, die entweder bereits bekannt und erforscht sind, oder durch die Programmiersprache geprägt werden sollen. Diese Konzepte

[1] Compiler zwischen zwei (Hoch-) Sprachen werden auch „Transcompiler“ genannt. Im Zuge dieser Arbeit werden diese Programme jedoch auch vereinfacht als „Compiler“ bezeichnet.
[2] Haru aus „Haskell to Ruby“
[3] Nach [10]

beeinflussen das Ausführungsmodell und die Elemente innerhalb der Sprache. Sie beschreiben statische und dynamische Eigenschaften von Programmiersprachen, beispielsweise über den Kontrollfluss, Variablen oder Objekte. Die Menge dieser Eigenschaften charakterisiert eine Programmiersprache zusätzlich zu möglichen syntaktischen Neuerungen beziehungsweise Änderungen.

Ein Programmierparadigma stellt eine Art Programmierstil da, der aus einer Menge von Sprachkonzepten definiert ist. Die einzelnen Programmierparadigmen sind allerdings nicht disjunkt definiert, so gibt es viele Programmiersprachen, die mehrere Programmiersprachen unterstützen, wie C++ (siehe unten) oder auch Ruby (unter anderem imperativ, objektorientiert, prozedural, funktional, nebenläufig).

Eine der wichtigsten Unterscheidungen innerhalb der Programmierparadigmen ist zwischen der deklarativen und imperativen Programmierung.

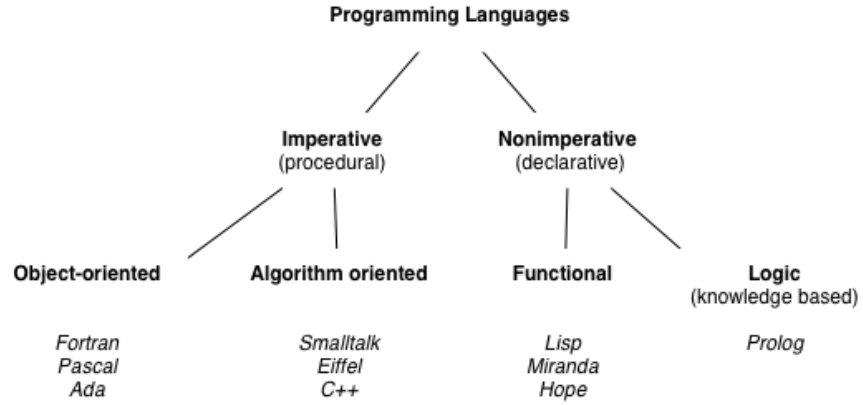


Abbildung 1: Übersicht über die Klassifizierung der Programmierparadigmen mit Beispielsprachen[4]

Bei der imperativen Programmierung besteht ein Programm aus einer Folge von Anweisungen, die sequenziell vom Computer abgearbeitet werden. Im Quellcode wird hierbei einerseits festgelegt, was in welcher Reihenfolge abzuarbeiten ist, andererseits werden Kontrollstrukturen (Sprünge, Schleifen, Verzweigung) bereit gestellt, die den Programmfluss steuern. Die imperative Programmierung ist das am längsten bekannte Programmierparadigma, sie ist nah angelegt an die tatsächliche Arbeitsweise eines Computers der Von-Neumann-Architektur. Bei der imperativen Programmierung wird also eine Art Handlungsanweisung für den Computer beschrieben, der diese Schritt für Schritt abarbeitet. Die prozedurale Programmierung ist ein wichtige Unterkategorie der imperativen Programmierung.

Im Gegensatz zum imperativen Ansatz wird bei der deklarativen Programmierung durch den Programmierer das Ziel der Berechnung angegeben, anstelle der klaren Berechnungsvorschrift. Die Vorteile sind ein höherer Abstraktionsgrad und dadurch eine größere Kompaktheit der Programme. Ein weiterer wichtiger Aspekt funktionaler Programme ist die Seiteneffektfreiheit. Sie ermöglicht partielle Auswertung[5] und einfachere Korrektheitsbeweise für Programme.

Ein Beispiel für die größere Inhaltsdichte pro Zeichen, aber auch die dadurch höhere Komplexität gibt folgendes Code-Beispiel. Auch wird deutlich, wie in der imperativen Sprache der Algorithmus

[4] Aus [14], Seite 8
[5] Insbesondere die parallele Auswertung wird deutlich vereinfacht.

schrittweise beschrieben wird, während in der funktionalen Programmierung schlicht die Berechnungsvorschrift deklariert wird. Zunächst eine imperative Implementierung von Quicksort in Pascal[6]:

```
procedure quicksort(l,r : integer);
 var x,i,j,tmp : integer;
 begin
   if r>l then
   begin
     x:=a[l]; i:=l; j:=r+1;
     repeat
       repeat  i:=i+1 until a[i]>=x;
       repeat  j:=j-1 until a[j]<=x;
       tmp:=a[j]; a[j]:=a[i]; a[i]:=tmp;
     until j<=i;
     tmp:=a[j]; a[j]:=a[l]; a[l]:=tmp;
     quicksort(l,j-1);
     quicksort(j+1,r)
   end
 end;
```

Quicksort in Pascal

Das Programm funktional in Haskell:

```
quicksort [] = []
quicksort (x:xs) =
    quicksort [n | n<-xs, n<x] ++ [x] ++ quicksort [n | n<-xs, n>=x]
```

Quicksort in Haskell

4.1.1 Imperatives Programmierparadigma

Auch wenn Ruby als Zielsprache eigentlich rein objektorientiert ist, erlaubt die Sprache es trotzdem, imperativen Code zu schreiben. Da der von Haru erzeugte Quellcode fast keinen direkten Nutzen aus der Objektorientierung zieht und der imperative Ansatz außerdem die Grundlage des objektorientierten Programmierparadigmas ist, soll hier zunächst das imperative Programmierparadigma erklärt werden.

Der Computer in der Von-Neumann-Architektur ist ein Objekt der Realität, somit unterliegt es zeitlicher Veränderung. Außerdem funktioniert der Computer als „Rechner“, er verwandelt Eingabe- in Ausgabedaten. Dafür verwendet er Objekte, die einen Namen, einen Speicherplatz und einen Wert besitzen.

Imperative Sprachen bilden dieses Konzept unterschiedlich genau auf ihre Sprachkonzepte ab. Ein imperatives Programm verwandelt einen Zustand in einen neuen Zustand, die Durchführung stellt also eine Reihe an Zustandsübergängen dar. Das Programm ist ein „Prozess über Zeit“[7], der Gesamtzustand des Systems wird durch die Menge aller existierenden Variablen zu einem bestimmten Zeitpunkt

[6] Der Quellcode der Pascal- und der Haskell-Implementierung ist von [15].
[7] Nach [13]

dargestellt. Die vom Von-Neumann-Rechner verwendeten Objekte für die Berechnung entsprechen genau den Variablen der Programmiersprache. Diese können über den Verlauf des Programms unterschiedliche Werte annehmen.

Am besten dargestellt wird diese zeitliche Abhängigkeit der Variablen durch folgende Formel:

```
x := x + 1
```

Code-Beispiel zur Verdeutlichung des imperativen Ansatzes

Das imperative Modell hat jedoch auch viele Probleme, die in anderen Paradigmen gelöst sind beziehungsweise konstruktionsbedingt nicht auftreten können:

- *Aliasing*: ein Speicherplatz kann mehrere Namen besitzen.
- *Mehrdeutigkeit der Namen:* ein Name kann einen Wert referenzieren (in Ausdrücken), eine Adresse (in einer Zuweisung) oder einen Zeiger auf eine andere Variable (Pointer).

Die Folge der Probleme ist, dass Werte von Variablen für Entwickler nicht offensichtlich geändert werden können, sie also von den erwarteten Werten abweichen. Dies kann zu versteckten Problemen führen, die oftmals eine lange Suche erfordern und das Programmverständnis erheblich einschränken. Außerdem produzieren Seiteneffekte versteckte Abhängigkeiten: ein unsichtbarer externer Zustand wird zu einer zusätzlichen Eingabegröße für Funktionen. Dies kann beispielsweise dafür sorgen, dass ein Unit-Test von bestimmten Funktionen nur in einer bestimmten Reihenfolge funktioniert, da der externe Zustand manipuliert und gelesen wird. Dieses Problem erbt die Objektorientierung, da sie auf diesem Konzept aufbaut, es jedoch nicht verändert.

4.1.2 Objektorientiertes Programmierparadigma

Während in der imperativen und insbesondere prozeduralen Programmierung das Hauptaugenmerk auf den Algorithmen und nicht auf den zu verarbeitenden Daten liegt, entfernt sich die Objektorientierung von dieser Ansicht. Sie bündelt Daten und ihre verarbeitenden Funktionen (in diesem Kontext heißen die Funktionen „Methoden„) in ein Objekt, mit dem als Einheit im System interagiert wird.

Es ist damit eine Abbildung der Realität, die ebenfalls aus interagierenden realen und imaginären Objekten besteht. Berechnungen sind in dieser Sicht Interaktionen der Objekte untereinander.

Auch werden Programme, die kein fest definiertes Ende haben, in der Objektorientierung erst konzeptionell sinnvoll[8]. In einer Datenbank etwa überdauern die Daten die Algorithmen oder in einem Betriebssystem gibt es keine Eingabedaten im eigentlichen Sinne.

Das objektorientierte Programmierparadigma ist aus der Sichtweise entstanden, dass die Welt aus einer Menge von Objekten aufgebaut ist und Berechnungen in einem Computer im Grunde nur Transformationen realer und imaginärer Objekte zu Objekten im Computerprogramm sind.

Diese Ansicht, zusammen mit abstrakten Datentypen und Datenkapselung sind die grundlegenden Konzepte in den objektorientierten Sprachen. Datenkapselung bedeutet, dass nicht direkt auf alle Daten zugegriffen werden kann, sondern dies über fest definierte Schnittstellen geschehen muss.

[8] Dies bedeutet nicht, dass man Betriebssysteme nicht prozedural programmieren könnte, sondern vielmehr, dass es ausgehend von den Annahmen des Programmierparadigmen konzeptuelle Fragen aufwirft.

Durch die Konzepte der dynamischen Bindung und der Vererbung wird das objektorientierte Modell noch erweitert. Die Vererbung ermöglicht eine Klassifikation der Objekte des Systems in Überklassen und Spezialisierungen davon. Dies erzeugt eine hierarchische Ordnung der Objekte, die der vom Menschen intuitiv durchgeführten hierarchischen Ordnung der realen Welt nahe kommt.

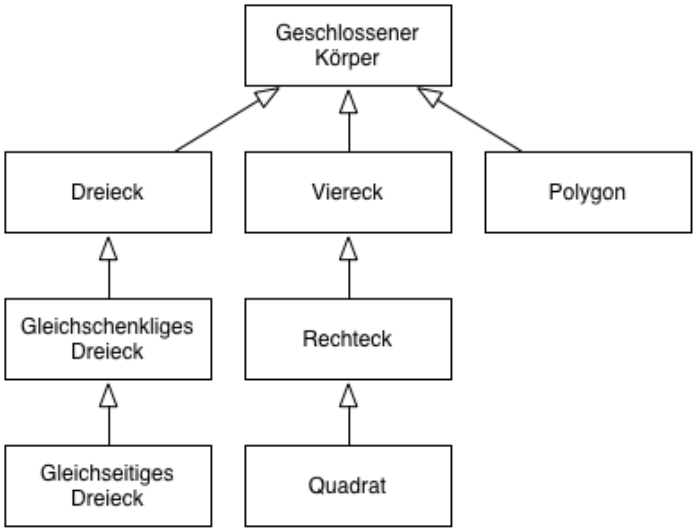


Abbildung 2: Beispiel-Klassenhierarchie[9]

Diese Klassenstruktur stellt direkt ein Modell des Programms dar. Die Programmkonstrukte mit Klassen und Objekten stellen dort das Metamodell dar (siehe 4.2).

Die meisten der objektorientierten Sprachen sind aus einer Erweiterung einer imperativen Sprache entstanden, nicht so Ruby, das stark an Smalltalk angelehnt ist. In einer solch reinen Sprache existieren einige nicht-intuitive Konzepte, so sind beispielsweise Zahlen auch Objekte:

```
irb(main):001:0> 5.class
=> Fixnum
```

Das Literal 5 ist ein Objekt der Klasse Fixnum.

4.1.3 Funktionales Programmierparadigma

Während der Funktionsbegriff in den imperativen Sprachen unter Umständen weit von der ursprünglichen mathematischen Definition abweicht, ist er in der funktionalen Programmierung direkt aus der Mathematik umgesetzt. Er ist daher speicherlos und zeitlos. Aus diesen Eigenschaften resultieren wichtige Eigenschaften:

- *Keine Seiteneffekte*: wie in der Mathematik hängt das Ergebnis einer Funktion nur von ihrer Eingabe ab, es gibt keinen (für den Entwickler) unsichtbaren Zustand.
- Variablen beschreiben *Werte*, keine Speicherplätze. Einmal gebunden, kann eine Variable keinen anderen Wert mehr annehmen. Formal existieren in formalen Sprachen keine Variablen, sondern es sind konstante Funktionen ohne Argument. Daher kann es auch *keine*

[9] Aus [14], Seite 11

zeitlichen Abfolgen geben. Durch die Seiteneffektfreiheit kann ein Ausdruck in beliebiger Reihenfolge ausgewertet werden[10].

Trotz der augenscheinlichen Einschränkungen durch den fehlenden Zustand innerhalb des Programms ist die Sprache dadurch nicht weniger mächtig, dies hat das Fachgebiet der Komplexitätstheorie bewiesen. Doch das funktionale Paradigma hat nicht nur Einschränkungen, sondern auch Erweiterungen im Vergleich zur imperativen Programmierung: Funktionen höherer Ordnung und Lazy Evaluation.

Lazy Evaluation bedeutet, dass Ausdrücke erst dann ausgewertet werden, wenn tatsächlich ihr genaues Ergebnis erfragt wird. Bis zu diesem Punkt werden die Ausdrücke nur soweit ausgewertet, dass die Berechnung fortfahren kann. Dadurch werden unter Umständen kostspielige Berechnungen vermieden beziehungsweise überhaupt erst ermöglicht – zum Beispiel mit unendlichen Listen.

```
-- Bei einer naiven Implementierung von foldr kann es schnell zu
-- Space Leaks kommen. Der gesamte Berechnungsbaum wird im Speicher gehalten.

foldr f z []     = z
foldr f z (x:xs) = x `f` foldr f z xs

foldr (+) 0 [1..100000000]
-- *** Exception: stack overflow
```

Lazy Evaluation kann auch ein Problem bereiten: da der gesamte Berechnungsbaum („Thunk“) im Speicher bleiben muss, kann es bei großen Berechnungen zu Speicherproblemen kommen („Space Leaks“). Aus diesem Grund bieten funktionale Sprachen oft auch explizit strikte Evaluation für bestimmte Funktionen an (oder man kann sie selbst definieren).[11]

Funktionen höherer Ordnung beschreiben die Ansicht, dass Funktionen im Grunde auch nur Daten sind und dadurch auch als solche durch das Programm gereicht werden können. So können Funktionen als Parameter in andere Funktionen gereicht werden oder als Ergebnis von Funktionsaufrufen zurückgegeben werden.

Dieses Prinzip wird noch erweitert durch *Currying*, das es erlaubt, einzelne Parameter von Funktionen zu binden. Als Beispiel sei eine Potenzfunktion genannt.

```
power_function n x = x ** n
```

Parametrisierte Potenzfunktion in Haskell

Wenn diese nur mit einem Argument, einer „2“ aufgerufen wird, erhält man keinen Fehler, sondern als Rückgabewert eine neue Funktion, die nur noch ein Argument fordert und die Quadratzahlen berechnet.

Durch die Funktionen höherer Ordnung und Currying ist es in der funktionalen Programmierung möglich, durch wenige Funktionen bereits eine sehr mächtige Berechnungsumgebung zu schaffen, in der mit wenig Zeilen Code aufwändige Algorithmen definiert werden können (vergleiche 4.1).

[10] Oder durch partielle Auswertung oder Lazy Evaluation sogar gar nicht ausgewertet.
[11] Eine genauere Ausführung findet sich im Wiki von Haskell: http://www.haskell.org/haskellwiki/Foldr_Foldl_Foldl'

4.2 Modelle und Metamodelle[12]

Modell

Ein Modell[13] ist gekennzeichnet durch drei wesentliche Merkmale: Abbildungsmerkmal, Verkürzungsmerkmal und pragmatisches Merkmal.

Ein Modell ist stets eine *Abbildung* eines Originals. Solche Objekte können beliebiger Natur sein, beispielsweise physische Objekte, als auch der Welt der Symbole, der Vorstellungen oder der Gedankenprozesse angehören. Das Verkürzungsmerkmal besagt, dass im Allgemeinen nicht alle Attribute, sondern nur solche, die als relevant erachtet werden (durch den Modellerschaffer oder den Modellbenutzer), abgebildet werden. Der Pragmatismus besagt, dass es nicht nur Modelle „von etwas" sind, sondern auch Modelle für eine Zielgruppe, zu einem bestimmten Zeitpunkt und für einen bestimmten Zweck.

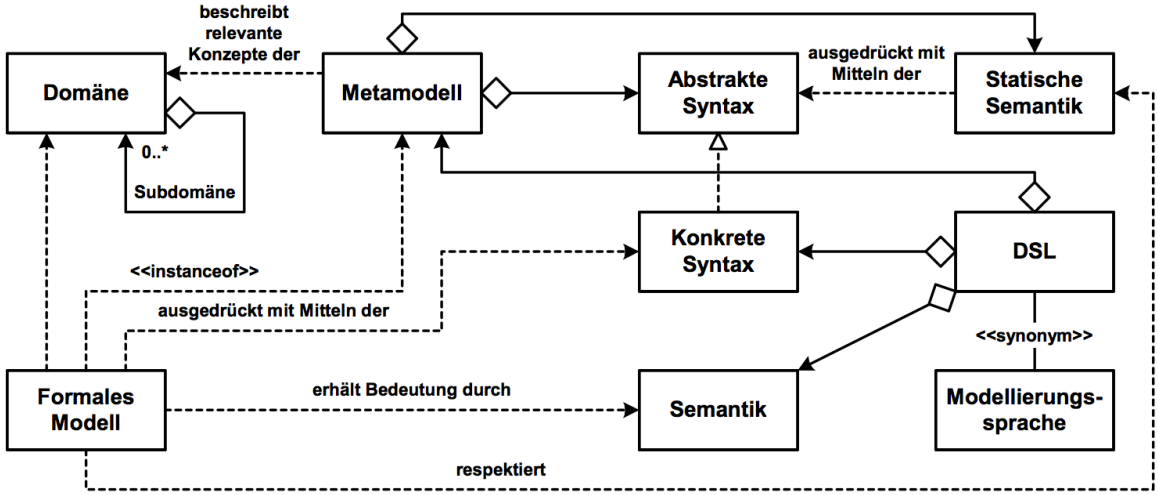


Abbildung 3: Begriffsdefinition: Modellierung[14]

Die zu analysierende Sprache entspricht der Domäne. Das zu erstellende Metamodell stellt eine Kombination aus der syntaktischen Beschreibung mit der statischen Semantik dar.

UML

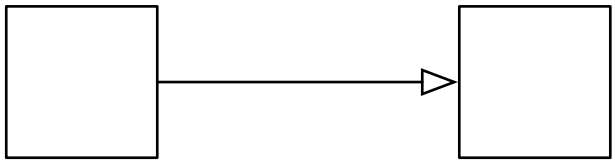
Modelle können auf unterschiedliche Arten dargestellt werden, eine weit verbreitete Art ist die grafische Darstellung mittels UML[15]. Dies ist eine standardisierte Beschreibungssprache, die Bezeichner für die bei einer Modellierung wichtigsten Begriffe definiert. UML spezifiziert eine Reihe von Diagrammtypen. Im Folgenden wird ausschließlich das Klassenmodell verwendet. Dieses Modell spezifiziert Klassen[16], Schnittstellen und deren Beziehungen. In dieser Arbeit wird eine begrenzte Menge der Objekte aus dem UML Klassendiagramm verwendet, die im Folgenden kurz erläutert werden.

[12] Nach [3]
[13] Nach [11]
[14] Aus [3], Seite 28
[15] UML wird hier nur grundlegend erläutert, für Details wird auf die Spezifikation verwiesen: http://www.omg.org/spec/UML/
[16] Der Begriff Klasse ist hier als abstrakter Begriff zu sehen, nicht als eine Klasse aus der Objektorientierung.

Klasse

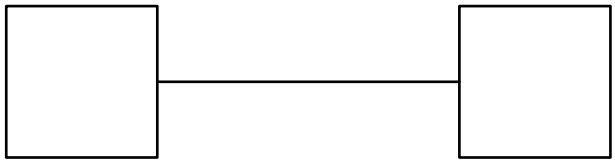
Eine Klasse stellt einen Typ dar. Sie kann eine oder mehrere Klassen spezialisieren, sowie Assoziationen und Abhängigkeiten zu Klassen besitzen. Die eigentlichen Details, die bei einer Klasse zusätzlich angegeben werden können, werden hier nicht verwendet, da sie für die Darstellung in dieser Arbeit nicht relevant sind. Eine Klasse kann abstrakt sein, dies bedeutet sie dient nur als Überklasse, sie kann nicht direkt instanziert werden. Eine abstrakte Klasse wird mit dem Kennwort „abstract“ gekennzeichnet.

Generalisierung



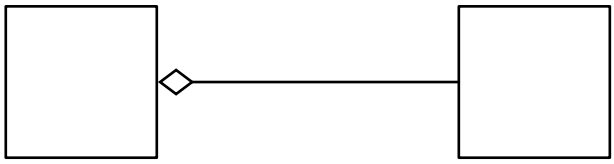
Eine Generalisierung ist eine gerichtete Beziehung zwischen zwei Klassen. Die spezialisierte Klasse ist auch Instanz der generalisierten Klassen (in diesem Beispiel ist links die Unterklasse).

Assoziation

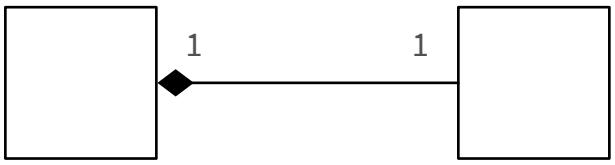


Eine Assoziation stellt eine Beziehung zwischen Klassen dar, die mit Multiplizitäten annotiert wird. Diese geben an, wie viele Objekte in Relation stehen. Typische Werte sind „1“, „0..1“ maximal eins, „*“ beliebig viele und „1..*“ mindestens eins.

Aggregation und Komposition



Die Aggregation stellt eine Beziehung zwischen Objekten und seinen Teilen dar.



Eine Komposition ist ein Spezialfall der Aggregation und stellt eine Existenzabhängigkeit dar[17].

Metamodell

Ein Metamodell wiederum beschreibt nun seinerseits bestimmte Aspekte eines Modells. Die Begriffe sind zunächst rein relativ zu sehen. Ein Metamodell, das ein Modell hat, kann wiederum ein Metamodell besitzen. Dieses liegt dann in Relation zum eigentlichen Modell auf der Metametamodellebene.

Diese Meta-Beziehungen wiederholen sich allerdings nicht endlos, die vierte Ebene (in Abbildung 4: Ebenen der OMG als M3 bezeichnet) ist selbstreferenziell.

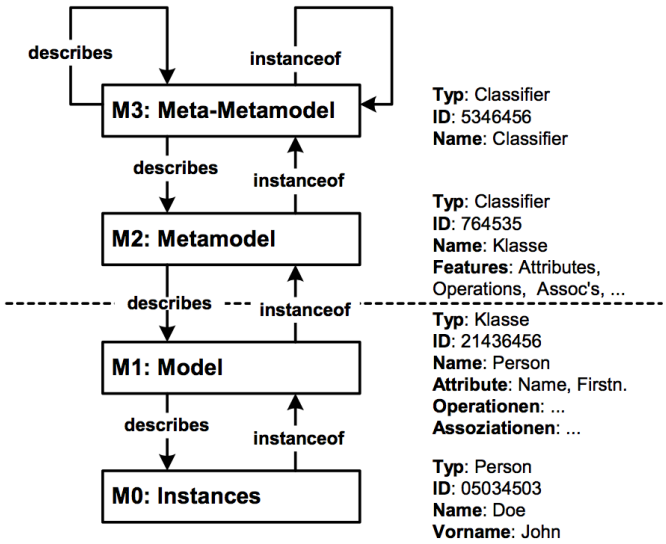


Abbildung 4: Ebenen der OMG[18]

In der Softwareentwicklung wird mit Modellen von Programmen eine Abbildung der zu erstellenden Software erstellt. Die modellgetriebene Softwareentwicklung bietet die Möglichkeit ein UML-Klassendiagramm so zu implementieren, dass eine Klasse des Metamodells einer Klasse des erzeugten Programms entspricht. Dies ist sinnvoll, wenn man aus den Modellen des Programms direkt Quellcode erzeugen will (und umgekehrt). Eine Variation dieses Ansatzes ist es, mithilfe eines Parsergenerators einen Parser für eine Sprache zu erzeugen. Dann entspricht die Grammatik mit ihren Regeln und Klassen des AST dem Metamodell.

4.3 Sprachtransformationen

Während diese Arbeit eine Sprachtransformation von einer Hochsprache zu einer anderen betrachtet, werden andere Sprachtransformationen deutlich häufiger eingesetzt. So ist ein typischer Kompiliervorgang eine Transformation von einer Sprache in (meist) Maschinencode – der ebenfalls eine Sprache darstellt.

Durch den Ansatz, die Sprachtransformation über Metamodelle durchzuführen, handelt es sich eher um eine Modelltransformation. Die Transformation wird dabei selbst als eine Art seiteneffektfreies

[17] Die Teile können nicht ohne das Ganze existieren.
[18] Aus [3], Seite 62

funktionales Programm gesehen, das die Bestandteile des einen Modells auf Teile des anderen Modells abbildet.

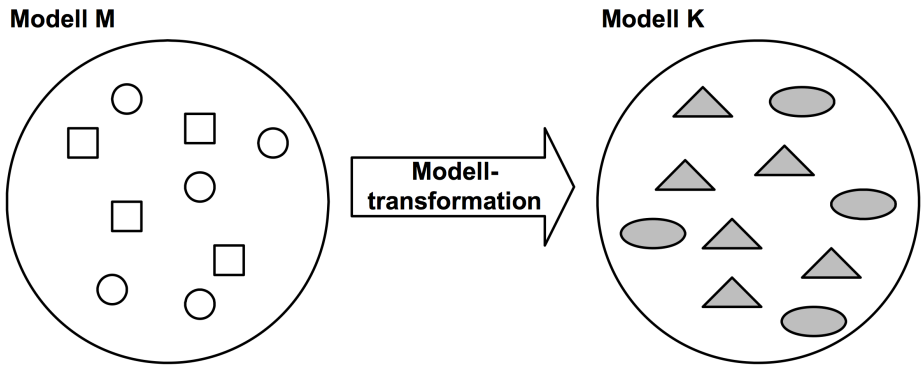


Abbildung 5: Modelltransformation[19]

Die Modelle sind Instanzen unterschiedlicher Metamodelle (in diesem Fall sind die Modelle die Sprachen und die Metamodelle entsprechen der Menge ihrer Paradigmen und Konzepte).

Für diese Arbeit wird zunächst das Metamodell M der Ausgangssprache erstellt, anschließend das Metamodell K der Zielsprache erarbeitet und zum Abschluss werden die einzelnen Transformationsschritte erläutert.

4.4 Eingesetzte Sprachen und Technologien

Im Folgenden werden alle eingesetzten Technologien beschrieben, um die praktische Aufgabenstellung zu verdeutlichen. Hierbei werden allerdings nur diejenigen Aspekte beschrieben, die im Umfang der praktischen Arbeit berücksichtigt sind.

4.4.1 Haskell

Haskell ist eine rein funktionale Programmiersprache. Ende der 1980er Jahre sollte eine einheitliche, standardisierte und moderne funktionale Sprache entwickelt werden, die fortan als Grundlage für wissenschaftliche Arbeiten dienen sollte. 1990 erschien die erste Version 1.0 von Haskell, 1998 und zuletzt 2010 wurde die Sprachspezifikation aktualisiert[20].

Haskell ist statisch und stark typisiert. Statische Typisierung bedeutet, dass dem Compiler bereits zur Kompilierzeit alle Typen von allen Ausdrücken bekannt sind. Starke Typisierung bedeutet, dass ein Programm mit inkompatiblen Typen nicht kompilieren wird und Typen auch nicht automatisch umgewandelt werden. Damit diese starke Typisierung allerdings keinen Mehraufwand in den Deklarationen für den Entwickler darstellt, verwendet Haskell ein mächtiges Typinferenzsystem, das einen Großteil der Typen eines Programms automatisch erkennen und prüfen kann. So sind in einem normalen Programm wenig explizite Typdeklarationen notwendig.

[19] Aus [3], Seite 200
[20] Diese Arbeit baut auf Haskell 2010 auf.

```
Prelude> let fun x = x * 5
Prelude> :t fun
fun :: Num a => a -> a
Prelude> let str x = x ++ ": "
Prelude> :t str
str :: [Char] -> [Char]
```

Automatische Typinferenz für Funktionen ohne explizite Typdeklaration (in GHCi)[21].

Als funktionale Sprache unterstützt Haskell das in 4.1.3 erwähnte Currying, Lazy Evaluation und die Seiteneffektfreiheit. Eine komplette Seiteneffektfreiheit ist allerdings oftmals sehr hinderlich: so können keine Dateien im Dateisystem gelesen oder geschrieben werden, es ist keine Interaktion mit Benutzern möglich und Datums- und Zufallsfunktionen sind auch nur sehr eingeschränkt verwendbar. Um diesem Umstand zu umgehen implementiert Haskell das Konzept der *Monaden*. Hierbei werden Typen in Monaden gekapselt, die damit eine Grenze für Seiteneffekte darstellen. Das Programm außerhalb verbleibt seiteneffektfrei, während innerhalb des Monaden Interaktion möglich ist. Monaden werden in dieser Arbeit eigentlich nicht behandelt, sie sollen aber aus Gründen der Vollständigkeit erwähnt werden. Hierzu definiert ein Monade typischerweise drei Schnittstellen: einen Konstruktor, eine Einheitsfunktion und einen sogenannten Bind-Operator, der Änderungen am innenliegenden Wert ermöglicht.

Haskell erlaubt Typvariablen, so können typpolymorphe Funktionen geschrieben werden. Mit monomorphen Typen müsste es beispielsweise unterschiedliche Funktionen für die Längenberechnung einer String-Liste und einer Integer-Liste geben. Typpolymorphie ermöglicht hierbei den tatsächlichen Typ zu abstrahieren, da er für die Berechnung unbedeutend ist.

Außerdem wird die Verwendung von *algebraischen Datentypen* in Haskell ermöglicht. Sie werden durch Typkonstruktoren definiert und dienen der Erzeugung von neuen, zusammengesetzten Datentypen. Sie stellen neben regulären Typ-Umbenennungen (Haru führt einen Typ „RubyFragment“ ein, der nur ein String mit einem anderen Namen ist) eine mächtige Möglichkeit dar, um geschachtelte Datenstrukturen wie Bäume zu erstellen. Konstruktoren ohne Argument erzeugen einen Enum-Typ. Dieses Konzept wird noch erweitert durch GADTs[22], die die algebraischen Datentypen um parametrisierte Typen erweitern.

Durch die im vorigen Kapitel erwähnte Bindung der Werte an Namen und die Darstellung der Werte als nullstellige Funktionen ist ein Aufruf von Funktionen und ein Zugriff auf (vermeintliche) Variablen transparent und Bedarf keiner besonderen Behandlung.

4.4.2 Glasgow Haskell Compiler (GHC)[23]

Der *Glasgow Haskell Compiler* (GHC), dessen erste Version 1992 im Zuge eines Forschungsprojektes entstand, diente in seiner bisher über 20 jährigen Geschichte als Forschungsprojekt für viele unterschiedliche Arbeiten im Compilerbau. Ein Ziel des GHC ist es, einen robusten und portablen Compiler zu entwickeln, der performanten Maschinencode erzeugt. Außerdem soll durch die modulare Struktur die Erweiterbarkeit des Compilers als großes Softwareprojekt gewährleistet werden. Und

[21] Achtung: in GHCi ist die Bedeutung von „let“ bedeutend anders als in einem regulären Haskell-Programm.
[22] Generalized Algebraic Datatypes
[23] Nach [2], [7] und [8]

zuletzt soll der Compiler selbst als Forschungsobjekt dienen, um festzustellen, wie sich reale Programme verhalten – um anschließend bessere Compiler bauen und entwerfen zu können.

Der GHC besteht aus unterschiedlichen Schritten, die in einer Pipeline abgearbeitet werden. Das Ergebnis eines Schrittes ist wiederum die Eingabe des darauffolgenden Schrittes. Da der erstellte Quellcode nah an dem Aufbau und der Programmierschnittstelle („API“) des GHC angelehnt ist, wird im Folgenden der strukturelle Aufbau und Ablauf einer Kompilierung erklärt.

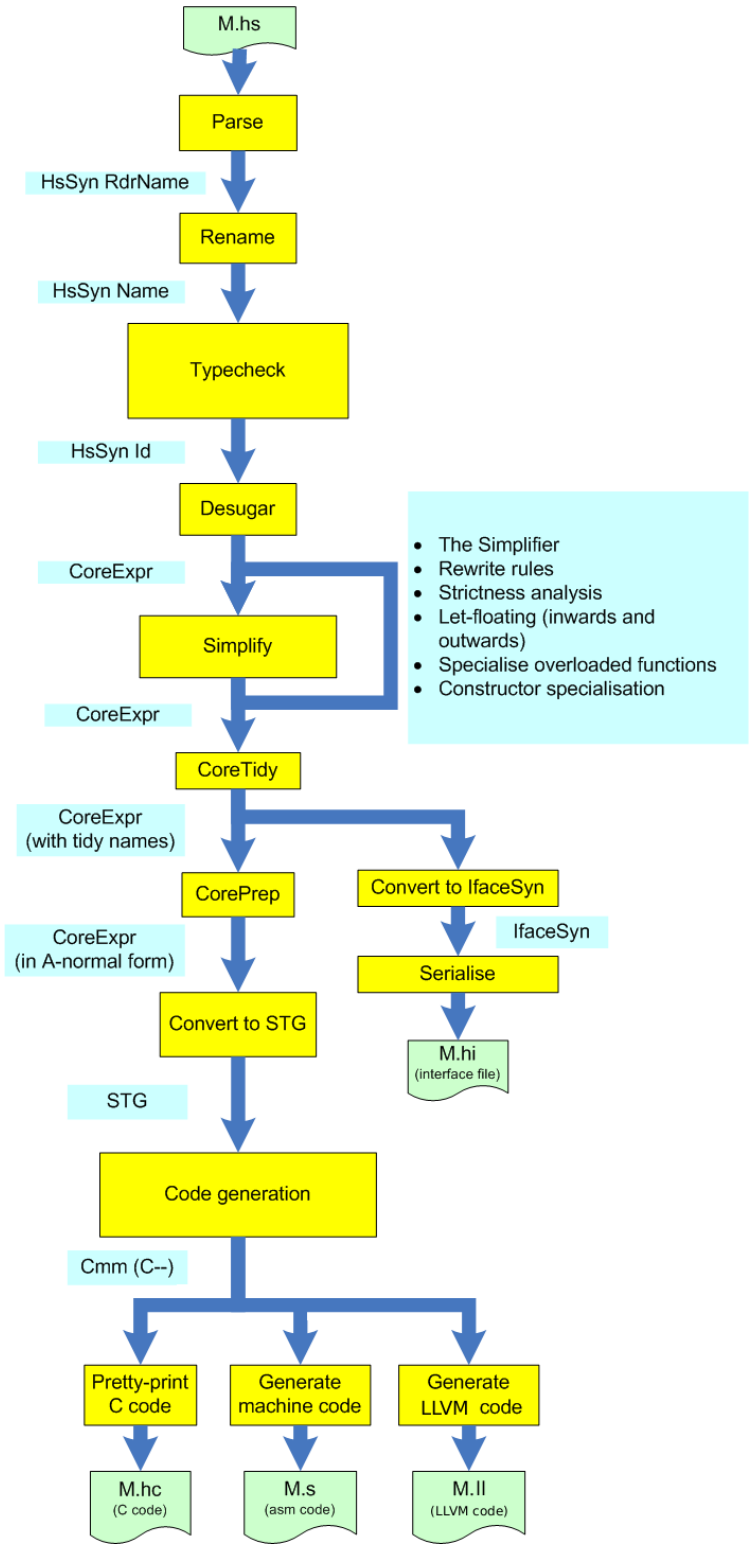


Abbildung 6: Die Compiler-Phasen des GHC

Die für Haru wesentlichen Phasen sind:

- Der *Parser* liest die Eingabedatei ein und erstellt aus dem Eingabequellcode einen abstrakten Syntaxbaum (AST).
- Der *Renamer* verbindet die einfachen Identifier-Strings im AST nun mit deren Bindings[24] und fügt ein eindeutiges Suffix für die Namen hinzu, um Namensüberschneidungen zu vermeiden.
- Im *Typecheck* wird die Typinferenz durchgeführt, die Typisierung validiert (Typprüfung) und die Namen werden durch eine Kombination aus Namen und Typ ersetzt.
- *Desugar* konvertiert den Haskell AST in eine Darstellung in Core, der internen Zwischensprache.
- Die *Simplify*-Phasen führen vielfältige Optimierungen durch wie Variable Floating[25] oder Common Sub-Expression Elimination[26].
- *CoreTidy* bereinigt Bindings und benennt sie global eindeutig.

Die ersten drei Schritte könnte man als das Frontend gruppieren, alle Eingabefehler werden dort entdeckt. Die nächsten drei Schritte bis einschließlich *CoreTidy* sind die Optimierungsphasen, danach beginnt das Backend. GHC verwendet eine Zwischensprache als Vorbereitung für die Codegenerierung namens STG[27], aus der dann C-- Code generiert wird.

Für diese Arbeit interessant sind insbesondere das Frontend und die Optimierungsphasen. Hier wird zunächst am Ende des Frontends die komplette Sprache Haskell in eine kleinere Sprache namens Core umgewandelt, die dann anschließend weiter optimiert wird.

4.4.3 Core

Core ist die interne Darstellungssprache des Programms. GHC hat hierfür viele Jahre System F verwendet. System F bezeichnet das Lambda-Kalkül nach Church und Kleene, mit der Erweiterung um polymorphe Typisierung. Dies ermöglicht die Arbeit mit schematisierten Funktionen.

Mit monomorpher Typisierung müsste es mehrere Längen-Funktionen für Listen geben, beispielsweise eine Funktion für eine Liste von Strings und eine für eine Liste von Integer. Dies ist bei polymorpher Typisierung nicht notwendig, da der Typ der Funktion mit dem Listen-Typ parametrisiert werden kann.

Doch dieses System hat sich bei der Implementierung von GADTs[28] als unzureichend herausgestellt und wurde deswegen 2006 zu System FC[29] erweitert.

Der Grund intern eine zweite, vereinfachte Darstellung der Sprache zu verwenden ist, dass dadurch viele Schritte entfallen. Core und Haskell sind gleich mächtig, allerdings ist die Syntax und das Metamodell von Core signifikant kleiner. Optimierungen und Code-Generierung geschehen auf Basis von Core. Bei einer Spracherweiterung von Haskell wird nur das Frontend angepasst, inklusive der Umwandlung zu Core. Dadurch kann Haskell erweitert werden, ohne dass man für diese neuen

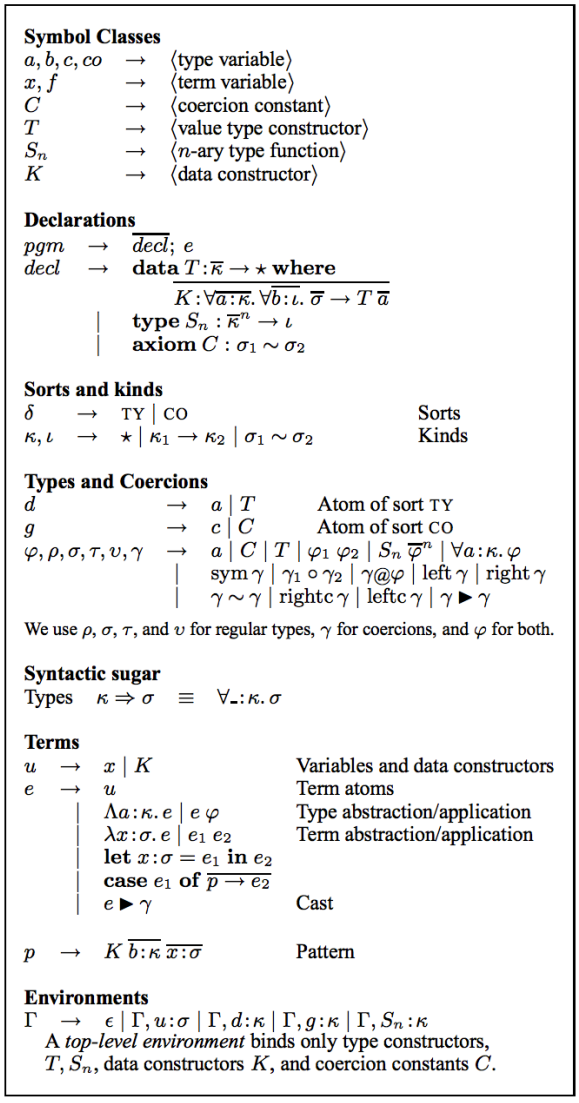
[24] Ein Binding in Haskell ist die Definition eines Namens. Dies kann der Name einer Funktion sein, als auch der Name einer Variablen.
[25] Verschiebt Let-Blöcke: beispielsweise aus sich wiederholenden Ausdrücken heraus (um die mehrfache Ausführung zu vermeiden) oder in Zweige eines Case-Ausdrucks hinein, falls das Let nur für diesen Zweig gilt (und nur dort die Variablenbindungen des Let verwendet werden).
[26] Eliminiert doppelte Code-Zweige. Durch die Seiteneffektfreiheit können gleiche Ausführungen mit gleicher Eingabe nur das gleiche Ergebnis erzeugen. Dies wird auch bei mehrfacher Verwendung nur einmal berechnet.
[27] Kurz für „Spineless Tagless G-machine“, [5]
[28] Siehe 4.4.1
[29] Siehe [12] (Appendix)

Funktionen die Optimierungen anpassen müsste. In der Gesamtschau besteht Haskell im GHC also aus 3 voneinander relativ stark getrennten Strukturen:

- Dem *Frontend* (die eigentliche „Sprache Haskell“), das zu Core transformiert
- *Core*, auf dem alle Optimierungen durchgeführt werden, das zu LLVM[30] transformiert
- *LLVM*, das nativen Code für die aktuelle Maschine erzeugt

Durch diese Separierung können einzelne Bereiche des Compilers ohne große Auswirkungen auf das restliche Programm erweitert oder umgebaut werden.

Die Verwendung von Core ist jedoch nicht in der Sprachspezifikation von Haskell. GHC hat dies für die eigene Compilerimplementierung so festgelegt. Da Haru allerdings auf dem GHC aufbaut, ist die Sprache für den Compiler essenziell. Die Verwendung von Core verringert signifikant den Aufwand, den ein Transformationstool aufbringen muss, um Haskell zu übersetzen – hier die komplette Grammatik von Core:



Symbol Classes

a, b, c, co	$\rightarrow$	⟨type variable⟩
x, f	$\rightarrow$	⟨term variable⟩
C	$\rightarrow$	⟨coercion constant⟩
T	$\rightarrow$	⟨value type constructor⟩
S_n	$\rightarrow$	⟨n-ary type function⟩
K	$\rightarrow$	⟨data constructor⟩

Declarations

$pgm \rightarrow \overline{decl};\ e$

$decl \rightarrow \mathbf{data}\ T : \overline{\kappa} \rightarrow \star\ \mathbf{where}\ \overline{K : \forall \overline{a:\kappa}.\ \forall \overline{b:\iota}.\ \overline{\sigma} \rightarrow T\ \overline{a}}$

$\mid\ \mathbf{type}\ S_n : \overline{\kappa}^n \rightarrow \iota$

$\mid\ \mathbf{axiom}\ C : \sigma_1 \sim \sigma_2$

Sorts and kinds

δ	$\rightarrow$	TY ∣ CO	Sorts
κ, ι	$\rightarrow$	$\star \mid \kappa_1 \rightarrow \kappa_2 \mid \sigma_1 \sim \sigma_2$	Kinds

Types and Coercions

d	$\rightarrow$	$a \mid T$	Atom of sort TY
g	$\rightarrow$	$c \mid C$	Atom of sort CO
$\varphi, \rho, \sigma, \tau, \upsilon, \gamma$	$\rightarrow$	$a \mid C \mid T \mid \varphi_1\ \varphi_2 \mid S_n\ \overline{\varphi}^n \mid \forall a:\kappa.\ \varphi$	
		$\mid\ \mathrm{sym}\ \gamma \mid \gamma_1 \circ \gamma_2 \mid \gamma @ \varphi \mid \mathrm{left}\ \gamma \mid \mathrm{right}\ \gamma$	
		$\mid\ \gamma \sim \gamma \mid \mathrm{rightc}\ \gamma \mid \mathrm{leftc}\ \gamma \mid \gamma \blacktriangleright \gamma$	

We use ρ, σ, τ, and υ for regular types, γ for coercions, and φ for both.

Syntactic sugar

Types $\kappa \Rightarrow \sigma \equiv \forall \_:\kappa.\ \sigma$

Terms

u	$\rightarrow$	$x \mid K$	Variables and data constructors
e	$\rightarrow$	u	Term atoms
	$\mid$	$\Lambda a:\kappa.\ e \mid e\ \varphi$	Type abstraction/application
	$\mid$	$\lambda x:\sigma.\ e \mid e_1\ e_2$	Term abstraction/application
	$\mid$	$\mathbf{let}\ x:\sigma = e_1\ \mathbf{in}\ e_2$	
	$\mid$	$\mathbf{case}\ e_1\ \mathbf{of}\ \overline{p \rightarrow e_2}$	
	$\mid$	$e \blacktriangleright \gamma$	Cast
p	$\rightarrow$	$K\ \overline{b:\kappa}\ \overline{x:\sigma}$	Pattern

Environments

$\Gamma \rightarrow \epsilon \mid \Gamma, u:\sigma \mid \Gamma, d:\kappa \mid \Gamma, g:\kappa \mid \Gamma, S_n:\kappa$

A *top-level environment* binds only type constructors, T, S_n, data constructors K, and coercion constants C.

Abbildung 7: Grammatik von Core[31]

[30] Dies gilt aktuell als der favorisierte Codeerzeugungspfad. Die direkte Maschinencode-Kompilierung ist hauptsächlich für GHCi interessant, den Kommandozeileninterpreter von GHC.

Haru verwendet zur Reduzierung der Komplexität als Ausgangssprache Core. Core bietet außerdem einige fundamentale Vorteile:

- Haskell ist stark statisch und implizit typisiert, d.h. viele Typen werden durch Typinferenz ermittelt. Core ist ebenso stark statisch typisiert, allerdings explizit. (Eigentlich wäre die Typisierung von Core nicht notwendig, da das Frontend das Programm bereits als typkorrekt akzeptiert hat – aber hierdurch können die Simplifizierungsschritte validiert werden [diese müssen die Typkorrektheit erhalten]).
- Core ist sehr stabil. In über 20 Jahren Verwendung musste es nur einmal erweitert werden (um Type Coercions inkl. Casts), während der gesamte GHC um einen Faktor von 5 gewachsen ist[32].

4.4.4 Ruby

Ruby wurde von Yukihiro Matsumoto entwickelt und die erste Version im Jahr 1995 veröffentlicht. Es ist eine interpretierte Sprache. Nachdem Ruby anfangs mangels englischsprachiger Dokumentation fast ausschließlich in Japan verwendet wurde, fand im Jahr 2000 eine Aktion statt, die Ruby auch außerhalb Japans bekannt machen sollte. Heute wird Ruby als Open-Source-Projekt gepflegt, der Quellcode findet sich auf der Code-Hosting-Plattform GitHub[33]. Die Sprache unterliegt keiner schriftlich festgehaltenen Spezifikation, vielmehr ist sie durch die Ausführung spezifiziert. Das Verhalten, das die Referenzimplementierung des Compilers erzeugt, legt die Semantik fest[34].

Paradigmen

Ruby ist eine vollkommen objektorientierte Sprache, die sehr durch Konzepte aus Smalltalk inspiriert wurde. So ist jedes Element in einem Programm auch tatsächlich ein Objekt („alles ist ein Objekt“), auch beispielsweise Klassen, Zahlen oder Zeichen. Weitere Programmierparadigmen sind ebenfalls in der Sprache berücksichtigt.

Ein rein prozedurales Programm kann durch den Umstand geschrieben werden, dass jedes Ruby Programm automatisch in einem globalen main-Objekt[35] erstellt wird, die eigentlich globalen Funktionen sind dann Methoden dieses main-Objekts, selbst definierte Klassen sind innere Klassen.

```
def example
    puts „Hello World“
end

example
```

Ein lauffähiges prozedurales Ruby-Programm.

Funktionale Programmierung wird dadurch ermöglicht, dass alle Ausdrücke („Expressions“) in Ruby einen Wert zurückliefern, Anweisungen („Statements“) im eigentlichen Sinne existieren in Ruby nicht[36]. Außerdem können anonyme Funktionen als Codeblöcke definiert werden.

---

[31] Aus [12]
[32] Von etwa 28.000 auf etwa 140.000, [2]
[33] https://github.com/
[34] Es gibt RubySpec, eine Initiative, die eine ausführbare Sprachspezifikation schreiben will (http://rubyspec.org/)
[35] Die geschieht für den Programmierer vollkommen transparent.
[36] Auch wenn in der Grammatik ein Ausdruck als „STMT“ bezeichnet wird.

Syntaktische und semantische Besonderheiten

Jeder Ausdruck in Ruby hat einen Wert, auch Konstrukte, die dies in anderen Sprachen üblicherweise nicht gewährleisten.

```
example = if cond then „wahr“ else „falsch“ end

language = case scrutinee
             when „de“  then „deutsch“
             when „en“  then „englisch“
             else „unbekannt“
           end
```

Jeder Ausdruck in Ruby hat einen Rückgabewert, auch Sprachkonstrukte wie case und if.

Ruby erzwingt in eindeutigen Fällen keine Klammern bei einem Methodenaufruf. Dies sorgt für eine transparente Verwendung von Variablen und Methoden ohne Argument.

```
a = „variable a“
def b ()
  „methode b“
end

puts a # „variable a“
puts b # „methode b“
```

Beim Aufruf von b werden keine Klammern benötigt.

Der Wert des letzten Ausdrucks in einer Methode oder einer Expression ist automatisch auch der Rückgabewert, es ist kein explizites „return“ notwendig.

4.4.5 Verwandte Arbeiten

Es gibt eine Reihe weiterer Arbeiten, die Haskell in eine andere Zielsprache definieren. Neben dem GHC, der als Hauptcompiler für Haskell in Maschinencode oder LLVM[37]-Code übersetzt (oder interpretiert), ist aktuell JavaScript als Zielsprache beliebt. So existieren drei Projekte, die versuchen, Haskell auch in der Webentwicklung zum Einsatz zu bringen:

Fay definiert eine Untermenge von Haskell als Quellsprache und kompiliert direkt von Haskell zu JavaScript (ohne Umweg über STG). Haskell Closures werden direkt zu JavaScript Closures kompiliert.

Haste implementiert den sogenannten Eval/Apply-Algorithmus, ähnlich zu STG, verwendet aber ebenfalls JavaScript Closures und den JavaScript Stack.

GHCJS implementiert STG mit einem Stack, ähnlich zur nativen Implementierung, konvertiert aber Closures selbst.

[37] LLVM ist ein Backend für Compiler, das in unterschiedlichste Zielarchitekturen übersetzt. Der Gedanke dahinter ist, dass Compiler nur noch das Frontend behandeln müssen, anschließend LLVM-Code erzeugen und damit das gesamte Backend mit allen aufwändigen Anpassungen für unterschiedliche Architekturen von LLVM übernommen wird.

5 Realisierung

Für die Transformation der Metamodelle werden zunächst die eigentlichen Metamodelle erstellt und erläutert. Anschließend werden die einzelnen Bestandteile in das andere Modell transformiert. Bei Haskell in Verbindung mit GHC ergibt sich eine Besonderheit, da intern erst in die Sprache Core umgewandelt wird. Deswegen wird zusätzlich ein Metamodell für Core erstellt und die eigentliche Transformation trennt sich dann in die Bestandteile Haskell zu Core und Core zu Ruby.

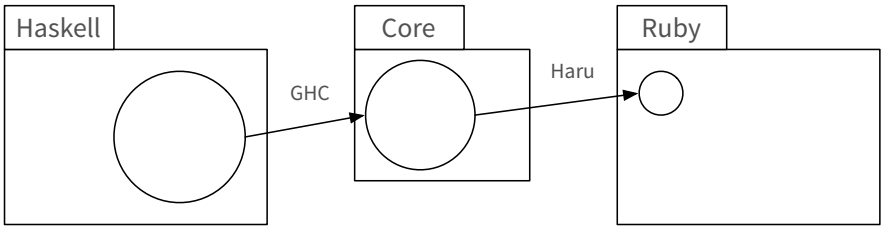


Abbildung 8: Übersicht über die Sprachtransformationsschritte

Hierzu werden zu allen Aspekten kompakte UML-Diagramme erstellt, die den Fokus auf einzelne Bereiche legen. Um die Diagramme übersichtlich zu halten wird die Gesamtansicht in viele Einzelteile aufgespalten. Es werden nicht immer alle Einzelteile in einem eigenen Diagramm erläutert, auch werden nicht alle Eigenschaften abgebildet – sondern jeweils nur die Teile, die für diese Arbeit relevant sind.

„Entfernte“ Klassen, die eine eigene Erklärung haben und auf die in dem jeweiligen Diagramm nur verwiesen wird, haben eine gestrichelte Umrandung. Assoziationen sind benannt, außer wenn die Zieltypen selbst bereits angeben, welchem Zweck die Verbindung dient.

Die Begriffe in den Datenmodellen sind auf Englisch, da dadurch der Bruch zwischen der Sprachspezifikation und dieser Arbeit möglichst gering bleibt. In Kapitel 5 wird also häufig zwischen Englisch und Deutsch gewechselt.

5.1 Metamodell von Haskell

Das Metamodell wurde aus dem Language Report von 2010 von Haskell erstellt, der die Sprachspezifikation darstellt. Ungenauigkeiten wurden durch Analyse von Spezifikationen des GHC geklärt, der eine sehr genaue Umsetzung der Sprachspezifikation darstellt – die internen Typen heißen sogar wie die Nichtterminale in der Grammatikdefinition.

Haskell verwendet einige globale Annahmen, die nicht direkt in den Metamodellen auftauchen: Ausdrücke werden *lazy*[38] ausgewertet, es wird Currying unterstützt und jeder Ausdruck muss zwingend einen Rückgabetyp haben.

5.1.1 Übersicht

Zunächst wird eine Übersicht über ein komplettes Programm gegeben, bevor dann der eigentliche Kern der Sprache, die Expressions modelliert werden.

[38] Das Ergebnis eines Ausdrucks wird nicht direkt berechnet, sondern erst wenn tatsächlich das konkrete Ergebnis benötigt wird.

Modul

Auf der obersten Ebene besteht ein Haskell-Programm aus *Modulen*. Wenn kein Modul explizit angegeben ist, wird der Modulname „Main“ verwendet und alle Funktionen exportiert.

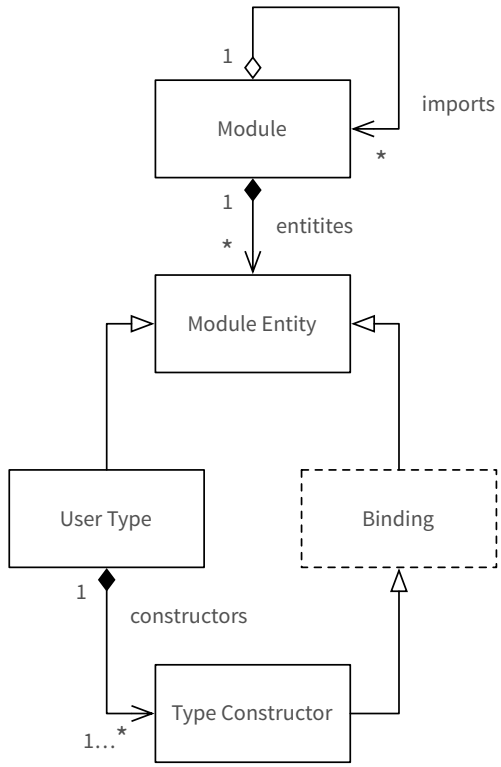


Abbildung 9: Haskell: Module

Ein Haskell-Modul kann andere Module oder nur Teile davon (Funktionen und Typen) importieren und anschließend verwenden. Das Modul „Prelude“ wird immer geladen und ist der Kern der Laufzeitumgebung.

Ein einzelnes Haskell-Modul wiederum besteht auf der obersten Ebene aus *Bindings* und *Typ-Konstruktoren*. Typausdrücke können viele verschieden Formen annehmen, wie Typumbenennung, Definition eigener algebraischer Datentypen und Klasseninstanzen. Ein Binding ist eine benannte Funktionsdefinition.

Typdefinitionen

Haskell-*Datentypen* sind entweder ein Basisdatentyp, eine zusammengesetzter Typ („Constructed Type“) oder ein vom Benutzer definierter Typ.

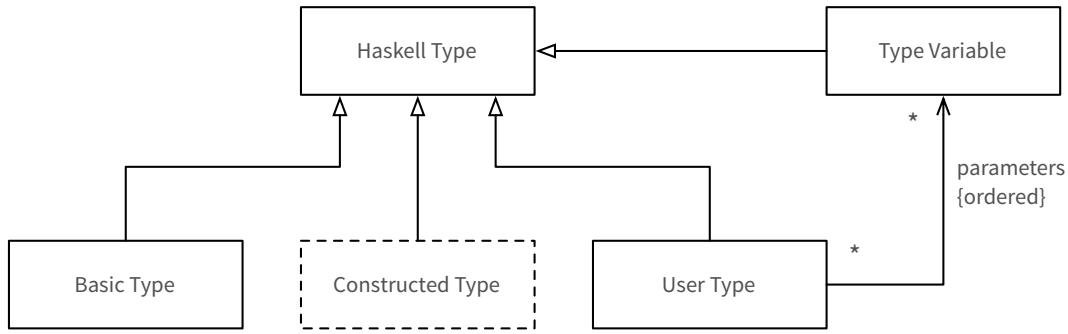


Abbildung 10: Haskell: Types

Basic Types sind die Basisdatentypen, wie Bool, Integer oder String[39]. User Type, also vom Benutzer definierte algebraische Datentypen wurden in der Haskell Sprachbeschreibung bereits ausführlich beschrieben.

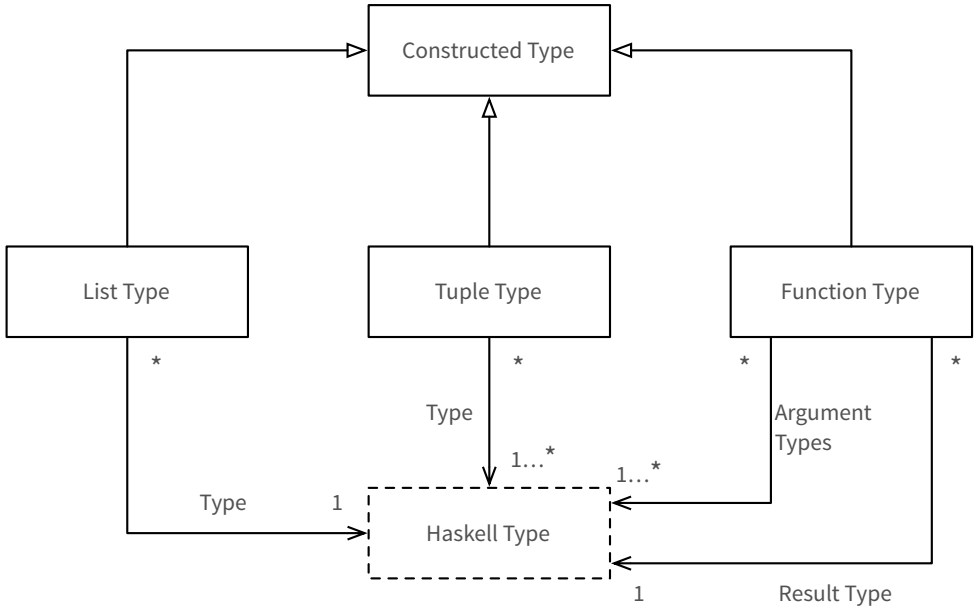


Abbildung 11: Haskell: Constructed Type

Constructed Types sind entweder eine Liste an Werten, ein Tupel aus Werten oder eine Funktion, die Argumenttypen und einen Ergebnistyp hat. Die Listenelemente müssen alle vom gleichen Typ sein, die Liste ist aber nach der Erstellung nicht in der Länge beschränkt. Das Tupel ist praktisch ein Gegenstück zur Liste: es erlaubt unterschiedliche Typen aber nach Erstellung erzwingt es eine feste Länge.

Datentypen werden in der Transformation zu einem großen Teil ausgelassen. Dies hat den Grund, dass die Typen zur Codeerzeugung zu weiten Teilen nicht mehr benötigt werden. Die Typisierung hilft die Typkorrektheit eines Programms festzustellen und Zuordnung von polymorphen Aufrufen aufzulösen.

[39] Tatsächlich sind in GHC nur sehr wenige Typen „wired-in“. Die meisten auch nativen Typen sind über data-Definitionen in der Standbibliothek definiert.

Alle diese Schritte sind bereits erfolgreich durch die vorhergehenden Schritte im GHC erledigt. Einzig die algebraischen Datentyp-Konstruktoren werden explizit übersetzt, Literale und Operatoren werden direkt bei der Erzeugung des Ruby-ASTs umgewandelt.

Bindings

Bindings sind ein allgemeiner Begriff für Bindungen von Ausdrücken an explizite Namen.

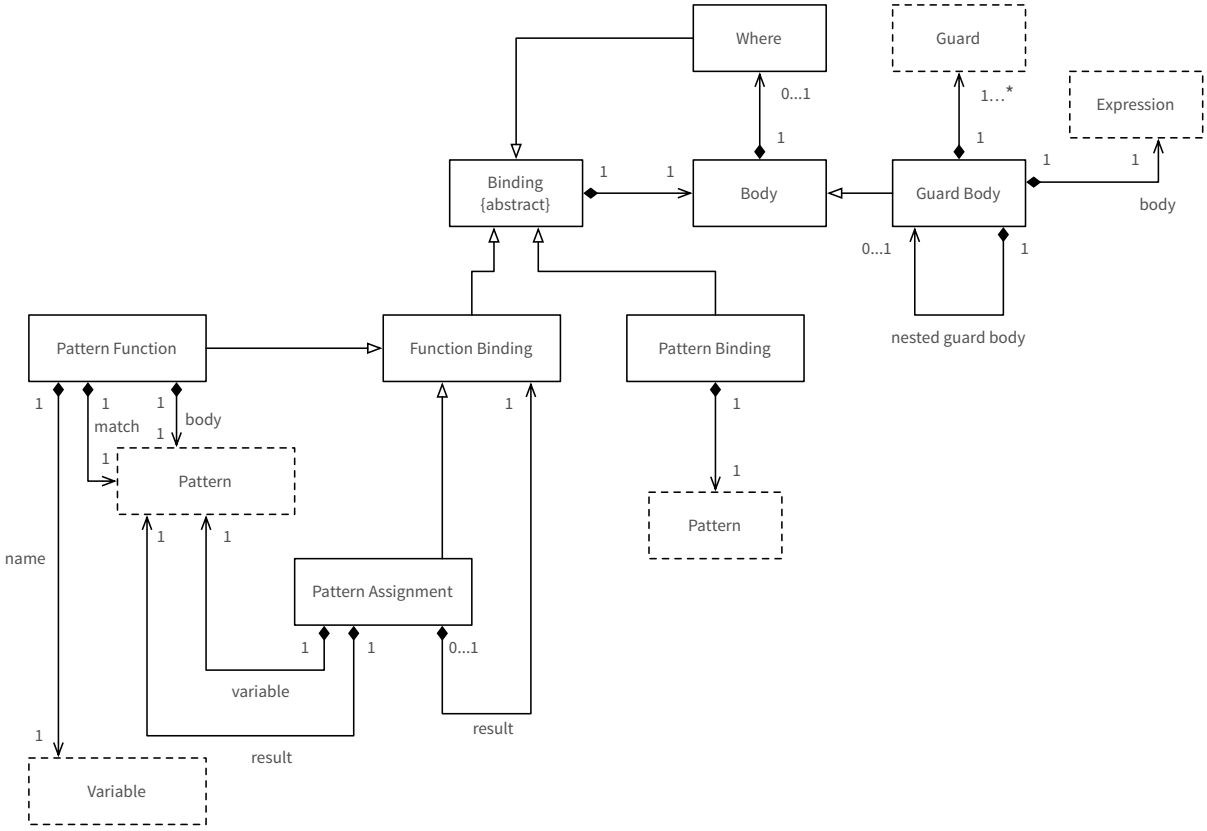


Abbildung 12: Haskell: Bindings

Bindings können einerseits Funktionsdefinitionen auf oberster Ebene sein, Bindungen in Let-Expressions oder Typdefinitionen für Funktionen. Sie können rekursiv oder nicht-rekursiv sein, was schlicht angibt, ob in den Bindings Referenzen auf Variablen aus den Bindings selbst existieren (diese also vor der Codeerzeugung topologisch sortiert werden müssen).

Ein Binding besteht aus einem Body, der den Rumpf der Definition angibt. Dieser kann entweder selbst eine Expression sein, oder eine Reihe von Guards. Guards stellen einen Pattern Matching-Mechanismus dar.

Der Kopf einer Funktion kann entweder ein Name mit einem optionalen Pattern sein (*Pattern Function*) oder eine Pattern Zuweisung (*Pattern Assignment*). Beide Mechanismen zusammen ermöglichen Pattern Matching auf Funktionsargumenten.

```
-- Pattern Matching auf Funktionsargumenten
factorial 0 = 1
factorial n = n * (n - 1)

-- Pattern Matching für Typ-Dekonstruktion
data Color = Red | Green | Blue
data ColoredText  = ColoredText Color String

printText (ColoredText Red string)  = „red: „ ++ string
printText (ColoredText Blue string) = „blue: „ ++ string
-- auch Wildcards sind möglich, die angeben, dass der eigentliche Wert nicht
-- von Interesse ist (nur das mögliche Vorhandensein)
printText (ColoredText _    string) = „sonstige Farbe: „ ++ string
```

Pattern-Matching in Funktionsargumenten

Type Signatures dienen der Spezifikation von Argument-Typen und Ergebnistypen von Funktionen.

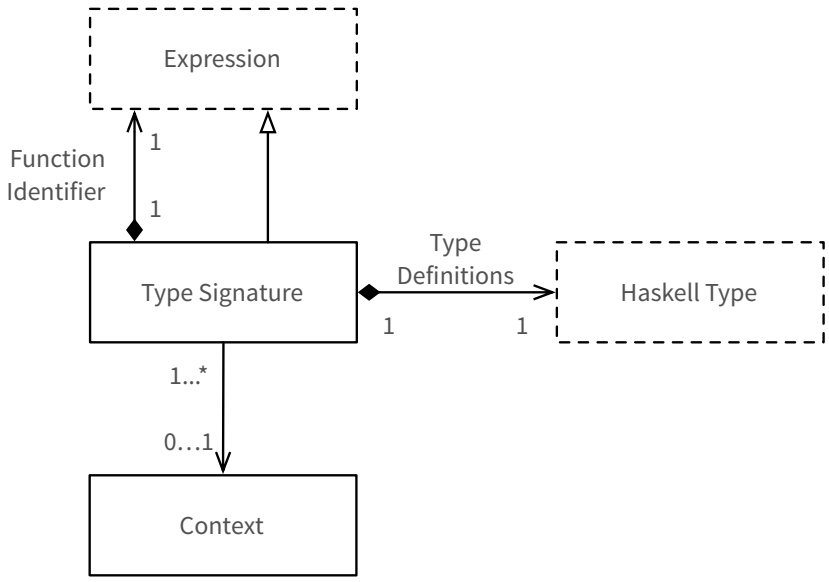


Abbildung 13: Haskell: Type Signature

Sie werden entweder benötigt, wenn der Compiler durch die Typinferenz Mehrdeutigkeiten entdeckt und der Benutzer die Möglichkeiten einschränken muss. Andererseits kann der Programmierer auch den eigenen Code erst spezifizieren und danach überprüfen, ob die Implementierung der Spezifikation genügt.

Context gibt hier den Kontext an, in dem die Funktion operiert. In Haskell sind dies meist Monaden – die Funktion führt Transformationen innerhalb des Monaden aus.

5.1.2 Expressions und Unterklassen

Nachdem die Grobstrukturen definiert sind, kommt nun der Kern der eigentlichen Sprache: die *Expressions*.

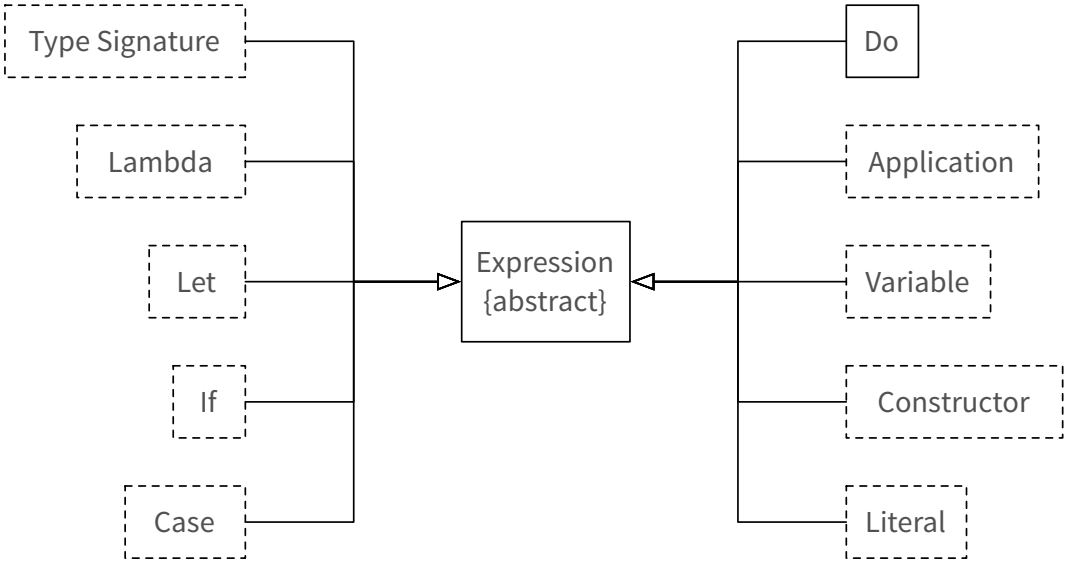


Abbildung 14: Haskell: Expressions[40]

Expressions sind der Kern von Haskell und bilden das Grundgerüst des AST. Die Unterklassen der Expressions sind die zentralen Sprachbestandteile, die den gesamten Programmfluss steuern.

Das *Do* Sprachkonstrukt dient der komfortableren Programmierung in Monaden. So können mehrere Expressions wie von anderen Programmiersprachen gewohnt einfach untereinander geschrieben werden und müssen nicht explizit mit den Monaden-Operatoren verbunden werden.

Im Schaubild nicht explizit aufgeführt sind Tupel- und Listen-Konstruktoren.

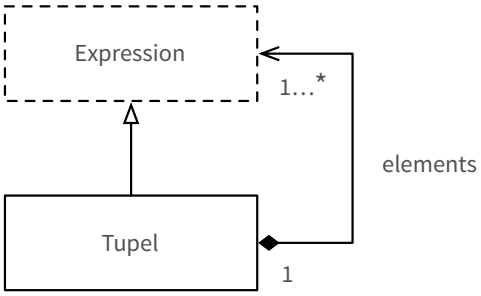


Abbildung 15: Haskell: Tuple

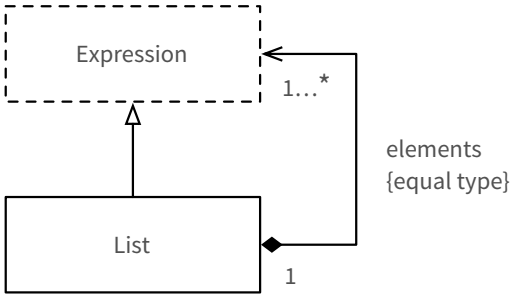


Abbildung 16: Haskell: List

Tupel und *Listen* sind eine (geordnete) Menge an Expressions mit dem einzigen Unterschied, dass Tupel eine feste Länge haben und beliebige Typen im Inneren besitzen. Eine Liste wiederum besitzt keine feste Länge, dafür müssen alle enthaltenen Elemente vom selben Typ sein.

[40] Die Anordnung der Elemente soll keine Gruppierung darstellen.

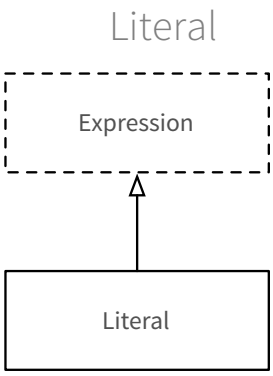


Abbildung 17: Haskell: Literal

Ein *Literal* stellt eine direkte Darstellung eines Wertes dar, beispielsweise Zahlen oder Zeichen.

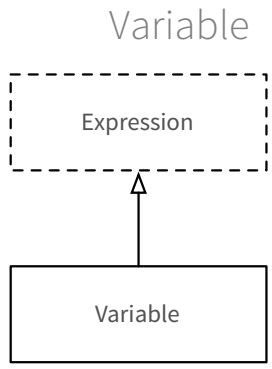


Abbildung 18: Haskell: Variable

Eine *Variable* ist grundlegend ein Name. So kann es sich beispielsweise um einen Funktionsnamen, einen Binding-Namen in einem Let-Block oder um den Namen eines benannten Case-Blocks handeln.

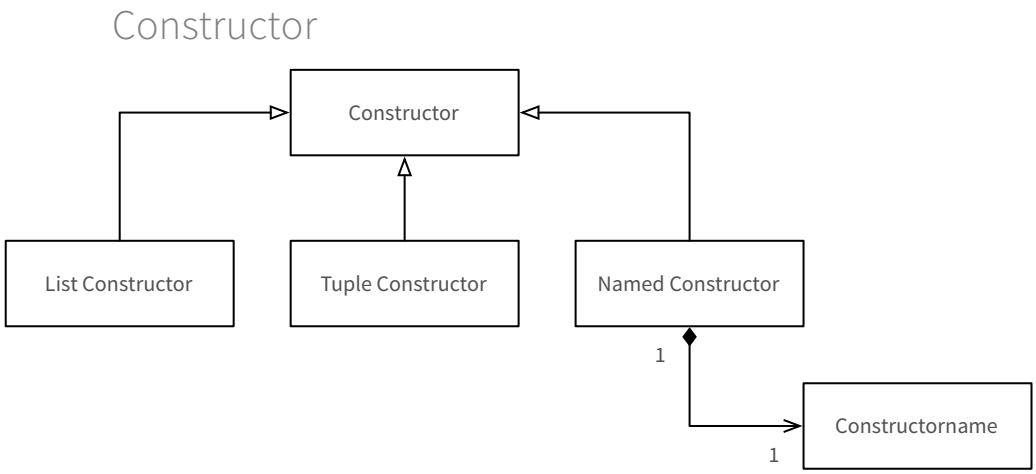


Abbildung 19: Haskell: Constructor

Ein allgemeiner *Konstruktor*. Kann entweder eine leere Liste oder ein leeres Tupel (oder beliebig langes Tupel ohne Werte „(,,,,)“) erstellen oder ein Name eines globalen Typkonstruktors sein.

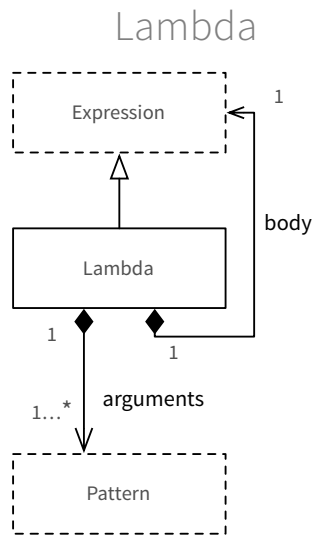


Abbildung 20: Haskell: Lambda

Eine *Lambda*-Definition stellt eine anonyme Funktion dar. Sie besteht aus einer Liste an Argumenten (*arguments*) und einem Funktionsrumpf (*body*). Der einzige Unterschied zwischen einer benannten Funktion und einem Lambda ist, dass die benannten Funktion zusätzlich einen Namen hat. Sobald man das Lambda einer Variablen zuweist sind sie jedoch anschließend äquivalent.

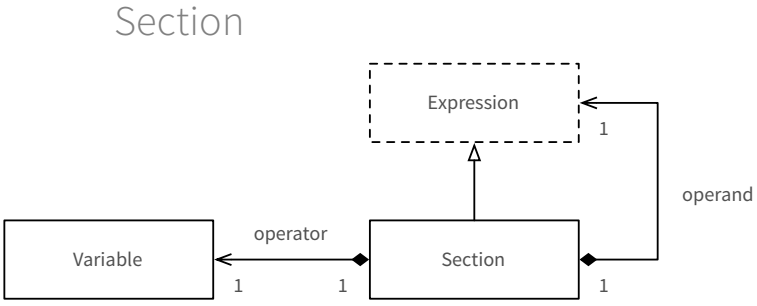


Abbildung 21: Haskell: Section

Eine *Section* ist ein gemeinsamer Spezialfall der Lambdadefinition und der partiellen Anwendung. Sie stellt eine Lambda-Definition eines *Operators* dar, bei dem eine Seite bereits gebunden ist. So gilt die folgende Gleichung:

```
(operator e) = \x -> x operator e
```

Links Section, rechts Lambda-Definition

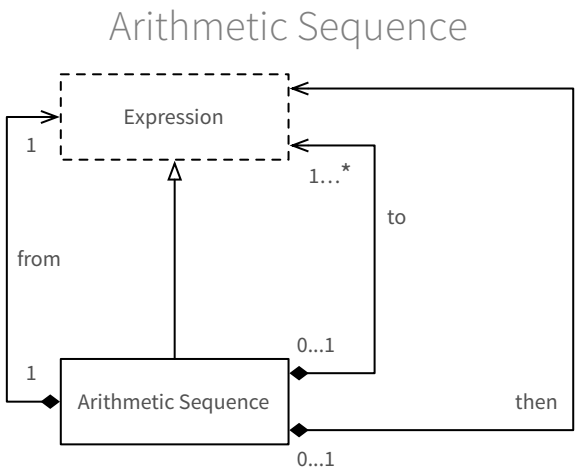


Abbildung 22: Haskell: Arithmetic Sequence

Eine *Arithmetic Sequence* ist ein Generatorionsmechanismus für Listen. So kann der Startwert (*from*), ein optionaler Folgewert (*then*) und ein optionaler Endwert (*end*, nur wenn auch *then* gegeben ist) angegeben werden. Die Folge beginnt beim Startwert, fährt mit dem Folgewert fort und wird anschließend so lange fortgeführt, bis der Endwert erreicht oder überschritten ist. Als Spezialfall können so auch unendliche Listen erzeugt werden.

```
-- erzeugt die Zahlen 1-10
[1, 2..10]

-- erzeugt eine unendliche Liste, beginnend bei 1
[1, 2..]
```

Arithmetic Sequence Beispiele

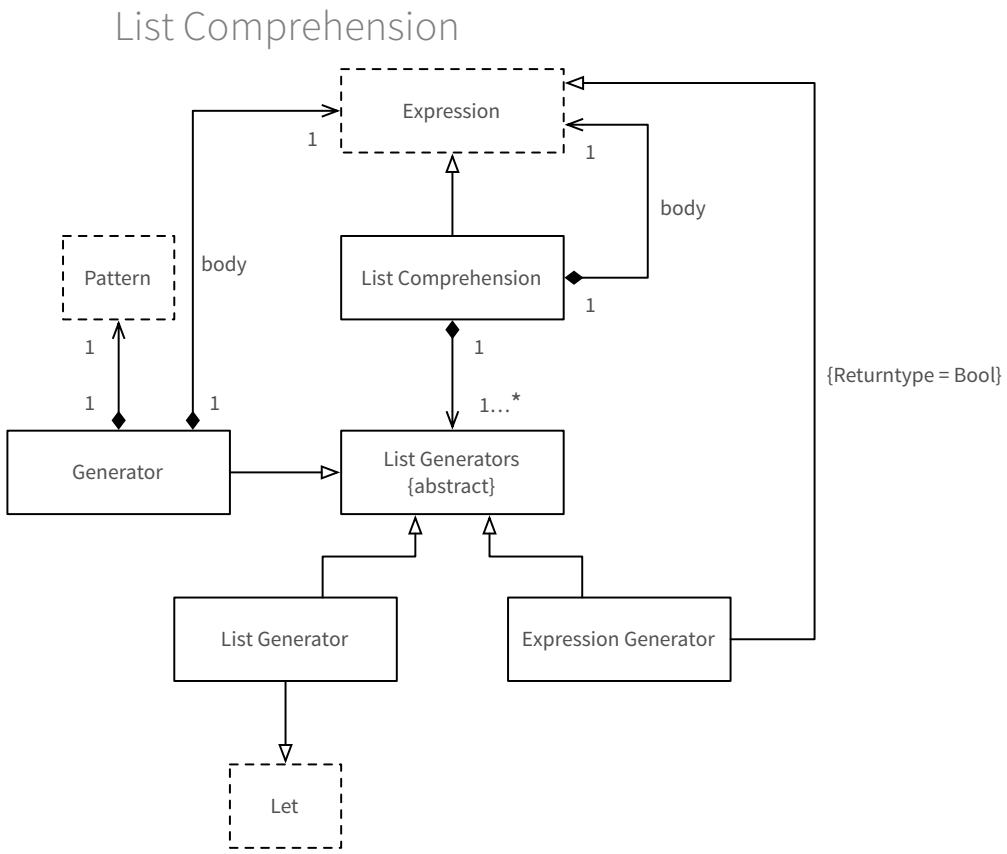


Abbildung 23: Haskell: List Comprehension

Eine List Comprehension beschreibt einen weiteren Weg, eine Liste zu füllen. Hierbei wird ein Pattern angegeben, anhand dieses dann die Liste gefüllt wird. Hierzu werden neben dem Pattern noch Generatoren angegeben, die die Werte für die einzelnen Variablen des Patterns erzeugen.

```
-- erzeugt: [(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
[(x, y) | x <- [1, 2, 3], y <- [4, 5]]
```

List Comprehension Beispiel

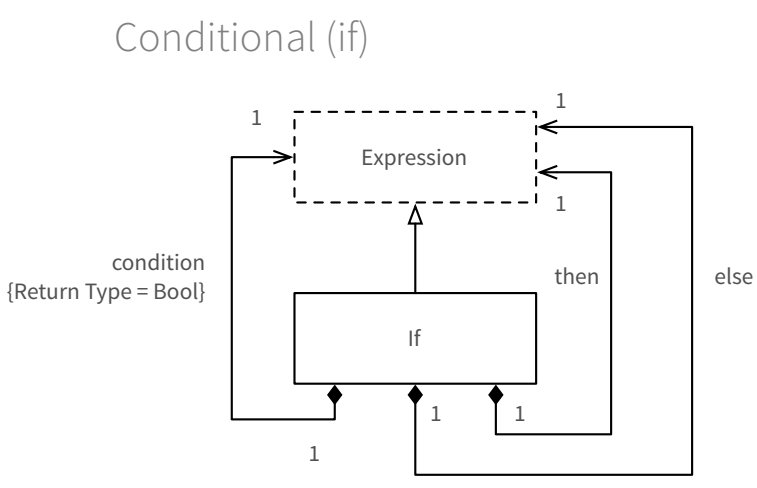


Abbildung 24: Haskell: Conditional

Ein *Conditional* ist eine if-Bedingung, wie sie in den meisten Programmier-Sprachen vorkommt. Allerdings gibt es zwei Besonderheiten: es gibt kein direktes elsif (es muss ein *if* im *else*-Zweig angelegt werden) und das Conditional muss vollständig sein. Dies bedeutet, dass immer ein *then*- und ein *else*-Zweig vorhanden sein muss. Dies ist eine direkte Folge aus der Forderung, dass jeder Ausdruck einen Rückgabewert hat.

Die eigentliche Bedingung (*condition*) ist eine allgemeine Expression, mit der Einschränkung, dass sie einen booleschen Wert zurückliefern muss.

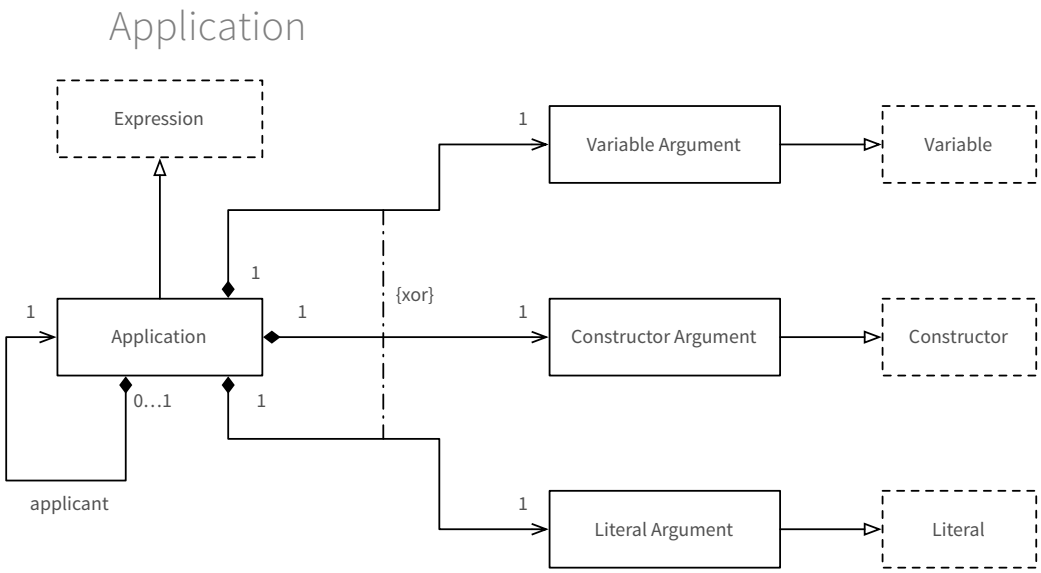


Abbildung 25: Haskell: Application

Eine *Application*, also ein Aufruf einer Funktion, weist einige Besonderheiten auf. Einerseits muss der Aufruf immer mit mindestens einem Argument geschehen. Dies ist logisch, da durch die Seiteneffektfreiheit ein Aufruf einer Funktion ohne Parameter nur einen konstanten Wert liefern kann (und dadurch intern nicht erst als Funktion dargestellt werden muss).

Andererseits muss der Aufruf immer mit *genau* einem Argument geschehen. Aufrufe mit mehreren Argument sind also in Wirklichkeit mehrere sequenzielle Aufrufe mit jeweils einem Argument.

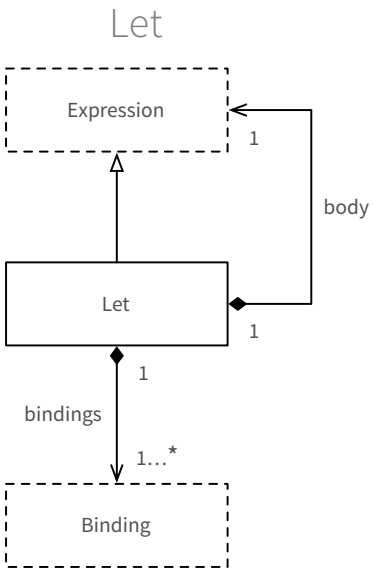


Abbildung 26: Haskell: Let

Eine Let-Bindung führt lokale Variablen und Umbenennungen ein. Let und where (siehe Binding) sind bis auf einen Unterschied identisch: ein Let-Binding ist lokal begrenzt, ein where-Ausdruck gilt immer für die gesamte Funktion.

Beim Let-Binding werden lokale *Bindings* eingeführt (lokale Namen), die dann für den Rumpf (*body*) gelten.

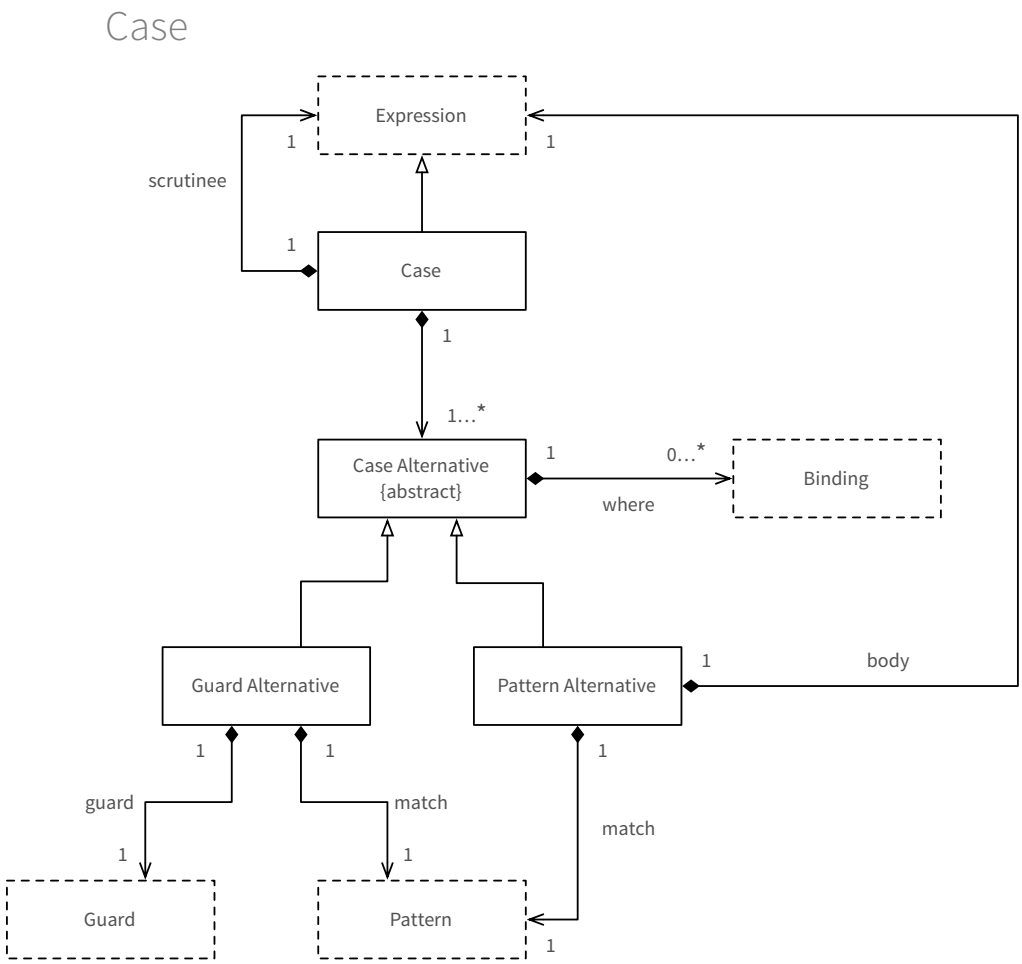


Abbildung 27: Haskell: Case

Die *Case*-Unterscheidung ist der vermutlich mächtigste Expression-Typ[41]. Sie besteht im Wesentlichen aus einem *scrutinee*[42] , dessen Wert die verwendete Alternative bestimmt. Die Alternativen sind entweder Guards oder Pattern Matchings. Außerdem kann optional noch ein globales *where* definiert werden, das Bindings, die über alle Alternativen gültig sind, einführt.

[41] Zumindest sehen das die Core-Forscher so und wandeln viele Konzepte aus Haskell in Case-Konstrukte um, siehe 5.4
[42] Dies ist eine Bezeichnung, die direkt aus dem GHC Kommentar entnommen ist und auf Deutsch in etwa „zu prüfendes Element“ bedeutet. [16]

5.1.3 Wichtige Unterklassen

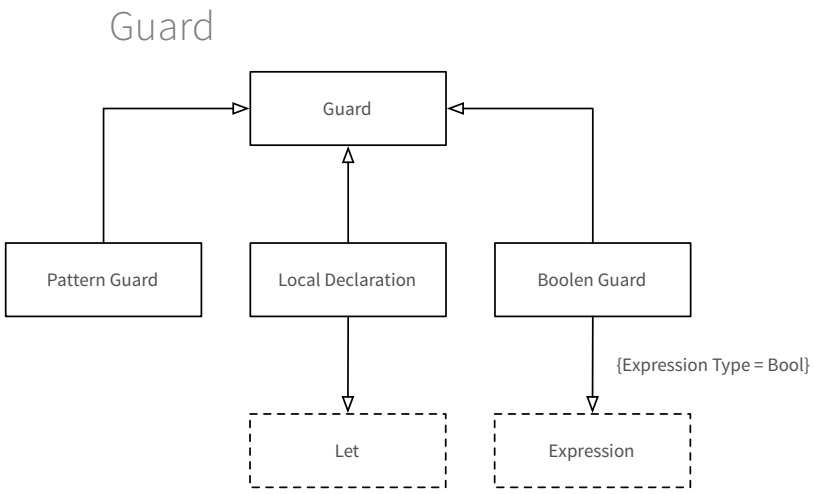


Abbildung 28: Haskell: Guard

Ein Guard wird für das Matching von Werten verwendet und ist entweder ein boolescher Ausdruck, eine lokale *Let*-Deklaration oder ein *Pattern*. Beim *Pattern Guard* wird ein Pattern mit einer gegeben Expression verglichen und entsprechend ausgewertet.

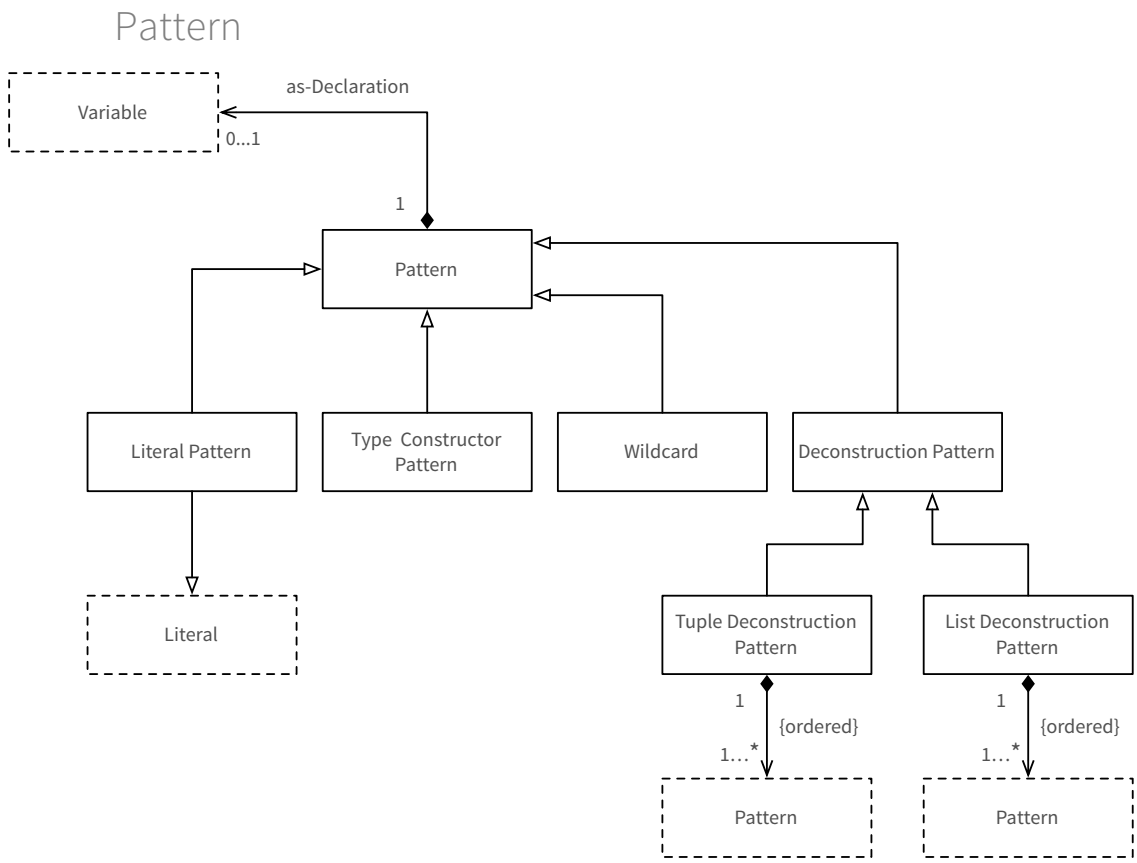


Abbildung 29: Haskell: Pattern[43]

Patterns werden an vielen Stellen in der Sprache verwendet und dienen dem Vergleichen von verschiedenen Werten. Sie lassen insbesondere aber die Typ-Dekonstruktion zu, die ein komfortabler Weg ist, um zusammengesetzte Typen zu zerlegen (zum Beispiel algebraische Datentypen, Listen oder Tupel).

5.2 Metamodell von Core

Aus der Haskell-Sprache und dessen Metamodell wird im GHC zunächst Core erzeugt. Core ist eine Zwischensprache, die viele Konzepte von Haskell vereinheitlicht, so dass die Vielfalt der Konzepte deutlich geringer ist – was die weitere Verarbeitung des Programms vereinfacht. Nichtsdestotrotz können alle Konzepte von Haskell in Core abgebildet werden. Alle Erweiterungen von Haskell über Core sind reiner „syntaktischer Zucker“, die die Programmierung in der Sprache angenehmer gestalten sollen und keinen funktionalen Mehrwert bieten.

The existence of Core has also proved to be a tremendous sanity check on the design of the source language. Our users constantly suggest new features that they would like in the language. Sometimes these features are manifestly "syntactic sugar", convenient new syntax for something you can do already. But sometimes they are deeper, and it can be hard to tell how far-reaching the feature is.

---

[43] Pattern ist hier nochmal explizit als Referenz aufgeführt, auch wenn die Klasse in diesem Schaubild definiert ist. Dies hat rein pragmatische Gründe: die Verbindungen quer durch das Schaubild hätten das Verständnis gestört.

Core gives us a precise way to evaluate such features. If the feature can readily be translated into Core, that reassures us that nothing fundamentally new is going on: the new feature is syntactic-sugar-like. On the other hand, if it would require an extension to Core, then we think much, much more carefully. – Simon Marlow und Simon Peyton-Jones[44]

Core vereinfacht viele Sonderfälle von Haskell, generalisiert aber gleichzeitig das Metamodell und den AST. Dies hat pragmatische Gründe: es macht die Definitionen im AST einheitlicher und senkt die Anzahl der Typkonstruktoren erheblich. Dafür hat man nun jedoch Feinheiten, die dadurch, dass allgemeine Expressions verwendet werden, so eigentlich nicht gegeben sind. Ein Beispiel[45] ist der Typ „Arg“ also ein Argument einer Applikation. Während Arg einfach als Synonym für Expression definiert ist, kann nur Arg als erstes Element eine Typdefinition haben. Während syntaktisch jede Expression eine Typdefinition einführen kann, ist es semantisch für das erste Argument nur so festgelegt, dass das nur für Argumente erlaubt ist.

5.2.1 Übersicht

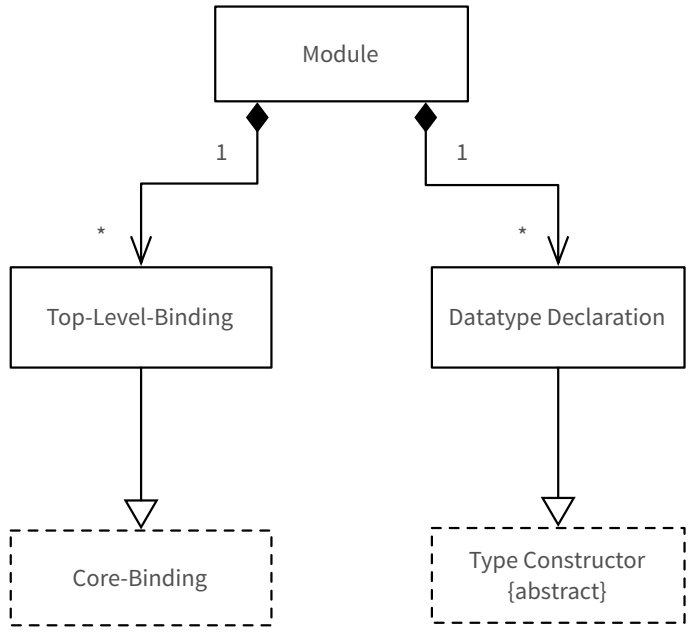


Abbildung 30: Core: Module

Ein *Modul* besteht, ähnlich zu Haskell, aus Typ-Deklarationen und Top-Level *Bindings*.

[44] Aus [2]
[45] Aus dem Kommentar zum Typ „Arg“ in [17]

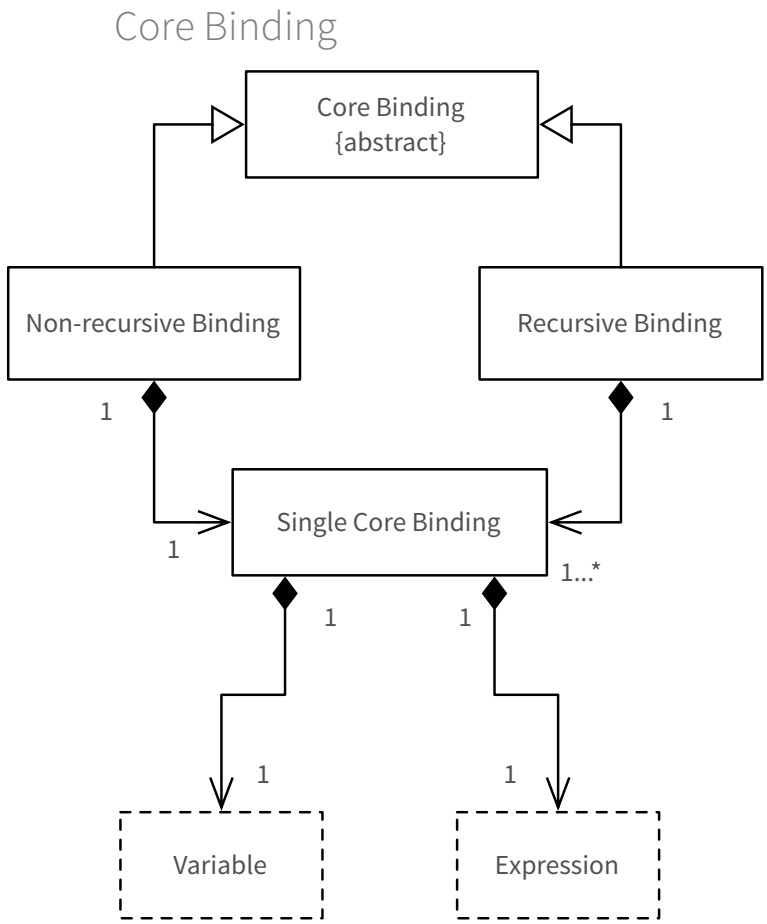


Abbildung 31: Core: Core Binding

Ein *Core-Binding* kann, ähnlich zu Haskell, rekursiv oder nicht-rekursiv sein. Hier hat sich die Definition allerdings erheblich vereinfacht: ein einzelnes Binding (beispielsweise eine Funktionsdefinition) besteht nur noch aus einem Namen (*Variable*) und einem Rumpf (*Expression*).

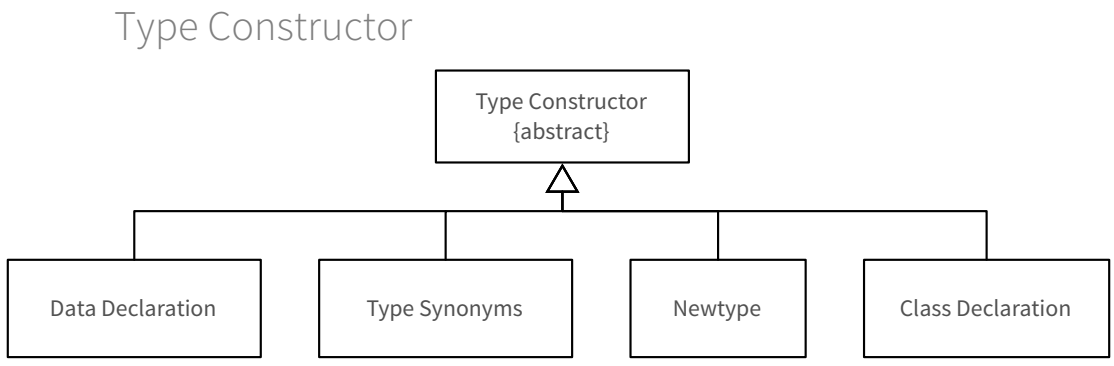


Abbildung 32: Core: Type Constructor

Type Constructors sind, bis auf Benennungsunterschiede zu Haskell, weitgehend unverändert. Es ist jedoch möglich, dass durch Inlining einzelne Typ-Konstruktor nun verschwunden sind, da alle Verweise inline verarbeitet werden konnten[46].

5.2.2 Expressions und Unterklassen

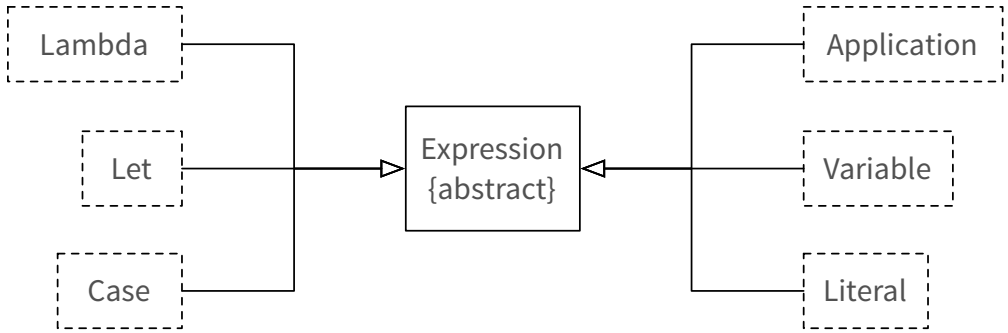


Abbildung 33: Core: Expression

Expressions sind im prinzipiellen Aufbau zu Haskell unverändert, es sind nur einige Fälle entfallen. Die Details zu den entfallenen Fällen wird im Kapitel 5.4 beschrieben. Expressions hat noch weitere Unterklassen, die hier nicht aufgeführt werden, namentlich *Tick*, *Type* und *Coercion*.

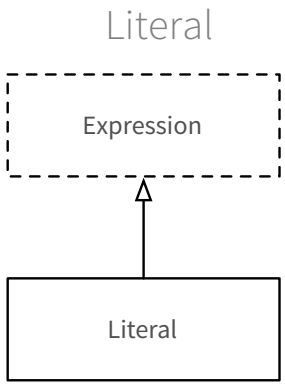


Abbildung 34: Core: Literal

Ein *Literal* ist die direkte Darstellung eines Wertes (in Core hat sie etwas an Abstraktionsgrad verloren und ist nun intern näher an der Hardwareebene, dies hat auf das Modell jedoch keinen Einfluss).

[46] Vergleiche 7.3

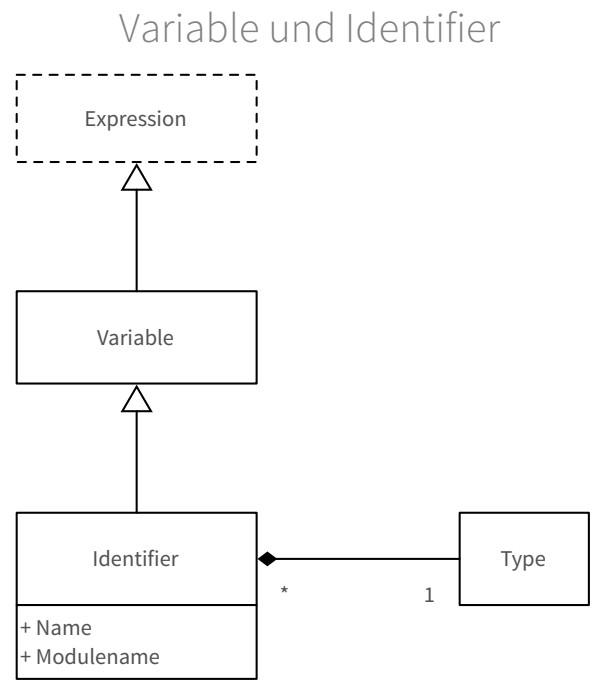


Abbildung 35: Core: Variable

Eine *Variable* ist im Grunde ein Name (*Identifier*) mit einem Typ und etwas erweiterten Informationen über die Variable und ihre Verwendungen.

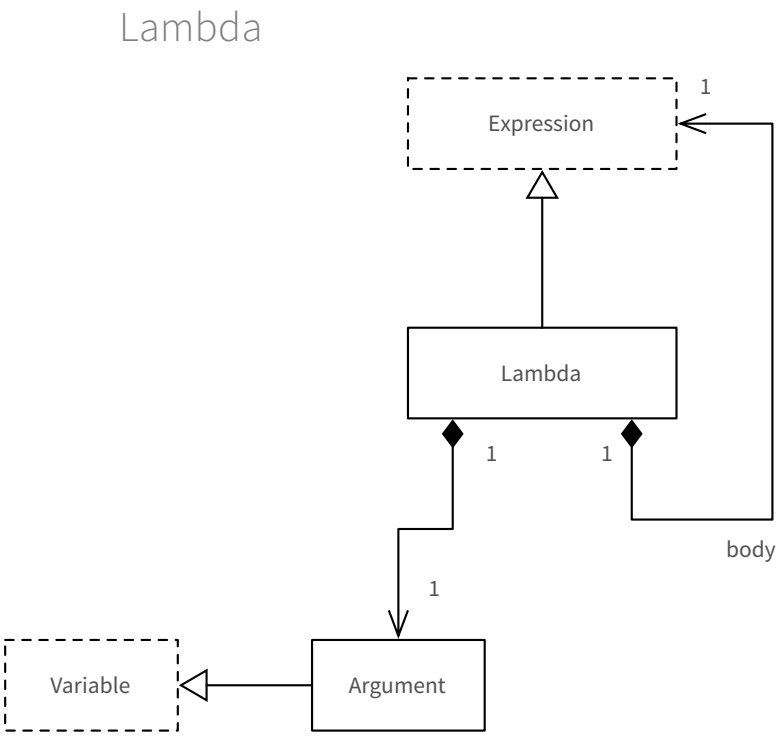


Abbildung 36: Core: Lambda

Lambda-Definition wurden im Vergleich zu Haskell etwas verändert. Sie bestehen nun aus genau einem Parameter und einer Expression als Rumpf (*body*).

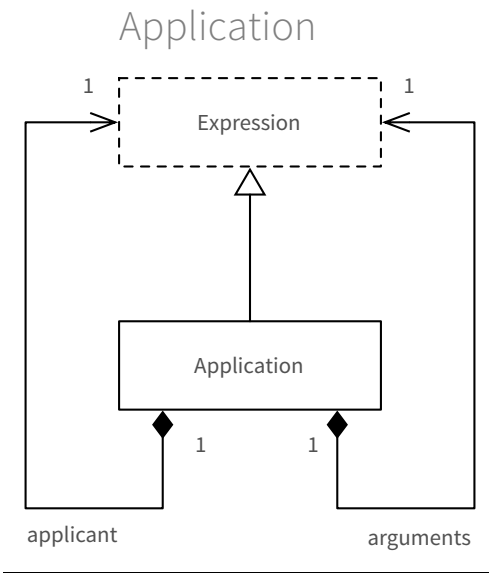


Abbildung 37: Core: Application

Applications verbleiben im Vergleich zu Haskell unverändert. Genau ein Argument und eine Expression als *body*. Der Ausdruck des Aufrufenden (*applicant*) kann entweder trivialerweise ein Name sein (als Spezialfall der Expression), oder auch eine allgemeine Expression. Dies ermöglicht direkte verkette Aufrufe (zum Beispiel durch Currying).

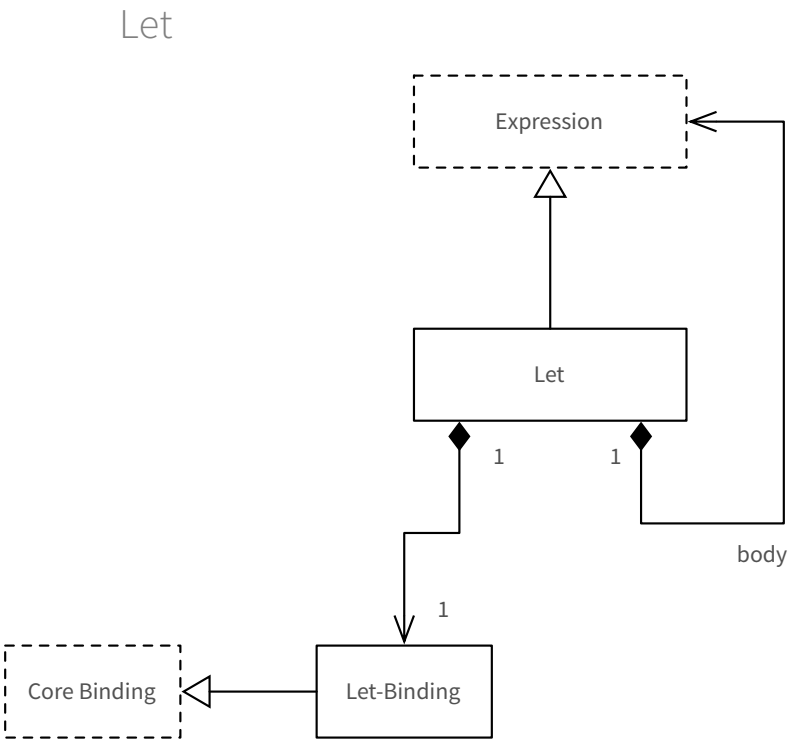


Abbildung 38: Core: Let

Let besteht aus einem Core Binding und dem Rumpf, in denen die Bindings gelten. Es führt in den Bindings lokale Namen und Umbenennung ein, die im Body Gültigkeit besitzen.

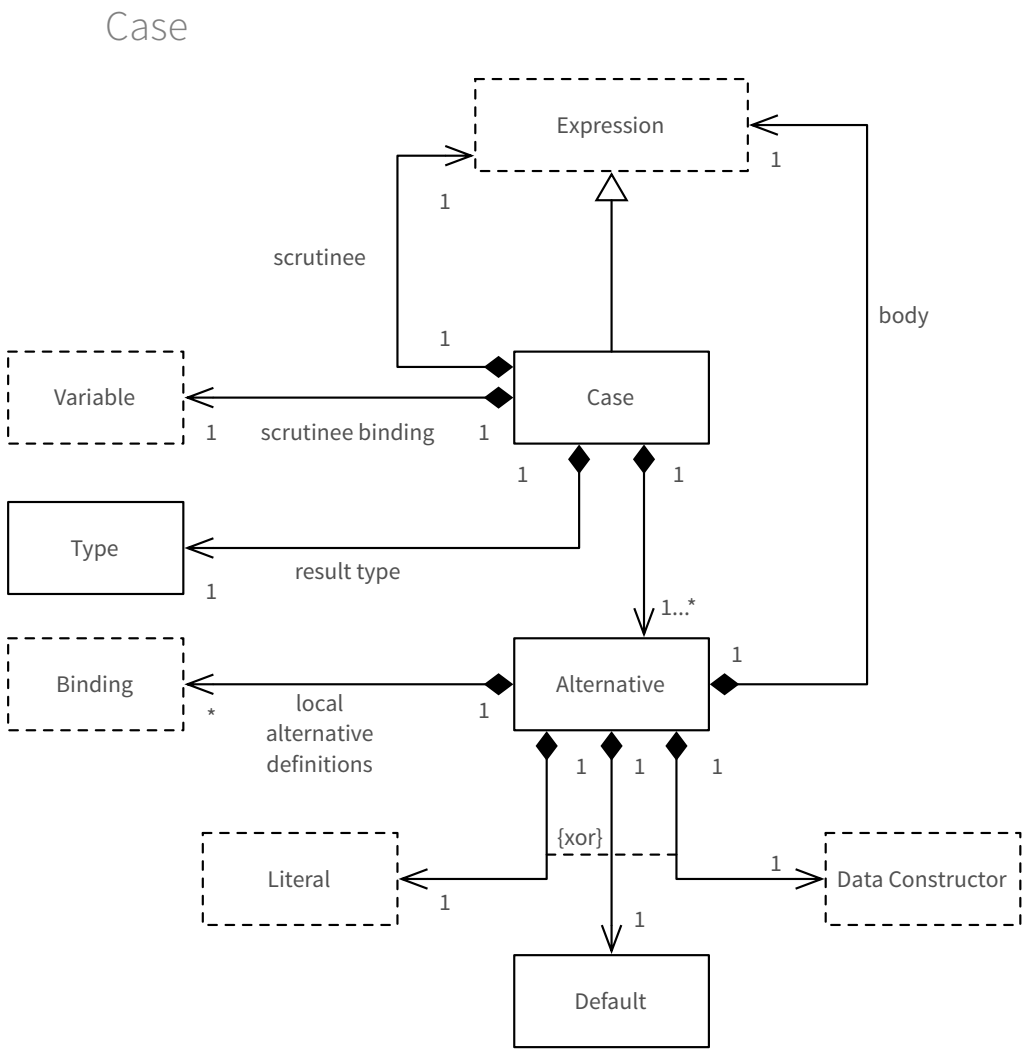


Abbildung 39: Core: Case

Auch das *Case*-Statement besteht im Wesentlichen aus den Elementen, die es bereits in Haskell hatte. Es hat immer noch einen *scrutinee*, der die auszuführende Alternative bestimmt, sowie ein *Binding*, dass das Ergebnis der scrutinee-Auswertung bindet. Die Alternativen haben jeweils einen *body* und eine Möglichkeit, lokale weitere Alternativen zu definieren. Forthin kann jede Alternative als Guard entweder ein *Literal* oder einen *Data Constructor* besitzen, oder immer gültig sein (*Default*).

5.3 Metamodell von Ruby

Das Metamodell von Ruby wird nur in dem Umfang behandelt, in dem es benötigt wird, um einerseits die Core-Metamodelle in Ruby zu transformieren. Dies umfasst nur einen sehr kleinen Teil der Konzepte der Sprache, was allerdings nur bedeutet, welche Mächtigkeit diese bereits besitzen.

Das Metamodell wurde zu großen Teilen aus der Grammatikdefinition der offiziellen Distribution erstellt, einige Details wurden in der Ausführung validiert.

5.3.1 Übersicht

Programm

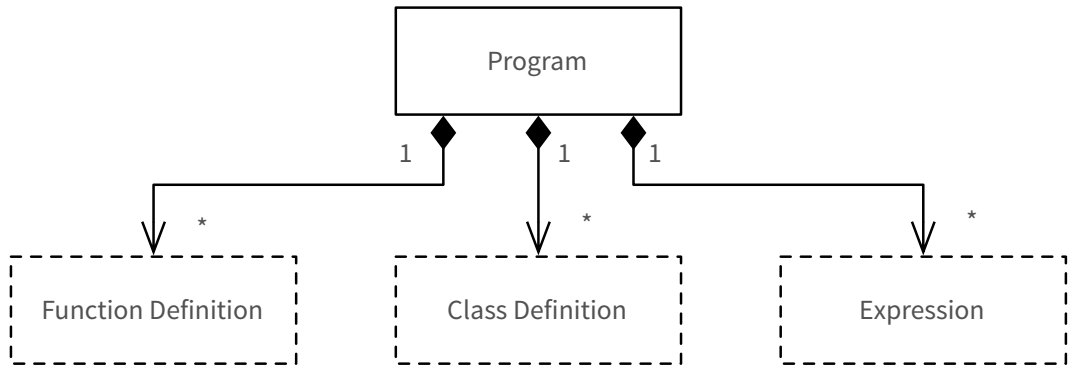


Abbildung 40: Ruby: Program

Ein Ruby-*Programm* besteht aus Modulen, Klassen, Funktionen und Expressions auf der obersten Ebene. Es sind also deutlich weniger Restriktionen, als das bei den anderen Sprachen der Fall war.

Function Definition

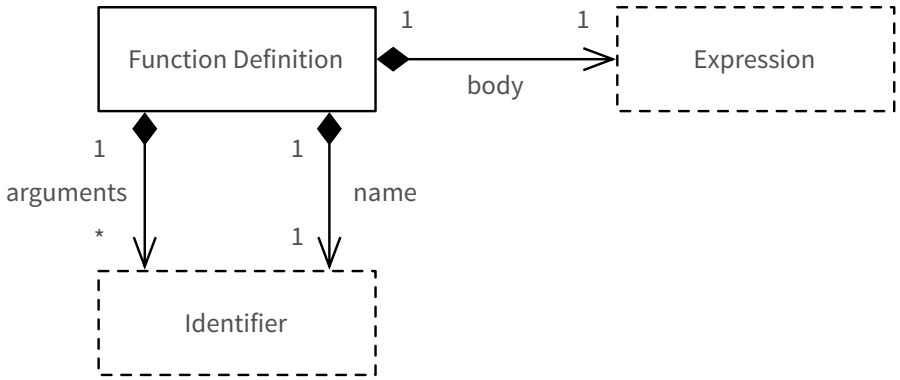


Abbildung 41: Ruby: Function Definition

Eine *Funktionsdefinition* in Ruby besteht aus einem *Namen*, *Argumenten* und einem *Body*. Ruby erlaubt außerdem Modifikatoren auf den Argumenten, wie beispielsweise den Splat-Operator „*“, der alle weiteren Argumente sammelt, in einem Array zusammenfasst und in die Funktion gibt.

```
def fun (*args)
  p args
end

fun 2, 3, 4, 5
# > [2, 3, 4, 5]
```

Einsatz des Splat-Operators

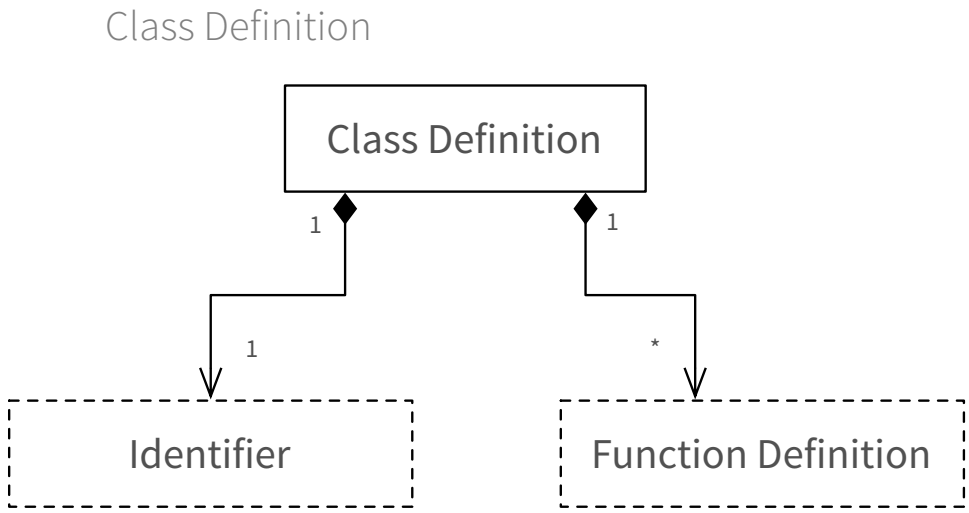


Abbildung 42: Ruby: Class Definition

Class Definition umfasst im Wesentlichen zunächst einen Namen für die Klasse. Das Klassenkonzept ist in einer objektorientierten Sprache natürlicherweise sehr umfangreich, hier wird aber nur der Teil vorgestellt, der für die tatsächliche Abbildung der Konzepte benötigt wird.

Klassen werden im erzeugten Code nur für die Implementierung des Typ-Systems verwendet und dort auch nur, um den Array-Typ mit eigenem Namen zu redefinieren.

5.3.2 Expression und Unterklassen

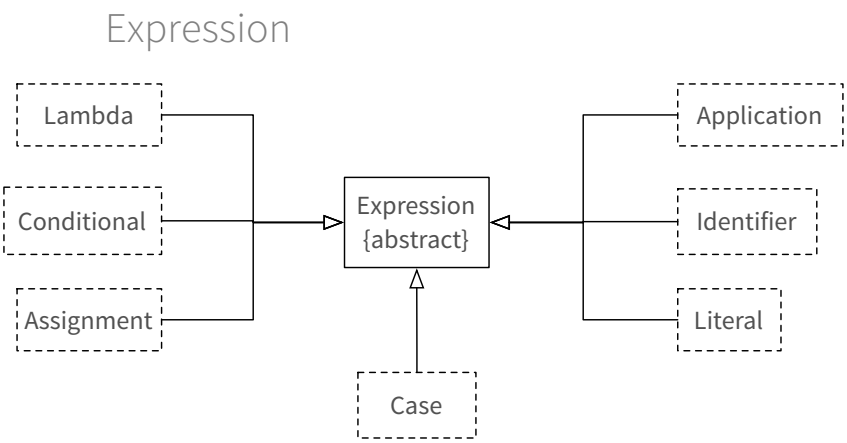


Abbildung 43: Ruby: Expression

Die Unterklassen der *Expression* sind zumindest namentlich weitgehend deckungsgleich mit denen, die auch Haskell unterstützt. Funktionale Expressions, wie *Let* fallen weg, dafür kommen imperative Konzepte wie *Assignment* hinzu.

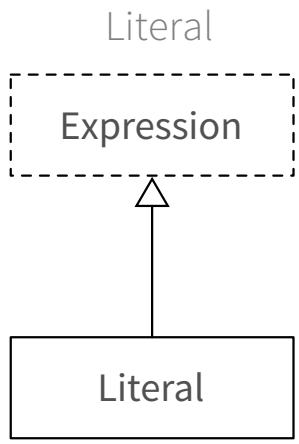


Abbildung 44: Ruby: Literal

Ein *Literal* in Ruby ist, wie alle anderen Elemente auch, ein Objekt. Es wird aber regulär über ein Literal im Quellcode definiert, dort gibt es also keine Änderung (wichtig für die Codeerzeugung).

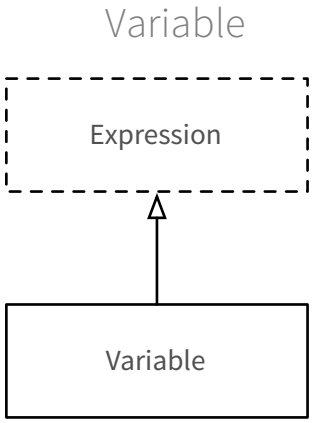


Abbildung 45: Ruby: Variable

Eine *Variable* in Ruby ist im Wesentlichen ein Namen. Durch die sehr dynamische Natur von Ruby kann über die Variable zur Laufzeit allerdings wesentlich mehr herausgefunden werden, als von einem kompilierten Haskell-Programm.

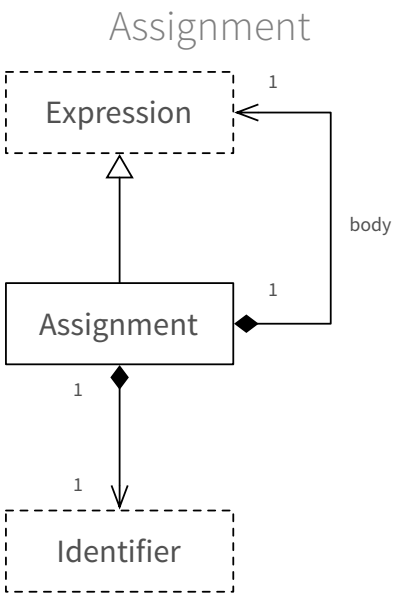


Abbildung 46: Ruby: Assignment

Eine Zuweisung (*Assignment*) besteht aus der *Expression*, deren Wert anschließend an einen *Identifier* zugewiesen wird.

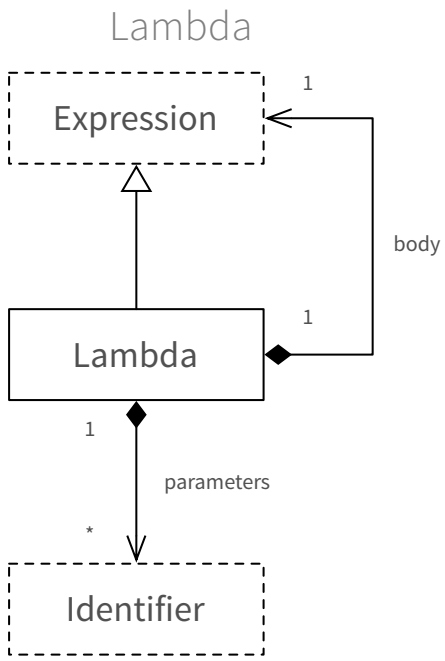


Abbildung 47: Ruby: Lambda

Eine *Lambda*-Definition funktioniert in Ruby über die Instanzierung eines Proc-Objekts. Konzeptuell bleibt aber, dass ein Lambda aus einer Expression als *body* und einer beliebigen Anzahl *Parametern* besteht.

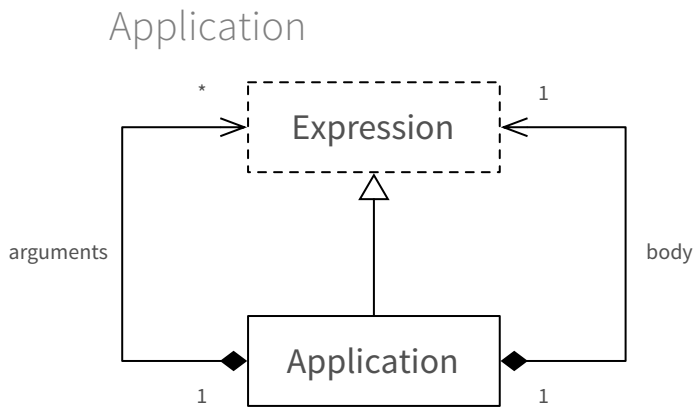


Abbildung 48: Ruby: Application

Ein Aufruf (*Application*) in Ruby besteht aus einer Expression, deren Ergebnis mit Argumenten aufgerufen wird.

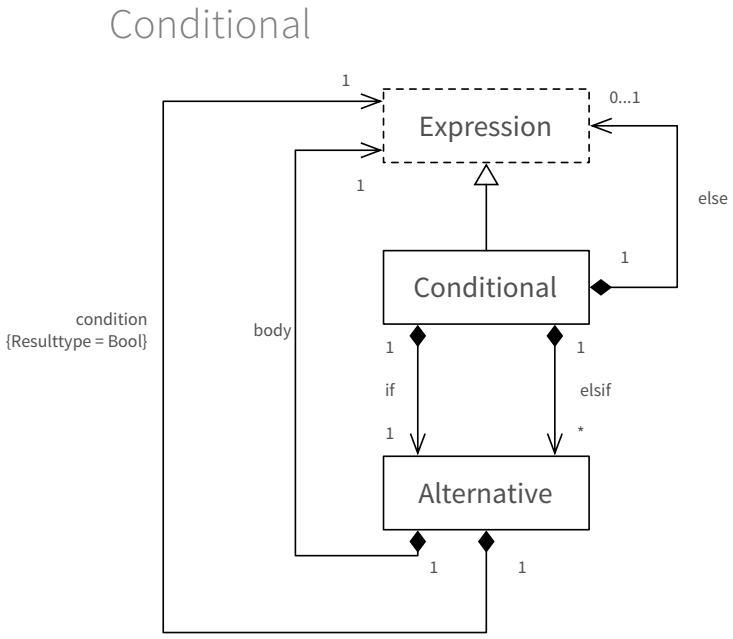


Abbildung 49: Ruby: Conditional

Ein *Conditional* ist eine *if-else*-Kette bestehend aus Bedingungen und Expressions als Rumpf des Zweigs. *Elsif* wird unterstützt.

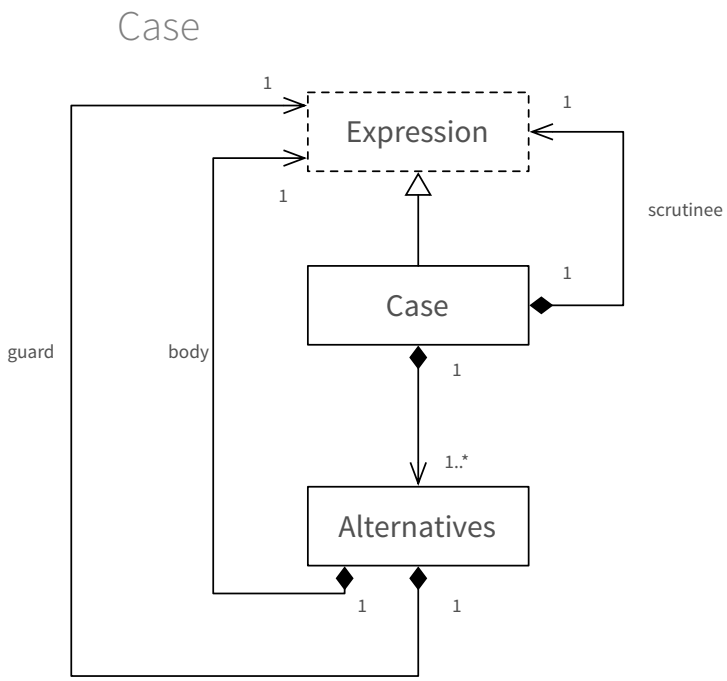


Abbildung 50: Ruby: Case

Case existiert in Ruby auch, hat jedoch eine Besonderheit. Beim Case wird sequenziell jeder *guard* mit dem Wert des *scrutinee* verglichen, in Code ausgedrückt „guard == scrutinee“. Dies schränkt die Ausdruckskraft der Case-Expression im Vergleich zu Haskell leicht ein.

5.4 Transformation Haskell zu Core

Prinzipiell ist die Transformation von Haskell zu Core eine Reduktion. Es fallen Konzepte weg, es kommen jedoch keine neuen hinzu. Dies dient vor allem einer Verringerung der Komplexität der Konzepte der Sprache und des Umfangs für die weitere Behandlung innerhalb des GHCs. Diese Veränderungen verringern jedoch nicht die Mächtigkeit der Sprache.

5.4.1 Gemeinsame Aspekte

Da Haskell eine Erweiterung von Core ist, ergeben sich natürlich viele gemeinsame Konzepte.

Let-Bindings, Variablen, Literale, Applikationen und die Modulstruktur, sowie alle nativen Datentypen sind unverändert. Auch das Typsystem bleibt bis auf eventuelles Inlining selten verwendeter Typen erhalten.

Top-Level-Bindings bleiben, ebenso wie Case-Expressions, bis auf die Vereinfachung der Bezeichnerausdrücke ebenfalls erhalten.

5.4.2 Unterschiede

Im Folgenden werden die Transformationen der einzelnen Aspekte, die in der Originalform nicht mehr vorhanden sind, beschrieben.

Pattern Matches und Guards

Patterns und Guards sind propagiert. Dies bedeutet, dass die geschachtelten Pattern Matches nun geschachtelte Case-Konstruktionen sind, die jeweils einzeln auf die Struktur der Elemente überprüfen. Auch Pattern Matching in Funktionsargumenten wurde in den Funktionsrumpf als Case-Anweisung propagiert.

Die Typinferenz ist beendet und die Typen wurden den Bezeichnern und Ausdrücken zugewiesen. Die Typdefinitionen sind, bis auf die Konstruktoren, aus dem AST entfernt.

List Comprehensions

List Comprehensions können direkt durch Aufrufe von map, filter und concat umgewandelt werden, es sind keine gesonderten Sprachfunktionen notwendig. Bei der Implementierung wird über alle generierten Werte mit *Map* iteriert, ungültige Kombinationen ge*filter*t, die Kombinationsfunktion (definiert durch das Pattern) angewendet und die Ergebnisse zu einer Liste mittels *concat* zusammengefasst.

Arithmetic Sequences

Genau wie List Comprehensions können auch Arithmetic Sequences sehr einfach in der Laufzeitumgebung implementiert werden. So wird dies intern von Core durchgeführt; die Generatoren aus Haskell werden zu simplen Aufrufen interner Laufzeitfunktionen von Core.

Conditional

Da ein If-Else-Konstrukt einen Spezialfall einer Case-Anweisung darstellt (mit den Alternativen True und False) kann dieses Konzept trivial transformiert werden.

Sections

Diese werden in Lambda-Definition umgewandelt werden (siehe das Kapitel zu Sections in der Metamodelldefinition von Haskell).

Lambda-Definition

Diese werden so transformiert, dass jede Definition maximal ein Argument hat. Auch dies ist trivial möglich, indem man für die Anzahl an Argumenten Lambda-Ausdrücke schachtelt, von der die innerste den eigentlichen Rumpf der Funktion erhält und jede Ebene das Argument aus der ursprünglichen Argumentsliste, das ihrer Schachtelungstiefe entspricht.

```
lambda { |a, b, c|
  a + b + c
}

# wird umgewandelt zu

lambda { |a|
  lambda { |b|
    lambda{ |c|
      a + b + c
    }
  }
}
```

Lambda Umstrukturierung beispielhaft in Ruby-Quellcode skizziert.

5.5 Transformation Core zu Ruby

Während die Transformation von Haskell zu Core noch innerhalb des gleichen Paradigmas statt fand und demnach keine großen konzeptuellen Veränderungen zu erwarten waren, könnten hier bei Ruby mehr Unterschiede auftreten.

Wie sich herausstellen wird, kann die Transformation jedoch trotzdem ohne komplexe Teilschritte durchgeführt werden.

5.5.1 Gemeinsame Aspekte

Direkt übernommen werden können alle Aspekte auf einer niedrigen Ebene wie Literale und Variablen. Die unterschiedliche interne Darstellung der Literale ist dabei keine konzeptuelle Differenz, sondern eine technische[47]. Prinzipiell könnten auch Lambda-Definition fast unverändert übernommen werden, aber im Folgenden wird versucht, dies noch zu optimieren.

5.5.2 Transformation der einzelnen Core-Aspekte

Binding

Ein Binding wird in Ruby direkt zu einer globalen Methoden-Definition (für Typdefinition siehe „Transformation des Typsystems“).

Transformation des Typsystems

Das Typsystem wird in Ruby-Klassen transformiert, da Typen in objektorientierten Sprachen von Klassen repräsentiert werden.

Hierzu wird für jeden Typ-Namen eine Klasse als Unterklasse von Array erstellt, die als Konstruktor-Parameter ein Array von Expressions nimmt. Dieses Array stellt die Typ-Konstruktor-Parameter dar. Auf diesem Weg sind sowohl geschachtelte algebraische Datentypen möglich, als auch die Typ-Überprüfung mittels Überprüfung der Klasse der Typvariable.

Dies ist ein vereinfachendes Verfahren, dass für die Transformation der Sprache jedoch genügt: die Typen werden bei der Code-Generierung im Grunde nur noch für Darstellungswechsel benötigt, aber nicht mehr um Typprüfung zu betreiben. Diese wurde bereits in Haskell statisch durchgeführt und bei der Transformation wird die Typsicherheit erhalten bzw. das Typsystem wird sogar permissiver. Typfehler zur Laufzeit sind also nicht möglich.

Case

Das *Case*-Statement kann in der Form aus Haskell nicht direkt in Ruby abgebildet werden, da durch die Bindung des Vergleichs mittels „==“ dies die Vergleichsmöglichkeiten leicht eischränkt[48].

Die Lösung ist jedoch einfach, da ein Case-Ausdruck in Ruby semantisch sein Spezialfall einer if-elsif-else-Kaskade ist. Die Transformation ist dann unkompliziert: die Alternativen werden zu if-elsif

[47] Übrigens auch Themen wie unterschiedliche Wertebereiche für native Datentypen.
[48] Im Grunde sind dies Implementierungsdetails, die allerdings zwingend Änderungen des Ziel-Metamodells zur Folge haben.

Zeigen, die optionale Default-Alternative zum else-Zweig. Die Vergleiche in den if-Bedingungen werden den Alternativen selbst überlassen, sie erhalten nur den scrutinee-Identifier und können mit diesem beliebige Vergleiche generieren.

Let

Während der Body der Let-Bindings direkt übernommen werden kann, muss der Binding-Block in eine (sortierte) Liste von Assignments transformiert werden. Als Besonderheit könnte man den Body und die Assignments in einen direkt ausgeführten Lambda-Block aus der Sichtbarkeit nehmen um auch das Konzept der lokalen Namen direkt abzubilden.

Dies ist allerdings nicht notwendig, da GHC in Core global eindeutige Namen vergibt, die bei der Code-Erzeugung verwendet werden.

Application und Lambda

Während die einstelligen Applications und Lambda-Definition sich zwar in dieser Form auch in Ruby darstellen lassen, kann auch versucht werden, etwas „typischeren“ imperativen Code zu erzeugen.

Hierzu können direkt geschachtelte Lambda-Blöcke wieder in einen Block mit mehreren Parametern transformiert werden; sowie bei den Aufrufen direkt aufeinanderfolgende Argumente zusammengefasst werden. Dies ist keine zwingend notwendige Transformation, erzeugt aber eine Struktur, die mehr mit der Idee einer sauberen Ruby-Implementierung zu tun hat[49].

Wie sich herausstellt, ist diese Transformation allerdings zunächst unsinnig, da je nach Currying-Implementierung diese Transformationsschritte in der Code-Generierung wieder rückgängig gemacht werden. Trotzdem lohnt sich die Transformation, da der Mehraufwand in der Code-Generierung trivial ist, aber die Analyse und Optimierung des Ruby-ASTs vereinfacht wird.

6 Implementierung des Compilers

Ähnlich wie Fay[50] geht Haru nicht den Weg über STG, sondern kompiliert aus Core direkt zu Ruby. Einerseits, weil Core ein besseres theoretisches Fundament darstellt und man daher die Modellierung darauf aufbauen kann, andererseits weil die Dokumentation zu Core bedeutend besser ist. Dies ist ein nicht zu unterschätzender Faktor in der Implementierung eines Systems. Ein Nachteil ist, dass einige Typoptimierungen dadurch noch nicht durchgeführt wurden – dies ist für die Codegenerierung in Ruby aber nicht hinderlich beziehungsweise kann, soweit benötigt, auch selbst durchgeführt werden.

Haru trifft einige vereinfachende Annahmen:

- Typisierung der Literale wird fast komplett ignoriert. Das Eingabeprogramm wurde bereits als typkorrekt analysiert, die Literale werden umgewandelt und das restliche Typsystem ist soweit kompatibel (bis auf Listen und Strings – dies wird allerdings in der Runtime gelöst).
- Das restliche Typsystem wird mittels Arrays von Werten abgebildet. Auch hier gilt: das Programm wurde bereits als typkorrekt bestätigt, wie die Typen intern dargestellt werden, ist irrelevant. Das Hauptaugenmerk liegt auf der korrekten Konstruktion und Dekonstruktion der geschachtelten Datentypen, die mit Arrays sehr einfach implementiert ist.

[49] Diese Idee hinter dem Ruby-Code ist dann ein informales Metametamodell.
[50] Siehe 4.4.5

Die Implementierung von Haru transformiert den GHC AST in einen Ruby AST[51] und diesen dann zu Ruby Quellcode. Die Runtime ist eine statische Ruby-Datei und wird nach der Quellcode-Generierung einfach vor jedes Modul kopiert[52].

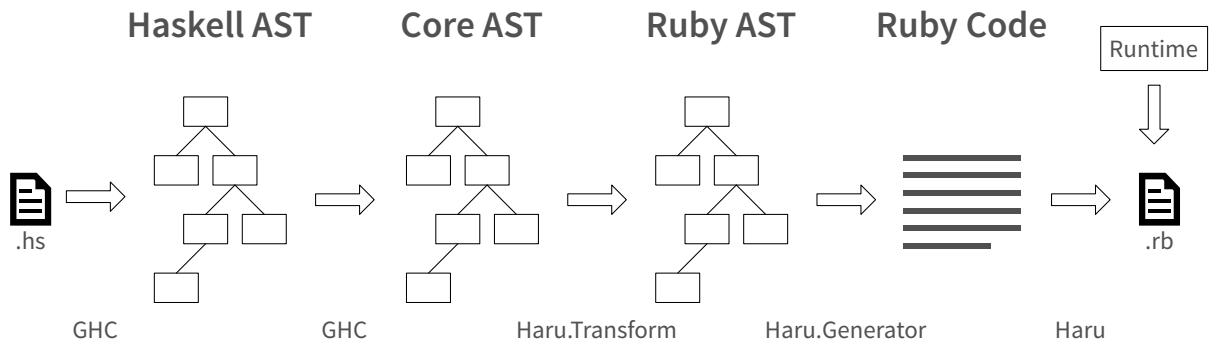


Abbildung 51: Haru Pipeline

Zunächst wird das Modul vorbereitet. Hierzu wird die Datei geladen, in die Core-Darstellung umgewandelt und die GHC Optimierungsschritte durchgeführt.

```
-- | Prepares a module for compilation
--
--   Takes the raw loaded module and generates the module details.
--   These details ("guts") contain the complete information about the
--   module, like bindings, type declarations and imports.
--
--   The different function calls are the single steps of the GHC pipeline.
--
--   Parameters:
--     - moduleSummary    The summary of the module to compile
prepareModule :: (GhcMonad m) => ModSummary -> m (CgGuts, ModDetails)
prepareModule moduleSummary = do
    env <- getSession
    pgm <- parseModule moduleSummary
        >>= typecheckModule
        >>= desugarModule
        >>= liftIO . hscSimplify env . coreModule
        >>= liftIO . tidyProgram env
    return pgm
```

Generierung des GHC-ASTs. Im Grunde werden hauptsächlich die fünf Schritte der GHC-Pipeline „parseModule“, „typecheckModule“, „desugarModule“, „hasSimplify“ und „tidyProgram“ ausgeführt.

Die eigentliche Kompilierung zu Ruby funktioniert ähnlich wie GHC auch regulären Haskell-Code kompiliert. Zunächst generiert Haru den GHC-AST aus einer Haskell-Datei mittels obiger prepareModule Funktion. Anschließend wird die Runtime geladen, die Typkonstruktoren in Ruby erstellt, aus dem Ruby-AST Quellcode erstellt und alles konkateniert in die Source-Datei geschrieben.

[51] Es ist nicht wirklich ein Ruby AST, aber ein an Ruby angepasster AST, der die Codegenerierung einfacher gestaltet.
[52] Als eine Art statischem Linken. Dies verhindert aktuell allerdings das importieren mehrerer Dateien gegenseitig, da dann die Funktionen mehrfach definiert sind. Dies lässt sich beim Einbau aber einfach dadurch lösen, dass die Runtime in ein eigenes Modul kommt, das jeweils durch Ruby eingebunden („require“) wird.

```
-- | Compiles a single module
--
--   Takes the flags, the output file name and the module summary,
--   generates ruby code and writes it to the output file.
--
--   Parameters:
--     - dynFlags         Dynamic flags
--     - outputFile       The output file name
--     - moduleSummary    The pregenerated module summary, including the core bindings
compileModule :: (GhcMonad m) => DynFlags -> String -> ModSummary -> m ()
compileModule dynFlags outputFile moduleSummary = do
    -- Generate the module details by GHC, containing the Core AST
    (cgGuts, modDetails) <- prepareModule moduleSummary
    -- Load the runtime code
    runtimeCode <- liftIO $ runtimeCode
    let
        -- Collect useful compiler data (data about the program being compiled)
        compilerData        = generateCompilerData rubyAST
        -- Generate code for type constructors
        typeConstructorCode = generateTypeConstructors $ transformTypeConstructors (cg_tycons cgGuts)
        -- Generate the Ruby AST from the Core AST
        rubyAST             = map transformBinding (cg_binds cgGuts)
        -- Generate Ruby Code from the Ruby AST
        programCode         = join "\n\n" $ map (generateProgram compilerData) rubyAST
        -- Concatenate all code parts to create the complete source file contents
        completeProgramCode = runtimeCode ++ "\n\n" ++ typeConstructorCode ++  "\n\n" ++ programCode
        in do
            -- Write the contents to the ruby file
            liftIO $ writeOutput outputFile completeProgramCode
            Log.log $ "  Compilation successfull."
```

Die Hauptfunktion des Compilers: compileModule kompiliert ein Haskell-Modul zu Ruby-Code.

Während einige Core-Bestandteil einfach zu transformieren sind wie zum Beispiel Literale, sind für andere Bestandteile teils komplizierte Anpassungen notwendig. Einige dieser Anpassungen werden im Folgenden erläutert.

6.1 Besonderheiten der Implementierung

Haru ignoriert einen Großteil der Typisierung des ursprünglichen Haskell-Programms, außerdem werden die skalaren Datentypen explizit umgewandelt. Beim Verhalten dieser Typen gibt es ebenfalls wenig Anpassungsbedarf, da das grundsätzliche Verhalten (Operatoren, Transformationen) zu Haskell ähnlich ist beziehungsweise permissiver.

6.1.1 Runtime

Haru übersetzt eigentlich nur den Sprachkern von Haskell und nicht die Standardbibliothek. Dadurch ist aber das Testen unmöglich, da beispielsweise Funktionen für die Ausgabe fehlen. Deswegen wurde eine kleine Runtime entwickelt, die die für das Testen wichtigsten Funktionen bereitstellt[53]:

- *Put*: Gibt einen Wert aus.
- *_Format*: Funktion, die eine Variable in ein an Haskell angepasstes Ausgabeformat anpasst[54].
- *Cons*: Implementiert den cons-Operator „:“
- *Uncons*: Wird benötigt, um Typen zu dekonstruieren[55]. Interne Funktion[56].

[53] Alle Runtime-Funktionen beginnen im Namen als „HaruRuntime_“
[54] Nativ wird [1, 2, 3] in Ruby zu „[1, 2, 3]“, in Haskell jedoch zu „[1,2,3]“.
[55] Siehe 6.1.3

- *Is_String_Or_Array*: Typüberprüfung, da nicht nur auf Liste überprüft werden kann. Interne Funktion.
- *Map*: Map-Implementierung.
- *Join*: Fasst Elemente eines Arrays mit einem Trennzeichen zu einem String zusammen.
- *Enum_From_To*: Helfer-Funktion, die eine arithmetische Folge (Enum) generiert.
- *Chain*: Funktion, die den Kompositionsoperator „.“ implementiert.

Zunächst war geplant, Currying auch in der Runtime zu implementieren, dies hat Ruby allerdings nicht zugelassen (siehe 6.1.5).

6.1.2 Strings, Listen und Cons

Haskell implementiert Strings als eine Liste von Zeichen. Ruby hat dies in dieser Form nicht, dort sind Strings vom Type „String“ und nicht vom Typ „Array“ (die einfachsten Ruby-Listen). Das hat Auswirkungen auf die Standardbibliothek: zunächst müssen Listen-Funktionen so angepasst werden, dass sie auch für Strings funktioniert. Das Problem könnte umgangen werden, in dem die Runtime und der Listen-Typ so angepasst wird, dass er intern nur auf Zeichen-Arrays arbeitet und sich nur bei wenigen Funktionen wie ein String verhält (beispielsweise zu „puts“). Da Ruby jedoch Strings anbietet, versucht Haru diese zu verwenden.

Dies wird so umgesetzt, dass in den Runtime-Funktionen zu Beginn eine Prüfung statt findet, ob der übergebene Wert einer Listenfunktion in Wirklichkeit ein String ist. In diesem Fall wird der String vor jeglicher Änderung in ein Array aus Strings mit einem Zeichen umgewandelt und nach den Operationen wieder zusammengefügt. Außerdem erkennt die Runtime, dass ein Zeichen in eine leere Liste mittels cons eingefügt wird und erstellt automatisch einen String.

6.1.3 Listen-Dekonstruktion

Listen-Dekonstruktion wird in der Runtime durchgeführt.

```
listFunction (x:xs) = ...
```

Einfaches Beispiel für eine Listendekonstruktion.

Für die Listendekonstruktion wurde eine Runtime-Funktion namens *uncons* erstellt. Diese erhält als Argument die ursprüngliche Liste und die Anzahl der Variablen, die extrahiert werden sollen. Hierbei wird den Variablen immer der Wert aus der Liste entsprechend ihres 0-basierten Indexes zugewiesen, das letzte Argument erhält die restliche Liste.

```
list = [1, 2, 3, 4, 5]
v1, v2, v3 = HaruRuntime_uncons.call(list, 3)

# v1 = 1 , 0. Element
# v2 = 2 , 1. Element
# v3 = [3, 4, 5] , letztes Element -> erhält Rest der Liste
```

Beispiel für die Verwendung von uncons.

[56] Er wird direkt von Haru erzeugt und es ist nicht vorgesehen, dass Benutzer die Methode selbst aufrufen.

Da das Typ-System wie es aktuell implementiert ist, komplett auf Listen aufbaut, kann uncons auch mit Typdekonstruktion umgehen.

6.1.4 Funktionsaufrufe

Funktionsaufrufe in Haskell sind, auch durch Currying, sehr transparent. Da es keine Variablen im eigentlichen Sinne gibt, ist die Aufrufschnittstelle von Funktionen sehr einheitlich. In Haskell gibt es kein Zustandskonzept[57], das bedeutet dass Variablen ihren Wert nicht ändern können (sie sind also Konstanten). Konstante Werte können durch konstante Funktionen ohne Argumente abgebildet werden. Hierdurch sind in Haskell alle verwendeten Identifier konzeptuell Funktionsaufrufe, die ihrerseits entweder Funktionen zurückgeben oder zu Werten ausgewertet werden.

Diese Transparenz existiert in dieser Form in Ruby nicht. Hier gibt es die Trennung zwischen Variablen und Funktionen, inklusive einiger bedeutender Unterschiede. Eine Gemeinsamkeit existiert allerdings, und zwar das optionale Auslassen von Klammern. So kann, zumindest bei tatsächlichen Funktionen, die Unklarheit ignoriert werden, ob ein Identifier nun einen Ruby-Variablenzugriff oder einen Ruby-Funktionsaufruf ohne Parameter darstellt (vergleiche Syntaktische und semantische Besonderheiten im Kapitel 4.4.4), da das Ruby selbst auflöst.

Ruby unterstützt anonyme Funktionen, die für die Umsetzung des Currying eingesetzt werden. Allerdings gibt es keine einheitliche Aufrufsyntax für reguläre und anonyme Funktionen:

```
def regular (x)
    x + 5
end

anonym = lambda { |x|
    x + 5
}

regular(5)
anonym.call(5)
```

Aufrufsyntax von regulären und anonymen Methoden.

Beide Aufrufe erlauben nur ihre eigene Syntax, es ist kein gemeinsames Interface vorhanden. Eine mögliche Lösung ist es, jeden Aufruf auf Identifier in einer Funktion zu kapseln, die automatisch erkennt, ob es ein Wert, eine reguläre Funktion oder ein Lambda ist und die entsprechende Aktion durchführt. Allerdings erlaubt dies Ruby nicht direkt, da man von einer regulären Funktion kein Funktionsobjekt erhält[58]. In Javascript beispielsweise kann man auf eine definierte per Namen zugreifen, dies geht in Ruby nicht. Der Grund liegt in den optionalen Klammern: Ruby kann nicht entscheiden, ob man auf das Funktionsobjekt zugreifen will, oder ob man einen verketten Aufruf durchführen will. Daher sind direkte Zugriffe auf den Namen einer Funktion in Ruby immer direkt Auswertungen.

[57] Außer in Monaden, aber die werden hier nicht betrachtet.
[58] Über viel Metaprogramming und Umwege könnte unter Umständen eine mögliche Lösung gefunden werden.

```
function fun (x, y) {
    return x + y;
}

console.log(fun.length);
```

In JavaScript kann auf das Funktions-Objekt per Namen zugegriffen werden. Das Attribute length einer Funktion gibt die Anzahl der erwarteten Argumente zurück (und eignet sich daher für die Implementierung des eval/apply-Ansatzes für Currying).

Ein weiteres Problem, das durch die Generierung von Funktionsdefinitionen entsteht ist die Sichtbarkeit. Da anonyme Funktionen regulären Variablen zugewiesen werden, sind diese nicht innerhalb von anderen Methoden sichtbar. Ein Lösungsansatz ist, alle Top-Level-Lambda-Variablen global zu machen – dies erfordert aber im Anschluss bei jedem Identifier-Zugriff eine globale Analyse, ob ein Identifier ein Aufruf zu einer globalen Funktion ist (globale Variablen beginnen in Ruby mit eine „$“-Zeichen).

```
anonym = lambda { „anonym“ }
def main () end

def additional_function ()
    anonym.call() # <- nicht erlaubt, da nicht im Sichtbarkeitsbereich
    main          # <- erlaubt, da Methoden immer global sichtbar sind
end

additional_anonym = lambda {
    anonym.call() # <- erlaubt, da additional_anonym im äußeren Scope ist
    main          # <- erlaubt, da Methoden immer global sichtbar sind
}
```

Sichtbarkeit von Funktionsnamen.

Eine mögliche Lösung wär es, komplett auf die Definition von regulären Funktionen zu verzichten und nur anonyme Funktionen zu verwenden. Für Funktionen ohne Argumente kann man direkt den Rumpf der Funktion der Variablen zuweisen, da diese konstant sind (per Definition in Haskell). Der Verlust der Transparenz zwischen regulärem Funktionsaufruf ohne Argument (ein Lambda ohne Argument erfordert trotzdem ein „.call“) und Variablenzugriff verschwindet dadurch, dass es keine Funktionen ohne Argumente mehr gibt (diese werden direkt ausgewertet[59]).

6.1.5 Currying

Es gibt im Grunde drei mögliche Varianten, Currying umzusetzen. Zwei geschehen in der Runtime zur Laufzeit, die dritte Methode umgeht einen Großteil der Logik und vertraut auf die Korrektheit der Typprüfung durch Haskell.

[59] Dies ist problemlos möglich, da diese nicht von unbekannten globalen Zuständen abhängen dürfen, höchstens von anderen Top-Level-Bindungen. Diese werden von Haskell allerdings bereits so vorsortiert, dass mögliche Abhängigkeiten und Reihenfolgen eingehalten werden.

Die Implementierung von Currying ist schwer, da für mögliche Optimierungen eine globale Datenflussanalyse durchgeführt werden muss. Das verdeutlicht das folgende Beispiel[60]:

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f [] []     = []
zipWith f (x:xs) (y:ys) = (f x y) : (zipWith f xy ys)
```

Definition von map.

In diesem Beispiel ist f eine unbekannte Funktion. Der Compiler kann nicht einfach f mit zwei Argumenten aufrufen, da es eine Funktion sein könnte, die erst ein Argument konsumiert, eine Weile rechnet und anschließend das zweite Argument konsumiert. Ebenso ist es möglich, dass f mehr als zwei Argumente fordert und zipWith also eine Liste von Funktionen generiert.

Push/Enter

Hier handhabt die Funktion selbst die Verwaltung ihrer Argumente. Jede Funktion besitzt einen Stack, auf den ihre Argumente beim Aufruf gepusht werden. Die Funktion analysiert in ihrem Einstiegscode den Stack , lädt ihre Argumente selbst und entfernt sie vom Stack. Die Funktion muss anschließend selbst dafür sorgen, dass wenn sie weniger Argumente auf dem Stack hat als sie benötigt, sie eine partielle Auswertung ausführt und eine neue Funktion zurückgibt. Wenn sie zu viele Argument erhält, lädt sie trotzdem nur die Argumente vom Stack, die sie benötigt (die restlichen Argumente werden von der Funktion konsumiert, die sie zurückgibt).

Dieser Ansatz funktioniert nicht direkt in Ruby, da man den Einstiegscode von Funktionen nicht in dieser Weise anpassen kann.

Eval/Apply

In diesem Ansatz analysiert der Aufrufende zunächst die Funktion und ruft sie dann mit der korrekten Anzahl an Argumenten auf. Dies erfordert eine Laufzeitanalyse des Closures: im Beispiel mit zipWith folgendes:

- Nimmt f nur ein Argument, werte f mit x aus und anschließend die Ergebnisfunktion mit y.
- Nimmt f zwei Argumente, kann es regulär ausgeführt werden.
- Nimmt f mehr als zwei Argumente wird ein neues Closure gebaut, in dem x und y bereits gebunden sind.

Dieser Ansatz funktioniert in Ruby. Der Kern dieses Ansatzes ist die Abfrage der Stelligkeit eines Closures, die man in Ruby über die Methode arity eines Lambdas erhält. Sie erfordert allerdings einige zusätzliche Logik bei jedem Funktionsaufruf.

Einstellige Closures

Dies ist die simpelste der drei Methoden[61]. Jeder Funktionsaufruf wird in geschachtelte Lambda-Aufrufe mit jeweils einem Argument umgebaut. Funktionsapplikationen sind dann immer Aufrufe mit einem Argument.

[60] Aus [4]

```
# Definition regulär
regular = lambda { |x, y, z|
    x + y + z
}

# Aufruf regulär
regular.call(1, 2, 3)


# Definition einstellig
modified = lambda { |x|
    lambda { |y|
        lambda { |z|
            x + y + z
        }
    }
}

# Aufruf einstellig
modified.call(1).call(2).call(3)
```

Umwandlung mehrstelliger Funktionen in einstellige Funktionen.

Diese Struktur ist sehr nah an das tatsächliche Currying von Haskell angelegt, die interne Darstellung des GHC sieht unmodifiziert ähnlich aus. Ein möglicher Fehlerfall wäre, dass eine Funktion ein Lambda zurückgibt, das kein Argument nimmt und damit nicht aufgerufen wird. Oder andersherum: dass ein Identifier aufgerufen wird, der kein Wert ist. Durch die Konstruktion der Transformationen in Haru wird dieses Problem allerdings umgangen – es gibt keine Funktionsdefinitionen ohne Argumente (diese sind schlicht Zuweisungen) also sind Applikationen ohne Argumente schlicht Variablenzugriffe.

Die strikte Typprüfung von Haskell im Vorfeld sorgt außerdem dafür, dass nur Aufrufe auf passende Identifier ausgeführt werden und auch die Anzahl der Argumente stimmt. Die Umstrukturierung in die neue Struktur kann direkt im Code-Generator für Funktionsdefinitionen und –Aufrufe geschehen und stellt dadurch nur einen kleinen Eingriff in das Programm dar.

6.1.6 Tupel

Tupel haben in Ruby kein direktes Gegenstück und werden mit Arrays (genauer gesagt, einer Klasse, die von Array erbt – um Typprüfung auf Array zu ermöglichen) umgesetzt. Die Struktur der Daten ist von der Darstellung unabhängig, die Typprüfung bereits durch Haskell erledigt und die Dekonstruktion kann dadurch einheitlich durch die uncons[62]-Funktion erledigt werden.

6.1.7 Data-Konstruktoren

Haskell erlaubt die Definition eigener Datentypen. Während eine Umbenennung eines skalaren Typen für die Codegenerierung wenig interessant ist (die Typen tauchen dort fast nirgends mehr auf und

[61] Aufgrund der vielen (unnötigen) anonymen Funktionen und Indirektionen ist sie jedoch in einem Produktionscompiler nicht zu empfehlen. Es muss davon ausgegangen werden, dass beide Faktoren einen erheblichen Einfluss auf die Laufzeit und das Speicherverhalten des kompilierten Programms haben.
[62] Siehe 6.1.1

Umbenennungen sind bereits aufgelöst), sind die GADTs[63] deutlich interessanter. Die GADTs sind zusammengesetzte Typen, die durch Konstruktoren erzeugt werden. Diese Konstruktoren können entweder keine Argumente nehmen, dann ist der Typ eine Art Enum, oder mit Argumenten versehen werden, dann ist es eine Art typisiertes Tupel. Auch geschachtelte Typen sind möglich.

```
data Color = Red | Green | Blue

-- Parameter:
--   String  Titel
--   Int     Produktionsjahr
data Film = Film String Int

data PosterColor = PosterColor Film Color
```

GADT Definition

Durch die logische Transformation auf Listen ist die Darstellung der GADTs einfach: der eigentliche Typ wird als von Array abgeleitete Klasse umgesetzt, die Enum-Werte werden ebenfalls als Klassen umgesetzt. Es wurde dieser Weg gewählt, da dann bei der Dekonstruktion der Typen in einem Case-Statement die Typprüfung einfach übernommen werden kann (der Typ einer Variablen in Ruby ist ihre Klasse).

Durch die Darstellung der Konstruktoren mit Argumenten als Array kann der Typ auch standardisiert über die uncons-Runtimefunktion dekonstruiert werden.

6.1.8 Interne Details des GHC

Currying erzeugt allerdings nicht nur bei der Codegenerierung Probleme. Vielmehr ist es auch durch die interne Verwendung innerhalb des GHC ein Punkt, der die Compilerimplementierung erschwert.

```
fun :: Int -> Int -> Int
fun x y = x + y

fun 2 3
```

Während der Auswertung des Core-ASTs sieht man hier folgende Konstruktion:

```
fun (Applikation)
   |- Stelligkeit: 2
   |- Argumente:
      |- GHC.Num.+ (Applikation)
         |- Stelligkeit: 1
         |- Argumente:
            |- GHC.Num.$fNumInt (Applikation)
               |- Stelligkeit: 0
               |- Argumente: -
            |- 2 (Literal)
            |- 3 (Literal)
```

[63] Siehe 4.4.1

Core-AST-Darstellung für eine Addition zweier Integer.

In diesem Beispiel wirkt es intuitiv verwunderlich, dass die Stelligkeit mit den Argumenten nicht übereinstimmt. Hier tritt (auch im Compiler selbst, der in Haskell programmiert ist) das Currying zu Tage. GHC.Num.+ ist in Wirklichkeit ein Dictionary und GHC.Num.$fNumInt der Schlüssel. In diesem Dictionary sind die tatsächlichen Implementierungen der Operatoren für die unterschiedlichen Datentypen gespeichert –diese haben dann Stelligkeit 2 und konsumieren die zwei folgenden Argumente.

Diese Implementierungsdetails benötigt Ruby allerdings nicht, da es selbst eine Hochsprache ist, außerdem kann nicht ohne größere Umwege auf die Funktion im Dictionary zugegriffen werden. Deswegen werden diese Aufrufe im Generator ignoriert beziehungsweise bei der Transformation gefiltert. Dadurch ergibt sich allerdings das Problem, dass die Stelligkeit der Funktionsdefinitionen manuell angepasst werden muss. So muss dem Compiler für die unterschiedlichen Operatoren explizit mitgeteilt werden, dass zum Beispiel die Stelligkeit von GHC.Num.+ in Wirklichkeit 2 ist und die andere (Schlüssel-)Funktion ignoriert werden kann. Dies hat allerdings zur Folge, dass die Argumentstruktur nicht mehr stimmt - so ist die Schachtelung der Funktionen für die Generierung ungünstig:

```
-- aktuell:
fun (+, 2, 3)

-- korrekt:
fun (+(2, 3))
```

Struktur, die nach dem Entfernen der Argumente entstehen kann.

Der Transformator versucht diese Struktur nun anzupassen. Dies ist allerdings keine korrekte Lösung, da man Currying beachten muss. So kann es bei dem Beispiel mit der Addition durchaus sein, dass fun zwei Argumente nimmt und +(2) eine partielle Applikation der Addition ist. Für dieses einfache Beispiel lässt sich das identifizieren, für mehrere (+)-Operatoren, die alle gecurried werden können, ist es im Allgemeinen nicht mehr entscheidbar.

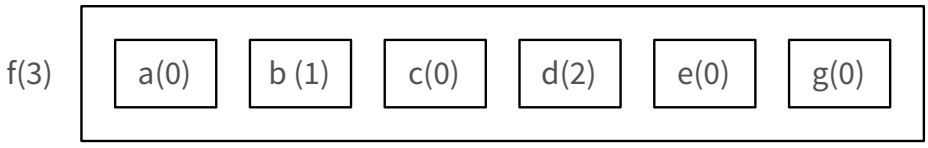


Abbildung 52: Fehlende Argumentschachtelung. Die Zahlen in Klammern geben die Stelligkeit des Bezeichners an.

Dieses Problem kann umgegangen werden, indem entweder eine neue, globale Typanalyse durchgeführt wird oder das Currying auch in den internen Compiler-Typen erkannt und teilweise angewandt wird. Beide Varianten erhöhen den Aufwand der Compiler-Implementierung immens.

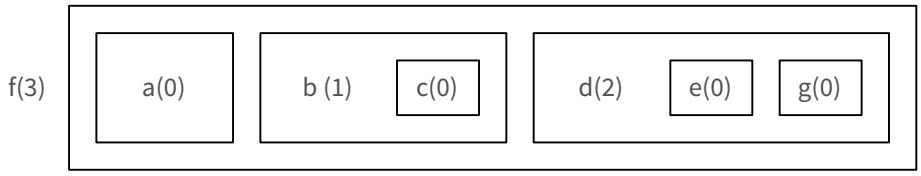


Abbildung 53: Korrekte Schachtelung der Argumente.

Da Haru nur für kleine Beispielprogramme als Testcompiler eingesetzt wird, ist hier eine pragmatische Heuristik eingebaut. So führt Haru Runtime-Funktionen und Operatoren grundsätzlich in der Transformation zusammen (diese unterstützen also kein Currying[64]). Ein Sonderfall ist jedoch implementiert: wenn die Summe der Stelligkeiten mit der Anzahl aller Argumente übereinstimmt, werden alle Aufrufe entsprechend gruppiert.

6.2 Nicht abgebildete Aspekte

Durch die in 6.1.8 angesprochenen Probleme hinsichtlich der Implementierungsdetails des GHCs ist *Currying* nur teilweise umgesetzt. Runtime-Funktionen und Operatoren binden immer kommende Argumente an sich, soweit es welche gibt, Sections (explizites Currying auf Operatoren) wird allerdings unterstützt.

Lazy Evaluation wurde nicht direkt umgesetzt, Ruby bietet sie an einigen Stellen allerdings nativ an. So gibt es zum Beispiel die „kurzschließenden“ Bedingungen, die nur soweit auswerten, wie sich das Ergebnis der Berechnung noch ändern kann. Im folgenden Beispiel wird also heavy_computation nie ausgewertet, da das Ergebnis den Wert des booleschen Ausdrucks nicht mehr ändern kann:

```
If true || heavy_computation
  # ...
end

if false && heavy_computation
  # ...
end
```

Beispiel für „kurzschließende“ Bedingungen in Ruby

Lazy Evaluation würde bedeuten, dass jeder Variablen-Zugriff und jede Zuweisung zunächst in eine anonyme Funktion verpackt wird, die dann zu gegebener Stelle aufgerufen wird. Dieser Aspekt wurde nicht umgesetzt, da er vielfältige negative Auswirkungen auf den Compiler gehabt hätte: die Komplexität des Erkennens, wann ausgewertet muss; der zu erwartende Performanceeinbruch und die Tatsache, dass das resultierende Ruby-Programm erheblich an Lesbarkeit verloren hätte – was eine spätere Validierung des Programms erschwert hätte.

Direkt damit verbunden ist das Arbeiten mit *unendlichen Listen*. In Haskell kann man sehr einfach eine unendliche Liste erstellen und auf dieser operieren:

[64] Sections werden trotzdem unterstützt, da diese bereits im Vorfeld zu Lambda-Funktionen umgebaut wurden.

```
infinite_list = [1, 2..]
```

Definition einer unendlichen Liste in Haskell.

Für die Umsetzung von unendlichen Listen muss zwangsläufige Lazy Evaluation implementiert sein, da diese offensichtlich nicht ausgewertet und in den Speicher geladen werden kann. In Ruby wäre dies mit Generatoren[65] möglich, sowie seit Version 2.0 ist ein Teil von Lazy Evaluation in Form von Lazy Enumeration implementiert. Diese erlaubt das Traversieren von und Arbeiten mit unendlichen Listen:

```
infinite_list = 1..Float::INFINITY
p infinite_list.lazy.collect { |x| x**2 }.first(10)
# => [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Lazy Enumeration in Ruby

Modulunterstützung ist ebenfalls ein Feature, das nicht unterstützt ist. Dies würde einen relativ kleinen Eingriff bedeuten: die Module werden sowieso separat übersetzt, die einzigen Erweiterungen würden die Imports („require“ in Ruby) und mögliche Aliase darstellen. Einzig die Einschränkungen der Imports (nicht alle Funktionen eines Moduls werden im- oder exportiert) müssten separat auf Umsetzungsmöglichkeiten in Ruby analysiert werden.

Zu guter Letzt wurden Teile des Typsystems nicht direkt übersetzt, so finden sich Klassen und Instanzen nicht in der Übersetzung. Der Grund hierfür ist der begrenzte Umfang des Compilers: für die sinnvolle Verwendung und ein umfangreiches Testen müssten ein Großteil der Standardbibliothek übersetzt werden, da die Funktionen sehr weit darauf aufbauen. Da der Fokus der Arbeit auf der eigentlichen Transformation und nicht auf einer großen Laufzeitumgebung liegt, wurde dieser Bereich deswegen ausgelassen.

7 Test und Validierung

Auch wenn die Modelltransformation als korrekt angenommen wird, muss die tatsächliche Compiler-Ausgabe von Haru validiert werden. Die Tests versuchen eine annähernde semantische Validierung zu geben, auch wenn das verwendete Testverfahren dies nicht garantiert.

Es wurde nur eine qualitative Validierung der Programme ausgeführt, es wurden keine Tests bezüglich der Laufzeit des Haru-Compilers gemacht. Dies liegt daran, dass die Menge von Haskell-Aspekten, die Haru übersetzt, relativ klein ist. Insbesondere ist die gesamte Standardbibliothek (Prelude) ausgenommen. Dies schränkt die Möglichkeit an den Testprogrammen drastisch ein, wodurch es keine größeren Tests gibt. Da die Tests aber alle relativ kurz sind (maximal 100 Zeichen), kann keine Aussage über die Kompilierlaufzeit getroffen werden.

Ein weiteres Problem ist, dass GHC teilweise zu umfassend optimiert[66]. Das folgende Beispielprogramm soll Funktionsaufrufe testen.

[65] In Ruby heißen diese Enumeratoren.
[66] Siehe auch 7.3

```
main = putStrLn (show (fun 5 6))
fun x y = x + y
```

Code-Beispiel für Inlining

Allerdings ist die Methode klein genug, dass GHC diese „inlined“ und dadurch den Funktionsaufruf (inklusive der damit verbundenen Funktionsaufruf-Kosten) umgeht. Einerseits wünschenswert, da das entstandene Programm dadurch performanter wird, allerdings für den Test hinderlich. Der tatsächlich kompilierte Code sieht dann so aus:

```
main = putStrLn (show 5 + 6)
```

7.1 Testmethodik

Da die Ausführungssemantik einzelner Teile nicht direkt getestet werden kann, werden kleine Beispielprogramme mittels Unit-Tests getestet. Hierzu werden Test-Programme in Haskell geschrieben, die ausgeführt werden. Anschließend wird mittels Haru das Programm von Haskell in Ruby kompiliert und das erzeugte Ruby-Programm ausgeführt. Zuletzt werden mittels einer Diff[67]-Implementierung die Ergebnisse beider Ausführungen verglichen. Wenn das Diff leer ist (es also keine Unterschiede zwischen den Ausgaben gibt), werden die Programme als semantisch korrekt angesehen.

Hierbei muss aber darauf geachtet werden, dass mindestens ein Programm überhaupt eine Ausgabe hat. Wenn beide Programme keine Ausgabe haben (aufgrund eines Fehlers oder weil sie regulär keine Ausgabe erzeugen), kann nicht von der Gleichheit der Programme ausgegangen werden.

Auch muss beachtet werden, dass durch die vereinfachten Tests, die nur die Ausgabe vergleichen, keine Aussagen über das Innere der Programme getroffen werden kann. Ein Programm, dass die 15. Fibonacci-Zahl ausrechnet und ein Programm, das nur die Zahl 610 ausgibt, werden als gleich angesehen. Die fortführende Validierung muss in diesem Fall manuell geschehen, soweit sie nicht bereits durch die Validation der Modelltransformationen geschehen ist.

7.2 Automatisierte Testsuite

Es wurde eine automatisierte Testsuite mit knapp 20 Tests entworfen, die sich an den abzubildenden Haskell-Konzepten orientiert und diese weitgehend abdecken soll. Diese Testsuite wurde in der Entwicklung einerseits zur testgetriebenen Entwicklung verwendet, andererseits konnten damit automatisch Regressionen erkannt und behoben werden.

Die Testsuite testet automatisch alle Programme im Beispiel-Ordner und gibt eine Übersicht über den Status der Tests aus:

[67] diff ist ein Unix-Programm, das die Unterschiede zweier Textdateien zeilenweise gegenüberstellt. Hier wird mit „Diff“ allerdings das Verfahren an sich bezeichnet und nicht das konkrete Programm.

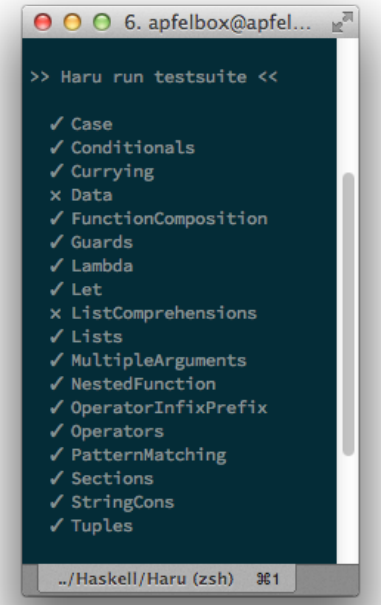


Abbildung 54: Ausgabe der Testsuite, zwei Tests schlagen in diesem Fall fehl

Neue Tests hinzuzufügen wird damit sehr komfortabel, es muss nur ein Haskell-Programm im Beispiel-Ordner hinzugefügt werden.

7.3 Testergebnisse

Die Testergebnisse bestätigen die korrekte Übersetzung, sowohl der Metamodelle, als auch des Programms. Wobei man die Testergebnisse mit Vorsicht genießen muss, da die Optimierungen im GHC teilweise so aggressiv sind, dass die Tests unter Umständen unbrauchbar werden.

Davon betroffen war beispielsweise die erste Iteration des Data-Tests:

```
-- Defines the color data type
data Color = Red | Green | Blue

-- Prints the color name
colorName :: Color -> String
colorName Red   = "Red"
colorName Green = "Green"
colorName Blue  = "Blue"


main = do
    putStrLn $ colorName Red
    putStrLn $ colorName Green
    putStrLn $ colorName Blue
```

Erster Test für Unterstützung von algebraischen Datentypen.

Dieser Test wird von Haskell optimiert und produziert folgenden Ruby-Code:

```
HaruExamples_Data_main = lambda {
    HaruRuntime_put.call('Red')
    HaruRuntime_put.call('Green')
    HaruRuntime_put.call('Blue')
}
```

Generierter Ruby-Code

GHC hat die gesamte Typdefinition optimiert und entfernt, zusammen mit der colorName-Funktion und die Ergebnisse statisch ermittelt und propagiert.

Da nur kleine Teile der Laufzeitumgebung übertragen wurden, können die Testfälle nur aus einer kleinen Auswahl von Methoden erstellet werden. Dies birgt die Gefahr, dass weitere Testfälle so weit optimiert wurden, dass die Testfälle etwas anderes testen. Somit muss die eigentlich automatisierte Testsuite nach den Tests nochmal manuell nach solchen Punkten überprüft werden, denn die Testsuite meldet den Test als erfolgreich.

8 Zusammenfassung

Zusammenfassend muss betont werden, dass die Sprachtransformation mittels Metamodellen einige erhebliche Vorteile bringt. Wenn die Sprachen formal spezifiziert sind, lassen sich daraus unkompliziert Metamodelle ableiten, die anschließend transformiert werden können. Falls keine formale Spezifikation existiert, ist es möglich, aus der Grammatik der Sprache große Teile des Metamodells direkt zu generieren.

Der große Vorteil im Hinblick auf Kompilierung durch Unterstützung der Metamodelle ist, dass man direkt eine grobe Überprüfung und Validierung erhält. Diese Verhindern konzeptuelle Fehler im generierten Quellcode.

Einer der größten und kompliziertesten Punkte[69] im gesamten Verfahren ist allerdings die korrekte Code-Generierung. Wenn die Zielsprache, wie in diesem Fall Ruby, selbst einige Beschränkungen und Bedingungen an die Programme stellt (wie Sichtbarkeiten, Gültigkeitsbereiche oder unterschiedliche Aufrufsyntax für reguläre und anonyme Funktionen) wird die Code-Generierung sehr aufwändig und komplex. Es müssen viele Sonderfälle beachtet werden, außerdem muss wesentlich mehr Hintergrundwissen als bei einer einfacheren Zielsprache vorhanden sein. Auch Layout (semantisches Whitespace[70]) kann ein erschwerender Faktor sein, wenn der Compiler Code erzeugen will, der nach der Kompilierung manuell weiterverwendet wird.

[69] Und am meisten unterschätzt.
[70] Haskell, Ruby (in Teilen) und beispielsweise Python haben semantisches Whitespace. Dort können Schlüsselworte oder Klammern ausgelassen werden, wenn die Aussage durch korrekte Einrückung oder Zeilenumbrüche erhalten bleibt.

8.1 Ausblick

Für die Erweiterung des Projekts zu einem vollwertigen Compiler müssen zunächst die Metamodelle zu den restlichen Sprachkonzepten von Core erstellt werden. Nach der Identifikation der nun zusätzlich benötigten Teile der Ruby-Spezifikation können die restlichen Core- (und damit Haskell-) Konzepte ebenfalls transformiert werden – auch wenn diese hauptsächlich das Typsystem betreffen.

Im Hinblick auf die Implementierung von Haru lassen sich ebenfalls einige Punkte optimieren. So ist das erzeugte Programm selbst nicht typsicher, die Sicherheit kommt nur durch die vorherige Validierung des GHCs. Dies bedeutet, dass die erstellten Programme manuell nicht angepasst werden sollten.

Weiterhin werden alle Identifier innerhalb des Compilers von ihrer eigentlichen Bedeutung gelöst – dies bedeutet, dass die Information, auf was sich ein Identifier bezieht, verloren ist. Dieser Designschritte ist bewusst so gewählt, da er die Komplexität des Compilers verringert und für die Codegenerierung zum jetzigen Zeitpunkt nicht zwingend notwendig ist. Allerdings werden dadurch Analysen auf den Identifiern erschwert oder verhindert, die zu einer besseren Codeerzeugung führen könnten.

Weiterhin könnten noch Optimierungsschritte eingebaut werden, die ähnlich zum GHC Literal propagieren und kombinieren. Vor allem die (eigentlich statische) Erstellung von Listen könnte deutlich optimiert werden – diese ist aktuell eine direkte Übersetzung des generierten Core-Codes.

```
-- in Haskell:
["a", "b", "c", "d", "e", "f"]

# generiert in Ruby (gekürzt)
cons(
    cons('a', []),
    cons(
        cons('b', []),
        cons(
            cons('c', []),
            cons(
                cons('d', []),
                cons(
                    cons('e', []),
                    cons(
                        cons('f', []),
                        []
                    )
                )
            )
        )
    )
)
```

Generierte Listenerzeugung

Auch die Erstellung von Strings (als Spezialfall der Listen) kann optimiert werden, die, wie hier zu sehen ist, als cons-Operation des Zeichens und einer leeren Liste erstellt werden. Diese

Funktionsaufrufe können statisch gefunden und in entsprechende Literale beziehungsweise Konstruktoraufrufe umgewandelt werde.

Und zuletzt kann der Compiler daraufhin optimiert werden, „typischeres“ Ruby zu erzeugen. So verwendet der Compiler nur kleine Teile des Sprachumfangs von Ruby. Dadurch entstehen Konstruktionen, die für den Compiler zwar einfach zu generieren sind, allerdings manuell von einem Entwickler so nicht geschrieben werden würden – unabhängig von Gründe, beispielsweise Lesbarkeit oder Ausführungsgeschwindigkeit. Dies erhöht zwar enorm die Komplexität des Generators und des Transformators, es erscheint aber verschwenderisch, Ruby „nur“ als ein etwas erweitertes C oder Assembler zu verwenden und nur die reinen Grundfunktionen bei der Codegenerierung zu verwenden.

9 Literaturverzeichnis

[1] Haskell: Language and library specification
http://www.haskell.org/haskellwiki/Language_and_library_specification
Simon Marlow (editor)
(abgerufen am 3.9.2013)

[2] The Architecture of Open Source Applications: The Glasgow Haskell Compiler
http://aosabook.org/en/ghc.html
Simon Marlow, Simon Peyton-Jones
(abgerufen am 3.9.2013)

[3] Modellgetriebene Softwareentwicklung
2. Auflage, dpunkt.verlag, 2007
Thomas Stahl, Markus Völter, Sven Efftinge, Arno Haase

[4] Making a Fast Curry: Push/Enter vs. Eval/Apply for High-order Languages
ICFP'04, September 19-21, 2004, Snowbird, Utah, USA
Simon Marlow, Simon Peyton Jones

[5] Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-Machine
Journal of Functional Programming, Volume 2 Number 2, Cambridge University Press, 1992
Simon L Peyton Jones
(abgerufen am 3.9.2013)

[6] Metamodel and UML Profile for Functional Programming Languages
Warsaw University of Technology, Institute of Control & Computation Engineering
Marcin Szlenk

[7] ghc-7.6.3: The GHC API
http://www.haskell.org/ghc/docs/7.6.3/html/libraries/ghc-7.6.3/
(abgerufen am 3.9.2013)

[8] The GHC Commentary
http://ghc.haskell.org/trac/ghc/wiki/Commentary
(abgerufen am 3.9.2013)

[9] Grammatik-Definition von Ruby in Version 2.0.0p247
https://github.com/ruby/ruby/blob/v2_0_0_247/parse.y
(abgerufen am 3.9.2013)

[10] Programming Language Concepts
Springer, 2012
Peter Sestoft

[11] Allgemeine Modelltheorie
Wien 1973, ISBN 3-211-81106-0
Herbert Stachowiak

[12] System F with Type Equality Coercions
The Third ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI'07),

Nice, France, ACM Press, 2007
Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, Kevin Donnelly

[13] Programming Languages as Thought Models
Structured Programming 11: 105-115, Springer-Verlag New York Inc., 1990
Peter Rechenberg (Institut für Informatik, Johannes Kelper University of Linz)

[14] „Die Beziehung zwischen Programmiersprachen und Denkprozessen“
Hauptseminarausarbeitung, Universität Stuttgart, Institut für Softwaretechnologie, 2012
Jannik Zschiesche

[15] Wikipedia: Deklarative Programmierung
http://de.wikipedia.org/w/index.php?title=Deklarative_Programmierung&oldid=116890670
(abgerufen am 3.9.2013)

[16] GHC Commentary: The Core type
http://ghc.haskell.org/trac/ghc/wiki/Commentary/Compiler/CoreSynType
(abgerufen am 3.9.2013)

[17] The GHC API: CoreSyn
http://www.haskell.org/ghc/docs/7.6.3/html/libraries/ghc-7.6.3/CoreSyn.html
(abgerufen am 3.9.2013)