Diplomarbeit

Programmverifikation mit Modulen und Axiomatischen Spezifikationen in √veriFun

Alexander Krauss
Juni 2005

Betreuer:

Andreas Schlosser
Dirk Stephan Schweitzer

Technische Universität Darmstadt
Fachbereich Informatik
Fachgebiet Programmiermethodik
Prof. Dr. Christoph Walther

Hiermit versichere ich, die vorliegende Diplomarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, 7. Juni 2005

Alexander Krauss

Zusammenfassung

Die vorliegende Arbeit umfasst die Konzeption und Implementierung einer Spracherweiterung für die Unterstützung von Modularität und axiomatischen Spezifikationen im Programmverifikationstool ✓`eriFun` . Modularität erlaubt die Strukturierung von Programmen und Beweisen in kleinere Einheiten, deren Interaktion durch Sichtbarkeiten eingeschränkt wird. Axiomatische Spezifikationen sind ein wichtiges Modellierungsinstrument zur Beschreibung sowohl von Programmierschnittstellen als auch von algebraischen Strukturen. Beides wurde bisher von ✓`eriFun` nicht unterstützt.

Nach einer Auseinandersetzung mit existierenden Modul- und Schnittstellenkonzepten in den funktionalen Sprachen Haskell und Standard ML sowie in den Theorembeweisern Isabelle/HOL und ACL2 werden zwei Erweiterungen für ✓`eriFun` vorgestellt.

Die erste Erweiterung erlaubt die Definition von Modulen und die Vergabe von Sichtbarkeiten für einzelne Programmelemente. Indem die Implementierungen von Funktionsprozeduren verborgen werden, lassen sich mit Hilfe von Sichtbarkeiten auch axiomatische Spezifikationen definieren. Theoreme über Spezifikationen können auf konkrete Instanzen übertragen werden, wenn zuvor bewiesen wird, dass die Instanz die Axiome der Spezifikation erfüllt.

Die Syntax der ersten Erweiterung lässt bestimmte Anwendungen von Spezifikationen noch nicht zu. Dazu gehören Vererbung (Eine Spezifikation wird als Erweiterung einer anderen Spezifikation definiert) und Mehrfachverwendung (Eine Definition bezieht sich auf zwei oder mehr verschiedene Strukturen derselben Spezifikation, z.B. zwei unterschiedliche Ordnungen).

Diese Schwächen werden dann in einer zweiten Erweiterung behoben, die es erlaubt, Spezifikationen vielfältig zu kombinieren.

Inhaltsverzeichnis

Kapitel 1

Einführung

In dieser Arbeit soll die Programmiersprache des ✓eriFun Systems um Modularität und axiomatische Spezifikationen erweitert werden. Die Erweiterung erlaubt es dem Benutzer, Programme in verschiedene Module zu unterteilen, die einen gewissen Grad von Unabhängigkeit voneinander haben. Weiterhin wird die Formulierung von axiomatischen Spezifikationen ermöglicht, über die dann Beweise geführt werden können.

Dafür werden zunächst einige Modulkonzepte in den funktionalen Sprachen Haskell und Standard ML sowie Aspekte der Theorembeweiser Isabelle/HOL und ACL2 untersucht, die dann als Vorlage für einen eigenen Entwurf dienen.

In einer ersten Erweiterung wird die Aufteilung eines Programms in Module und die Zuordnungen der Sichtbarkeiten *public* und *private* zu einzelnen Programmelementen unterstützt. Mit Hilfe der Sichtbarkeiten kann man erreichen, dass Hilfslemmata nur lokal innerhalb eines Moduls angewendet werden.

Durch die zweite Erweiterung um axiomatische Spezifikationen lassen sich erstmals in ✓eriFun auch Beweise über allgemeine Strukturen wie Gruppen, Ordnungen, Ringe, Körper u.s.w. führen. Diese Aussagen können auf konkrete Instanzen übertragen werden, wenn zuvor bewiesen wird, dass die Instanz die Axiome der Spezifikation erfüllt. Aussagen über Ordnungen können so z.B. auf konkret implementierte Ordnungen übertragen werden.

In Verbindung mit axiomatischen Spezifikationen sind bestimmte weitergehende Konstruktionen nützlich und wünschenswert: Eine dieser Konstruktionen ist Vererbung, also die Definition von aufeinander aufbauenden Spezifikationen. Beispielsweise möchte man die Spezifikation eines Rings auf der Spezifikation einer Gruppe aufbauen. Die zweite Konstruktion besteht darin, dass eine Definition auf zwei oder mehr verschiedenen Strukturen derselben Spezifikation aufbaut („Mehrfachverwendung“). So kann man z.B. aus zwei beliebigen Ordnungen die lexikographische Ordnung bilden. Die vorgestellte Spracherweiterung erlaubt ein flexibles Kombinieren von Spezifikationen zu sogenannten Umgebungen, wodurch sowohl Vererbung als auch Mehrfachverwendung möglich sind.

Aufbau dieser Arbeit Der Rest von Kapitel 1 führt in die allgemeine Thematik von Modularität, axiomatischen Spezifikationen und das ✓eriFun-System ein. In Kapitel 2 werden bestehende Modularitäts- und Schnittstellenkonzepte in funktionalen Sprachen und Theorembeweisern untersucht. Kapitel 3 beschreibt eine Erweiterung für ✓eriFun um Module und Sichtbarkeiten. In Kapitel 4 wird eine zweite Spracherweiterung vorgestellt, die axiomatische Spezifikationen unterstützt. Kapitel 5 beinhaltet eine Fallstudie, die mit dem erweiterten System durchgeführt wurde und die neuen Funktionalitäten in der Praxis erprobt. Kapitel 6 gibt einen kurzen Ausblick.

1.1 Modularität von Software

Grundlage des modernen Modularitätsbegriffs für Software ist das Paradigma der *Separation of Concerns* (Aufteilung der Zuständigkeiten). Dieser Begriff wurde bereits 1976 von E. Dijkstra geprägt [6] und kann heute als eine Leitdoktrin der Informatik angesehen werden. *Separation of concerns* bezeichnet die Aufteilung eines Problems in überschaubare Teilprobleme, die klar voneinander abgegrenzt sind. Diese können dann unabhängig voneinander gelöst und die Lösungen kombiniert werden.

Modularität von Software ist für sich genommen noch kein Qualitätsmerkmal. Sie stellt vielmehr eine Art Sekundärtugend dar, welche sich positiv auf andere Eigenschaften auswirkt, die ihrerseits Kosten und Qualität maßgeblich bestimmen:

Überschaubarkeit Verschiedene Module können von verschiedenen Personen oder Teams entwickelt und gewartet werden, ohne dass dabei jeder Beteiligte das gesamte System in allen Details kennen muss.

Wiederverwendbarkeit Es besteht die Möglichkeit, einzelne Module in einem anderen Zusammenhang wiederzuverwenden. Für regelmäßig wiederkehrende Anforderungen können Bibliotheken erstellt werden, auf die man in anderen Projekten zurückgreifen kann. Entwicklungszeiten und -kosten können so stark verringert werden.

Testbarkeit Da die Anforderungen an jedes Modul klar definiert sind, können einzelne Module gegen ihre Anforderungen getestet werden (Komponententest). So können Fehler in der Implementierung früher gefunden werden als beim Systemtest, für den erst das gesamte System fertiggestellt sein muss.

Wartbarkeit Falls in einem späteren Stadium des Softwarelebenszyklus Änderungen notwendig sind, dann lassen sich diese in vielen Fällen lokal in einem Modul durchführen, ohne dass andere Module verändert werden müssen.

Die Erfahrung, dass sich diese Eigenschaften bei umfangreichen Systemen nur durch eine geeignete Modularisierung erreichen lassen, hat dafür gesorgt, dass Modularität aus moderner Softwareentwicklung nicht mehr wegzudenken ist.

1.1.1 Entwurfsproblematik

Die richtige Dekomposition eines Programms in Module ist im Allgemeinen eine sehr schwierige Aufgabe, da sie nicht nur technisches Wissen und Erfahrung in der Softwareentwicklung, sondern auch Kenntnis der jeweiligen Anwendungsdomäne erfordert. Auch sollten neben den bereits bekannten auch mögliche zukünftige Anforderungen beachtet werden, um die Erweiterbarkeit und Anpassbarkeit eines Systems sicherzustellen.

Ein falscher oder der Problemstellung unangemessener Entwurf kann hingegen schwerwiegende Folgen haben, die sich später nur mit sehr hohem Aufwand korrigieren lassen: Diese Art von strukturellen Änderungen sind in der Regel globaler Natur, da sie die Interaktionen zwischen den einzelnen Modulen betreffen.

Für den Erfolg von Softwareprojekten sind also zuverlässige Vorgehensweisen zur Dekomposition von Software notwendig. Mit dieser Thematik beschäftigt sich das Gebiet der Entwurfsmethodik, welches besonders im Bereich des objektorientierten Entwurfs wichtige Beiträge zur Entwicklung besserer Software leistet (vgl. [7], um nur das prominenteste Werk zu nennen).

1.1.2 Sprachunterstützung

Eine andere Perspektive nimmt der Bereich der Sprachunterstützung ein, der sich damit beschäftigt, geeignete Mechanismen zu entwickeln, mit denen Modularität durch eine Programmiersprache unterstützt werden kann. Diese Mechanismen gehören zur *Pragmatik* einer Programmiersprache: Durch Modularität kann man keine Algorithmen formulieren, die in der Theorie nicht auch ohne Modularität realisierbar wären. Die Vorteile modularer Programmierung sind also ausschließlich praktischer Natur.

Eine Programmiersprache, die Modularität unterstützt, muss unter anderem folgende (nicht immer klar zu trennenden) Funktionalitäten bieten:

- Sprachliche Mittel zur Definition von Modulen
- Sprachliche Mittel zur Definition von Schnittstellen zwischen Modulen
- Verfahren zur Überprüfung, ob die Schnittstellen eingehalten wurden
- Verfahren, um aus einer Menge von Modulen ein Gesamtprogramm zu generieren

In der konkreten Ausgestaltung dieser Ziele gibt es große Unterschiede zwischen einzelnen Sprachen. Auch Sprachen, die derselben Gruppe zuzuordnen sind (beispielsweise der Gruppe der *funktionalen* Programmiersprachen), unterscheiden sich mitunter erheblich in der Art, wie sie Modularität unterstützen.

Diese Arbeit ist dem Bereich der Sprachunterstützung zuzuordnen. Dabei spielt jedoch der Blickwinkel der formalen Verifikation eine besondere Rolle.

1.2 Formale Programmverifikation mit VeriFun

✓VeriFun [20, 21, 22, 25] ist ein semi-automatischer, induktiver Theorembeweiser für die Verifikation funktionaler Programme. Das System ermöglicht es seinem Benutzer, Programme in der eingebauten funktionalen Sprache $\mathcal{FP}$ zu formulieren und über diese Programme Korrektheitsaussagen zu machen. Diese Aussagen können dann mit Hilfe eines speziellen Beweiskalküls formal verifiziert werden.

Da Aussagen über solche Programme im allgemeinen nicht semi-entscheidbar sind, ist ein vollständige Automatisierung aller Beweise prinzipiell unmöglich. Das System enthält aber eine Reihe praxiserprobter Heuristiken und Verfahren, die den Benutzer bei der Beweisführung unterstützen.

Die einfache Logik und die intuitive Benutzerführung ermöglichen bereits nach kurzer Einarbeitung ein Arbeiten mit dem System. Dadurch ist ✓VeriFun besonders für den Einsatz in der Lehre geeignet, wo es eine gute praktische Ergänzung zum eher theoretischen Gebiet der Semantik von Programmiersprachen bietet. Das System wurde bereits in verschiedenster Form erfolgreich in Lehrveranstaltungen eingesetzt [25]. Gleichzeitig dient das System als experimentelle Plattform, um neue Methoden und Ansätze zur Automatisierung von Beweisen unmittelbar testen zu können.

Obwohl ✓VeriFun hauptsächlich zum Zwecke der Programmverifikation entwickelt wurde, können auch andere mathematische Gebiete modelliert werden, sofern sich die entsprechenden Objekte durch rekursive Funktionsdefinitionen beschreiben lassen. Beispielsweise kann mit Hilfe des Systems die Existenz und Eindeutigkeit der Primfaktorzerlegung rechnergestützt bewiesen werden [23].

1.2.1 Die Sprache $\mathcal{FP}$

Programme für ✓VeriFun werden in der Sprache $\mathcal{FP}$ definiert. Die Sprache $\mathcal{FP}$ ist eine rein funktionale, statisch getypte Sprache mit polymorphen Datentypen. Die Grundzüge der Syntax sollen hier kurz an einigen Beispielen verdeutlicht werden.

Die rekursive Definition der Addition auf natürlichen Zahlen wird in $\mathcal{FP}$ wie folgt geschrieben:

```
function plus(x,y:nat):nat ⇐
  if x=0
    then y
    else succ(plus(pred(x),y))
  end_if
```

Der Datentyp *nat* mit der Vorgänger- und Nachfolgerfunktion *pred* und *succ* und der Konstante 0 sind dabei vordefiniert.

Über eine Datenstrukturdefinition können neue Datentypen eingeführt werden:

```
structure sexpr ⇐
  nil,
  atom(index:nat),
  cons(car:sexpr, cdr:sexpr)
```

Diese Definition beschreibt eine freie Datenstruktur *sexpr* mit den Konstruktoren *nil*, *atom* und *cons* und den Selektoren *index*, *car* und *cdr*. Die möglichen Werte sind genau die Terme, die nur aus den Konstruktoren aufgebaut sind (sog. Konstruktorgrundterme). Beispiele für Terme vom Typ *sexpr* sind *nil*, *atom(0)*, *cons(nil,atom(succ(0)))*.

In $\mathcal{FP}$ können polymorphe Funktionen[1] und Datenstrukturen definiert werden. Dabei handelt es sich um sogenannte parametrische Polymorphie [4]: Polymorphe Datenstrukturen haben einen oder mehrere Typparameter und definieren damit Typoperatoren. Beispielsweise werden Listen wie folgt definiert:

```
structure list[@X] ⇐
  empty,
  add(hd:@X,tl:list[@X])
```

Damit ist *list* ein einstelliger Typoperator. Listen über natürliche Zahlen haben dann beispielsweise den Typ *list[nat]*. Typvariablen werden generell mit einem @-Zeichen versehen. Polymorphe Funktionen können dann ebenfalls definiert werden:

```
function append(k,l:list[@X]):list[@X] ⇐
  if k=empty
    then l
    else add(hd(k),append(tl(k),l))
  end_if
```

Hier ist die Funktion *append* über allgemeine Listen definiert und kann auf alle Arten von Listen angewandt werden. Die Typvariable $@X$ in der Signatur von *append* ist dabei durch einen impliziten Allquantor gebunden. Damit hat die Funktion die Signatur

$$append : \forall @X : list[@X], list[@X] \rightarrow list[@X].$$

Beweise über benutzerdefinierte Funktionen können erst geführt werden, wenn zuvor die Terminierung dieser Funktionen gezeigt wurde. Durch die eingebaute Terminierungsanalyse [19, 26] geschieht dies in der Mehrzahl der Fälle vollautomatisch.

1.2.2 Lemmata und Beweise

Über $\mathcal{FP}$-Programme kann der Benutzer Aussagen in Form sogenannter Lemmata machen. Das folgende Lemma beschreibt z.B. die Assoziativität der Addition:

[1] Zur Abgrenzung gegenüber dem mathematischen Funktionsbegriff werden `function`-Definitionen auch als *Funktionsprozeduren* bezeichnet. In dieser Arbeit werden aber beide Begriffe synonym verwendet.

`lemma` *plus_associative* $\Leftarrow \forall$ *x,y,z:nat*
plus(x,plus(y,z))=plus(plus(x,y),z)

Ein Beweis wird im allgemeinen dadurch gestartet, dass der Benutzer durch Auswahl des *Verify*-Befehls die Standardtaktik startet. Das System versucht nun, über eine Reihe erprobter Heuristiken den Beweis selbstständig durchzuführen. Falls das fehlschlägt, bricht der Beweisversuch ab und der Benutzer hat die Möglichkeit, den begonnenen Beweis durch die Anwendung einer von dreizehn Beweisregeln fortzuführen oder abzuändern. Dabei gibt es zum einen interaktive Beweisregeln wie etwa *Use Lemma* (um ein Lemma anzuwenden) oder *Case Analysis* (um im Beweis mehrere unterschiedliche Fälle zu unterscheiden) und zum anderen automatische Beweisregeln. Automatische Beweisregeln starten den sogenannten *Symbolischen Auswerter*, einen vollautomatischen Rewrite-Kalkül, der das Beweisziel soweit wie möglich simplifiziert. In der Standardkonfiguration übernimmt ✓`eriFun` nach einer manuellen Regelanwendung automatisch wieder die Initiative und versucht mittels eingebauter Heuristiken, den Beweis selbstständig zu Ende zu führen.

1.3 Die Rolle axiomatischer Spezifikationen

In gängigen Programmiersprachen bestehen Schnittstellendefinitionen meist aus einer Menge von Funktionsnamen und deren Typ.

Ein einfaches Beispiel ist die Schnittstelle eines Sortieralgorithmus für Listen natürlicher Zahlen. Sie besteht nur aus einer Funktion:

$$sort : list[nat] \rightarrow list[nat]$$

In der Schnittstellendefinition sind alle Informationen enthalten, die der Compiler benötigt, um zu überprüfen, ob ein Aufruf wohlgetypt ist. Damit können andere Programmteile auf die Sortierfunktion zugreifen, ohne dass sie die genaue Implementierung des Verfahrens kennen müssen. Die Schnittstelle garantiert, dass sich die Programmteile hinterher kombinieren lassen und das entstehende Programm korrekt getypt ist.

Betrachten wir eine Funktion, die die obige Schnittstelle implementiert:

`function` *sort(l:list[nat]):list[nat]* $\Leftarrow$ *l*

Offensichtlich wird die Schnittstelle eingehalten: Die Funktion nimmt eine Liste natürlicher Zahlen entgegen und gibt auch eine solche Liste zurück. Allerdings ist das Ergebnis genau die Eingangsliste. Die Funktion sortiert also nicht, wie das eigentlich gewünscht war.

Das Problem liegt darin, dass die Schnittstellendefinition zwar die Argument- und Rückgabetypen der Funktionen beschreibt, aber nichts über das Verhalten der Funktionen aussagt. Das bedeutet nicht, dass das genaue Verhalten in der Praxis unwichtig ist. Es wird aber in der Regel nur informell in Form von Kommentaren oder anderen Beschreibungen ausgedrückt.

Für die Programmverifikation ist diese Art der Schnittstellenbeschreibung ganz offensichtlich ungenügend. Ein Korrektheitsbeweis für eine Anwendungskomponente A, die auf einen Sortieralgorithmus S aufbaut, kann nur gelingen, wenn auch die Eigenschaften von S mit einbezogen werden. Prinzipiell gibt es zwei Möglichkeiten, wie das geschehen kann:

- Im Beweis für A wird mangels anderer Information auf die Implementierung von S und eventuell bekannte Aussagen über S zugegriffen.
- Die Eigenschaften, die über S vorausgesetzt werden, werden vorab explizit als Axiome formuliert und als Bestandteil der Schnittstelle betrachtet. Der Beweis von A kann sich auf diese Axiome stützen. Von S muss bewiesen werden, dass es die Axiome erfüllt.

Die erste Variante erscheint zunächst einfacher, widerspricht aber dem Prinzip der Separation of Concerns: Der Beweis für A ist von der Implementierung einer anderen Komponente abhängig. Damit ist es z.B. nicht mehr möglich, S nachträglich durch einen anderen Sortieralgorithmus zu ersetzen.

Also sollte die Schnittstelle des Sortieralgorithmus durch Axiome erweitert werden:

$$\begin{gathered} sort : list[nat] \to list[nat] \\ \forall l : list[nat] \quad ordered(sort(l)) \\ \forall l : list[nat], x : nat \quad occurs(x, l) = occurs(x, sort(l)) \end{gathered}$$

Die Funktion *ordered* überprüft dabei, ob eine Liste geordnet ist. Die Funktion *occurs* zählt die Anzahl der Vorkommen eines Elements in einer Liste. Damit beschreiben die Axiome, dass das Ergebnis der Funktion *sort* erstens eine geordnete Liste und zweitens eine Permutation der Ausgangsliste ist. Es wird aber nicht festgelegt, *wie* diese sortierte Liste zustande kommt. Damit kann als Implementierung ein beliebiger korrekter Sortieralgorithmus verwendet werden. Beweise für die Komponente A können nun auf diese Aussagen zurückgreifen.

Diese Art von Schnittstellenbeschreibungen nennt man *axiomatische Spezifikationen*.

Der Grund, warum axiomatische Spezifikationen sich in der Programmierpraxis nicht als Schnittstellenbeschreibungen durchgesetzt haben, liegt darin, dass es nicht entscheidbar ist, ob eine Schnittstelle von einer gegebenen Implementierung auch erfüllt wird. Denn anstelle des einfachen Typechecking muss dann ein formaler Beweis geführt werden, der die Gültigkeit der Axiome garantiert. Für die Programmierung ist dieser Aufwand im allgemeinen zu groß, und so begnügt man sich damit, die syntaktischen Eigenschaften als Schnittstellen anzugeben, die vom Compiler vollautomatisch überprüft werden können. Für ✓`eriFun` gilt dieses Argument selbstverständlich nicht, denn hier ist es ja gerade das Ziel, formale Beweise zu führen.

Man sieht, dass ein Modulkonzept für die Programmverifikation gleichzeitig ein Konzept für die axiomatische Spezifikation beinhalten muss. Daher beanspruchen axiomatische Spezifikationen auch den Hauptteil dieser Arbeit.

1.3.1 Algebraische Strukturen

Axiomatische Spezifikationen treten aber nicht nur als Schnittstellenbeschreibungen von Programmen auf. Eine Vielzahl wichtiger algebraischer Strukturen ist axiomatisch definiert:

Definition. *Eine Gruppe* $(G, \cdot, e, ^{-1})$ *besteht aus einer Menge* G*, einer binären Operation* $\cdot$ *auf* G*, einem Element* $e \in G$ *und einer Abbildung* $x \to x^{-1}$ *über* G*, so dass folgende Axiome erfüllt sind:*

1. $\forall x, y, z \in G \quad x \cdot (y \cdot z) = (x \cdot y) \cdot z$
2. $\forall x \in G \quad x \cdot e = x = e \cdot x$
3. $\forall x \in G \quad x \cdot x^{-1} = e = x^{-1} \cdot x$

Diese Art von Definition kann man direkt als Schnittstellenspezifikation interpretieren. Dabei werden ein Typ G für die Grundmenge der Gruppe und drei Funktionssymbole *op*, *neut* und *inv* für die Operationen definiert (Das neutrale Element kann man als nullstellige Operation auffassen). Die Axiome beschreiben das Verhalten der Operationen.

In ✓eriFun sind Beweise über solche Strukturen bisher nicht möglich gewesen, da es an der Möglichkeit fehlte, entsprechende Axiomatisierungen anzugeben und zu verwalten. Die einzige Möglichkeit bestand darin, eine konkrete Implementierung der Struktur zu programmieren und über diese Implementierung Beweise zu führen. Diese Beweise sind dann aber nicht allgemeingültig und müssen für jede Implementierung neu geführt werden.

Durch die Ähnlichkeit zwischen Schnittstellenbeschreibungen einerseits und algebraischen Strukturen andererseits kann man mit den Spracherweiterungen aus dieser Arbeit gleich zwei interessante Ziele erreichen: Die Modularisierung und die Möglichkeit der Behandlung algebraischer Strukturen. Für Beispiele wird daher auch immer wieder auf solche Strukturen zurückgegriffen.

Kapitel 2

Modulkonzepte in Programmiersprachen und Beweisern

2.1 Grundlegende Prinzipien

Bereits früh in der Entwicklung von Programmiersprachen hat sich herausgestellt, dass besondere Sprachkonzepte notwendig sind, um Modularität zu ermöglichen und zu unterstützen.

Eine der ersten Sprachen, die diese Möglichkeiten bot, war Niklas Wirths Modula. Im Jahre 1977 entworfen, sollte Modula einige strukturelle Probleme der Sprache Pascal beheben. Eines dieser Probleme war die fehlende Möglichkeit, verschiedene Programmteile separat kompilieren zu können. Mit immer größer werdenden Programmen wurde diese Funktionalität dringend gebraucht, was bereits zu einem Wildwuchs von verschiedenen (zueinander inkompatiblen) Erweiterungen von Pascal geführt hatte.

Das Modulkonzept von Modula ist sehr einfach und überraschend modern: Module sind Sammlungen von Konstanten-, Variablen-, Typ- und Funktionsdefinitionen, die außerdem eine „Import-“ und eine „Exportliste“ besitzen. In der Importliste werden einzelne Symbole aus anderen Modulen importiert. Die Exportliste bestimmt, welche Symbole von anderen Modulen verwendet (also importiert) werden dürfen.

Im Nachfolger Modula-2 wird die Exportliste dann durch eine explizite Schnittstellendefinition in einer separaten Datei ersetzt. Damit unterscheidet man zwischen sogenannter Moduldefinition (der Schnittstelle) und Modulimplementierung. Die Definition enthält nur die Namen und Typen der Symbole, aber keine Werte oder Implementierungen. Folgendes Beispiel ist eine Moduldefinition für einen Stack:

```
DEFINITION MODULE IntStack;
  TYPE anIntStack;

  PROCEDURE New( VAR s : anIntStack );
  PROCEDURE Push( VAR s : anIntStack; i : INTEGER );
  PROCEDURE Pop( VAR s : anIntStack ) : INTEGER;
  PROCEDURE Empty( s : anIntStack ) : BOOLEAN;
END IntStack.
```

Um ein Modul zu übersetzen, müssen dem Compiler nur die Schnittstellen der importierten Module bekannt sein. Diese stellen genug Informationen bereit, um das Programm auf syntaktische Korrektheit zu überprüfen. Erst zur Laufzeit wird das Modul dann mit anderen Modulen verknüpft (Linking) und damit ausführbar gemacht. So ist es möglich, Änderungen an einzelnen Modulen vorzunehmen, ohne dass das gesamte System neu kompiliert werden muss.

Das Modulsystem von Modula-2 findet sich nahezu unverändert in vielen heute gängigen Programmiersprachen wieder. Beispielsweise sind die in C benutzten Header-Files genau die Moduldefinitionen von Modula. Das Modulsystem von Haskell zeigt, dass das Konzept auch für funktionale Sprachen geeignet ist.

Jeder Moduldefinition steht in diesem Konzept genau eine Modulimplementierung gegenüber. Durch die abstrakten Schnittstellen ist es zwar theoretisch möglich, mehrere Implementierungen für eine Schnittstelle zu haben. Ein Austausch der Implementierung ist aber nur durch Austausch der betreffenden Datei und Neukompilierung möglich, da es keine innersprachlichen Mittel zum Ansprechen mehrerer Implementierungen gibt.

Erst mit Einzug der Objektorientierung wurde auch die Programmkomposition in die Sprachen selbst verlagert. Verschiedene Implementierungen einer Schnittstelle können dort nebeneinander existieren. Das Typsystem stellt dabei sicher, dass die Schnittstellen beim Zusammensetzen des Programms eingehalten werden.

Alle Sprachen haben aber als Grundprinzip gemeinsam, dass der Zugriff auf ein Modul eingeschränkt wird. Diese Einschränkung dient nicht der Umsetzung von anwendungsbezogenen Schutzzielen (wie etwa der Vertraulichkeit bestimmter Daten), sondern ist eher als Schutz gegen „unsachgemäße Verwendung“ durch andere Module anzusehen[2]. Gewisse Implementierungsdetails werden vor der Außenwelt verborgen und sind dadurch leichter austauschbar.

Unterschiede gibt es in der Art, wie Schnittstellen angegeben werden. Dies kann entweder dadurch geschehen, dass alle Elemente zur Schnittstelle gehören, die als öffentlich zugänglich deklariert sind (z.B. Modula-Exportliste, Java-Klassen, Haskell-Module). Oder es existiert eine separate Schnittstellendefinition (z.B. Modula2-Moduldefinition, Java-Interfaces, C-Header-Files).

In diesem Kapitel sollen anhand der funktionalen Sprachen *Haskell* und *Standard ML* zwei weitergehende Konzepte beschrieben werden. Aus Haskell wird das System der Typklassen betrachtet, welches auch im Theorembeweiser *Isabel-*

[2]Dementsprechend ist der Schutz auch häufig leicht zu umgehen und sollte nicht für sicherheitskritische Zwecke verwendet werden

le/HOL verwendet wird, um axiomatisch definierte Strukturen zu unterstützen. Das Modulsystem von Standard ML wurde hingegen noch nicht in einem Theorembeweiser implementiert. Am Ende wird das Prinzip der *Funktionalen Instanziierung* des Theorembeweisers ACL2 vorgestellt.

2.2 Typklassen

Die Sprache Haskell [11] bietet mit Typklassen [18, 8] eine Form, Schnittstellenspezifikationen anzugeben, die an das Typsystem gekoppelt ist. Das eigentliche Modulsystem von Haskell ist dem von Modula sehr ähnlich und soll daher hier nicht weiter betrachtet werden.

Typklassen wurden mit dem Ziel eingeführt, eine uniforme Behandlung von Funktionen zu ermöglichen, die auf verschiedenen Typen unterschiedlich definiert sind (Overloading): So gibt es beispielsweise verschiedene Darstellungsformen für Zahlen (Integer, Float, ...) und für alle Zahltypen die entsprechenden arithmetischen Operationen. Typklassen ermöglichen es, Zahltypen zu einer gemeinsamen Typklasse `Num` zusammenzufassen. Dann können auf uniforme Weise Funktionen über diese Klasse definiert werden (z.B. funktioniert Mittelwertbildung auf allen Zahltypen gleich, nur die Implementierung der Basisoperationen „+“ und „/“ ist darstellungsabhängig). Typische Anwendungen von Typklassen sind neben den Zahlendarstellungen auch die einheitliche Behandlung der Konversion in einen String (zur Ausgabe), oder die Behandlung von Ordnungsrelationen oder Gleichheit.

Obwohl Typklassen zunächst keine Modularisierung zum Ziel haben, sind sie für diese Arbeit von Interesse, da sie die Definition von Schnittstellen erlauben.

2.2.1 Einführung

Beispiel 2.1. Die folgende Definition beschreibt eine Typklasse `Ord` für geordnete Typen[3].

```
class Ord a where
  eq  ::   a -> a -> Bool
  leq ::   a -> a -> Bool
```

Hier ist `a` eine Typvariable und `Bool` der vordefinierte Boolesche Typ. Diese Definition kann man lesen als „Ein Typ `a` gehört zur Klasse `Ord`, wenn es Funktionen `eq` und `leq` mit den angegebenen Signaturen gibt.“

Die Zugehörigkeit eines Typs zu einer Klasse wird durch die Angabe einer Instanz begründet. Instanzen werden durch Angabe von Implementierungen der Funktionssymbole definiert:

[3] Funktionstypen sind in Haskell in der Regel „curried“ angegeben: Der Typ $a \to b \to c$ sollte daher intuitiv als $a \times b \to c$ gelesen werden. Für einen Typ a bezeichnet $[a]$ den Listentyp über a, Listenkonstruktoren sind [] für die leere Liste und : für die add-Operation. Selektoren gibt es nicht, stattdessen wird in Definitionen Pattern-Matching verwendet.

```
instance Ord Integer where
  eq  = (==)
  leq = (<=)
```

Hier bezeichnen `==` und `<=` die vordefinierten Vergleichsfunktionen.

Nun ist es möglich, Funktionen sozusagen „simultan“ für alle Instanzen dieser Klasse zu definieren, wie die folgende Funktion, die überprüft, ob eine Liste geordnet ist:

```
ordered :: (Ord a) => [a] -> Bool

ordered []       = True
ordered (x:[])   = True
ordered (x:y:ys) = (leq x y) && ordered y:ys
```

Die Signatur der Funktion `ordered` ist hier nur zum besseren Verständnis explizit angegeben, sie wird automatisch durch das Typinferenzsystem [5] erkannt. Der erste Teil des Typausdrucks (links von „`=>`“) wird *Kontext* genannt. Er enthält die Klassenbeschränkung für den ansonsten unspezifizierten Typ `a`.

Wenn die überladene Funktion `ordered` auf eine Liste angewandt wird, wählt Haskell automatisch die richtige Implementierung von `leq` aus. Das Kriterium dafür ist der Elementtyp der Liste. Durch den Kontext `(Ord a)` ist sichergestellt, dass `ordered` nur für Typen definiert ist, für die es eine `Ord`-Instanz gibt.

Daraus ergibt sich die Einschränkung, dass eine Klasse stets nur maximal eine Instanz für einen bestimmten Typ haben kann. Eine weitere Instanz mit dem Typ `Integer` und beispielsweise der umgekehrten Ordnung ist nicht möglich, denn dann wäre der Aufruf von `ordered([8,4,3])` nicht mehr eindeutig.

Beispiel 2.2. In einem Kontext können auch mehrere unterschiedliche Typen derselben Klasse enthalten sein. Folgende Funktion ist z.B. auf Paaren definiert und repräsentiert die lexikographische Ordnung:

```
lex :: (Ord a, Ord b) => (a,b) -> (a,b) -> Bool

lex (x1,x2) (y1,y2) |  (eq x1 y1) = (leq x2 y2)
                    |  otherwise  = (leq x1 y1)
```

Die gewöhnungsbedürftige Syntax erklärt sich, wenn man weiß, dass es sich hier um Gleichungen zwischen dem `lex`-Ausdruck ganz links und den `leq`-Ausdrücken ganz rechts handelt, und die Ausdrücke in der Mitte die Bedingungen sind, unter denen die jeweilige Gleichung gilt.

2.2.2 Typklassen vs. Typabstraktion

Obwohl Typklassen ein Mittel zur Spezifikation von Schnittstellen sind, ermöglichen sie nicht die Definition von abstrakten Datentypen, da die Implementierung nicht gekapselt wird. Das verdeutlicht der folgende Versuch, einen abstrakten Integer-Stack mit Hilfe von Typklassen zu definieren:

```
class IntStack a where
  emptyStack :: a
  push :: a -> Integer -> a
  top :: a -> Integer
  pop :: a -> a
```

Neben der Konstante für den leeren Stack werden drei Operationen `push`, `top` und `pop` definiert. Als nächstes geben wir eine Implementierung eines Stacks mittels Listen an:

```
instance IntStack [Integer] where
  emptyStack = []
  push xs x = (x:xs)
  top (x:xs) = x
  pop (x:xs) = xs
```

Diese Definitionen werden ohne weiteres vom Compiler akzeptiert. Der Versuch, einen solchen Stack zu verwenden, führt aber zu einem Fehler:

```
> emptyStack
ERROR - Unresolved overloading
*** Type       : Stack a => a
*** Expression : emptyStack
```

Das ist nicht verwunderlich. Der Interpreter kann die Typvariable `a` aus dem Typ von `emptyStack` nicht aus dem Kontext mit einem konkreten Typ belegen. Daher ist es nicht möglich, automatisch die richtige Instanz der Typklasse `Stack` auszuwählen und deren `emptyStack`-Implementierung aufzurufen, da gar nicht klar ist, welche Implementierung jetzt gemeint ist. Durch eine konkrete Typannotation produziert man klare Verhältnisse:

```
> (emptyStack :: [Integer])
[]
```

Dass man als Ergebnis die leere Liste erhält, ist zwar konsequent (es wurde schließlich explizit ein Wert vom Typ `[Integer]` verlangt), aber nicht unbedingt erwünscht, denn wir wollten ja eigentlich einen abstrakten Datentyp konstruieren, dessen Implementierung nach außen nicht sichtbar ist.

Es gibt aber keinen Mechanismus, der verhindert, dass die Implementierung eines Stacks von aussen sichtbar ist oder manipuliert wird. Dies ist auch nicht das Designziel von Typklassen, die es lediglich ermöglichen sollen, Funktionen wie `ordered` aus dem vorangegangenen Abschnitt zu definieren.

Aus Sicht des Beweisens ist diese Einschränkung allerdings weniger wichtig. Dort wird die Unabhängigkeit von Komponenten ja nicht allein dadurch hergestellt, dass man verhindert, dass andere Komponenten an der internen Repräsentation manipulieren. Stattdessen beweist man Aussagen, die über alle möglichen Implementierungen quantifizieren. Der folgende Abschnitt über axiomatische Typklassen wird zeigen, dass eine Verwendung von Typklassen in Beweissystemen durchaus möglich ist, wenn auch einige Einschränkungen bestehen bleiben.

2.2.3 Axiomatische Typklassen in Isabelle/HOL

Im Theorembeweiser Isabelle/HOL [14, 15] wurde das System der Typklassen zu *axiomatischen Typklassen* [27] erweitert, so dass es verwendet werden kann, um abstrakte Beweise über axiomatische Spezifikationen zu führen und diese Beweise auf konkrete Instanzen zu übertragen.

Axiomatische Typklassen sind Typklassen, die zusätzlich Axiome enthalten dürfen. So können neben der rein syntaktischen Schnittstelleninformation auch Forderungen an das Verhalten der Funktionen gestellt werden. In Beispiel 2.1 würde man etwa die Antisymmetrie der Ordnung fordern. Bei der Definition einer Instanz müssen diese Axiome dann bewiesen werden.

Beispiel 2.3. Wir definieren eine axiomatische Typklasse für ein Monoid. In Isabelle/HOL werden Typklassen nur durch die Axiome angegeben. Die Funktionssymbole müssen vorher als globale, polymorph getypte Symbole eingeführt werden[4]:

```
consts
  times :: 'a ⇒ 'a ⇒ 'a      (infixl ⊙ 70)
  one   :: 'a
```

Danach definiert man die eigentliche Typklasse:

```
axclass monoid ⊆ type
  assoc         : x ⊙ (y ⊙ z) = (x ⊙ y) ⊙ z
  left_neutral  : one ⊙ x = x
  right_neutral : x ⊙ one = x
```

Nun lässt sich die Funktion *fold* definieren, die Listen mit der ⊙-Operation zusammenfaltet:

```
consts
  fold :: 'a list ⇒ 'a

primrec
  fold [] = one
  fold (x # xs) = x ⊙ (fold xs)
```

Für Listen mit Elementen aus einem Monoid zeigt man das Lemma *fold_append*, welches beschreibt, wie *fold* auf die Verkettung zweier Listen (in Isabelle/HOL mit @ bezeichnet) wirkt. Um die Forderung auszudrücken, dass die beiden Listen *xs* und *ys* Elemente der Typklasse *monoid* haben, fügt man eine explizite Typannotation ein, wenn *xs* bzw. *ys* das erste Mal verwendet werden:

```
theorem fold_append:
  fold (xs::( 'a::monoid list) @ ys::('a::monoid list))
    = (fold xs) ⊙ (fold ys)
```

[4] Durch consts-Deklarationen werden in Isabelle/HOL neue Symbole eingeführt und deren Typ angegeben. Typvariablen erkennt man am führenden Hochkomma. Typoperatoren werden nachgestellt: *'a list* ist der polymorphe Listentyp. Listenkonstruktoren sind [] und #. Zusätzlich zur Standard-Präfixschreibweise kann in Klammern eine alternative Syntax vereinbart werden, indem man ein Symbol, die Art der Dartellung (Infix, Präfix etc.) und die Bindungsstärke angibt.

Der Ausdruck *xs::('a::monoid list)* ist folgendermaßen zu lesen: Die Variable *xs* hat den Typ *'a list*, wobei *'a* zur Typklasse *monoid* gehört.

Um nun Instanzen der axiomatischen Typklasse *monoid* zu definieren, müssen zunächst die Operationen mit konkretem Inhalt gefüllt werden. Isabelle/HOL unterstützt Overloading, über das eine Konkretisierung der Symbole für den Typ *nat* angegeben wird. Dann kann die Instanz definiert werden.

`defs` *(*`overloaded`*)*
  one ≡ *0::nat*
  x:: nat ⊙ *y::nat* ≡ *x* + *y*

`instance` *nat :: monoid*

Aus der Instanz ergeben sich die Axiome aus der Typklasse als Beweisverpflichtungen für den Typ *nat*. Sind die Axiome bewiesen, so wird *nat* als ein Untertyp von *'a* :: *monoid* angesehen. Damit ist das bewiesene Theorem *fold_append* automatisch auch für den Instanztyp *nat* anwendbar.

2.2.4 Probleme

An den Beispielen zu axiomatischen Typklassen kann man sehen, dass Typklassen zur Beschreibung axiomatischer Spezifikationen prinzipiell geeignet sind. In einigen Fällen gibt es jedoch Einschränkungen, da Spezifikationen mit einer bestimmten Struktur hier Mehrdeutigkeiten produzieren. Die Ursache der Probleme ist in dem Mechanismus zu suchen, der die Typinformation verwendet, um danach die richtige Instanz einer Klasse auszuwählen. Man kann Fälle konstruieren, in denen die Typen nicht ausreichend Informationen dafür bieten. Diese Fälle sollen im folgenden kurz besprochen werden.

Spezifikationen ohne Typsymbol

Durch die Bindung an den Typ lassen sich bestimmte Spezifikationen nicht ohne weiteres mit Typklassen beschreiben. Betrachten wir noch einmal die Sortierspezifikation aus Abschnitt 1.3:

$$sort : list[nat] \to list[nat]$$

$$\forall l : list[nat] \quad ordered(sort(l))$$

$$\forall l : list[nat], x : nat \quad occurs(x, l) = occurs(x, sort(l))$$

Diese Spezifikation macht lediglich Aussagen über eine einzige nicht-polymorphe Funktion: Alle Implementierungen der Sortierfunktion haben den selben Typ. Es ist daher nicht möglich, die richtige Implementierung anhand der Parametertypen zu bestimmen. Die Auflösung anhand des Typs ist in diesem Beispiel prinzipiell nicht möglich.

Mit einem unschönen Trick kann man das Problem aber umgehen. Dafür erweitert man die Signatur von `sort` um einen Dummy-Typ, über den sich eine

Typklasse formulieren lässt (Wir verwenden wieder Haskell-Syntax und ignorieren daher die Axiome):

```
class Sorting a where
  sort :: a -> [Integer] -> [Integer]
```

Möchte man nun zwei Sortieralgorithmen als Instanzen der Typklasse definieren, erzeugt man zwei (einelementige) Typen, die als Unterscheidungsmerkmal dienen:

```
data ISort = ISortC

data MSort = MSortC
```

Mit Hilfe dieser Konstanten lassen sich dann die Instanzen definieren:

```
instance ISort of Sorting where
  sort == (lambda x,l. (isort l))

instance MSort of Sorting where
  sort == (lambda x,l. (msort l))
```

Man sieht, dass es zwar prinzipiell möglich ist, auch Spezifikationen ohne ein Typsymbol mit Typklassen auszudrücken, allerdings kann diese Notation allenfalls als Workaround bezeichnet werden.

Typrelationen

Es sind auch Spezifikationen denkbar, die mehr als einen Typ enthalten. Ein Beispiel dafür sind Vektorräume, in denen es zwei Grundmengen gibt: Skalare und Vektoren.

Prinzipiell ist es möglich, das Konzept der Typklassen auf „mehrstellige“ Typklassen (also Typrelationen) zu erweitern. In der offiziellen Fassung [11] erlaubt Haskell keine Definition von Typrelationen. Mittlerweile werden diese allerdings von den meisten Compilern und Interpretern unterstützt. So kann man über Typrelationen z.B. definieren, dass zwei Typen aufeinander abgebildet werden können.

```
class Mappable a b where
  forward  :: a -> b
  backward :: b -> a
```

Bereits in diesem simplen Beispiel lässt sich der Typ des Ausdrucks der Form `(forward 1)` aber nicht mehr eindeutig bestimmen, sobald es mehrere Typen gibt, auf die der Typ Integer abgebildet werden kann.

Dementsprechend sind Typrelationen in Haskell zur Zeit nur sehr eingeschränkt verwendbar. Sogenannte *funktionale Abhängigkeiten* [10] bieten die Möglichkeit,

die Menge der legalen Instanzen von vornherein so einzuschränken, dass Mehrdeutigkeiten nicht auftreten können. Für die obige Klasse würde man dann einfach verbieten, dass mehrere Instanzen definiert werden, bei denen der Typ `a` durch `Integer` instanziiert wird.

Mehrfachinstanziierung

In Abschnitt 2.2.1 hat man bereits gesehen, dass es nicht möglich ist, mehrere Instanzen einer Klasse anzugeben, die auf dem gleichen Typ aufbauen. Das ist jedoch eine starke Einschränkung für die Praxis. Aus der operationalen Sicht von Haskell ist es sinnvoll, dies zu verbieten, da sonst die Zuordnung der Instanzen nicht mehr eindeutig möglich ist. Im Allgemeinen wäre es aber dennoch nützlich und wichtig, z.B. folgendes spezifizieren zu können:

- Verschiedene Ordnungen auf derselben Grundmenge
- Verschiedene Gruppenstrukturen auf derselben Grundmenge
- Verschiedene Implementierungen von Prioritätswarteschlangen, die alle auf Listen beruhen, aber unterschiedliche Sortierungsvarianten verwenden

Die Verknüpfung von Typklassen mit Overloading macht diese Mehrfachinstanziierung aber unmöglich. Daher kann es auch z.B. in Isabelle/HOL maximal eine Gruppenstruktur für einen Typ geben.

2.3 Das Modulsystem von Standard ML

Das Modulsystem von Standard ML [13] bietet einige mächtige Werkzeuge zum Definieren modularer Programme und zur typsicheren Komposition einzelner Module zu größeren Einheiten.

Die Kernsprache von ML hat wie Haskell ein polymorphes Typsystem mit Hindley-Milner-Typinferenz [5]. Die Modulsprache ist auf diese Kernsprache aufgesetzt, aber weitgehend von ihr unabhängig, was eine Übertragung in einen anderen Kontext erleichtert. Das Modulsystem von Standard ML basiert im wesentlichen auf zwei Konzepten: Signaturen und Strukturen.

2.3.1 Signaturen

Signaturen sind Blöcke, in denen Deklarationen zusammengefasst sind[5]:

```
signature QUEUE =
  sig
    type 'a queue
    exception Empty
```

[5] Viele Beispiele in diesem Abschnitt sind [9] entnommen. Die Syntax für Typausdrücke entspricht der von Isabelle/HOL. Listenkonstruktoren sind *nil* und ::.

```
    val empty : 'a queue
    val insert : 'a * 'a queue -> 'a queue
    val remove : 'a queue -> 'a * 'a queue
  end
```

Dabei ist `QUEUE` eine Signaturvariable. Diese bekommt die durch `sig` ... `end` beschriebene Signatur zugewiesen. Die Notation `'a queue` steht für einen einstelligen Typkonstruktur `queue` (`'a` ist eine Typvariable). Der Typoperator `*` steht für das karthesische Produkt aus zwei Typen. Das Schlüsselwort `exception` steht für einen undefinierten Wert, der zurückgegeben werden kann, um z.B. Fehler anzuzeigen.

Unter anderem sind in einer Signatur folgende Deklarationen erlaubt:

Typdeklarationen Typdeklarationen definieren einen neuen (möglicherweise nullstelligen) Typkonstruktor. Optional ist es möglich, zusätzlich bereits eine Implementierung anzugeben. Typdeklarationen beginnen mit dem Schlüsselwort `type`.

Symboldeklarationen Deklariert ein Symbol mit einem bestimmten Typ. Da dieser Typ auch ein Funktionstyp sein kann, können so Funktionssymbole deklariert werden. Symboldeklarationen beginnen mit dem Schlüsselwort `val`. Eine Wertzuweisung für das Symbol bzw. eine Implementierung der Funktion erfolgt nicht.

2.3.2 Strukturen

Strukturen sind Implementierungen von Signaturen, in denen die Typ- und Wertdeklarationen konkretisiert werden. Beispielsweise implementiert folgende Struktur die Queue-Signatur von oben. Dabei wird eine Queue durch ein Paar von Listen dargestellt. Bei dieser Darstellung wird in die erste Liste eingefügt und aus der zweite Liste gelesen. Ist die zweite Liste leer, wird die erste Liste umgekehrt und in die zweite Liste geschrieben. So ist das Einfügen und das Auslesen in (im Durchschnitt) konstanter Zeit möglich:

```
structure Queue =
  struct
    type 'a queue = 'a list * 'a list
    exception Empty
    val empty = (nil, nil)
    fun insert (x, (b,f)) = (x::b, f)
    fun remove (nil, nil) = raise Empty
      | remove (bs, nil) = remove (nil, reverse bs)
      | remove (bs, f::fs) = (f, (bs, fs))
  end
```

Im Gegensatz zu Signaturen werden in Strukturen den `type`- und `val`-Komponenten konkrete Werte zugewiesen[6]. Die Komponenten einer Struktur können

[6] fun-Definitionen sind lediglich eine lesbarere Schreibweise für val-Definitionen, die der entsprechenden Variable einen Lambda-Ausdruck zuordnen.

dann über die Dot-Notation referenziert werden. Die Funktion `insert` aus der Struktur `Queue` wird also mit `Queue.insert` angesprochen.

Zu jeder Struktur wird automatisch eine *principal signature*, also die speziellstmögliche passende Signatur gebildet und der Struktur zugewiesen. Alternativ ist es möglich, einer Struktur explizit eine Signatur zuzuweisen:

```
structure Queue2 :> QUEUE = Queue
```

Hier wird der Strukturvariable `Queue2` die Struktur `Queue` zugewiesen, diese aber mit der Signatur `QUEUE` versehen. Durch die explizite Zuweisung einer Signatur ist es möglich, Implementierungsdetails der Struktur `Queue` zu verstecken. Da in der Signatur `QUEUE` der Implementierungstyp für `'a queue` nicht angegeben ist, wird dieser vor der Außenwelt verborgen. Diese Abstraktion führt dann dazu, dass der erste der beiden Aufrufe

```
Queue2.insert (1, ([], []))
Queue2.insert (1, Queue2.empty)
```

nicht mehr wohlgetypt ist, denn `Queue2.insert` muss mit einem Wert vom Typ `'a Queue2.queue` aufgerufen werden, der aktuelle 2. Parameter hat aber den Typ `('a list * 'a list)`. Der zweite Aufruf ist hingegen korrekt.

Dieses Abstraktionsprinzip durch Zuschreibung (*ascription*, vgl. [9]) einer allgemeineren Signatur erlaubt es, Implementierungsdetails zu verbergen und dient damit der Separation of Concerns.

2.3.3 Unterstrukturen

Strukturen dürfen in ML wiederum andere Strukturen enthalten, wodurch man hierarchische Strukturen erhält. Signaturen von hierarchischen Strukturen sind ebenfalls hierarchisch. In folgendem Beispiel wird eine Signatur `DICT` für eine Dictionary-Struktur definiert, die eine andere Signatur `ORDERED` enthält:

```
signature ORDERED =
  sig
    type t
    val lt : t * t -> bool
    val eq : t * t -> bool
  end

signature DICT =
  sig
    structure Key : ORDERED
    type 'a dict
    val empty : 'a dict
    val insert : 'a dict * Key.t * 'a -> 'a dict
    val lookup : 'a dict * Key.t -> 'a option
  end
```

Einträge im Dictionary sind vom Typ `'a` und werden durch den geordneten Typ `Key.t` referenziert. Das Dictionary selbst hat den Typ `'a dict`.

Strukturen mit der Signatur `DICT` haben dann eine Unterstruktur der Signatur `ORDERED`:

```
structure LexString : ORDERED =
  struct
    type t = string
    val eq = (op =)
    val lt = (op <)
  end

structure StringDict :> DICT =
  struct
    structure Key : ORDERED = LexString

    type 'a dict = ...[7]
    val empty = ...
    val insert = ...
    val lookup = ...
  end
```

2.3.4 Parametrisierung durch Funktoren

Eine natürliche Forderung ist es nun, das oben definierte Dictionary durch eine Ordnung zu parametrisieren. In ML wird das über Funktoren ermöglicht. Funktoren sind Abbildungen zwischen Strukturen oder (was gleichbedeutend ist) parametrisierte Strukturen.

Beispiel 2.4. Der folgende Funktor liefert für jede Struktur mit der Signatur `ORDERED` eine Struktur mit der Signatur `DICT`, die diese Ordnung verwendet:

```
functor DictFun (structure K : ORDERED) :> DICT =
  struct
    structure Key : ORDERED = K
    type 'a dict = ...
    val empty = ...
    val insert = ...
    val lookup = ...
  end
```

Über diesen Funktor kann man dann die Struktur `StringDict` aus dem letzten Abschnitt direkt gewinnen:

```
structure StringDict = DictFun(LexString)
```

[7] Die Implementierungen sind hier nicht von Interesse und werden ausgelassen

2.3.5 Sharing constraints

Die Parameter eines Funktors sind Strukturen, und ihre Typen sind Signaturen. Im Unterschied zu normalen Funktionen, die auf Werten operieren, können die Parameter von Funktoren aber noch teilweise voneinander abhängig sein. Sei beispielsweise folgende Signatur gegeben, die einen Typ und eine Funktion auf dem Typ enthält:

```
signature OPERATION =
  sig
    type t
    val f : t -> t
  end
```

Nun definieren wir einen Funktor, der zwei solche Operationen verketten soll:

```
functor Combine(structure Op1: OPERATION,
                structure Op2: OPERATION): OPERATION =
  struct
    type t = Op1.t;
    fun f x = Op1.f (Op2.f x)
  end
```

Diese Definition erzeugt aber einen Typfehler: Die Funktion `Op1.f` kann nicht auf den Typ `Op2.t` angewandt werden, denn im allgemeinen sind `Op1.t` und `Op2.t` nicht kompatibel. Diese Kompatibilität kann man aber als zusätzliche Bedingung an die Parameter stellen:

Beispiel 2.5.
```
functor Combine(structure Op1: OPERATION,
                structure Op2: OPERATION
                     where Op1.t = Op2.t): OPERATION =
  struct
    type t = Op1.t;
    fun f x = Op1.f (Op2.f x)
  end
```

Neu hinzugekommen ist hier nur die `where`-Klausel in der Parameterliste, welche ausdrückt, dass bestimmte Typen aus den Parametern kompatibel sein müssen. Diese sogenannten *sharing constraints* werden Teil der Signatur des Funktors `Combine`.

2.4 Funktionale Instanziierung in ACL2

Der Theorembeweiser ACL2 [12] erlaubt die Definition abstrakter Funktionen, die durch Axiome beschrieben werden. Theoreme über diese Funktionen können dann durch sogenannte *Funktionale Instanziierung* [3] auf andere Funktionen übertragen werden, die die Axiome erfüllen.

Programme werden in ACL2 in einer Teilsprache von Common LISP definiert. Die Interaktion basiert auf sogenannten *Events*. Das sind Eingaben an den Beweiser, die dessen internen Zustand verändern. Das können beispielsweise Definitionen von Funktionen oder Lemmata sein. Events sind ebenfalls in LISP-Schreibweise notiert.

Folgendes Event definiert ein abstraktes Funktionsssymbol `LT`, welches eine kleiner-als-Relation darstellen soll: [8]

```
(CONSTRAIN LT-INTRO (REWRITE)
  (IMPLIES (LT X Y) (NOT (LT Y X)))
  ((LT (LAMBDA (X Y) F))))
```

Das neue Funktionssymbol hat den Namen `LT`. Die zweite Zeile enthält das über `LT` geforderte Axiom, nämlich die Antisymmetrie. Die dritte Zeile enthält eine Liste von Paaren (in diesem Fall einelementig), durch die dem abstrakten Funktionssymbol `LT` eine konkrete Funktion zugeordnet wird. Diese konkrete Funktion (auch *Witness-Implementierung* genannt) dient allerdings nur dazu, die Konsistenz der Axiome sicherzustellen. Außerhalb der Definition ist sie nicht von Interesse. In diesem Beispiel wählt man eine triviale Funktion, die immer `F` (für False) zurückliefert.

Bei der Verarbeitung dieses Events muss zunächst bewiesen werden, dass die Witness-Implementierung das Axiom erfüllt. In diesem Fall ist der Beweis trivial. Damit ist die Konsistenz der Axiomatisierung sichergestellt, und im System ist ab sofort ein neues Funktionssymbol LT verfügbar, über das lediglich das Antisymmetrieaxiom bekannt ist.

Ausgehend von dieser Funktion kann nun ein Prädikat `ORDERED-LT` für eine nach `LT` geordnete Liste und ein Sortieralgorithmus `SORT-LT` zum Sortieren einer Liste definiert werden.

```
(DEFN ORDERED-LT (L)
  (IF (LISTP L)
      (IF (LISTP (CDR L))
          (IF (LT (CADR L) (CAR L))
              F
              (ORDERED-LT (CDR L)))
          T)
       T))

(DEFN INSERT-LT (X L)
  (IF (LISTP L)
      (IF (LT X (CAR L))
          (CONS X L)
          (CONS (CAR L) (INSERT-LT X (CDR L))))
      (LIST X)))
```

[8] Die folgenden Beispiele entstammen [3] und beziehen sich auf Nqthm, den Vorgänger von ACL2. In neueren Versionen hat sich die Syntax der Kommandos geändert, das Prinzip der funktionalen Instanziierung ist jedoch grundsätzlich gleich geblieben.

```
(DEFN SORT-LT (L)
  (IF (LISTP L)
      (INSERT-LT (CAR L) (SORT-LT (CDR L)))
      NIL))

(PROVE-LEMMA ORDERED-SORT-LT (REWRITE)
  (ORDERED-LT (SORT-LT L)))
```

Im Beweis des Lemmas `ORDERED-SORT-LT` wird das über `LT` vorausgesetzte Axiom verwendet, nicht aber die Witness-Implementierung.

Die Funktionen `ORDERED-LT`, `INSERT-LT` und `SORT-LT` sind nicht ausführbar, da `LT` keine Implementierung besitzt. Das bewiesene Lemma `ORDERED-SORT-LT` ist somit ebenfalls abstrakt: Es ist keine Aussage über ein konkretes Programm.

In einem nächsten Schritt kann man das Lemma `ORDERED-SORT-LT` jedoch konkretisieren und damit für die Verifikation eines konkret implementierten Sortieralgorithmus nutzbar machen. Durch funktionale Instanziierung können die Aussagen über `LT` und darauf aufbauende Konstrukte auf die Relation `LESSP`[9] und darauf aufbauende strukturgleiche Konstrukte übertragen werden. Dazu müssen zunächst die Funktionen `ORDERED-LESSP`, `INSERT-LESSP` und `SORT-LESSP` definiert werden. Diese Definitionen entsprechen genau den obigen Definitionen von `ORDERED-LT`, `INSERT-LT` bzw. `SORT-LT`, nur dass jeweils das Symbol `LT` durch `LESSP` ersetzt wird[10].

Das folgende Event erzeugt dann das instanziierte Lemma:

```
(FUNCTIONALLY-INSTANTIATE ORDERED-SORT-LESSP (REWRITE)
  (ORDERED-LESSP (SORT-LESSP L))
  ORDERED-SORT-LT
  ((LT         LESSP)
   (ORDERED-LT ORDERED-LESSP)
   (INSERT-LT  INSERT-LESSP)
   (SORT-LT    SORT-LESSP)))
```

Entscheidendes Element dieses Ausdrucks ist die Liste von Assoziationen von Funktionssymbolen. Dabei handelt es sich um eine *funktionale Substitution*, die die Zuordnung von `LT`-Funktionen zu `LESSP`-Funktionen beschreibt. Das System prüft nun, ob das neu zu beweisende Lemma

```
(ORDERED-LESSP (SORT-LESSP L))
```

tatsächlich eine Instanz des Lemmas `ORDERED-SORT-LT` unter dieser funktionalen Substitution ist. Ist das der Fall, so werden alle über `LT`, `ORDERED-LT` `INSERT-LT` und `SORT-LT` bekannten Axiome mit den neuen Funktionssymbolen

[9] Diese Relation ist in ACL2 genau wie in ✓eriFun vordefiniert und repräsentiert die übliche <-Relation auf natürlichen Zahlen.

[10] Im Prinzip könnten diese Definitionen auch automatisch erzeugt werden. In [3] wird aber argumentiert, dass eine manuelle Neueingabe mit einem guten Editor auch kein großer Aufwand sei.

instanziiert und müssen dann bewiesen werden. Aus dem Axiom über `LT` wird die Antisymmetrie von `LESSP` erzeugt. Da in ACL2 Funktionsdefinitionen auch Axiome sind, werden die Definitionsgleichungen von `ORDERED-LT`, `INSERT-LT` und `SORT-LT` ebenfalls instanziiert und müssen bewiesen werden. Diese Instanzen der Definitionen sind aber exakt gleich mit den Definitionen der Instanzfunktionen, die ja völlig analog definiert wurden. Damit sind diese Beweise trivial.

Sind alle diese Voraussetzungen erfüllt, wird das neue Lemma `ORDERED-SORT-LESSP` erzeugt und als wahr angenommen.

2.5 Zusammenfassung

Bei den Mechanismen, mit denen in Haskell und Standard ML Strukturen und Schnittstellen beschrieben werden, gibt es teilweise erhebliche Unterschiede. Beim System der Typklassen werden Schnittstellenbeschreibung an das Typsystem geknüpft, was in bestimmten Fällen (Bsp. 2.3) sehr gut funktioniert. In Schnittstellen, die keine oder mehrere Typen enthalten, kommt es hier aber zu Zuordnungsproblemen. In ML wird hingegen mit den Signaturen ein eigenes Typsystem für Strukturen eingeführt, welches vom Typsystem der Kernsprache unabhängig ist. So tauchen die Probleme aus Abschnitt 2.2.4 nicht auf. Daher wird sich der Begriff der *Spezifikation*, den wir für ✓`eriFun` entwickeln werden, eher an den Signaturen in ML orientieren.

Die funktionale Instanziierung von ACL2 bietet ein Verfahren, wie abstrakte Ergebnisse auf konkrete Instanzen übertragen werden können. Der Vorteil dieses Verfahrens liegt in seiner leichten Implementierbarkeit, da keine Eingriffe ins Typsystem (durch Untertypen oder ähnliches) notwendig sind. In ✓`eriFun` wird diese Instanziierung von Lemmata allerdings vollautomatisch erfolgen.

Kapitel 3

Module und Sichtbarkeiten

In diesem Kapitel wird ein einfaches Modulsystem für √`eriFun` beschrieben, welches auf dem eingangs beschriebenen Prinzip der Sichtbarkeiten beruht. Dabei wird es dem Benutzer ermöglicht, Sichtbarkeiten der Programmelemente einzuschränken. Auf diese Weise kann der Suchraum für Lemmata eingeschränkt werden, indem man gewisse Hilfslemmata für andere Module unsichtbar macht. Davon kann die Effizienz des symbolischen Auswerters (also des Beweisers) profitieren.

Da sich die neue Funktionalität im wesentlichen auf die Verwaltung der Programmelemente und Sichtbarkeiten beschränkt, gestaltet sich die Implementierung und Integration in das bestehende System vergleichsweise einfach, da hier noch keine Eingriffe in Typsystem, den symbolischen Auswerter etc. erforderlich sind. Ein Prototyp dieser Erweiterung wurde im Rahmen dieser Arbeit implementiert.

3.1 Syntax

Module sind Strukturierungsmittel für $\mathcal{FP}$-Programme. Ein Programm ist in mehrere Module unterteilt. Dabei gibt es zwei besonders ausgezeichnete Module `Program` und `Predefined`, welche die gleichnamigen Ordner[11] in der bisherigen Implementierung ersetzen. Zusätzlich zu diesen vordefinierten Modulen kann der Benutzer neue Module definieren.

Im Gegensatz zu anderen Programmelementen[12] sind Module sogenannte *Container*, die andere Elemente enthalten können. In Kapitel 4 werden wir noch

[11] Ordner (*Folder*) erlauben in der bisherigen Implementierung die Darstellung des Programms in einer Baumstruktur, die allerdings rein optischen Zwecken dient und keine Bedeutung für das Programm hat. Ordner bleiben auch nach der Einführung von Modulen für diesen Zweck verfügbar und können z.B. zur Strukturierung innerhalb eines Moduls verwendet werden.

[12] Programmelement ist die allgemeine Bezeichnung für eine abgeschlossene Definition in √`eriFun`. Bislang gab es drei Arten von Programmelementen: Funktionsdefinitionen, Datenstrukturdefinitionen und Lemmata.

einen weiteren Container, die Spezifikation, definieren. Elemente, die keine Container sind, nennen wir *atomare* Elemente.

Ein neues und zu Anfang leeres Modul wird definiert durch

`module` M

Jedes atomare Programmelement ist einem Container zugeordnet, welchen der Benutzer vor dem Einfügen des Elements mit der Maus ausgewählt hat. Weiterhin wird jedem Element eine der beiden Sichtbarkeiten *public* oder *private* zugeordnet. Damit ist die vollständige Syntax für eine Funktionsdefinition nun folgende (Analoges gilt für Datenstruktur- und Lemmadefinitionen):

$$[\ \mathit{public} \mid \mathit{private}\ ]\ [\mathit{Modul}]\ \texttt{function}\ f(x_1 : T_1, \ldots, x_n : T_n) : T_{n+1} \Leftarrow R_f$$

Um Abwärtskompatibilität zu gewährleisten, sind beide neuen Angaben für die Eingabe optional. Wird keine Sichtbarkeit angegeben, so ist die Sichtbarkeit *public*. Wird kein Container angegeben, so wird das Element in den Container eingefügt, den der Benutzer vor der Eingabe selektiert hatte, bzw. in das Hauptmodul des Programms, falls die Definition aus einer Datei eingelesen wird.

Module sind in ✓`eriFun` nicht hierarchisch gegliedert. Ein Modul kann keine anderen Module enthalten. Für die Darstellung hat der Benutzer dennoch die Möglichkeit, geschachtelte Module zu erzeugen. Dies hat aber keine semantische Konsequenz und dient allein dazu, die Präsentation übersichtlicher zu gestalten.

3.2 Beschränkungen durch Sichtbarkeiten

Zweck der Sichtbarkeit ist es, den Zugriff anderer Module auf ein Programmelement einzuschränken. Damit gilt für die Definition von neuen Elementen folgende Beschränkung:

> Ein Element E_1 darf nur dann in der Definition eines anderen Elements E_2 referenziert werden, wenn entweder E_1 öffentlich ist, oder E_1 und E_2 zum selben Modul gehören.

Diese Einschränkung soll aber nicht nur für die Definition von Elementen gelten, sondern auch für Beweise. In Beweisen ist es aber möglich, Funktionsdefinitionen zu „öffnen“, also einen Funktionsaufruf durch den entsprechend instanziierten Rumpf zu ersetzen. Dabei kann eine Funktion, die nicht in der Definition eines Elements benutzt wird, im Beweis doch auftauchen, da sie über das Öffnen einer zweiten Funktion eingeführt wird. Daraus ergibt sich zwangsläufig folgende weitere Einschränkung:

> Alle Elemente, die in öffentlichen Definitionen verwendet werden, müssen ebenfalls öffentlich sein.

```
module InsertionSort

public InsertionSort function insert(x:nat, l:list[nat]): list[nat] ⇐
  if ?empty(l)
    then add(x,empty)
    else if hd(l)>x
           then add(x,l)
           else add(hd(l),insert(x,tl(l)))
         end_if
  end_if

public InsertionSort function isort(l:list[nat]): list[nat] ⇐
  if ?empty(l)
    then empty
    else insert(hd(l),isort(tl(l)))
  end_if

private InsertionSort lemma insert_keeps_order ⇐ ∀ x:nat, l:list[nat]
  if(ordered(l),ordered(insert(x,l)),true)

public InsertionSort lemma isort_sorts ⇐ ∀ l:list[nat]
  ordered(isort(l))
```

Abbildung 3.1: Das Modul *InsertionSort*

Man beachte, dass dies nur für Referenzen in der *Definition* des Elements gilt. Im Beweis des Elements hingegen dürfen auch private Elemente des gleichen Moduls verwendet werden.

3.3 Beispiele

Beispiel 3.1. In Abbildung 3.1 wird ein Sortieralgorithmus definiert.

Dabei muss die Funktion *insert* öffentlich sein, da sie in *isort* verwendet wird. Ein Element, das *isort* verwendet, muss auch auf *insert* zugreifen dürfen, da die Definition von *isort* geöffnet werden kann. Das Lemma *insert_keeps_order* hingegen kann als privat deklariert werden, da diese Aussage außerhalb des Moduls nicht von Belang ist. Sie wird lediglich zum Beweis des Hauptsatzes *isort_sorts* benötigt.

Die Notwendigkeit, Funktionen, die von öffentlichen Funktionen verwendet werden, ebenfalls als öffentlich zu deklarieren, erscheint zunächst befremdlich, zumal dies bei anderen Programmiersprachen nicht notwendig ist.

Hier wird aber ein wesentlicher Unterschied zwischen Verifikationstools und reinen Programmiersprachen deutlich. Für das bloße Programmieren genügt es, die

Signaturen der Funktionen zu veröffentlichen, denn mehr Informationen werden für das Typechecking nicht benötigt. Programmverifikation geht aber darüber hinaus und beschäftigt sich auch mit semantischen Eigenschaften der Funktionen. Dafür ist es aber notwendig, nicht nur auf die Schnittstellen, sondern auch auf die Implementierung – und damit die verwendeten Funktionen – zuzugreifen.

Ein unangenehmer Nebeneffekt dieser *Transparenz* von Funktionen ist die Tatsache, dass letztlich alle für einen Algorithmus relevanten Funktionen öffentlich sein müssen. Private Funktionen können lediglich dazu verwendet werden, Eigenschaften zu modellieren, die ausschließlich für den Beweis relevant sind. Das können beispielsweise Invarianten oder Eigenschaften von Hilfsfunktionen sein[13].

Beispiel 3.2. In der Fallstudie zum *Binary Search* Algorithmus [24] wird für den Beweis der Vollständgkeit ein Hilfsprädikat *in.partition* benötigt, welches wie folgt definiert ist:

```
function in.partition(key:@ITEM, a:list[@ITEM], i:nat, j:nat):bool ⇐
if ?empty(a)
  then false
  else if ?0(i)
         then if key=hd(a)
                then true
                else if ?0(j)
                       then false
                       else in.partition(key,tl(a),i,pred(j))
                     end_if
              end_if
         else if ?0(j)
                then false
                else if pred(i)>pred(j)
                       then false
                       else in.partition(key,tl(a),pred(i),pred(j))
                     end_if
              end_if
       end_if
end_if
```

Der Hauptsatz über *in.partition* ist folgende Aussage:

```
lemma in.partition_entails_find ⇐ ∀ j,i,key:nat, a:list[nat]
  if(length(a)>j,
    if(ordered(a),if(in.partition(key,a,i,j),find(key,a,i,j),true),true),
    true)
```

Für den Beweis dieser Aussage werden acht Hilfslemmata über *in.partition* benötigt, die im übrigen Beweis nicht von Belang sind. Das System versucht aber dennoch, die Lemmata bei der symbolischen Auswertung anzuwenden, was den Suchraum unnötig vergrößert. Betroffen sind davon im konkreten Beispiel

[13] In der Quicksort-Fallstudie können z.B. die Prädikate für untere und obere Schranken als privat deklariert werden

noch zehn Auswertungen in denen jeweils alle acht Hilfslemmata als anwendbar gelten.

Deklariert man hingegen die Hilfslemmata als privat im Modul *Completeness_of_Binary_Search*, dann werden sie in nachfolgenden Beweisen zur Komplexität des Algorithmus, die in einem anderen Modul stattfinden, nicht mehr verwendet.

Aus einer Einschränkung des Suchraums resultiert im Prinzip auch eine potenzielle Zeitersparnis. Diese konnte allerdings in diesem Beispiel leider nicht konkret nachgewiesen werden, da die gemessenen Auswertungszeiten absolut gesehen bereits sehr kurz waren und starken statistischen Schwankungen unterlagen. Bei der Übertragung alter Fallstudien sollte in Zukunft auf eine geeignete Strukturierung in Module geachtet und untersucht werden, ob damit ein Effizienzgewinn messbar ist.

3.4 Änderungen an der Benutzerschnittstelle

Die Benutzeroberfläche von ✓`eriFun` erlaubt es, gewisse Änderungen am Programm vorzunehmen, die – würde man sie direkt im Quelltext ausführen – nur mit viel Aufwand durchführbar wären. Das wird dadurch ermöglicht, dass das System nach Eingabe und Parsing nicht mehr auf dem Quellcode des Programms operiert, sondern auf einer internen Repräsentation, die nur zur Ausgabe wieder in einen formatierten Quellcode zurückverwandelt wird.

So ist beispielsweise das globale Umbenennen einer Funktion möglich, ohne dass alle Referenzen auf die Funktion manuell geändert werden müssen. Auch können jederzeit Elemente zwischen Ordnern verschoben werden.

Als konsequente Fortführung dieses Konzepts wird es dem Benutzer erlaubt, die Sichtbarkeit eines Elements nachträglich zu ändern und Elemente zwischen Modulen zu verschieben. Diese spezielle Funktionalität ist sinnvoll, da ein Löschen und anschließendes Einfügen des veränderten Elements meist aufgrund von Abhängigkeiten nicht möglich ist.

Vor dem Ändern der Sichtbarkeit eines Elements muss überprüft werden, ob durch die neue Sichtbarkeit evtl. Beschränkungen verletzt werden. So kann ein Element E nur dann auf private Sichtbarkeit umgestellt werden, wenn kein Element eines anderen Moduls direkt von E abhängig ist und keine öffentliche Funktion aus dem gleichen Modul E verwendet. Ein Element kann von privater auf öffentliche Sichtbarkeit umgestellt werden, wenn es kein privates Element in seiner Definition verwendet.

Private Elemente können grundsätzlich nicht in andere Module verschoben werden. Ein öffentliches Element kann in ein anderes Modul verschoben werden, wenn es weder in seiner Definition noch in seinem Beweis ein privates Element verwendet.

Kapitel 4

Axiomatische Spezifikationen

Dieses Kapitel beschreibt ein Konzept zur Unterstützung axiomatischer Spezifikationen in ✓eriFun. Dabei wird teilweise auf Konzepte aus Kapitel 2 aufgebaut.

4.1 Motivation

Ein Wesenszug axiomatisch definierter Strukturen ist, dass sie zu komplexeren Strukturen zusammengesetzt werden können. Definitionen wie die folgende sind in der Algebra typisch:

Definition. *Seien G und G' zwei Gruppen. Eine Abbildung $h : G \mapsto G'$ heißt* Gruppenhomomorphismus*, falls gilt*

$$h(x)h(y) = h(xy) \quad \forall x, y \in G.$$

Diese Definition greift gleich zweimal auf die Spezifikation einer Gruppe zurück. Es werden zwei Strukturen eingeführt, von denen jede die Axiome einer Gruppe erfüllt. Obige Definition enthält aber viel implizite Notation, die man zum besseren Verständnis zunächst sichtbar machen sollte:

Definition. *Seien $(G, \cdot, e, inv)$ und $(G', \odot, e', inv')$ zwei Gruppen. Eine Abbildung $h : G \mapsto G'$ heißt* Gruppenhomomorphismus*, falls gilt*

$$h(x) \odot h(y) = h(x \cdot y) \quad \forall x, y \in G.$$

Entscheidend ist hier der erste Satz, der die beiden Gruppen einführt und ohne den der Rest der Definition keinen Sinn ergäbe. Der Satz beschreibt, was für Strukturen für die Definition angenommen werden und benennt deren Elemente.

Wir erfahren also einerseits, *welche Symbole* es gibt und *welchen Typ* diese haben (so hat die Operation $\cdot$ beispielsweise den Typ $G \times G \to G$). Andererseits wissen wir auch, dass für jede der Strukturen die Axiome einer Gruppe gelten. Damit

gibt es in diesem Zusammenhang das Axiom des rechtsneutralen Elements in zwei Formen:

$$x \cdot e = x \quad \forall x \in G$$

$$x \odot e' = x \quad \forall x \in G'$$

Das Ziel dieses Kapitels ist es, Formulierungen dieser Art in ✓eriFun zu ermöglichen. Die neu eingeführte Syntax hält sich dabei so eng wie möglich an die mathematisch übliche Form.

4.2 Einführende Beispiele

Bevor wir den Gruppenhomomorphismus modellieren, soll die neue Notation an einem bereits bekannten Beispiel dargestellt werden. Dazu betrachten wir die Spezifikation eines Monoids:

```
specification Monoid
  domain @M
  operator op: @M,@M -> @M
  operator neut: @M
  axiom op_associative ⇐ ∀ x,y,z:@M op(x,op(y,z))=op(op(x,y),z)
  axiom neut_left_neutral ⇐ ∀ x:@M op(neut,x)=x
  axiom neut_right_neutral ⇐ ∀ x:@M op(x,neut)=x
```

Wie Module sind Spezifikationen in ✓eriFun Container, die andere Programmelemente enthalten. Anstelle von Funktionsprozeduren, Datenstrukturen und Lemmata enthalten Spezifikationen aber Elemente vom Typ `domain`, `operator` und `axiom`. Diese beschreiben zusammen ein gewisses Verhalten, aber keine konkrete Implementierung. Wie in Abschnitt 1.3 erläutert, kann eine Spezifikation als formale Beschreibung einer Schnittstelle aufgefasst werden. Spezifikationen enthalten ausschließlich Elemente vom Typ `domain`, `operator` und `axiom`. (Ausnahme: Siehe Abschnitt 4.3.6)

4.2.1 Strukturparameter

Im Gegensatz zu Elementen von Modulen sind Elemente in Spezifikationen stets lokal. Sie sind zunächst außerhalb der Spezifikation nicht sichtbar. Um die Monoidspezifikation in einer anderen Definitionen verwenden zu können, muss sie dort zunächst explizit importiert werden:

Beispiel 4.1. Die Funktion *fold*, die Listen mit Elementen aus einem Monoid zusammenfaltet, kann folgendermaßen definiert werden:

```
function fold[M:Monoid(@M,op,neut)](l:list[@M]):@M ⇐
  if ?empty(l)
    then neut
    else op(hd(l),fold[M](tl(l)))
  end_if
```

Die Prosa-Formulierung „Sei $M = (@M, op, neut)$ ein Monoid“ wird in den Ausdruck *M:Monoid(@M,op,neut)* in eckigen Klammern hinter dem Funktionssymbol übersetzt. Dieser Ausdruck ist ein sogenannter *Strukturparameter*: Die Funktion *fold* ist durch ein Monoid M parametrisiert. Zum Monoid gehören der Typ $@M$ und die Operatoren *op* und *neut*. Dabei sind M, $@M$, *op* und *neut* lokale Namen, die nur innerhalb der nachfolgenden Definition gültig sind. Diese Namen lauten nur zufällig genauso wie in der Spezifikation.

Beispiel 4.2. Die Definition aus Beispiel 4.1 könnte ebenso folgendermaßen lauten:

```
function fold[K:Monoid(@W, concat, nil)](l:list[@W]): @W ⇐
  if ?empty(l)
    then nil
    else concat(hd(l),fold[K](tl(l)))
  end_if
```

Die Typvariablen $@M$ (bzw. $@W$) sind hier nicht wie sonst als allquantifiziert zu lesen, sondern werden fixiert, wenn sie in einem Strukturparameter vorkommen. Damit ist die Funktion *fold* nicht polymorph für Listen beliebigen Typs definiert, sondern (wenn man ein festes Monoid wählt) genau über Listen des festen, aber unbekannten Typs $@M$.

Die Zuordnung der neuen Namen zu den Symbolen in der Spezifikation erfolgt über die Reihenfolge der Aufschreibung. Jeder Spezifikation ist das Tupel der Typ- und Funktionssymbole zugeordnet, die in ihr definiert werden. Dieses Tupel nennt man die *Signatur* einer Spezifikation. Die Monoidspezifikation hat beispielsweise die Signatur $(@M, op, neut)$.

Nicht nur Funktionen können auf diese Weise parametrisiert sein, sondern auch Lemmata, Spezifikationen und Instanzen. Später werden dazu jeweils Beispiele betrachtet.

Bei jedem Aufruf von *fold* muss nun stets eine Monoidstruktur übergeben werden, auf der die Funktion operieren soll. Die Funktion kann also nur aus einer Umgebung aufgerufen werden, in der ein Monoid existiert.

Beispiel 4.3. Das folgende Lemma *fold_append* wird also ebenfalls parametrisiert (als Übersetzung der Schreibweise „Sei M ein Monoid. Dann gilt...“):

```
lemma fold_append[M:Monoid] ⇐ ∀ k,l:list[@M]
  fold[M](append(k,l))=op(fold[M](k),fold[M](l))
```

Hier wird eine Kurzschreibweise verwendet: Nach dem Namen der Spezifikation kann die Symbolliste ganz oder teilweise weggelassen werden. Die fehlenden Symbole werden dann als gleichnamig mit der Spezifikation angenommen, also in diesem Fall als $@M$, *op* und *neut*.

Für den Beweis des Lemmas stehen nun die Axiome des Monoids zur Verfügung, die mit den entsprechenden lokalen Namen instanziiert werden.

4.2.2 Mehrfachverwendung

Durch die lokalen Namen ist es auch möglich, dieselbe Spezifikation mehrfach in einer Umgebung zu verwenden, ohne dass es zu Mehrdeutigkeiten kommt. Diese Art von Mehrfachverwendung funktioniert ähnlich wie beim Konzept der Typklassen.

Beispiel 4.4. Folgendes Beispiel definiert die lexikographische Kombination zweier Ordnungen (vgl. Bsp. 2.2). Man nimmt an, dass eine Spezifikation *Ordering* existiert, die einen Typ und eine zweistellige Relation definiert:

```
function gt_lex[O1:Ordering(@O1,gt1),O2:Ordering(@O2,gt2)]
                (x,y:pair[@O1,@O2]): bool ⇐
if fst(x)=fst(y)
  then gt2(snd(x),snd(y))
  else gt1(fst(x),fst(y))
end_if
```

Dabei ist die Datenstruktur *pair* definiert als

```
structure pair[@X,@Y] ⇐
  mkpair(fst:@X,snd:@Y)
```

Der zweimalige Import der Spezifikation *Ordering* erzeugt hier die lokalen Typvariablen $@O1$ und $@O2$, sowie die Funktionen

$$gt1 : @O1 \times @O1 \to bool$$

$$gt2 : @O2 \times @O2 \to bool$$

Damit sind die beiden Ordnungen in der Definition sauber getrennt, und es kann nicht zu Mehrdeutigkeiten kommen.

Man beachte, dass im Gegensatz zum Typklassenkonzept hier nicht nur die Typen, sondern auch die Operatoren neue Namen erhalten. Dadurch ist es nicht notwendig, aus dem Kontext zu entscheiden, welche Operation jeweils gemeint ist, sondern dies wird durch die unterschiedliche Benennung explizit angegeben: Im Gegensatz zu Bsp. 2.2 gibt es hier zwei Größer-als-Funktionen, eine für jede Ordnung, anstelle einer einzigen Funktion, die durch Overloading beide Ordnungen repräsentiert.

Diese saubere Trennung erlaubt nun auch Modellierungen, die bei Typklassen nicht ohne weiteres möglich sind, wie z.B. die in Abschnitt 2.2.4 beschriebene Sortierspezifikation, die in der neuen Syntax wie folgt lautet:

```
specification Sorting
  operator sort: list[nat] -> list[nat]

  axiom sort_sorts ⇐ ∀ l:list[nat]
    ordered(sort(l))

  axiom sort_permutes ⇐ ∀ l:list[nat], n:nat
    occurs(n,l)=occurs(n,sort(l))
```

Folgendes Lemma macht nun eine Aussage über zwei verschiedene, aber nicht näher bekannte Sortieralgorithmen:

```
lemma sort_unique[Sorting(sort1),Sorting(sort2)] ⇐ ∀ l:list[nat]
  sort1(l)=sort2(l)
```

Dieses Beispiel illustriert, dass die Neubenennung der Funktionssymbole in Strukturparametern entscheidend für die Eindeutigkeit ist.

4.2.3 Vererbung

Eine Spezifikation kann von anderen Spezifikationen erben. Dies geschieht ebenfalls durch die Angabe von Strukturparametern:

```
specification Group[M:Monoid(@G,op,neut)]
  operator inv: @G->@G
  axiom inv_left_inverse ⇐ ∀ x:@G op(inv(x),x)=neut
  axiom inv_right_inverse ⇐ ∀ x:@G op(x,inv(x))=neut
```

Die entstehende Spezifikation hat die Signatur $(@G, op, neut, inv)$. Die ererbten Symbole können also auch hier umbenannt werden. Sie stehen in der Signatur vor den neu eingeführten Symbolen.

Anders als z.B. in der objektorientierten Programmierung ist die Vererbung in ✓eriFun konservativ: Es ist sichergestellt, dass die erbende Spezifikation alle Elemente der vererbenden Spezifikation ebenfalls in der selben Form enthält, möglicherweise mit anderen Namen. Ein „Überschreiben“ von Eigenschaften ist nicht möglich.

Obwohl die Syntax dieselbe ist wie bei Funktionen und Lemmata, trifft der Begriff *Vererbung* die Sache hier besser als *Parametrisierung*. Denn es ist nicht so, dass für jedes Monoid M eine Spezifikation *Group[M]* existierte. Es wird schlicht eine Spezifikation *Group* definiert, die ein Monoid *umfasst*, also alle Elemente eines Monoids ebenfalls enthält.

Als einfaches Lemma über Gruppen betrachten wir die Rechtskürzungsregel:

```
lemma right_cancellation[G:Group] ⇐ ∀ x,y,z:@G
  if(op(x,z)=op(y,z),x=y,true)
```

Das Lemma *right_cancellation* ist zwar eine Aussage über Gruppen, aber gehört nicht zur Definition einer Gruppe und ist daher nicht Teil der Spezifikation (genausowenig wie das Lemma *fold_append* Teil der Monoidspezifikation ist). Als Darstellungsoption kann das Lemma aber in die Spezifikation verschoben werden (siehe Abschnitt 4.3.6).

Beispiel 4.5. Durch mehrfaches Erben von einer Gruppe kann nun z.B. der Gruppenhomomorphismus aus Abschnitt 4.2 in ✓eriFun definiert werden:

```
specification GroupHomomorphism[G1:Group(@G1,op1,neut1,inv1),
                                G2:Group(@G2,op2,neut2,inv2)]
  operator h: @G1 -> @G2
  axiom homomorphism ⇐ ∀ x,y:@G1 op2(h(x),h(y))=h(op1(x,y))
```

Die resultierende Spezifikation hat die Signatur

$$(@G1, @G2, op1, neut1, inv1, op2, neut2, inv2, h).$$

Gruppenhomomorphismen bilden immer neutrale Elemente auf neutrale Elemente ab, und Inverse auf Inverse. Diese Aussagen können als Lemmata formuliert werden:

```
lemma h_keeps_neut[H:GroupHomomorphism] ⇐
  h(neut1)=neut2

lemma h_keeps_inv[H:GroupHomomorphism] ⇐ ∀ x:@G1
  h(inv1(x))=inv2(h(x))
```

Der Beweis der Lemmata gelingt mit Hilfe der Kürzungsregel für Gruppen.

Aus Gruppen und Monoiden lassen sich durch fortgesetzte Vererbung Ringe und Körper definieren. Wiederum darauf aufbauend erhält man die Spezifikation eines Vektorraums:

```
specification RingUnit[G:Group(@R,plus,zero,inv),M:Monoid(@R,mult,one)]
  axiom plus_commutativity ⇐ ∀ x,y:@R
    plus(x,y)=plus(y,x)
  axiom left_distributivity ⇐ ∀ x,y,z:@R
    mult(x,plus(y,z))=plus(mult(x,y),mult(x,z))
  axiom right_distributivity ⇐ ∀ x,y,z:@R
    mult(plus(y,z),x)=plus(mult(y,x),mult(z,x))

specification Field[R:RingUnit(@F)]
  operator inv_mult: @F -> @F
  axiom mult_commutative ⇐ ∀ x,y:@F
    mult(x,y)=mult(y,x)
  axiom inv_mult_inverse ⇐ ∀ x:@F
    mult(x,inv(x))=one

specification VectorSpace[F:Field(@S), M:Monoid(@V,v.plus,null)]
  signature s.mult: @S,@V -> @V
  axiom v.plus_comm ⇐ ∀ x,y:@V
    v.plus(x,y)=v.plus(y,x)
  axiom s.mult_zero ⇐ ∀ x:@V
    s.mult(zero,x)=null
```

4.2.4 Instanzen

Durch eine Instanz wird die Behauptung, dass bestimmte Symbole die Axiome einer Spezifikation erfüllen, ins System eingeführt. Diese Aussage muss dann vom System bewiesen werden. Damit bilden Instanzen ein Bindeglied zwischen den abstrakten Spezifikationen und konkreten Implementierungen. Die Syntax für Instanzdefinitionen lehnt sich an die Schreibweise für Strukturparameter an, nur dass hier konkrete Typ- und Funktionssymbole anstelle von „frischen“ Namen verwendet werden.

Beispiel 4.6. Es wird eine Instanz der Spezifikation *Monoid* definiert:

```
instance Plus ⇐
  Monoid(nat,plus,0)
```

Nach Eingabe einer Instanz werden vom System die entsprechenden Beweisverpflichtungen generiert, also in diesem Fall:

$$\forall x, y, z : nat \quad plus(x, plus(y, z)) = plus(plus(x, y), z)$$

$$\forall x : nat \quad plus(0, x) = x$$

$$\forall x : nat \quad plus(x, 0) = x$$

Verifizierte Instanzen können als Strukturparameter übergeben werden. So kann die Funktion *fold* aus Bsp. 4.1 durch die Instanz *Plus* parametrisiert werden. Die entstehende Funktion *fold[Plus]* operiert dann auf Listen von natürlichen Zahlen und berechnet die Summe über eine Liste. Aussagen, die bereits über das allgemeine *fold* bewiesen wurden, stehen automatisch für die Instanz zur Verfügung. Man erhält also über die Funktion *fold[Plus]* automatisch folgende Aussage, falls man bereits das Lemma *fold_append* bewiesen hat:

```
lemma fold_append[Plus] ⇐ ∀ k,l:list[nat]
  fold[Plus](append(k,l))=plus(fold[Plus](k),fold[Plus](l))
```

Instanzen können ebenfalls Strukturparameter besitzen, wie das folgende Beispiel zeigt:

Beispiel 4.7. Wir definieren eine Instanz für das karthesische Produkt aus zwei Monoiden, welches wiederum ein Monoid ist. Dafür werden zunächst die zwei parametrisierten Funktionen *prod_neut* und *prod_op* definiert:

```
function prod_neut[M1:Monoid(@M1, op1, neut1),
                   M2:Monoid(@M2, op2, neut2)]:pair[@M1,@M2] ⇐
  mkpair(neut1,neut2)

function prod_op[M1:Monoid(@M1, op1, neut1),
                 M2:Monoid(@M2, op2, neut2)]
                (x,y:pair[@M1,@M2]):pair[@M1,@M2] ⇐
  mkpair(op1(fst(x),fst(y)),op2(snd(x),snd(y)))

instance ProductMonoid[M1:Monoid(@M1, op1, neut1),
                       M2:Monoid(@M2, op2, neut2)] ⇐
  Monoid(pair[@M1,@M2], prod_op[M1,M2], prod_neut[M1,M2])
```

Man beachte, dass in der Instanzdefinition Parameter für die Funktionen *prod_op* und *prod_neut* angegeben werden müssen.

Die Beweisverpflichtungen, die sich aus der Definition ergeben, haben dieselben Parameter wie die Instanz. Beim Beweis können dann die Axiome, die sich aus diesen Parametern ergeben, verwendet werden:

```
lemma op_associative[ProductMonoid][M1:Monoid(@M1,op1,neut1),
                                    M2:Monoid(@M2,op2,neut2)]
  ⇐ ∀ x,y,z:pair[@M1,@M2]
 prod_op[M1,M2](x,prod_op[M1,M2](y,z))
  =prod_op[M1,M2](prod_op[M1,M2](x,y),z)

lemma neut_left_neutral[ProductMonoid][M1:Monoid(@M1,op1,neut1),
                                       M2:Monoid(@M2,op2,neut2)]
  ⇐ ∀ x:pair[@M1,@M2]
 prod_op[M1,M2](prod_neut[M1,M2],x)=x

lemma neut_right_neutral[ProductMonoid][M1:Monoid(@M1,op1,neut1),
                                        M2:Monoid(@M2,op2,neut2)]
  ⇐ ∀ x:pair[@M1,@M2]
 prod_op[M1,M2](x,prod_neut[M1,M2])=x
```

Parametrisierte Instanzen kann man als *Funktoren* ansehen, also Abbildungen von Strukturen auf Strukturen: Für zwei beliebige Monoide $M1$ und $M2$ ist *ProductMonoid[M1,M2]* wieder ein Monoid. Damit sind parametrisierte Instanzen das Äquivalent zu den in Abschnitt 2.3.4 beschriebenen Funktoren in Standard ML.

4.3 Syntax

In diesem Abschnitt werden die neuen Syntaxelemente im Detail beschrieben.

4.3.1 Notation und grundlegende Definitionen

Im folgenden bezeichnet P immer ein Programm der Sprache $\mathcal{FP}$. Für eine Menge T von Typvariablen und eine Menge TO von Typoperatorvariablen bezeichnet $\textsc{Type}(P, T, TO)$ die Menge aller Typausdrücke, die über Typoperatoren aus P und freien Typ- und Typoperatorvariablen aus T bzw. TO gebildet werden. Dabei können andere Typvariablen vorkommen, die dann als allquantifiziert angesehen werden.[14]

Die Signatur von P (bezeichnet mit $\Sigma(P)$) ist die Menge aller durch das Programm gegebenen Funktionen (Konstrukturen, Selektoren, if, eq, benutzerdefinierte Funktionen, etc.). Dabei ist jedem Funktionssymbol ein Typ (bzw. eine Signatur) $\tau = (\tau_1, \ldots, \tau_{n+1}) \in \textsc{Type}(P, \emptyset, \emptyset)^{n+1}$ zugeordnet, und man schreibt $f : \tau_1 \times \ldots \times t_n \rightarrow \tau_{n+1}$, wenn f den Typ τ hat. Mit TO_P bezeichnen wir die Menge der im Programm P definierten Typoperatoren.

Durch die neue Möglichkeit, Spezifikationen zu importieren, werden lokale Typ- und Funktionssymbole eigeführt, was wir durch lokale Signaturen modellieren, die die globale Signatur $\Sigma(P)$ erweitern.

[14] Die genaue formale Definition der Menge aller Typen ist nicht Teil dieser Arbeit und für diese Betrachtungen auch nicht relevant.

Definition 4.8 (Lokale Signatur). *Eine lokale Signatur ist ein Tripel* $\Sigma = (T, TO, OP)$*. Dabei ist* T *eine endliche Menge von Typvariablen,* TO *eine endliche Menge von Typoperatorvariablen und* OP *eine endliche Menge von Funktionssymbolen, deren Typ aus* $\textsc{Type}(P, T, TO)$ *ist.*

Ersetzungen von Typ- und Funktionssymbolen können durch funktionale Substitutionen beschrieben werden.

Definition 4.9 (Erweiterte Typsubstitution). *Seien* $\Sigma_1 = (T_1, TO_1, OP_1)$ *und* $\Sigma_2 = (T_2, TO_2, OP_2)$ *lokale Signaturen. Eine* erweiterte Typsubstitution $\sigma : \Sigma_1 \to \Sigma_2$ *ist ein Paar* (σ_T, σ_{TO})*, bestehend aus einer Typsubstitution* $\sigma_T : T_1 \to \textsc{Type}(P, T_2, TO_2)$ *und einer Abbildung* $\sigma_{TO} : TO_1 \to TO_P \cup TO_2$*, welche die Typoperatorvariablen auf Typoperatoren gleicher Stelligkeit abbildet. Die Anwendung einer erweiterten Typsubstitution ersetzt Typvariablen und Typoperatorvariablen in einem Typausdruck* τ *und wird mit* $\sigma(\tau)$ *bezeichnet.*

Definition 4.10 (Funktionale Substitution). *Seien* $\Sigma_1 = (T_1, TO_1, OP_1)$ *und* $\Sigma_2 = (T_2, TO_2, OP_2)$ *lokale Signaturen. Eine* funktionale Substitution $\sigma : \Sigma_1 \mapsto \Sigma_2$ *ist ein Paar* (σ_T, σ_{OP})*, so dass* $\sigma_T : \Sigma_1 \to \Sigma_2$ *eine erweiterte Typsubstitution und* $\sigma_{OP} : OP_1 \mapsto \Sigma(P) \cup OP_2$ *eine partielle Abbildung von Funktionssymbolen ist, für die gilt:*

$$f : \tau \quad \Rightarrow \quad \sigma_{OP}(f) : \sigma_T(\tau) \quad \forall f \in Dom(\sigma_{OP})$$

Zwei funktionale Substitutionen σ_1 *und* σ_2 *sind* kompatibel*, wenn sie keine unterschiedlichen Abbildungen für dasselbe Symbol beinhalten. In diesem Fall ist die Vereinigung* $\sigma_1 \cup \sigma_2$ *auf natürliche Art und Weise definiert.*

Funktionale Substitutionen werden im folgenden immer in der Paarschreibweise angegeben, wobei die Typ- und Funktionssymbolersetzungen zusammengefasst werden: $\{@T_1/T_1', \ldots, @T_n/T_n', f_1/f_1', \ldots, f_m/f_m'\}$

Die Anwendung einer funktionalen Substitution σ auf einen Typausdruck τ, einen Term t und eine Formel ϕ ist auf bekannte Weise induktiv über der Struktur der jeweiligen Objekte definiert und wird mit $\sigma(\tau)$, $\sigma(t)$ bzw. $\sigma(\phi)$ bezeichnet. Per Induktion zeigt man, dass funktionale Substitutionen auf Termen typerhaltend modulo der erweiterten Typsubstitution sind, dass also $t : \tau \Rightarrow \sigma(t) : \sigma(\tau)$ gilt.

4.3.2 Spezifikationen und neue Programmelemente

Spezifikationen sind, wie im letzten Kapitel erläutert, Container für Elemente. Die in einer Spezifikation definierten Elemente sind aber nicht mehr Teil des globalen Namensraums. Auf sie kann man nur Bezug nehmen, indem man vorher die Spezifikation als Parameter aufführt.

Da Elemente in Spezifikationen keine globale Gültigkeit haben, dürfen verschiedene Spezifikationen Elemente gleichen Namens enthalten. Ähnlich wie bisher schon bei lokalen Variablen (wie sie z.B. als formale Parameter in Funktionsdefinitionen vorkommen) darf es aber keine Überschneidungen mit globalen Namen (etwa von Funktionen oder Datenstrukturen) geben.

Spezifikationen

Spezifikationen ähneln den im vorigen Kapitel definierten Modulen und sind auch in der konkreten Syntax an diese angelehnt. Zusätzlich können Spezifikationen Parameter haben, deren Syntax später konkretisiert wird:

`specification` $S[\langle Parameter \rangle]$

Die Zugehörigkeit eines Elements zu einer Spezifikation wird wie bei Modulen durch Angabe des Namens der Spezifikation vor dem Schlüsselwort ausgedrückt. Im Gegensatz zu Modulen kennen Spezifikationen aber keine Sichtbarkeiten.

Spezifikationen können ausschließlich die folgenden drei Elementtypen enthalten:

Typdeklarationen

Typdeklarationen definieren ein neues abstraktes Typsymbol, welches einen festen aber ansonsten unbekannten Typ repräsentiert. Eine Typdeklaration hat folgende Form:

$\langle Spezifikation \rangle$ `domain` $@T$

Anstatt Typen können auch Typoperatoren deklariert werden. Das ist z.B. nützlich, wenn man Datenstrukturen wie Listen oder Bäume spezifizieren will. Ein Typoperator wird wie folgt deklariert:

$\langle Spezifikation \rangle$ `domain` $T[@X_1, \ldots, @X_n]$

Hierbei wird lediglich das führende Symbol T als neuer n-stelliger Typoperator eingeführt. Die Typvariablen $@X_i$, die für die Parameter stehen, dienen lediglich dazu, die Stelligkeit des Typoperators zu bestimmen. Ob es sich um bereits definierte Typvariablen handelt oder nicht, ist dabei unerheblich.

Operatordeklarationen

Operatordeklarationen definieren ein neues abstraktes Funktionssymbol, welches für eine unbekannte totale Funktion steht:

$\langle Spezifikation \rangle$ `operator` $s: T_1, \ldots, T_n \; -> \; T_{n+1}$

In den Typen $T_1, \ldots, T_{n+1}$ dürfen neben den globalen Typoperatoren auch die Typsymbole verwendet werden, die in Typdeklarationen derselben Spezifikation vorkommen. Alle anderen Typvariablen werden wie bei Funktionsdefinitionen als allquantifiziert angenommen.

Axiome

Axiome unterscheiden sich lediglich durch das verwendete Schlüsselwort von Lemmata:

$\langle$*Spezifikation*$\rangle$ `axiom` $a \Leftarrow \forall\, x_1 : T_1, \ldots, x_n : T_n \quad t_{bool}$

Dabei dürfen in $T_1, \ldots, T_n$ und t_{bool} die in der Spezifikation definierten Typ- bzw. Funktionssymbole verwendet werden.

Axiome repräsentieren Aussagen, die als gültig angenommen werden und werden im Gegensatz zu Lemmata nicht bewiesen.

Signatur einer Spezifikation

Jeder Spezifikation ist eine Signatur zugeordnet. Das ist das Tupel der Typ- und Funktionssymbole, die in der Spezifikation definiert sind. Dabei stehen in der Signatur stets zuerst die Typen, dann die Typoperatoren und schließlich die Funktionssymbole jeweils in der Reihenfolge, in der sie definiert wurden. Geerbte Elemente stehen in der jeweiligen Gruppe immer vor den neu definierten Elementen. Diese Festlegung ist willkürlich und dient nur dazu, eine eindeutige Reihenfolge zu haben, um die Umbenennungen kürzer schreiben zu können. Die Axiome sind nicht Teil der Signatur.

Die Schreibweise von Spezifikationen und deren Inhalt als verschiedene, voneinander getrennte Definitionen hat ihre Wurzeln in der internen Repräsentation von Spezifikationen in ✓`eriFun` als Sammlung von Elementen. Übersichtlicher ist jedoch die Blockschreibweise, bei der eine Spezifikation stets zusammen mit allen ihren Elementen angegeben wird. Diese Schreibweise wird auch in den folgenden Beispielen verwendet:

`specification` $S[\langle Parameter \rangle]$
`domain` $@T_1$
$\vdots$
`domain` $@T_n$
`domain` $OP_1[\ldots]$
$\vdots$
`domain` $OP_k[\ldots]$
`operator` $s_1 : T_{1,1}, \ldots, T_{1,n_1} \rightarrow T_{1,n_1+1}$
$\vdots$
`operator` $s_m : T_{m,1}, \ldots, T_{m,n_m} \rightarrow T_{m,n_m+1}$
`axiom` $ax_1 \Leftarrow \forall\, x_1 : T'_{1,1}, \ldots, x_n : T'_{1,k_1} \quad t_{1,bool}$
$\vdots$
`axiom` $ax_\ell \Leftarrow \forall\, x_1 : T'_{\ell,1}, \ldots, x_n : T'_{\ell,k_\ell} \quad t_{\ell,bool}$

Die lokale Signatur Σ_S der oben angegebenen Spezifikation S ist definiert als

$$\Sigma_S = (\{@T_1, \ldots, @T_n\}, \{OP_1, \ldots, OP_k\}, \{s_1, \ldots, s_m\})$$

mit $s_i : T_{i,1}, \ldots, T_{i,n_i} \rightarrow T_{i,n_i+1}$.

Instanzen

Instanzdefinitionen für eine Spezifikation S mit der Signatur $(@T_1, \ldots, @T_n, OP_1, \ldots, OP_k, s_1, \ldots, s_m)$ haben die allgemeine Form

`instance` $I[\langle Parameter \rangle] \Leftarrow$
$S(\tau_1, \ldots, \tau_n, OP'_1, \ldots, OP'_k, f_1, \ldots f_m)$

Dabei ist I ein neuer Bezeichner, S der Name einer Spezifikation, $\tau_1, \ldots, \tau_n$ sind Typausdrücke, $OP'_1, \ldots, OP'_k$ Typoperatoren aus dem Programm und $f_1, \ldots, f_m$ Funktionssymbole. Dabei dürfen Typ- und Funktionssymbole aus den Parametern verwendet werden. Falls eine Funktion f_i wiederum parametrisiert ist, so müssen geeignete Parameter angegeben werden (vgl. Beispiel 4.7). Aus einer Instanzdefinition ergibt sich eine erweiterte Typsubstitution

$$\theta = \{@T_1/\tau_1, \ldots @T_n/\tau_n, OP_1/OP'_1, \ldots, OP_k/OP'_k\}.$$

Für ein Funktionssymbol $s_i : \tau$ aus der Spezifikation muss f_i den Typ $\theta(\tau)$ haben. Damit beschreibt jede Instanzdefinition I für S eine funktionale Substitution $\sigma_I : \Sigma_S \mapsto \Sigma_I$ mit

$$\sigma_I := \{@T_1/\tau_1, \ldots @T_n/\tau_n, OP_1/OP'_1, \ldots, OP_k/OP'_k, s_1/f_1, \ldots, s_m/f_m\}.$$

4.3.3 Strukturparameter

Wie die Beispiele schon gezeigt haben, besteht die entscheidende Neuerung dieser Spracherweiterung darin, dass Programmelemente durch Strukturen parametrisiert werden können.

Sei S eine Spezifikation mit der Signatur

$$(@T_1, \ldots, @T_n, OP_1, \ldots, OP_k, s_1, \ldots, s_m).$$

Ein Strukturparameter für S hat die folgende Syntax:

$$N : S(@T'_1, \ldots, @T'_n, OP'_1, \ldots, OP'_k, s'_1, \ldots, s'_m)$$

Dabei ist N ein neuer lokaler Bezeichner. Die Typvariablen $@T'_i$ sind lokale Typvariablen, die für die Typen aus der Spezifikation S stehen. Die Bezeichner OP'_i und s'_i sind lokale Typoperator- bzw. Funktionssymbole, die für die Typoperatoren bzw. Funktionen aus der Spezifikation stehen. Bei der in Klammern angegebenen Liste von Bezeichnern können vom Ende her Einträge weggelassen werden. Diese werden dann implizit als gleichlautend mit den Namen in der Signatur von S angenommen.

Die Angabe konkreter Typen oder Funktionen in Strukturparametern ist nicht möglich, da dann ja ein Beweis erforderlich wäre, dass die Axiome tatsächlich erfüllt sind. Dies ist nur durch Instanzdefinitionen möglich.

Ein Strukturparameter macht die angegebenen Typ- und Funktionssymbole als lokale Symbole für die folgende Definition verfügbar. Aus der Umbenennung der Typsymbole ergibt sich die erweiterte Typsubstitution

$$\theta = \{@T_1/@T'_1, \ldots, @T_n/@T'_n, OP_1/OP'_1, \ldots, OP_k/OP'_k\}.$$

Für ein Funktionssymbol $s_i : \tau$ aus der Spezifikation hat die umbenannte Funktion s'_i den Typ $\theta(\tau)$. Der Bezeichner N steht für die gesamte Struktur. Er wird benötigt, um die gesamte Struktur wiederum als Parameter übergeben zu können.

Eine Liste $\mathcal{U} = \langle N_1, \ldots, N_k \rangle$ von Strukturparametern bezeichnen wir auch als *Umgebung*. Jedem parametrisierten Element (also Funktionen, Lemmata, Spezifikationen und Instanzen) ist eine solche Umgebung zugeordnet.

Für einen Strukturparameter N ergibt sich die lokale Signatur Σ_N, die genau die durch den Parameter definierten lokalen Typ- und Funktionssymbole enthält. Die lokale Signatur der Umgebung $\Sigma_\mathcal{U}$ ist die Vereinigung aller Σ_{N_i}.

4.3.4 Strukturterme

Strukturen, die als Parameter in den eckigen Klammern übergeben werden können, werden durch *Strukturterme* beschrieben. Diese Parameter müssen bei Funktionsaufrufen angegeben werden. Ebenso werden sie verwendet, um instanziierte Lemmata zu benennen. So entsteht z.B. aus dem Lemma *fold_append* das Lemma *fold_append[Plus]*.

In der Analogie zu gewöhnlichen Parametern von Funktionen entsprechen die Strukturterme den Termen, die man für die Parameter einsetzt, also den *aktuellen Parametern*.

Spezifikationen dienen als Typen von Strukturtermen. Jedem Strukturterm T ist eindeutig eine Spezifikation S zugeordnet, die die Struktur beschreibt. Analog zur Typnotation schreiben wir $T :: S$ (mit zwei Doppelpunkten statt einem zur besseren Unterscheidung).

Definition 4.11 (Menge der Strukturterme). *Für eine Umgebung $\mathcal{U}$ ist die Menge der Strukturterme $\mathcal{ST}_\mathcal{U}$ induktiv definiert:*

1. *Für jeden Parameter $N : S(@T'_1, \ldots, @T'_n, s'_1, \ldots, s'_m) \in \mathcal{U}$ ist $N \in \mathcal{ST}_\mathcal{U}$ und $N :: S$.*
2. *Hat für einen Strukturterm $A \in \mathcal{ST}_\mathcal{U}$ mit $A :: S$ die Spezifikation S einen Parameter $B : T(@T'_1, \ldots, @T'_n, s'_1, \ldots, s'_m) \in \mathcal{U}_S$, dann ist $B(A) \in \mathcal{ST}_\mathcal{U}$ und $B(A) :: T$.*
3. *Für eine Instanz I der Spezifikation S und Strukturterme $M_1, \ldots, M_n \in \mathcal{ST}_\mathcal{U}$ ist $I[M_1, \ldots, M_n] \in \mathcal{ST}_\mathcal{U}$, falls die Strukturterme $M_1, \ldots, M_n$ kompatibel zu den Parametern der Instanz I sind. Dann gilt $I[M_1, \ldots, M_n] :: S$. Kompatibilität wird im folgenden Abschnitt genau definiert. Bei leerer Parameterliste ($n = 0$) entfallen die eckigen Klammern.*

Strukturterme der Form $A(B)$ können verwendet werden, um auf Teilstrukturen einer Struktur zuzugreifen.

Beispiel 4.12. Wenn in einem Programm die Spezifikationen *Monoid* und *Group*, sowie die Instanzen *Plus* und *ProductMonoid* enthalten sind, und die Umgebung $\mathcal{U}$ die Parameter *M1:Monoid(@M,op,neut)* und *G:Group(@G,plus,*

null,negate) enthält, so sind unter anderem folgende Ausdrücke gültige Strukturterme (die zugehörige Spezifikation ist mit angegeben):

- *M1* :: *Monoid*
- *G* :: *Group*
- *M(G)* :: *Monoid*
- *ProductMonoid[Plus,M(G)]* :: *Monoid*

4.3.5 Sharing und Kompatibilität

Um Namenseindeutigkeit zu ermöglichen, beinhalten Strukturparameter eine Neubenennung aller Typ- und Funktionssymbole. In den meisten Fällen wird man die Umbenennung so wählen, dass alle Elemente verschiedene Namen bekommen. Das muss aber nicht unbedingt so sein.

Beispiel 4.13. Folgendes Lemma sagt aus, dass die Komposition zweier Gruppenhomomorphismen ebenfalls die Homomorphiebedingung erfüllt:

```
lemma composition
      [H1:GroupHomomorphism(@G1,@G2,op1,neut1,inv1,op2,neut2,inv2,g),
       H2:GroupHomomorphism(@G2,@G3,op2,neut2,inv2,op3,neut3,inv3,h)]
      ⇐ ∀ x,y:@G1
  op3(h(g(x)),h(g(y)))=h(g(op1(x,y)))
```

Hier werden bewusst die gleichen Symbolnamen *@G2*, *op2*, *neut2* und *inv2* für den Wertebereich des ersten Homomorphismus und den Definitionsbereich des zweiten Homomorphismus gewählt, um die Verkettung der Abbildungen auszudrücken.

Die Strukturparameter dieses Lemmas sind nicht unabhängig voneinander: Durch die übereinstimmenden Symbolnamen wird implizit ein Zusammenhang zwischen den Parametern gefordert. Das entspricht genau den *Sharing constraints*, die in Abschnitt 2.3.5 für ML beschrieben wurden. Anstatt zusätzliche Gleichungen anzugeben, wird Sharing in ✓`eriFun` durch die Vergabe gleicher Namen für die betreffenden Symbole ausgedrückt.

Sharing hat aber Auswirkungen auf das Typechecking einer Parametrisierung. Das liegt daran, dass das obige Lemma *composition* nicht mit zwei beliebigen Gruppenhomomorphismen parametrisiert werden kann, sondern diese zusammenhängen müssen. Hier unterscheidet sich Typechecking von Strukturparametrisierungen stark vom Typechecking normaler Funktionsaufrufe.

Im Rest dieses Abschnitts wird definiert, wann eine Liste von Strukturtermen zu einer Liste von (formalen) Strukturparametern kompatibel ist. Diese Kompatibilitätsprüfung ist eine Entsprechung für das Typechecking auf der Ebene von Strukturen.

Für einen Strukturterm T definieren wir zunächst eine funktionale Substitution σ_T, die den Symbolen aus der Spezifikation die Symbole aus der jeweiligen

Umgebung zuordnet und somit angibt, welche Typ- und Funktionssymbole eigentlich die Struktur bilden, die ein Strukturterm beschreibt.

Definition 4.14 (Substitution zu einem Strukturterm). *Sei $\mathcal{U}$ eine Umgebung. Für einen Strukturterm $T :: S$ wird eine funktionale Substitution $\sigma_T : \Sigma_S \rightarrow \Sigma_\mathcal{U}$ definiert, die den Elementen aus S Symbole aus der Umgebung $\mathcal{U}$ zuordnet.*

1. *Wenn T ein Parameter $T : S(@T'_1, \ldots, @T'_n, s'_1, \ldots, s'_m) \in \mathcal{U}$ ist, dann ist $\sigma_T := \{@T_1/@T'_1, \ldots, @T_n/@T'_n, s_1/s'_1, \ldots, s_m/s'_m\}$.*
2. *Wenn $T = B(T')$ für ein $T' :: S'$ und ein $B : S(@T'_1, \ldots, @T'_n, s'_1, \ldots, s'_m) \in \mathcal{U}_{S'}$, dann ist $\sigma_T := \sigma_{T'} \circ \sigma_B$, wobei σ_B in der Umgebung $\mathcal{U}_{S'}$ ermittelt wird.*
3. *Wenn $T = I[M_1, \ldots, M_n]$ für eine Instanz I und Strukturterme $M_1, \ldots, M_n$, dann ist $\sigma_T := \theta_{\mathcal{U}_I \rightarrow M_1,\ldots,M_n} \circ \sigma_I$, wobei $\theta_{\mathcal{U}_I \rightarrow M_1,\ldots,M_n}$ weiter unten definiert ist.*

Beispiel 4.15. In der modifizierten Definition von *fold* aus Beispiel 4.2 ist für den Strukturterm K die Substitution σ_K definiert als

$$\sigma_K = \{@M/@W, op/concat, neut/nil\}$$

Für den Strukturterm *Plus* (vgl. Beispiel 4.6) ist σ_{Plus} definiert als

$$\sigma_{Plus} = \{@M/nat, op/plus, neut/0\}$$

Anhand dieser Substitutionen kann man feststellen, ob eine Liste von Strukturen auf eine Liste von (formalen) Strukturparametern „passt“. Dazu definiert man zunächst eine Transformation, die die in den formalen Parametern aufgeführten Symbole durch Symbole aus den aktuellen Parametern (den Strukturtermen) ersetzt:

Definition 4.16 (Transformation). *Seien $\mathcal{U}_1$ und $\mathcal{U}_2$ Umgebungen und S eine Spezifikation mit der Signatur $(@T_1, \ldots, @T_n, s_1, \ldots, s_m)$. Für einen Strukturparameter $N : S(@T'_1, \ldots, @T'_n, s'_1, \ldots, s'_m) \in \mathcal{U}_1$ und einen Strukturterm $T \in \mathcal{ST}_{\mathcal{U}_2} :: S$ ist die* Transformation *$\theta_{N \rightarrow T} : \Sigma_{\mathcal{U}_1} \mapsto \Sigma_{\mathcal{U}_2}$ eine funktionale Substitution mit*

$$\theta_{N \rightarrow T} := \{@T'_1/\sigma_T(@T_1), \ldots, @T'_n/\sigma_T(@T_n), s'_1/\sigma_T(s_1), \ldots, s'_m/\sigma_T(s_m)\}$$

Man beachte, dass $\theta_{N \rightarrow T}$ nicht immer wohldefiniert ist, sondern nur unter der Bedingung, dass $\sigma_T(@T_i) = \sigma_T(@T_j)$, falls $@T'_i = @T'_j$ und analog $\sigma_T(s_i) = \sigma_T(s_j)$, falls $s'_i = s'_j$.

Beispiel 4.17. Sei *fold* definiert, wie in Beispiel 4.2. Wir benennen im Lemma *fold_append* zur besseren Unterscheidung ebenfalls die Symbole um:

```
lemma fold_append[M2:Monoid(@M2,op2,neut2)] ⇐ ∀ k,l:list[@M2]
  fold[M2](append(k,l))=op2(fold[M2](k),fold[M2](l))
```

Beim Aufruf von *fold* muss nun die Transformation $\theta_{K\to M2}$ für den Strukturparameter *K:Monoid(@W,concat,nil)* aus der Definition von *fold* und den Strukturterm *M2* aus der Umgebung des obigen Lemmas gebildet werden. Die Transformation lautet

$$\theta_{K\to M2} = \{@W/@M2, concat/op2, nil/neut2\}.$$

Eine Liste von Parameterpaaren ist kompatibel, wenn sich die einzelnen Transformationen miteinander kombinieren lassen:

Definition 4.18 (Kompatibilität). *Sei $\mathcal{U}_1 = \langle N_1, \ldots, N_k\rangle$ eine Umgebung und $T_1, \ldots, T_k \in \mathcal{ST}_{\mathcal{U}_2}$ eine Liste von Strukturtermen aus einer anderen Umgebung $\mathcal{U}_2$. Dann sind $\mathcal{U}_1$ und $T_1, \ldots, T_k$* kompatibel*, falls*

- *$T_i :: S_i$ für alle $i \in \{1..k\}$*
- *die Transformationen $\theta_{N_i\to T_i}$ für alle $i \in \{1..k\}$ wohldefiniert und paarweise kompatibel sind.*

Damit existiert auch eine gemeinsame Transformation

$$\theta_{\mathcal{U}\to T_1,\ldots,T_k} = \theta_{N_1\to T_1} \cup \ldots \cup \theta_{N_k\to T_k}$$

So lässt sich nun formulieren, wann wir die Parametrisierung $E[T_1, \ldots, T_n]$ eines Elements E als wohlgeformt ansehen. Das ist nämlich genau dann der Fall, wenn die formalen Parameter $\mathcal{U}_E$ und die aktuellen Parameter $T_1, \ldots, T_n$ kompatibel im Sinne von Definition 4.18 sind.

Beispiel 4.19. Das Kompositionslemma für Gruppenhomomorphismen aus Beispiel 4.13 soll durch die zwei Strukturen $Hom1$ und $Hom2$ aus folgender Umgebung parametrisiert werden:

[Hom1:GroupHomomorphism(@G1,@G2,op1,neut1,inv1,op2,neut2,inv2,g),
Hom2:GroupHomomorphism(@G3,@G4,op3,neut3,inv3,op4,neut4,inv4,h)]

Man bildet die beiden Transformationen

$$\theta_{H1\to Hom1} = \{@G1/@G1, @G2/@G2, op1/op1, \ldots, op2/op2, \ldots, g/g\}$$

$$\theta_{H2\to Hom2} = \{@G2/@G3, @G3/@G4, op2/op3, \ldots, op3/op4, \ldots, h/h\}$$

Diese Transformationen sind aber nicht kompatibel, da die Typvariable $@G2$ in $\theta_{H1\to Hom1}$ auf $@G2$ und in $\theta_{H2\to Hom2}$ auf $@G3$ abgebildet wird. Damit ist die Parametrisierung *composition[Hom1,Hom2]* nicht zulässig.

4.3.6 Kurzschreibweisen

Für die Eingabe gelten folgende Kurzschreibweisen:

- Bei der Eingabe von Strukturparametern können Symbolnamen vom Ende der Liste weggelassen werden (siehe Bsp. 4.3). Diese werden dann als gleichnamig zu den Elementen der Spezifikation angesehen.

- In rekursiven Aufrufen können Strukturparameter weggelassen werden. Es wird dann die Umgebung der Funktion übergeben.

- Lemmata gehören strenggenommen nicht in eine Spezifikation, da sie keine definierenden Eigenschaften sind, sondern allenfalls zusätzliche Aussagen *über* eine Spezifikation machen. Manche Lemmata (wie etwa die Kürzungsregel für Gruppen) sind aber doch so fundamentale Eigenschaften, dass man sie gerne als Teil der Spezifikation betrachten möchte. Um dem Rechnung zu tragen, wird es erlaubt, auch Lemmata in Spezifikationen einzufügen. Diese werden aber in jeder Hinsicht so behandelt, als würden sie außerhalb stehen und können auch jederzeit vom Benutzer verschoben werden. Lediglich bei der Eingabe kann der Strukturparameter weggelassen werden.

  Beispiel 4.20. Die Kürzungsregel für Gruppen kann also wie folgt eingegeben werden:

  Group `lemma` *right_cancellation* $\Leftarrow$ $\forall$ *x,y,z:@G*
  `if`*(op(x,z)=op(y,z),x=y,true)*

  Das wird vom System dann interpretiert als

  Group `lemma` *right_cancellation[Group1:Group(@G,op,neut,inv)]*
  $\Leftarrow$ $\forall$ *x,y,z:@G*
  `if`*(op(x,z)=op(y,z),x=y,true)*

4.4 Behandlung im Beweiser

Nachdem nun die Spracherweiterung beschrieben ist, betrachten wir, welche Änderungen sich für ✓`eriFun` im Umgang mit Funktionen und Lemmata ergeben.

4.4.1 Transformation von Funktionen

Die Parametrisierung einer Funktion macht ihren Rumpf abhängig von den aktuellen Strukturparametern. Beim Öffnen von Funktionsaufrufen muss der Rumpf entsprechend instanziiert werden.

Beispiel 4.21. Wir betrachten die Definition von *fold* mit den geänderten Symbolnamen aus Beispiel 4.2 und das folgende triviale Lemma:

`lemma` *fold_test[M:Monoid]* $\Leftarrow$
fold[M](empty)=neut

Für den Beweis des Lemmas muss die Definition von *fold* geöffnet werden. Dabei werden zuvor im Rumpf die Symbole aus den formalen Parametern durch die Symbole der aktuellen Parameter ersetzt:

$$if(?empty(empty), neut, op(hd(empty), fold[M](tl(empty)))) = neut$$

Diese Ersetzung wird genau durch die Transformation $\theta_{\mathcal{U}\to T_1,\ldots,T_k}$ aus Definition 4.18 beschrieben. Beim Öffnen der Funktion muss also diese funktionale Substitution auf den Rumpf der Funktion angewandt werden.

4.4.2 Verfügbarkeit von Axiomen und Lemmata bei der symbolischen Auswertung

Formale Strukturparameter definieren nicht nur lokale Typ- und Funktionssymbole, sondern stehen auch für die Gültigkeit der entsprechenden Axiome. Für den Beweis von Lemmata mit Strukturparametern müssen diese Axiome daher verfügbar gemacht werden.

Für jeden Strukturparameter $N : S(@T'_1, \ldots, @T'_n, s'_1, \ldots, s'_m)$ und jedes Axiom a aus der Spezifikation S muss der Rumpf ϕ_a des Axioms mit der funktionalen Substitution σ_N instanziiert werden. Die entstandene Formel $\sigma_N(\phi_a)$ kann dann für Beweise als gültig angenommen werden.

Zusätzlich zu den Axiomen sollen aber auch alle bereits gezeigten Aussagen über entsprechende Strukturen verfügbar sein. Ein verifiziertes Lemma mit Strukturparametern kann mit konkreten Strukturen instanziiert und dann verwendet werden. Die verwendeten Strukturen können dabei alle Arten von Strukturtermen sein, insbesondere auch Instanzen. Nach dem Beweis des Lemmas *fold_append* und der Instanz *Plus* kann sofort die entsprechende Instanz des Lemmas *fold_append[Plus]* verwendet werden. Die Instanziierung der Lemmata findet automatisch vor einer symbolischen Auswertung statt. Dabei wird für kompatible Parametrisierungen $T_1, \ldots, T_k$ des Lemmas die Transformation $\theta_{\mathcal{U}\to T_1,\ldots,T_k}$ berechnet und auf die Formel des Lemmas angewandt. Der symbolische Auswerter arbeitet dann mit den instanziierten Lemmata, die wie gewöhnliche Assumptions betrachtet werden können.

Problematisch bei dieser Vorgehensweise ist, dass die Anzahl der Parametrisierungen in vielen Fällen unendlich ist und, falls sie endlich ist, sehr stark wächst. Bereits aus den Instanzen *Plus* und *ProductMonoid* kann man durch wiederholte Produktbildung unendlich viele Monoidstrukturen zusammensetzen. Für alle diese Strukturen gilt natürlich das Lemma *fold_append*. Daher ist es prinzipiell unmöglich, zuvor alle Instanziierungen zu erzeugen. Selbst wenn man eine Tiefenbeschränkung für Funktorapplikationen festlegt, müssen oft sehr viele Lemmata erzeugt werden.

Diese Vorab-Generierung von Aussagen über Strukturen kann die Performanz des Systems in einigen Fällen merklich beeinträchtigen. Dabei ist nicht so sehr die Performanz der symbolischen Auswertungen betroffen, da viele der unnötigen Aussagen danach durch den eingebauten Lemmafilter von ✓`eriFun` wieder aussortiert werden. Stattdessen ist hier bereits die Generierung der entsprechenden Instanzen zeitaufwändig.

Für die Lösung des Problems gibt es zwei unterschiedliche Ansätze:

1. Anhand des auszuwertenden Terms kann man versuchen, durch eine Heuristik zu bestimmen, welche Instanzen eines Lemmas bei der Auswertung benötigt werden könnten. Dann generiert man nur diese Instanzen. Damit

eine solche Heuristik sinnvoll sein kann, muss sie allerdings mindestens so effizient sein, dass die Entscheidung für oder gegen die Instanziierung eines Lemmas in kürzerer Zeit möglich ist als die Konstruktion der Instanz.

2. Alternativ kann man ganz auf die Vorabinstanziierung verzichten. Dann müsste der symbolische Auswerter allerdings in der Lage zu sein, mit Strukturparametern umzugehen und diese während des Matchings von Klauseln automatisch zu instanziieren. Prinzipiell ist das sicher die elegantere Lösung[15]. Es stellt sich aber die Frage, inwieweit die Effizienz der Auswertungen durch eine solche Erweiterung beeinträchtigt würde.

Die bisherige Implementierung, die die Anzahl der Strukturen beschränkt, indem jeder Funktor maximal einmal angewendet werden darf, kann daher nur als Notlösung angesehen werden.

4.4.3 Umgang mit Inkonsistenzen

Die Möglichkeit, beliebige Axiome zu spezifizieren, kann leicht zu Inkonsistenzen führen. Niemand kann den Benutzer daran hindern, in einer Spezifikation widersprüchliche Axiome zu formulieren, für die es kein Modell gibt. Erst nachdem eine Instanz zu einer Spezifikation bewiesen wurde, ist die Konsistenz der Spezifikation sichergestellt.

Allein durch die Forderung nach einer Instanz zu jeder Spezifikation ist das Problem aber noch nicht gelöst. Denn durch die Kombination widersprüchlicher Spezifikationen lässt sich eine inkonsistente Umgebung konstruieren, obwohl jede Spezifikation für sich genommen konsistent ist.

Beispiel 4.22. Folgendes Beispiel demonstriert den Beweis von $0 = 1$:

```
specification Spec1
  operator c: nat
  axiom a1 ⇐ c=0

specification Spec2
  operator c: nat
  axiom a2 ⇐ c=1

lemma zero_equals_one[S1:Spec1(c),S2:Spec2(c)] ⇐ 0=1
```

Tatsächlich lässt sich dieses Lemma beweisen, in dem man jedes der Axiome einmal anwendet.

Das, was zunächst nach einem schwerwiegenden Problem aussieht, relativiert sich jedoch, wenn man die mit den Spezifikationen importierten Annahmen als Teil des Lemmas betrachtet. Dann ist das Lemma nur eine unspektakuläre Instanz von *ex falso quodlibet*. Für den Beweis „normaler“ Aussagen ist es

[15] Man würde schließlich auch nicht auf die Idee kommen, vor der Auswertung die Variablen in einem Lemma durch viele unterschiedliche Terme zu instanziieren und dann zu hoffen, dass man die Instanzen irgendwie anwenden kann.

auch unbrauchbar, da man diese Aussagen ebenfalls inkonsistent parametrisieren müsste, um das Lemma überhaupt anwenden zu können.

Beispiel 4.23. Ein Student hat mit hohem Arbeitseinsatz das Lemma *zero_equals_one* aus Beispiel 4.22 bewiesen. Irgendwie bereitet ihm die Aussage zwar Unbehagen, das ignoriert er aber zunächst und beweist daraus munter per Induktion die Aussage $\forall n : nat \quad 0 = n$ und daraus mit ein paar Handgriffen das Lemma

`lemma` *all_the_same_to_me[S1:Spec1(c),S2:Spec2(c)]* $\Leftarrow$ $\forall$ *n,m:nat*
n=m

Die seltsamen Strukturparameter stören ihn dabei nur geringfügig. Dann versucht er, mit diesem Lemma die Permutationsaussage für Heapsort zu beweisen (welche ja eine Gleichung zwischen natürlichen Zahlen ist, also im Prinzip eine Instanz von *all_the_same_to_me*). Das schlägt aber fehl, da das Lemma in dieser Umgebung nicht anwendbar ist.

Er beschließt daher, die lästigen Parameter durch Instanziierung loszuwerden. Dazu muss er nur geeignete Instanzen für die beiden Spezifikationen finden. Er findet auch schnell eine Funktion, die die erste Spezifikation erfüllt, nämlich die Null. Für die zweite Spezifikation muss die konstante Einsfunktion herhalten, und schon hat man für jede Spezifikation eine bewiesene Instanz. Aber das Lemma lässt sich immer noch nicht anwenden, denn die beiden Instanzen ergeben zusammen keine kompatible Parametrisierung (vgl. Definition 4.18) für die inkonsistente Umgebung. Kompatibel wären nur zwei Instanzen mit derselben Funktion. Dass es die nicht geben kann, leuchtet dem Studenten ein und er beweist Heapsort daraufhin auf die herkömmliche Art und Weise.

Als zusätzliche Sicherheit kann das System darüber Buch führen, ob eine Umgebung bereits als konsistent angesehen werden kann. Dafür benötigt man ein geeignetes entscheidbares und hinreichendes Kriterium für die Konsistenz einer Umgebung $\mathcal{U}$ unter Einbeziehung möglicher sharing constraints. Das ist aber nicht schwer: Man muss einfach geeignete Strukturterme $T_1, \ldots, T_k \in \mathcal{ST}_{\mathcal{U}_\emptyset}$ finden, die zu $\mathcal{U}$ kompatibel sind, wobei $\mathcal{U}_\emptyset$ die leere Umgebung ist. Diese Suche nach einer geeigneten Parametrisierung für eine Umgebung ist ohnehin bereits für die Generierung der für einen Beweis verfügbaren Lemmata notwendig (vgl. Abschnitt 4.4.2). Das System kann also anhand der verfügbaren Instanzen automatisch feststellen, ob durch diese bereits die Konsistenz einer Umgebung sichergestellt ist oder nicht. Nur wenn die Umgebung eines Elements als konsistent nachgewiesen wurde, kann das Element den Status *verified* erhalten.

4.5 Semantik der Parametrisierung

Dieser Abschnitt beschreibt eine operationale Semantik für die beschriebenen Erweiterungen, durch die definiert wird, wann ein Lemma als wahr anzusehen ist.

Bisher ist die Gültigkeit eines Lemmas wie folgt definiert [21]:

Definition 4.24. *Ein Lemma* `lemma LEM <=` $\forall x_1 : T_1, \ldots, x_n : T_n \;\; t_{bool}$ *ist* gültig*, wenn für jede Belegung* $q_1, \ldots, q_n$ *der Variablen* $t_1, \ldots, t_n$ *mit Konstruktorgrundtermen gilt:*

$$eval_P(t_{bool}[x_1/q_1, \ldots, x_n/q_n]) = true$$

Dabei ist $eval_P$ eine Auswertungsfunktion, die jedem variablenfreien Term einen Konstruktorgrundterm zuordnet.

Man beachte, dass diese Definition nichts damit zu tun hat, wann das System ein Lemma als bewiesen ansieht (also dann, wenn ein Beweis existiert und alle verwendeten Aussagen bewiesen sind). Man möchte lediglich eine abstrakte Vorstellung davon haben, wann man eine Aussage als gültig ansieht. Diese Vorstellung muss unabhängig von der Herleitbarkeit in einem bestimmten Kalkül sein[16]. Vom System erwartet man dann natürlich, dass es sich *korrekt* bezüglich dieses Gültigkeitsbegriffs verhält, dass es also nur solche Lemmata als bewiesen akzeptiert, die auch tatsächlich gültig sind.

Einen analogen Gültigkeitsbegriff benötigt man für Lemmata mit Strukturparametern, die Spezifikationen verwenden. Dabei ergeben sich zwei wesentliche Schwierigkeiten: Zum einen kann man nicht mehr einfach jeden Term zu einem Konstruktorgrundterm auswerten, da der Term unter Umständen axiomatisch definierte Operatoren enthält, die nicht ausgewertet werden können. Zum zweiten müssen Strukturparameter und parametrisierte Funktionen und Lemmata geeignet interpretiert werden.

Dazu greifen wir auf eine andere Spracherweiterung zurück [1], die parallel zu dieser Arbeit entstanden ist und Funktionen höherer Ordnung in ✓`eriFun` einführt. Parametrisierungen durch Strukturen lassen sich in weiten Teilen mit Funktionen höherer Ordnung ausdrücken, so dass man die Spracherweiterung dieser Arbeit in die Spracherweiterung aus [1] übersetzen kann. Aufbauend auf der operationalen Semantik für Funktionen höherer Ordnung gewinnt man dann recht einfach eine Semantik für unsere Spracherweiterung. Die Übersetzung wird hier allerdings nur an Beispielen erläutert, um das Prinzip zu demonstrieren.

4.5.1 Höherstufige Funktionen

Funktionen höherer Ordnung gehören zu den wichtigsten Sprachkonzepten funktionaler Programmiersprachen (wiederum seien hier Haskell und ML als Beispiele genannt), da sie oft besonders kompakte Formulierungen ermöglichen. Grundidee dabei ist, dass Funktionen andere Funktionen als Parameter besitzen können.

[16] Ansonsten käme man letztendlich zu einer wenig sinnvollen Definition im Stil von „Ein Lemma ist genau dann gültig, wenn es mit dem System bewiesen werden kann“, wodurch das System bereits per Definition immer korrekt wäre.

Beispiel 4.25. In ✓eriFun kann man die bekannte Map-Funktion wie folgt definieren:

```
function map(f: @A → @B, l:list[@A]):list[@B] ⇐
  if ?empty(l)
    then empty
    else add(f(hd(l)),map(f,tl(l)))
  end_if
```

Der Parameter f hat den Funktionstyp $@A \to @B$. Man kann an *map* jede einstellige Funktion übergeben. Diese Funktion wird dann auf jedes Element der Liste angewandt.

Ebenso können Lemmata definiert werden, die Funktionstypen enthalten.

Beispiel 4.26. Folgendes Lemma beschreibt eine einfache Aussage über *map*:

```
lemma map_append ⇐ ∀ f: @A → @B, k,l:list[@A]
  map(f,append(k,l))=append(map(f,k),map(f,l))
```

4.5.2 Übersetzung von parametrisierten Funktionen

Die Idee zur Übersetzung besteht darin, die lokalen Funktionssymbole aus den Strukturparametern einer Funktionsdefinition zu funktionalen Parametern der Funktion zu machen.

Beispiel 4.27. Die Funktion *fold* aus Beispiel 4.1 kann wie folgt übersetzt werden:

```
function fold(op: @M,@M → @M, neut: @M, l:list[@M]):@M ⇐
  if ?empty(l)
    then neut
    else op(hd(l),fold(op,neut,tl(l)))
  end_if
```

Die Typvariable $@M$ aus dem Strukturparameter *M:Monoid(@M,op,neut)* entfällt und wird zu einer implizit allquantifizierten Typvariable. Die Funktion *fold* ist polymorph und damit für alle Typen $@M$ definiert. Die benötigten Operationen erwartet sie als Parameter. Auf diese Weise lassen sich alle lokalen Funktionssymbole in Parameter umwandeln.

4.5.3 Interpretation von parametrisierten Lemmata

Auch bei Lemmata lassen sich lokale Funktionssymbole als Funktionsparameter codieren. Hier müssen aber die Axiome, die über die Funktionsparameter vorausgesetzt werden, als Voraussetzungen mit in das Lemma übernommen werden. Die entstehenden Formeln lassen sich aber nicht mehr in der quantorenfreien Logik von ✓eriFun ausdrücken, da sie in den Prämissen zusätzliche Allquantoren enthalten.

Beispiel 4.28. Aus dem Lemma *fold_append* entsteht nach der Übersetzung folgende Formel, die nicht mehr als ✓eriFun-Lemma, also als universelle Formel, geschrieben werden kann:

$$\begin{aligned}
&\forall op : @M, @M \to @M, neut : @M \\
&\quad ((\forall x, y, z : @M \quad op(x, op(y, z)) = op(op(x, y), z)) \\
&\quad \wedge (\forall x : @M \quad op(neut, x) = x) \\
&\quad \wedge (\forall x : @M \quad op(x, neut) = x)) \\
&\quad \to \forall k, l : list[@M]\ fold(op, neut, append(k, l)) \\
&\qquad\qquad\qquad = op(fold(op, neut, k), fold(op, neut, l))
\end{aligned}$$

An dieser Stelle wird auch deutlich, dass unsere Spracherweiterung tatsächlich die Logik des Systems ausdrucksstärker macht: Obige Formel lässt sich nicht so umformen, dass alle Allquantoren außen stehen. In der Tat hat man in $\mathcal{FP}$ mit Funktionen höherer Ordnung häufig den Wunsch, Annahmen über einen Funktionsparameter zu machen, wie beispielsweise die Kommutativität der übergebenen Funktion. Spezifikationen bieten genau diese Funktionalität, indem sie Axiome enthalten können. Tatsächlich wäre unsere Erweiterung ohne Axiome lediglich eine umständliche Formulierung von Funktionen höherer Ordnung und ließe sich eins-zu-eins in diese übersetzen.

Für die Definition der Semantik kann man hier also leider nicht direkt auf eine bekannte Semantik für $\mathcal{FP}$ mit Funktionen höherer Ordnung zurückgreifen, da das Ergebnis der Übersetzung kein ✓eriFun -Lemma ist. Der Begriff der *Gültigkeit eines Lemmas* muss also noch geringfügig erweitert werden.

Definition 4.29 (Erweitertes Lemma). *Ein erweitertes Lemma ist eine Formel folgender Form (Typbezeichnungen wurden der Übersicht halber ausgelassen):*

$$\begin{aligned}
&\forall x_1, \ldots, x_n \\
&\quad ((\forall y_{1,1}, \ldots, y_{1,m_1} \quad t_1) \\
&\quad \wedge \quad \vdots \\
&\quad \wedge (\forall y_{k,1}, \ldots, y_{k,m_k} \quad t_k)) \\
&\quad \to \forall z_1, \ldots, z_q \quad t_{k+1}
\end{aligned}$$

Dabei sind $t_1, \ldots, t_k$ boolesche Terme mit den Variablen $x_1, \ldots, x_n, y_{k,1}, \ldots, y_{k,m_k}$ und t_{k+1} ein boolescher Term mit den Variablen $x_1, \ldots, x_n, z_1, \ldots, z_q$.

Man kann sich leicht klarmachen, dass jedes Lemma mit Strukturparametern in ein erweitertes Lemma mit funktionalen Parametern übersetzt werden kann. In dieser Übersetzung sind die Variablen $x_1, \ldots, x_n$ die Operatoren aus der Umgebung. Die Prämissen der Implikation enthalten die Axiome, die sich aus der Umgebung ergeben, und die Konklusion ist das eigentliche Lemma.

Beispiel 4.30. Die Übersetzung des Lemmas *fold_append* aus Beispiel 4.28 ist ein erweitertes Lemma.

Wir definieren nun, wann ein erweitertes Lemma als wahr anzusehen ist. Dazu setzen wie die Existenz einer operationalen Semantik für $\mathcal{FP}$ mit Funktionen höherer Ordnung voraus. Das heißt, für ein gegebenes terminierendes Programm

P existiert eine Auswertungsfunktion $eval_P : T(\Sigma(P)) \to T(\Sigma^c)$, die jedem variablenfreien Term einen Konstruktorgrundterm zuordnet.

Definition 4.31 (Gültigkeit eines erweiterten Lemmas). *Sei L ein erweitertes Lemma mit den Bezeichnungen aus Definition 4.29. Eine Belegung $q_1, \ldots, q_n \in T(\Sigma^c)$ der Variablen $x_1, \ldots, x_n$ heißt* zulässig *in P, wenn für alle $i \in \{1, \ldots, k\}$ und für alle Belegungen $r_1, \ldots, r_{m_i} \in T(\Sigma^c)$ der Variablen $y_{i,1}, \ldots, y_{i,m_i}$ der Term t_i zu* true *auswertet, das heißt*

$$eval_P(t_i[x_1/q_1, \ldots, x_n/q_n, y_{i,1}/r_1, \ldots, y_{i,m_i}/r_m]) = true$$

Das erweiterte Lemma L ist gültig *in einem Programm P, falls für jede zulässige Erweiterung P' von P und für jede zulässige Belegung $q_1, \ldots, q_n$ in P' der Konklusionsterm t_k für alle $s_1, \ldots, s_q \in T(\Sigma^c)$ in P' zu* true *auswertet:*

$$eval_{P'}(t_k[x_1/q_1, \ldots, x_n/q_n, z_1/s_1, \ldots, z_q/s_q]) = true$$

Ein Lemma mit Strukturparametern betrachten wir als gültig, wenn seine Übersetzung in ein erweitertes Lemma gültig im Sinne von Definition 4.31 ist.

Instanzen kann man einfach als Abkürzungen der durch sie entstehenden Beweisverpflichtungen ansehen. Bei der Übersetzung einer Instanz in die Sprache mit Funktionen höherer Ordnung schreibt man einfach die entsprechenden Lemmata.

4.5.4 Überlegungen zur Korrektheit

An dieser Stelle stellt sich nun die Frage nach der Korrektheit des Systems, also die Frage, ob jedes Lemma, welches mit ✓`eriFun` als gültig nachgewiesen wurde, auch tatsächlich gültig im Sinne von Definition 4.31 ist. Ein formaler Korrektheitsbeweis kann hier schon deshalb nicht geführt werden, weil dazu die genaue Syntax der Erweiterungen für Funktionen höherer Ordnung, die Übersetzung in diese Sprache und die (informell in Abschnitt 4.4.2 beschriebene) Arbeitsweise des Systems bei der Generierung der lokalen Annahmen erst rigoros definiert werden müsste.

Man kann aber doch zumindest skizzieren, wie ein Beweis aussehen müsste, der die Korrektheit des Systems formal rechtfertigt. Der springende Punkt liegt hier in der Generierung der instanziierten Lemmata (Abschnitt 4.4.2), die für einen Beweis verfügbar gemacht werden. Dabei müsste gezeigt werden, dass ein Lemma, das mit Strukturtermen aus einer Umgebung $\mathcal{U}$ instanziiert wird, in dieser Umgebung auch gültig ist, es also aus den Axiomen der Umgebung folgt. Dabei müssen die durch die Parametrisierung impliziten Voraussetzungen (die Prämissen aus Definition 4.29) im Kontext von $\mathcal{U}$ gezeigt werden. Je nach Art der Strukturterme wird man dabei entweder auf die lokalen Axiome oder auf die Beweise der benutzten Instanzen zurückgreifen.

4.6 Änderungen aus Benutzersicht

Neben den Änderungen in der Syntax ergeben sich für den Benutzer weitere Neuerungen, die dieser Abschnitt kurz beschreibt.

4.6.1 Eingabe von Spezifikationen

Neu eingefügte Spezifikationen sind zunächst leer. Der Benutzer fügt dann der Reihe nach die gewünschten Deklarationen als neue Elemente ein. Im Gegensatz zu Modulen und Ordnern spielt aber bei Spezifikationen die Reihenfolge eine Rolle, da durch sie die Signatur der Spezifikation festgelegt wird. Da ein nachträgliches Ändern der Signatur relativ aufwändig wäre, ist das Verschieben von Elementen innerhalb von Spezifikationen generell nicht erlaubt. Sobald eine Spezifikation verwendet wird (als Strukturparameter oder durch Angabe einer Instanz), können keine Elemente mehr eingefügt, modifiziert oder gelöscht werden. Das Kommando „Delete Depending“, angewandt auf ein Element der Spezifikation, löscht dann auch alle Elemente, die die Spezifikation verwenden.

4.6.2 Status von Elementen

In ✓`veriFun` ist jedem Programmelement ein Status zugeordnet, der den Fortschritt (und Erfolg) des Beweises beschreibt und sich im Verlauf eines Beweises ändert. Mögliche Statuswerte sind *ready*, *developed*, *verified*, *falsified* und *ignored* (vgl. [21]). Für die Darstellung von Containern gibt es zusätzlich den Status *mixed*, der anzeigt, dass enthaltene Elemente unterschiedlichen Status haben.

Für den Status der neu eingeführten Elemente wurden folgende Festlegungen getroffen:

Module Da ein Modul keine eigene Beweisverpflichtung hat, hat es den Status *verified*, falls alle enthaltenen Elemente *verified* sind, ansonsten den Status *mixed*.

Domains, Operatoren, Axiome Elemente in einer Spezifikation haben keine Beweisverpflichtungen und immer den Status *verified*.

Spezifikationen Eine Spezifikation S hat den Status *verified*, sobald eine Instanz von S existiert oder eine von S erbende Spezifikation *verified* ist. Ansonsten hat sie den Status *developed*. Der Status zeigt somit an, ob die Konsistenz der Spezifikation bereits bewiesen ist.

Instanzen Eine Instanz hat den Status *verified*, sobald alle generierten Beweisverpflichtungen den Status *verified* haben. Ansonsten hat die Instanz den Status *ready*. Hat eine der Beweisverpflichtungen den Status *falsified*, so erhält die Instanz ebenfalls diesen Status.

Alle Elemente mit Strukturparametern Für Elemente mit Strukturparametern gilt generell, dass sie erst den Status *verified* erhalten, wenn neben den sonstigen Beweisverpflichtungen auch die Konsistenz der Umgebung

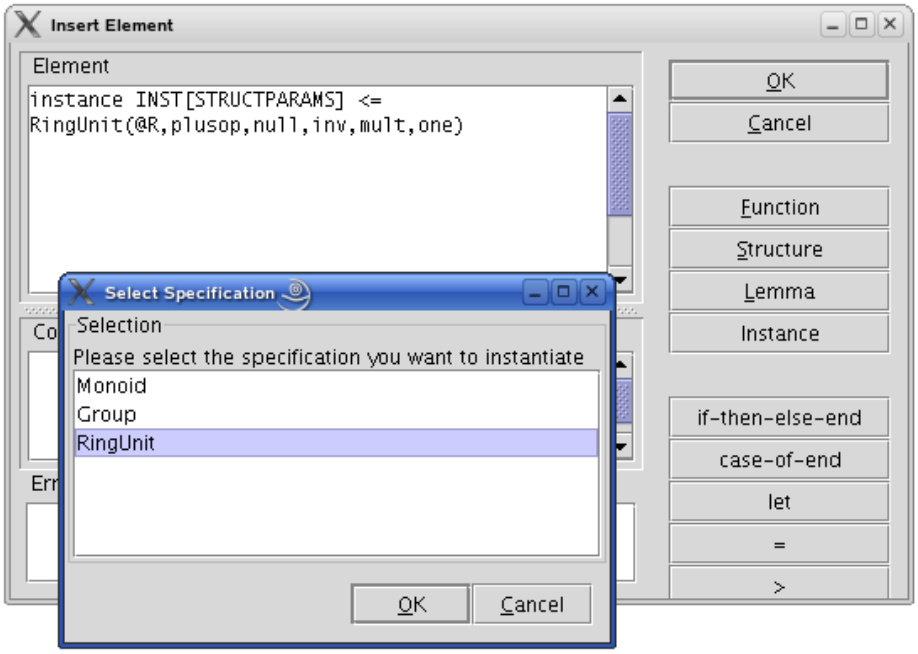

Abbildung 4.1: Generierung einer Vorlage für eine neue Instanz

gezeigt ist. Das ist der Fall, wenn kompatible Instanzen für die Umgebung existieren.

4.6.3 Unterstützung durch Dialoge

Für eine vereinfachte Eingabe der neuen Elemente stehen, ähnlich wie bei den bereits vorhandenen Elementen, Schablonen bereit, die im Insert Dialog durch Klick auf die entsprechenden Buttons verfügbar sind. Wird ein Element in eine Spezifikation eingefügt, so erscheinen hier statt der Buttons *Function*, *Structure* und *Lemma* entsprechende Buttons für *Operator*, *Domain* und *Axiom*. Für die Eingabe von Instanzen gibt es ebenfalls eine Hilfe. Dabei wählt der Benutzer vorher die Spezifikation aus, die er instanziieren möchte. Dann wird eine für die Spezifikation passende Schablone generiert, in der nur noch die entsprechenden Typ- und Funktionssymbole ersetzt werden müssen. Abb. 4.1 zeigt den Dialog zur Auswahl der Spezifikation und die generierte Schablone.

Da Spezifikationen von anderen Spezifikationen erben können, ist es wichtig, dass der Benutzer den Überblick über alle Elemente einer Spezifikation behalten kann. Dafür wurde im Program Viewer eine neue Ansicht „Signature“ hinzugefügt (Abb. 4.2), die die Signatur einer Spezifikation und die Menge der (eigenen und ererbten) Axiome sowie die Unterstrukturen anzeigt.

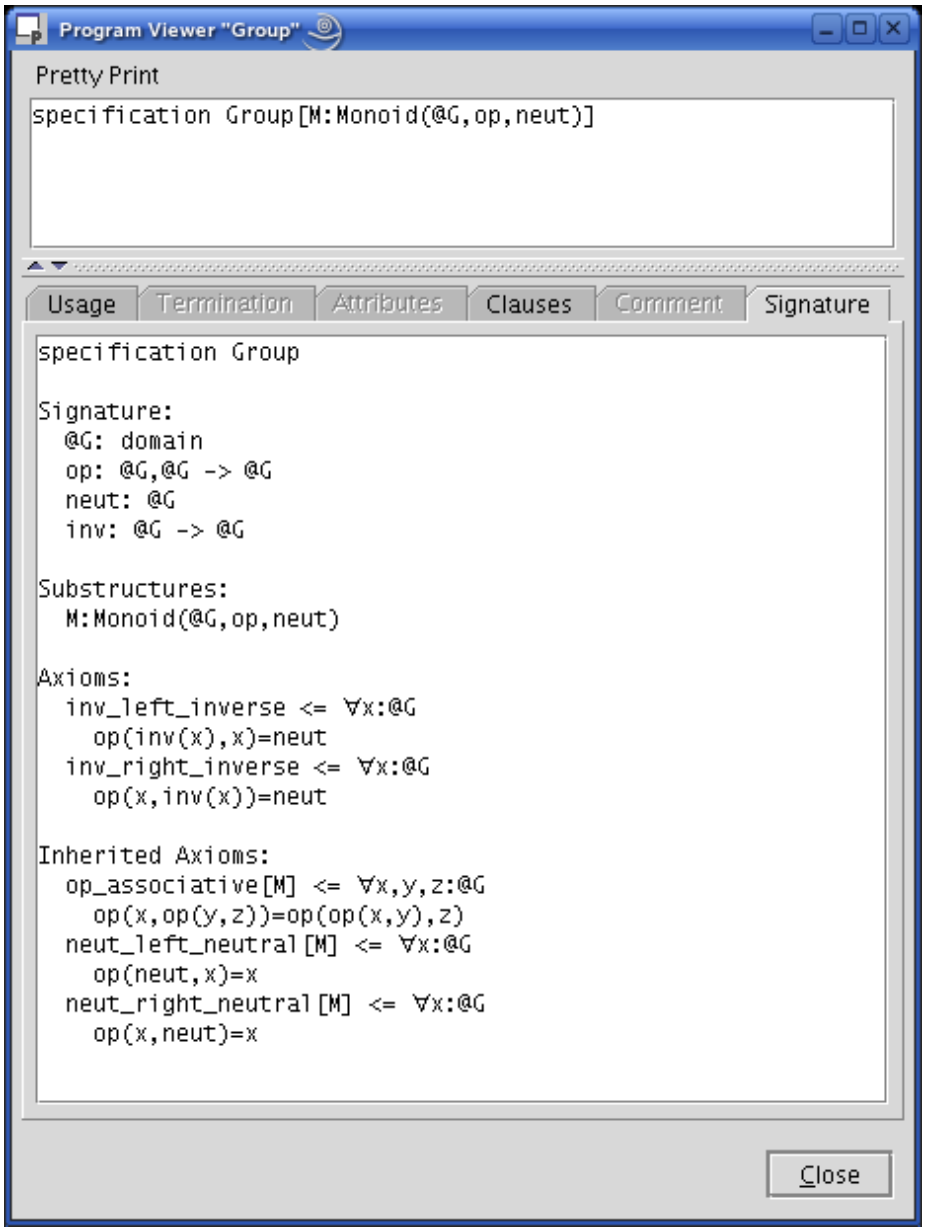

Abbildung 4.2: Anzeige der Spezifikation *Group* im Program Viewer

4.7 Vergleich mit zuvor behandelten Konzepten

Durch die explizitere Syntax und den Verzicht auf Overloading sind die Probleme, die bei Typklassen sowohl in Haskell als auch in Isabelle/HOL auftreten (vgl. Abschnitt 2.2.4) gelöst: √`eriFun` unterstützt sowohl Spezifikationen ohne Typsymbol als auch Typrelationen und mehrere Instanzen für eine Spezifikation. Jede Typklasse lässt sich direkt in eine Spezifikation übersetzen.

Beispiel 4.32. Eine Typklasse

```
class C a
  op1 :: T1[a]
  op2 :: T2[a]
```

wird wie folgt in eine Spezifikation übersetzt:

```
specification C
  domain @A
  operator op1 : T1[@A]
  operator op2 : T2[@A]
  axiom ...
```

Während Typklassen also spezielle Spezifikationen mit genau einer `domain`-Deklaration sind, werden Typrelationen durch Spezifikationen mit zwei oder mehr Typsymbolen beschrieben (vgl. Beispiel 4.5).

Die entscheidende Verbesserung, die die Probleme aus Abschnitt 2.2.4 löst, besteht darin, dass in √`eriFun` neben den Typsymbolen auch die Operatorsymbole aus der Spezifikation/Typklasse neue Namen erhalten, wenn die Spezifikation/Typklasse verwendet wird (vgl. Beispiel 2.2 mit Beispiel 4.4). So ist man nicht auf Overloading angewiesen, und es können keine Mehrdeutigkeiten bezüglich der Verwendung der Operatorsymbole entstehen.

Im Vergleich mit Standard ML entsprechen √`eriFun`-Spezifikationen den ML-Signaturen. Instanzen entsprechen den Strukturen in ML, mit dem Unterschied, dass in √`eriFun` aufgrund der Axiome die Korrektheit einer Instanz nicht entscheidbar ist, weshalb hier ein Beweis geführt werden muss.

Wie auch in ML gibt es in √`eriFun` hierarchisch gegliederte Strukturen, die durch Vererbung bei Spezifikationen entstehen und auf die über Selektoren zugegriffen werden kann.

In beiden Modellen gibt es *sharing constraints*, durch die bei der Kombination verschiedener Strukturen Bedingungen über die Kompatibilität verschiedener Typen ausgedrückt werden können. Das geschieht jedoch auf unterschiedliche Weise: In √`eriFun` durch die Vergabe gleicher Namen in den Strukturparametern, in ML durch explizite Angabe von Gleichungen wie in Beispiel 2.5.

Parametrisierte Funktionen, wie sie in √`eriFun` geboten werden, haben kein direktes Äquivalent in ML. Dort können nur Strukturen durch andere Strukturen parametrisiert werden. Um also die Funktion *fold* in ML auszudrücken, muss man einen Funktor angeben, der die entsprechende Funktion in eine Struktur packt und dann zurückliefert.

Insgesamt scheinen die Unterschiede zum Modulsystem von Standard ML rein syntaktisch zu sein. Alle untersuchten Beispiele lassen sich in beiden Systemen gleichermaßen ausdrücken und ineinander übersetzen (natürlich mit Ausnahme der Axiome, die nur von ✓eriFun unterstützt werden). Ein Beweis für die Gleichmächtigkeit wurde jedoch nicht geführt.

Die funktionale Instanziierung aus Abschnitt 2.4 diente als Vorlage für die Bildung von Instanzen. Darüber hinaus bietet ✓eriFun deutlich mehr Komfort: Die instanziierten Elemente müssen nicht neu vom Benutzer eingegeben werden, sondern werden automatisch vom System generiert. Auch ist kein zusätzlicher Interaktionsschritt notwendig, um aus allgemeinen Lemmata die entsprechenden instanziierten Aussagen zu erhalten. Diese stehen automatisch zur Verfügung, nachdem die Gültigkeit der Axiome für die Instanz bewiesen wurde.

4.8 Probleme und Grenzen

Die bisher betrachteten Beispiele haben bereits demonstriert, dass Spezifikationen ein sehr ausdrucksstarkes Sprachmittel sind. Dennoch gibt es natürlich auch hier ein paar Schwierigkeiten, die in diesem Abschnitt diskutiert werden sollen. Dabei werden nach Möglichkeit auch Ansätze zu deren Lösung vorgestellt.

4.8.1 Terminierung

Ein zentraler Baustein der Terminierungsanalyse von ✓eriFun beruht auf Abschätzungen über die Größe von Termen. Ein Terminierungsbeweis besteht im wesentlichen darin zu zeigen, dass gewisse Argumente beim rekursiven Aufruf stets kleiner bezüglich einer fundierten Relation sind als die ursprünglichen Argumente. Die zugrunde liegende Ordnung ist dabei sehr häufig die Teiltermordnung auf Konstruktorgrundtermen. Diese Analyse lässt sich leider auf abstrakte Datentypen nicht übertragen. Denn die Struktur der Terme ist bei einem abstrakten Datentyp unbekannt.

Beispiel 4.33. Als Beispiel betrachten wir die Standardaxiomatisierung [2] einer Queue von natürlichen Zahlen und eine Funktion, die solche Queues in Listen transformiert:

```
specification NatQueue
  domain @Q
  operator emptyQueue: @Q
  operator enqueue: nat,@Q -> @Q
  operator front: @Q -> nat
  operator remove: @Q -> @Q
  axiom front_single ⇐ ∀ x:nat
    front(enqueue(x,emptyQueue))=x
  axiom remove_single ⇐ ∀ x:nat
    remove(enqueue(x,emptyQueue))=emptyQueue
  axiom front_more ⇐ ∀ x,y:nat,q:@Q
    front(enqueue(x,enqueue(y,q)))=front(enqueue(y,q))
  axiom remove_more ⇐ ∀ x,y:nat,q:@Q
    remove(enqueue(x,enqueue(y,q)))=enqueue(x,remove(enqueue(y,q)))
```

```
function queue_to_list[Q:NatQueue](q:@Q):list[nat] ⇐
  if q=emptyQueue
    then empty
    else add(front(q),queue_to_list[Q](remove(q)))
  end_if
```

Der Terminierungsbeweis für die Funktion *queue_to_list* schlägt fehl: Es ist nicht garantiert, dass das Argument des rekusiven Aufrufs *remove(q)* bezüglich irgendeiner fundierten Ordnung kleiner ist als q. Das ist aber kein Problem von schlechten Heuristiken, denn man kann die Terminierung tatsächlich nicht garantieren. Das zeigt die folgende Implementierung einer Queue:

```
structure queue ⇐
  emptyq,
  addq(hdq:nat,tlq:queue),
  strange

function emptyQueue: queue ⇐
  emptyq

function enqueue(x:nat,q:queue):queue ⇐
  case q of
    emptyq: addq(x,emptyq)
    addq: addq(hdq(q),enqueue(x,tlq(q)))
    strange: strange
  end_case

function front(q:queue):nat ⇐
  case q of
    emptyq: 0
    addq: hdq(q)
    strange: 0
  end_case

function remove(q:queue):queue ⇐
  case q of
    emptyq: 0
    addq: tlq(q)
    strange: strange
  end_case
```

Die Implementierung des Typs umfasst neben einer listenartigen Struktur noch ein zusätzliches Objekt *strange*, welches so behandelt wird wie eine Queue, die unendlich viele Nullen enthält. Damit terminiert die Auswertung von *queue_to_list(strange)* nicht.

Das Problem liegt hier also in der Spezifikation, die nicht verbietet, dass es unendlich grosse Queues gibt. Man muss also die Spezifikation um eine Maßfunktion erweitern:

```
operator size: @Q -> nat

axiom remove_decreases_size ⇐ ∀ q:@Q
  if(q=emptyQueue,true,size(q)>size(remove(q)))
```

In dieser erweiterten Spezifikation wird jeder Queue eine Größe zugeordnet und gefordert, dass sich diese Größe bei jeder *remove*-Operation verringert. Nun gelingt der Terminierungsbeweis von *queue_to_list*, wenn der Benutzer *size(q)* als Maßterm eingibt.

Allgemein sind axiomatische Spezifikationen von Datenstrukturen schwerer zu überblicken als die bisher in VeriFun existierenden algebraischen Typen, die mit ihren initialen Modellen („No Junk, No Confusion“) sehr übersichtlich sind. Bei der auf den ersten Blick vollständig spezifizierten Queue hat sich erst beim Versuch des Terminierungsbeweises für *queue_to_list* herausgestellt, dass die Spezifikation auch unerwünschte Modelle hat. Hier wird dem Benutzer eine besondere Sorgfalt abverlangt, die ihm das System aber prinzipiell nicht abnehmen kann.

Eine kleine konkrete Verbesserung könnte darin bestehen, dass das System die Maßfunktion *size* (falls sie in der Spezifikation vorhanden ist) automatisch findet und benutzt. Das würde die Benutzerinteraktion vor jedem Terminierungsbeweis einer rekursiv über Queues definierten Funktion überflüssig machen. Ein erster Ansatz wäre, genau die Funktionen, die vom betreffenden Typ auf *nat* abbilden, als Kandidaten für Maßfunktionen zu berücksichtigen.

4.8.2 Spezifikation der Gleichheit

Eine weitere Problematik besteht in der Behandlung der Gleichheit auf axiomatisch spezifizierten Datenstrukturen. Das wird deutlich, wenn man versucht, die Menge der ganzen Zahlen als Paare von natürlichen Zahlen auszudrücken. Die Idee hinter der Konstruktion ist, dass ein Paar (x, y) die ganze Zahl $x - y$ repräsentieren soll. (Die analoge Konstruktion von $\mathbb{Q}$ aus $\mathbb{Z}$ ist eine Standardkonstruktion in der Algebra.) Von der Konstruktion möchte man zeigen, dass sie die Axiome einer Gruppe erfüllt. Man definiert also zuerst die Addition, die Null und die Inversenbildung auf den Paaren:

```
function add_pair(x,y:pair[nat,nat]): pair[nat,nat] ⇐
  mkpair(plus(fst(x),fst(y)),plus(snd(x),snd(y)))

function zero_pair: pair[nat,nat] ⇐
  mkpair(0,0)

function inv_pair(x:pair[nat,nat]): pair[nat,nat] ⇐
  mkpair(snd(x),fst(x))
```

Als nächsten Schritt würde man nun definieren, dass zwei Paare (x_1, y_1) und (x_2, y_2) als gleich anzusehen sind, wenn $x_1 + y_2 = x_2 + y_1$ ist. Eine solche Redefinition der Gleichheit (und damit Bildung einer Quotientenstruktur) ist aber im System nicht vorgesehen. Daher scheitert auch der Beweis des Axioms über das inverse Element, denn $(0, 1) + (1, 0) = (1, 1) \neq (0, 0)$.

Um das Problem zu umgehen, kann der Benutzer bereits in der Spezifikation der Gruppe (bzw. der Monoidspezifikation, von der die Gruppe erbt) eine eigene Relation definieren, die die Gleichheit in der Quotientenstruktur modelliert:

`operator` *equals @G,@G −> bool*

`axiom` *equals_reflexive* ⇐ ∀ *x:@G*
equals(x,x)

`axiom` *equals_transitive* ⇐ ∀ *x,y,z:@G*
`if`*(equals(x,y),*`if`*(equals(y,z),equals(x,z),true),true)*

`axiom` *equals_symmetric* ⇐ ∀ *x,y:@G*
`if`*(equals(x,y),equals(y,x),true)*

Für die Paare kann die Relation dann entsprechend implementiert werden:

`function` *pairequal(x,y:pair[nat,nat]):bool* ⇐
plus(fst(x),snd(y))=plus(fst(y),snd(x))

Diese Modellierung hat zwei Nachteile: Erstens wird die Spezifikation dadurch größer und damit unübersichtlicher. Neben der Gleichheit durch *equals* existiert auch nach wie vor die =-Gleichheit, die jetzt aber nicht mehr verwendet wird, da sie nicht die gewünschte Relation beschreibt. Insbesondere muss von Anfang an in einer Spezifikation vorgesehen werden, dass es eine Instanz geben könnte, die eine eigene Gleichheit definiert. Wenn noch nicht klar ist, welche Instanzen es zu einer (evtl. sehr allgemeinen) Spezifikation geben wird, müsste man immer sicherheitshalber eine Gleichheitsrelation einbauen, um nicht später bei der Instanzbildung eingeschränkt zu sein. Der zweite Nachteil besteht darin, dass die neue Gleichheit zwar als Äquivalenzrelation bewiesen wurde, man aber nicht modellieren kann, dass auch eine *Kongruenz* vorliegt, dass also für jede Funktion f mit einem Parameter x_i des entsprechenden Typs aus $\mathit{pairequals}(t_i, t_i')$ auch $\mathit{pairequals}(f(t_1, \ldots, t_i, \ldots, t_n), f(t_1, \ldots, t_i', \ldots, t_n))$ bzw. $f(t_1, \ldots, t_i, \ldots, t_n) = f(t_1, \ldots, t_i', \ldots, t_n)$ folgt (je nach Rückgabetyp der Funktion f). Damit sind die Mechanismen zur Gleichheitsbehandlung für diese Relation nicht anwendbar, und eine Beweisregel wie *Apply Equation* kann nicht verwendet werden.

Um dieses Problem elegant zu lösen, müsste der Benutzer bei der Angabe einer Instanz zusätzlich die Möglichkeit haben, für jedes Typsymbol $@T_i$, welches durch die Instanz durch einen Typ T_i instanziiert wird, eine benutzerdefinierte Relation equals_{T_i} als Gleichheit anzugeben, die auf dem entsprechend instanziierten Typ operiert. Wie andere Operatoren der Spezifikation wird so auch die Gleichheit instanziiert. Da die Gleichheit aber den Kongruenzforderungen genügen muss, würde ✓`veriFun` dann zusätzlich zu den instanziierten Axiomen folgende Beweisverpflichtungen generieren:

- Die Reflexivität, Transitivität und Symmetrie der Relation
- Die Kongruenz bezüglich aller Operatoren der Spezifikation (bzw. deren Instanzen)

Im Rahmen der Instanz *NatPairGroup* wären dann neben der Äquivalenz noch die beiden folgenden Kongruenzlemmata zu zeigen:

```
lemma pairequals_op_congruent ⇐ ∀ x1,x2,y1,y2:pair[nat,nat]
  if(pairequals(x1,x2),
     if(pairequals(y1,y2),
        pairequals(add_pair(x1,y1),add_pair(x2,y2)), true), true)

lemma pairequals_inv_congruent ⇐ ∀ x1,x2:pair[nat,nat]
  if(pairequals(x1,x2), pairequals(inv(x1),inv(x2)), true)
```

Die Axiome wären dann nicht mehr über die vordefinierte Gleichheit, sondern über die neue benutzerdefinierte Gleichheit zu beweisen.

Die Instanziierung der Gleichheit müsste dann auch bei der Instanziierung bewiesener Lemmata (vgl. Abschnitt 4.4.2) berücksichtigt werden. Die Kürzungsregel für Gruppen würde für die oben beschriebene Paar-Instanz dann so lauten:

```
lemma right_cancellation[NatPairGroup] ⇐ ∀ x,y,z:pair[nat,nat]
  if(pairequals(add_pair(x,z),add_pair(y,z)),pairequals(x,y),true)
```

In der Praxis wird sich zeigen, ob es für eine derartige Erweiterung eine Notwendigkeit gibt. Beispiele, in denen man sich eine benutzerdefinierte Gleichheit wünschen würde, gibt es jedoch nicht nur bei der Modellierung algebraischer Strukturen, sondern auch in der Programmierung. So ist eine Queue-Implementierung durch ein Paar von Listen nur durch eine benutzerdefinierte Gleichheit (die Relation lautet in diesem Fall $mkpair(k1, k2) \sim mkpair(l1, l2) :\Leftrightarrow append(reverse(k1), k2) = append(reverse(l1), l2)$) als Instanz der Spezifikation *Queue* beweisbar.

Kapitel 5

Fallstudie zu Reduktionssystemen

Um die Möglichkeiten der implementierten Erweiterungen zu testen und zu demonstrieren, wurde eine Fallstudie durchgeführt, in der Beweise über axiomatisch spezifizierte Strukturen eine entscheidende Rolle spielen. Dabei geht es um sogenannte Reduktionssysteme, die z.B. verwendet werden, um Termersetzungssysteme zu modellieren, wie sie auch in Theorembeweisern wie ✓eriFun vorkommen. Die Fallstudie ist inspiriert durch eine ähnliche Arbeit, die mit dem Theorembeweiser ACL2 durchgeführt wurde [16].

Ein Reduktionssystem R besteht in seiner allgemeinsten Form aus einer nichtleeren Elementmenge E und einer Operatormenge Op, sowie einer partiellen Funktion $reduce : E \times Op \mapsto E$. Für $reduce(x, op) = y$ schreibt man auch $x \rightarrow_R^{op} y$ und sagt, x wird durch op in einem Schritt auf y reduziert. Interessiert man sich nicht für den Operator, schreibt man einfach $x \rightarrow_R y$. Die Relation $\rightarrow_R$ heißt auch *Reduktionsrelation* von R. Ein Element x heißt reduzibel, falls es ein $op \in Op$ gibt, so dass $reduce(x, op)$ definiert ist. Ansonsten heisst x irreduzibel.

Man interessiert sich unter anderem für folgende Eigenschaften von Reduktionssystemen:

Existenz und Eindeutigkeit von Normalformen Eine Normalform von x ist ein irreduzibles $y \in E$, so dass $x \rightarrow_R^* y$.

Konfluenz Ein Reduktionssystem ist konfluent, wenn für die Reduktionsrelation $\rightarrow_R$ gilt: $\leftarrow_R^* \circ \rightarrow_R^* \subseteq \rightarrow_R^* \circ \leftarrow_R^*$. Das bedeutet anschaulich, dass zwei auseinanderlaufende Reduktionspfade auch immer wieder zusammengeführt werden können.

Wir modellieren ein Reduktionssystem in ✓eriFun als Spezifikation:

```
specification Reduction
  domain @E
  domain @OP

  operator legal: @E,@OP -> bool
  operator reduce_one_step: @E,@OP -> @E
  operator reducible: @E -> yields[@OP]

  axiom reducible_implies_legal ⇐ ∀ x:@E
    if(?return(reducible(x)),
       legal(x,result(reducible(x))),
       true)

  axiom not_reducible_implies_not_legal ⇐ ∀ x:@E, op:@OP
    if(?failed(reducible(x)),
       if(legal(x,op),false,true),
       true)
```

Dabei ist der Typoperator *yields* definiert als

```
structure yields[@X] ⇐
  failed,
  return(result:@X)
```

Die Spezifikation enthält als Typen die Elementmenge und die Operatormenge. Die Funktionen *legal* und *reduce_one_step* repräsentieren zusammen die partielle Funktion *reduce*. Ist für ein bestimmtes x und op $reduce(x, op)$ definiert, so liefert $legal(x, op)$ *true* und $reduce\_one\_step(x, op)$ das entsprechende Ergebnis. Ansonsten liefert $legal(x, op)$ den Wert *false* und $reduce\_one\_step(x, op)$ ein beliebiges Ergebnis. Die dritte Funktion *reducible* überprüft, ob ein Element x reduzibel ist und liefert, falls das der Fall ist, einen beliebigen Operator zurück, der x reduziert. Ansonsten liefert die Funktion das Ergebnis *failed*.

Die Axiome beschreiben die Korrektheit und Vollständigkeit von *reducible*, also die Aussage, dass ein Element, für das *reducible* einen Operator zurückliefert, sich auch tatsächlich durch diesen reduzieren lässt, und dass es irreduzibel ist, falls *reducible failed* zurückgibt.

5.1 Äquivalenz und Äquivalenzbeweise

Wir definieren aufbauend auf dem Reduktionssystem eine Äquivalenzrelation $\sim_R$ als die kleinste Äquivalenzrelation, die die Reduktionsreduktion $\rightarrow_R$ enthält. Zwei Elemente sind also äquivalent, wenn es eine Kette von Reduktionsschritten gibt, die beide Elemente miteinander verbindet. Dabei ist es egal, in welcher Richtung die Reduktionsschritte ausgeführt werden.

Diese Äquivalenz soll als Funktionsprozedur (aufbauend auf der Spezifikation *Reduction*) definiert werden, die überprüft, ob zwei Elemente äquivalent sind. In der quantorenfreien Logik von ✓`eriFun` muss die Aussage „Es existiert eine

Kette von Reduktionsschritten“ allerdings immer konstruktiv angegeben werden. Dafür dient der Begriff des *Beweises*. Beweise sind konstruktive Argumente für die Äquivalenz zweier Elemente[17]. Ein Beweisschritt ist ein 4-Tupel (x, y, op, dir). Dabei sind $x, y \in E$ zwei Elemente, $op \in Op$ ein Operator und $dir \in \{forward, backward\}$ die Richtung, in die reduziert wird. Ein Beweisschritt ist *gültig*, falls diese Reduktion wirklich so möglich ist:

```
structure direction ⇐ forward, backward

structure proofstep[@E,@O] ⇐
  mkstep(first:@E, second:@E, dir:direction, used_op:@O)

function valid[R:Reduction](step:proofstep[@E,@OP]):bool ⇐
  case dir(step) of
    forward:
      if legal(first(step),used_op(step))
        then reduce_one_step(first(step),used_op(step))=second(step)
        else false
      end_if
    backward:
      if legal(second(step),used_op(step))
        then reduce_one_step(second(step),used_op(step))=first(step)
        else false
      end_if
  end_case
```

Beweise sind nun schlicht und einfach Listen von Beweisschritten. Damit kann man jetzt die Äquivalenzrelation definieren, die nichts anderes tut, als die Beweisschritte einen nach dem anderen zu überprüfen:

```
function equivalent[R:Reduction]
                    (e1:@E, e2:@E, p:list[proofstep[@E,@OP]]):bool ⇐
  if e1=e2
    then true
    else if ?empty(p)
           then false
           else if e1=first(hd(p))
                  then if valid[R](hd(p))
                         then equivalent[R](second(hd(p)),e2,tl(p))
                         else false
                       end_if
                  else false
                end_if
         end_if
  end_if
```

Als kleines Zwischenziel gilt es nun zu zeigen, dass die Äquivalenzrelation wirklich eine solche ist, dass sie also reflexiv, transitiv und symmetrisch ist. Die ersten

[17] Es besteht selbstverständlich keine Verwechselungsgefahr mit den Beweisen, die wir in ✓eriFun über diese Modellierung durchführen werden.

beiden Aussagen (man beachte auch hier wieder die konstruktive Formulierung) beweist ✓eriFun vollautomatisch:

```
lemma equivalent_reflexive[R:Reduction] ⇐ ∀ x,y:@E
  if(x=y,equivalent[R](x,y,empty),true)


lemma equivalent_transitive[R:Reduction]
      ⇐ ∀ p12,p23:list[proofstep[@E,@OP]], e1,e2,e3:@E
  if(equivalent[R](e1,e2,p12),
    if(equivalent[R](e2,e3,p23),equivalent[R](e1,e3,append(p12,p23)),true),
    true)
```

Für die Symmetrie müssen zunächst Hilfsfunktionen definiert werden, die aus einem Beweis für $x \sim_R y$ einen Beweis für $y \sim_R x$ konstruieren. Die generische Funktion zum Umdrehen von Listen genügt hier nicht, da zusätzlich jeder einzelne Beweisschritt umgekehrt werden muss:

```
function reverse_proofstep(s:proofstep[@X,@Y]):proofstep[@X,@Y] ⇐
  case dir(s) of
    forward: mkstep(second(s),first(s),backward,used_op(s))
    backward: mkstep(second(s),first(s),forward,used_op(s))
  end_case

lemma reverse_proofstep_is_valid[R:Reduction] ⇐ ∀ s:proofstep[@E,@OP]
  if(valid[R](s),valid[R](reverse_proofstep(s)),true)

function reverse_proof(p:list[proofstep[@X,@Y]]):list[proofstep[@X,@Y]] ⇐
  if ?empty(p)
    then empty
    else append(reverse_proof(tl(p)),add(reverse_proofstep(hd(p)),empty))
  end_if

lemma equivalent_symmetric[R:Reduction]
      ⇐ ∀ p:list[proofstep[@E,@OP]], x,y:@E
  if(equivalent[R](x,y,p),equivalent[R](y,x,reverse_proof(p)),true)
```

5.2 Ableitungen und Konfluenz

Um Konfluenz zu formulieren, benötigen wir neben dem Begriff des Äquivalenzbeweises auch den Begriff einer Ableitung, also einer Folge von Schritten, die ein Element x zu einem Element y reduziert. Diese Ableitungen modellieren wir einfach als Listen von Operatoren und definieren, wann eine solche Liste die Ableitung zwischen zwei Elementen ist:

```
function reduces_to[R:Reduction](x:@E, y:@E, p:list[@OP]):bool ⇐
  if ?empty(p)
    then x=y
    else if legal(x,hd(p))
           then reduces_to[R](reduce_one_step(x,hd(p)),y,tl(p))
           else false
         end_if
  end_if
```

Für die Definition von Konfluenz führen wir noch die Datenstruktur *peak* ein. Ein Peak besteht aus drei Elementen x, y und z so dass $x \rightarrow_R^* y$ und $x \rightarrow_R^* z$. Die Ableitungen werden ebenfalls in der Struktur gespeichert. Ein Peak ist gültig, wenn die Ableitungen zu den Elementen passen:

```
structure peak[@E,@OP] ⇐
  mkpeak(left:@E,top:@E,right:@E,leftproof:list[@OP],rightproof:list[@OP])

function peak.valid[R](p:peak[@E,@OP]):bool ⇐
  if reduces_to[R](top(p),left(p),leftproof(p))
    then reduces_to[R](top(p),right(p),rightproof(p))
    else false
  end_if
```

Der Konfluenzbegriff wird nun durch eine weitere Spezifikation definiert, die von *Reduction* erbt. Zusätzlich werden drei Funktionssymbole eingeführt, die für einen peak (x, y, z) ein Element liefern, an dem die Elemente y und z wieder zusammengeführt werden können, sowie die passenden Ableitungen dazu:

```
specification ConfluentReduction[R:Reduction]

  operator left_join: peak[@E,@OP] -> list[@OP]

  operator right_join: peak[@E,@OP] -> list[@OP]

  operator join_point: peak[@E,@OP] -> @E

  axiom left_join_joins ⇐ ∀ pk:peak[@E,@OP]
    if(peak.valid[R](pk),
      reduces_to[R](left(pk),join_point(pk),left_join(pk)),
      true)

  axiom right_join_joins ⇐ ∀ pk:peak[@E,@OP]
    if(peak.valid[R](pk),
      reduces_to[R](right(pk),join_point(pk),right_join(pk)),
      true)
```

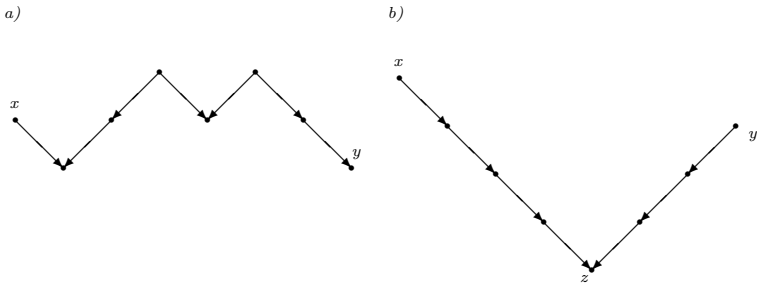

Abbildung 5.1: Äquivalenzbeweis (a) und Valley-Proof (b) für $x \sim_R y$.

5.3 Die Church-Rosser-Eigenschaft

Konfluenz ist äquivalent zur Church-Rosser-Eigenschaft, die besagt, dass für zwei Elemente x, y mit $x \sim_R y$ ein Element z gibt, so dass $x \to_R^* z$ und $y \to_R^* z$, dass sich also zwei äquivalente Elemente immer zusammenführen lassen. Diese Zusammenführbarkeit modellieren wir in Anlehnung an [16] mit sogenannten *Valley-Proofs*: Valley-Proofs bestehen zunächst nur aus Vorwärtsschritten, dann nur aus Rückwärtsschritten (vgl. Abb. 5.1).

In diesem Abschnitt wird die nicht-triviale Richtung dieser Äquivalenz (Konfluenz ⇒ Church-Rosser) gezeigt. Der Beweis stützt sich darauf, dass sich ein Äquivalenzbeweis für zwei Elemente x und y als Folge von endlich vielen lokalen Peaks schreiben lässt. Diese Peaks lassen sich nun mit Hilfe der Konfluenz nacheinander in Täler umwandeln.

Wir definieren, wann Beweise, die aus Listen von Peaks bestehen, als Beweis für die Äquivalenz von Elementen akzeptiert werden. Man beachte, dass ein Großteil der folgenden Definitionen noch auf der allgemeinen Reduktionsspezifikation basieren. Die Konfluenz wird erst später benötigt.

```
function peakproof.eqv[R:Reduction]
                        (x:@E, y:@E, l:list[peak[@E,@OP]]):bool ⇐
  if ?empty(l)
    then x=y
    else if x=left(hd(l))
           then if peak.valid[R](hd(l))
                  then peakproof.eqv[R](right(hd(l)),y,tl(l))
                  else false
                end_if
           else false
         end_if
  end_if
```

Das Ziel ist es nun, eine Funktionsprozedur *peakify* zu schreiben, die einen Beweis in eine Liste von Peaks transformiert. Eine zweite Funktionsprozedur *unpeakify* wandelt diese Peaks unter Ausnutzung der Konfluenz in einen Valley-Proof um.

```
function is_valley_proof(p:list[proofstep[@X,@Y]]):bool ⇐
  if ?empty(p)
    then true
    else if ?empty(tl(p))
           then true
           else if ?forward(dir(hd(tl(p))))
                  then if ?forward(dir(hd(p)))
                         then is_valley_proof(tl(p))
                         else false
                       end_if
                  else is_valley_proof(tl(p))
                end_if
         end_if
  end_if
```

Die Funktion *peakify* und und ihre Hilfsfunktionen sind wie folgt definiert:

```
function can_add_to_peak(pk:peak[@E,@OP], step:proofstep[@E,@OP]):bool ⇐
  if ?forward(dir(step))
    then true
    else ?empty(rightproof(pk))
  end_if

function add_to_peak(pk:peak[@E,@OP], step:proofstep[@E,@OP]):
                    peak[@E,@OP] ⇐
  if ?forward(dir(step))
    then mkpeak(left(pk),
                top(pk),
                second(step),
                leftproof(pk),
                append(rightproof(pk),add(used_op(step),empty)))
    else mkpeak(left(pk),
                second(step),
                second(step),
                add(used_op(step),leftproof(pk)),
                empty)
  end_if

function peakify(l:list[proofstep[@E,@OP]], p:peak[@E,@OP]):
                 list[peak[@E,@OP]] ⇐
  if ?empty(l)
    then add(p,empty)
    else if can_add_to_peak(p,hd(l))
           then peakify(tl(l),add_to_peak(p,hd(l)))
           else add(p,
                    peakify(tl(l),
                            mkpeak(first(hd(l)), second(hd(l)), second(hd(l)),
                                   add(used_op(hd(l)),empty), empty)))
         end_if
  end_if
```

Das zweite Argument p der Funktion *peakify* dient als Akkumulator, in dem Beweisschritte angesammelt werden, die in einem Peak zusammengefasst werden können. Das folgende Lemma beschreibt die Korrektheit der Transformation:

```
lemma peakify_correct[R:Reduction]
      ⇐ ∀ pk:peak[@E,@OP], x,y:@E, p:list[proofstep[@E,@OP]]
  if(peak.valid[R](pk),
    if(x=right(pk),
      if(equivalent[R](x,y,p),
        peakproof.eqv[R](left(pk),y,peakify(p,pk)),
        true),
      true),
    true)
```

Das Gegenstück – die Funktion *unpeakify* – muss natürlich durch das Konfluente Reduktionssystem parametrisiert sein, da sie direkt Gebrauch von der Konfluenz macht:

```
function add_steps_to_peak_left(x:@E, l:list[@OP],
                                p:peak[@E,@OP]):peak[@E,@OP] ⇐
  mkpeak(x,top(p),right(p),append(leftproof(p),l),rightproof(p))

function unpeakify[C:ConfluentReduction]
         (x:@E, prefix:list[@OP], y:@E, l:list[peak[@E,@OP]]):
         list[proofstep[@E,@OP]] ⇐
  if ?empty(l)
    then reverse_proof(mkproof[R(C)](y,prefix))
    else let pk:=add_steps_to_peak_left(x,prefix,hd(l)) in
           append(mkproof[R(C)](x,left_join(pk)),
                  unpeakify[C](join_point(pk),right_join(pk),right(pk),tl(l)))
         end_let
  end_if
```

Mit einigen Hilfslemmata beweist man die Korrektheit von *unpeakify*:

```
lemma unpeakify_correct[C:ConfluentReduction]
      ⇐ ∀ ops:list[@OP],l:list[peak[@E,@OP]],z,y,x:@E
  if(reduces_to[R(C)](y,x,ops),
    if(peakproof.eqv[R(C)](y,z,l),
      equivalent[R(C)](x,z,unpeakify[C](x,ops,y,l)),
      true),
    true)
```

Die Funktionen *peakify* und *unpeakify* kann man dann zu einer Funktion *make_valley_proof* kombinieren, die aus jedem Beweis für $x \sim_R y$ einen passenden Valley Proof macht:

```
function make_valley_proof[C:ConfluentReduction]
         (x:@E, l:list[proofstep[@E,@OP]]):list[proofstep[@E,@OP]] ⇐
  unpeakify[C](x,empty,x,peakify(l,mkpeak(x,x,x,empty,empty)))
```

```
lemma make_valley_proof_correct[C:ConfluentReduction]
      ⇐ ∀ x,y:@E, p:list[proofstep[@E,@OP]]
  if(equivalent[R(C)](x,y,p),
    equivalent[R(C)](x,y,make_valley_proof[C](x,p)),
    true)

lemma make_valley_proof_makes_valley_proof[C:ConfluentReduction]
      ⇐ ∀ x:@E, p:list[proofstep[@E,@OP]]
  is_valley_proof(make_valley_proof[C](x,p))
```

Damit ist die Church-Rosser-Eigenschaft für konfluente Relationen gezeigt.

5.4 Eindeutigkeit von Normalformen

Aufbauend auf der Church-Rosser-Eigenschaft zeigen wir, dass ein Element x höchstens eine Normalform haben kann. Dazu zeigt man zunächst die Aussage, dass zwei irreduzible Elemente x und y, zwischen denen ein Valley-Proof existiert, gleich sein müssen:

```
lemma equivalent_irreducible_are_equal[R:Reduction]
      ⇐ ∀ p:list[proofstep[@E,@OP]], x1,x2:@E
  if(is_valley_proof(p),
    if(?failed(reducible(x1)),
      if(?failed(reducible(x2)),
        if(equivalent[R](x1,x2,p),x1=x2,true),
        true),
      true),
    true)
```

Man beachte, dass dieses Lemma wieder allgemein für alle Reduktionen gilt und auch so formuliert ist. Denn hier ist die Voraussetzung, dass ein Valley-Proof zwischen beiden Elementen existiert, noch explizit enthalten.

Daraus folgt nun mit Hilfe der Church-Rosser-Eigenschaft das Theorem über die Eindeutigkeit der Normalformen: Ist in einem konfluenten Reduktionssystem ein Element x zu zwei irreduziblen Elementen n_1, n_2 äquivalent, dann ist $n_1 = n_2$:

```
lemma normal_forms_unique[C:ConfluentReduction]
      ⇐ ∀ x,n1,n2:@E, p1,p2:list[proofstep[@E,@OP]]
  if(?failed(reducible(n1)),
    if(?failed(reducible(n2)),
      if(equivalent[R(C)](x,n1,p1),
        if(equivalent[R(C)](x,n2,p2),n1=n2,true),
        true),
      true),
    true)
```

Für den Beweis dieser Aussagen wurden 15 weitere Hilfslemmata benötigt. Von insgesamt 156 Regelanwendungen erfolgten 107 automatisch. Damit wurde ein Automatisierungsgrad von 68,6% erreicht.

5.5 Mögliche Fortführungen

Man sieht an dieser Fallstudie, dass sich durch axiomatische Spezifikationen neue Anwendungsfelder für ✓`eriFun` eröffnen. Beweise über allgemeine oder konfluente Reduktionssysteme sind ohne axiomatische Spezifikationen nicht möglich. Das Beispiel schöpft aber noch nicht alle Möglichkeiten des Systems aus. Man könnte den Beweis fortführen, indem man nun eine Instanz für die Spezifikation eines konfluenten Reduktionssystems angibt (z.B. indem man den Beweis des Church-Rosser-Theorems für das Lambda-Kalkül aus [17] überträgt). Das Theorem der eindeutigen Normalform überträgt sich dann auf die Instanz. In [16] werden Termersetzungssysteme modelliert und das Critical-Pair-Theorem bewiesen.

Kapitel 6

Ausblick

Mit der Einführung von Modulen wurde ein aus der Programmierung bekanntes und bewährtes Konzept übernommen, mit dem Programme und Beweise strukturiert werden können. Durch das Verbergen von lokalen Hilfslemmata bei nicht-lokalen Beweisen ist prinzipiell ein Effizienzgewinn bei symbolischen Auswertungen möglich. Leider ist es bisher nicht gelungen, diesen Effekt bei den im Dateiformat der aktuellen ✓`eriFun` -Version vorliegenden Fallstudien wirklich in der Praxis nachzuweisen (vgl. Bsp. 3.2). Wenn größere Fallstudien auf die neue Version übertragen werden, sollte von den Strukturierungsmöglichkeiten Gebrauch gemacht und geprüft werden, inwieweit die Auswertungen dadurch beschleunigt werden.

Durch die Einführung axiomatischer Spezifikationen wird die Modellierungsstärke des Systens deutlich verbessert. Erstmals können nun auch so wichtige Strukturen wie allgemeine Gruppen oder Ordnungen modelliert werden. Die neue Funktionalität wurde allerdings im Rahmen dieser Arbeit nur an vergleichsweise kleinen Beispielen untersucht. Auch die Fallstudie aus Kapitel 5 ist lediglich ein erster Versuch der praktischen Anwendung. Ein Schwerpunkt nachfolgender Arbeiten sollte daher darin liegen, die neuen Funktionen in der Praxis zu erproben und eventuell die bestehenden Probleme (vgl. Abschnitt 4.8) zu beseitigen:

- Die bestehenden ✓`eriFun` -Fallstudien sollten daraufhin untersucht werden, ob sie mit Hilfe der neuen Spracherweiterungen verbessert werden können.
- Durch axiomatische Spezifikationen können neue Fallstudien, z.B. auf dem Gebiet der Algebra oder Logik durchgeführt werden.
- Im Gegensatz zu Aussagen über rekursiv definierte Funktionen sind Aussagen über axiomatisch spezifizierte Strukturen in vielen Fällen *first order*, d.h. durch reines prädikatenlogisches Schliessen und Anwenden von Gleichungen beweisbar. Das System wurde aber bisher daraufhin optimiert, bei Induktionsbeweisen eine möglichst hohe Automatisierung zu bieten. Um nicht-induktive Beweise besser zu automatisieren, müssen andere Verfahren und Heuristiken eingesetzt werden.

- Für das Problem der Generierung instanziierter Lemmata (vgl. Abschnitt 4.4.2) sollte eine Lösung gefunden werden.
- Es sollte untersucht werden, ob es sinnvoll ist, eine gesonderte Gleichheitsbehandlung für axiomatisch spezifizierte Datentypen und deren Instanzen (wie in Abschnitt 4.8.2 beschrieben) zu implementieren.

Durch axiomatische Spezifikationen entwickelt sich √`eriFun` ein Stück weiter von einem spezialisierten Programmverifikationstool hin in Richtung eines universell einsetzbaren Theorembeweisers. Solche Erweiterungen fordern jedoch auch eine höhere Flexibilität von den Heuristiken und voreingestellten Taktiken. Eine große Herausforderung bei der Weiterentwicklung von √`eriFun` wird darin bestehen, die Erweiterungen so zu integrieren, dass der hohe Automatisierungsgrad auch bei einem wesentlich breiteren Anwendungsspektrum erhalten bleibt.

Anhang A

Zur Implementierung

Hier sollen einige wichtige Details zur Implementierung erwähnt werden, die für den Benutzer nicht relevant sind, aber für die Weiterentwicklung des Systems von Interesse sind.

Für die Implementierung der beschriebenen Funktionalität mussten folgende Erweiterungen am System vorgenommen werden.

- Repräsentation der neuen Programmelemente, Strukturparameter und Strukturterme
- Anpassung des Parsers, so dass die neuen Elemente, Sichtbarkeiten, Strukturparameter und Strukturterme eingelesen werden können
- Einführung von lokalen Typvariablen (s. unten)
- Benutzerschnittstelle zum Ändern der Sichtbarkeit, zum Verschieben von Elementen zwischen Modulen und zur Eingabe und benutzerfreundlichen Anzeige der neuen Elemente
- Gültigkeitsbereiche für Symbole und Überprüfen auf Einhaltung der Sichtbarkeitsbeschränkungen
- Kompatibilitätsprüfung und Generierung von funktionalen Substitutionen gemäß Def. 4.18
- Ermitteln der für einen Beweis verfügbaren Funktionen, Axiome und Lemmata und entsprechende Beschränkung des symbolischen Auswerters und der Beweisregeln.
- Generierung der Beweisverpflichtungen für Instanzen
- Management der Abhängigkeiten zwischen Elementen

Wichtige Details aus der Implementierung sollen im folgenden beschrieben werden.

A.1 Repräsentation

A.1.1 Strukturierte Programme

Die Verwaltung der Programmelemente musste so angepasst werden, dass es möglich wurde, hierarchisch gegliederte Programme zu erstellen.

Obwohl Programme in $\mathcal{FP}$ bisher keine innere Struktur hatten, gab es bereits die Möglichkeit, sogenannte Ordner (*Folder*) anzulegen, mit denen das Programm optisch gegliedert werden kann. Bisher wurden diese Ordner aber nicht als echte Programmelemente verwaltet, sondern existierten nur auf der Darstellungsebene. Dem lag die Auffassung zugrunde, dass die Ordner ausschließlich für die Darstellung existieren und daher strenggenommen nicht zum Programm gehören. Dieses Prinzip hat sich in der Praxis aber als sehr umständlich herausgestellt, da in sehr vielen Fällen (z.B. Sicherung der Namenseindeutigkeit, benutzergesteuerte Aktionen, Import von Elementen aus anderen Programmen) umfangreiche Sonderbehandlungen notwendig waren.

Im Zuge der Einführung von Modulen (die einige Eigenschaften mit Ordnern gemein haben) wurde diese Verwaltung umgestellt. Ordner, Module und Spezifikationen sind als sogenannte *Container* Teil des funktionalen Programms, welches dadurch eine hierarchische Gliederung erhält.

A.1.2 Spezifikationen

Die wichtigsten Klassen für die Behandlung von Spezifikationen, Strukturparametern und Instanzen sind im Paket `pmg.logic.fp.specifications` zusammengefasst. Es folgt eine kurze Beschreibung der einzelnen Klassen:

Specification Repräsentiert eine Spezifikation und enthält Methoden, um die entsprechenden Symbole zu verwalten.

Instance Repräsentiert eine Instanz und enthält Methoden, um die Zuordnungen der Symbole zu verwalten.

FunctionalSubstitution Beschreibt eine funktionale Substitution und beinhaltet Methoden, um funktionale Substitutionen auf Typen, Variablen, Terme und Formeln anzuwenden. Wie man in Kapitel 4 gesehen hat, kommen funktionale Substitutionen in vielen Situationen zum Einsatz, wo Ersetzungen von Typ- und Funktionssymbolen beschrieben werden müssen.

StructureParameter Beschreibt einen Strukturparameter und beinhaltet Methoden, um die zum Parameter gehörigen lokalen Symbole und Axiome zu generieren.

Environment Beschreibt eine Umgebung, bestehend aus mehreren Strukturparametern. Bietet Methoden zur Abfrage der verfügbaren lokalen Symbole und Axiome.

StructureExpression Abstraktes Interface für Strukturterme. Dieses Interface wird von drei Klassen implementiert, die genau den drei unterschiedlichen Arten von Strukturtermen aus Def. 4.11 entsprechen:

StructureParameter siehe oben

SubstructureSelection Repräsentiert einen Term der Form $M(G)$, durch den eine Unterstruktur ausgewählt wird

InstanceApplication Repräsentiert eine Instanzanwendung, also eine Instanz, für deren formale Strukturparameter konkrete Strukturen eingesetzt sind.

StructureList Repräsentiert eine Liste von Strukturtermen. Eine solche Liste stellt die aktuellen Strukturparameter bei einem Funktionsaufruf dar. Bietet Methoden, um zu prüfen, ob die Parameterliste mit einer bestimmten Umgebung kompatibel ist, und zur Generierung der entsprechenden funktionalen Substitution.

A.1.3 Terme

Beim Aufruf von Funktionen können zusätzlich zu gewöhnlichen Parametern nun Strukturparameter übergeben werden. Um die Veränderungen an der Repräsentation von Termen so gering wie möglich zu halten, wurden Strukturparameter nicht in die Termrepräsentation selbst integriert. Stattdessen wurden Termannotationen verwendet, wie sie bisher zur Steuerung des symbolischen Auswerters zum Einsatz kamen. Ein Term für einen Funktionsaufruf ist jeweils mit den nötigen Strukturparametern annotiert.

Beim Öffnen eines Funktionsaufrufs muss die Parametrisierung berücksichtigt werden: Der Rumpf der Funktion muss dafür zunächst entsprechend instanziiert werden. Dies geschieht durch Anwendung einer funktionalen Substitution, die aus der Umgebung der Funktion und den aktuellen Strukturparametern gewonnen wird. Die Instanziierung wird zentral in der Klasse *SymAnnotations* vorgenommen, die danach auch dafür zuständig ist, den Funktionsrumpf mit weiteren notwendigen Annotationen zu versehen. Für die Anwendung von Lemmata ist eine solche Behandlung nicht notwendig, da die instanziierten Lemmata vorab erzeugt werden (vgl. Abschnitt 4.4.2).

A.1.4 Lokale Typvariablen

Typsymbole in Spezifikationen und Umgebungen werden durch sogenannte lokale Typvariablen repräsentiert. Diese sind im Gegensatz zu anderen Typvariablen nicht durch ihren Namen eindeutig beschrieben, da es ja ohne weiteres vorkommen kann, dass verschiedene Spezifikationen jeweils Typvariablen desselben Namens einführen. Um eine reibungslose Verwaltung zu gewährleisten, muss es sich hier auch intern um verschiedene Objekte handeln. Das wäre mit herkömmlichen Typvariablen aufgrund der Implementierung durch eine zentrale Verwaltung, die den Namen als eindeutigen Schlüssel verwendet, nicht möglich. Lokale Typvariablen aus der aktuellen Umgebung gelten als „fest“ und können nicht bei der Typunifikation mit konkreten Werten belegt werden.

A.2 Abhängigkeiten zwischen Elementen

Um Änderungen am Programm konsistent durchführen zu können, verwaltet ✓eriFun die Abhängigkeiten zwischen Programmelementen. Im Prinzip gibt es dabei zwei verschiedene Arten von Abhängigkeit. Ein Element E_1 ist von einem anderen Element E_2 abhängig, wenn einer der folgenden Fälle zutrifft:

1. In der Definition von E_1 wird auf das Element E_2 zugegriffen (z.B. durch den Aufruf einer Funktion). Um die Definition von E_1 zu parsen, muss also E_2 bereits im System vorhanden sein. Das hat unter anderem folgende Konsequenzen:
   - E_2 kann nicht gelöscht oder modifiziert werden, ohne das E_1 vorher gelöscht wird.
   - Beim Speichern im FP-Format muss E_2 vor E_1 geschrieben werden, um sicherzustellen, dass die Datei wieder korrekt eingelesen werden kann.
2. Der Zustand[18] von E_1 ist vom Zustand von E_2 abhängig und muss neu berechnet werden, wenn sich der Zustand von E_2 ändert.

Die erste Form der Abhängigkeit ist in der Regel schon durch die Definition bedingt, während die zweite meist dadurch zustande kommt, dass E_2 im Beweis von E_1 benutzt wird.

In der bisherigen Implementierung wurden diese beiden Arten von Abhängigkeit nicht unterschieden, da keine Notwendigkeit bestand, sie zu unterscheiden. Das würde aber bei den Spezifikationen nun zu einer zyklischen Abhängigkeit führen, denn für eine Spezifikation S und ein Element E in S gilt:

- E ist von S abhängig, da bei seiner Definition die Spezifikation referenziert wird.
- S ist von E abhängig, da eine Spezifikation genau dann den Status *verified* hat, wenn alle ihre Elemente diesen Status haben.

Um zyklische Abhängigkeiten zu vermeiden, muss man die beiden Arten von Abhängigkeiten auch getrennt behandeln. Es wird also in der neuen Implementierung zwischen *Definitions-* und *Zustandsabhängigkeit* unterschieden. Für alle bisherigen Elemente fallen diese beiden Abhängigkeiten zusammen. Für die neuen Elemente gilt:

Module Ein Modul ist von keinem Element abhängig. Elemente im Modul sind von diesem definitionsabhängig.

Spezifikationen Elemente in einer Spezifikation sind stets von dieser definitionsabhängig. Umgekehrt ist eine Spezifikation von ihrem Inhalt zustandsabhängig. Weiterhin ist eine Spezifikation von allen ihren Instanzen und

[18] Jedes Element hat einen internen Zustand, der beispielsweise automatisch generierte Beweisverpflichtungen beinhaltet und vom Zustand anderer Elemente abhängen kann.

allen von ihr erbenden Spezifikationen zustandsabhängig (um die Konsistenz prüfen zu können).

Domaindeklarationen Domaindeklarationen sind lediglich von der Spezifikation definitionsabhängig.

Operatoren Ein Operator ist von allen verwendeten Typsymbolen (Datenstrukturen, Domains) definitionsabhängig.

Axiome Ein Axiom ist von allen verwendeten Typ- und Funktionssymbolen definitionsabhängig.

Instanzen Eine Instanz ist von der zugehörigen Spezifikation und den verwendeten Funktionen und Datenstrukturen definitionsabhängig.

Instanziierte Axiome Die durch eine Instanz generierten Beweisverpflichtungen sind von der Instanz definitionsabhängig. Die Instanz ist von diesen Beweisverpflichtungen zustandsabhängig.

Allgemein Elemente mit Strukturparametern sind von den darin vorkommenden Spezifikationen sowie allen in der Spezifikation definierten Elementen definitionsabhängig. So ist sichergestellt, dass nicht einzelne Elemente aus einer bereits benutzten Spezifikation gelöscht werden können.

Literaturverzeichnis

[1] Auer, J. *Erweiterung des ✓eriFun-Systems um Funktionen höherer Ordnung.* Programmiermethodik, Technische Universität Darmstadt, 2005. Diplomarbeit.

[2] Black, P. E. Dictionary of algorithms and data structures. http://www.nist.gov/dads/.

[3] Boyer, R. S., Goldschag, D. M., Kaufman, M., and Moore, J. S. Functional instantiation in first-order logic. In *Artificial Intelligence and Mathematical Theory of Computation*, V. Lifschitz, Ed. Academic Press, San Diego, 1991, pp. 7–26.

[4] Cardelli, L., and Wegner, P. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys 17*, 4 (Dec. 1985), 471–522.

[5] Damas, L., and Milner, R. Principal type schemes for functional programs. In *Proceedings of the ACM Conference on Principles of Programming Languages* (1982), pp. 207–212.

[6] Dijkstra, E. W. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[7] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, Massachusetts, 1995.

[8] Hall, C. V., Hammond, K., Jones, J., and Wadler, P. L. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems 18*, 2 (Mar. 1996), 109–138.

[9] Harper, R. *Programming in Standard ML.* To Appear.

[10] Jones, M. P. Type classes with functional dependencies. In *Proc. 9th European Symp. on Prog. (ESOP 2000)* (Mar. 2000), vol. 1782 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 230–244.

[11] Jones, S. P., Ed. *Haskell 98 Language and Libraries, the Revised Report.* Cambridge University Press, Apr. 2003.

[12] Kaufmann, M., Manolios, P., and Moore, J. S. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, June 2000.

[13] Milner, R., Tofte, M., and Harper, R. *The Definition of Standard ML.* MIT Press, Aug. 1990.

[14] Nipkow, T., Paulson, L. C., and Wenzel, M. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, vol. 2283 of *Lecture Notes in Computer Science*. Springer Verlag, 2002.

[15] Paulson, L. C. *Isabelle: A Generic Theorem Prover*, vol. 828 of *Lecture Notes in Computer Science*. Springer Verlag, 1994.

[16] Ruiz-Reina, J.-L., Alonso, J.-A., Hidalgo, M.-J., and Martín-Mateos, F.-J. Formalizing rewriting in the ACL2 theorem prover. In *Artificial Intelligence and Symbolic Computation, International Conference 2000* (2001), vol. 1930 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 92–106.

[17] Shankar, N. A mechanical proof of the Church-Rosser theorem. *Journal of the Association for Computing Machinery 35*, 3 (1988), 475–522.

[18] Wadler, P., and Blott, S. How to make ad-hoc polymorphism less ad hoc. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1989), ACM Press, pp. 60–76.

[19] Walther, C. On proving the termination of algorithms by machine. *Artificial Intelligence 71*, 1 (1994), 101–157.

[20] Walther, C., and Schweitzer, S. *The √veriFun Tutorial*. Programmiermethodik, Technische Universität Darmstadt, Nov. 2002. Technical Report VFR 02/04.

[21] Walther, C., and Schweitzer, S. *√veriFun User Guide*. Programmiermethodik, Technische Universität Darmstadt, Oct. 2002. Technical Report VFR 02/01.

[22] Walther, C., and Schweitzer, S. About √veriFun . In *CADE-19, 19th International Conference on Automated Deduction* (2003), vol. 2741 of *Lecture Notes in Computer Science*, Springer, pp. 322–327.

[23] Walther, C., and Schweitzer, S. *A Machine Supported Proof of the Unique Prime Factorization Theorem*. Programmiermethodik, Technische Universität Darmstadt, 2003. Technical Report VFR 02/03.

[24] Walther, C., and Schweitzer, S. *A Verification of Binary Search*. Programmiermethodik, Technische Universität Darmstadt, 2003. Technical Report VFR 02/02.

[25] Walther, C., and Schweitzer, S. Verification in the classroom. *Journal of Automated Reasoning 32*, 1 (2004), 35–73.

[26] Walther, C., and Schweitzer, S. Automated termination analysis for incompletely defined programs. In *Proc. of the 11th Inter. Conf. on Logic for Programming, Artificial Intelligence and Reasoning (LPAR-11)* (2005), F. Baader and A. Voronkov, Eds., vol. 3452 of *Lecture Notes in Artificial Intelligence*, Springer Verlag, pp. 332–346.

[27] Wenzel, M. Using axiomatic type classes in Isabelle, May 2004. Part of the Isabelle system documentation.