Fachbereich 3
Mathematik - Informatik
Universität Bremen

Monadische Hoare-Logik in Isabelle

Diplomarbeit von
Tina Kraußer

tina@krausser.net

Betreut von
Lutz Schröder und Till Mossakowski

September 2005

Ehrenwörtliches Erklärung

Ich versichere, dass ich die vorliegende Arbeit, für die ich als Verfasser genannt werde, selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen sind, wurden als solche kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

Bremen, der 6. September 2005,

(Tina Kraußer)

There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.

C.A.R. Hoare

Inhaltsverzeichnis

1. Motivation und Ziele

In immer mehr Bereichen werden Softwaresysteme zur Unterstützung des Menschen eingesetzt. Sie übernehmen dabei immer komplexere Aufgaben. Es ist jedoch gerade bei umfangreichen Programmen nicht immer ganz einfach, die korrekte Arbeitsweise zu garantieren.

So hat 2001 ein Softwarefehler im Online-System der Berliner Sparkassen und der Berliner Bank die Bankdaten der Kunden für jeden zugänglich gemacht [Kur01], nachdem eine falsche Kapazitätsplanung eine Kettenreaktion ausgelöst hatte.

2002 führte ebenfalls ein Softwarefehler dazu, dass beim 7er BMW die Benzin-Einspritzpumpe falsche Daten erhielt und dadurch die Kraftstoffzufuhr zu gering ausfiel [Vog02]. Etwa 15000 Fahrzeuge mussten zum Software-Update in die Werkstatt.

Im selben Jahr blieben die Nordstaaten der USA sowie Teile Kanadas für mehrere Tage ohne Strom. Die Ursache war ein Fehler im Überwachungs- und Steuerungssystem der Firma FirstEnergy [Bac04]. Dieses war durch eine ungünstige Kombination von Ereignissen so überfordert, dass es auftretende Alarme nicht mehr an das zuständige Personal weiterleiten konnte. Die Symptome konnten zwar nach fünf Tagen behoben werden. Die Fehlersuche dauerte jedoch Wochen.

Dies sind nur einige Beispiele, in denen fehlerhafte Systeme zu erheblichen finanziellen Schäden und Gefährdung von Personen geführt hat.

Gerade bei routinierten Programmierern werden komplexe Programmelemente oft durch abkürzende Schreibweisen für den Außenstehenden in nicht nachzuvollziehender Weise formuliert. Die Dokumentation des entsprechenden Codes fällt auch all zu häufig der angeblichen Intuitivität zum Opfer. Wie alle Menschen sind aber auch Programmieren nicht unfehlbar.

Herkömmliche Methoden zur Fehlererkennung wie Testreihen werden dabei den Sicherheitsanforderungen nicht mehr gerecht. Sie können lediglich die Anwesenheit von Fehlern zeigen, ihre Abwesenheit aber nicht nachweisen.

C.A.R. Hoare besser bekannt als Tony Hoare hat eine formale Methode entwickelt, die Fehler in komplexeren Programmen durch Beweisen auszuschließen. Die 1969 von Hoare veröffentlichte und nach ihm benannte Hoare-Logik [Hoa69] basiert auf so genannten Hoare-Tripeln $\{\!|\ \Phi\ |\!\}\ x \leftarrow p\ \{\!|\ \Psi x\ |\!\}$. Diese beschreiben welchen Nachfolgezustand Ψx ein System nach der Abarbeitung des Programms x$\leftarrow$ p hat, wenn es sich zuvor im Zustand Φ befand.

Das in [Hoa69] vorgestellte Kalkül gibt Regeln für die Herleitungen eines Korrektheitsbeweises für alle elementaren Bestandteile imperativer Programme an. In [SM03] haben Lutz Schröder und Till Mossakowski das Kalkül für einen Bereich der funktionalen Programmierung, den so genannten Monaden, umgesetzt. Diese Arbeit beschäftigt sich mit der Implementierung dieses Kalküls im Beweissystem *Isabelle*.

1.1. Zielsetzung

Neben der Umsetzung von Monaden in Isabelle sollen die monadischen Eigenschaften der *Seiteneffektfreiheit*, *Kopierbarkeit* und *Vertauschbarkeit*, sowie die *deterministische Seiteneffektfreiheit* implementiert werden. Diese bilden die Grundlage für die Isabelle-Umsetzung und den formalen Beweis des Kalküls aus [SM03]. Anhand eines konkreten Beispiels soll abschließend die Verwendbarkeit für komplexe Software-Systeme untersucht werden.

1.2. Gliederung der Arbeit

Nach diesem einleitenden Kapitel werden in den nächsten beiden Kapiteln die theoretischen Grundlagen erörtet. In **Kapitel 2** beleuchten wir zunächst die Probleme, die im Zusammenhang mit *Seiteneffekten* in funktionalen Sprachen auftreten und stellen mögliche Lösungsansätze vor. *Monaden* werden dabei als ein sehr vielversprechender Ansatz genauer betrachtet. Zu dem werden für die weitere Entwicklung der Arbeit benötigte Eigenschaften vorgestellt.

In **Kapitel 3** zeigen wir eine mögliche Herangehensweise an das in [SM03] vorgestellte monadische Hoare-Kalkül. Das Kalkül wird bis auf die Regel für Schleifen auf Grundlage der im ersten Kapitel entwickelten Lemmata bewiesen.

Kapitel 4 liefert eine Einführung in den Theorembeweiser *Isabelle* und stellt die Grundlagen für das Verständnis der *Isabelle*-Beweise zur Verfügung.

Die Umsetzung der monadischen Hoare-Logik und des entsprechenden Kalküls sind Thema des **Kapitel 5**. Anschließend wird in **Kapitel 6** der von Hoare [Hoa69] vorgestellte Korrektheitsbeweis mit Hilfe der implementierten monadischen Hoare-Logik umgesetzt und bewiesen.

Die abschließende Betrachtung der durch die *Isabelle*-Umsetzung entstandenen Möglichkeiten und Ansätze zu Erweiterungen werden in **Kapitel 7** beschrieben.

1.3. Konventionen

Die dieser Arbeit zu Grunde liegenden Konzepte werden schrittweise eingeführt und bedürfen wenige Vorkenntnisse zum Verständnis.

Kenntnisse in der funktionalen Sprache Haskell sind beim Verständnis der Beispiele von Vorteil, da sich diese an der Haskell-Syntax orientieren. Eine umfangreiche Dokumentation findet sich unter [Jon03].

Ebenso wird von grundlegenden Kenntnissen im Bereich der Logik ausgegangen. Das Buch [And02] bietet eine gute Einführung in diesem Gebiet.

Die folgenden Begriffe werden im Weiteren in der hier beschriebenen Weise verwendet sofern es nicht explizit anders angegeben ist:

Korrektheit: Die Korrektheit von Programmen bezieht sich jeweils auf die dem Programm zugehörige Spezifikation. Im Zusammenhang mit der Hoare-Logik beziehen wir uns zudem immer auf die partielle Korrektheit. Diese stellt im Gegensatz zur totalen Korrektheit noch keine Überprüfung der Terminierung zur Verfügung.

Theorem/Lemma: Isabelle verwendet die beiden Begriffe als Synonyme. In dieser Arbeit werden Regeln, die für die Umsetzung des monadischen Hoare-Kalküls von Bedeutung sind als *Lemmata* bezeichnet, Beispiele als *Theoreme*.

Sequenzen der Länge n: Die Länge von Programmsequenzen der Form `x←p; y←q;...` bezieht sich auf die Anzahl der Programmschritte.

In den Grundlagen-Kapiteln werden bei den Lemmata in Klammern die Namen der entsprechenden *Isabelle*-Implementation mit angegeben. Sie dienen zur Referenzierung im Umsetzungs-Kapitel.

Viele der in den Grundlagen-Kapiteln eingeführten Regeln und Beweise sind konzeptuell der Vorgehensweise aus [SM03] entnommen. Da sie aber für die Beweise des Hoare-Kalküls die Grundlage bilden, werden sie an dieser Stelle eingeführt und ihre Herleitung skizzenhaft dargestellt. Die Beweise orientieren sich dabei an der in der Isabelle umgesetzten Version.

Die Darstellung der Lemmata lehnt sich an die aus der Logik bekannten Schreibweise für Beweisregeln an:

$$\frac{\begin{array}{c}A\\B\end{array}}{C}$$

Dabei werden alle Bedingungen (Prämissen) oberhalb des Strichs angegeben. Darunter befindet sich die Schlussfolgerung (Konklusion), die daraus gezogen werden kann.

2. Seiteneffekte in funktionalen Sprachen

Programmiersprachen lassen sich anhand der umgesetzten Paradigmen in verschiedene Gruppen einteilen. Die allgemein bekannten Sprachen gehören der Gruppe der imperativen (Pascal, C) oder objektorientierten (Java, C++) Sprachen an. Zwei vor allem für mathematische Berechnungen und die Entwicklung von Algorithmen interessante Sprachgruppen sind die der funktionalen (Haskell, ML) und logischen (Prolog) Sprachen.

Während sich der imperative Ansatz auf die Ausführung von Befehlen konzentriert, haben diese Sprachen die Auswertung von Ausdrücken als Grundlage. Funktionale Sprachen setzen dabei, wie der Name schon propagiert, auf eine Programmdarstellung, die an mathematische Funktionen erinnert.

Funktionen bzw. *Konstanten höherer Ordnung* zeichnen sich vor allem dadurch aus, dass sie bei gleichen Parametern immer ein konstantes Ergebnis liefern. Die strikte Verwendung von Funktionen als Grundlage der Programmierung verzichtet auf die Anwendung der so genannten *Seiteneffekte*. Dadurch entsteht Quellcode, dessen einzelne Funktionen separat verständlich und wartbar sind.

Dieses Kapitel bietet einen Einblick, welche Einschränkungen man durch den Verzicht auf *Seiteneffekte* hat und welche Möglichkeiten sich bieten diese in funktionale Sprachen zu integrieren.

2.1. Seiteneffekte in der Programmierung

Das folgende funktionale Programm `divFun` zeigt beispielhaft zwei Probleme, die durch eine strikte Umsetzung funktionaler Programmierung auftreten können. Die Funktion beschreibt die Division zweier natürlicher Zahlen (`Nat`) ohne Rest:

```
divFun :: Nat -> Nat -> Nat
divFun x y = IF (x>y) THEN (1+divFun (x-y) y) ELSE (res 0)
```

Die im ELSE-Fall verwendete Funktion `res` soll zunächst den Parameter den es bekommt unverändert zurückliefern. Wie in der Mathematik wird das Ergebnis einer Funktion berechnet, indem man die Teilterme auswertet und dann die Funktionsdefinition anwendet. Bei `divFun 7 3` geht man wie folgt vor:

```
divFun 7 3
--> IF (7>3) THEN (1+divFun (7-3) 3) ELSE (res 0)
--> 1+IF (4>3) THEN (1+divFun (4-3) 3) ELSE (res 0)
--> 1+1+IF (1>3) THEN (1+divFun (1-3) 3) ELSE (res 0)
--> 1+1+0
--> 2
```

Wir erhalten als Ergebnis 2. Die ersten Probleme dieser Umsetzung treten bei der Division durch Null auf. So wird zum Beispiel beim Aufruf `divFun 7 0` während der Termauswertung unendlich tief in die Rekursion eingestiegen.

```
divFun 7 0
--> IF (7>0) THEN (1+divFun (7-0) 0) ELSE (res 0)
--> 1+IF (7>0) THEN (1+divFun (7-0) 0) ELSE (res 0)
--> 1+1+IF (7>0) THEN (1+divFun (7-0) 0) ELSE (res 0)
...
```

Man muss die Funktion um eine Regel für diesen Spezialfall erweitern.

```
divFun :: Nat -> Nat -> Nat
divFun x 0 = ???
divFun x y = IF (x>y) THEN (1+divFun (x-y) y) ELSE (res 0)
```

Bei dieser Modifikation ergibt sich jedoch das Problem, dass ein Rückgabewert für die Division durch Null festgelegt werden muss[1]. Es kommen zunächst zwei Möglichkeiten in Frage:

- eine festgelegte Konstante `err` aus Nat: diese Variante entfällt, da sich `err` nicht von einem regulären Ergebnis unterscheiden läßt
- Erweiterung des Rückgabe-Typs: man kann `divFun` den erweiterten Typ `Nat-> Nat -> Int` zuweisen und negative Rückgabewerte als Fehlerwerte definieren. `Int` beschreibt den Bereich der ganzen Zahlen. Dies verschiebt das Problem jedoch nur. Bei einer Erweiterung der Funktion auf Ganzzahl-Division steht man wieder vor dem gleichen Problem.

Nicht nur die Fehlerbehandlung bringt in rein funktionalen Sprachen Probleme mit sich. Auch bei der Erstellung interaktiver Programme, das heißt Ein- und Ausgabe während des Programmablaufes, gerät man an die Grenzen des Machbaren.

Die Ursache liegt im Fehlen globaler Variablen bzw. eines global Zustandes. Jede Funktion kennt nur ihren persönlichen lokalen Variablenraum. Dadurch ist es in der Funktion `divFun` zum Beispiel nicht möglich, neben der Publikation des eigentlichen Ergebnisses, als so genannten *Seiteneffekt* einen externen Fehlerzustand zu setzen.

[1] In Java würde dieses Problem eine *java.math.ArithmeticException* auslösen.

2.1.1. imperative Behandlung von Seiteneffekten

Zunächst werden an Hand der Prozedur[2] `divImp` einige Vor- und Nachteile von seiteneffektbehafteter Programmierung am konkreten Beispiel aufgezeigt. `divImp` beschreibt dabei die imperative, um Fehlerbehandlung erweiterte Version der im vorangegangenen Abschnitt eingeführten Funktion `divFun`.

```
1 PROCEDURE divImp (x,y: Nat){
2   IF (y==0) THEN
3     err := "division by zero"
4   ELSE
5     IF (x>y) THEN
6       z := 1 + divImp (x-y,y)
7     ELSE
8       z := 0
9   return z
10 }
```

Bei dieser Umsetzung wird überprüft, ob der Divisor Null ist (Zeile 2). Ist dies der Fall, wird der globalen Variable `err` eine Fehlermeldung übergeben (Zeile 3). Diese kann dann nach dem Aufruft der Prozedur ausgelesen und ausgewertet werden. Für den Fall, dass keine Division durch Null vorliegt, wird vorgegangen wie in `divFun` (Zeile 5-8). Zum Schluss wird der aktuelle Wert von `z` zurückgegeben.

Besonderes Interesse gilt den beiden globalen Variablen `z` und `err`. Das erste Problem, dass in Verbindung mit diesen Variablen steht, zeigt sich in Zeile 9. Sollte eine Division durch Null vorliegen, ist der Wert von `z` abhängig von der Umgebung in der die Prozedur aufgerufen wird. Im Code-Fragment

```
z := 10;
a := divImp (10,0);
z := 1;
b := divImp(10,0);
```

haben `a` und `b` verschiedene Werte, obwohl sie den gleichen Prozeduraufruf zugewiesen bekommen haben. `a` hat den Wert 10 und `b` den Wert 1. Eine Prozedur, die lesend auf den globalen Zustand einer Variablen zugreift, ist ohne seinen aufrufenden Kontext nicht eindeutig zu verstehen. Ein weiteres Problem zeigt sich bei folgendem Aufruf von `divImp`:

```
y := 7;
z := 3;
err := " ";
```

[2] Da Seiteneffekte dem Konzept der Funktionen widersprechen, verwenden wir für die imperative Umsetzung den Begriff Prozedur.

```
4 a := divImp (y, z);
5 IF (err == " ") THEN
6   print y ++ "/"++ z ++ -"++ a
7 ELSE
8   print err
```

Zunächst werden die Variablen in den ersten beiden Zeilen `y` und `z` gesetzt und `err` initialisiert mit der leeren Zeichenkette (Zeile 3). Nach Aufruf der Division (Zeile 4) wird überprüft, dass `divImp` fehlerlos abgearbeitet wurde (Zeile 5). In diesem Fall wird die mathematische Schreibweise der berechneten Funktion auf dem Bildschirm ausgegeben (Zeile 6). Ansonsten wird die Fehlermeldung angezeigt (Zeile 8).

Man erwartet nun, dass nach Abarbeitung dieses Code-Fragments `7/3=2` auf dem Bildschirm steht. Da `divImp` auf die globale Variable `z` schreibend zugreift, wird diese während der Berechnung verändert. Auf dem Bildschirm erscheint `7/2=2`.

Rein funktionale Sprachen vermeiden diese Probleme durch die strikte Umsetzung der folgenden zwei Programmier-Paradigmen:

referenzielle Transparenz
: Die Auswertung von Funktionsaufrufen ist unabhängig von dem sie umgebenden Kontext. Das heißt, Werte können nur durch explizite Parameter an die Funktion übergeben werden. Auf globale „Variablen" kann in der Funktions-Auswertung nicht zugegriffen werden.

Seiteneffektfreiheit
: Ergebnisse können nur durch explizite Rückgabe publiziert werden. Auf globale „Variablen" kann nicht schreibend zugegriffen werden.

Das verwendete Beispiel zeigt einen entscheidenden Vorteil bei der Verwendung von globalen Variablen auf. Mit Hilfe der Variable `err` kann problemlos eine Fehlermeldung publiziert werden.

2.1.2. Probleme bei Seiteneffekten in funktionalen Sprachen

In `divFun` lassen sich nur Informationen vom Typ `Nat` nach außen weitergeben. Mit einigen kleinen Änderungen ließe sich auch `divFun` dazu bringen, eine Fehlermeldung zu publizieren. Dazu muss der Ergebnistyp zu einem Tupel erweitert werden. Auch der Zugriff auf weiter Werte wie `z` aus der imperativen Version `divImp` lässt sich simulieren durch eine erweiterte Parameterliste.

```
divFun_Tup :: Nat -> Nat -> Ex -> ... -> (Nat × Ex × ...)
divFun_Tup x 0 err ... = (res 0, err++"division by zero")
divFun_Tup x y err ... = (divFun x y, err++" ")
```

Ein Funktionsaufruf, der die Fehlerbehandlung aufgreift sieht dann wie folgt aus:

```
let (val,err, ...) = divFun_Tup (7,3," ", ...) in
  IF (err == " ") THEN
    print "7/3 ="++ val
  ELSE
    print err
```

Diese Methode führt zu unübersichtlichen Programmen. Zu dem muss jeder Funktionsaufruf bei einer Erweiterung der Funktionalität manuell angepasst werden. Sollen die übergebenen Parameter wieder mit zurückgegeben werden, muss neben der Anpassung des Rückgabe-Tupels im Funktions-Rumpf auch das Tupel erweitert werden, an das der `let`-Ausdruck das Ergebnis der Funktion bindet.

Das zweite Problem das in Verbindung mit Seiteneffekten in funktionalen Sprachen auftritt, hängt mit den verwendeten Auswertungs-Strategien zusammen. Um es sichtbar zu machen, erweitern wir die Funktion `res`. Sie soll weiterhin den Parameter zurückliefern und zusätzlich den Benutzer durch die Ausgabe „ready with computation“ darauf aufmerksam machen, dass die Berechnung abgeschlossen ist.

Für den Term `divFun 7 3` sind zwei grundlegende Ansätze zur Auswertung denkbar. In der vorgestellten Auswertung wird das so genannte *call-by-name* verwendet. Dabei werden Teilterme erst dann ausgewertet, wenn sie wirklich für die Berechnung benötigt werden. Insbesondere wird der ELSE-Fall mit dem Aufruf von `res 0` nur beim letzten Funktionsaufruf ausgewertet. Es erscheint wie geplant erst dann ein „ready with computation“ auf dem Bildschirm, wenn das Ende der Berechnung erreicht ist.

Es gibt jedoch auch funktionale Sprachen, die das so genannte *call-by-value* als Strategie verwenden. Dabei werden zunächst alle Teilterme ausgewertet bevor sie in die Funktionsanwendung eingesetzt werden. Schon die ersten Auswertungs-Schritte für `divFun 7 3` zeigen, welches Problem dabei auftritt. Im folgenden Auswertungsbeispiel sei `val (s)` eine Notation dafür, dass der Teilterm s noch ausgewertet werden muss, bevor mit der Gesamtauswertung fortgefahren werden kann. Desweiteren beschreibt `print; 0` die Ausgabe von „ready with computation“ auf dem Bildschirm und die Rückgabe des Wertes `0`, wie wir sie für `res 0` vorgesehen haben. Zu dem wird davon ausgegangen, dass eine Subtraktion auf natürlichen Zahlen, die ein negatives Ergebnis zur Folge hätte, zu `0` auswertet um mit dem Ergebnis im Bereich der natürlichen Zahlen zu bleiben.

Der Term `divFun 7 3` wertet dann wie folgt aus:

$\underbrace{\texttt{divFun 7 3}}$

```
IF (True) THEN (val (1+ val(divFun 4 3)) ELSE (print; 0))

IF (True) THEN (val (1+ val(divFun 1 3)) ELSE (print; 0))

IF (False) THEN (val (1+ val(divFun 0 3)) ELSE (print; 0))
...
```

Das Problem bei der Auswertung des Terms liegt darin, dass bei jedem rekursiven Aufruf der Funktion eine Bildschirmausgabe erfolgt. Dies entspricht nicht unseren Erwartungen, denn durch die Ausgabe sollte der Benutzer darüber informiert werden, dass die Berechnung abgeschlossen ist.

Nun kann man argumentieren, dass die *call-by-name* Strategie die Lösung des Problems ist. Doch auch sie führt unter Umständen zu unerwartetem Verhalten in Zusammenhang mit Seiteneffekten. [Jon01] präsentiert ein einfaches Beispiel, das dies verdeutlicht.

Beispiel 2.1. *Sei `printChar` eine Funktion, die als Seiteneffekt ihren Parameter auf dem Bildschirm ausgibt. Angenommen wir haben eine Liste der Form:*

xs = [printChar ’a’, printChar ’b’]

Die call-by-value-Auswertung hat beim Auftreten von `xs` die Ausgabe a b zur Folge. Bei Anwendung von call-by-name wird nur das ausgewertet, was für die Berechnung benötigt wird. Ist zum Beispiel `length xs` die einzige Anwendung der Liste, so erfolgt keine Ausgabe, da `length` nur die Elemente der Liste zählt und den Inhalt unberührt lässt.

Es fehlt bei beiden Strategien die Möglichkeit die Auswertungs-Reihenfolge festzulegen. Das in Haskell verwendete *call-by-need* bringt ähnliche Probleme mit sich. Diese auch als `lazy` bezeichnete Variante lässt sich am einfachsten als *call-by-value* mit Speicherung schon ausgewerteter Terme beschreiben. Sie wertet `res 0` nur einmal aus, es wird also auch nur einmal eine Bildschirmausgabe erfolgen. Jedoch tauchen bei der Liste in Beispiel 2.1 die gleichen Probleme auf wie bei *call-by-name*.

Selbst wenn man sich sicher ist, dass die Anzahl der Auswertungen den Vorstellungen entspricht, kann die Reihenfolge noch ein Problem beinhalten. Eventuell wird erst der ELSE-Fall ausgewertet, dann würde schon ganz am Anfang der Berechnung der Benutzer darüber informiert, dass die Auswertung zum Ende gekommen ist.

Alle diese Beispiele zeigen, dass bei der Verwendung von Seiteneffekten in funktionalen Sprachen Probleme auftreten. Die Paradigmen der referenzielle Transparenz und Seiteneffektfreiheit führen dazu, dass man bei der Modellierung von Seiteneffekten mit Tupeln als Funktions-Parametern arbeiten muss, die unterschiedlichen

Auswertungs-Strategien führen zum Teil zu unerwarteten Ergebnissen. Wir benötigen daher einen Typ, der Seiteneffekte kapselt und somit für die reine funktionale Programmierung zur Verfügung stellt. Zu dem brauchen wir Funktionen auf diesem Typ, die die Festlegung der Auswertungsreihenfolge ermöglichen.

1991 hat Moggi in [Mog91] Monaden als eine elegante Möglichkeit vorgestellt, Seiteneffekte in funktionale Sprachen zu integrieren.

2.2. Monadischer Ansatz für Seiteneffekte in funktionalen Sprachen

Ursprünglich kommen Monaden aus dem Bereich der Kategorien-Theorie. Sie lassen sich jedoch auch ohne detaillierte mathematische Kenntnisse aus diesem Bereich verstehen.

Eine Monade besteht zunächst aus einem Typkonstruktor `T`, der dem gewünschten Effekt wie Fehlerbehandlung oder der Ein- bzw. Ausgabe entspricht. Der dadurch definierte polymorphe Typ `'a T` ermöglicht es, den Effekt auf unterschiedliche Parameter anzuwenden. So kann für die Fehlerbehandlung bei der Division eine Instanz vom Typ `Nat T` verwendet werden.

Alle Funktionen vom Typ `'a →'b` erhalten den neuen Typ `'a →'b T` und bilden somit in den Bereich der Berrechnungen von Typ `'b` ab. Die Funktion `divFun` hat dadurch den neue Typ `Nat -> Nat -> Nat T`. Außerdem werden für die Verwendung von Monaden zwei Funktionen benötigt. Eine, die einen Wert vom Typ `'a` in eine Monade `'a T` liftet:

```
ret :: 'a →'a T
```

und eine zweite, um Funktionen vom Typ `'a →'b T` auf eine Berechnung vom Typ `'a T` anzuwenden. Dazu wird ein Bindungs-Operator $\gg=$ (gesprochen bind) definiert:

```
≫=:: 'a T →('a →'b T) →'b T
```

Definition 2.1. *Sei `T` ein Typkonstruktor, $\gg=$ und `ret` Funktionen wie oben beschrieben, dann ist eine **Monade** ein Tripel (`T, ret,` $\gg=$).*

Häufig wird eine weiter Funktion als syntaktischer Zucker hinzugefügt.

```
≫ :: 'a T →'b T →'b T
```

Wobei p $\gg$ q die Abkürzung für p $\gg=\lambda$`x. q x` ist.

Ein Beispiel für Monaden bietet die Exception-Monade (`EX, ret,` $\gg=$):

```
type 'a EX = throw Exception | return 'a

ret  :: 'a →'a EX
ret 'a = return 'a

≫= :: 'a EX → ('a → 'b EX) → 'b EX
(throw e) ≫= q     = throw e
(return x)≫= q     = q (x)
```

Die Division durch Null in `divFun` kann mit Hilfe einer solchen Monade abgefangen werden und eine Fehlermeldung weitergeben.

```
divFun_Mon :: Nat -> Nat -> Nat EX


divFun_Mon x 0 = throw "division by zero"
divFun_Mon x y = return IF(x>y) THEN
                          1+(divFun_Mon (x-y) y)
                       ELSE 0
```

Die in dieser Arbeit aufgezeigten Gesetzmäßigkeiten abstrahieren jedoch von der konkreten Beschaffenheit der Monade. Sie lassen sich daher nicht nur auf die Fehlerbehandlung anwenden. Weitere Einsatzmöglichkeiten sind zum Beispiel:

- Zustands-Monade: Zustände sind ein typisch imperativer Ansatz, der wie bei `divImp` motiviert, zu einigen Problemen führen kann. Trotzdem wird manchmal das Konzept des globalen Variablenraums bevorzugt. Hierzu muss in funktionalen Sprachen auf Monaden zurückgegriffen werden. Die Zustands-Monade ist dabei definiert als `'a State = (state → state × 'a)`, also dem Übergang von einem Zustand in einen nächsten und Rückgabe eines Wertes. Die Zustands-Monade ist die allgemeinste Form der Monade. Alle anderen lassen sich auf das Lesen und Schreiben eines globalen Zustandes zurückführen.

- Maybe-Monade: Dieser Datentyp bietet die Möglichkeit ein undefiniertes Ergebnis zurückzugeben. In der Umsetzung $\texttt{divFun}_{Tup}$ kann im Fehlerfall das Ergebnis `(Nothing, "division by zero")` lauten, bei fehlerloser Auswertung `(Just divFun x y, " ")`.

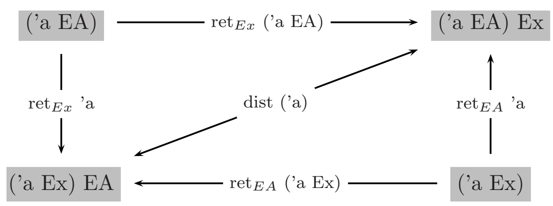
- interaktive Ein-/Augabe-Monade: Diese Monade ist eine erweiterte Zustands-Monade, die die Kommunikation mit dem Benutzer ermöglicht.

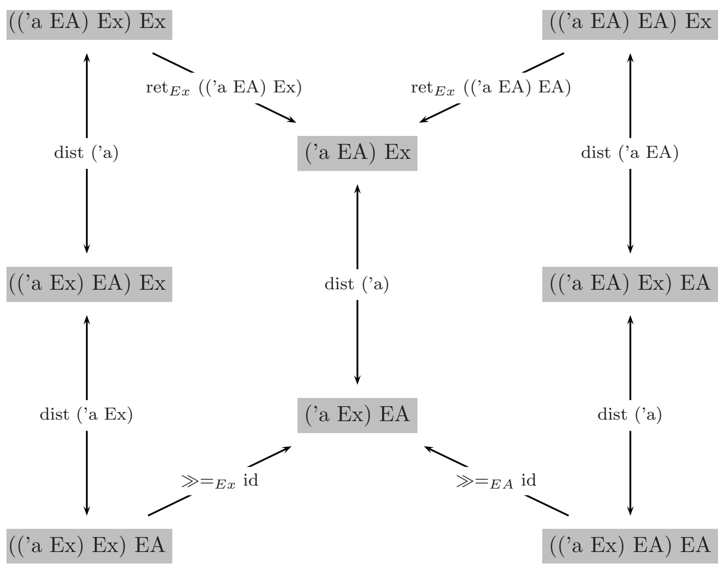
- Fortsetzungs-Monade: Für Berechnungen, die unterbrochen und eventuell später fortgesetzt werden sollen, wird diese Art der Monaden eingesetzt. Mögliche Anwendungsgebiete sind komplexe Kontrollstrukturen, eingeschobene Fehlerbehandlung oder parallele Berechnungen. Dabei repräsentiert eine Fortsetzung die Zukunft der Berechnung als Funktion von Zwischen-Ergebnissen vom Typ `'a` in die Menge der Endergebnisse (`R`). Die Monade ist von der Form `'a T = ('a →R) →R`.

Monaden lassen sich auch miteinander kombinieren. So ist zum Beispiel eine interaktive Exception-Monade denkbar, die bei fehlerhafter Eingabe eine entsprechende Meldung auf dem Bildschirm ausgibt.

Bei einer solchen Verschmelzung zweier Monaden muss auch ein Möglichkeit geschaffen werden, die auf den Monaden definierten Methoden `ret` und $\gg=$ auf die kombinierte Variante anzuwenden. Dazu wird eine Funktion `dist` benötigt, die die Distributivität zwischen den beiden Monaden beschreibt. `dist` muss also eine Übersetzung von $('a\ T_1)\ T_2$ nach $('a\ T_2)\ T_1$ und umgekehrt zur Verfügung stellen.

Beispiel 2.2. *Ein Beispiel ist die erwähnte interaktive Exception-Monade. Die Abbildungen zeigen, wie sich nun alle Varianten von `ret` und $\gg=$ unter Verwendung von `dist` anwenden lassen. Dabei sind mit $\texttt{ret}_{Ex}$ und $\texttt{ret}_{EA}$ jeweils die entsprechenden Umsetzungen der Funktion `ret` in der Exception- bzw. E/A-Monade gemeint. Entsprechendes gilt für $\gg=_{Ex}$ und $\gg=_{EA}$. Zusätzlich ist jeweils angegeben, auf welche Parameter die Funktion angewendet wird.*





Für die einfachere Anwendung von Monaden wird an dieser Stelle noch die do-Notation eingeführt. Diese vor allem aus Haskell bekannte Schreibweise für Monaden ist syntaktischer Zucker, der die Lesbarkeit jedoch stark erhöht. Dabei ist ein Programm der Form $do\{x \leftarrow p; q\}$ zu verstehen als p $\gg\!=\lambda$x. q und $do\{p; q\}$ ist eine Umschreibung für p $\gg$ q.

Um auszudrücken, dass von der Länge und dem konkreten Aufbau eines monadischen Ausdrucks der Form $do\{x_1 \leftarrow p_1; \ldots; x_n \leftarrow p_n\ x_1 \ldots x_{n-1}\}$ abstrahiert werden kann, führen wir für die Programmsequenz $x_1 \leftarrow p_1; \ldots; x_n \leftarrow p_n\ x_1 \ldots x_{n-1}$ die abkürzende Schreibweise $\bar{x} \leftarrow \bar{p}$ ein.

Aufbauend auf der Definition von Monaden werden die für die Programmierung hilfreichen Kontrollstrukturen wie Fallunterscheidung, Rekursion und Schleifen definiert. Diese müssen den möglichen Seiteneffekten ihrer Argumente Rechnung tragen.

Definition 2.2. *Sei b, p und q monadische Programme, wobei b nach bool T auswertet, dann ist die **bedingte Auswertung** if_T definiert als:*

$$if_T\ b\ then\ p\ else\ q = do\{x \leftarrow b; if\ x\ then\ p\ else\ q\}$$

Definition 2.3. *Sei `test` eine Funktion, die ein Formel nach `bool T` auswertet, dann lautet die auf `if`$_T$ aufbauende Definition der **rekursiven Auswertung** $iter_T$:*

$$iter_T\ test\ f\ x = do\{if_T\ (test\ x)\ then\ (do\{y \leftarrow (f\ x); iter_T\ test\ f\ y\})\ else\ (ret\ x)\}$$

Definition 2.4. *Mit Hilfe der Rekursion lässt sich die **wiederholte Auswertung** $while_T$ definieren als:*

$$while_T\ b\ p\ = iter_T(\lambda x.b)(\lambda x.p)()$$

2.2.1. Monaden-Gesetze

Damit das in Definition 2.1 eingeführte Tripel im mathematischen Sinn eine Monade bildet, muss es noch die drei folgenden Gesetze erfüllen.

1. Links-Einheit: $do\{y \leftarrow ret\ x; py\} = p\ x$

2. Rechts-Einheit: $do\{x \leftarrow p; ret\ x\} =\ p$

3. Assoziativität: $do\{x \leftarrow do\{x \leftarrow p; qx\}; rx\}$ = $do\{x \leftarrow p; y \leftarrow qx; ry\}$

Die ersten beiden Gesetze beschreiben `ret` als Einheitselement bezüglich der Bindung, das dritte die Möglichkeit, dass Teilsequenzen beliebig geklammert werden können, ohne dass sich das Ergebnis der Auswertung ändert.

Mit Hilfe der drei Gesetze lässt sich eine Erweiterung von Gleichungen einführen.

Die Idee dabei ist, dass zwei Berechnungen auch dann noch zum gleichen Ergebnis führen, wenn jeweils die gleiche Sequenz angehängt wird.

So sollten zwei unterschiedliche Implementierungen der Division zweier natürlicher Zahlen ohne Rest (`div`$_1$ und `div`$_2$) die gleichen Ergebnisse bei einer anschließenden Addition (`sum`) liefern, wenn sie an sich äquivalent auswerten.

$$
\begin{aligned}
&div_1\ a\ b = div_2\ a\ b \Longrightarrow \\
&do\{x \leftarrow div_1\ a\ b;\ sum\ x\ c\} = do\{x \leftarrow div_2\ a\ b;\ sum\ x\ c\}
\end{aligned}
$$

So kann `div`$_1$ die Umsetzung der vorgestellten Methode `divFun`$_{Mon}$ durch Addition mit einer Exception bei Division durch Null sein. `div`$_2$ kann hingegen eine Variante auf Basis einer binären Darstellung des Problems umsetzen. Diese muss ebenfalls bei Division durch Null die entsprechende Exception liefern. Allgemeiner formuliert gilt:

Lemma 2.1 (ret2seq). *Seine* $\bar{x} \leftarrow \bar{p}$ *und* $\bar{x} \leftarrow \bar{q}$ *Programmsequenzen, dann gilt:*

$$
\frac{do\{\bar{x} \leftarrow \bar{p}; ret\ \bar{x}\} = do\{\bar{x} \leftarrow \bar{q}; ret\ \bar{x}\}}{do\{\bar{x} \leftarrow \bar{p}; r\ \bar{x}\} = do\{\bar{x} \leftarrow \bar{q}; r\ \bar{x}\}}
$$

Beweis. Der Beweis erfolgt durch Anwenden des zweiten Monaden-Gesetzes und im zweiten Schritt durch Rewriting mit Hilfe der Voraussetzung:

$$
\begin{aligned}
do\{\bar{x} \leftarrow \bar{p}; ret\ \bar{x}\} &= do\{z \leftarrow do\{\bar{x} \leftarrow \bar{p}; ret\ \bar{x}\}; r\ z\} \\
&= do\{z \leftarrow do\{\bar{x} \leftarrow \bar{q}; ret\ \bar{x}\}; r\ z\} \\
&= do\{\bar{x} \leftarrow \bar{q}; ret\ \bar{x}\}
\end{aligned}
$$

□

Die folgenden Ausführungen beschreiben aufbauend auf den nun vorhandenen Grundlagen für monadische Berechnungen, die Einführung eines Hilfskalküls, das die Beweise der Haore-Regeln stark vereinfacht. Es wird der Begriff der deterministisch seiteneffektfreien Auswertung monadischer Sequenzen entwickelt und einige Eigenschaften betrachtet, die damit einhergehen.

2.3. Globale Dynamische Aussagen

Um das Hoare-Kalkül für monadische Programme beweisen zu können, wird wie in [SM03] und [SM04] vorgeschlagen, zunächst ein Hilfs-Kalkül für *global gültige dynamische Aussagen* formuliert. Die Idee dabei ist, logische Formeln von den bei ihrer Auswertung auftretenden Seiteneffekten zu separieren. Somit lassen sich die aus der Logik bekannten Regeln auf sie anwenden.

Definition 2.5. *Sei $\bar{x} \leftarrow \bar{p}$ eine Programmsequenz und Φ eine boolesche*[3] *Formel, dann wird die Gleichung*

$$do\{\bar{x} \leftarrow \bar{p}; ret(\Phi\ \bar{x}, \bar{x})\} = do\{\bar{x} \leftarrow \bar{p}; ret(True, \bar{x})\}$$

als so genanntes ***gdj*** *(von engl.* ***g****lobal* ***d****ynamic* ***j****udgements) abgekürzt durch*

$$[\bar{x} \leftarrow \bar{p}]\ \Phi\ \bar{x}.$$

Nach Auswertung der Sequenz $\bar{x} \leftarrow \bar{p}$ hält die logische Formel $\Phi\bar{x}$.

Beispiel 2.3. *Sei S eine Programmsequenz, die die Multiplikation zweier natürlicher Zahlen auf Basis der Addition umsetzt. Dann soll für `S c d` gelten, dass das Ergebnis der Auswertung `res` die Gleichung `res = c * d` erfüllt. Als gdj formuliert heisst das:*

$$[res \leftarrow S\ c\ d]\ (res = c * d).$$

Die ausgelagerte Gleichung arbeitet direkt auf den natürlichen Zahlen und wertet in den Bereich der Wahrheitswerte und nicht nach `bool T` aus.

Eine häufig benötigte Eigenschaft dieser globalen Aussagen ist die Erweiterung von Lemma 2.1.

Lemma 2.2 (gdj2doSeq). *Sei $\bar{x} \leftarrow \bar{p}$ eine Programmsequenz, q ein monadisches Programm und Φ eine Formel die in die Wahrheitswerte auswertet, dann gilt:*

$$\frac{[\bar{x} \leftarrow \bar{p}]\ \Phi\bar{x}}{do\{\bar{x} \leftarrow \bar{p}; q\ \bar{x}\ (\Phi\ \bar{x})\} = do\{\bar{x} \leftarrow \bar{p}; q\ \bar{x}\ (True)\}}$$

Beweis. Aus der Voraussetzung lässt sich durch Anwendung der gdj-Definition die Gleichung

$$do\{\bar{x} \leftarrow \bar{p}; ret(\Phi\bar{x}, \bar{x})\} = do\{\bar{x} \leftarrow \bar{p}; ret(True, \bar{x})\}$$

herleiten. Mit Hilfe des ersten Monaden-Gesetzes können Φx und $True$ aus `ret` herausgezogen werden.

$$\begin{aligned} & do\{(y, \bar{x}) \leftarrow do\{\bar{x} \leftarrow \bar{p}; ret(\Phi\bar{x}, \bar{x})\}; ret(y, \bar{x})\} \\ = {} & do\{(y, \bar{x}) \leftarrow do\{\bar{x} \leftarrow \bar{p}; ret(True, \bar{x})\}; ret(y, \bar{x})\} \end{aligned}$$

Nun wird `ret (x,y)` auf beiden Seiten mit Hilfe von Lemma 2.1 durch `q x y` ersetzt. Die erneute Anwendung des ersten Monaden-Gesetzes zeigt dann die Korrektheit der Behauptung. □

[3] Sowohl die Schreibweisen `boolsche` als auch `boolesche` sind gebräuchlich. Ich habe mich für die, dem Namensgeber Boole entsprechende zweite Variante entschieden. Nach www.googlefight.com ist dies auch die im Netz am häufigsten zu findende Schreibweise.

Im Verfeinerungsschritt lässt sich zeigen, dass zwei Programme, die nach Abarbeitung einer Programmsequenz und unter Beachtung der dabei auftretenden Seiteneffekte unter jeder Eingabe gleich sind, in ihrem Vorkommen austauschbar sind. Für die Programme $\texttt{div}_1$ und $\texttt{div}_2$, die zwar unterschiedliche Ansätze für die Division zweier natürlicher Zahlen verfolgen aber zum selben Ergebnis auswerten, hat dies zur Folge, dass wir sie beliebig gegeneinander austauschen können ohne an den geltenden globalen dynamischen Aussagen etwas zu verändern.

Da der Beweis der entsprechenden Regel eine genauere Betrachtung zur Gleichheit zweier monadischer Programme benötigt, wird dieser in Kapitel 5 nachgereicht. An dieser Stelle führen wird die Aussage für die weiter Verwendung zunächst nur axiomatisch ein.

Axiom 2.1 (rp). *Seien $\bar{x} \leftarrow \bar{p}$, $y \leftarrow q_1$, $y \leftarrow q_2$ und $\bar{z} \leftarrow \bar{r}$ monadische Programmsequenzen und Φ eine boolesche Formel, dann gilt:*

$$\frac{\begin{array}{c}\forall \bar{x}.q_1\bar{x} = q_2\bar{x}\\ [\bar{x} \leftarrow \bar{p}; y \leftarrow q_1\bar{x}; \bar{z} \leftarrow \bar{r}\bar{x}y] \;\; \Phi\bar{x}y\bar{z}\end{array}}{[\bar{x} \leftarrow \bar{p}; y \leftarrow q_2\bar{x}; \bar{z} \leftarrow \bar{r}\bar{x}y] \;\; \Phi\bar{x}y\bar{z}}$$

Die von Seiteneffekten befreiten booleschen Formeln bieten unter anderem die Möglichkeit, aus der Logik bekannte Regeln auf sie anzuwenden. Die Konjunktions-Einführung ([A, B] $\Longrightarrow$ A $\wedge$ B) und der *Modus Ponens* ([A, A $\Longrightarrow$ B] $\Longrightarrow$ B) lassen sich in entsprechender Weise umsetzen:

Lemma 2.3 ($\wedge I$). *Seien Φ und ξ boolesche Formeln und $\bar{x} \leftarrow \bar{p}$ eine monadische Programm-Sequenz, dann gilt:*

$$\frac{\begin{array}{c}[\bar{x} \leftarrow \bar{p}] \;\; \Phi\;\bar{x}\\ [\bar{x} \leftarrow \bar{p}] \;\; \xi\;\bar{x}\end{array}}{[\bar{x} \leftarrow \bar{p}] \;\; (\Phi\;\bar{x} \wedge \xi\bar{x})}$$

Beweis. Wenn man die Definition für die globalen Aussagen auf die Behauptung anwendet, erhält man die Gleichung

$$do\{\bar{x} \leftarrow \bar{p}; ret(\Phi\bar{x} \wedge \xi\bar{x}, \bar{x})\} = do\{\bar{x} \leftarrow \bar{p}; ret(True, \bar{x})\}.$$

Der erste Teil der Gleichung wird durch Anwendung der beiden Voraussetzungen umgeformt zu:

$$do\{\bar{x} \leftarrow \bar{p}; ret(True \wedge True, \bar{x})\}$$

Dies ist durch Anwendung der Regeln für den logischen Operator $\wedge$ äquivalent zur rechten Seite der Gleichung. □

Lemma 2.4 (wk). *Seien Φ und ξ boolesche Formeln und $\bar{x} \leftarrow \bar{p}$ eine monadische Sequenz, dann gilt:*

$$\frac{\begin{array}{c}\forall \bar{x}.\Phi\bar{x} \Rightarrow \xi\ \bar{x} \\ [\bar{x} \leftarrow \bar{p}]\ \ \Phi\ \bar{x}\end{array}}{[\bar{x} \leftarrow \bar{p}]\ \ \xi\ \bar{x}}$$

Beweis. Ähnlich wie beim vorangegangenen Beweis, liegt der Schlüssel hier im Anwenden der gdj-Definition. Zunächst wandeln wir durch die logische Äquivalenz von $\xi\ \ \bar{x}$ und `True` $\wedge\ \ \xi\ \ \bar{x}$ die linke Seite der zu zeigenden Gleichung um zu

$$do\{\bar{x} \leftarrow\ \bar{p}; ret(True \wedge \xi\ \bar{x},\ \bar{x})\}.$$

Dies ist nach der zweiten Voraussetzung äquivalent zu

$$do\{\bar{x} \leftarrow\ \bar{p}; ret(\Phi\ \bar{x} \wedge \xi\ \bar{x},\ \bar{x})\}$$

Da für alle $\bar{x}$ gilt, dass $\xi\ \bar{x}$ aus $\Phi\ \bar{x}$ folgt, wird dies zusammengefasst zu

$$do\{\bar{x} \leftarrow \bar{p}; ret(\Phi\bar{x}, \bar{x})\}bzw.do\{\bar{x} \leftarrow \bar{p}; ret(True, \bar{x})\}$$

□

Die separierten booleschen Formeln sind zwar von den auftretenden Seiteneffekten entkoppelt, können aber durch Parameter von der Ausführung der vorangegangenen Programme beeinflusst werden. Interessant ist nun also zu betrachten, wie sich die Formeln verhalten, wenn die Sequenzen auf deren Ergebnis sie sich beziehen in eine längere Sequenz eingebettet werden.

Lemma 2.5 (app). *Seien p und q bzw. $\bar{p}$ und $\bar{q}$ monadische Programme, dann lässt für eine gegebene boolesche Formel Φ die Sequenz $\bar{x} \leftarrow \bar{p}$ erweitern durch Anhängen (engl. append):*

$$\frac{[\bar{x} \leftarrow \bar{p}]\ \ \Phi\ \bar{x}}{[\bar{x} \leftarrow \bar{p}; y \leftarrow q\ \bar{x}]\ \ \Phi\ \bar{x}}$$

Lemma 2.6 (pre). *Unter den gleichen Voraussetzungen kann ein Präfix die Sequenz erweitern:*

$$\frac{\forall x. [\bar{y} \leftarrow \bar{q}\ x]\ \ \Phi\ x\bar{y}}{[x \leftarrow p; \bar{y} \leftarrow \bar{q}\ x]\ \ \Phi\ x\bar{y}}$$

Beweis.

app: Mit Hilfe von Lemma 2.2 lässt sich aus der Voraussetzung

$$\begin{aligned} &do\{\bar{x} \leftarrow \bar{p}; do\{y \leftarrow q\bar{x}; ret(\Phi\bar{x}, \bar{x}, y)\}\} \\ =\ &do\{\bar{x} \leftarrow \bar{p}; do\{y \leftarrow q\bar{x}; ret(True, \bar{x}, y)\}\} \end{aligned}$$

herleiten. Dies zeigt mit Hilfe des Assoziativ-Gesetzes für Monaden die Behauptung.

pre: Aus der Voraussetzung erhalten wir durch Anwendung von Lemma 2.2 die Gleichheit von $do\{\bar{y} \leftarrow \bar{q}x; ret(\Phi x\bar{y}, x, \bar{y})\}$ und $do\{\bar{y} \leftarrow \bar{q}x; ret(True, x, \bar{y})\}$. Wird diese für die Substitution in der, durch Identität wahren Gleichung

$$do\{x \leftarrow p; \bar{y} \leftarrow \bar{q}x; ret(\Phi x\bar{y}, x, \bar{y})\} = do\{x \leftarrow p; \bar{y} \leftarrow \bar{q}x; ret(\Phi x\bar{y}, x, \bar{y})\}$$

angewendet, erhalten wir die do-Notation der Behauptung.

□

In den vorangegangenen Beweisen ist deutlich geworden, wie elementar die Monaden-Gesetze für die Beweisführung auf monadischen Programmen sind. Damit wir diese Eigenschaften innerhalb der gdj anwenden können, werden entsprechende Regeln benötigt.

Lemma 2.7 (η). *Seien $\bar{x} \leftarrow \bar{p}$ und $\bar{z} \leftarrow \bar{q}\ \bar{x}\ y$ monadische Programmsequenzen und Φ eine boolesche Formel. Die beiden ersten Monaden-Gesetze, die die Links- bzw. Rechtseinheit von* `ret` *beschreiben, können in die gdj übertragen werden als:*

$$\frac{[\bar{x} \leftarrow \bar{p}; y \leftarrow ret(a\ \bar{x}); \bar{z} \leftarrow \bar{q}\ \bar{x}y]\ \ \Phi\ \bar{x}y\bar{z}}{\overline{[\bar{x} \leftarrow \bar{p}; \bar{z} \leftarrow \bar{q}\ \bar{x}\ (a\ \bar{x})]\ \ \Phi\ \bar{x}(a\ \bar{x})\bar{z}}}$$

Die beiden Striche zeigen an, dass das Lemma in beide Richtungen auswertbar ist.

Beweis. Der untere gdj folgt aus dem oberen durch Anwendung von Lemma 2.2. Wir erhalten

$$\begin{aligned} &do\{(\bar{x}, y, \bar{z}) \leftarrow do\{\bar{x} \leftarrow \bar{p}; y \leftarrow ret\ (a\ \bar{x}); \bar{z} \leftarrow \bar{q}\ \bar{x}\ y; ret(\bar{x}, y, \bar{z})\}; ret(\Phi\ \bar{x}\ (a\ \bar{x})\ \bar{z},\ \bar{z})\} \\ =\ &do\{(\bar{x}, y, \bar{z}) \leftarrow do\{\bar{x} \leftarrow \bar{p}; y \leftarrow ret\ (a\ \bar{x}); \bar{z} \leftarrow \bar{q}\ \bar{x}\ y; ret(\bar{x}, y, \bar{z})\}; ret(True,\ \bar{z})\} \end{aligned}$$

Dies lässt sich durch die Monaden-Gesetze umformen in die do-Notation der Behauptung. Die andere Richtung verläuft äquivalent. □

Lemma 2.8 (ctr). *Wenn $\bar{v} \leftarrow \bar{t}$, $x \leftarrow p\ \bar{v}$, $y \leftarrow q\ \bar{v}$ und $\bar{z} \leftarrow \bar{r}\ \bar{v}\ y$ monadische Programmsequenzen sind und Φ eine boolesche Formel, dann lässt sich die, in die gdj übertrage Version des dritten Monaden-Gesetzes (Assoziativität) formulieren als:*

$$\frac{[\bar{v} \leftarrow \bar{t}; x \leftarrow p\ \bar{v}; y \leftarrow q\ \bar{v}; \bar{z} \leftarrow \bar{r}\ \bar{v}y]\ \ \Phi\ \bar{v}y\bar{z}}{[\bar{v} \leftarrow \bar{t}; y \leftarrow (do\{x \leftarrow p\ \bar{v}; q\ \bar{v}\}); \bar{z} \leftarrow \bar{r}\ vy]\ \ \Phi\ \bar{v}y\bar{z}}$$

Beweis. Wie bei Lemma 2.7 wird auch hier durch Anwendung von Lemma 2.2 die benötigte do-Notation hergeleitet. □

Wir haben gezeigt, welche Möglichkeiten sich bieten, wenn man logische Formeln von ihrer eventuell seiteneffektbehafteten Auswertung separiert. Im Folgenden wenden wir uns Programmen zu, deren Auswertung seiteneffektfrei verläuft.

2.4. Seiteneffektfreiheit

Wie in Abschnitt 2.2 motiviert, dienen Monaden dazu, Seiteneffekte in das funktionale Paradigma zu integrieren. Im Folgenden werden einige Eigenschaften aufgezeigt, die gelten falls eine *deterministische seiteneffektfreie Auswertung* zugesichert werden kann. Für die in Abschnitt 2.2 eingeführten Beispiele von Monaden ist *deterministische Seiteneffektfreiheit* wie folgt zu verstehen:

- Zustands-Monade: bei keiner Auswertung der Programmsequenz wird der globale Zustand gelesen oder verändert
- Maybe-Monade: jede Auswertung des monadischen Programms liefert einen definierten Wert zurück
- interaktive Ein-/Ausgabe-Monade: das Programm terminiert, ohne dass eine Benutzereingabe oder eine Ausgabe erfolgt
- Exception-Monade: das Programm terminiert bei jeder Auswertung ordnungsgemäß ohne eine Exception zu werfen

Um den Begriff der deterministischen seiteneffektfreien Auswertung formal zu beschreiben, verfolgen wir den in [SM04] beschriebenen Ansatz über die Kopierbarkeit von Programmen und die Seiteneffektfreiheit ohne Betrachtung des Determinismus.

Definition 2.6. *Sei p eine monadische Programmsequenz. p ist* ***seiteneffektfrei*** *(kurz sef) gdw.*

$$do\{x \leftarrow p; ret\ ()\} = ret\ ()$$

Definition 2.7. *Sei p eine monadische Sequenz. Dann heißt p* ***kopierbar*** *(cp) gdw. gilt:*

$$do\{x \leftarrow p; y \leftarrow p; ret(x, y)\} = do\{x \leftarrow p; ret(x, x)\}$$

Definition 2.8. *Sei p eine monadische Sequenz. Dann heißt p* ***deterministisch seiteneffektfrei*** *(dsef) wenn folgende drei Eigenschaften erfüllt sind:*

- *p ist seiteneffektfrei*

- *p ist kopierbar*
- *für alle seiteneffektfreien, kopierbaren Programme q gilt die Kopierbarkeit von* $do\{x \leftarrow p; y \leftarrow q; ret(x,y)\}$.

Die erste Eigenschaft sichert zu, dass bei separater Ausführung des Programms keine Seiteneffekte auftreten. Die zweite besagt, dass jede Auswertung zum gleichen Ergebnis führt. Insbesondere ist das Ergebnis unabhängig von der Auswertungsreihenfolge.

In [SM03] wird an Hand einer Fortsetzungsmonade gezeigt, dass Seiteneffektfreiheit und Kopierbarkeit den Begriff der deterministischen Seiteneffektfreiheit nicht vollständig beschreiben und die dritte, im Folgenden als Vertauschbarkeit bezeichnete Eigenschaft dazu benötigt wird.

2.4.1. Eigenschaften von Monaden

Bevor wir uns den Möglichkeiten zuwenden, die deterministisch seiteneffektfreie Programme zur Verfügung stellen, werden zunächst einige Eigenschaften für seiteneffektfreie, kopierbare und vertauschbare Programme aufgezeigt.

Lemma 2.9 (seFree). *Seien p und q monadische Programme. Dann gilt:*

$$\frac{sef\ p}{do\{x \leftarrow p; q\} = q}$$

Beweis. Die Behauptung lässt sich mit Hilfe der Monaden-Gesetze umwandeln in

$$do\{x \leftarrow do\{p; ret()\}; q\} = q$$

Unter der Voraussetzung und der Definition für `sef` reduziert sich der innere do-Term zu `ret ()`, welches wiederrum durch das erste Monaden-Gesetz verworfen wird. □

Beispiel 2.4. *Sei S wie in Beispiel 2.3 eine Programmsequenz, die die Multiplikation zweier natürlicher Zahlen auf Basis der Addition umsetzt. Wenn S normal terminiert, wird das Ergebnis an x gebunden.* `sum` *berechnet die Summe zweier Werte. In der Programmsequenz*

```
x← S a b; sum 1 2
```

wird x im weiteren Verlauf nicht mehr in die Berechnung einbezogen. Somit kann man die Auswertung der Multiplikation entfallen und lediglich die Sequenz

```
sum 1 2
```

ausgewertet werden. Wenn S keine Seiteneffekte hat und die angehängte Sequenz keinen Rückgabewert hat, dann kann auch die angehängte Sequenz verworfen werden. Damit gilt:

```
x← S a b; ret() = S a b
```

Lemma 2.10 (seFree2). *Seien p und q monadische Programme. Dann gilt:*

$$\frac{sef\ p \qquad q = ret()}{do\{x \leftarrow p; q\} = p}$$

Beweis. Aus der Seiteneffektfreiheit von p erhalten wir durch Anwendung der Definition und der Voraussetzung `p = ret() = q` Durch Lemma 2.9 wissen wir, dass $q = do\{x \leftarrow p; q\}$ gilt. □

Lemma 2.10 lässt sich ähnlich auch für die Anwendung innerhalb der gdj formulieren.

Lemma 2.11 (sef$_0$). *Wenn p und q monadische Programme sind, dann gilt:*

$$\frac{\begin{array}{c}\forall x.sef\ (q\ x)\\ [x \leftarrow p; q\ x]\ \ \Phi\ x\end{array}}{[x \leftarrow p]\ \ \Phi\ x}$$

Beweis. Mit Lemma 2.2 erhalten wir aus der Voraussetzung die Gleichung

$$do\{x \leftarrow p; y \leftarrow qx; ret(\Phi x, x)\}\ =\ do\{x \leftarrow p; y \leftarrow qx; ret(True, x)\}$$

Die Seiteneffektfreiheit von q lässt uns die Sequenz $y \leftarrow qx$ unter Anwendung von Lemma 2.9 entfernen. Mit Hilfe der Monaden-Gesetze können wir das `ret` verwerfen und schließlich durch Anwendung der gdj-Definition die Behauptung zeigen. □

Es können auch seiteneffektfreie Sequenzen, die an beliebiger Stelle in einem monadischen Programm stehen, entfernt werden, wenn ihr Ergebnis nicht weiterverwendet wird.

Lemma 2.12 (sef). *Seien p, q und r monadische Sequenzen und Φ eine boolesche Formel. Dann gilt:*

$$\frac{\begin{array}{c}\forall x.sef\ (q\ x)\\ [x \leftarrow p; qx; z \leftarrow rx]\ \ \Phi xz\end{array}}{[x \leftarrow p; z \leftarrow rx]\ \ \Phi xz}$$

Beweis. Durch die Seiteneffektfreiheit von q wird mit Lemma 2.9 gezeigt, dass für alle x die Gleichung $do\{y \leftarrow q\ x; r\ x\} = r\ x$ gilt. Weiterhin erhält man aus der zweiten Bedingung durch Lemma 2.8 das gdj $[x \leftarrow p; z \leftarrow do\{y \leftarrow q\ x; r\ x\}]\ \Phi\ x\ z$. Wird Axiom 2.1 auf beides angewandt, erhält man die Behauptung. □

Für seiteneffektfrei auswertende Programme wird die Definition von Kopierbarkeit auf gdj erweitert.

Lemma 2.13 (sef2cp). *Sei p ein monadisches Programm, dann gilt:*

$$\frac{sef\ p}{cp\ p\ =\ [x \leftarrow p; y \leftarrow p]\ (x = y)}$$

Beweis. `cp p` $\Longrightarrow [x \leftarrow p; y \leftarrow p]\ \ (x = y)$
Es wird mit Hilfe der Definition von `cp` gezeigt, dass die Gleichung (2.1) gilt. Da laut Voraussetzung p seiteneffektfrei auswertet, wird im zweiten Teil der Gleichung mit Hilfe von Lemma 2.9 $y \leftarrow p$ eingefügt. Mit Hilfe der Monaden-Gesetze wird diese Gleichung neu geklammert und in die Gleichung (2.3) umgeformt. Durch Anwendung von Lemma 2.1 und nochmalige Umformung mit Hilfe der Monaden-Gesetze erhält man die gewünschte globale dynamische Aussage (2.4).

$$do\{x{\leftarrow}p;y{\leftarrow}p;ret(x,y)\}\ =\ do\{x{\leftarrow}p;ret(x,x)\} \tag{2.1}$$

$$do\{x{\leftarrow}p;y{\leftarrow}p;ret(x,y)\}\ =\ do\{x{\leftarrow}p;y{\leftarrow}p;ret(x,x)\} \tag{2.2}$$

$$\begin{aligned} &do\{(x,y){\leftarrow}do\{x{\leftarrow}p;y{\leftarrow}p;ret(x,y)\};ret\ ((x,y),x)\} \\ =\ &do\{(x,y){\leftarrow}do\{x{\leftarrow}p;y{\leftarrow}p;ret(x,y)\};ret\ ((x,y),y)\} \end{aligned} \tag{2.3}$$

$$[x{\leftarrow}p;y{\leftarrow}p](x{=}y) \tag{2.4}$$

Die Rückrichtung verläuft äquivalent dazu. □

2.4.2. Deterministische Seiteneffektfreiheit

Die Definition von `dsef` zeigt, dass alle beschriebenen Regeln insbesondere für deterministisch seiteneffektfreie Programme gelten. Um sie nicht nur auf atomare Programme anwenden zu können, führen wir eine Erweiterungsmöglichkeit ein. Die Idee dabei ist, dass zwei an sich deterministisch seiteneffektfreie Programme auch in Komposition keine Seiteneffekte aufweisen.

Beispiel 2.5. *Zwei Programme auf Basis einer Ein-/Ausgabe-Monade sind seiteneffektfrei, wenn sie keine Interaktion mit dem Benutzer zulassen. Wenn beide für sich diese Möglichkeit ausschließen, wird der Benutzer auch bei Ausführung beider Programme nicht mit ihnen interagieren können.*

Um diese Eigenschaft der Komposition zweier deterministisch seiteneffektfreier monadischer Programme nachzuweisen, beweisen wir sie zunächst für die Seiteneffektfreiheit, die Kopierbarkeit und die Vertauschbarkeit.

Lemma 2.14 (weak_sef2seq). *Für alle monadischen Programme p und r gilt:*

$$\frac{sef\ p \quad \forall x.\ sef\ (r\ x)}{sef\ (do\{x \leftarrow p;\ r\ x\})}$$

Beweis. Zu zeigen ist, dass $do\{do\{x \leftarrow p; r\ \ x\}; ret()\}$ `= ret ()` gilt. Gleichung (2.5) erhält man durch die Monaden-Gesetze. Durch die Seiteneffektfreiheit von p und q lässt sich durch Anwendung der entsprechenden Definition die Gleichheit mit `ret()` zeigen.

$$
\begin{aligned}
do\ \{z{\leftarrow}do\ \{x{\leftarrow}p;\ r\ x\};\ ret\ ()\} &= do\ \{x{\leftarrow}p;\ do\{r\ x;\ ret\ ()\}\} && (2.5)\\
&= do\ \{x{\leftarrow}p;\ ret()\} && (2.6)\\
&= ret\ () && (2.7)
\end{aligned}
$$

□

Für die Kopierbarkeit reicht es nicht, dass `p` und `r` kopierbar sind. Ohne weitere Bedingungen lässt sich zunächst nur eine Erweiterbarkeit um eine Rückgabe zeigen.

Lemma 2.15 (weak_cp2retSeq). *Sei p ein monadisches Programm, dann gilt:*

$$
\frac{cp\ p}{cp\ (do\{x \leftarrow p; ret(a\ x)\})}
$$

Beweis. Für die Kopierbarkeit wird gezeigt, dass f"'ur eine zweifache Ausführung der Sequenz das Ergebnis der Auswertung das selbe ist wie bei einfacher Auswertung. Durch Anwendung des Assoziativ-Gesetzes für Monaden auf die doppelte Sequenz und anschließende Umformung mit Hilfe des ersten Monaden-Gesetzes erhalten wir (2.10). Die Vertauschbarkeit von `r` führt zu (2.11). Durch Anwendung der beiden Monaden-Gesetze lässt sich die zu zeigende Gleichung herleiten.

$$
\begin{aligned}
&do\{u{\leftarrow}do\{x{\leftarrow}p;ret(a\ x)\};v{\leftarrow}do\{y{\leftarrow}p;ret\ (a\ y)\};ret\ (u,v)\} && = && (2.8)\\
&do\{x{\leftarrow}p;u{\leftarrow}ret(a\ x);y{\leftarrow}p;v{\leftarrow}ret\ (a\ y);ret\ (u,v)\} && = && (2.9)\\
&do\{x{\leftarrow}p;\ y{\leftarrow}p;ret(a\ x,\ a\ y)\} && = && (2.10)\\
&do\{x{\leftarrow}p;ret(a\ x,a\ x)\} && = && (2.11)\\
&do\{x{\leftarrow}p;u{\leftarrow}ret(a\ x);ret(u,u)\} && = && (2.12)\\
&do\{u{\leftarrow}do\{x{\leftarrow}p;ret(a\ x)\};ret(u,u)\} && && (2.13)
\end{aligned}
$$

□

Um die Erweiterbarkeit von `dsef` zeigen zu können, muss auch für `cp` eine entsprechende Regel zur Verfügung stehen. Dies ist möglich, wenn die beiden Programme nicht nur Kopierbarkeit aufweisen, sonder zusätzlich frei von Seiteneffekten sind. Zu dem muss mindestens eine der beiden Sequenzen Vertauschbarkeit aufweisen und somit insgesamt deterministisch seiteneffektfrei auswerten. Da wir für den übergeordneten Beweis dies gerade zusichern, bedeutet es hier keine Einschränkung.

Lemma 2.16 (weak_cp2seq). *Seien p, q und r monadische Programme, dann gilt:*

$$\frac{cp\ p \quad sef\ p \quad \forall x.dsef\ (r\ x)}{cp\ (do\{x \leftarrow p;\ r\ x\})}$$

Beweis. Mit Hilfe des ersten Monaden-Gesetzes und der Assoziativität lässt sich eine geeignete Klammerung (2.15) schaffen, um die Vertauschbarkeit von r anzuwenden. Durch erneute Anwendung der beiden Monaden-Gesetze wird die Klammerung dann aufgelöst. Die Kopierbarkeit von r und die Seiteneffektfreiheit von p ermöglichen die überflüssigen Vorkommen dieser beiden Programme in (2.17) zu verwerfen.

$$do\ \{u{\leftarrow}do\ \{x{\leftarrow}p;\ r\ x\};\ v{\leftarrow}do\ \{y{\leftarrow}p;\ r\ y\};\ ret\ (u,v)\} = \quad (2.14)$$
$$do\ \{x{\leftarrow}p;\ (u,y){\leftarrow}do\{u{\leftarrow}r\ x;y{\leftarrow}p;ret(u,y)\};\ v{\leftarrow}r\ y;\ ret\ (u,v)\} = \quad (2.15)$$
$$do\ \{x{\leftarrow}p;\ (u,y){\leftarrow}do\{y{\leftarrow}p;u{\leftarrow}r\ x;ret(u,y)\};\ v{\leftarrow}r\ y;\ ret\ (u,v)\} = \quad (2.16)$$
$$do\ \{x{\leftarrow}p;\ y{\leftarrow}p;\ u{\leftarrow}r\ x;\ v{\leftarrow}r\ y;\ ret\ (u,v)\} = \quad (2.17)$$
$$do\ \{x{\leftarrow}p;\ u{\leftarrow}r\ x;\ ret\ (u,u)\} \quad (2.18)$$

□

Bei der Erweiterung der Vertauschbarkeit müssen sogar beide Teilsequenzen deterministisch seiteneffektfrei auswerten.

Lemma 2.17 (weak_com2seq). *Für gegebene monadische Programme p, q und r gilt:*

$$\frac{dsef\ p \quad \forall\, x.\ dsef\ (r\ x)}{\forall\, q.\ (cp\ q \wedge sef\ q) \longrightarrow cp\ (do\ \{x{\leftarrow}do\{x{\leftarrow}p;r\ x\};\ y{\leftarrow}q;\ ret(x,y)\})}$$

Beweis. Nach der Anwendung der cp-Definition, bleibt zu zeigen, dass für alle seiteneffektfreien, kopierbaren Programme q folgende Gleichung gilt:

$$\begin{aligned}&do\{u \leftarrow do\{y_1 \leftarrow do\{x_1 \leftarrow p; r\ x_1\}; z_1 \leftarrow q; ret(y_1, z_1)\};\\ &\quad v \leftarrow do\{y_1 \leftarrow do\{x_1 \leftarrow p; r\ x_1\}; z_1 \leftarrow q; ret(y_1, z_1)\}; ret(u, v)\} =\\ &do\{u \leftarrow do\{y_1 \leftarrow do\{x_1 \leftarrow p; r\ x_1\}; z_1 \leftarrow q; ret(y_1, z_1)\}; ret(u, u)\}\end{aligned}$$

Nach Anwendung der Monaden-Gesetze erhält man die von Verschachtelungen befreite Form der Gleichung:

$$\begin{aligned}&do\{x_1 \leftarrow p; y_1 \leftarrow r\ x_1; z_1 \leftarrow q;\\ &\quad x_2 \leftarrow p; y_2 \leftarrow r\ x_2; z_2 \leftarrow q; ret((y_1, z_1), (y_2, z_2))\} =\\ &do\{x_1 \leftarrow p; z_1 \leftarrow r\ x_1; y_1 \leftarrow q; ret((z_1, y_1), (z_1, y_1))\}\end{aligned}$$

Zunächst wird durch geeignete Klammerung und Anwendung der Vertauschbarkeit von p und r die zweite Auswertung von p nach vorne verschoben. Da p kopierbar ist, kann diese Doppelung durch Anwendung der cp-Definition zusammengefasst werden. Ebenso verfährt man mit r und q. □

Nachdem die Erweiterbarkeit von `sef`, `cp` und `com` gezeigt wurden, kann durch Anwendung der Definition 2.8 gezeigt werden, dass sich auch die deterministische Seiteneffektfreiheit auf Sequenzen ausweiten lässt.

Lemma 2.18 (weak_dsef2seq). *Es gilt*

$$\frac{\begin{array}{c} dsef\ p \\ \forall\, x.\ dsef\ (r\ x) \end{array}}{dsef\ (do\{x \leftarrow p; r\ x\})}$$

für alle monadischen Programme p, q und r

Beweis. Dies folgt unmittelbar aus Definition 2.8 und den Lemmata 2.14, 2.16 und 2.17. □

Durch die Erweiterbarkeit der Eigenschaften auf Sequenzen ergeben sich weitere Möglichkeiten, die im folgenden Abschnitt aufgegriffen werden.

2.4.3. Eigenschaften kopierbarer, seiteneffektfreier Programme

Im Weiteren benötigen wir Umformungen zwischen verschiedenen Gleichungen, die in Zusammenhang mit Seiteneffektfreiheit und Kopierbarkeit stehen. Damit sie nicht jedes Mal von neuem bewiesen werden müssen, wird ihre Äquivalenz durch das folgende Lemma manifestiert.

Lemma 2.19 (cpsefProps). *Unter der Voraussetzung, dass die Programme p und q seiteneffektfrei und kopierbar sind, sind folgende Aussagen äquivalent:*

(i) *cp(do$\{$x $\leftarrow$p;y$\leftarrow$q ;ret(x,y)$\}$)*

(ii) *do$\{$x$\leftarrow$p;y $\leftarrow$q;ret(x ,y)$\}$ = do$\{$y$\leftarrow$q; x$\leftarrow$p;ret(x ,y)$\}$*

(iii) *do$\{$x$\leftarrow$p; y$\leftarrow$q;r x y $\}$ = do$\{$y $\leftarrow$q;x$\leftarrow$p ;r x y$\}$*

(iv) *[x$\leftarrow$ p;y$\leftarrow$q; z$\leftarrow$p](x=z)*

Beweis. Sei S im Folgenden eine Abkürzung für $do\{x \leftarrow p; y \leftarrow q; ret(x, y)\}$.

(i→ii)
$$\frac{\begin{array}{c} sef\ p \quad sef\ q \\ cp(do\{x \leftarrow p; y \leftarrow q; ret(x,y)\}) \end{array}}{do\{x \leftarrow p; y \leftarrow q; ret(x,y)\} = do\{y \leftarrow q; x \leftarrow p; ret(x,y)\}}$$

Durch die Seiteneffektfreiheit von p, q und `ret` erhalten wir die Seiteneffektfreiheit von S mit Hilfe von Lemma 2.14. Die durch die Monaden-Gesetze gültige Gleichung (2.19) wird durch eine Substitution mit Hilfe von Lemma 2.9 umgeformt zu (2.20). Durch Aufspaltung von z in seine Einzelteile, lässt sich die Kopierbarkeit von S ausnutzen und wir erhalten (2.22). Wenn wir die abkürzende Schreibweise für S auflösen, wird durch die Seiteneffektfreiheit jeweils das überflüssigen Vorkommen von p und q unter Anwendung von Lemma 2.9 verworfen.

$$\begin{aligned} S &= do\{z{\leftarrow}S;\, ret\ z\} && (2.19)\\ &= do\{z{\leftarrow}do\{S;S\};ret\ z\} && (2.20)\\ &= do\{w{\leftarrow}S;z{\leftarrow}S;ret\ (fst\ z,\ snd\ z)\} && (2.21)\\ &= do\{w{\leftarrow}S;z{\leftarrow}S;ret\ (fst\ z,\ snd\ w)\} && (2.22)\\ &= do\{u{\leftarrow}p;v{\leftarrow}q;x{\leftarrow}p;y{\leftarrow}q;ret(x,v)\} && (2.23)\\ &= do\{v{\leftarrow}q;x{\leftarrow}p;ret(x,v)\} && (2.24) \end{aligned}$$

(ii→iv) $$\frac{\begin{array}{c} cp\ p \quad sef\ p \\ (do\{x{\leftarrow}p;y{\leftarrow}q;ret(x,y)\} = do\{y{\leftarrow}q;x{\leftarrow}p;ret(x,y)\}) \end{array}}{[x{\leftarrow}p;y{\leftarrow}q;z{\leftarrow}p](x{=}z)}$$

Durch die Seiteneffektfreiheit von p lässt sich aus der Kopierbarkeit das gdj (2.25) mit Hilfe von Lemma 2.13 herleiten. Unter Anwendung von Lemma 2.2 erhält man die Gleichung (2.26) die sich zu (2.27) vereinfachen lässt. Durch Anwendung der Definition wird die Gleichung wieder zu dem gdj (2.28) zusammenfasst. Nach geeigneter Klammerung und Anwendung der Voraussetzung lässt sich die Behauptung herleiten.

$$[(x, y) \leftarrow do\{x \leftarrow p; y \leftarrow p; ret(x, y)\}]\ (x = y) \tag{2.25}$$

$$\begin{aligned} do\{(x, y) \leftarrow do\{x \leftarrow p; y \leftarrow p; ret(x, y)\}; z \leftarrow q; ret(x = y, x, y, z)\} \ &= \\ do\{(x, y) \leftarrow do\{x \leftarrow p; y \leftarrow p; ret(x, y)\}; z \leftarrow q; ret(True, x, y, z)\} \ & \end{aligned} \tag{2.26}$$

$$\begin{aligned} do\{x \leftarrow p; y \leftarrow p; z \leftarrow q; ret(x = y, x, y, z)\} \ &= \\ do\{x \leftarrow p; y \leftarrow p; z \leftarrow q; ret(True, x, y, z)\} \ & \end{aligned} \tag{2.27}$$

$$[x \leftarrow p; y \leftarrow p; z \leftarrow q]\ (x = y) \tag{2.28}$$

$$[x \leftarrow p; (y, z) \leftarrow do\{y \leftarrow p; z \leftarrow q; ret(y, z)\}]\ (x = y) \tag{2.29}$$

$$[x \leftarrow p; z \leftarrow q; y \leftarrow p]\ (x = y) \tag{2.30}$$

(iv→iii) $$\frac{\begin{array}{c} sef\ p \\ [x{\leftarrow}p;y{\leftarrow}q;z{\leftarrow}p](x{=}z) \end{array}}{do\{x{\leftarrow}p;y{\leftarrow}q;r\ x\ y\} = do\{y{\leftarrow}q;x{\leftarrow}p;r\ x\ y\}}$$

Die Gleichung (2.31) erhalten wir durch die Identität. Die linke Seite lässt sich durch Seiteneffektfreiheit von p unter Anwendung von Lemma 2.9 vereinfachen. Die Voraussetzung ermöglicht es die Variable x durch z zu ersetzen (2.33). Somit lässt sich auf die erste Auswertung von p wiederum Lemma 2.9 anwenden.

$$do\{x \leftarrow p; y \leftarrow q; z \leftarrow p; r\ x\ y\} = do\{x \leftarrow p; y \leftarrow q; z \leftarrow p; r\ x\ y\} \quad (2.31)$$
$$do\{x \leftarrow p; y \leftarrow q; \quad r\ x\ y\} = do\{x \leftarrow p; y \leftarrow q; z \leftarrow p; r\ x\ y\} \quad (2.32)$$
$$= do\{x \leftarrow p; y \leftarrow q; z \leftarrow p; r\ z\ y\} \quad (2.33)$$
$$= do\{\quad y \leftarrow q; z \leftarrow p; r\ x\ y\} \quad (2.34)$$

(iii→i) $$\frac{sef p \qquad \forall x. sef q \qquad do\{x{\leftarrow}p;y{\leftarrow}q;r\ x\ y\} = do\{y{\leftarrow}q;x{\leftarrow}p;r\ x\ y\}}{cp(do\{x{\leftarrow}p;y{\leftarrow}q;ret(x,y)\})}$$

Für die Kopierbarkeit muss gezeigt werden, dass

$$do\{w \leftarrow S; z \leftarrow S; ret(w, z)\} = do\{w \leftarrow S; ret(w, w)\}$$

gilt. Durch Auflösen der Abkürzung und Anwendung der Monaden-Gesetze erhalten wir Gleichung (2.36). Die Anwendung der Voraussetzung und das Auflösen der Klammerung führt zu Gleichung (2.37). Die Seiteneffektfreiheit von p und q kann dann unter Verwendung von Lemma 2.9 ausgenutzt werden. Durch Verwendung der Voraussetzung erhalten wir in (2.38) die Möglichkeit, Lemma 2.15 anzuwenden. Die restlichen Umformungen erfolgen mit Hilfe der Monaden-Gesetze.

$$do\{w \leftarrow S; z \leftarrow S; ret(w, z)\} \quad (2.35)$$
$$= do\{u \leftarrow p; (v, x) \leftarrow do\{v \leftarrow q; x \leftarrow p; ret(v, x)\}; y \leftarrow q; ret((u, v), (x, y))\} \quad (2.36)$$
$$= do\{u \leftarrow p; x \leftarrow p; v \leftarrow q; y \leftarrow q; ret((u, v), (x, y))\} \quad (2.37)$$
$$= do\{u \leftarrow p; v \leftarrow q; y \leftarrow ret((u, v), (u, v))\} \quad (2.38)$$
$$= do\{w \leftarrow S; ret(w, w)\} \quad (2.39)$$

□

Eine Anwendung dieses Lemma findet sich zum Beispiel bei der vertauschten Auswertungsreihenfolge zweier Programmsequenzen, wie sie im nächsten Lemma beschrieben ist.

Lemma 2.20 (switch). *Seien p und q monadische Programme, dann gilt:*

$$\frac{\forall r.\ p\ comwith\ r \quad sef\ p \quad cp\ q \quad sef\ q}{do\{x{\leftarrow}p;y{\leftarrow}q;ret(x,y)\} = do\{y{\leftarrow}q;x{\leftarrow}p;ret(x,y)\}}$$

Beweis. Laut Definition von `com` gilt

$$\forall r.\ (cp\ r) \wedge (sef\ r) \longrightarrow cp\ (do\ \{x{\leftarrow}p;\ y{\leftarrow}r;\ ret\ (x,\ y)\})$$

Da q die passenden Bedingungen erfüllt, erhalten wir

$$cp\ (do\ \{x{\leftarrow}p;\ y{\leftarrow}q;\ ret\ (x,\ y)\})$$

Die Anwendung von Lemma 2.19 (i$\rightarrow$ ii) liefert dann die Behauptung. $\square$

Zusammen mit den globalen dynamischen Aussagen haben wir nun alle Voraussetzungen, um eine Hoare-Logik über monadische Programme zu entwickeln.

3. Hoare-Kalkül

Wie im einleitenden Kapitel motiviert, ist das Ziel dieser Arbeit, eine Verifikationsmethode für monadische Programme bereitzustellen. Die Einführung der globalen dynamischen Aussagen bietet bereits die Möglichkeit, Formeln, die nach Auswertung eines Programms gelten sollen, zu separieren. In Beispiel 2.3 haben wir gezeigt, wie sich eine solche globale Aussage im Fallen einer Multiplikation formulieren lässt.

Um ein Programm gegenüber einer gegebenen Spezifikation verifizieren zu können bedarf es jedoch noch einiger weiterer Überlegungen. Zunächst müssen die globalen Aussagen so erweitert werden, dass sie die Auswirkungen des Programmablaufs genauer modellieren. Das heißt insbesondere, dass ein Zusammenhang zwischen den Eigenschaften, die vor der Auswertung galten und den danach zutreffenden, hergestellt werden muss.

Beispiel 3.1. *Das in 2.3 formulierte gdj* $[res \leftarrow S]\ (res = c * d)$ *soll so umgeformt werden, dass das Verhältnis von Vor- und Nachbedingung ersichtlich wird.*

Schaut man sich das Beispiel an, so stellt man fest, dass für die Vorbedingung keine expliziten Angaben existieren. Wir beschreiben dies dadurch, dass die Vorbedingung zu „wahr“ (`True`) auswerten und keine weiteren Einschränkungen erfüllen muss. Die Nachbedingung ist gegeben durch `res = c * d`. Um den Zusammenhang zwischen diesen beiden Eigenschaften herzustellen, formulieren wir das gdj wie folgt:

$$[a \leftarrow ret\ True; res \leftarrow S; b \leftarrow ret(res = c * d)]\ (a \rightarrow b)$$

Zunächst wird also die Vorbedingung ausgewertet. Nach Abarbeitung der Programmsequenz wird die Nachbedingung in Abhängigkeit des Programmergebnisses ausgewertet. Diese muss sich dann aus der Vorbedingung herleiten lassen.

An diesem Beispiel lässt sich auch erkennen, welche Rolle die deterministische Seiteneffektfreiheit spielt. Die Auswertung der Vor- und Nachbedingung dürfen an der globalen Aussage nichts verändern, da sich dadurch auch die Bedingungen an sich ändern würden. Beide müssen daher deterministisch ohne Seiteneffekte auswerten.

So wie mit den gdj eine Möglichkeit geschaffen wurde, globale dynamische Aussagen hervorzuheben, ist es auch sinnvoll die besondere Bedeutung der Vor- und Nachbedingung durch eine geeignete Schreibweise kenntlich zu machen.

3.1. Hoare-Kalkül für imperative Programme

Die von Tony Hoare 1996 in seinem Aufsatz [Hoa69] eingeführten und mit der Zeit leicht veränderte Schreibweise der Hoare-Tripel sind eine, in der Verifikation recht verbreitete Möglichkeit, diesen Anforderungen gerecht zu werden. Ein solches Tripel der Form $\{\!|\,\Phi\,|\!\}\ \bar{x} \leftarrow \bar{p}\ \{\!|\,\Psi\,|\!\}$ schließt die zu verifizierende Programmsequenz $\bar{x} \leftarrow \bar{p}$ zwischen der Vor- und Nachbedingung (Φ bzw. Ψ) ein.

Definition 3.1. *Sei $\bar{x} \leftarrow \bar{p}$ eine Programm-Sequenz, Φ und Ψ boolesche Formeln. Dann ist das* ***Hoare-Tripel*** $\{\!|\Phi|\!\}\ \bar{x} \leftarrow \bar{p}\ \{\!|\Psi\bar{x}|\!\}$ *definiert als*

$$[a \leftarrow \Phi; \bar{x} \leftarrow \bar{p}; b \leftarrow \Psi\bar{x}]\ a \rightarrow b$$

Damit auf Grundlage dieser Tripel die Korrektheit eines Programms nachgewiesen werden kann, hat Hoare in [Hoa69] ein entsprechendes Kalkül eingeführt. Er beschränkt sich dabei auf eine kleine Teilmenge an imperativen Programmelementen:

Zuweisung:

$$\{\!|\Phi y|\!\}\ x \leftarrow y\ \{\!|\Phi x|\!\}$$

Konsequenzregeln:

$$\{\!|\Phi|\!\}\ x \leftarrow y\ \{\!|\Psi|\!\}\ \wedge\Psi \longrightarrow \xi \Longrightarrow\ \{\!|\Phi|\!\}\ x \leftarrow y\ \{\!|\xi|\!\}$$
$$\{\!|\Phi|\!\}\ x \leftarrow y\ \{\!|\Psi|\!\}\ \wedge\chi \longrightarrow \Phi \Longrightarrow\ \{\!|\chi|\!\}\ x \leftarrow y\ \{\!|\Psi|\!\}$$

Sequenzregel:

$$\{\!|\Phi|\!\}\ x \leftarrow p\ \{\!|\Psi|\!\}\ \wedge\ \{\!|\Psi|\!\}\ y \leftarrow q\ \{\!|\xi|\!\}\ \Longrightarrow\ \{\!|\Phi|\!\}\ x \leftarrow p; y \leftarrow q\ \{\!|\xi|\!\}$$

Iteration:

$$\{\!|\Phi \wedge b|\!\}\ x \leftarrow p\ \{\!|\Phi|\!\}\ \Longrightarrow$$
$$\{\!|\Phi|\!\}\ WHILE\ b\ DO\ x \leftarrow p\ \{\!|\Phi \wedge \neg b|\!\}$$

Die Iterations-Regel führt dazu, dass sich mit diesem Kalkül nur die partielle Korrektheit von Programmen zeigen lässt, da eine Terminierung der Schleife nicht bewiesen werden kann. In Kapitel 7.1 werden mögliche Erweiterungen aufgezeigt, die einen entsprechenden Regelsatz für totale Korrektheit ermöglichen.

Wie das Konzept des Hoare-Kalküls angewendet werden kann, um die partielle Korrektheit von Programmen nachzuweisen, soll an Hand eines Beispiel verdeutlicht werden.

Beispiel 3.2. *Es soll gezeigt werden, dass die Programmsequenz*

```
S = res← 0;i ← 0;
      WHILE (i<c) DO (i← (i+1);res← (res + d))
```

unserer, in Beispiel 2.3 entwickelten Spezifikation einer Multiplikation erfüllt.

Theorem 3.1. *Seien c und d beliebige Variablen vom Typ Integer. Dann gilt unter jeder Belegung von c und d, dass nach Abarbeitung von S* `res=c*d` *ist. Als Hoare-Tripel ausgedrückt:*

$$\{\!|True|\!\}\ S\ \{\!|res = c * d|\!\}$$

Beweis. Zunächst muss die Sequenz in ihre Einzelteile zerlegt werden. Dadurch erhalten wir drei Teilbeweise:

$\{\!|True|\!\}\ res \leftarrow 0\ \{\!|res = 0|\!\}$,

$\{\!|res = 0|\!\}\ i \leftarrow 0\ \{\!|res = 0 \wedge i = 0|\!\}$ und

$$\begin{array}{l} \{\!|res = 0 \wedge i = 0|\!\} \\ \quad WHILE(i < c)DO \\ \quad\quad i \leftarrow (i+1); res \leftarrow (res + d) \\ \{\!|res = c * d|\!\} \end{array}$$

Die ersten beiden Teilbeweise sind mit der Zuweisungsregel trivial. Für den Beweis über die While-Schleife müssen die Vor- und Nachbedingung so angepasst werden, dass die Iterationsregel anwendbar wird. Wir müssen daher eine passende Invariante, das heißt einen Term, der durch die Abarbeitung eines While-Durchlaufes nicht beeinflusst wird, finden. Die Gleichung `res+(c-i)*d = c*d` erfüllt diese Anforderung, da sich `res` mit jedem Durchlauf um d erhöht und `c-i` um eins verringert und die Gleichung für den Basisfall `i=0` und `res=0` wahr ist. Mit Hilfe der beiden Konsequenzregeln wird das Tripel deshalb umgeformt zu:

$$\begin{array}{l} \{\!|res + (c-i) * d\ =\ c * d|\!\} \\ \quad WHILE(i < c)DO \\ \quad\quad i \leftarrow (i+1); res \leftarrow (res + d) \\ \{\!|res + (c-i) * d\ =\ c * d \wedge \neg i < c|\!\} \end{array}$$

Durch die Iterationsregel reicht es zu zeigen, dass folgendes Hoare-Tripel korrekt ist:

$$\begin{array}{l} \{\!|res + (c-i) * d\ =\ c * d \wedge i < c|\!\} \\ \quad i \leftarrow (i+1); res \leftarrow (res + d)) \\ \{\!|res + (c-i) * d\ =\ c * d|\!\} \end{array}$$

Auch hier kommt wieder die Sequenzregel zum Einsatz und man erhält:

$$\begin{array}{l} \{\!|res + (c-i) * d\ =\ c * d \wedge i < c|\!\} \\ \quad i \leftarrow (i+1); res \leftarrow (res + d)) \\ \{\!|(res + d) + (c - (i+1) * d\ =\ c * d|\!\} \end{array}$$

Was sich durch die erste Konsequenzregel in das gewünschte Tripel umwandeln lässt. □

3.2. monadisches Hoare-Kalkül

Im Laufe der Zeit wurde das Hoare-Kalkül erweitert und an spezielle Anwendungen angepasst. In [SM03] stellen Lutz Schröder und Till Mossakowski eine für die Verifikation von monadischen Programmen geeignete Variante vor. Diese diente als direkte Vorlage für das in Abbildung 3.1 beschriebene Kalkül.

(dsef) $$\frac{dsef\ q}{\{\!|\Phi|\!\}\ q\ \{\!|\Phi|\!\}}$$

(ctr) $$\frac{\{\!|\Phi|\!\}\ x \leftarrow p; y \leftarrow q; z \leftarrow r\ \{\!|\Psi yz|\!\}}{\{\!|\Phi|\!\}\ y \leftarrow do\{x \leftarrow p; q\}; z \leftarrow r\ \{\!|\Psi yz|\!\}}$$

(stateless) $$\frac{}{\{\!|ret\Phi|\!\}\ p\ \{\!|ret\Phi|\!\}}$$

(seq) $$\frac{\{\!|\Phi|\!\}\ x \leftarrow p\ \{\!|\Psi|\!\} \quad \forall x.\{\!|\Psi x|\!\}\ y \leftarrow qx\ \{\!|\xi xy|\!\}}{\{\!|\Phi|\!\}\ x \leftarrow p; y \leftarrow qx\ \{\!|\xi xy|\!\}}$$

(conj) $$\frac{\{\!|\Phi|\!\}\ x \leftarrow p\ \{\!|\Psi x|\!\} \quad \{\!|\Phi|\!\}\ x \leftarrow p\ \{\!|\xi x|\!\}}{\{\!|\Phi|\!\}\ x \leftarrow p\ \{\!|\Psi x \wedge \xi x|\!\}}$$

(disj) $$\frac{\{\!|\Phi|\!\}\ x \leftarrow p\ \{\!|\xi x|\!\} \quad \{\!|\Psi|\!\}\ x \leftarrow p\ \{\!|\xi x|\!\}}{\{\!|\Phi \vee \Psi|\!\}\ x \leftarrow p\ \{\!|\xi x|\!\}}$$

(wk) $$\frac{\Phi' \Rightarrow_h \Phi \quad \forall x.\Psi x \Rightarrow_h \Psi' x \quad \{\!|\Phi|\!\}\ x \leftarrow p\ \{\!|\Psi x|\!\}}{\{\!|\Phi'|\!\}\ x \leftarrow p\ \{\!|\Psi' x|\!\}}$$

(if) $$\frac{\{\!|\Phi \wedge b|\!\}\ x \leftarrow p\ \{\!|\Psi x|\!\} \quad \{\!|\Phi \wedge \neg b|\!\}\ x \leftarrow p\ \{\!|\Psi x|\!\}}{\{\!|\Phi|\!\}\ x \leftarrow if_T\ b\ then\ p\ else\ q\ \{\!|\Psi x|\!\}}$$

(iter) $$\frac{\{\!|\Phi\ x\ \wedge (b\ x)|\!\}\ \ y \leftarrow p\ x\ \{\!|\Phi\ y|\!\}}{\{\!|\Phi\ x|\!\}\ \ y \leftarrow \mathrm{iter}_{\mathrm{T}}\, b\ p\ x\ \{\!|\Phi\ y \wedge \neg(b\ y)|\!\}}$$

Abbildung 3.1.: Monadischer Hoare-Kalkül

Zwar stellte Hoare in [Hoa69] seine Regeln nur in axiomatischer Form da, um die Korrektheit von Programmen zu zeigen ist es allerdings notwendig, die Korrektheit des verwendeten Kalküls ebenfalls nachzuweisen.

Die Schleifen-Regel wird in dieser Arbeit noch axiomatisch formuliert, da der Beweis recht umfangreich ist und den Rahmen sprengen würde.

Behauptung 3.1. *Für alle seiteneffektfrei auswertenden booleschen Formeln Φ, Ψund ξ und alle monadischen Programme p, q, r und b wobei b zu `bool T` auswertet, ist das in Abbildung 3.1 formulierte Kalkül korrekt.*

Dabei ist $\Phi' =>_h \Phi$ syntaktischer Zucker für das Hoare-Tripel mit leerer Sequenz und Φ' und Φ als Vor- und Nachbedingungen.

Beweis.

dsef: Da Φ deterministisch seiteneffektfrei ist, kann nach Anwendung der Definitionen für deterministische Seiteneffektfreiheit und Vertauschbarkeit gezeigt werden, dass gilt:

$$cp\ (do\{a \leftarrow \Phi; y \leftarrow q; ret(a, y)\}).$$

Durch Anwendung von Lemma 2.19 kann dies zunächst in die dort beschriebene erste Form (`i`) umgewandelt werden und in einem weiteren Schritt zu:

$$[a \leftarrow \Phi; y \leftarrow q; b \leftarrow \Phi]\ (a = b).$$

Die Anwendung von Lemma 2.4 führt zur gdj-Variante der Behauptung.

ctr: Der Beweis erfolgt auf Basis der gdj-Form von Voraussetzung und Behauptung. Zunächst wird mit Hilfe von Lemma 2.2 die interne Klammerung im Rückgabe-Tupel aufgehoben. Mit dem auf den gdj definierten Lemma 2.8 lässt sich die Behauptung dann zeigen.

stateless: Da der Variablen Φ kein neuer Wert zugewiesen wird, muss er nach Abarbeitung von p der gleiche sein wie zuvor.

seq: Die beiden Voraussetzungen lassen sich durch Lemma 2.5 bzw. 2.6 erweitern zu

$$[a \leftarrow \Phi; x \leftarrow p; b \leftarrow \Psi\ x; y \leftarrow q\ x; c \leftarrow \xi\ x\ y]\ (a \Rightarrow b) \qquad \text{und}$$
$$[a \leftarrow \Phi; x \leftarrow p; b \leftarrow \Psi\ x; y \leftarrow q\ x; c \leftarrow \xi\ x\ y]\ (b \Rightarrow c).$$

Durch Lemma 2.3 erhalten wir als boolesche Formel ($a \Rightarrow b \wedge b \Rightarrow c$). Durch Lemma 2.4 lässt sich das vereinfachen zu:

$$[a \leftarrow \Phi; x \leftarrow p; b \leftarrow \Psi\ x; y \leftarrow q\ x; c \leftarrow \xi\ x\ y]\ (a \Rightarrow c).$$

Nach Lemma 2.9 kann nun Ψx noch aus der Sequenz entfernt werden und wir erhalten die Behauptung.

conj: Aus den beiden Vorbedingungen lassen sich durch Lemma 2.6 und 2.20 bzw.

nur durch Lemma 2.6 die beiden Formeln

$$[a \leftarrow \Phi; c \leftarrow \Psi; x \leftarrow p; b \leftarrow \xi\ x]\ (a \Rightarrow b) \qquad \text{und}$$
$$[a \leftarrow \Phi; c \leftarrow \Psi; x \leftarrow p; b \leftarrow \xi\ x]\ (c \Rightarrow b)$$

herleiten. Die durch Lemma 2.3 entstehende Formel lässt sich durch Lemma 2.4 umformen zu

$$[a \leftarrow \Phi; c \leftarrow \Psi; x \leftarrow p; b \leftarrow \xi\ x]\ (a \vee c \Rightarrow b).$$

Mit Hilfe von Lemma 2.7 kann das $\vee$ nun in das gdj gezogen werden und durch Anwendung der Definition des Hoare-Tripels dann die Behauptung gezeigt werden.

disj: Der Beweis verläuft äquivalent zum `conj`-Beweis.

wk: Die aufgefaltete erste Bedingung kann durch Lemma 2.6 erweitert werden zu

$$[a \leftarrow \Phi'; b \leftarrow \Phi; x \leftarrow p; c \leftarrow \Psi x]\ (b \Rightarrow c),$$

die zweite durch Lemma 2.5 zu

$$[a \leftarrow \Phi'; b \leftarrow \Phi; x \leftarrow p; c \leftarrow \Psi x]\ (a \Rightarrow b).$$

Durch Lemma 2.3 und Lemma 2.4 erhält man

$$[a \leftarrow \Phi'; b \leftarrow \Phi; x \leftarrow p; c \leftarrow \Psi x]\ (a \Rightarrow c).$$

Nachdem Φ mit Hilfe von Lemma 2.9 aus der Sequenz entfernt wurde, können die obigen Schritte für die Kombination der entstandenen Sequenz mit der Voraussetzung angewendet werden. Wir erhalten:

$$[a \leftarrow \Phi'; b \leftarrow \Phi; x \leftarrow p; d \leftarrow \Psi' x]\ (a \Rightarrow d)$$

Dies ist nichts anders als die gdj-Schreibweise der Behauptung.

if: Dieser Teilbeweis wird auf Basis des ausgeschlossenen Dritten gezeigt. Diese Regel besagt, dass ein boolescher Ausdruck nach vollständiger Auswertung genau einen der beiden Werte „wahr" oder „falsch" ergibt.

Für boolesche Monaden gilt diese Regel nicht, da neben den beiden Wahrheitswerten Seiteneffekte auftreten können. Um auf die rein booleschen Werte zurückgreifen zu können, erweitern wir die Vorbedingung durch Lemma 2.4 zu

$(\Phi \wedge b) \wedge (\Phi \wedge \neg b)$

Das dadurch entstandene Hoare-Tripel lässt sich durch `disj` aufspalten und durch Anwendung von Definition 2.2 umwandeln in:

$\{\!|\Phi \ \wedge \ b|\!\}\ v \leftarrow b; x \leftarrow if\ v\ then\ p\ else\ q\ \{\!|\Psi\ x|\!\}$ und
$\{\!|\Phi \ \wedge \ \neg b|\!\}\ v \leftarrow b; x \leftarrow if\ v\ then\ p\ else\ q\ \{\!|\Psi\ x|\!\}$

Nach Anwendung der Sequenzregel kann der zweite Teil jeweils durch folgende Regeln unter Anwendung der Voraussetzungen bewiesen werden.

(if_True) $$\frac{\{\!|(\Phi \wedge b) \wedge v|\!\}\ x \leftarrow p\ \{\!|\Psi\ x|\!\}}{\{\!|(\Phi \wedge b) \wedge v|\!\}\ x \leftarrow (if\ v\ then\ p\ else\ q) \{\!|\Psi\ x|\!\}}$$

(if_False) $$\frac{\{\!|(\Phi \wedge \neg b) \wedge \neg v|\!\}\ x \leftarrow q\ \{\!|\Psi\ x|\!\}}{\{\!|(\Phi \wedge \neg b) \wedge \neg v|\!\}\ x \leftarrow (if\ v\ then\ p\ else\ q) \{\!|\Psi\ x|\!\}}$$

Da beide Regeln innerhalb der Sequenz auf booleschen Werten arbeiten, lassen sie sich über das ausgeschlossene Dritte herleiten.

Der erste, durch die Anwendung der Sequenzregel enstandene Teil, lässt sich jeweils Anwendung der Hoare-Definition zeigen.

□

Die Konsequenzregel aus diesem Kalkül fasst die beiden von Hoare vorgestellten Regeln in einer zusammen. Die beiden ursprünglichen Regeln sehen wie folgt aus:

(wk_pre) $$\frac{\{\!|\Phi|\!\} x \leftarrow p \{\!|\Psi\ x|\!\} \quad \Phi' \Rightarrow_T \Phi}{\{\!|\Phi'|\!\} x \leftarrow p \{\!|\Psi\ x|\!\}}$$

(wk_post) $$\frac{\{\!|\Phi|\!\} x \leftarrow p \{\!|\Psi\ x|\!\} \quad \forall x.\ (\Psi\ x) \Rightarrow_T (\Psi'\ x)}{\{\!|\Phi|\!\} x \leftarrow p \{\!|\Psi'\ x|\!\}}$$

Sie lassen sich ohne weiteres aus Lemma 2.4 herleiten, da mit Hilfe von Lemma 2.12 die jeweils andere Bedingung einfach in sich selbst überführt werden kann.

4. Isabelle

Der Korrektheitsbeweis für Beispiel 3.2 war eine kleine Demonstration, wie mit Hilfe des Hoare-Kalküls ein Programm gegenüber seiner Spezifikation bewiesen werden kann. Wie in Abschnitt 1.3 beschrieben, soll das Anwendungsgebiet der Arbeit bei der Verifikation von grösseren Programmstücken bis hin zu ganzen Software-Systemen liegen. Um einen solchen Umfang bewältigen zu können, reichen Beweise mit Stift und Papier wie Kapitel 2 und 3 nicht mehr aus. Die Gefahr ist dann hoch, dass sich Flüchtigkeitsfehler einschleichen.

Zu dem muss man sich mit Möglichkeiten der Dokumentation auseinandersetzen. Zwar sind formale Beweise universell verständlich, eine begleitende Dokumentation vereinfacht jedoch die Lesbarkeit und die Akzeptanz der Beweisstruktur. Bei der Arbeit mit Stift und Papier muss der Beweis zu Dokumentationzwecken neu aufgearbeitet werden. Seit einigen Jahren werden verstärkt Computer eingesetzt, um dem Menschen bei diesen Aufgaben zu unterstützen. Computer können zwar nicht die koplette Beweisführung übernehmen [1], sie können diese aber vereinfachen.

Der in Kooperation zwischen der Universitäten München und Cambridge entwickelte halbautomatische Theorembeweiser *Isabelle* [isa05] verfolgt diese Ziele unter anderem durch

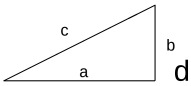
- Kapselung zusammengehöriger Objekte und Beweise zu Theorien
- Bereitstellung fertiger in einem Repository [isa05], auf die die eigenen Beweise aufgesetzt werden können.
- mathematische Beweisform: Während in den ersten Isabelle-Versionen die Beweise noch in der funktionalen Sprach ML formuliert wurden, ist seit Isabelle2002 mit der Unterstützung von Beweisen in Isabelle/Isar eine Möglichkeit integriert worden um Beweise nahe der mathematischen Formulierung umzusetzen.
- Zweistufiges Dokumenations-System: neben der reinen Code-Dokumentation stellt Isabelle auch *JavaDoc*-ähnliche Kommentare zur Verfügung,

Dieses Kapitel gibt eine kurze Einführung in das Beweissystem. Der Schwerpunkt liegt dabei auf den für diese Arbeit verwendeten Eigenschaften und Methoden.

[1] Dies zeigte Gödel bereits 1931 mit seinem Unvollständigkeitssatz.

Eine ausführliche Beschreibung würde den Rahmen dieser Arbeit sprengen. Die Internet-Präsenz des Isabelle-Projektes [isa05] bietet aber umfangreiches Material. Neben den Manuals zum Grundsystem [WB04], [Pau04] und Isabelle/Isar [Wen04] finden sich dort auch verschiedene Tutorials [Nip03]. Eine umfangreiche Einführung in die Umsetzung der Prädikaten-Logik höherer Ordnung (engl. **h**igher **o**rder **l**ogic - HOL) in Isabelle bietet [NPW02].

4.1. Theorien

Bei einem mathematische Beweis wird man selten weiter formal argumentieren, wenn lediglich noch zu zeigen ist, dass bspw. für ein rechtwinkliges Dreieck der Form 

die Gleichung $a^2 + b^2 = c^2$ gilt. Schließlich ist diese Eigenschaft ja im „Satz des Pythagoras“ manifestiert. Es reicht also ein Verweis auf den Satz und man muss den konkreten Beweis weder gesehen noch im Detail verstanden haben. Auch besteht nicht die Notwendigkeit zu definieren, was man mit einem rechtwinkligen Dreieck, mit a^2 oder „+“ und „=“ bezeichnen möchte. Diese Begriffe stehen bereits zur Verfügung.

Da man auch in der formalen Spezifikation und Verifikation das Rad nicht ständig neu erfinden will, bietet Isabelle die Möglichkeit auf vorhandenen Sachen aufzubauen. Dazu werden Objekt-Definitionen und Aussagen über diese Objekte in Theorien zusammengefasst, die dann referenziert werden können. Theorien sind die größte Verarbeitungseinheit in Isabelle. Das Grundgerüst einer solchen Theorie sieht wie folgt aus:

```
theory MyThy = thyA + thyB:

end
```

„MyThy“ ist dabei der Name der neuen Theorie, die in diesem Fall auf die Definitionen und Beweise von thyA und thyB zurückgreift. Durch dieses System der Referenzierung entsteht eine Hierarchie von Abhängigkeiten an deren Spitze die Theorie `Pure` steht. Sie stellt lediglich eine Metalogik bestehend aus der Implikation „==>“, der Allquantisierung „/\“ und der Konjunktion „[|P;Q|]“ zur Verfügung.

Aufbauend auf diesen elementaren Bausteinen sind eine ganze Reihe von Logiken in Isabelle implementiert worden. So können zum Beispiel durch `FOL` und `HOL` prädikatenlogische Aussagen erster bzw. höherer Ordnung formuliert werden. Meist wird jedoch nicht direkt auf diese beiden Theorien zurückgegriffen sondern Main als Grundlage gewählt. Main stellt neben `FOL` und `HOL` noch häufige benötigte Spezifikationen zur Verfügung wie zum Beispiel Listen, Mengen und natürliche Zahlen.

4.2. Spezifikation und Verifikation mit Isabelle

Eine typische Theorie besteht aus zwei Teilen:

- Einführung von Typen und Konstanten (Funktionen)
- logische Aussagen über diese Objekte

Die Einführung eines neuen Typs kann auf verschiedene Weisen geschehen. Die Theorien dieser Arbeit verwenden zwei durch die Theorie *HOL* bereitgestellte Methoden `typedecl` und `typedef`.

Die Anwendung von `typedecl` bietet die Möglichkeit, einen neuen Typ einzuführen ohne genauere Angaben über die mit ihm verbundenen Eigenschaften zu machen. Eine Deklaration der Form:

typedecl 'a T

führt zum Beispiel einen polymorphen Typ mit dem Namen T ein.

Mit Hilfe von `typedef` lassen sich Untertypen definieren. Da Typen in *HOL* mindestens ein Element haben müssen, geht mit der Typ-Definition eine entsprechende Beweispflicht einher. Eine Definition der Form

typedef (Dsef) ('a) D = "{p::'a T. dsef p}"

beschreibt den Type `'a D` als die Untermenge von `'a T`, die die Eigenschaft `dsef` erfüllt. Neben der Konstanten `Dsef`, die die Menge aller Elemente des Untertyps enthält, stehen noch zwei Funktionen zur Verfügung, die die Umwandlung zwischen den Typen ermöglichen:

Abs_Dsef :: 'a T => 'a D
Rep_Dsef :: 'a D => 'a T

Die Regeln für die Umwandling stehen durch die Lemmata `Rep_Dsef_inverse` und `Abs_Dsef_inverse` zur Verfügung:

$$\text{Abs\_Dsef (Rep\_Dsef p) = p}$$
$$\text{q} \in \text{Dsef} \Rightarrow \text{Rep\_Dsef (Rep\_Dsef q) = q}$$

Auch für die Einführung von Konstanten stehen verschiedene Möglichkeiten zur Verfügung. Neben der Konstanten-Definition durch `consts` und `defs` wird in den Theorien dieser Arbeit mit Syntax-Übersetzung (`syntax`, `translations`) gearbeitet.

Durch `consts` wird zunächst nur die Typisierung vorgenommen und optional eine alternative Syntax angegeben. Die durch

consts
bind :: "'a T $\Rightarrow$ ('a $\Rightarrow$ 'b T) $\Rightarrow$ 'b T" ("_ $\gg$= _")

eingeführte Konstante erwartet zwei Parameter und kann als `bind p q` oder in Infix-Schreibweise p $\gg=$ q verwendet werden. Mit `defs` wird der Konstante eine Semantik zugeordnet. Diese beiden Schritte können durch `constdefs` zusammengefasst werden wie das folgende Beispiel zeigt:

constdefs
bind' :: "'a T $\Rightarrow$ 'b T $\Rightarrow$ 'b T" ("_ $\gg$ _")
"bind' p q == bind p (λx. q)"

Bei der Syntax-Übersetzung wird zunächst wieder der Typ festgelegt. Dies geschieht durch das Schlüsselwort `syntax`. Danach können durch `translations` Übersetzungsregeln festgelegt werden. Eine alternative Umsetzung von `bind'` hat dann die Form:

syntax
bind' :: "'a T $\Rightarrow$ 'b T $\Rightarrow$ 'b T" ("_ $\gg$ _")

translations
"bind' p q" == "bind p (_K q)"

Eine Syntax-Übersetzung verhindert im Gegensatz zur Konstanten-Definition die Einführung neuer Variablen. Mit _K kann zwar ein Lambda-Term bereitgestellt werden, sobald man aber die eingeführte Variable ein zweites Mal verwenden möchte, muss man eine Typ-Definition verwenden.

Allerdings hat die Übersetzung den entscheidenen Vorteil, dass sie bei der Auswertung von Termen automatisch berücksichtigt wird. Die Konstaten-Definition muss dagegen explizit durch `unfold` aufgefaltet werden.

Syntax-Übersetzungen eigenen sich daher vor allem für die Implementierung von *syntaktischem Zucker*, während eigenständige Konstanten besser durch eine Definition umgesetzt werden.

Durch die bereitgestellten Objekten lassen sich die mit ihnen verknüpften Regeln formulieren. Die Grundlage bieten wie in der Mathematik Axiome.

axioms
injective: "ret x = ret z $\Longrightarrow$ x = z"

Aufbauend auf diesen Grundregeln können dann Lemmata und Theorem formuliert werden. Es gibt prinzipiell zwei Vorgehensweisen bei der Beschreibung und dem Beweis solcher Regeln. Die ursprüngliche Methode stellt die Anwendung der Beweisregeln (`apply <rule>`) in den Vordergrund. Diese so genannten Taktik-Skripte besitzen keine Angaben zu Zwischenschritten des Beweises, wie das folgende Beispiel zeigt:

```
lemma "sef_retUnit": "(p = ret ()) = sef p"
  apply (rule iffI)
  apply (simp_all add: sef_def)
done
```

Gerade bei langen Beweisen geht die Lesbarkeit schnell verloren. Der in [Wal05] präsentierte Korrektheits-Beweis für die russische Multiplikation demonstriert dieses Phänomen eindrucksvoll. Der über hundert Zeilen lange Taktik-Beweis hat nichts mehr mit mathematischen Beweisen gemein. Der reine Isabelle-Code lässt sich ohne den Beweiser und die durch ihn generierten Beweiszustand, nicht mehr verstehen.

Die Sprache Isabelle/Isar, bietet die Möglichkeit Beweise nahe der mathematischen Schreibweise zu formulieren. Ein typischer Isabelle/Isar-Beweis hat die folgende Form:

```
lemma "sef_retUnit" : "(p = ret ()) = sef p"
  proof
    assume "p = ret ()"
    from this have "sef (ret ())"
      by (simp add: sef_def)
    from prems this show "sef p"
      by blast
  next
    assume sef_p: "sef p"
    from sef_p have "p = do {y←p;ret ()}"
      by simp
    moreover
    from prems this have "p = ret ()"
      apply (unfold sef_def)
      by simp
    ultimately show "p = ret ()"
      by blast
  qed
```

Die einzelnen Beweiselemente sind dabei wie folgt zu verstehen:

lemma, theorem: Diese Schlüsselwörter leiten die Umsetzung einer zu beweisenden logischen Aussage ein. Zur späteren Referenzierung kann ein Name vergeben werden. Danach wird die Aussage formuliert. Dieses kann in verschiedenen Formen geschehen. Neben einer Gleichung bzw. Implikation wie im obigen Beispiel lassen sich Voraussetzung und Behauptung auch durch die Schlüsselwörter `assumes` und `shows` einzeln hervorheben.

proof ... **qed:** Mit `proof -` wird ein Beweis eröffnet, `qed` schliesst ihn ab. Wird `proof` wie Beispiel ohne „-“ verwendet, so versucht Isabelle den ersten Beweisschritt automatisch vorzunehmen. Im Beispiel werden durch `IffI` direkt die beiden zu beweisenden Richtungen aufgeteilt.

assume: Während des Beweises können durch `assume` Annahmen formuliert werden, die jedoch vor Abschluss aus den Voraussetzungen hergeleitet werden müssen.

from ... **have:** Zwischenschritte in der Beweisführung lassen sich durch `have` formulieren. Mit `from` können Voraussetzungen oder schon bewiesene Schritte als Voraussetzungen für den Teilbeweis herangezogen werden. Ein solchen Zwischenschritt kann zur späteren Referenzierung innerhalb der aktuellen Beweisebene mit einem Namen versehen werden. Im Beispiel wurde „sef p“ mit dem Namen `sef_p` versehen.

this, prems: Diese Referenzen werden von Isabelle zur Verfügung gestellt. Mit `this` kann dabei der vorangengangene Teilbeweis referenziert werden, mit `prems` die Voraussetzungen des aktuellen Teilbeweises. Daneben lässt sich durch `?thesis` auch auf die Behauptung verweisen. Diese Referenzen lassen sich durch die Namensgebung nach dem Schlüsselwort `have` erweitern.

apply, by: Die Regelanwendung erfolgen wie beim Taktik-Beweis durch `apply`. Das Schlüsselwort `by` ist für die finale Regelanwendung reserviert und kürzt `apply ... done` ab.

simp, blast: Isabelle wendet durch diese Methoden automatisch einen Satz von Regeln auf das aktuelle Beweisziel an. `simp` versucht dabei mit Hilfe des *simpset* offene Beweisziele zu vereinfachen. Durch Angabe des `[simp]`-Attributs beim erstellen eines Lemmas wird dieses zum *simpset* hinzugefügt. Durch

```
simp add: <rule>
simp del: <rule>
simp only: <rule>
```

lässt sich der *simpset* lokal modifizieren und zusätzliche Regeln hinzufügen, nicht erwünschte Regelanwendungen unterdrücken oder eine einzelne Regel zur Vereinfachung nutzen.

Die Methode `blast` wendet Einführungs- und Eliminations-Regeln aus dem `claset` um das aktuelle Beweisziel herzuleiten. Sie kann wie `simp` im konkreten Fall angepasst werden durch `add`, `del` oder `only`.

`auto` verwendt eine Kombination beider Methoden.

show: Dieses Schlüsselwort leitet den finalen Teilbeweis ein.

next: Leitet den nächsten Fall bei einer Fallunterscheidung ein.

... **moreover** ... **ultimately** ...**:** Diese Schlüsselwörter beschreiben eine Beweisfolge. Durch die mit `moreover` verknüpften Teilbeweise lässt sich das durch `ultimately` gekennzeichneten Beweisziels herleiten.

unfold: Definitionsbeweise werden mit `unfold` bzw. `fold` auf das aktuelle Beweisziel angewendet.

Isar bietet die Möglichkeit, Taktik-Skripte einzubeziehen. Dies hat zwei entscheidende Vorteile:

- ältere Taktik-basierte Beweise können ohne Überarbeitung weiterentwickelt und in neuere Theorien integriert werden.
- man muss die Zwischenziele nicht in alle Einzelheiten zerlegen sondern kann sie so weit aufteilen, wie es der Verständlichkeit zuträglich ist. Uninteressante Teilbeweise können dann durch Taktik-Anwendungen gezeigt werden.

Die Kunst beim Schreiben lesbarer Beweise in Isabelle liegt also darin, die richtige Abstraktionsebene zu finden, auf der man sich bewegen will.

Zu dem erhöht eine gute Dokumentation die Verständlichkeit. Isabelle bietet dazu verschiedene Möglichkeiten an. Die einfachste ist die reine Incode-Dokumentation durch `(*<Text>*)` Diese sollte aber nur für kleinere Notizen und persönliche Hinweise dienen. Um den Code für andere verständlich zu machen, bietet Isabelle ein ähnliches Konzept wie *JavaDoc* an. Mit Hilfe des Werkzeugs `isatool make` wird eine LaTeX und eine HTML-Variante des Codes erstellt.

Die Incode-Dokumentation wird dabei nicht mit publiziert. Nur Kommentare die durch `text{*<Text>*}` bzw. `txt{*<Text>*}` gekennzeichnet sind werden von `isatool` bearbeitet.

Die Vorteile dieser Dokumentations-Methode liegen auf der Hand. Wird im Isabelle-Code etwas verändert, so reicht ein Aufruf des Werkzeugs und ein Kompilieren der Arbeit aus, um die Dokumentation auf den neuesten Stand zu bringen. Zu dem hat man die Gewissheit, dass in Isabelle formulierte Beweise, die direkt eingebunden werden, funktionstüchtig sind. Sie werden einfach in Isabelle formuliert und danach so wie sie sind in die Arbeit eingebunden.

Eine Ausarbeitung wie diese lässt auch komplette als Isabelle-Kommentare verwirklichen. Dies hätte allerdings zwei große Nachteile:

- Sonderzeichen in den Kommentaren (z.B. Φ, $\leftarrow$ und $\vee$) müssen durch Antiquotations gekennzeichnet werden.

- Werden die erstellten Theorien von anderen weiterverwendet, sind die langen Texte störend. So ist eine Isabelle-Einführung, wie sie in diesem Kapitel gegeben wird im Quellcode unnötig.

Man muss also eine gesunde Mischung zwischen Dokumentation im Code und externen Dokumenten finden.

4.3. zusätzliche Paket

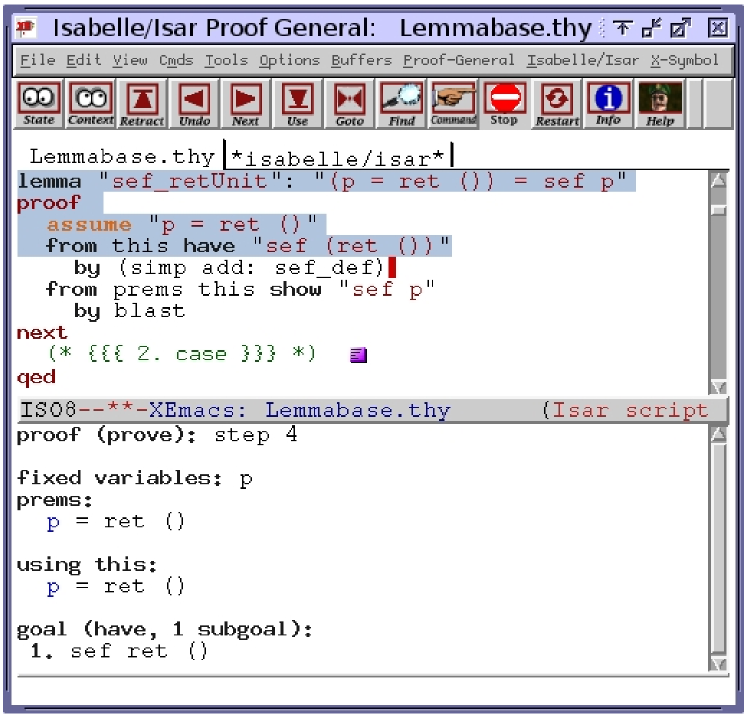


Abbildung 4.1.: ProofGeneral

Isabelle an sich ist ein reines Kommandozeilen-Werkzeug und damit eher unkomfortabel in der Benutzung. Um Isabelle sinnvoll einsetzen zu können macht es daher Sinn, neben dem eigentlichen Beweiser und dem ML-Compiler einige Pakete zur Benutzerunterstützung zu installieren.

ProofGeneral

Zwei der hilfreichsten Pakete sind die beiden XEmacs-Erweiterungen *ProofGeneral* [Asp05] und *X-Symbols* [Wed03]. Ersteres stellt eine grafische Schnittstelle für die Arbeit mit verschiedenen Beweisern zur Verfügung. Abbildung 4.1 zeigt den *ProofGeneral* bei der Bearbeitung einer Isabelle-Theorie. Auf häufig benötigte Isabelle-Befehle wie `Next`, `Undo` oder `Goto` kann über ein Menü zugegriffen werden.

Zu dem wird der aktuelle Beweis-Zustand im `*isabelle/isar*`-Puffer ausgegeben und die schon bearbeitete Schritte des Beweises eingefärbt. Somit behält man den Überblick über offene Beweis-Ziele und eventuell aufgetretene Fehler in der Beweisführung.

Durch die Installation des ProofGeneral erscheint beim Öffnen einer Isabelle Theorie der Menü-Punkt `Isabelle/Isar`. Mit ihm ist es möglich Einstellungen am Isabelle-System vorzunehemen. Dies beinhaltet unter anderem die bei der Fehlersuche hilfreichen Optionen zu Typ- und Klammerungsangaben für die Ausgabe des Beweiszustandes.

X-Symbols

Das Paket X-Symbols ersetzt Sonderzeichen durch eine lesbare Form. So wandelt er zum Beispiel „$< -$“ um in $\leftarrow$ und „$\backslash < Phi >$“ wird angezeigt als Φ. Zu dem lassen sich Indices durch höher und tiefer gestellte Zeichen visualisieren.

folding-mode

Da Isabelle-Beweise teilweise recht lang und unübersichtlich werden können, ist es zu dem ratsam, den folding-mode für XEmacs zu installieren. Dieser sorgt dafür, dass der Text zwischen zwei festgelegten Textbausteinen ausgeblendet werden kann.

Die Isabelle-Theorien dieser Arbeit sind mit entsprechenden Markern versehen. Durch **(*** $\{\{\{ <$ **Beschreibung des Abschnitts**$>$ $\}\}\}$ ***)** und **(*** $\}\}\}$ ***)** umschlossene Textabschnitte können bei aktivem folding-mode durch Rechtsklick auf den öffnenden Marker ausgeblendet werden. Im XEmacs wird dann lediglich die Kurzbeschreibung angezeigt. In Abbildung 4.1 wurde zum Beispiel der zweite Fall des Beweises zusammengefaltet. Eine leicht verständliche Installations-Anleitung für diesen Emacs-Mode findet sich unter [FM].

5. Umsetzung in Isabelle/Isar

Dieses Kapitel beschreibt die konkrete den vorangengangenen Kapiteln Umsetzung der vorgestellten Ideen in Isabelle/Isar auf Grundlage der Theorie *Main*. *Main* beinhaltet neben der Prädikatenlogik höherer Ordnung (engl. higher order logic - *HOL*) auch Erweiterungen wie Listen, Mengen und natürliche Zahlen. Dadurch lässt sich zum Beispiel der Einheitstyp für Monaden `()` durch den bereits in *HOL/-Product_Type.thy* definierte Typ `Unit` beschreiben.

Die Implementierung der monadischen Hoare-Logik ist in vier Stufe unterteilt: die Umsetzung von Monaden, globalen Zusicherungen und den Eigenschaften deterministisch seiteneffektfreier Programme bilden die Grundlage für die Formulierung und den Beweis des Hoare-Kalküls.

Die vier Teile sind jeweils in zwei Theorien aufgeteilt. Die Syntax und grundlegende semantische Eigenschaften werden dabei vom eigentlichen Regelsatz getrennt. Die acht dadurch entstandenen Theorien erfüllen im Einzelnen folgende Aufgaben:

MonadSyntax Neben dem Typkonstruktor `T` und den monadischen Funktionen `ret` und $\gg=$ werden in dieser Theorie auch die Monaden-Gesetze implementiert. Da sich Seiteneffektfreiheit, Kopierbarkeit und Vertauschbarkeit auf viele Lemmata auswirkt, werden diese Monaden-Eigenschaften ebenfalls in *MonadSyntax* eingeführt. Abschließend werden durch Lifting die Kontrollstrukturen `if` und `while` für die Anwendung auf Monaden bereitgestellt.

Lemmabase Alle grundlegenden Lemmata über die Verwendung von Monaden sind in dieser Theorie zusammengefasst.

gdjSyntax Es werden zwei Varianten der gdj-Schreibweise eingeführt. Neben der in Abschnitt 2.3 vorgestellten wird der Spezialfall der leeren Sequenz implementiert. Das Parsen längerer Programmsequenzen und die Übersetzung in eine für die Definition geeignete Sequenz der Länge eins sind ebenfalls in *gdjSyntax* umgesetzt.

gdjCalc Neben den Regeln, die über die gdj formuliert sind, finden sich in dieser Theorie noch einige grundlegende Lemmata.

dsefSyntax Die Definition der deterministischen Seiteneffektfreiheit wird ergänzt durch die Einführung eines Untertyps der Monaden, der diese Eigenschaft erfüllt. Zu dem werden Lemmata für die Umwandlung von Elementen des neuen Typs in Monaden und umgekehrt beschrieben. Ein entsprechendes

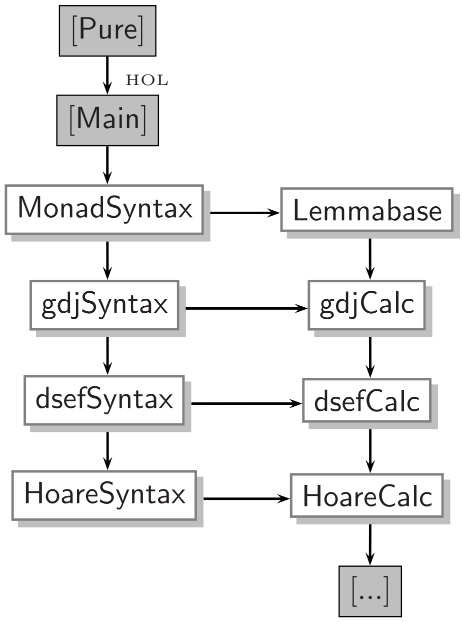
Lifting für die Kontrollstrukturen sowie für die booleschen Operatoren $\wedge$, $\vee$ und $\neg$ schließt die Theorie ab.

dsefCalc Lemmata, die die deterministische Seiteneffektfreiheit ausnutzen werden an dieser Stelle eingeführt und bewiesen. Insbesondere Möglichkeiten zum Verwerfen und Vertauschen von Teilsequenzen werden beleuchtet.

HoareSyntax Neben der in Kapitel 3 vorgestellten Form des Hoare-Tripels werden in dieser Theorie zwei syntaktische Varianten eines Hoare-Tupels eingeführt. Dieses beinhaltet lediglich die Vor- und Nachbedingung.

HoareCalc Das in Abschnitt 3.2 beschriebene Hoare-Kalkül wird in dieser Theorie bewiesen.

Die Theorien sind dabei in folgender Weise voneinander abhängig:



In den folgenden Abschnitten wird der Inhalt der Isabelle-Theorien näher beleuchtet. Da durch Spezialfälle bei der Syntax-Übersetzung umfangreiche Regelsätze entstanden sind, wird die Einführung neuer Syntax jeweils mit einem konkreten Beispiel hinterlegt. Dabei werden in den Beispielen die entsprechenden Regelanwendungen referenziert durch (Anw. <Nr.>).

5.1. Monaden in Isabelle

In Definition 2.1 wurden Monaden als Tripel (T, ret, $\gg=$) mit Typkonstruktor T und den beiden darauf definierten Funktionen $\gg=$ (gesprochen bind) und `ret` eingeführt. Bevor wir uns der Semantik zuwenden, die mit diesen Funktionen verknüpft ist, stellen wir zunächst die Syntax zur Verfügung.

5.1.1. Monaden-Syntax in Isabelle

Zunächst werden die Typdeklaration und die Einführung der Funktionen in ihrer Umsetzung beschrieben:

typedecl ’a T

consts

ret :: "’a ⇒ ’a T"

bind :: "’a T ⇒ (’a ⇒ ’b T) ⇒ ’b T" ("_ ≫= _" [5, 6] 5)

Auch die in Abschnitt 2.2 eingeführte Funktion $\gg$ wird implementiert. Sie ist lediglich syntaktischer Zucker und kommt zum Einsatz, wenn nur die sequenzielle Bearbeitung zweier monadischer Programme von Interesse ist, das Ergebnis des ersten aber nicht in gebundener Form an das zweite weitergereicht wird.

constdefs

bind’ :: "’a T ⇒ ’b T ⇒ ’b T" ("_ ≫ _" [5, 6] 5)

"bind’ p q == bind p (λx. q)"

Die Funktionen $\gg=$ bzw. $\gg$ sind zwar die ursprüngliche Darstellungsweise von Monaden, im Weiteren werden wir aber alle Regeln mit Hilfe der aus Haskell bekannten do-Notation formulieren. Wie in Abschnitt 2.2 kurz anklang, ist $do\{x \leftarrow p; q\ x\}$ dabei zu verstehen als $p \gg= \lambda x.q \ \ x$ und $do\{p; q\}$ als $p \gg q$.

Eine Syntax-Einführung in Kombination mit einer Syntax-Übersetzung ermöglicht, dass die in do-Notation formulierten Regeln auch auf Monaden in der ursprünglichen Schreibweise anwendbar sind.

nonterminals

monseq

syntax

”_monseq” :: ”monseq $\Rightarrow$ ’a T” (”(do $\{$(_)$\}$)” [5] 100)
”_mongen” :: ”[pttrn, ’a T, monseq]$\Rightarrow$ monseq” (”((_$\leftarrow$(_));/ _)” [110,6,5]5)
”_monexp” :: ”[’a T, monseq]$\Rightarrow$ monseq” (”((_);/ _)” [6,5] 5)
”_monexp0” :: ”[’a T] $\Rightarrow$ monseq” (”(_)” 5)

Mit dieser Syntax haben wir die Möglichkeit, alle Formen von monadischen Sequenzen zu parsen.

Beispiel 5.1. *Das Programm* $do\{x \leftarrow p; q\ x; ret\ x\}$ *wird geparst als*

_monseq (_mongen x p (monexp (q x) (monexp0 (ret x))))

Die Umformung in die do-Schreibweise folgt nach den Übersetzungsregeln:

translations

”_monseq(_mongen x p q)” $=>$ ”p $\gg$= (%x. (_monseq q))” (5.1)
”_monseq(_monexp p q)” $=>$ ”p $\gg$ (_monseq q)” (5.2)
”_monseq(_monexp0 q)” $=>$ ”q” (5.3)

Beispiel 5.2. *Das obige Beispiel wird dann schrittweise umgeformt:*

. . . $\Rightarrow$ *p* $\gg=$ *(*λ *x. (_monseq (_monexp (q x) (_monexp0 (ret 0)))))* *(Anw. 5.1)*
$\Rightarrow$ *p* $\gg=$ *(*λ *x. (q x) (_monexp0 (ret 0)))* *(Anw. 5.2)*
$\Rightarrow$ *p* $\gg=$ *(*λ *x. (q x)* $\gg$ *(ret 0))* *(Anw. 5.3)*

Da Isabelle beim Einlesen eines Terms alle möglichen Übersetzungen direkt vornimmt, wird nach diesen drei Regeln im `*isabelle/isar*`-Puffer alle do-Terme durch $\gg=$ und $\gg$ dargestellt werden. Da wir aber bei Formulierungen in do-Schreibweise diese auch im aktuelle Beweis-Zustand präsentiert bekommen wollen, müssen wir Isabelle noch Regeln für die Rückübersetzung zur Verfügung stellen.

translations

”_monseq(_mongen x p q)” $<$= ”p $\gg$= (%x. q)” (5.4)
”_monseq(_monexp p q)” $<$= ”p $\gg$ q” (5.5)

”_monseq(_monexp p q)” $<$= ”_monseq (_monexp p (_monseq q))” (5.6)
”_monseq(_mongen x p q)” $<$= ”_monseq (_mongen x p (_monseq q))” (5.7)

Beispiel 5.3. *Die Sequenz aus Beispiel 5.2 wird in der folgenden Weise zurück in die do-Notation übersetzt:*

$$\begin{aligned} \ldots \quad &\Rightarrow \textit{\_monseq (\_mongen x p (q x} \gg \textit{ret x))} && \textit{(Anw. 5.4)} \\ &\Rightarrow \textit{\_monseq (\_mongen x p (\_monseq (\_monexp (q x) (ret x))))} && \textit{(Anw. 5.4)} \end{aligned}$$

Damit die Äquivalenz der beiden Bindungsvarianten `x ← p; q` und `p;q` nicht bei jedem Auftreten explizit nachgewiesen werden muss, nehmen wir sie in den *simpset* auf. Dies geschieht über ein entsprechendes Lemma `delBind`. Die Übersetzung des do-Terms in die ursprüngliche Schreibweise erfolgt automatisch durch die oben beschriebenen Regeln. Somit lässt sich die Korrektheit des Lemmas durch Anwendung der Definition von $\gg$ und einer Vereinfachung der entstehenden Gleichung zeigen.

lemma delBind[simp]: "do {x←p; q} = do {p; q}"
apply (unfold bind'_def)
by simp

Im Allgemeinen, erleichtert die Aufnahme dieser Regel in den *simpset* die Arbeit mit Monaden deutlich. Da einige auf Monaden aufbauende Lemmata explizite Bindungen benötigen, muss an manchen Stellen die Verwendung der Regel durch `simp del: delBind` unterdrückt werden.

Die für die Anwendung auf Monaden gelifteten Kontrollstrukturen haben in Isabelle die folgende Form:

constdefs
if$_T$ :: "bool T ⇒ 'a T ⇒ 'a T ⇒ 'a T" ("if$_T$(_)then(_)else(_)")
"if$_T$ b then p else q == do{x←b;if x then p else q}"

consts
iter$_T$:: "('a ⇒ bool T) ⇒ ('a ⇒ 'a T) ⇒ 'a ⇒ 'a T"
while$_T$ :: "bool T ⇒ unit T ⇒ unit T" ("while$_T$ (_) (_)")

`iter` und `while` sind in axiomaticher Form implementiert, da `defs` keine rekursiven Definitionen ermöglichen.

Die in 2.2 eingeführte abkürzende Schreibweise $\bar{x} \leftarrow \bar{p}$ für wiederholt gekapselte Sequenzen lässt sich in Isabelle nicht ohne weiteres einführen. Zwar lässt sich für eine Sequenz $x_1 \leftarrow p_1; \ldots; x_n \leftarrow p_n\ x_1 \ldots x_{n-1}$ die gekapselte Version

$$(x_1, \ldots, x_n) \leftarrow do\{x_1 \leftarrow p_1; \ldots; x_n \leftarrow p_n\ x_1 \ldots x_{n-1}; ret\ (x_1 \ldots x_n)\}$$

verwenden, Lemmata können aber nicht für beliebige Sequenzen zur Verfügung gestellt werden. Die Ursachen dafür und mögliche Lösungsansätze werden in Kapitel 7.1 besprochen.

Bis eine entsprechende Variante implementiert ist, müssen die Regeln für jede benötigte Sequenzlänge explizit formuliert werden. Ein Beispiel dafür ist das in 2.2.1 eingeführte Lemma `ret2seq`. Für dieses sind Versionen für Sequenzen der Länge zwei und drei implementiert.

lemma ret2seq:
assumes "do {x←p;ret x} = do {x←q;ret x}"
shows "do {x←p;r x} = do {x←q;r x}"

lemma ret2seq_exp:
assumes "do {x←p1;y←p2 x;ret (x,y)} = do {x←q1;y←q2 x;ret (x,y)}"
shows "do {x←p1;y←p2 x;r x y} = do {x←q1;y←q2 x;r x y}"

Zu dem werden noch zwei weitere Versionen umgesetzt, die einer möglichen Vertauschung der Rückgabewerte Rechnung tragen.

lemma ret2seqSw:
assumes "do {x←p1;y←p2 x;ret (y,x)} = do {x←q1;y←q2 x;ret (y,x)}"
shows "do {x←p1;y←p2 x;r x y} = do {x←q1;y←q2 x;r x y}"

lemma ret2seqSw':
assumes "do {x←p1;y←p2;ret (x,y)} = do {y←q1;x←q2;ret (x,y)}"
shows "do {x←p1;y←p2;r x y} = do {y←q1;x←q2;r x y}"

5.1.2. Implementation der Monaden-Gesetze

Bisher haben wir lediglich die Syntax betrachtet. Um diese mit einer Semantik zu belegen und somit Regeln wie `ret2seq` beweisen zu können, formulieren wir die in Kapitel 2.2.1 eingeführten Monaden-Gesetze in Isabelle. Die Umsetzung ist in drei Schritte unterteilt. Zunächst wird eine axiomatische Version mit Hilfe der ursprünglichen Monaden-Schreibweise implementiert.

axioms
lunit: "(ret x ≫= p) = p x"
runit: "(p ≫= ret) = p"
assoc: "(p ≫= q ≫= r) = (p ≫= (λx. q x ≫= r))"

Darauf aufbauend wird eine Version in do-Schreibweise zur Integration in den *simpset* umgesetzt. Diese lassen sich durch die automatische Übersetzung in die ursprüngliche Monaden-Schreibweise aus den Axiomen herleiten.

lemma fstUnitLaw [simp]: "(do {y←ret x; p y}) = p x"

by (rule lunit)

lemma sndUnitLaw [simp]: "(do {x←p; ret x}) = p"
by (rule runit)

lemma assocLaw [simp]: "do {y←do {x←p; q x}; r y} = do {x←p; y←q x;r y}"
by (rule assoc)

Die Integration in den *simpset* vereinfacht an vielen Stellen die Beweisführung auf monadischen Strukturen. Wie schon bei `delBind` sollte man sich jedoch immer bewusst sein, dass die automatische Anwendung auch Probleme mit sich bringen kann.

Als dritter Schritt bei der Beschreibung der Semantik wird die Assoziativität an die verschiedenen Bindungsvarianten angepasst. Da wir die Monaden-Gesetzte und die Gleichheit von `x← p;q` und `p;q` bereits in den *simpset* übernommen haben, würde eine `by simp` für den Beweis der drei Lemmata ebenso ausreichen. Um die Beweis-Struktur besser verfolgen zu können, werden die Regelanwendungen hier jedoch einzeln aufgeführt.

lemma do_assoc1[simp]: "(do{do{x←p;q x}; r}) = (do {x←p; q x; r})"
apply (unfold "bind'_def")
by (rule assocLaw)

lemma do_assoc2[simp]: "(do {x←(do {p; q}); r x}) = (do {p; x←q; r x})"
apply (unfold "bind'_def")
by (rule assocLaw)

lemma do_assoc3[simp]: "(do {(do {p; q}); r})= (do {p; q; r})"
apply (unfold "bind'_def")
by (rule assocLaw)

Durch die Einführung dieser Gesetzmäßigkeiten, lassen sich nun einfache Beweise wie der für Lemma 2.1 vorgestellte durchführen:

lemma ret2seq:
assumes "do {x←p;ret x} = do {x←q;ret x}"
shows "do {x←p;r x} = do {x←q;r x}"
proof -
have "do {x←p;r x} = do {z←(do {x←p;ret x});r z}"
apply (subst sndUnitLaw) **..**
moreover from prems
have "... = do {z←(do {x←q;ret x});r z}"
by auto

```
  moreover
  have "... = do {x←q;r x}"
    apply (subst sndUnitLaw) ..
  ultimately show ?thesis by simp
qed
```

Aufbauend auf dieser Monaden-Implementierung lassen sich weitere Eigenschaften, die für die Umsetzung des monadischen Hoare-Kalküls benötigt werden, formulieren.

5.2. Umsetzung globaler dynamischer Aussagen

Wie in Kapitel 2.3 beschrieben, ist der nächste Schritt zur Entwicklung einer monadischen Hoare-Logik, die Separierung logischer Formeln von den bei ihrer Auswertung auftretenden Seiteneffekten. Dazu wird das Konzept der globalen Zusicherungen (engl. **g**lobal **d**ynamic **j**udgenemnts - gdj) in Isabelle umgesetzt. Im Gegensatz zur Implementierung der Monaden wird dabei zunächst die Semantik eingeführt.

5.2.1. Syntax und Semantik globaler Aussagen

Für die Umsetzung in Isabelle müssen zwei Formen der gdj unterschieden werden. Neben der vergestellten Form $[x \leftarrow p]\ \Phi\ x$ kann die Programmsequenz auch leer sein. Da sich die Anzahl der Parameter zwischen diesen beiden Varianten unterscheidet, müssen sie separat formuliert werden.

constdefs

gdj :: "'a T ⇒ ('a ⇒ bool) ⇒ bool" ("[_]_" [0, 100] 100) (5.8)

"gdj p Φ == ((do {x←p; ret (Φ x, x)}) = (do {x←p; ret (True, x)}))" (5.9)

empty_gdj :: "bool ⇒ bool" (5.10)

"empty_gdj Φ == (Φ = True)" (5.11)

Wie in Kapitel 4.2 beschrieben lässt sich direkt in der Konstanten-Definition eine Syntax für das eingeführte Konstrukt angeben. Um aber längere Sequenzen nicht in gekapselter Form schreiben zu müssen, ist die Syntax über eine Kombination von Syntax-Einführung und Syntax-Übersetzung implementiert, die die Kapselung automatisiert.

Die syntaktischen Elemente, die bei der Formulierung eines gdj in Isabelle verwendet werden können sind die Folgenden:

nonterminals

bndseq bndstep (5.12)

syntax

"_empty_gdj" :: "bool ⇒ bool" ("[ ]_ " 100) (5.13)

"_gdj" :: "[bndseq, bool] ⇒ bool" ("[_]_ " [0,100]100) (5.14)

"_gdjBnd ":: "[pttrn, 'a T] ⇒ bndstep" ("_ ←_") (5.15)

"_gdjSeq ":: "[bndstep, bndseq] ⇒ bndseq" ("_ ;/ _") (5.16)

"" :: "[bndstep] ⇒ bndseq" ("_ ") (5.17)

Beispiel 5.4. *Ein in Isabelle formuliertes gdj der Form*

$$[x \leftarrow p; y \leftarrow q\ x; z \leftarrow r\ x\ y]\ \Phi\ x\ y\ z$$

wird geparst als:

gdj (gdjSeq (gdjBnd x p) (gdjSeq (gdjBnd y (q x)) (gdjBnd z (r x y))))

Dabei erfolgt die Umformung vom `(gdjBnd z (r x y))` *in eine* `bndseq` *durch die Syntax-Regel (5.17).*

Die Konstantendefinition für das gdj (5.8) erwartet als Parameter eine Monade und eine Funktion, die in die Wahrheitswerte abbildet. In Beispiel 5.4 lässt sich jedoch erkennen, dass die Sequenz nicht immer direkt als Monade vorliegt. Sie muss deshalb durch einen do-Term gekapselt werden.

Beispiel 5.5. *Die gdj-Sequenz aus Beispiel 5.4 kann durch Umformung zu*

$$do\{x \leftarrow p; y \leftarrow q\ x; z \leftarrow r\ x\ y; ret\ (x,\ y,\ z)\}$$

in die Definition eingesetzt werden.

Wie an dem Beispiel ersichtlich wird, muss bei einer solchen Kapselung eine Möglichkeit geschaffen werden, alle gebundenen Variablen zu sammeln und durch ein `ret` zurückzugeben. Ein weiteres Problem ergibt sich durch eine mögliche Abhängigkeit der separierten logischen Formeln von den gebunden Variablen. In einem Term der Form

$$[do\{x \leftarrow p; \ldots\}]\ \Phi\ x$$

ist das in der Zusicherung verwendete `x` zunächst ungebunden. Mit Hilfe eines Lambda-Terms lässt sich eine gebundene Variable erzeugen. Dazu wird, wie für das `ret` eine Liste aller in der Sequenz gebunden vorkommender Variablen benötigt.

Um eine solche Liste generieren zu können, wird die Sequenz schrittweise durchlaufen und alle gebundenen Variablen in einem Tupel aufgesammelt. Am Ende der Sequenz wird ein `ret` mit dem entstandenen Tupel angehängt. Damit dieses auch für den Lambda-Term zur Verfügung steht, muss es wieder nach außen geführt werden. Um dies durch eine Syntax-Übersetzung realisieren zu können werden zwei weitere Syntax-Konstrukte implementiert.

"_gdjIn ":: "[pttrn, bndseq] $\Rightarrow$ bndseq" (5.18)

"_gdjOut ":: "[pttrn, bndseq] $\Rightarrow$ bndseq" (5.19)

Beispiel 5.6. *Im Folgenden wird anhand des obigen Beispiels gezeigt, wie die Tupel-Bildung abläuft. Wir werden allerdings zu Gunsten der Lesbarkeit darauf verzichten, auf der geparsten Variante zu arbeiten. Stattdessen wird eine der Monaden-Sequenz ähnliche Schreibweise verwendet. Hinter den einzelnen Umformungsschritten findet sich jedoch die Angabe zur jeweils verwendeten Regel. Diese werden nach dem Beispiel eingeführt.*

Das obige Beispiel wird dann in folgender Weise übersetzt:

[x← p;y← q x;z← r x y] Φ *x y z*

[x← p; ***gdjIn*** *x (y← q x;z← r x y)]* Φ *x y z* *(Anw. 5.22)*

[x← p;y← q x; ***gdjIn*** *(x,y) (z← r x y)]* Φ *x y z* *(Anw. 5.23)*

[x← p;y← q x; ***gdjOut*** *((x,y),z) (*$do\{z \leftarrow r\ x\ y; ret((x,y),z)\}$*)]* Φ *x y z* *(Anw. 5.24)*

*[x← p;****gdjOut*** *((x,y),z) (*$do\{y \leftarrow q\ x; do\{z \leftarrow r\ x\ y; ret((x,y),z)\}\}$*)]* Φ *x y z* *(Anw. 5.25)*

*[****gdjOut*** *((x,y),z) (*$do\{x \leftarrow p; do\{y \leftarrow q\ x; do\{z \leftarrow r\ x\ y; ret((x,y),z)\}\}\}$*)]* Φ *x y z* *(Anw. 5.25)*

$[do\{x \leftarrow p; do\{y \leftarrow q\,x; do\{z \leftarrow r\,x\,y; ret((x,y),z)\}\}\}]\ \lambda((x,y),z).\ \Phi\,x\,y\,z$ *(Anw. 5.26)*

$[do\{x \leftarrow p; y \leftarrow q\ x; z \leftarrow r\ x\ y; ret((x,y),z)\}]\ \lambda((x,y),z).\ \Phi\ x\ y\ z$

Der letzte Schritt wird durch Anwendung der Monaden-Gesetze möglich. Da wir bei dieser Übersetzung eine unerwünschte linksseitige Klammerung des Tupels erhalten, wird vor der Übergabe an `ret` und `gdjOut` die Klammerung korrigiert. Dazu wird ein weiteres Syntax-Konstrukt eingeführt:

"_reTpl ":: "[pttrn, pttrn] $\Rightarrow$ pttrn" (5.20)

Die Isabelle-Implementierung der Übersetzungsregeln hat dann die folgende Form:

translations

"_gdj (_gdjBnd x p) phi" == "gdj p (λx. phi)" (5.21)

"_gdj (_gdjSeq (_gdjBnd x p) r) phi"
=> "gdj (_gdjSeq (_gdjBnd x p) (_gdjIn x x r)) phi" (5.22)

"_gdjIn tpl tpl' (_gdjSeq (_gdjBnd x p) r)"
=> "_gdjSeq (_gdjBnd x p) (_gdjIn (tpl, x) (tpl', x) r)" (5.23)

"_gdjIn tpl tpl' (_gdjBnd x p)"
=> "_gdjOut (_reTpl tpl' x) (do {x$\leftarrow$p; ret(_reTpl tpl x)}) " (5.24)

"_gdjSeq (_gdjBnd x p) (_gdjOut tpl r)" == "_gdjOut tpl (do {x$\leftarrow$p; r})" (5.25)

"gdj (_gdjOut tpl r) phi" == "gdj r (λtpl. phi)" (5.26)

"_reTpl (_tuple tpl (_tuple _arg x)) y" => "_reTpl tpl (_tuple x (_tuple _arg y))" (5.27)

"_reTpl x y" => "(x,y)" (5.28)

Da `gdjBnd` ein Pattern als ersten Parameter erwartet, lassen sich mit dieser Syntax bereits Sequenzen durch einen do-Term in folgender Weise kapseln:

$$[(x,y) \leftarrow do\{x \leftarrow p; y \leftarrow q\ x; ret\ (x,y)\}; z \leftarrow r\ x\ y\ z]\ \Phi\ x\ y\ z$$

Damit in solchen Fällen die Klammerung einheitlich ist, wird noch folgende Übersetzungsregeln hinzugefügt:

"Pair x (_pattern y z)" => "Pair x (Pair y z)" (5.29)

"Pair (_pattern x y) z" => "Pair x (Pair y z)" (5.30)

"Pair x (_patterns y z)" => "Pair x (Pair y z)" (5.31)

"Pair (_patterns x y) z" => "Pair x (Pair y z)" (5.32)

"(_ pattern x (_patterns y z))" => "Pair x (Pair y z)" (5.33)

In Kapitel 7.1 werden einige Ideen vorgestellt, um Lemmata durch gekapselte Sequenzen auf beliebige Sequenzlängen anzuwenden. Erste Versuche, diese Ideen umzusetzen spiegeln sich in der getrennten Generierung der Tupel für die Rückgabe über `ret` und den Lambda-Term wider. Zu dem wird in den Isabelle-Theorien innerhalb der gdj-Sequenzen eine weiter Bindungsart (gdjPBnd - "_ $\prec$- _ $\leftarrow$ _") implementiert. Näherer Erläuterungen zu dieser Syntax und ihrer Verwendung finden sich in Kapitel 7.1.

5.2.2. Regelsatz für globale Aussagen

Nachdem nun die Syntax und Semantik der gdj zur Verfügung stehen, werden im Folgenden einige auf diesem Konstrukt aufbauende Beweise in ihrer Isabelle-Umsetzung beschrieben.

Auf die in Kapitel 2.3 vorgestellten Regeln für globale Aussagen soll an dieser Stelle jedoch nicht noch einmal im Detail eingegangen werden. Sie orientieren sich bereits sehr stark an der Isabelle-Umsetzung. An dieser Stelle werden einige aufgetretene Besonderheiten näher beleuchtet.

Wie schon bei den monadischen Sequenzen ergeben sich bei der Umsetzung der gdj in Isabelle Probleme durch die fehlende Flexibilität des Regelsatzes bezüglich der Länge der Sequenzen. Die in Kapitel 2.3 beschriebenen und im Weiteren als Stammlemmata bezeichneten Formen, sind so gewählt, dass sie alle gebotenen Möglichkeiten in einfachster Form widerspiegeln.

Dadurch ergeben sich für den Regelsatz für gdj neben den Lemmata für erweiterte Sequenzen auch solche, die Beweise über verkürzte Sequenzen ermöglichen. Erstere werden durch das Anhängen von „_exp“ an den Namen des Stammlemmatas gekennzeichnet, zweitere durch „_cut“.

Eine Ausnahme bei dieser Namensgebung bilden dabei `postOp` und `preOp`, die eine Erweiterung von `ctr` darstellen. Die Namensgebung außerhalb des Schemas soll ihre Anwendung im Zusammenhang mit Vor- und Nachbedingung der Hoare-Tripel in gdj-Schreibweise hervorheben.

Neben der expliziten Bereitstellung von angepassten Lemmata für häufig verwendete Sequenzlängen, wird noch eine zweite, auf Kapselung basierende Methode für die Bearbeitung unterschiedlicher Sequenzlängen angewandt.

Zwar lassen sich die in Isabelle formulierten Lemmata nicht auf gekapselte Sequenzen wie sie in Abschnitt 5.2.1 vorgestellt wurden anwenden, es gibt aber eine andere Form, die dafür geeignet ist. Dabei erfolgt die Bindung der gekapselten Sequenz nicht an ein Tupel sondern an eine einzelne Variable. Die Aufspaltung in die einzelnen Tupel-Komponenten erfolgt erst bei ihrer Verwendung.

Beispiel 5.7. *Das in Beispiel 5.5 verwendete gdj kann für die Anwendung von Lemmata, die auf Sequenzen der Länge zwei definiert sind, umgeformt werden zu:*

$$[v \leftarrow do\{x \leftarrow p; y \leftarrow q\,x; ret\,(x,\ y)\}; z \leftarrow r\,(fst\,v)\,(snd\,v)\,z]\ \Phi\ (fst\,v)\,(snd\,v)\,z$$

Je länger die auf diese Weise gekapselten Sequenzen sind, desto stärker nimmt die Lesbarkeit des Beweises ab. Dies wird noch dadurch verstärkt, dass bei der

Umformung zwischen gekapselter und ungekapselter Form die Definition der gdj aufgefaltet werden muss. Nur auf Basis der dadurch entstehenden Gleichung lässt sich zeigen, dass die an die globale Aussage übergebenen Parameter dieser Umformung standhalten.

Beispiel 5.8. *Die Umformung verläuft jeweils in zwei Schritten wie das folgende Isabelle-Theorem demonstriert.*

theorem
  assumes *"[x←p;y←q x;z←r x y] Φ x y z"*
  shows *"[v←do{x←p;y←q x;ret(x,y)};z←r (fst v) (snd v)] Φ (fst v) (snd v) z"*
proof -

Zunächst muss eine Kapselung des gesamten Terms vorgenommen werden. Diese ist unproblematisch, da sie der Definition der gdj entspricht.

  from *prems* **have**
  "[u←do{x←p;y←q x;z←r x y;ret(x,y,z)}] Φ (fst u) (fst(snd u)) (snd(snd u))"
  by *(simp add: gdj_def)*

Die gekapselte Version kann nun durch `gdj2doSeq` mit einer geeigneten Klammerung für die Rückgabewerte des späteren Tupels versehen werden

  from *this* **have**
  "do{u←do{x←p;y←q x;z←r x y;ret(x,y,z)};
    ret(Φ (fst u) (fst(snd u)) (snd(snd u)),
      (fst u, fst(snd u)), snd(snd u))} =
  do{u←do{x←p;y←q x;z←r x y;ret(x,y,z)};
    ret(True,
      (fst u, fst(snd u)),snd(snd u))}"
  by *(rule gdj2doSeq)*

Mit Hilfe der im *simpset* enthaltenen Monaden-Gesetze lässt sich daraus die entsprechend der Behauptung gekapselten Gleichung herleiten.

  from *this* **have**
  "do{v←do{x←p;y←q x;ret(x,y)};z←r (fst v) (snd v);
    ret(Φ (fst v) (snd v) z,
      (fst v, snd v), z)} =
  do{v←do{x←p;y←q x;ret(x,y)};z←r (fst v) (snd v);
    ret(True,
      (fst v, snd v), z)}"
  by *simp*

Die Anwendung der gdj-Definition ermöglicht nun die Herleitung der Behauptung.

  from *this* **show** *?thesis*
  by *(simp add: gdj_def)*
qed

Eine Anwendung dieser Methode findet sich in den Herleitungen der erweiterten Lemmata aus den Stammlemmata.

Für die Herleitung der im Vergleich zum Stammlemma verkürzten Form wird die Seiteneffektfreiheit von `ret ()` ausgenutzt. Dadurch lassen sich Sequenzen durch Anhängen dieses Programms verlängern, bis sie die für die Anwendung des entsprechenden Stammlemmas benötigte Länge haben.

Da diese Erweiterung mehrfach benötigt wird, ist sie in Hilfslemmata ausgelagert. Dabei verlängern `*-`, `**-` und `***-` eine Sequenz der Länge eins, zwei bzw. drei durch Anhängen von `ret ()`. Die Lemmata `-*`, `-**` sowie `-***` stellen der Sequenz ein `ret ()` voran.

Anhand des für die verkürzte Sequenz formulierten Lemmatas `ctr_cut` lässt sich die Anwendung dieser Hilfslemmata zeigen:

lemma ctr_cut:
  assumes a: "[x←p;y←q x;z←r y] (Φ y z)"
  shows "[y←(do {x←p;q x});z←r y] (Φ y z)"
proof -
  from a **have** "[v←ret();x←p;y←q x;z←r y] (Φ y z)"
    by (simp add: "***-")
  from this **have** "[v←ret();y←do{x←p;q x};z←r y] (Φ y z)"
    by (rule ctr)
  from this **show** ?thesis
    apply (subst "**-")
    by simp
qed

5.2.3. Beweise auf Basis des res-Operators

Ein in den bisherigen Kapiteln unberücksichtigter Teil der Arbeit ist der in [SM04] eingeführte Operator `res`. Dieser ist insbesondere für den Beweis von Lemma `rp`, das in 2.3 nur axiomatisch eingeführt werden konnte von Bedeutung.

Der Operator `res` ist dabei wie folgt definiert:

constdefs
  "res" :: "'a $\Rightarrow$ bool $\Rightarrow$ 'a" ("_ res _" 100)
  "α res Φ == if Φ then α else arbitrary"

Der Term α `res` Φ ist definiert und dann gleich α, falls α definiert ist und Φ wahr ist. Die entscheidende Eigenschaft, die sich aus dieser Definition ergibt, ist die Ersetzbarkeit von α nach folgender Regel:

lemma eq_res:
  "∀x. ((r x) res ((r x)=(s x))) = ((s x) res ((r x)=(s x)))"
proof (cases "∀x.(r x)=(s x)")
  case True
  from this **show** ?thesis
   by (simp add: res_def)
 next
  case False
  from this **show** ?thesis
   by (simp add: res_def)
qed

Der Nachweis, dass

lemma rp:
  assumes a:"∀x.(q_1 x = q_2 x)" **and**
    b: "[x←p; y←q_1 x; z←r x y] Φ x y z"
  shows "[x←p; y←q_2 x; z←r x y] Φ x y z"

korrekt ist, erfolgt dann in drei Schritten. Zunächst wird gezeigt, dass das gdj [x←p] ((q1 x)=(q2 x)) die Gleichheit von do {x←p;r x (q1 x)} und do {x←p;r x (q2 x)} impliziert. Dieses als `gdjEq2doSeq` implementierte Lemma lässt sich dann dahingehend erweitern, dass eine Ersetzung innerhalb eines gdj in folgender Weise möglich wird:

lemma eq:
  assumes a: "[x←p](q_1 x = q_2 x)" **and**
    b:"[x←p; y←q_1 x; z←r x y] Φ x y z"
  shows "[x←p; y←q_2 x; z←r x y] Φ x y z"

Lemma `rp` lässt sich dann daraus herleiten durch:

lemma tau:
  assumes "∀x.(Φ x)"
  shows "[x←p] Φ x"
proof -
  from prems **show** ?thesis
   by (simp add: gdj_def)
qed

5.2.4. Herleitung des rp-Lemmas

Im Folgenden werden die beiden Lemmata `gdjEq2doSeq` und `eq` in ihrer Isabelle-Umsetzung vorgestellt. Sie bilden die Grundlage für den Beweis der als Axiom 2.1 eingeführten Regel `rp`.

lemma gdjEq2doSeq:
  assumes "[x←p] ((q1 x)=(q2 x))"
  shows "do {x←p;r x (q1 x)} = do {x←p;r x (q2 x)}"
proof -
  have "do {x←p;r x (q1 x)} = do {x←p;r x ((q1 x) res ⊤)}"
   by (simp add: res_def)
 moreover
  from prems **have** "... = do {x←p;r x ((q1 x) res ((q1 x)=(q2 x)))}"

Die Voraussetzung lässt sich mit Hilfe von `gdj2doSeq` zu der Gleichung erweitern.

 moreover
  have "... = do {x←p;r x ((q2 x) res ((q1 x)=(q2 x)))}"

Nach Lemma `eq_res` sind die beiden Formeln `(q1 x) res (q1 x = q2 x)` und `(q2 x) res (q1 x = q2 x)` äquivalent und können gegeneinander ausgetauscht werden.

 moreover
  from prems **have** "... = do {x←p;r x (q2 x)}"

Die Definitionen von `res` und `if` führen durch Anwendung von `gdj2doSeq` und der Voraussetzung zu dieser Gleichheit.

 ultimately show ?thesis
   by simp
qed

Dieses Lemma ist die Voraussetzung dafür, dass sich die Gleichheit von `q1` und `q2` auch innerhalb der gdj ausnutzen lässt:

lemma eq:
  assumes a: "[x←p](q_1 x = q_2 x)" **and**
      b: "[x←p; y←q_1 x; z←r x y] Φ x y z"
  shows "[x←p; y←q_2 x; z←r x y] Φ x y z"
proof -
  from a **have**
   "do {x←p;y←q_2 x;z←r x y;ret (Φ x y z,x,y,z)} =
   do {x←p;y←q_1 x;z←r x y;ret (Φ x y z,x,y,z)}"

Nachdem eine geeignete Klammerung der Sequenz geschaffen ist, wird mit `gdjEq2doSeq` aus der ersten Voraussetzung die Gleichung gezeigt.

```
  moreover
   from b  have "... = do {x←p;y←q₁ x;z←r x y;ret (⊤,x,y,z)}"
    by (simp add: gdj_def)
  moreover
   have "... = (do {x←p;(y,z)←(do{y←q₁ x;z←r x y;ret(y,z)});
                                    ret (⊤,x,y,z)})"
    by simp
  moreover
   from a have "... = (do {x←p;(y,z)←(do{y←q₂ x;z←r x y;ret(y,z)});
                                    ret (⊤,x,y,z)})"
    by (rule gdjEq2doSeq)
  moreover
   have "... = do {x←p;y←q₂ x;z←r x y;ret (⊤,x,y,z)}"
    by simp
  ultimately have
    "do {x←p;y←q₂ x;z←r x y;ret (Φ x y z,x,y,z)} =
     do {x←p;y←q₂ x;z←r x y;ret (⊤,x,y,z)}"
    by simp
   from this show ?thesis
    by (simp add: gdj_def)
qed
```

Nachdem nun eine genaue Definition der Gleichheit zweier Programme implementiert ist und die Eigenschaften, die sich daraus ergeben bewiesen wurden, kann schlussendlich das Axiom 2.1 als Lemma gezeigt werden.

```
lemma rp:
   assumes a:"∀x.(q₁ x = q₂ x)" and
         b: "[x←p; y←q₁ x; z←r x y] Φ x y z"
   shows "[x←p; y←q₂ x; z←r x y] Φ x y z"
proof -
   from a have "[x←p] (q₁ x = q₂ x)"
    by (rule tau)
   from this b show "[x←p; y←q₂ x; z←r x y] Φ x y z"
    by (rule eq)
qed
```

5.3. Seiteneffektfreiheit

Zur Beschreibung der monadischen Hoare-Logik entsprechend den Ausführungen in Kapitel 3 wird nich der Begriff der deterministischen Seiteneffektfreiheit benötigt. Wie in Abschnitt 2.4 beschrieben erfolgt eine Aufteilung der Definition in die drei

Monaden-Eigenschaften der Seiteneffektfreiheit, Kopierbarkeit und Vertauschbarkeit. Im Folgenden Abschnitt werden daher zunächst diese drei Begriffe in ihrer Umsetzung beschrieben.

5.3.1. Seiteneffektfreiheit, Kopierbarkeit und Vertauschbarkeit

Die in Abschnitt 2.4 vorgestellten Definition für Seiteneffektfreiheit und Kopierbarkeit lassen sich ohne Einschränkungen als Konstanten-Definition nach Isabelle übertragen:

constdefs
"sef" :: "'a T ⇒ bool" ("sef _")
"sef p == (do{y←p; ret ()} = ret ())"

"cp" :: "'a T ⇒ bool"
"cp p == (do{x←p;y ←p;ret(x,y)} = do{x←p; ret (x,x) }) "

Die Umsetzung der Vertauschbarkeit bedarf zunächst der Einschränkung auf die Vertauschbarkeit mit Programmen vom Typ `bool T`.

"com" :: "'a T ⇒ bool T ⇒ bool" ("_ comwith _")
"com p q == (cp q ∧ sef q ⟶ cp(do {x←p; y←q; ret(x,y)}))"

Die Ursache liegt in der Implementierung der deterministischen Seiteneffektfreiheit, die auf dieser Definition aufbaut. Bevor wir diese aber genauer beleuchten, stellen wir zunächst ein Axiom zur Verfügung, dass die Verallgemeinerung zum Typ `'b T` erlaubt.

axioms commute_tcoerc:
"∀q::bool T. (cp q) ∧ (sef q) ⟶ (cp (do{x←p; y←q; ret(x,y) })) ⟹
∀q. (cp q) ∧ (sef q) ⟶ (cp (do{x←p; y←q; ret(x,y) })) "

Der in [SM03] beschriebene Beweis dieser Verallgemeinerung lässt sich bisher noch nicht in Isabelle formulieren. Die Ursachen dafür und mögliche Lösungsansätze werden in Kapitel 7.1 vorgestellt.

5.3.2. Deterministische Seiteneffektfreiheit

Deterministische Seiteneffektfreiheit lässt sich nun auf Basis der Definition der drei Eigenschaften wie folgt umsetzen:

constdefs

"dsef" : : "'a T $\Rightarrow$ bool"
"dsef p == (sef p) $\wedge$ (cp p) $\wedge$ ($\forall$ q. (p comwith q))"

An dieser Stelle wird nun auch ersichtlich, warum die Definition der Vertauschbarkeit zunächst nur in eingeschränkter Form erfolgen konnte: Isabelle ermöglicht keine Quantifizierung über Typen wie man sie für die allgemeine Form bräuchte.

In Kapitel 3 haben wir motiviert, dass die Vor- und Nachbedingung eines Hoare-Tripels deterministisch seiteneffektfrei auswerten müssen. Um diese Eigenschaft direkt in de Definition der Tripel verankern zu können, fassen wir alle Monaden, die diese Eigenschaft erfüllen im Untertyp `'a D` zusammen.

typedef (Dsef) 'a D = "{p: :'a T. dsef p}"

Wie in Kapitel 4 beschrieben, stehen durch diese Typdefinition nun die zwei Funktionen

```
Rep_Dsef:: 'a D => 'a T
Abs_Dsef:: 'a T => 'a D
```

und

zur Verfügung, die Werte so weit das möglich ist zwischen den beiden Typen `'a T` und `'a D` umwandeln. Ausserdem beinhaltet die Einführung des Untertyps die beiden Axiome, die `Abs_Dsef` als inverse Funktion zur `Rep_Dsef` beschreiben.

```
Rep_Dsef_inverse: "Abs_Dsef (Rep_Dsef p) = p"
Abs_Dsef_inverse: "p ∈ Dsef ⇒ Rep_Dsef (Abs_Dsef p) = p"
```

Damit in den Beweisen die häufigen Vorkommen von Abs_Dsef und Rep_Dsef sich nicht störend auf die Lesbarkeit auswirken, werden Abkürzungen dafür bereit gestellt. Das Lifting von 'a T nach 'a D durch wird dabei durch $\uparrow$ gekennzeichnet, die Reduktion durch $\downarrow$. Ausserdem wird eine geliftete Version vorn `ret` bereitgestellt. Damit die Umwandlung automatisch erfolgt, ist das Lifting über eine Syntax-Übersetzung implementiert. Somit lassen sich die Vorteile des deterministisch seiteneffektfreien Untertyps jederzeit nutzen, ansonsten muss aber keine Unterscheidung bei der Behandlung von `'a T` und `'a D` getroffen werden.

syntax
Rep :: "'a D $\Rightarrow$ 'a T" ("$\downarrow$_")
Abs :: "'a T $\Rightarrow$ 'a D" ("$\uparrow$_")
Ret :: "'a $\Rightarrow$ 'a D" ("$\Uparrow$_")

translations
"$\downarrow$p" == "Rep_Dsef p"
"$\uparrow$p" == "Abs_Dsef p"

"$\Uparrow$p" == "$\uparrow$(ret p)"

Die logischen Junktoren $\wedge$ und $\vee$, sowie die Negation $\neg$ sind bereits für das Arbeiten mit booleschen Werten reserviert. Für die Anwendung im Hoare-Kalkül benötigen wir jedoch entsprechend geliftete Versionen. Da wir die Operatoren nur in Vor- und Nachbedingung der Hoare-Tripel benötigen, kann das Lifting direkt nach `bool D` erfolgen.

Man erspart sich dadurch eine Reduktionsangabe in der Vor- und Nachbedingung der Tripel. Diese taucht dann erst bei der Anwendung der Definition für die geliften Operatoren auf. Ein weiter Vorteil liegt darin, dass schon anhand der Schreibweise klar wird, dass Vor- und Nachbedingung vom Typ `'a D` sind.

constdefs
condConj :: "bool D $\Rightarrow$ bool D $\Rightarrow$ bool D" ("_$\wedge_D$_")
"condConj Φ ξ $\equiv$ $\uparrow$do{x$\leftarrow\downarrow\Phi$;y$\leftarrow\downarrow\xi$;ret(x$\wedge$y)}"
condDisj :: "bool D $\Rightarrow$ bool D $\Rightarrow$ bool D" ("_$\vee_D$_")
"condDisj Φ ξ $\equiv$ $\uparrow$do{x$\leftarrow\downarrow\Phi$;y$\leftarrow\downarrow\xi$;ret(x$\vee$y)}"
condNot :: "bool D $\Rightarrow$ bool D" ("$\neg_D$_")
"condNot Φ $\equiv$ $\uparrow$do{x$\leftarrow\downarrow$ Φ;ret($\neg$ x)}"

Auch für if und while ist ein Lifting nach `bool D` sinnvoll. Die Umsetzung kann dabei auf zwei Weisen geschehen. Zum Einen kann auf die für `'a T` definierten Kontrollstrukturen aufgebaut werden. Dies hätte allerdings zur Folge, dass zur Auffaltung der gelifteten Version beide Definitionen angewendet werden müssten. Aus diesem Grund ist die Definition direkt auf die boolesche Variante aufgesetzt.

constdefs
if$_D$ :: "bool D $\Rightarrow$ 'a T $\Rightarrow$ 'a T $\Rightarrow$ 'a T" ("if$_D$(_)then(_)else(_)")
"if$_D$ b then p else q == do{x$\leftarrow\downarrow$b;if x then p else q}"

consts
iter$_D$:: "('a $\Rightarrow$ bool D) $\Rightarrow$ ('a $\Rightarrow$ 'a T) $\Rightarrow$ 'a $\Rightarrow$ 'a T"
while$_D$ :: "bool D $\Rightarrow$ unit T $\Rightarrow$ unit T" ("while$_D$ (_) (_)")

axioms
iter$_D$_def: "iter$_D$ test f x =

do{if$_D$ (test x) then (do{y$\leftarrow$(f x);iter$_D$ test f y}) else (ret x)}"
while$_D$_def: "while$_D$ b p == iter$_D$ (λx. b) (λx. p) ()"

Die Umsetzung der deterministischen Seiteneffktfreiheit und der gdj ermöglicht uns nun die Implementierung der in Kapitel 3 eingeführten Hoare-Logik.

5.4. Monadische Hoare-Logik in Isabelle

Für die Umsetzung der in Kapitel 3 beschriebenen Hoare-Tripel werden, wie schon bei den gdj in Abschnitt 5.2.1, zwei Varianten implementiert. Zum einen die allgemeine Form, wie wir sie bisher kennengelernt haben:

$$\{\!|\Phi|\!\}\ \bar{x} \leftarrow p\ \{\!|\Psi x|\!\}.$$

Desweiteren als „Hoare-Tupel" mit leerer Programmsequenz. Dies entspricht einem gdj der Form $[a \leftarrow \Phi; b \leftarrow \Psi]\ a \rightarrow b$.

Um deutlicher zu machen, was ein solches „Tupel" aussagen soll, wird an manchen Stellen auch die alternative Schreibweise $\Phi \Rightarrow_h \Psi$ verwendet.

constdefs
  ”hoare_Tupel” :: ”bool D ⇒ bool D ⇒ bool” (”{|_|} {|_|}”)
  ”hoare_Tupel Φ Ψ == [(a,b)←do{a←↓Φ;b←↓Ψ;ret(a,b)}] (a⟶b)”

syntax
  ”hoare$_2$” :: ”bool D ⇒ bool D ⇒ bool” (”(_) $\Rightarrow_h$ (_)”)

translations
  ”hoare$_2$ Φ Ψ” == ”hoare_Tupel Φ Ψ”

Das „normale" Hoare-Tripel wird, wie in Kapitel 3 beschrieben in ein gdj der Form $[a \leftarrow \Phi; x \leftarrow p; b \leftarrow \Psi x]\ a \rightarrow b$ übersetzt. Da uns diese Herkunft der Hoare-Tripel aber beim Arbeiten mit ihnen nicht interessiert, wird sie durch eine entsprechende Konstanten-Definition versteckt. Durch explizite Anwendung der Definition lässt sich zwar über die Eigenschaften der gdj verfügen, ansonsten sollen aber keine gdj mehr in den Beweisen erscheinen. In der konkreten Umsetzung der Hoare-Tripel finden genauer genommen zwei Übersetzungen statt. Zum einen die eben erwähnte.

constdefs
”hoare”:: ”bool D ⇒ ’a T ⇒ (’a ⇒ bool D) ⇒ bool”
”hoare Φ p Ψ == [a←↓Φ ;x←p;b ←↓(Ψ x)] (a ⟶ b)”

Dadurch, dass die globalen Aussagen vom deterministisch seiteneffektfreien Untertyp der Monaden sind, müssen sie bei der Übersetzung in zunächst in `'a T` umgewandelt werden. Der Schritt der der Umformung in gdj vorangeht ist durch eine Syntax-Übersetzung realisiert. Dabei werden die gelifteten Bool-Operatoren $\wedge_D$, $\vee_D$ und $\neg_D$ bearbeitet. Vor allem werden aber, wie schon bei der Umsetzung der gdj, längere Sequenzen zu einer Programm-Sequenz $x \leftarrow p$ gekapselt und alle gebundenen Variablen aus p aufgesammelt. Diese können dann als Lambda-Term der Nachbedingung in gebundener Form zur Verfügung gestellt werden.

Beispiel 5.9. *Am Beispiel des Tripels* $\{\!|\, \Phi \,|\!\}\ x \leftarrow p; y \leftarrow q\ x\ \{\!|\, \Psi\ x\ y \,|\!\}$ *lässt sich erkennen, welchen Hintergrund dies hat. Wäre nur die reine Definition vorhanden, ließe sich das Hoare-Tripel so nicht schreiben, da die Sequenz* $x \leftarrow p; y \leftarrow q\ x$ *nicht von Type ′a T ist. Man müsste sie also umformen zu*

$$\{\!|\Phi|\!\}\ do\{x \leftarrow p; y \leftarrow q\ x; ret(x, y)\}\ \{\!|\Psi\ x\ y|\!\}\ .$$

Ψ *kann in diesem Fall aber gar nicht mehr auf x und y zugreifen, da sie durch das* do *gekapselt und nach außen nicht mehr sichtbar sind.*

Wie schon beim Parsen und Übersetzen der gdj, müssen an dieser Stelle die gebundenen Variablen in einem Tupel aufgesammelt, sortiert und zurückgegeben werden. `hIn` und `hOut` sind wie `gdjIn` und `gdjOut` für den Aufbau der entsprechenden Tupel zuständig. Allerdings werden diese bei der Übersetzung der Hoare-Tripel jeweils in zwei Formen benötigt. Eine, die wie bei der gdj-Variante ein Tupel generiert und eine zweite, die zum Einsatz kommt, falls das Ergebnis eines Programms ungebunden bleibt und somit keine Variable aufgesammelt werden muss.
.

Die vollständige für die Übersetzung benötigte Syntax sieht daher wie folgt aus:

nonterminals

”hseq” ”hstep” (5.34)

syntax

”_hoare” :: ”bool D ⇒ hseq ⇒ bool D ⇒ bool” (”{| _|}_ {|_|} ”) (5.35)

”_hbind” :: ”[pttrn, ’a T] ⇒ hstep” (”_ ←_”) (5.36)
”_hseq” :: ”[hstep, hseq] ⇒ hseq” (”_ ;_”) (5.37)
”_hsingle ” :: ”idt ⇒ hstep” (”_ ”) (5.38)
”_hstep” :: ”hstep ⇒ hseq” (”_ ”) (5.39)

”_hIn” :: ”hseq ⇒ hseq” (5.40)
”_hIn’ ” :: ”[pttrn, hseq] ⇒ hseq”” _hOut” :: ”[pttrn, hseq] ⇒ hseq” (5.41)
”_hOut’ ” :: ”[pttrn, hseq] ⇒ hseq” (5.42)

”_tpl” :: ”[pttrn, pttrn] ⇒ pttrn” (5.43)

Beispiel 5.10. *Das Hoare-Tripel* $\{\!|\Phi\ \wedge_D b|\!\}\ p; x \leftarrow q; rx\ \{\!|\Psi\ x|\!\}$ *wird durch (5.35) bis (5.39) geparst als:*

hoare ($\Phi \wedge_D$ *b) (hseq (hsingle p) (hseq (hbind x q) (hstep (hsingle (r x))))) (*Ψ*x)*

Da sich durch die Aufsplittung der Tupelbildung die Komplexität der Übersetzungsregeln insbesondere für `hIn` erhöht, fasst die folgende Tabelle das Vorgehen in Abhängigkeit der aktuellen Form und der noch zu übersetzenden Sequenz zusammen. Dabei stehen die Spaltenbeschriftungen jeweils für die noch zu bearbeitende Sequenz und den aktuellen Zustand des Tupels.

	p		x← p		p;q		x← p;q	
	-	tpl	-	tpl	-	tpl	-	tpl
neue Fkt.	hOut	hOut’	hOut’	hOut’	hIn	hIn’	hIn’	hIn’
Tupel	-	_tpl tpl	x	_tpl (x,tpl)	-	tpl	x	(x,tpl)
Sequenz	p	p	p	p	q	q	q	q
Regel	(5.48)	(5.49)	(5.50)	(5.51)	(5.52)	(5.53)	(5.54)	(5.55)

Beispiel 5.11. *Für* `hIn (x← p)` *ist die Umformung somit nach der dritten Spalte durchzuführen und man erhält* `hOut' x p`*.*

Die letzte Zeile der Tabelle liefert jeweils die Referenz zur Übersetzungsregel aus dem folgenden vollständigen Regelsatz:

translations

”_hoare Φ (_hstep (_hsingle p)) Ψ” => ”hoare Φ p (_K Ψ)” (5.44)
”_hoare Φ (_hstep (_hsingle p)) Ψ” <= ”hoare Φ p (λx. Ψ)” (5.45)
”_hoare Φ (_hstep (_hbind x p)) Ψ” == ”hoare Φ p (λx. Ψ)” (5.46)
”_hoare Φ (_hseq p q) Ψ” == ”_hoare Φ (_hIn (_hseq p q)) Ψ” (5.47)

”_hIn (_hstep (_hsingle p))” => ”_hOut p” (5.48)
”_hIn (_hstep (_hbind x p))” => ”_hOut’ x p” (5.49)
”_hIn (_hseq (_hsingle p) q)” => ”_hseq (_hsingle p) (_hIn q)” (5.50)
”_hIn (_hseq (_hbind x p) q)” => ”_hseq (_hbind x p) (_hIn’ x q)” (5.51)

”_hIn’ tpl (_hstep (_hsingle p))” => ”_hOut’ (_tpl tpl) (do{p;ret (_tpl tpl)}) ” (5.52)
”_hIn’ tpl (_hstep (_hbind x p))” => ”_hOut’ (_tpl (tpl,x)) (do{x $\leftarrow$ p;ret (_tpl (tpl,x))}) ” (5.53)
”_hIn’ tpl (_hseq (_hsingle p) q)” => ”_hseq (_hsingle p) (_hIn’ tpl q)” (5.54)
”_hIn’ tpl (_hseq (_hbind x p) q)” => ”_hseq (_hbind x p) (_hIn’ (tpl,x) q)” (5.55)

”_hseq (_hsingle p) (_hOut q)” => ”_hOut (p $\gg$ q)” (5.56)
”_hseq (_hsingle p) (_hOut’ tpl q)” => ”_hOut’ tpl (p $\gg$ q)” (5.57)
”_hseq (_hbind x p) (_hOut’ tpl q)” => ”_hOut’ tpl (p $\gg$= (λx. q))” (5.58)

”_tpl (Pair (Pair x y) z)” => ”_tpl (Pair x (Pair y z))” (5.59)
”_tpl (Pair x y)” => ”(Pair x y)” (5.60)

”_hoare Φ (_hOut r) Ψ” => ”hoare Φ r (_K Ψ)” (5.61)
”_hoare Φ (_hOut r) Ψ” <= ”hoare Φ r (λx. Ψ)” (5.62)
”_hoare Φ (_hOut’ tpl r) Ψ” == ”hoare Φ r (λtpl. Ψ)” (5.63)

Beispiel 5.12. *Das Hoare-Tripel aus Beispiel 5.10 wird mit Hilfe dieses Regelsatzes in folgender Weise übersetzt:*

$$\mathit{hoare}\ (\Phi \wedge_D b)\underbrace{(\mathit{hseq}\ (\mathit{hsingle}\ p)\ (\mathit{hseq}\ (\mathit{hbind}\ x\ q)\ (\mathit{hstep}\ (\mathit{hsingle}\ (r\ x)))))}\ (\Psi x)$$

$$\underbrace{(\mathit{hIn}\ \mathit{hseq}\ (\mathit{hsingle}\ p)\ (\mathit{hseq}\ (\mathit{hbind}\ x\ q)\ (\mathit{hstep}\ (\mathit{hsingle}\ (r\ x)))))} \qquad \text{(Anw. 5.47)}$$

$$\mathit{hseq}\ (\mathit{hsingle}\ p)\ \underbrace{(\mathit{hIn}\ (\mathit{hseq}\ (\mathit{hbind}\ x\ q)\ (\mathit{hstep}\ (\mathit{hsingle}\ (r\ x)))))} \qquad \text{(Anw. 5.50)}$$

$$\mathit{hseq}\ (\mathit{hbind}\ x\ q)\ \underbrace{(\mathit{hIn'}\ x\ (\mathit{hstep}\ (\mathit{hsingle}\ (r\ x))))} \qquad \text{(Anw. 5.51)}$$

$$\mathit{hOut'}\ (\underbrace{(\_\mathit{tpl}\ x)}\ (do\{r\ x; ret\ \underbrace{(\_\mathit{tpl}\ x)}\})) \qquad \text{(Anw. 5.52)}$$

$$\underbrace{\qquad x \qquad\qquad\qquad x \qquad} \qquad \text{(Anw. 5.59)}$$

$$\underbrace{\mathit{hOut'}\ x\ (q \gg= \lambda.\ do\{r\ x; ret\ x\})} \qquad \text{(Anw. 5.58)}$$

$$\underbrace{\mathit{hOut'}\ x\ (p \gg q \gg= \lambda.\ do\{r\ x; ret\ x\})} \qquad \text{(Anw. 5.57)}$$

$$\mathit{hoare}\ (\Phi \wedge_D b)\ (p \gg q \gg= \lambda.\ do\{r\ x; ret\ x\})\ (\lambda x.\ \Psi x) \qquad \text{(Anw. 5.63)}$$

Durch die Anwendung der Hoare-Tripel-Definition lässt sich dieser Term auffalten zu:

$$[a \leftarrow (\Phi \wedge_D b); z \leftarrow do\{p \gg q \gg\!= \lambda.do\{r\ x; ret\ x\}\}; b \leftarrow (\lambda x.\Psi x)\ z]\ a \rightarrow b$$

und mit Hilfe der Monaden-Gesetze vereinfachen zu:

$$[a \leftarrow (\Phi \wedge_D b); p; x \leftarrow q; r\ x; b \leftarrow (\lambda x.\Psi x)\ z]\ a \rightarrow b$$

Die in Kapitel 3 beschriebenen Beweise für das Hoare-Kalkül lassen sich mit der Bereitstellung der Syntax in Isabelle umsetzen. Da sie sich die Beschreibung der Beweise in Kapitel 3 sehr stark an der Isabelle-Umsetzung orientiert, wird sie hier nicht nochmals beschrieben. Stattdessen wird im nächsten Kapitel ein Beispiel für ihre Anwendung beschrieben.

6. Anwendungsbeispiel

Zu Beginn der Arbeit wurde propagiert, dass mit Hilfe des in Isabelle/Isar umgesetzten Kalküls die partielle Korrektheit monadischer Programme nachgewiesen werden kann. Diese Behauptung soll nun durch ein Beispiele untermauert werden.

6.1. Division Natürlicher Zahlen mit Rest

theory *ExampleProof = HoareCalc:*

Um die Anwendung des implementierten Hoare-Kalüls zu demonstrieren, wird der in [Hoa69] vorgestellte Beweis implementiert. Dies war der erste veröffentlichte Korrektheitsbeweis auf Basis der Hoare-Triple. Er zeigt, dass die folgende Programmsequenz eine korrekte Umsetzung der Division mit Rest beschreibt.

$$do\{r \leftarrow x; q \leftarrow 0; z \leftarrow while(y \leq r) do\{r \leftarrow (r-y); q \leftarrow (q+1); ret(r,q)\}\}$$

Da der Beweis unabhängig von der konkreten Belegung der einzelnen Variablen vor Eintritt in das Programm sein soll, wird als Vorbedingung `True` gewählt. Für die Nachbedingung muss `x=r+y*q` gelten. Dies ist nichts anderes als eine Umschreibung dafür, dass nach Programmabarbeitung `q` das ganzzahlige Ergebnis der Division sein soll und `r` der Rest, der dabei entsteht. Zu dem soll die Abbruchbedingung der Schleife erreicht sein. Es muss also zusätzlich *¬(y≤(r)* gelten.

Der hier aufgeführte Beweis folgt dem Schema, dass Hoare in [Hoa69] vorgeschlagen hat. Allerdings müssen die einzelnen Schritte zum Teil etwas detaillierter bearbeitet werden.

```
lemma
  "{|⇑True|}
     r←ret (x::nat);
     q←ret (0::nat);
     z←(while_D ↑(ret(y≤r)) do{
            r←ret(r-y);
            q←ret(1+q);
            ret()
        })
   {|(⇑(x=r+y*q))∧_D ¬_D(⇑(y≤r))|}"
proof -
```

Zunächst wird gezeigt, dass für die Sequenz der ersten beiden Programmzeilen *{|⇑True|}r←ret x;q←ret 0{|⇑(x=r+y*q)|}* gilt. [1]-[4] dienen dabei lediglich als Vorbereitung für den eigentlichen Sequenzierungsschritt, der in [5] durchgeführt wird.

```
have "[1]": "{|⇑True|}{|⇑(x=x+y*0)|}"
```

```
    proof -
      have "ret True ∈ Dsef"
        by (simp add: dsef_ret Dsef_def)
      moreover have "{|⇑True|}{|⇑True|}"
        by (simp add: emptySeq)
      ultimately show ?thesis
        by simp
    qed
  have "[2]": "{|⇑(x=x+y*0)|}r←ret x{|⇑(x=r+y*0)|}"
    proof -
      have "ret True ∈ Dsef"
        by (simp add: dsef_ret Dsef_def)
      moreover have "∀p. {|⇑True|}p{|⇑True|}"
        by (simp add: stateless)
      ultimately show ?thesis
        by (simp add: hoare_def gdj_def)
    qed
  have "[3]": "∀r. {|⇑(x=r+y*0)|}q←ret 0{|⇑(x=r+y*q)|}"
    proof -
      have "∀r. ret(x=r)∈ Dsef"
        by (simp add: dsef_ret Dsef_def)
      from this show ?thesis
        by (simp add: hoare_def gdj_def)
    qed
  from "[2]" and "[1]"
  have "[4]": "{|⇑True|}r←ret x{|⇑(x=r+y*0)|}"
    by (rule wk_pre)

  from "[4]" and "[3]"
  have "[5]": "{|⇑True|}r←ret x;q←ret 0{|⇑(x=r+y*q)|}"
    by (rule seq)
```

Um die schwächere Vorbedingung {|⇑*(x=(r-y)+y*(1+q))*|}, die bei der Bearbeitung der Sequenz *r←ret(r-y);q←ret(1+q)* benötigt wird, zu einer geeignet Vorbedingung für die Anwendung von while zu erweitern, muss gezeigt werden, dass aus der neuen Vorbedingung die schwächere folgt.

```
  have "[6]": "{|⇑(x=r+y*q)∧D ⇑(y≤r)|}{|⇑(x=(r-y)+y*(1+q))|}"
    proof -
      have "ret (x = r + y * q) ∈ Dsef"
        by (simp add: dsef_ret Dsef_def)
      moreover
      have "ret (y ≤ r) ∈ Dsef"
        by (simp add: dsef_ret Dsef_def)
      moreover
      have "ret (x = r - y + (y + y * q)) ∈ Dsef"
        by (simp add: dsef_ret Dsef_def)
      moreover
      have "ret (x = r + y * q ∧ y ≤ r) ∈ Dsef"
        by (simp add: dsef_ret Dsef_def)
      ultimately show ?thesis
```

```
      by (simp add: condConj_def hoare_Tupel_def gdj_def)
  qed
```

[7]-[9] bearbeiten die Sequenz in der Schleife und zeigen, dass für sie
`⦃⇑(x=(r-y)+y*(1+q))⦄r←ret(r-y);q←ret(1+q)⦃⇑(x=r+y*q)⦄` gilt.

```
have "[7]": "⦃⇑(x=(r-y)+y*(1+q))⦄r←ret(r-y)⦃⇑(x=r+y*(1+q))⦄"
  proof -
    have "ret (x = r - y + (y + y * q)) ∈ Dsef"
      by (simp add: dsef_ret Dsef_def)
    from this show ?thesis
      by (simp add: hoare_def gdj_def)
  qed
have "[8]": "∀r. ⦃⇑(x=r+y*(1+q))⦄q←ret(1+q)⦃⇑(x=r+y*q)⦄"
  proof -
    have "∀r. ret (x = r + (y + y * q)) ∈ Dsef"
      by (simp add: dsef_ret Dsef_def)
    from this show ?thesis
      by (simp add: hoare_def gdj_def)
  qed

from "[7]" and "[8]"
have "[9]":
  "⦃⇑(x=(r-y)+y*(1+q))⦄
      r←ret(r-y);q←ret(1+q)
   ⦃⇑(x=r+y*q)⦄"
  by (rule seq)
```

Mit Hilfe von [6] wird unter Verwendung der Regel wk_pre die Vorbedingung von [9] nun noch um die Schleifenbedingung erweitert. Das resultierende Hoare-Tripel ist die Vorraussetzung für die Anwendung der while-Regel in [11].

```
from "[9]" and "[6]"
have "[10]":
  "⦃⇑(x=r+y*q)∧D ⇑(y≤r)⦄
      r←ret(r-y);q←ret(1+q)
   ⦃⇑(x=r+y*q)⦄"
  by (rule wk_pre)

from "[10]" have "[11]":
  "∀r q. ⦃⇑(x=r+y*q)⦄
     z←whileD ↑(ret(y≤r)) do{r←ret(r-y);q←ret((1::nat)+q);ret()}
  ⦃⇑(x=r+y*q)∧D ¬D⇑(y≤r)⦄"
  proof -
    from "[10]" have "∀r q.
      ⦃⇑(x = r+y*q)∧D ⇑(y ≤ r)⦄
          z←do {r←ret (r - y); q←ret (1 + q); ret ()}
      ⦃⇑(x=r+y*q)⦄"
      apply (simp add: hoare_def condConj_def)
      apply (simp add: dsef_ret Dsef_def gdj_def)
      by simp
    moreover have
```

```
      "∀r q.
      ({|⇑(x = r+y*q)∧D ⇑(y ≤ r)|}
          z←do {r←ret (r - y); q←ret (1 + q); ret ()}
      {|⇑(x=r+y*q)|} ⟶
      {|⇑(x = r+y*q)|}
          z←iterD (λx. ↑ret(y ≤ r))
                   (λx. do {r←ret (r - y); q←ret (1+q);ret()})
                   ()
      {|⇑(x = r+y*q)∧D ¬D⇑(y ≤ r)|})"
      apply auto
      apply (rule iter)
      by simp
    ultimately have "
      ∀r q. {|⇑(x = r+y*q)|}
          z←iterD (λx. ↑ret(y ≤ r))
                   (λx. do {r←ret (r - y); q←ret (1+q);ret()})
                   ()
      {|⇑(x = r+y*q)∧D ¬D⇑(y ≤ r)|}"
      by simp
    from this have
      "∀r q. {|⇑(x=r+y*q)|}
          z←whileD ↑ret(y ≤ r) do {
              r←ret (r - y);
              q←ret (1 + q);
              ret ()}
       {|⇑(x=r+y*q)∧D ¬D⇑(y ≤ r)|}"
      by (simp add: whileD_def)
    from this show ?thesis
      by simp
  qed
```

Nun muss die Schleifensequenz nur noch angeängt werden. Dies lässt sich durch die erweiterte Regel für die Sequenzierung ohne weiteres tun.

```
  from "[5]" this show ?thesis
    by (rule seq_exp)
qed

end
```

7. Fazit

Nach der erfolgreichen Implementierung der Monaden in Isabelle konnten die mit ihnen in Verbindung stehenden Eigenschaften der Seiteneffektfreiheit, Kopierbarkeit, Vertauschbarkeit und deterministischen Seiteneffekfreiheit umgesetzt werden. Diese dienten uns als Grundlage für die Implementierung des monadischen Hoare-Kalküls. Zu dem konnte die Korrektheit des Kalküls durch den Beweis der einzelnen Lemmata gezeigt werden.

Wir haben damit die Möglichkeit geschaffen, monadische Programme bezüglich der ihnen zu Grunde liegenden Spezifikation formal zu überprüfen.

Anhand des in Kapitel 6 vorgestellten Beispiels konnten wie zeigen, wie die Programmverifikation unter Verwendung des implementierten Kalküls abläuft. Es wurde jedoch auch ersichtlich, dass das Konzept des Hoare-Kalküls sich nur begrenzt für die Anwendung auf grosse Programme eignet.

In der jetzigen Umsetzung wäre eine Hybridlösung bei der Fehlersuche denkbar. Kritische Programmabschnitte könnten mit Hilfe des Kalküls formal auf Korrektheit überprüft werden. Programmteile, die sich mit Randaufgaben beschäftigen können durch herkömmliche Tests auf Fehler untersucht werden. Bei einem solchen Lösungsansatz emfiehlt sich, das System möglichst fehlertollerant zu gestalten.

Für den letzendlichen Einsatz des in Isabelle umgesetzten monadischen Hoare-Kalküls sind einige Weiterentwicklungen der Theorien sinnvoll, die durch diese Arbeit nicht abgedeckt werden konnten.

7.1. Ausblick

Im Beispiel in Kapitel 6 zeigt sich deutlich, dass der eigentliche Beweis an vielen Stellen durch „triviale“ Regelanwendungen unterbrochen wird. So muss zum Beispiel explizit nachgewiesen werden, dass `ret True` deterministisch seiteneffektfrei auswertet. Durch eine geeignete Minipulation des *simpset* können diese Schritte automatisiert werden. Dazu bedarf es allerdings einer eingehenden Analyse, welche Regeln sich dafür eignen.

Ein weiteres Problem der implementierten Theorien das die Beweisführung unnötig erschwert ist die unflexible Anwendbarkeit der Lemmata bezüglich unterschiedlicher Sequenzlängen. Eine Möglichkeit dieses zu beheben ist die Kapselung von Sequenzen entsprechend der für die Anwendung des Lemmas benötigten Form. Die durch die bisherige Implementierung zur Verfügung stehende Kapselung der Form

$$(x, y) \leftarrow do\{x \leftarrow\ p; y \leftarrow\ q; ret\ (x, y)\}$$

reicht für diese Anwendung nicht aus, da Isabelle das Tupel nicht mit der gebundenen Variable in den Lemmata unifizieren kann. Ein möglicher Ansatz ist, die Bindung unabhängug von den durch den Lambda-Term zur Verfügung gestellten Variablen zu machen. Die Kapselung könnte dann wie folgt aussehen:

$$[u \leftarrow do\{x \leftarrow\ p; y \leftarrow\ q; ret\ (x, y)\}]\ \lambda xy.\Phi xy$$

Damit können Lemmata, die auf Sequenzen der Länge eins arbeiten, angewendet werden und man umgeht die bisherige Auflösung des Tupels durch `fst` und `snd` (siehe Beispiel 5.7).

Die Idee bei den folgenden bereits implementierten Regeln ist, bei der Syntax-Übersetzung von

$$[(y, x) \leftarrow do\{x \leftarrow\ p; y \leftarrow\ q; ret\ (x, y)\}]\ \lambda xy.\Phi xy$$

die obige Form im Hintergrund automatisch zu generieren.

consts
gdj' :: "pttrn ⇒ 'a T ⇒ ('a ⇒ bool) ⇒ bool" ("[_≺-_]_" [0, 100] 100)

syntax
"_gdjPBnd" :: "[idt, pttrn, 'a T] ⇒ bndstep" ("_≺-_←_")

translations
"_gdj (_gdjSeq (_gdjPBnd u x p) r) phi"
=> "gdj (_gdjSeq (_gdjBnd u p) (_gdjIn u x r)) phi"
"_gdj (_gdjPBnd u x p) phi" => "gdj p (λx. phi)"

"_gdjIn tpl tpl' (_gdjSeq (_gdjPBnd u x p) r)"
=> "_gdjSeq (_gdjBnd u p) (_gdjIn (tpl, u) (tpl', x) r)"
"_gdjIn tpl tpl' (_gdjPBnd x u p)"
=> "_gdjOut (_reTpl tpl' u) (do {x←p; ret(_reTpl tpl x)})"

Dadurch lassen sich gdj der Form

$$[u \prec -\ (y, x) \leftarrow do\{x \leftarrow\ p; y \leftarrow\ q; ret\ (x, y)\}]\ \lambda xy.\Phi xy$$

formulieren. Die explizite Angabe von u ist dadurch bedingt, dass wie in Kapitel 4 beschrieben, bei einer Syntax-Übersetzung keine neuen Variablen eingeführt werden können.

Mit der Verfeinerung dieses Konzepts und einer äquivalenten Implementierung für die Hoare-Tripel würde sich die Beweisführung stark vereinfachen lassen.

Ein dritter Ansatzpunkt für die Erweiterung der Theorien, der weniger der Anwendbarkeit denn der Akzeptanz des Kalküls dient, ist die Formulierung der Axiome `commute_tcoerc` und `iter` als Lemmata und damit einhergehend der Beweis ihrere Korrektheit.

7.2. Danksagung

Mein besonderer Dank gilt zunächst ersteinmal meiner Familie, die mir während meines ganzen Studiums als Ruhepol den Rücken gestärkt hat und Henning Mersch, der mich insbesondere während der Diplomarbeit aufgefangen hat, wenn die Motivation mal wieder fehlte. Desweiteren ein Dankeschön an all die fleissigen Helferlein, die sich die Mühe gemacht haben, die von mir so sorgfälltig eingebauten Fehler in dieser Arbeit aufzuspüren. Insbesondere sei Annett Wenzel lobend erwähnt, die mich mit hilfreichen Tipps unterstützt hat. Auch Dennis Walter, der mir an vielen aussichtslos erscheinenden Stellen gute Hinweise geliefert hat, sei an dieser Stelle nicht veressen.

A. Isabelle-Theorien

A.1. MonadSyntax

theory *MonadSyntax = Main:*

Definition of monad type and the two monadic funtions „≫=“ and ret

```
typedecl 'a T

consts
  bind  :: "'a T ⇒ ('a ⇒ 'b T) ⇒ 'b T"  ("_ ≫= _" [5, 6] 5)
  ret   :: "'a ⇒ 'a T"

constdefs
  bind' :: "'a T ⇒ 'b T ⇒ 'b T"               ("_ ≫ _" [5, 6] 5)
  "bind' p q == bind p (λx. q)"
```

This sets up a Haskell-style „do{x←p;y←q x;...}“ syntax. Initial taken from Christop Lueth and extended by Dennis Walter.

```
nonterminals
  monseq

syntax
  "_monseq"  :: "monseq ⇒ 'a T"                  ("(do {(_)})"    [5]
100)
  "_mongen"  :: "[pttrn, 'a T, monseq]⇒ monseq"  ("((_←(_));/ _)" [110,6,5]5)
  "_monexp"  :: "['a T, monseq]⇒ monseq"         ("((_);/ _)"     [6,5]
5)
  "_monexp0" :: "['a T] ⇒ monseq"                ("(_)"           5)

translations

  "_monseq(_mongen x p q)"    => "p ≫= (%x. (_monseq q))"
  "_monseq(_monexp p q)"      => "p ≫ (_monseq q)"
  "_monseq(_monexp0 q)"       => "q"

  "_monseq(_mongen x p q)"    <= "p ≫= (%x. q)"
  "_monseq(_monexp p q)"      <= "p ≫ q"

  "_monseq(_monexp p q)"      <= "_monseq (_monexp p (_monseq q))"
  "_monseq(_mongen x p q)"    <= "_monseq (_mongen x p (_monseq q))"
```

fstUnitLaw: left unity of ret
sndUnitLaw: right unity of ret
assocLaw: assiciativity

```
axioms
 lunit: "(ret x ≫= p) = p x"
 runit: "(p ≫= ret) = p"
 assoc: "(p ≫= q ≫= r) = (p ≫= (λx. q x ≫= r))"
 injective: "ret x = ret z ⟹ x = z"

lemma fstUnitLaw [simp]:  "(do {y←ret x; p y}) = p x"
 by (rule lunit)

lemma sndUnitLaw [simp]:  "(do {x←p; ret x}) = p"
 by (rule runit)

lemma assocLaw [simp]:
  "do {y←do {x←p; q x}; r y} = do {x←p; y←q x;r y}"
  by (rule assoc)
```

We also want associtivity for q and r coming along with no parameter. For this purpose we need three more variants of the assocLaw.

```
lemma do_assoc1[simp]:
  "(do{do{x←p;q x}; r}) = (do {x←p; q x; r})"
  apply (unfold "bind'_def")
  by (rule assocLaw)

lemma do_assoc2[simp]:
  "(do {x←(do {p; q}); r x}) = (do {p; x←q; r x})"
  apply (unfold "bind'_def")
  by (rule assocLaw)

lemma do_assoc3[simp]:
  "(do {(do {p; q}); r})= (do {p; q; r})"
  apply (unfold "bind'_def")
  by (rule assocLaw)

lemma delBind[simp]: "do {x←p; q} = do {p; q}"
  apply (unfold bind'_def)
  by simp
```

Syntactic sugar for True and False

```
syntax
  "_b1"      :: "bool"                ("⊤")
  "_b2"      :: "bool"                ("⊥")

translations
  "_b1" == "True"
  "_b2" == "False"
```

α *res* Φ = α *iff* α is defined and Φ holds

constdefs

```
"res" :: "'a ⇒ bool ⇒ 'a" (infixl "res" 100)
"α res Φ == if Φ then α else arbitrary"
```

if_then_else lifted to monadic values

constdefs

```
ifT :: "bool T ⇒ 'a T ⇒ 'a T ⇒ 'a T"   ("ifT(_)then(_)else(_)")
"ifT b then p else q == do{x←b;if x then p else q}"
```

consts

```
iterT:: "('a ⇒ bool T) ⇒ ('a ⇒ 'a T) ⇒ 'a ⇒ 'a T"
whileT :: "bool T ⇒ unit T ⇒ unit T" ("whileT (_) (_)")
```

axioms

```
iterT_def: "iterT test f x =
     do{ifT (test x) then (do{y←(f x);iterT test f y}) else (ret
x)}"
whileT_def: "whileT b p == iterT (λx. b) (λx. p) ()"
```

some definitions for monadic-sequenzes

constdefs

```
"sef" :: "'a T ⇒ bool"  ("sef _")
"sef p == (do{y←p; ret ()} = ret ())"


"cp" :: "'a T ⇒ bool"
"cp p == (do{x←p;y←p;ret(x,y)} = do{x←p;ret(x,x)})"


"com" :: "'a T ⇒ bool T ⇒ bool" ("_ comwith _")
"(com p q) == (((cp q) ∧ (sef q)) ⟶
                (cp(do {x←p; y←q; ret(x,y)})))"
```

we can not quantify over types in com_def. so we need a rule which lift the definition from bool T up to 'a T

axioms *commute_tcoerc:*

```
"∀q::bool T.(cp q) ∧ (sef q) ⟶ (cp (do{x←p; y←q; ret(x,y)})) ⟹
 ∀q. (cp q) ∧ (sef q) ⟶ (cp (do{x←p; y←q; ret(x,y)}))"
```

axioms *tst:*

```
"∀x. ∀q::'b T.
     (cp q) ∧ (sef q) ⟶ (cp (do{z←p x; y←q; ret(z,y)}))⟹
 ∀x. (∀q. (cp q) ∧ (sef q) ⟶ (cp (do{z←p x; y←q; ret(z,y)})))"
```

end

A.2. Lemmabase

```
theory Lemmabase = MonadSyntax:


lemma "sef_retUnit": "(p = ret ()) = sef p"
  apply (rule iffI)
  apply (simp_all add: sef_def)
done



lemma "sef_retUnit": "(p = ret ()) = sef p"
proof
  assume "p = ret ()"
  from this have "sef (ret ())"
    by (simp add: sef_def)
  from prems this show "sef p"
    by blast
next

  assume "sef p"
  from this have "p = do {y←p;ret ()}"
    by simp
  moreover
  from prems this have "p = ret ()"
    apply (unfold sef_def)
    by simp
  ultimately show "p = ret ()"
    by blast

qed

lemma "seFree":
  assumes "sef p"
  shows "do {p;q} = q"

proof -
  have "do {p;q} = do {p; ret (); q}"
    by (simp add: bind'_def del: delBind)
  also
  have "... = do {do{p; ret ()}; q}"
    by simp
  also
  from prems have "... = do {ret ();q}"
    by (simp add: sef_def)
  also
  have "... = q"
```

```
    by (simp add: bind'_def del: delBind)
  finally show "do {p;q} = q" .
qed


lemma "seFree2":
  assumes ret_q: "q = ret()" and sef_p:"sef p"
  shows "do {p;q} = p"

proof -
  from prems have "p = ret()"
    by (simp only: sef_retUnit)
  from this ret_q have "p = q"
    by blast
  moreover
  from sef_p have "do{p;q} = q"
    by (simp only: seFree)
  ultimately show ?thesis
    by simp
qed


lemma ret2seq:
  assumes "do {x←p;ret x} = do {x←q;ret x}"
  shows "do {x←p;r x} = do {x←q;r x}"

proof -
  have "do {x←p;r x} = do {z←(do {x←p;ret x});r z}"
    apply (subst sndUnitLaw) ..
  moreover from prems
  have "... = do {z←(do {x←q;ret x});r z}"
    by auto
  moreover
  have "... = do {x←q;r x}"
    apply (subst sndUnitLaw) ..
  ultimately show ?thesis by simp
qed


lemma ret2seq_exp:
  assumes "do {x←p1;y←p2 x;ret (x,y)} = do {x←q1;y←q2 x;ret (x,y)}"
  shows "do {x←p1;y←p2 x;r x y} = do {x←q1;y←q2 x;r x y}"

proof -
  from prems have
    "do {z←(do{x←p1;y←p2 x;ret (x,y)});ret z} =
     do {z←(do{x←q1;y←q2 x;ret (x,y)});ret z}"
    by (simp only: sndUnitLaw)
  from this have
```

```
    "do {z←(do{x←p1;y←p2 x;ret (x,y)});r (fst z) (snd z)} =
     do {z←(do{x←q1;y←q2 x;ret (x,y)});r (fst z) (snd z)}"
    by (rule ret2seq)
  from this show ?thesis
    by simp
qed


lemma ret2seqSw:
  assumes "do {x←p1;y←p2 x;ret (y,x)} =
           do {x←q1;y←q2 x;ret (y,x)}"
  shows "do {x←p1;y←p2 x;r x y} = do {x←q1;y←q2 x;r x y}"

proof -
  have
    "do {z←(do{x←p1;y←p2 x;ret (y,x)});ret z} =
     do {z←(do{x←q1;y←q2 x;ret (y,x)});ret z}"
    by (simp only: sndUnitLaw)
  from this have
    "do {z←(do{x←p1;y←p2 x;ret (y,x)});r (snd z) (fst z)} =
     do {z←(do{x←q1;y←q2 x;ret (y,x)});r (snd z) (fst z)}"
    by (rule ret2seq)
  from this show ?thesis
    by simp
qed


lemma ret2seqSw':
  assumes "do {x←p1;y←p2;ret (x,y)} =
           do {y←q1;x←q2;ret (x,y)}"
  shows "do {x←p1;y←p2;r x y} = do {y←q1;x←q2;r x y}"

proof -
  have
    "do {z←(do{x←p1;y←p2;ret (x,y)});ret z} =
     do {z←(do{y←q1;x←q2;ret (x,y)});ret z}"
    by (simp only: sndUnitLaw)
  from this have
    "do {z←(do{x←p1;y←p2;ret (x,y)});r (fst z)(snd z)} =
     do {z←(do{y←q1;x←q2;ret (x,y)});r (fst z)(snd z)}"
    by (rule ret2seq)
  from this show ?thesis
    by simp
qed


lemma cp_ret2seq:
  assumes "cp p"
  shows "do {x←p;y←p;r x y} = do{x←p;r x x}"
```

```
proof -
  from prems have
    "do {x←p; y←p; ret (x, y)} = do {x←p; ret (x, x)}"
    by (simp add: cp_def)
  from this have
    "do{z←do {x←p; y←p; ret (x, y)};ret z} =
     do{z←do {x←p; ret (x, x)};ret z}"
    by simp
  from this have
    "do{z←do {x←p; y←p; ret (x, y)};r (fst z) (snd z)} =
     do{z←do {x←p; ret (x, x)};r (fst z) (snd z)}"
    by (simp add: "seFree")
  from this show ?thesis
    by simp
qed


lemma eq_res:
  "∀x. ((r x) res ((r x)=(s x))) = ((s x) res ((r x)=(s x)))"

proof (cases "∀x.(r x)=(s x)")
  case True
  from this show ?thesis
    by (simp add: res_def)
next
  case False
  from this show ?thesis
    by (simp add: res_def)
qed



lemma "cpsefProps(i→ii)":
  assumes sef_p: "sef p" and sef_q: "sef q" and
          i: "cp(do{x←p;y←q;ret(x,y)})"
  shows "do{x←p;y←q;ret(x,y)} = do{y←q;x←p;ret(x,y)}"

proof -
  let ?s = "do{x←p;y←q;ret(x,y)}"
  from sef_p sef_q have sef_s: "sef ?s"
    by (simp add: sef_def del: delBind)
  from i have cp_s: "cp ?s"
    by simp

  have "?s = do{z←?s; ret z}"
    by simp
  moreover from sef_s have "... = do{z←do{?s;?s};ret z}"
    apply (subst "seFree")
```

```
    apply blast ..
  moreover have "... = do{w←?s;z←?s;ret (fst z, snd z)}"
    by simp
  moreover
  from cp_s sef_s have "... = do{w←?s;z←?s;ret (fst z, snd w)}"

  proof -
    from sef_s have
      "do{w←?s;z←?s;ret (fst z, snd z)} =
       do{w←?s;ret(fst w,snd w)}"
      by (simp add: seFree del: do_assoc1)
    moreover
    from prems have "... = do{w←?s;z←?s;ret(fst z, snd w)}"
      by (simp add: cp_ret2seq del: assocLaw)
    ultimately show ?thesis
      by (simp del: assocLaw)
  qed

  moreover have "... = do{u←p;v←q;x←p;y←q;ret(x,v)}"
    by simp
  moreover from sef_p sef_q have "... = do{v←q;x←p;ret(x,v)}"
    by (simp add: seFree)
  ultimately show ?thesis
    by simp
qed


lemma "cpsefProps(ii→iii)":
  assumes ii: "(do{x←p;y←q;ret(x,y)} = do{y←q;x←p;ret(x,y)})"
  shows "do{x←p;y←q;r x y} = do{y←q;x←p;r x y}"

proof -
  from ii have
    "do{x←do{x←p;y←q;ret(x,y)};ret x} =
     do{x←do{y←q;x←p;ret(x,y)};ret x}"
    by simp
  from this have
    "do{x←do{x←p;y←q;ret(x,y)};r (fst x) (snd x)} =
     do{x←do{y←q;x←p;ret(x,y)};r (fst x) (snd x)}"
    by (rule ret2seq)
  from this show ?thesis
    by simp
qed


lemma weak_sef2seq:
  assumes "sef p" "∀x. sef (r x)"
  shows "sef (do {x←p; r x})"
```

```
proof -
  have
    "do {z←do {x←p; r x}; ret ()} =
     do {x←p; do{r x; ret ()}}"
    by simp
  moreover from prems have "... = do {x←p; ret()}"
    by (simp add: sef_def)
  moreover from prems have "... = ret ()"
    by (simp add: sef_def)
  ultimately show ?thesis
    by (simp add: sef_def)
qed


lemma weak_cp2retSeq:
  assumes "cp p"
  shows "cp (do{x←p; ret(a x)})"

proof -
  have
    "(do{u←do{x←p;ret(a x)};v←do{y←p;ret (a y)};ret (u,v)})=
     (do{x←p;u←ret(a x);y←p;v←ret (a y);ret (u,v)})"
    by (simp only: assocLaw)
  moreover have
    "... = (do{x←p; y←p;ret(a x, a y)})"
    by (simp only: fstUnitLaw)
  moreover from prems have
    "... = (do{x←p;ret(a x,a x)})"
    by (simp only: cp_ret2seq)
  moreover have
    "... = do{x←p;u←ret(a x);ret(u,u)}"
    by (simp only: fstUnitLaw)
  moreover have
    "... = do{u←do{x←p;ret(a x)};ret(u,u)}"
    by (simp only: assocLaw)
  ultimately show ?thesis
    by (simp only: cp_def)
qed



end
```

A.3. gdjSyntax

```
theory gdjSyntax = MonadSyntax:
```

Definition of gdj-syntax for a calculus on which the Hoare-calculs can be proven easily.

constdefs

```
gdj :: "'a T ⇒ ('a ⇒ bool) ⇒ bool"     ("[_]_" [0, 100] 100)
"gdj p Φ ==
  ((do {x_g←p; ret (Φ x_g, x_g)}) = (do {x_g←p; ret (True, x_g)}))"

empty_gdj :: "bool ⇒ bool"
"empty_gdj Φ == (Φ = True)"
```

Syntax translations that let you write e.g. `[x←p; y←q](ret (x=y))` for `gdj (do {x←p; y←q; ret (x,y)}) (λ(x,y). (x=y))` Essentially, these translations collect all bound variables inside the boxes and return them as a tuple. The lambda term that constitutes the second argument of gdj will then also take a tuple pattern as its sole argument. Moreover the tuple is sorted so that nested tupeling is solved.

nonterminals

```
bndseq bndstep
```

syntax

```
"_empty_gdj"  :: "[bool] ⇒ bool"                   ("[ ]_" 100)
"_gdj"        :: "[bndseq, bool] ⇒ bool"          ("[_]_" [0,100]100)

"_gdjBnd"     :: "[pttrn, 'a T] ⇒ bndstep"        ("_←_")
"_gdjSeq"     :: "[bndstep, bndseq] ⇒ bndseq"     ("_;/ _")
""             :: "[bndstep] ⇒ bndseq"              ("_")

"_gdjIn"      :: "[pttrn, bndseq] ⇒ bndseq"
"_gdjOut"     :: "[pttrn, bndseq] ⇒ bndseq"

"_reTpl"      :: "[pttrn, pttrn] ⇒ pttrn"
```

translations

```
"_empty_gdj Φ" == "empty_gdj Φ"

"_gdj (_gdjBnd x p) phi"  == "gdj p (λx. phi)"
"_gdj (_gdjSeq (_gdjBnd x p) r) phi"
            =>  "gdj (_gdjSeq (_gdjBnd x p) (_gdjIn x x r)) phi"

"_gdjIn tpl tpl' (_gdjSeq (_gdjBnd x p) r)"
            =>  "_gdjSeq (_gdjBnd x p) (_gdjIn (tpl, x) (tpl', x) r)"
"_gdjIn tpl tpl' (_gdjBnd x p)"
            =>  "_gdjOut (_reTpl tpl' x) (do {x←p; ret(_reTpl tpl
x)})"

"_gdjSeq (_gdjBnd x p) (_gdjOut tpl r)" ==  "_gdjOut tpl (do {x←p;
r})"

"_reTpl (_tuple tpl (_tuple_arg x)) y"
        => "_reTpl tpl (_tuple x (_tuple_arg y))"
```

```
"_reTpl x y" => "(x,y)"

"Pair x (_pattern y z)" => "Pair x (Pair y z)"
"Pair (_pattern x y) z" => "Pair x (Pair y z)"
"Pair x (_patterns y z)" => "Pair x (Pair y z)"
"Pair (_patterns x y) z" => "Pair x (Pair y z)"
"(_pattern x (_patterns y z))" => "Pair x (Pair y z)"

"gdj (_gdjOut tpl r) phi" == "gdj r (λtpl. phi)"
```

syntax for encapsulation of sequences

consts
```
gdj' :: "pttrn ⇒ 'a T ⇒ ('a ⇒ bool) ⇒ bool"
                                        ("[_≺-_]_" [0, 100] 100)
```

syntax
```
"_gdjPBnd"   :: "[idt, pttrn, 'a T] ⇒ bndstep" ("_≺-_←_")
```

translations
```
"_gdj (_gdjSeq (_gdjPBnd u x p) r) phi"
        =>  "gdj (_gdjSeq (_gdjBnd u p) (_gdjIn u x r)) phi"
"_gdj (_gdjPBnd u x p) phi"  => "gdj p (λx. phi)"

"_gdjIn tpl tpl' (_gdjSeq (_gdjPBnd u x p) r)"
     =>  "_gdjSeq (_gdjBnd u p) (_gdjIn (tpl, u) (tpl', x) r)"
"_gdjIn tpl tpl' (_gdjPBnd x u p)"
     =>  "_gdjOut (_reTpl tpl' u) (do {x←p; ret(_reTpl tpl x)})"
```

end

A.4. ddjCalc

theory *gdjCalc = gdjSyntax + Lemmabase:*

lemma *gdj2doSeq:*
  assumes *"[x←p] Φ x"*
  shows *"do {x←p;q x (Φ x)} = do {x←p;q x ⊤}"*

proof -
  from *prems*
  have *"do {x←p;ret (Φ x, x)} = do{x←p;ret(⊤, x)}"*
    by *(fold gdj_def)*
  from *this*
  have
    "do{x←p;y←ret (Φ x);ret (y, x)} =
     do{x←p;y←ret ⊤;ret(y, x)}"
    by *(simp only: fstUnitLaw)*

```
  from this
  have "do {x←p;y←ret (Φ x);q x y} = do{x←p;y←ret ⊤;q x y}"
    by (rule  ret2seqSw)
  from this show ?thesis
    by (simp only: fstUnitLaw)
qed

lemma gdjEq2doSeq:
  assumes a: "[x←p] ((q1 x)=(q2 x))"
  shows "do {x←p;r x (q1 x)} = do {x←p;r x (q2 x)}"

proof -
  have "do {x←p;r x (q1 x)} = do {x←p;r x ((q1 x) res ⊤)}"
    by (simp add: res_def)
  moreover
  from prems have "... = do {x←p;r x ((q1 x) res ((q1 x)=(q2 x)))}"
    apply (subst gdj2doSeq)
    apply simp
    by blast
  moreover
  have "... = do {x←p;r x ((q2 x) res ((q1 x)=(q2 x)))}"
    proof -
      have
        "∀x. ((q1 x) res ((q1 x)=(q2 x)) =
             ((q2 x) res ((q1 x)=(q2 x))))"
        by (rule eq_res)
      from this show ?thesis
        by simp
    qed
  moreover
  from prems have "... = do {x←p;r x (q2 x)}"
    apply (unfold res_def)
    apply (subst gdj2doSeq)
    apply simp
    apply (unfold if_def)
    by simp
  ultimately show ?thesis
    by simp
qed

lemma doSeq2gdjEq:
  assumes "do{x←p;ret(x,(a₁ x)::'a)} = do{x←p;ret(x,(a₂ x)::'a)}"
  shows "[x←p](a₁ x=a₂ x)"

proof -
  from prems have
    "do{x←p;y←ret (a₁ x); ret(x,y)} =
     do{x←p;y←ret (a₂ x); ret(x,y)}"
    by simp
```

```
  from this have
    "do{u←do{x←p;y←ret (a₁ x); ret(x,y)};ret u} =
     do{u←do{x←p;y←ret (a₂ x); ret(x,y)};ret u}"
    by simp
  from this have
    "do{(x,y)←do{x←p;y←ret (a₁ x); ret(x,y)};
                  ret (y=(a₂ x), x)} =
     do{(x,y)←do{x←p;y←ret (a₂ x); ret(x,y)};
                  ret (y=(a₂ x), x)}"
    by (rule ret2seq)
  from this show ?thesis
    by (simp add: gdj_def)
qed

lemma sef2cp:
  assumes "sef p"
  shows "cp p = [x←p;y←p](x=y)"

proof
  assume "cp p"
  from this have "(do{x←p;y←p;ret(x,y)} = do{x←p;ret(x,x)})"
    by (simp add: cp_def)
  from this prems have
    "(do{x←p;y←p;ret(x,y)} = do{x←p;y←p;ret(x,x)})"
    by (simp add: "seFree")
  from this have
    "(do{(x,y)←do{x←p;y←p;ret(x,y)};ret ((x,y),x)} =
    do{(x,y)←do{x←p;y←p;ret(x,y)};ret ((x,y),y)})"
    by simp
  from this have "[z←do{x←p;y←p;ret(x,y)}](fst z=snd z)"
    by (simp add: doSeq2gdjEq)
  from this show "[x←p;y←p](x=y)"
    by (simp add: gdj_def)
next
  assume "[x←p;y←p](x=y)"
  from this have "[z←do{x←p;y←p;ret(x,y)}](fst z=snd z)"
    by (simp add: gdj_def)
  from this have
    "(do{z←do{x←p;y←p;ret(x,y)};ret(fst z, (fst z))} =
    do{z←do{x←p;y←p;ret(x,y)};ret (fst z, (snd z))})"
    by (rule gdjEq2doSeq)
  from this have "(do{x←p;y←p;ret(x,y)} = do{x←p;y←p;ret(x,x)})"
    by simp
  from this prems have
    "(do{x←p;y←p;ret(x,y)} = do{x←p;ret(x,x)})"
    by (simp add: "seFree")
  from this show "cp p"
    by (simp add: cp_def)
qed
```

```
lemma "cpsefProps(ii→iv)":
  assumes cp_p:  "cp p" and sef_p: "sef p" and
          ii: "(do{x←p;y←q;ret(x,y)} = do{y←q;x←p;ret(x,y)})"
  shows "[x←p;y←q;z←p](x=z)"

proof -
  from prems have t: "[(x, z)←do {x←p; z←p; ret (x, z)}](x = z)"
    by (simp add: sef2cp)
  from this have "[u←do{x←p;z←p;ret(x,z)}] (fst u=snd u)"
    by (simp add: gdj_def)
  from this have
    "do {u←do{x←p; z←p; ret(x,z)};
         v←q;
         ret ((fst u)=(snd u),fst u,snd u,v)} =
     do {u←do{x←p; z←p; ret(x,z)};
         v←q;
         ret (⊤,fst u,snd u,v)}"
    by (rule gdj2doSeq)
  from this have
    "do {x←p;z←p;v←q;ret (x=z,x,z,v)} =
     do {x←p;z←p;v←q;ret (⊤,x,z,v)}"
    by simp
  from this
  have "[(x,z,y)←do{x←p; z←p; y←q;ret(x,z,y)}](x = z)"
    by (simp add: gdj_def)
  from this have
      "[(x,z,y)←do{x←p; (z,y)←do{z←p;y←q;ret(z,y)}; ret(x,z,y)}]
                                                          (x = z)"
    by simp
  moreover
  from ii have "do{z←p;y←q;ret(z,y)} = do{y←q;z←p;ret(z,y)}"
    by simp
  ultimately have
    "[(x,z,y)←do{x←p; (z,y)←do{y←q;z←p;ret(z,y)};ret(x,z,y)}]
                                                          (x = z)"
    by simp
  from this have
    "[u←do{x←p;y←q;z←p;ret(x,z,y)}]((fst u) = (fst(snd u)))"
    by (simp add: gdj_def)
  from this have
    "(do{u←do{x←p;y←q;z←p;ret(x,z,y)};
         ret((fst u)=fst(snd u),fst u,snd(snd u),fst(snd u))}) =
     (do{u←do{x←p;y←q;z←p;ret(x,z,y)};
         ret(True,fst u,snd(snd u),fst(snd u))})"
    by (rule gdj2doSeq)
  from this show ?thesis
    by (simp add: gdj_def)
qed
```

```
lemma "cpsefProps(iv→iii)":
  assumes sef_p: "sef p" and iv: "[x←p;y←q;z←p](x=z)"
  shows "do{x←p;y←q;r x y} = do{y←q;x←p;r x y}"

proof -
  from sef_p have "do{x←p;y←q;r x y} = do{x←p;y←q;z←p;r x y}"
    by (simp add: seFree)
  moreover have
    "... = do{x←p;y←q;z←p;r z y}"
    proof -
      from iv have
        "[u←do{x←p;y←q;z←p;ret(x,y,z)}](fst u=snd(snd u))"
        by (simp add: gdj_def)
      from this have
        "do{u←do{x←p;y←q;z←p;ret(x,y,z)};
                      r (fst u) (fst(snd u))} =
         do{u←do{x←p;y←q;z←p;ret(x,y,z)};
                      r (snd(snd u)) (fst(snd u))}"
        by (rule gdjEq2doSeq)
      from this show ?thesis
        by simp
    qed
  moreover from prems have
    "... =  do{y←q;z←p;r z y}"
    by (simp add: seFree)
  ultimately show ?thesis
    by simp
qed

axioms seq2ret:
  "do{x←p;y←q;r x y} = do{y←q;x←p;r x y} ⟹ do{y←q;x←p;ret(x,y)}
= do{x←p;y←q;ret(x,y)}"

lemma "cpsefProps(iii→i)":
  assumes sef_p: "sef p" and sef_q: "∀x. sef q" and
          cp_p: "cp p" and cp_q: "cp q" and
          iii: "do{x←p;y←q;r x y} = do{y←q;x←p;r x y}"
  shows "cp(do{x←p;y←q;ret(x,y)})"

proof -
  let ?s = "do{x←p;y←q;ret(x,y)}"

  have "do{w←?s;z←?s;ret (w,z)} = do{u←p;w←do{v←q;x←p;ret(v,x)};y←q;ret((u,fst
w),(snd w,y))}"
    by simp
  moreover from prems have "... = do{u←p;w←do{x←p;v←q;ret(v,x)};y←q;ret((u,fst
w),(snd w,y))}"
    apply (subst seq2ret)
```

```
    apply auto
    apply (rule seq2ret)
    by simp
  moreover have "... = do{u←p;x←p;do{v←q;y←q;ret((u,v),(x,y))}}"
    by simp
  moreover from cp_p have "... = do{u←p;do{v←q;y←q;ret((u,v),(u,y))}}"
    by (simp only: cp_ret2seq)
  moreover from cp_q have "... = do{u←p;do{v←q;ret((u,v),(u,v))}}"
    by (simp only: cp_ret2seq)
  ultimately have "do{w←?s;z←?s;ret (w,z)} = do{u←p;do{v←q;ret((u,v),(u,v))}}"
    by simp
  moreover have "... = do{w←?s;ret(w,w)}"
    by simp
  ultimately have "do{w←?s;z←?s;ret (w,z)} = do{w←?s;ret(w,w)}"
    by (rule trans)
  from this show ?thesis
    by (simp only: cp_def)
qed

lemma switch:
  assumes com_a: "∀q. a comwith q" and
          sef_a: "sef a" and
          cp_b:"cp b" and sef_b: "sef b"
  shows "do{x←a;y←b;ret(x,y)} = do{y←b;x←a;ret(x,y)}"

proof -
  from com_a sef_a have
    "(∀q::'b T. (cp q) ∧ (sef q) ⟶
                  cp (do {x←a; y←q; ret (x, y)}))"
    apply (unfold com_def)
    by (rule com_def commute_tcoerc)
  from com_a sef_b cp_b this have cp_do:
    "cp(do {x←a; y←b; ret(x,y)})"
    by (simp add: com_def)
  from sef_a sef_b cp_do show ?thesis
    by (rule "cpsefProps(i→ii)")
qed

lemma switch2:
  assumes  com_b: "∀q::bool T. b comwith q" and
           sef_b: "sef b"  and
           cp_a:  "cp a" and sef_a: "sef a"
  shows "do{x←a;y←b;ret(x,y)} = do{y←b;x←a;ret(x,y)}"

proof -
  from prems have "do{y←b;x←a;ret(y,x)}=do{x←a;y←b;ret(y,x)} "
    by (rule switch)
  from this have "do{y←b;x←a;ret(x,y)}=do{x←a;y←b;ret(x,y)}"
    by (rule ret2seqSw')
```

```
  from this show ?thesis
    by simp
qed

lemma switch':
  assumes com_a: "∀z. (∀q::bool T. (a z) comwith q)" and
          sef_a: "∀z. sef (a z)"  and
          cp_b:  "∀z. cp (b z)" and
          sef_b: "∀z. sef (b z)"
  shows "∀z. do{x←a z;y←b z;ret(x,y)} =
             do{y←b z;x←a z;ret(x,y)}"

proof -
  from com_a have
    "∀z. (∀q::bool T. (cp q) ∧ (sef q) ⟶
                       cp (do {x←(a z); y←q; ret (x, y)}))"
    by (simp add: com_def)
  from this have
    "∀z. (∀q::'c T. (cp q) ∧ (sef q) ⟶
               cp (do {x←(a z); y←q; ret (x, y)}))"
    by (rule tst)
  from com_a sef_b cp_b this have cp_do:
    "∀z. cp(do {x←(a z); y←(b z); ret(x,y)})"
    by (simp add: com_def)
  have all:
      "∀z. ((sef (a z)) ∧ (sef (b z)) ∧
            cp(do {x←(a z); y←(b z); ret(x,y)})) ⟹
       ∀z. (do {x←(a z); y←(b z); ret (x, y)} =
            do {y←(b z); x←(a z); ret (x, y)})"
    apply (rule allI)
    apply (rule "cpsefProps(i→ii)")
    by simp_all
  from sef_a sef_b cp_do have
    "∀z. ((sef (a z)) ∧ (sef (b z)) ∧
          cp(do {x←(a z); y←(b z); ret(x,y)}))"
    by simp
  from all this show ?thesis
    by simp
qed

lemma switch3:
  assumes com_a: "∀z q. (a z) comwith q" and
          sef_a: "∀z. sef (a z)" and
          cp_b:  "cp b" and
          sef_b: "sef b"
  shows "∀z. do {x←a z; y←b; ret (x, y)} = do {y←b; x←a z; ret (x,
y)}"

proof (rule switch')
```

```
  from prems show "∀z q. a z comwith q"
    by simp
  from prems show "∀z. sef a z"
    by simp
  from prems show "∀z. cp b"
    by simp
  from prems show "∀z. sef b"
    by simp
qed
```

example: assumes *[a←Φ;x←p;b←Ψ x;c←ϒ x] (a⟶(r b c))* shows *[a←Φ;x←p;d←do{b←Ψ x;c←ϒ x;ret(r b c)}] (a⟶d)*

```
lemma postOp:
  assumes "[x←p;y←q x;z←r x y;v←s x y z] Φ x y (t z v)"
  shows "[x←p;y←q x;d←do{z←r x y;v←s x y z;ret(t z v)}] Φ x y d"

proof -
  from prems have
    "do{u←do{x←p;y←q x;z←r x y;v←s x y z;ret(x,y,z,v)};
        z←ret(Φ (fst u) (fst(snd u))
             (t (fst(snd(snd u))) (snd(snd(snd u)))));
        ret(z,u)}=
    do{u←do{x←p;y←q x;z←r x y;v←s x y z;ret(x,y,z,v)};
        z←ret(⊤);
        ret(z,u)}"
    by (simp add: gdj_def)
  from this have
    "do{u←do{x←p;y←q x;z←r x y;v←s x y z;ret(x,y,z,v)};
        z←ret(Φ (fst u) (fst(snd u))
              (t (fst(snd(snd u))) (snd(snd(snd u)))));
        ret(z,(fst u),(fst(snd u)),
              (t (fst(snd(snd u))) (snd(snd(snd u)))))}=
     do{u←do{x←p;y←q x;z←r x y;v←s x y z;ret(x,y,z,v)};
        z←ret(⊤);
        ret(z,(fst u),(fst(snd u)),
              (t (fst(snd(snd u))) (snd(snd(snd u)))))}"
    by (rule ret2seqSw)
  from this show ?thesis
    by (simp add: gdj_def)
qed
```

expample: assumes *[a←Φ;b←Ψ;x←p;c←ϒ x] ((r a b)⟶c)* shows *[d←do{a←Φ;b←Ψ;ret(r a b)};x←p;c←ϒ x] (d⟶c)*

```
lemma preOp:
  assumes "[x←p;y←q x;z←r;v←s z] Φ (t x y) z v"
  shows "[d←do{x←p;y←q x;ret(t x y)};z←r;v←s z] Φ d z v"

proof -
    from prems have
```

```
      "do{u←do{x←p;y←q x;z←r;v←s z;ret(x,y,z,v)};
          z←ret(Φ (t (fst u) (fst(snd u)))
                (fst(snd(snd u))) (snd(snd(snd u))));
          ret(z,u)}=
       do{u←do{x←p;y←q x;z←r;v←s z;ret(x,y,z,v)};
          z←ret(⊤);
          ret(z,u)}"
      by (simp add: gdj_def)
    from this have
      "do{u←do{x←p;y←q x;z←r;v←s z;ret(x,y,z,v)};
          z←ret(Φ (t (fst u) (fst(snd u)))
               (fst(snd(snd u))) (snd(snd(snd u))));
          ret(z,(t (fst u) (fst(snd u))),
               (fst(snd(snd u))),(snd(snd(snd u))))}=
       do{u←do{x←p;y←q x;z←r;v←s z;ret(x,y,z,v)};
          z←ret(⊤);
          ret(z,(t (fst u) (fst(snd u))),
               (fst(snd(snd u))),(snd(snd(snd u))))}"
      by (rule ret2seqSw)
  from this show ?thesis
    by (simp add: gdj_def)
qed

lemma pre:
  assumes "∀x. ([y←q x] Φ x y)"
  shows "[x←p;y←q x] Φ x y"

proof -
  have
    "∀x. ([y←q x] Φ x y) ⟹
     ∀x. (do {y←q x;ret (Φ x y,x,y)} = do {y←q x;ret (⊤,x,y)})"
    apply (rule allI)
    apply (rule gdj2doSeq)
    by simp
  from this prems have
    "(λx. do {y←q x;ret (Φ x y,x,y)}) =
     (λx. do {y←q x;ret (⊤,x,y)})"
    by simp
  from this show ?thesis
    by (simp add: gdj_def)
qed

lemma "pre_exp":
  assumes "∀u v. ([x←p u v] Φ u v x)"
  shows "[u←r;v←s u;x←p u v] Φ u v x"

proof -
  from prems have "∀u. ([v←s u;x←p u v] Φ u v x)"
    by (simp add: pre)
```

```
  from this have "
    ∀u. ([w≺-(v,x)←do{v←s u;x←p u v;ret(v,x)}] Φ u v x)"
    by simp
  from this have
    "([u←r;w≺-(v,x)←do{v←s u;x←p u v;ret(v,x)}] Φ u v x)"
    by (rule pre)
  from this show ?thesis
    by simp
qed

lemma sef0:
  assumes a: "[x←p;y←q x] Φ x" and b: "∀x. sef (q x)"
  shows   "[x←p] Φ x"

proof -
  from prems have "[u←do{x←p;y←q x;ret(x,y)}] Φ (fst u)"
    by (simp add: gdj_def)
  from this
  have "do{u←do{x←p;y←q x;ret(x,y)};ret(Φ (fst u),(fst u))} =
        do{u←do{x←p;y←q x;ret(x,y)};ret(⊤,(fst u))}"
    by (rule gdj2doSeq)
  from b this have
    "do{x←p;y←ret(Φ x,x);ret y} = do{x←p;y←ret(⊤,x);ret y}"
    by (simp add: seFree)
  from this show ?thesis
    by (simp add: gdj_def)
qed

lemma andI:
  assumes a:"[x←p] Φ x" and b:"[x←p] ξ x"
  shows "[x←p] (Φ x ∧ ξ x)"

proof -
  from a have
    "(do {x←p;ret (Φ x ∧ ξ x, x)}) = do {x←p;ret (⊤ ∧ ξ x, x)}"
    apply (unfold gdj_def)
    apply (rule gdj2doSeq)
    by (fold gdj_def)
  moreover
  from b have "... = do {x←p;ret (⊤ ∧ ⊤, x)}"
    apply (unfold gdj_def)
    apply (rule gdj2doSeq)
    by (fold gdj_def)
  moreover
  have "... = do {x←p;ret (⊤, x)}"
    by simp
  ultimately show ?thesis
    by (simp add: gdj_def)
qed
```

```
lemma andI_exp:
  assumes a:"[x←p;y←q x;z←r x y;v←s x y z] Φ x y z v" and
          b:"[x←p;y←q x;z←r x y;v←s x y z] ξ x y z v"
  shows
  "[x←p;y←q x;z←r x y;v←s x y z] ((Φ x y z v) ∧ (ξ x y z v))"

proof -
  from a have a':
    "[u←do{x←p;y←q x;z←r x y;v←s x y z;ret(x,y,z,v)}]
                    Φ (fst u) (fst (snd u)) (fst (snd (snd u)))
                      (snd (snd (snd u)))"
    by (simp add: gdj_def)
  from b have b':
    "[u←do{x←p;y←q x;z←r x y;v←s x y z;ret(x,y,z,v)}]
                    ξ (fst u) (fst (snd u)) (fst (snd (snd u)))
                      (snd (snd (snd u)))"
    by (simp add: gdj_def)
  have "[u←do{x←p;y←q x;z←r x y;v←s x y z;ret(x,y,z,v)}]
              ((Φ (fst u) (fst (snd u)) (fst (snd (snd u)))
                  (snd (snd (snd u))) ∧
               (ξ (fst u) (fst (snd u)) (fst (snd (snd u)))
                  (snd (snd (snd u))))))"
    proof -
      from a' have
        "do {u←do {x←p; y←q x; z←r x y; v←s x y z;
              ret (x, y, z, v)};
              ret (Φ (fst u) (fst (snd u)) (fst (snd (snd u)))
                     (snd (snd (snd u))) ∧
                   ξ (fst u) (fst (snd u)) (fst (snd (snd u)))
                     (snd (snd (snd u))),
             u)} =
        do {u←do {x←p; y←q x; z←r x y; v←s x y z;
              ret (x, y, z, v)};
              ret (True ∧
                   ξ (fst u) (fst (snd u)) (fst (snd (snd u)))
                     (snd (snd (snd u))),
             u)}"
        by (rule gdj2doSeq)
      moreover
      from b' have
        "... = do {u←do {x←p; y←q x; z←r x y; v←s x y z;
                 ret (x, y, z, v)};
                 ret (True ∧ True,u)}"
        by (rule gdj2doSeq)
      ultimately
      have
        "do {u←do {x←p;y←q x;z←r x y;v←s x y z;ret (x,y,z,v)};
                 ret (Φ (fst u) (fst (snd u)) (fst (snd (snd u)))
```

```
                        (snd (snd (snd u))) ∧
                      ξ (fst u) (fst (snd u)) (fst (snd (snd u)))
                        (snd (snd (snd u))),
             u)} =
         do {u←do {x←p;y←q x;z←r x y;v←s x y z; ret (x,y,z,v)};
                 ret (True,u)}"
        by simp
      from this show ?thesis
        by (simp add: gdj_def)
    qed
  from this show ?thesis
    by (simp add: gdj_def)
qed

lemma wk:
  assumes a: "∀x. (Φ x ⟶ ξ x)" and b: "[x←p] Φ x"
  shows  "[x←p] ξ x"

proof -
  have "do {x←p;ret (ξ x, x)} = do {x←p;ret (⊤ ∧ ξ x, x)}"
    by simp
  moreover
  from b have "... = do {x←p;ret (Φ x ∧ ξ x, x)}"
    proof -
      from b have
        "do {x←p;ret (Φ x ∧ ξ x, x)} = do {x←p;ret (⊤ ∧ ξ x, x)}"
        by (rule gdj2doSeq)
      from this show ?thesis ..
    qed
  moreover
  have "... = do {x←p;ret (Φ x, x)}"
  proof -
    from a have "∀x. (Φ x ∧ ξ x) = Φ x"
      by (simp add: allandI)
    from this show ?thesis
      by simp
  qed
  moreover
  from b have "... = do {x←p;ret (⊤, x)}"
    by (simp add: gdj_def)
  ultimately show ?thesis
    by (simp add: gdj_def)
qed

lemma wk_exp:
  assumes a: "∀x y z v. (Φ x y z v ⟶ ξ x y z v)" and
          b: "[x←p;y←q x;z←r x y;v←s x y z] Φ x y z v"
  shows  "[x←p;y←q x;z←r x y;v←s x y z] ξ x y z v"
```

```
proof -
  from a have "∀x y z v. (Φ x y z v ⟶ ξ x y z v)"
    by simp
  from this have
    "∀u. (Φ (fst u) (fst(snd u)) (fst(snd(snd u)))
            (snd(snd(snd u))) ⟶
         ξ (fst u) (fst(snd u)) (fst(snd(snd u)))
            (snd(snd(snd u))))"
    by simp
  moreover
  from b have
    "[u≺-(x,y,z,v)←do{x←p;y←q x;z←r x y;v←s x y z;ret(x,y,z,v)}]
                                                   (Φ x y z v)"
    by simp
  from this have
    "[u←do{x←p;y←q x;z←r x y;v←s x y z;ret(x,y,z,v)}]
      (Φ (fst u) (fst(snd u)) (fst(snd(snd u))) (snd(snd(snd u))))"
    by (simp add: gdj_def)
  ultimately have
    "[u←do{x←p;y←q x;z←r x y;v←s x y z;ret(x,y,z,v)}]
      (ξ (fst u) (fst(snd u)) (fst(snd(snd u))) (snd(snd(snd u))))"
    by (rule wk)
  from this have
    "[u≺-(x,y,z,v)←do{x←p;y←q x;z←r x y;v←s x y z;ret(x,y,z,v)}]
                                                   (ξ x y z v)"
    by (simp add: gdj_def)
  from this show ?thesis
    by simp
qed

lemma eq:
  assumes a: "[x←p](q₁ x = q₂ x)" and
          b:"[x←p; y←q₁ x; z←r x y] Φ x y z"
  shows "[x←p; y←q₂ x; z←r x y] Φ x y z"

proof -
  from a have
    "(do {x←p;y←q₂ x;z←r x y;ret (Φ x y z,x,y,z)}) =
     (do {x←p;u←(do{y←q₂ x;z←r x y;ret(y,z)});
                          ret (Φ x (fst u) (snd u),x,fst u,snd u)})"
    by simp
  moreover
  from a have "... = (do {x←p;u←(do{y←q₁ x;z←r x y;ret(y,z)});
                          ret (Φ x (fst u) (snd u),x,fst u,snd u)})"
    proof -
      from a have
        "(do {x←p;u←(do{y←q₁ x;z←r x y;ret(y,z)});
                          ret (Φ x (fst u) (snd u),x,fst u,snd u)})=
```

```
         (do {x←p;u←(do{y←q₂ x;z←r x y;ret(y,z)});
                         ret (Φ x (fst u) (snd u),x,fst u,snd u)})"
        by (rule gdjEq2doSeq)
      from this show ?thesis
        by simp
    qed
  moreover
  have"... = do {x←p;y←q₁ x;z←r x y;ret (Φ x y z,x,y,z)}"
    by simp
  moreover
  from b  have "... = do {x←p;y←q₁ x;z←r x y;ret (⊤,x,y,z)}"
    by (simp add: gdj_def)
  ultimately have
    "(do {x←p;y←q₂ x;z←r x y;ret (Φ x y z,x,y,z)}) =
      do {x←p;y←q₁ x;z←r x y;ret (⊤,x,y,z)}"
    by simp
  moreover have
    "... = (do {x←p;(y,z)←(do{y←q₁ x;z←r x y;ret(y,z)});
                                             ret (⊤,x,y,z)})"
    by simp
  ultimately
  have
    "(do {x←p;y←q₂ x;z←r x y;ret (Φ x y z,x,y,z)}) =
     (do {x←p;(y,z)←(do{y←q₁ x;z←r x y;ret(y,z)});ret (⊤,x,y,z)})"
    by (rule trans)
  moreover
  from a have
    "... = (do {x←p;(y,z)←(do{y←q₂ x;z←r x y;ret(y,z)});
                                             ret (⊤,x,y,z)})"
    by (rule gdjEq2doSeq)
  moreover
  have "... = do {x←p;y←q₂ x;z←r x y;ret (⊤,x,y,z)}"
    by simp
  ultimately have
    "(do {x←p;y←q₂ x;z←r x y;ret (Φ x y z,x,y,z)}) =
      do {x←p;y←q₂ x;z←r x y;ret (⊤,x,y,z)}"
    by simp
  from this show ?thesis
    by (simp add: gdj_def)
qed

lemma app:
  assumes a:"[x←p] Φ x"
  shows "[x←p;y←q x] Φ x"

proof -
  from a have
    "do {x←p; y←q x; ret (Φ x,x,y)} =
     do {x←p; y←q x; ret (⊤,x,y)}"
```

```
    by (rule gdj2doSeq)
  from this show ?thesis
    by (simp add: gdj_def)
qed

lemma app_exp:
  assumes a:"[x←p;y←q x] Φ x y"
  shows "[x←p;y←q x;z←r x y] Φ x y"

proof -
  from a have
    "[u←do{x←p;y←q x;ret(x,y)}] Φ (fst u) (snd u)"
    by (simp add: gdj_def)
  from this have
    "do {u←do{x←p; y←q x; ret(x,y)};
              z←r (fst u) (snd u);
              ret (Φ (fst u) (snd u),fst u,snd u,z)} =
     do {u←do{x←p; y←q x; ret(x,y)};
              z←r (fst u) (snd u);
              ret (⊤,fst u,snd u,z)}"
    by (rule gdj2doSeq)
  from this have
    "do {x←p; y←q x; z←r x y; ret (Φ x y,x,y,z)} =
    do {x←p; y←q x; z←r x y; ret (⊤,x,y,z)}"
    by simp
  from this show ?thesis
    by (simp add: gdj_def)
qed

lemma app_exp':
  assumes a:"[x←p;y←q x;z←r x y] Φ x y z"
  shows "[x←p;y←q x;z←r x y;v←t x y z] Φ x y z"

proof -
  from a have
    "[u←do{x←p;y←q x;z←r x y;ret(x,y,z)}]
                              Φ (fst u) (fst(snd u)) (snd(snd u))"
    by (simp add: gdj_def)
  from this have
    "do {u←do{x←p; y←q x; z←r x y;ret(x,y,z)};
              v←t (fst u) (fst(snd u)) (snd(snd u));
              ret (Φ (fst u) (fst(snd u)) (snd(snd u)),(fst u),
                  (fst(snd u)),(snd(snd u)),v)} =
     do {u←do{x←p; y←q x; z←r x y;ret(x,y,z)};
              v←t (fst u) (fst(snd u)) (snd(snd u));
              ret (⊤,(fst u),(fst(snd u)),(snd(snd u)),v)}"
    by (rule gdj2doSeq)
  from this have
    "do {x←p; y←q x; z←r x y;v←t x y z; ret (Φ x y z,x,y,z,v)} =
```

```
    do {x←p; y←q x; z←r x y;v←t x y z; ret (⊤,x,y,z,v)}"
    by simp
  from this show ?thesis
    by (simp add: gdj_def)
qed

lemma η:
  "[x←p; y←ret (a x); z←(q x y)] (Φ x y z) =
   [x←p; z←(q x (a x))] (Φ x (a x) z)"

proof
  assume "[x←p; y←ret (a x); z←(q x y)] (Φ x y z)"
  from this have
    "[u←(do{x←p; y←ret (a x); z←(q x y);ret(x,y,z)})]
                       (Φ (fst u) (a (fst u))) (snd (snd u))"
    by (simp add: gdj_def)
  from this
  have
    "do{u←do {x←p; y←ret (a x); z←q x y; ret (x,y,z)};
                ret(Φ (fst u) (a (fst u)) (snd (snd u)),fst u,
                    snd (snd u))} =
       do{u←do {x←p; y←ret (a x); z←q x y; ret (x,y,z)};
                ret(⊤,fst u,snd(snd u))}"
    by (rule gdj2doSeq)
  from this show "[x←p; z←(q x (a x))] (Φ x (a x) z)"
    by (simp add: gdj_def)
next
  assume "[x←p; z←(q x (a x))] (Φ x (a x) z)"
  from prems have
    "[u←do {x←p; z←q x (a x); ret (x, z)}]
                                    Φ (fst u) (a (fst u)) (snd u)"
    by (simp add: gdj_def)
  from this
  have
    "do{u←do {x←p; z←q x (a x); ret (x, z)};
        ret(Φ (fst u) (a (fst u)) (snd u),fst u,a (fst u),snd u)} =
     do{u←do {x←p; z←q x (a x); ret (x, z)};
        ret(⊤,fst u,a (fst u),snd u)}"
    by (rule gdj2doSeq)
  from this show "[x←p; y←ret (a x); z←(q x y)] (Φ x y z)"
    by (simp add: gdj_def)
qed

lemma η_cut: "[y←ret a; z←(q y)] (Φ y) = [z←(q a)] (Φ a)"

proof -
  have
    "[y←ret a; z←(q y)] (Φ y) = [u←ret();y←ret a; z←(q y)] (Φ y)"
```

```
    by (rule "**-")
  moreover
  have "[z←(q a)] (Φ a) = [u←ret();z←(q a)] (Φ a)"
    by (rule "*-")
  moreover
  have
    "[u←ret();y←ret a; z←(q y)] (Φ y) = [u←ret();z←(q a)] (Φ a)"
    by (rule η)
  ultimately show ?thesis
    by simp
qed

lemma ctr:
  assumes a: "[v←t;x←p v;y←q v x;z←r v y] (Φ v y z)"
  shows "[v←t;y←(do {x←p v;q v x});z←r v y] (Φ v y z)"

proof -
  from prems
  have "[u←do{v←t;x←p v;y←q v x;z←r v y;ret(v,x,y,z)}]
        (Φ (fst u) (fst (snd (snd u))) (snd (snd (snd u))))"
    by (simp add: gdj_def)
  from this have
    "do{u←do{v←t;x←p v;y←q v x;z←r v y;ret(v,x,y,z)};
           ret (Φ (fst u) (fst (snd (snd u))) (snd (snd (snd u))),
                (fst u), (fst (snd (snd u))), (snd (snd (snd u))))}=
     do{u←do{v←t;x←p v;y←q v x;z←r v y;ret(v,x,y,z)};
           ret (True,
                (fst u), (fst (snd (snd u))), (snd (snd (snd u))))}"
    by (rule gdj2doSeq)
  from this show ?thesis
    by (simp add: gdj_def)
qed

lemma ctr_cut:
  assumes a: "[x←p;y←q x;z←r y] (Φ y z)"
  shows "[y←(do {x←p;q x});z←r y] (Φ y z)"

proof -
  from a have "[v←ret();x←p;y←q x;z←r y] (Φ y z)"
    by (simp add: "***-")
  from this have "[v←ret();y←do{x←p;q x};z←r y] (Φ y z)"
    by (rule ctr)
  from this show ?thesis
    apply (subst "**-")
    by simp
qed

lemma ctr_cut2:
  assumes "[u←t;x←p u;y←q u x] (Φ u y)"
```

```
  shows "[u←t;y←(do {x←p u;q u x})] (Φ u y)"

proof -
  from prems have "[u←t;x←p u;y←q u x;z←ret ()] (Φ u y)"
    by (simp add: "-***")
  from this have "[u←t;y←(do {x←p u;q u x});z←ret()] (Φ u y)"
    by (rule ctr)
  from this show ?thesis
    apply (subst "-**")
    by simp
qed

lemma tau:
  assumes "∀x.(Φ x)"
  shows "[x←p] Φ x"

proof -
  from prems show ?thesis
    by (simp add: gdj_def)
qed

lemma tau_exp:
  assumes "∀x. ∀y.(Φ x y)"
  shows "[x←p;y←q] Φ x y"

proof -
  from prems show ?thesis
    by (simp add: tau pre)
qed

lemma rp:
  assumes a:"∀x.(q1 x = q2 x)" and
          b: "[x←p; y←q1 x; z←r x y] Φ x y z"
  shows "[x←p; y←q2 x; z←r x y] Φ x y z"

proof -
  from a have "[x←p] (q1 x = q2 x)"
    by (rule tau)
  from this b show "[x←p; y←q2 x; z←r x y] Φ x y z"
    by (rule eq)
qed

lemma rp_cut:
  assumes a: "∀x.((q1 x) = (q2 x))" and b:"[x←p; y←q1 x] Φ x y"
  shows "[x←p; y←q2 x] Φ x y"

proof -
  from b have "[x←p; y←q1 x;z←ret()] Φ x y"
    by (simp add: "-**")
```

```
  from a this have  "[x←p; y←q₂ x;z←ret()] Φ x y"
    by (rule rp)
  from this show ?thesis
    by (simp add: "-**")
qed

lemma sef:
  assumes a: "[x←p;y←q x;z←r x] Φ x z" and b: "∀x. sef (q x)"
  shows "[x←p;z←r x] Φ x z"

proof -
  from b have "∀x. ((do{y←q x;r x}) = r x)"
    by (simp add: seFree)
  moreover
  from a
  have "[x←p;z←do{y←q x;r x}] Φ x z"
    by (rule ctr_cut2)
  ultimately show ?thesis
    by (rule rp_cut)
qed

end
```

A.5. dsefSyntax

```
theory dsefSyntax = gdjSyntax + Lemmabase:
```

Seqences fulfilling all three (sef, cp, com) properties are called deterministic sideeffect free

```
constdefs
  "dsef" :: "'a T ⇒ bool"
  "dsef p == (sef p) ∧ (cp p) ∧ (∀q. (p comwith q))"
```

Introducing the subtype *('a D)* of *('a T)* comprising the dsef programs; since Isabelle lacks true subtyping, it is simply declared as a new type with coercion functions *Rep_Dsef :: 'a D ⇒ 'a T* and *Abs_Dsef :: 'a T ⇒ 'a D* where (Abs_Dsef p) is of course only sensibly defined if (dsef p) holds.

```
typedef (Dsef) ('a) D = "{p::'a T. dsef p}"

  apply(rule_tac x = "ret x" in exI)
  apply (unfold dsef_def)
  apply (simp add: cp_def)
  apply (simp add: sef_def com_def del:delBind)
  apply (rule allI)
  apply (simp add: com_def)
  by (simp add: weak_cp2retSeq)
```

```
syntax
  Rep :: "'a D ⇒ 'a T"  ("↓_")
  Abs :: "'a T ⇒ 'a D" ("↑_")
  Ret :: "'a ⇒ 'a D" ("⇑_")

translations
  "↓p" == "Rep_Dsef p"
  "↑p" == "Abs_Dsef p"
  "⇑p" == "↑(ret p)"

lemma avoidAbsRep [simp]: "Abs_Dsef (Rep_Dsef p) = p"
  by (rule Rep_Dsef_inverse)

lemma avoidRepAbs [simp]:"p ∈ Dsef ⟹ Rep_Dsef (Abs_Dsef p) = p"
  by (rule Abs_Dsef_inverse)

lemma avoidAbsRep' [simp]:"↑↓(p::'a D) == p"
  by (simp add: Rep_Dsef_inverse)

lemma avoidRepAbs' [simp]:"p ∈ Dsef ⟹ (↓↑(p::'a T)) == p"
  by (simp add: Abs_Dsef_inverse)


constdefs
  if_D :: "bool D ⇒ 'a T ⇒ 'a T ⇒ 'a T"   ("if_D (_) then (_) else (_)")
  "if_D b then p else q == do{x←↓b;if x then p else q}"

consts
  iter_D:: "('a ⇒ bool D) ⇒ ('a ⇒ 'a T) ⇒ 'a ⇒ 'a T"
  while_D :: "bool D ⇒ unit T ⇒ unit T" ("while_D (_) (_)")

axioms
  iter_D_def: "iter_D test f x =
       do{if_D (test x) then (do{y←(f x);iter_D test f y}) else (ret
x)}"

  while_D_def: "while_D b p == iter_D (λx. b) (λx. p) ()"



constdefs
  condConj :: "bool D ⇒ bool D ⇒ bool D"   ("(_∧_D _)")
  "condConj Φ ξ ≡ ↑do{x←↓Φ;y←↓ξ;ret(x∧y)}"
  condDisj :: "bool D ⇒ bool D ⇒ bool D"   ("(_∨_D _)")
  "condDisj Φ ξ ≡ ↑do{x←↓Φ;y←↓ξ;ret(x∨y)}"
  condNot :: "bool D ⇒ bool D"             ("¬_D_")
```

```
"condNot Φ ≡ ↑do{x←↓Φ;ret(¬x)}"
```

end

A.6. dsefCalc

```
theory dsefCalc = dsefSyntax + gdjCalc:

lemma weak_cp2seq:
  assumes dsef_r: "∀x. dsef (r x)" and
          cp_p:"cp p" and sef_p: "sef p"
  shows "cp (do {x←p; r x})"
proof -
  from dsef_r have cp_r:  "∀x. cp (r x)"
    by (simp add: dsef_def)
  from dsef_r have com_r: "∀x. (∀q. (r x) comwith q)"
    by (simp add: dsef_def)
  from dsef_r have sef_r: "∀x. (sef (r x))"
    by (simp add: dsef_def)

  have "do {u←do {x←p; r x}; v←do {y←p; r y}; ret (u,v)} =
    do {x←p; z←do{u←r x;y←p;ret(u,y)}; v←r (snd z); ret ((fst z),v)}"
    by simp
  moreover
  from com_r sef_r cp_p sef_p have
    "... = do {x←p; z←do{y←p;u←r x;ret(u,y)}; v←r (snd z); ret ((fst
z),v)}"
    by (simp add: switch3)
  moreover
  have
    "... = do {x←p; y←p; u←r x; v←r y; ret (u,v)}"
    by simp
  moreover from cp_p cp_r have
    "... = do {u←p; x←r u; ret (x,x)}"
    by (simp only: cp_ret2seq)
  ultimately show ?thesis by (simp add: cp_def)
qed

lemma weak_com2seq:
  assumes dsef_r:"∀x. dsef (r x)" and dsef_p:"dsef p"
  shows "∀q::bool T. (cp q ∧ sef q) ⟶ cp (do {x←do{x←p;r x}; y←q;
ret(x,y)})"
proof -
  from dsef_r have cp_r: "∀x. cp (r x)"
    by (simp add: dsef_def)
  from dsef_p have cp_p: "cp p"
```

```
    by (simp add: dsef_def)

  have
    "∀q::bool T. (cp q ∧ sef q) ⟶
     do {u←do{x←do{x←p;r x}; y←q;ret(x,y)};
         v←do{x←do{x←p;r x}; y←q; ret(x,y)};ret(u,v)}=
     do {x₁←p;y₁←r x₁;z₁←q;
         x₂←p;y₂←r x₂;z₂←q;ret((y₁,z₁),(y₂,z₂))}"
    by simp
  from this have
    "∀q::bool T. (cp q ∧ sef q) ⟶
     do {u←do{x←do{x←p;r x}; y←q;ret(x,y)};
         v←do{x←do{x←p;r x}; y←q; ret(x,y)};ret(u,v)}=
     do {x₁←p;y₁←r x₁;u←do{z₁←q;
         x₂←p;ret(z₁,x₂)};y₂←r (snd u);z₂←q;ret((y₁,fst u),(y₂,z₂))}"
    by simp
  from this dsef_p have
    "∀q::bool T. (cp q ∧ sef q) ⟶
     do {x₁←p;y₁←r x₁;z₁←q;
         x₂←p;y₂←r x₂;z₂←q;ret((y₁,z₁),(y₂,z₂))} =
     do {x₁←p;u←do{y₁←r x₁;
         x₂←p;ret(y₁,x₂)};z₁←q;y₂←r (snd u);z₂←q;ret((fst u,z₁),(y₂,z₂))}"
    apply (unfold dsef_def)
    by (simp add: switch2)
  moreover from dsef_r dsef_p have
    "∀q::bool T. (cp q ∧ sef q) ⟶
     do {x₁←p;u←do{y₁←r x₁;
         x₂←p;ret(y₁,x₂)};z₁←q;y₂←r (snd u);z₂←q;ret((fst u,z₁),(y₂,z₂))}=
     do {x₁←p;x₂←p;do{y₁←r x₁;z₁←q;y₂←r x₂;z₂←q;ret((y₁,z₁),(y₂,z₂))}}"
    by (simp add: dsef_def switch2)
  ultimately have
    "∀q::bool T. (cp q ∧ sef q) ⟶
     do {x₁←p;y₁←r x₁;z₁←q;
         x₂←p;y₂←r x₂;z₂←q;ret((y₁,z₁),(y₂,z₂))} =
     do {x₁←p;x₂←p;do{y₁←r x₁;z₁←q;y₂←r x₂;z₂←q;ret((y₁,z₁),(y₂,z₂))}}"
    by simp
  from this cp_p have
    "∀q::bool T. (cp q ∧ sef q) ⟶
     do {x₁←p;y₁←r x₁;z₁←q;
         x₂←p;y₂←r x₂;z₂←q;ret((y₁,z₁),(y₂,z₂))} =
     do {x₁←p;y₁←r x₁;u←do{z₁←q;y₂←r x₁;ret(z₁,y₂)};z₂←q;ret((y₁,fst
u),(snd u,z₂))}"
    by (simp add: cp_ret2seq)
  moreover from dsef_r have
    "∀q::bool T. (cp q ∧ sef q) ⟶
     do {x₁←p;y₁←r x₁;u←do{z₁←q;y₂←r x₁;ret(z₁,y₂)};z₂←q;ret((y₁,fst
u),(snd u,z₂))} =
     do {x₁←p;y₁←r x₁;y₂←r x₁;z₁←q;z₂←q;ret((y₁,z₁),(y₂,z₂))}"
    by (simp add: dsef_def switch2)
```

```
  ultimately have
    "∀q::bool T. (cp q ∧ sef q) ⟶
     do {x₁←p;y₁←r x₁;z₁←q;
         x₂←p;y₂←r x₂;z₂←q;ret((y₁,z₁),(y₂,z₂))} =
     do {x₁←p;y₁←r x₁;y₂←r x₁;z₁←q;z₂←q;ret((y₁,z₁),(y₂,z₂))}"
    by simp
  from this cp_r have
    "∀q::bool T. (cp q ∧ sef q) ⟶
     do {x₁←p;y₁←r x₁;z₁←q;
         x₂←p;y₂←r x₂;z₂←q;ret((y₁,z₁),(y₂,z₂))} =
     do {x₁←p;y₁←r x₁;z₁←q;ret((y₁,z₁),(y₁,z₁))}"
    by (simp add: cp_ret2seq)
  from this show ?thesis
    by (simp add: cp_def)
qed

lemma dsef_ret: "dsef (ret x)"

proof (unfold dsef_def)
  have "cp (ret x)" by (simp add: cp_def)
  moreover have "sef (ret x)"
    by (simp add: sef_def com_def del:delBind)
  moreover have
    "(∀q. (cp q) ∧ (sef q) ⟶
          (cp (do{x←ret x; y←q; ret (x, y)})))"
    by (simp add: weak_cp2retSeq)
  from this have "∀q. ((ret x) comwith q)"
    by (simp add: com_def)
  ultimately show
    "(sef (ret x)) ∧ (cp (ret x)) ∧ (∀q. ((ret x) comwith q))"
    by blast
qed


lemma sef_rm2of3:
  assumes "sef Φ"
  shows "[a←Φ';b←Φ;c←Ψ a](ξ a c) ⟹ [a←Φ';c←Ψ a](ξ a c)"
proof -
  assume "[a←Φ';b←Φ;c←Ψ a](ξ a c)"
  from this have
    "[u←do{a←Φ';b←Φ;c←Ψ a;ret(a,b,c)}](ξ (fst u) (snd(snd u)))"
    by (simp add: gdj_def)
  from this have
    "do{u←do{a←Φ';b←Φ;c←Ψ a;ret(a,b,c)};
              ret(ξ (fst u) (snd(snd u)),
              (fst u),(snd(snd u)))} =
     do{u←do{a←Φ';b←Φ;c←Ψ a;ret(a,b,c)};
              ret(True,
              (fst u),(snd(snd u)))}"
```

```
    by (rule gdj2doSeq)
  from this have
    "do{a←Φ';b←Φ;c←Ψ a;ret(ξ a c,a,c)} =
     do{a←Φ';b←Φ;c←Ψ a;ret(True,a,c)}"
    by (simp add: gdj_def)
  from this have
    "do{a←Φ';c←do{b←Φ;Ψ a};ret(ξ a c,a,c)} =
     do{a←Φ';c←do{b←Φ;Ψ a};ret(True,a,c)}"
    by simp
  moreover from prems have "sef Φ"
    by simp
  ultimately have
    "do{a←Φ';c←Ψ a;ret(ξ a c,a,c)} =
     do{a←Φ';c←Ψ a;ret(True,a,c)}"
    by (simp add: seFree)
  from this
  show ?thesis
    by (simp add: gdj_def)
qed


lemma sef_rm2of4:
  assumes "sef Φ"
  shows "[a←Φ';b←Φ;x←p;c←Ψ x](ξ a x c) ⟹
         [a←Φ';x←p;c←Ψ x](ξ a x c)"

proof -
  assume "[a←Φ';b←Φ;x←p;c←Ψ x](ξ a x c)"
  from this have
    "[a←Φ';b←Φ;u≺-(x,c)←do{x←p;c←Ψ x;ret(x,c)}](ξ a x c)"
    by simp
  from prems this
  have "[a←Φ';u≺-(x,c)←do{x←p;c←Ψ x;ret(x,c)}](ξ a x c)"
    by (simp only: sef_rm2of3)
  moreover
  from this have "[a←Φ';x←p;c←Ψ x](ξ a x c)"
    by (simp add: ctr_cut2)
  ultimately show ?thesis
    by simp
qed


lemma sef_rm3of4:
  assumes "∀x. sef (Ψ x)"
  shows "[a←Φ';x←p;c←Ψ x;d←Ψ' x](ξ a x d) ⟹
         [a←Φ';x←p;d←Ψ' x](ξ a x d)"

proof -
  assume "[a←Φ';x←p;c←Ψ x;d←Ψ' x](ξ a x d)"
```

```
  from this have
    "[u←do{a←Φ';x←p;c←Ψ x;d←Ψ' x;ret(a,x,c,d)}]
                        (ξ (fst u) (fst(snd u)) (snd(snd(snd u))))"
    by (simp add: gdj_def)
  from this have
    "do{u←do{a←Φ';x←p;c←Ψ x;d←Ψ' x;ret(a,x,c,d)};
              ret(ξ (fst u) (fst(snd u)) (snd(snd(snd u))),
              (fst u),(fst(snd u)),(snd(snd(snd u))))} =
     do{u←do{a←Φ';x←p;c←Ψ x;d←Ψ' x;ret(a,x,c,d)};
              ret(True,
              (fst u),(fst(snd u)),(snd(snd(snd u))))}"
    by (rule gdj2doSeq)
  from this have
    "do{a←Φ';x←p;c←Ψ x;d←Ψ' x; ret(ξ a x d,a,x, d)} =
     do{a←Φ';x←p;c←Ψ x;d←Ψ' x; ret(True,a,x, d)}"
    by (simp add: gdj_def)
  from this have
    "do{a←Φ';x←p;d←do{c←Ψ x;Ψ' x}; ret(ξ a x d,a,x, d)} =
     do{a←Φ';x←p;d←do{c←Ψ x;Ψ' x}; ret(True,a,x, d)}"
    by simp
  moreover from prems have "∀x. sef (Ψ x)"
    by simp
  ultimately
  have
    "do{a←Φ';x←p;d←Ψ' x; ret(ξ a x d,a,x, d)} =
     do{a←Φ';x←p;d←Ψ' x; ret(True,a,x, d)}"
    by (simp add: seFree)
  from this show ?thesis
    by (simp add: gdj_def)
qed


lemma sef_rm3of5:
  assumes "∀x. sef (Ψ x)"
  shows "[a←Φ;x←p;b←Ψ x;y←q x;c←ϒ x y]ξ a x c y ⟹
         [a←Φ;x←p;y←q x;c←ϒ x y]ξ a x c y"
proof -
  assume "[a←Φ;x←p;b←Ψ x;y←q x;c←ϒ x y]ξ a x c y"
  from this
  have
    "[a←Φ;x←p;b←Ψ x;u←do{y←q x;c←ϒ x y;ret(y,c)}]ξ a x (snd u)
(fst u)"
    by (simp add: gdj_def)
  moreover from this prems have
    "[a←Φ;x←p;u←do{y←q x;c←ϒ x y;ret(y,c)}]ξ a x (snd u) (fst u)"
    by (simp only: sef_rm3of4)
  moreover from this have
    "[a←Φ;x←p;y←q x;c←ϒ x y]ξ a x c y"
    by (simp add: gdj_def)
```

```
  ultimately show ?thesis
    by (simp add: gdj_def)
qed

lemma dsef_switch:
  assumes dsef_a: "dsef a" and dsef_b: "dsef b"
  shows "([x←a;y←b;z←p;v←r z] Φ x y z v) =
          ([y←b;x←a;z←p;v←r z] Φ x y z v)"

proof
  assume "([x←a;y←b;z←p;v←r z] Φ x y z v)"
  from this have
    "([(x,y)←do{x←a;y←b;ret(x,y)};z←p;v←r z] Φ x y z v)"
    by simp
  from dsef_a dsef_b this have
    "([(x,y)←do{y←b;x←a;ret(x,y)};z←p;v←r z] Φ x y z v)"
    by (simp add: dsef_def switch)
  from this have
    "[u←do{(x,y)←do{y←b;x←a;ret(x,y)};z←p;v←r z;ret(x,y,z,v)}]
       Φ (fst u) (fst(snd u)) (fst(snd(snd u))) (snd(snd(snd u)))"
    by (simp add: gdj_def)
  from this have
   "do{u←do{(x,y)←do{y←b;x←a;ret(x,y)};z←p;v←r z;ret(x,y,z,v)};
              ret(Φ (fst u) (fst(snd u)) (fst(snd(snd u)))
                  (snd(snd(snd u))),
                  (fst(snd u)),(fst u),(fst(snd(snd u))),
                  (snd(snd(snd u))))} =
   do{u←do{(x,y)←do{y←b;x←a;ret(x,y)};z←p;v←r z;ret(x,y,z,v)};
              ret(True     ,
                  (fst(snd u)),(fst u),(fst(snd(snd u))),
                  (snd(snd(snd u))))}"
    by (rule gdj2doSeq)
  from this have
    "do{(x,y)←do{y←b;x←a;ret(x,y)};z←p;v←r z;
                               ret(Φ x y z v,y,x,z,v)} =
     do{(x,y)←do{y←b;x←a;ret(x,y)};z←p;v←r z;
                               ret(True     ,y,x,z,v)}"
    by simp
  from this show "([y←b;x←a;z←p;v←r z] Φ x y z v)"
    by (simp add: gdj_def)
next
  assume "([y←b;x←a;z←p;v←r z] Φ x y z v)"
  from this have
    "([(y,x)←do{y←b;x←a;ret(y,x)};z←p;v←r z] Φ x y z v)"
    by simp
  from dsef_a dsef_b this have
    "([(y,x)←do{x←a;y←b;ret(y,x)};z←p;v←r z] Φ x y z v)"
    by (simp add: dsef_def switch)
  from this have
```

```
    "[u←do{(y,x)←do{x←a;y←b;ret(y,x)};z←p;v←r z;ret(y,x,z,v)}]
       Φ (fst(snd u)) (fst u) (fst(snd(snd u))) (snd(snd(snd u)))"
    by (simp add: gdj_def)
  from this have
   "do{u←do{(y,x)←do{x←a;y←b;ret(y,x)};z←p;v←r z;ret(y,x,z,v)};
             ret(Φ (fst(snd u)) (fst u) (fst(snd(snd u)))
             (snd(snd(snd u))),(fst(snd u)),(fst u),
             (fst(snd(snd u))),(snd(snd(snd u))))} =
    do{u←do{(y,x)←do{x←a;y←b;ret(y,x)};z←p;v←r z;ret(y,x,z,v)};
             ret(True    ,
             (fst(snd u)),(fst u),(fst(snd(snd u))),
             (snd(snd(snd u))))}"
    by (rule gdj2doSeq)
  from this have
    "do{x←a;y←b;z←p;v←r z;ret(Φ x y z v,x,y,z,v)} =
     do{x←a;y←b;z←p;v←r z;ret(True,x,y,z,v)}"
    by simp
  from this show "([x←a;y←b;z←p;v←r z] Φ x y z v)"
    by (simp add: gdj_def)
qed


lemma dsef_switch':
  assumes dsef_a: "∀y. dsef (a y)" and dsef_b: "∀y. dsef (b y)"
  shows "[x←p;y←q;z←a y;v←b y] Φ x y z v =
         [x←p;y←q;v←b y;z←a y] Φ x y z v"

proof
  assume "[x←p;y←q;z←a y;v←b y] Φ x y z v"
  from this have
    "[x←p;y←q;(z,v)←do{z←a y;v←b y;ret(z,v)}] Φ x y z v"
    by simp
  from dsef_a dsef_b this have
    "[x←p;y←q;(z,v)←do{v←b y;z←a y;ret(z,v)}] Φ x y z v"
    by (simp add: dsef_def switch')
  from this have
    "[u←do{x←p;y←q;(z,v)←do{v←b y;z←a y;ret(z,v)};ret(x,y,z,v)}]
      Φ (fst u) (fst(snd u)) (fst(snd(snd u))) (snd(snd(snd u))) "
    by (simp add: gdj_def)
  from this have
    "do {u←do{x←p;y←q;(z,v)←do{v←b y;z←a y;ret(z,v)};
            ret(x,y,z,v)};
            ret(Φ (fst u) (fst(snd u)) (fst(snd(snd u)))
               (snd(snd(snd u))),(fst u),(fst(snd u)),
               (snd(snd(snd u))),(fst(snd(snd u))))} =
    do {u←do{x←p;y←q;(z,v)←do{v←b y;z←a y;ret(z,v)};
            ret(x,y,z,v)};
            ret(True,
               (fst u),(fst(snd u)),(snd(snd(snd u))),
```

```
               (fst(snd(snd u))))}"
    by (rule gdj2doSeq)
  from this show "[x←p;y←q;v←b y;z←a y] Φ x y z v"
    by (simp add: gdj_def)
next
  assume "[x←p;y←q;v←b y;z←a y] Φ x y z v"
  from this have
    "[x←p;y←q;(v,z)←do{v←b y;z←a y;ret(v,z)}] Φ x y z v"
    by simp
  from dsef_a dsef_b this have
    "[x←p;y←q;(v,z)←do{z←a y;v←b y;ret(v,z)}] Φ x y z v"
    by (simp add: dsef_def switch')
  from this have
    "[u←do{x←p;y←q;(v,z)←do{z←a y;v←b y;ret(v,z)};ret(x,y,v,z)}]
      Φ (fst u) (fst(snd u)) (snd(snd(snd u))) (fst(snd(snd u)))"
    by (simp add: gdj_def)
  from this have
    "do {u←do{x←p;y←q;(v,z)←do{z←a y;v←b y;ret(v,z)};
            ret(x,y,v,z)};
            ret(Φ (fst u) (fst(snd u)) (snd(snd(snd u)))
                (fst(snd(snd u))),(fst u),(fst(snd u)),
                (snd(snd(snd u))),(fst(snd(snd u))))} =
     do {u←do{x←p;y←q;(v,z)←do{z←a y;v←b y;ret(v,z)};
            ret(x,y,v,z)};
            ret(True,
                (fst u),(fst(snd u)),(snd(snd(snd u))),
                (fst(snd(snd u))))}"
    by (rule gdj2doSeq)
  from this show "[x←p;y←q;z←a y;v←b y] Φ x y z v"
    by (simp add: gdj_def)
qed


lemma weak_dsef2seq:
  assumes dsef_p: "dsef p" and dsef_r: "∀x. dsef (r x)"
  shows "dsef (do{x←p;r x})"

proof -

  from dsef_r dsef_p have cp_do: "cp (do{x←p;r x})"
    by (simp add: dsef_def weak_cp2seq)
  from sef_p sef_r have sef_do: "sef (do{x←p;r x})"
    by (simp add: sef_def)
  have com_do:
    "∀q::bool T. ((cp q ∧ sef q) ⟶ do{x←p;r x} comwith q)"
    proof -
      from dsef_r dsef_p have com_do:
        "∀q::bool T. ((cp q ∧ sef q) ⟶ cp (do{x←do{x←p;r x}; y←q;
ret (x, y)}))"
```

```
        by (rule weak_com2seq)
      from this show ?thesis
        by (simp add: com_def)
    qed
  from cp_do sef_do com_do show "dsef (do{x←p;r x})"
    by (simp add: dsef_def com_def)
qed


lemma weak_dsef2seq_exp:
  assumes dsef_p: "dsef p" and
          dsef_q: "dsef q" and
          dsef_r: "∀x y. dsef (r x y)"
  shows "dsef (do{x←p;y←q;r x y})"

proof -
  from dsef_q dsef_r have
    "∀x. (∀y. dsef (r x y) ⟶ dsef(do{y←q;r x y}))"
    by (simp add: weak_dsef2seq)
  from dsef_r this have "∀x. dsef(do{y←q;r x y})"
    by simp
  from dsef_p this show ?thesis
    by (simp add: weak_dsef2seq)
qed


lemma weak_Dsef2seq:
  assumes Dsef_p: "p ∈ Dsef" and Dsef_r: "∀x. (r x) ∈ Dsef"
  shows "(do{x←p;r x}) ∈ Dsef"

proof -
  from Dsef_p have dsef_p: "dsef p"
    by (simp add: Dsef_def)
  from Dsef_r have dsef_r: "∀x. dsef (r x)"
    by (simp add: Dsef_def)
  from dsef_p dsef_r have "dsef (do{x←p;r x})"
    by (simp add: weak_dsef2seq)
  from this show ?thesis
    by (simp add: Dsef_def)
qed


lemma weak_Dsef2seq_exp:
  assumes Dsef_p: "p ∈ Dsef" and
          Dsef_q: "q ∈ Dsef" and
          Dsef_r: "∀x y. (r x y) ∈ Dsef"
  shows "(do{x←p;y←q;r x y}) ∈ Dsef"

proof -
```

```
  from Dsef_p have dsef_p: "dsef p"
    by (simp add: Dsef_def)
  from Dsef_q have dsef_q: "dsef q"
    by (simp add: Dsef_def)
  from Dsef_r have dsef_r: "∀x y. dsef (r x y)"
    by (simp add: Dsef_def)
  from dsef_p dsef_q dsef_r have "dsef (do{x←p;y←q;r x y})"
    by (simp add: weak_dsef2seq_exp)
  from this show ?thesis
    by (simp add: Dsef_def)
qed


lemma double:
  assumes dsef_Φ: "dsef Φ" and dsef_Ψ: "dsef Ψ" and
          a: "[(a,b)←do{a←Φ;b←Ψ;ret(Q a b,P a b a b)}](a⟶b)"
  shows "[(a,b)←do{a←Φ;b←Ψ;c←Φ;d←Ψ;ret(Q a b,P a b c d)}](a⟶b)"
proof -
  have "∀a b. dsef (ret(a,b))"
    by (simp add: dsef_ret)
  from dsef_Φ  dsef_Ψ this have "dsef (do{a←Φ;b←Ψ;ret(a,b)})"
    by (rule weak_dsef2seq_exp)
  from this have "cp (do{a←Φ;b←Ψ;ret(a,b)})"
    by (simp add: dsef_def)
  moreover
  from a have
    "[(a,b)←do{u←do{a←Φ;b←Ψ;ret(a,b)};
          ret(Q (fst u) (snd u),P (fst u) (snd u) (fst u) (snd u))}](a⟶b)"
    by (simp add: gdj_def)
  ultimately have
    "[(a,b)←do{u←do{a←Φ;b←Ψ;ret(a,b)};v←do{a←Φ;b←Ψ;ret(a,b)};
          ret(Q (fst u) (snd u),P (fst u) (snd u) (fst v) (snd v))}](a⟶b)"
    by (simp only: cp_ret2seq)
  from this show ?thesis
    by simp
qed

lemma double2:
  assumes dsef_Φ: "dsef Φ" and dsef_Ψ: "dsef Ψ" and
          a: "[(a,b,b)←do{a←Φ;b←Ψ;ret(Q a b,b,P a b a b)}](a⟶b)"
  shows "[(a,b,b)←do{a←Φ;b←Ψ;c←Φ;d←Ψ;ret(Q a b,b,P a b c d)}](a⟶b)"
proof -
  have "∀a b. dsef (ret(a,b))"
    by (simp add: dsef_ret)
  from dsef_Φ  dsef_Ψ this have "dsef (do{a←Φ;b←Ψ;ret(a,b)})"
    by (rule weak_dsef2seq_exp)
  from this have "cp (do{a←Φ;b←Ψ;ret(a,b)})"
    by (simp add: dsef_def)
  moreover
```

```
  from a have
    "[(a,x,b)←do{u←do{a←Φ;b←Ψ;ret(a,b)};
          ret(Q (fst u) (snd u),(snd u),P (fst u) (snd u) (fst u) (snd
u))}](a⟶b)"
    by (simp add: gdj_def)
  ultimately have
    "[(a,x,b)←do{u←do{a←Φ;b←Ψ;ret(a,b)};v←do{a←Φ;b←Ψ;ret(a,b)};
          ret(Q (fst u) (snd u),(snd u),P (fst u) (snd u) (fst v) (snd
v))}](a⟶b)"
    by (simp only: cp_ret2seq)
  from this show ?thesis
    by simp
qed

end
```

A.7. HoareSyntax

```
theory HoareSyntax = dsefSyntax:



constdefs
  "hoare":: "bool D ⇒ 'a T ⇒ ('a ⇒ bool D) ⇒ bool"
  "hoare Φ p Ψ == [a←↓Φ;x←p;b←↓(Ψ x)] (a ⟶ b)"

nonterminals
  "hseq" "hstep" "cond"

syntax
  "_hoare"  :: "bool D ⇒ hseq ⇒ bool D ⇒ bool" ("⦃_⦄_⦃_⦄")

  "_hbind"  :: "[pttrn, 'a T] ⇒ hstep" ("_←_")
  "_hseq"   :: "[hstep, hseq] ⇒ hseq"  ("_;_")
  "_hsingle":: "idt ⇒ hstep"  ("_")
  "_hstep"  :: "hstep ⇒ hseq"  ("_")

  "_hIn"   :: "hseq ⇒ hseq"
  "_hIn'"  :: "[pttrn, hseq] ⇒ hseq"
  "_hOut"  :: "[pttrn, hseq] ⇒ hseq"
  "_hOut'" :: "[pttrn, hseq] ⇒ hseq"

  "_tpl" :: "[pttrn, pttrn] ⇒ pttrn"

translations
  "_hoare Φ (_hstep (_hsingle p)) Ψ"  => "hoare Φ p (_K Ψ)"
  "_hoare Φ (_hstep (_hsingle p)) Ψ"  <= "hoare Φ p (λx. Ψ)"
```

```
"_hoare Φ (_hstep (_hbind x p)) Ψ"  == "hoare Φ p (λx. Ψ)"
"_hoare Φ (_hseq p q) Ψ" ==  "_hoare Φ (_hIn (_hseq p q)) Ψ"

"_hoare Φ (_hOut r) Ψ" =>  "hoare Φ r (_K Ψ)"
"_hoare Φ (_hOut r) Ψ" <=  "hoare Φ r (λx. Ψ)"
"_hoare Φ (_hOut' tpl r) Ψ" ==  "hoare Φ r (λtpl. Ψ)"

"_hIn (_hstep (_hsingle p))" => "_hOut p"
"_hIn (_hstep (_hbind x p))" => "_hOut' x p"
"_hIn (_hseq (_hsingle p) q)" => "_hseq (_hsingle p) (_hIn' q)"
"_hIn (_hseq (_hbind x p) q)" => "_hseq (_hbind x p) (_hIn' x q)"

"_hIn' tpl (_hstep (_hsingle p))" =>
          "_hOut' (_tpl tpl) (do{p;ret (_tpl tpl)})"
"_hIn' tpl (_hstep (_hbind x p))" =>
          "_hOut' (_tpl (tpl,x)) (do{x←p;ret (_tpl (tpl,x))})"
"_hIn' tpl (_hseq (_hsingle p) q)" =>
          "_hseq (_hsingle p) (_hIn' tpl q)"
"_hIn' tpl (_hseq (_hbind x p) q)" =>
          "_hseq (_hbind x p) (_hIn' (tpl,x) q)"

"_hseq (_hsingle p) (_hOut q)" =>
          "_hOut (p ≫ q)"

"_hseq (_hsingle p) (_hOut' tpl q)" =>
          "_hOut' tpl (p ≫ q)"
"_hseq (_hbind x p) (_hOut' tpl q)" =>
          "_hOut' tpl (p ≫= (λx. q))"

"_tpl (Pair (Pair x y) z)" => "_tpl (Pair x (Pair y z))"
"_tpl (Pair x y)" => "(Pair x y)"
```

Somtimes we need Hoare-triples without a programm-sequence. For this purpose we have three posible versions 1. $\{\Phi\}\{a{:}\Psi\}$ - equvivalent to normal Hoare-Triples 2./3. $\{\Phi\}\{\Psi\}$ and $\Phi \Rightarrow_T \Psi$ - as syntactic sugar for version 1

constdefs

```
"hoare_Tupel" :: "bool D ⇒ bool D ⇒ bool"  ("{|_|} {|_|}")
"hoare_Tupel Φ Ψ == [(a,b)←do{a←↓Φ;b←↓Ψ;ret(a,b)}] (a⟶b)"
```

syntax

```
"hoare2" :: "bool D ⇒ bool D ⇒ bool"  ("(_) ⇒h (_)")
```

translations

```
"hoare2 Φ Ψ" == "hoare_Tupel Φ Ψ"
```

end

A.8. HoareCalc

```
theory HoareCalc = HoareSyntax + dsefCalc:

lemma "dsef" :
  assumes dsef_p: "dsef p"
  shows "⦃↑(ret True)⦄ x←p ⦃↑(do{y←p;ret(x=y)})⦄"

  proof -
  from dsef_p have "sef p"
    by (simp add: dsef_def)
  moreover
  from dsef_p have "cp p"
    by (simp add: dsef_def)
  ultimately
  have "[x←p;y←p](x=y)"
    by (simp add: sef2cp)
  from this
  have "[u←do {x←p; y←p; ret (x, y)}](fst u = snd u)"
    by (simp add: gdj_def)
  from this
  have "do{u←do {x←p; y←p; ret (x, y)};
           ret (fst u = snd u, True, fst u, fst u = snd u)} =
    do{u←do {x←p; y←p; ret (x, y)};
           ret (True, True, fst u, fst u = snd u)}"
    by (rule gdj2doSeq)
  moreover have "∀x. (do {y←p; ret (x = y)}) ∈ Dsef"
    proof -
      from dsef_p have Dsef_p: "p ∈ Dsef"
        by (simp add: Dsef_def)
      have "∀(x::'a) y. dsef (ret(x=y))"
        by (simp add: dsef_ret)
      from this have Dsef_ret: "∀(x::'a) y. ret(x=y) ∈ Dsef"
        by (simp add: Dsef_def)
      from Dsef_p have
        "∀x. ((∀y. (ret(x=y)) ∈ Dsef) ⟶
             (do {y←p; ret (x=y)}) ∈ Dsef)"
        by (simp add: weak_Dsef2seq)
      from Dsef_ret this have "∀x. (do {y←p; ret (x=y)}) ∈ Dsef"
        by simp
      from this show ?thesis
        by simp
    qed
  moreover have "(ret True) ∈ Dsef"
    proof -
      have "dsef (ret (True::bool))"
        by (simp add: dsef_ret)
      from this show ?thesis
```

```
        by (simp add: Dsef_def)
    qed
  ultimately show ?thesis
    apply (simp add: hoare_def)
    by (simp add: gdj_def)
qed

lemma "dsef'":
  assumes dsef_q: "dsef q"
  shows "⦃Φ⦄q⦃Φ⦄"

proof -


  have "∀u. ((fst u=snd(snd u)) ⟶ (fst u⟶snd(snd u)))"
    by simp
  moreover
  have "[a←↓Φ; x←q; b←↓Φ](a=b)"
    proof -
      from dsef_Φ have
        "(∀(q:: bool T). ((cp q) ∧ (sef q)) ⟶
                         (cp (do {x←↓Φ; y←q; ret (x, y)})))"
        by (simp add: dsef_def com_def)
      from this have
        "(∀(q:: 'a T). ((cp q) ∧ (sef q)) ⟶
                       cp (do {x←↓Φ; y←q; ret (x, y)}))"
        by (rule commute_tcoerc)
      from dsef_Φ sef_q cp_q this have cp_do:
        "cp(do {x←↓Φ; y←q; ret(x,y)})"
        by (simp add: dsef_def com_def)
      from sef_Φ sef_q cp_do have
        "do {x←↓Φ; y←q; ret (x, y)} = do {y←q; x←↓Φ; ret (x, y)}"
        by (rule "cpsefProps(i→ii)")
      from cp_Φ sef_Φ this show ?thesis
        by (simp add: "cpsefProps(ii→iv)")
    qed
  from this have
    "[u←do{a←↓Φ; x←q; b←↓Φ;ret(a,x,b)}](fst u=snd(snd u))"
    by (simp add: gdj_def)
  ultimately
  have "[u←do{a←↓Φ; x←q; b←↓Φ;ret(a,x,b)}](fst u ⟶ snd(snd u))"
    by (rule wk)
  from this have "[a←↓Φ; x←q; b←↓Φ] (a⟶b)"
    by (simp add: gdj_def)
  from this show ?thesis
    by (simp add: hoare_def)
qed

lemma "stateless": "⦃⇑Φ⦄p⦃⇑Φ⦄"
```

proof -
  have
    "[(a,x,b)←do {a←ret Φ; x←p; b←ret Φ; ret (a,x,b)}](a⟶b)"
    apply *(unfold gdj_def)*
    by *simp*
  moreover have *"↓⇑Φ = ret Φ"*
    by *(simp add: dsef_ret Dsef_def)*
  ultimately show *?thesis*
    by *(simp only: hoare_def)*
qed

lemma *"emptySeq": "∀x. {|Φ x|}{|Φ x|}"*
  proof -
    have *"sef (ret ())"*
      by *(simp add: sef_def)*
    moreover
    have *"∀x. {|Φ x|}y←ret(){|Φ x|}"*
      by *(simp add: dsef_ret dsef')*
    ultimately show *?thesis*
      apply *(simp add: hoare_def hoare_Tupel_def dsef_def)*
      by *(simp add: sef_rm2of3)*
  qed

You can concat two sequences, one ending up in state Ψ and the other one starting in this state.

Proof:
[a←Φ;x←p;b←Ψ x;y←q x;c←ξ x y](a⟶b)
[a←Φ;x←p;b←Ψ x;y←q x;c←ξ x y](b⟶c)
———————————————— (andI)
[a←Φ;x←p;b←Ψ x;y←q x;c←ξ x y]((a⟶b) ∧(b⟶c))
———————————————— (wk)
[a←Φ;x←p;b←Ψ x;y←q x;c←ξ x y](a⟶c)
———————————————— (dsef-remove Ψ)
[a←Φ;x←p;   y←q x;c←ξ x y](a⟶c)

lemma *"seq":*
  assumes *a: "{|Φ|}x←p{|Ψ x|}"* **and** *b: "∀x. {|Ψ x|}y←q x{|Υ x y|}"*
  shows *"{|Φ|}x←p;y←q x{|Υ x y|}"*

proof -

  have *a': "[a←↓Φ;x←p;b←↓(Ψ x);y←q x;c←↓(Υ x y)](a⟶b)"*

    proof -
      from *a* **have** *"[a←↓Φ;x←p;b←↓(Ψ x)](a⟶b)"*
        by *(simp add: hoare_def)*

```
      from this have
        "[a←↓Φ;x←p;b←↓(Ψ x);u←do{y←q x;c←↓(ϒ x y);ret (y,c)}]
                                                          (a⟶b)"
        by (rule app_exp')
      from this show ?thesis
        by (simp add: gdj_def)
    qed

  moreover
  have b': "[a←↓Φ;x←p;b←↓(Ψ x);y←q x;c←↓(ϒ x y)](b⟶c)"

    proof -
      from b have
        "∀a x. [u≺-(b,y,c)←do{b←↓(Ψ x);y←q x;c←↓(ϒ x y);
                                             ret(b,y,c)}](b⟶c)"
        by (simp add: hoare_def gdj_def)
      from this have
        "[a←↓Φ;x←p;u≺-(b,y,c)←do{b←↓(Ψ x);y←q x;c←↓(ϒ x y);
                                             ret(b,y,c)}](b⟶c)"
        by (rule pre_exp)
      from this show ?thesis
        by simp
    qed

  ultimately have conj:
    "[a←↓Φ;x←p;b←↓(Ψ x);y←q x;c←↓(ϒ x y)]((a⟶b)∧(b⟶c))"

    proof -
      from a' have
        "[u≺-(a,x,b,y,c)←do{a←↓Φ;x←p;b←↓(Ψ x);y←q x;c←↓(ϒ x y);
                          ret(a,x,b,y,c)}](a⟶b)"
        by simp
      from this have
        "[u←do{a←↓Φ;x←p;b←↓(Ψ x);y←q x;c←↓(ϒ x y);
                          ret(a,x,b,y,c)}](fst u⟶fst(snd(snd u)))"
        by (simp add: gdj_def)
      moreover
      from b' have
        "[u≺-(a,x,b,y,c)←do{a←↓Φ;x←p;b←↓(Ψ x);y←q x;c←↓(ϒ x y);
                          ret(a,x,b,y,c)}](b⟶c)"
        by simp
      from this have
        "[u←do{a←↓Φ;x←p;b←↓(Ψ x);y←q x;c←↓(ϒ x y);
                          ret(a,x,b,y,c)}]
                          (fst(snd(snd u))⟶snd(snd(snd(snd u))))"
        by (simp add: gdj_def)
      ultimately
      have
        "[u←do{a←↓Φ;x←p;b←↓(Ψ x);y←q x;c←↓(ϒ x y);
```

```
                        ret(a,x,b,y,c)}]
                        ((fst u⟶fst(snd(snd u))) ∧
                         (fst(snd(snd u))⟶snd(snd(snd(snd u)))))"
          by (rule andI)
       from this have
          "[u≺-(a,x,b,y,c)←do{a←↓Φ;x←p;b←↓(Ψ x);y←q x;c←↓(ϒ x y);
                                    ret(a,x,b,y,c)}]((a⟶b) ∧ (b⟶c))"
          by (simp add: gdj_def)
       from this show ?thesis
          by simp
    qed

  from this have "[a←↓Φ;x←p;b←↓(Ψ x);y←q x;c←↓(ϒ x y)](a⟶c)"

    proof -
       have "∀u. (((fst u)⟶(fst(snd(snd u)))) ∧
                 (fst(snd(snd u))⟶snd(snd(snd (snd u))))) ⟶
                  (fst u ⟶ snd(snd(snd (snd u))))"
          by simp
       moreover
       from conj have
          "[u←do {a←↓Φ;x←p;b←↓Ψ x;y←q x;c←↓ϒ x y;ret (a,x,b,y,c)}]
                                      (((fst u)⟶(fst(snd(snd u)))) ∧
                        (fst(snd(snd u))⟶snd(snd(snd (snd u)))))"
          by (simp add: gdj_def)
       ultimately have
          "[u←do {a←↓Φ;x←p;b←↓Ψ x;y←q x;c←↓ϒ x y;ret (a,x,b,y,c)}]
                                   (((fst u)⟶snd(snd(snd (snd u)))))"
          by (rule wk)
       from this show ?thesis
          by (simp add: gdj_def)
    qed

  moreover
  from dsef_Ψ
  have "∀x. sef ↓(Ψ x)"
    by (simp add: dsef_def)
  from this have
    "[a←↓Φ;x←p;b←↓(Ψ x);y←q x;c←↓(ϒ x y)](a⟶c) ⟹
     [a←↓Φ;x←p;y←q x;c←↓(ϒ x y)](a⟶c)"
    by (simp add: sef_rm3of5)
  ultimately
  have box: "[a←↓Φ;x←p;y←q x;c←↓(ϒ x y)](a⟶c)"
    by simp
  from this show ?thesis

    proof -
       from box have
          "[u←do{a←↓Φ;x←p;y←q x;c←↓(ϒ x y);ret(a,x,y,c)}]
```

```
                              (fst u⟶(snd(snd(snd u))))"
        by (simp add: gdj_def)
      from this have
        "do{u←do{a←↓Φ;x←p;y←q x;c←↓(ϒ x y);ret(a,x,y,c)};
            ret(fst u⟶snd(snd(snd u)),fst u,
                (fst(snd u),fst(snd(snd u))),snd(snd(snd u)))} =
        do{u←do{a←↓Φ;x←p;y←q x;c←↓(ϒ x y);ret(a,x,y,c)};
            ret(True,fst u,
                (fst(snd u),fst(snd(snd u))),snd(snd(snd u)))}"
        by (rule gdj2doSeq)
      from this show ?thesis
        by (simp add: hoare_def gdj_def)
    qed

qed
```

sequencing works also for longer sequences

```
lemma "seq_exp":
  assumes a: "⦃Φ⦄x←p;y←q x⦃Ψ x y⦄" and
          b: "∀x y. ⦃Ψ x y⦄z←r x y⦃ϒ x y z⦄"
  shows "⦃Φ⦄x←p;y←q x;z←r x y⦃ϒ x y z⦄"
proof -
  from a have "⦃Φ⦄u←do{x←p;y←q x;ret(x,y)}⦃Ψ (fst u) (snd u)⦄"
    by (simp add: hoare_def)
  moreover from b have
    "∀u. ⦃Ψ (fst u) (snd u)⦄
           z←r (fst u)(snd u)
        ⦃ϒ (fst u) (snd u) z⦄"
    by (simp add: hoare_def)
  ultimately have
    "⦃Φ⦄
        u←do{x←p;y←q x;ret(x,y)};z←r (fst u) (snd u)
     ⦃ϒ (fst u) (snd u) z⦄"
    by (rule seq)
  from this have
    "[u←do {a←↓Φ; x←p; y←q x; z←r x y; b←↓ϒ x y z;
                        ret (a,((x,y),z),b)}](fst u ⟶ snd(snd u))"
    by (simp add: hoare_def gdj_def)
  from this have
    "do {u←do{a←↓Φ; x←p; y←q x; z←r x y; b←↓ϒ x y z;
                ret(a,((x,y),z),b)};
                ret (fst u ⟶ snd(snd u),
                     fst u,(fst(fst(fst(snd u))),
                     snd(fst(fst(snd u))),snd(fst(snd u))),
                     snd(snd u))} =
     do {u←do{a←↓Φ; x←p; y←q x; z←r x y; b←↓ϒ x y z;
                ret(a,((x,y),z),b)};
                ret (True,
                     fst u,(fst(fst(fst(snd u))),
```

```
                      snd(fst(fst(snd u))),snd(fst(snd u))),
                      snd(snd u))}"
    by (rule gdj2doSeq)
  from this show ?thesis
    by (simp add: hoare_def gdj_def)
qed

lemma "hoare_ctr":
  assumes "{|Φ|}x←p;y←r x{|Ψ y|}"
  shows "{|Φ|}y←(do{x←p;r x}){|Ψ y|}"
proof -
  from prems have
    "[u←do {a←↓Φ; x←p; y←r x; b←↓Ψ y; ret (a, (x, y), b)}]
                                                  (fst u⟶snd(snd u))"
    by (simp add: hoare_def gdj_def)
  from this have
    "do {u←do {a←↓Φ; x←p; y←r x; b←↓Ψ y; ret (a, (x, y), b)};
         ret (fst u⟶snd(snd u), fst u, fst(fst(snd u)),
         snd (fst(snd u)), snd(snd u))} =
    do {u←do {a←↓Φ; x←p; y←r x; b←↓Ψ y; ret (a, (x, y), b)};
         ret (True, fst u, fst(fst(snd u)),
         snd (fst(snd u)), snd(snd u))}"
    by (rule gdj2doSeq)
  from this have
    "[(a,x,y,b)←do {a←↓Φ; x←p; y←r x; b←↓Ψ y;
                                         ret (a, x, y, b)}](a ⟶ b)"
    by (simp add: gdj_def)
  moreover from this show ?thesis
    apply (simp only: hoare_def)
    apply (rule ctr)
    by simp
qed
```

You can weaken pre- and postconditions Φ and Ψ with Φ' and Ψ' if from Φ follows Φ' and from Ψ' follows Ψ.

Proof:

[a←Φ';b←Φ](a⟶b)
———————————————— (app)
(1)*[a←Φ';b←Φ;x←p;c←Ψ x](a⟶b)*
—
[b←Φ;x←p;c←Ψ x](b⟶c)
———————————————— (pre)
(2)*[a←Φ';b←Φ;x←p;c←Ψ x](b⟶c)*
—
(1) (2)
———————————————————— (andI)
[a←Φ';b←Φ;x←p;c←Ψ x]((a⟶b) ∧(b⟶c))

———————————————— (wk) (dsef-remove Φ)

(3) *[a←Φ';x←p;c←Ψ x] (a⟶c)*

—

with ∀*a x. [c←Ψ x;d←Ψ' x](c⟶d)* and (3) follow the same tactic as the proof of (3) gets:

[a←Φ';x←p;d←Ψ' x] (a⟶d)

```
lemma "wk":
  assumes a: "⦃Φ⦄x←p⦃Ψ x⦄" and b: "Φ'⇒h Φ" and
  c: "∀x. (Ψ x) ⇒h (Ψ' x)"
  shows "⦃Φ'⦄x←p⦃Ψ' x⦄"

proof -


  have d: "[a←↓Φ';b←↓Φ;x←p;c←↓(Ψ x)] ((a⟶b) ∧ (b⟶c))"

  proof -
    from b have "[a←↓Φ';b←↓Φ](a⟶b)"
      by (simp add: hoare_Tupel_def)
    from this have
      "[a←↓Φ';b←↓Φ;u←do{x←p;c←↓(Ψ x);ret(x,c)}](a⟶b)"
      by (rule app_exp)
    from this have d: "[a←↓Φ';b←↓Φ;x←p;c←↓(Ψ x)](a⟶b)"
      by (simp add: gdj_def)
    from a have
      "∀a. [u←do{b←↓Φ;x←p;c←↓(Ψ x);ret(b,x,c)}]
                                        (fst u⟶snd(snd u))"
      by (simp add: hoare_def gdj_def)
    from this have
      "[a←↓Φ';u←do{b←↓Φ;x←p;c←↓(Ψ x);ret(b,x,c)}]
                                        (fst u⟶snd(snd u))"
      by (rule pre)
    from this have "[a←↓Φ';b←↓Φ;x←p;c←↓(Ψ x)](b⟶c)"
      by (simp add: gdj_def)
    from d this show ?thesis
      by (rule andI_exp)
  qed

  from d have "[a←↓Φ';b←(↓Φ);x←p;c←↓Ψ x] (a⟶c)"

  proof -
    have "∀a b x c. ((a⟶b) ∧ (b⟶c)) ⟶ (a⟶c)"
      by simp
    moreover
    assume "[a←↓Φ';b←↓Φ;x←p;c←↓Ψ x] ((a⟶b) ∧ (b⟶c))"
    ultimately show ?thesis
      by (rule wk_exp)
  qed
```

```
  from sef_Φ this have "[a←↓Φ';x←p;c←↓Ψ x] (a⟶c)"
    by (rule sef_rm2of4)
  from this have e: "[a←↓Φ';x←p;c←↓Ψ x;d←↓Ψ' x] ((a⟶c) ∧ (c⟶d))"

  proof -
    assume "[a←↓Φ';x←p;c←↓Ψ x] (a⟶c)"
    from this have "[a←↓Φ';x←p;c←↓Ψ x;d←↓Ψ' x] (a⟶c)"
      by (rule app_exp')
    moreover
    from c have
      "∀a x. [u←do{c←↓Ψ x;d←↓Ψ' x;ret(c,d)}](fst u⟶snd u)"
      by (simp add: gdj_def hoare_Tupel_def)
    from this have
      "[a←↓Φ';x←p;u←do{c←↓Ψ x;d←↓Ψ' x;ret(c,d)}](fst u⟶snd u)"
      by (rule pre_exp)
    from this have "[a←↓Φ';x←p;c←↓Ψ x;d←↓Ψ' x](c⟶d)"
      by (simp add: gdj_def)
    ultimately
    show ?thesis
      by (rule andI_exp)
  qed

  from e have "[a←↓Φ';x←p;c←↓Ψ x;d←↓Ψ' x] (a⟶d)"

  proof -
    have "∀a x c d. ((a⟶c) ∧ (c⟶d)) ⟶ (a⟶d)"
      by simp
    moreover
    assume "[a←↓Φ';x←p;c←↓Ψ x;d←↓Ψ' x] ((a⟶c) ∧ (c⟶d))"
    ultimately
    show ?thesis
      by (rule wk_exp)
  qed

  from sef_Ψ this have "[a←↓Φ'; x←p; d←↓Ψ' x](a ⟶ d)"
    by (rule sef_rm3of4)
  from this show ?thesis
    by (simp add: hoare_def)
qed
```

Most of the time you only want to weaken pre- or postcondition instead of both. For this purpose wk_pre and wk_post take care of the unchanged condition.

```
lemma "wk_pre":
  assumes a: "⦃Φ⦄x←p⦃Ψ x⦄" and b: "Φ'⇒_h Φ"
  shows "⦃Φ'⦄x←p⦃Ψ x⦄"
proof -
  have "∀x. ⦃Ψ x⦄⦃Ψ x⦄"
    by (rule emptySeq)
```

```
  from a b this show ?thesis
    by (rule wk)
qed

lemma "wk_post":
  assumes a: "{|Φ|}x←p{|Ψ x|}" and c: "∀x. (Ψ x) ⇒h (Ψ′ x)"
  shows "{|Φ|}x←p{|Ψ′ x|}"
proof -
  have "{|Φ|}{|Φ|}"
    apply (rule allE)
    apply (rule emptySeq)
    by simp
  from a this c show ?thesis
    by (rule wk)
qed
```

Two Hoare-Triples with differnt preconditions Φ and Ψ but eqal sequence and postcondition can be combined to one Hoare-Triple with precondition Φ ∨ Ψ

Proof:

$$\frac{[a\leftarrow\Phi;\quad x\leftarrow p;b\leftarrow\xi\ x](a\longrightarrow b)}{(1)[a\leftarrow\Phi;c\leftarrow\Psi;x\leftarrow p;b\leftarrow\xi\ x](a\longrightarrow b)}\ \text{(pre) (dsef-switch)}$$

—

$$\frac{[c\leftarrow\Psi;\ x\leftarrow p;\ b\leftarrow\xi\ x](c\longrightarrow b)}{(2)[a\leftarrow\Phi;c\leftarrow\Psi;x\leftarrow p;b\leftarrow\xi\ x](c\longrightarrow b)}\ \text{(pre)}$$

—

$$\frac{\dfrac{\dfrac{(1)\ (2)}{[a\leftarrow\Phi;c\leftarrow\Psi;x\leftarrow p;b\leftarrow\xi\ x]((a\longrightarrow b)\wedge(c\longrightarrow b))}\ \text{(andI)}}{[a\leftarrow\Phi;c\leftarrow\Psi;x\leftarrow p;b\leftarrow\xi\ x](a\vee c\longrightarrow b)}\ \text{(wk)}}{[d\leftarrow do\{a\leftarrow\Phi;c\leftarrow\Psi;ret(a\vee c)\};x\leftarrow p;b\leftarrow\xi\ x]\ (d\longrightarrow b)}\ \text{(preOp)}$$

```
lemma "disj":
  assumes a: "{|Φ|} x←p {|Υ x|}" and b:"{|Ψ|} x←p {|Υ x|}"
  shows "{|(Φ∨D Ψ)|} x←p {|Υ x|}"

proof -


  have
    "[(a,c,x,b)←do{a←↓Φ;c←↓Ψ;x←p;b←↓Υ x;ret(a,c,x,b)}] (a ⟶ b)"

    proof -
      from a have
```

```
      "∀c. [u←do {a←↓Φ; x←p; b←↓ϒ x; ret (a, x, b)}]
                                        (fst u ⟶ snd (snd u))"
      by (simp add: hoare_def gdj_def)
    from this have
      "[c←↓Ψ; u←do {a←↓Φ; x←p; b←↓ϒ x; ret (a, x, b)}]
                                        (fst u ⟶ snd (snd u))"
      by (rule pre)
    from this
    have "[c←↓Ψ; a←↓Φ; x←p; b←↓ϒ x](a ⟶ b)"
      by (simp add: gdj_def)
    from this dsef_Φ dsef_Ψ have
      "[a←↓Φ; c←↓Ψ;x←p; b←↓ϒ x](a ⟶ b)"
      apply (subst dsef_switch)
      apply simp
      apply simp .
    from this prems show ?thesis
      by (simp add: gdj_def)
  qed

moreover
from b have
  "[(a,c,x,b)←do{a←↓Φ;c←↓Ψ;x←p;b←↓ϒ x;ret(a,c,x,b)}] (c ⟶ b)"

  proof -
    from prems have
      "∀a. [u←do {c←↓Ψ; x←p; b←↓ϒ x; ret (c, x, b)}]
                                        (fst u ⟶ snd (snd u))"
      by (simp add: hoare_def gdj_def)
    from this have
      "[a←↓Φ; u←do {c←↓Ψ; x←p; b←↓ϒ x; ret (c, x, b)}]
                                        (fst u ⟶ snd (snd u))"
      by (rule pre)
    from this show ?thesis
      by (simp add: gdj_def)
  qed

ultimately have
  "[(a,c,x,b)←do{a←↓Φ;c←↓Ψ;x←p;b←↓ϒ x;ret(a,c,x,b)}]
                                            ((a⟶b) ∧ (c⟶b))"
  by (rule andI_exp)
from this have
  "[(a,c,x,b)←do{a←↓Φ;c←↓Ψ;x←p;b←↓ϒ x;ret(a,c,x,b)}] (a∨c⟶b)"
  by (simp add: wk)
from this have
  "[(d,x,b)←do{d←do{a←↓Φ;c←↓Ψ;ret(a∨c)};x←p;b←↓ϒ x;
                                          ret(d,x,b)}] (d⟶b)"
  by (rule preOp)
moreover have "(do{x←↓Φ; y←↓Ψ; ret (x ∨ y)}) ∈ Dsef"
  proof -
```

```
      have "∀x y. dsef (ret (x∨y::bool))"
        by (simp add: dsef_ret)
      from this have "∀x y. (ret (x∨y::bool)) ∈ Dsef"
        by (simp add: Dsef_def)
      from Dsef_Φ Dsef_Ψ this show ?thesis
        by (simp add: weak_Dsef2seq_exp)
    qed
  ultimately show ?thesis
    by (simp add: hoare_def condDisj_def)
qed
```

Two Hoare-Triples with equal precondition and sequence but differnt postconditions Ψ and Υ can be combined to one Hoare-Triple with postcondition $\Psi \ \wedge \ \Upsilon$

```
lemma "conj":
  assumes a: "⦃Φ⦄x←p⦃Ψ x⦄" and b: "⦃Φ⦄x←p⦃Υ x⦄"
  shows "⦃Φ⦄x←p⦃(Ψ x)∧D(Υ x)⦄"


proof -


  have "[a←↓Φ;x←p;b←↓(Ψ x);c←↓(Υ x)] (a⟶b)"

  proof -
    from a have "[a←↓Φ;x←p;b←↓(Ψ x)] (a⟶b)"
      by (simp add: hoare_def)
    from this show ?thesis
      by (rule app_exp')
  qed

  moreover have "[a←↓Φ;x←p;b←↓(Ψ x);c←↓(Υ x)] (a⟶c)"

  proof -
    from b have "[a←↓Φ;x←p;c←↓(Υ x)] (a⟶c)"
      by (simp add: hoare_def)
    from this have "[a←↓Φ;x←p;c←↓(Υ x);b←↓(Ψ x)] (a⟶c)"
      by (rule app_exp')
    from dsef_Ψ dsef_Υ this show ?thesis
      apply (subst dsef_switch') .
  qed

  ultimately
  have c: "[a←↓Φ;x←p;b←↓(Ψ x);c←↓(Υ x)] ((a⟶b) ∧ (a⟶c))"
    by (simp add: andI_exp)
  have "∀a x b c. ((a⟶b) ∧ (a⟶c)) ⟶ (a⟶(b∧c))"
    by simp
  from this c have
    "[a←↓Φ;x←p;b←↓(Ψ x);c←↓(Υ x)] (a⟶(b∧c))"
    by (rule wk_exp)
```

```
  from this have
    "[(a,x,b,c)←do{a←↓Φ;x←p;b←↓(Ψ x);c←↓(Υ x);
                                                    ret(a,x,b,c)}](a⟶(b∧c))"
    by simp
  from this have
    "[(a,x,d)←do{a←↓Φ;x←p;d←do{b←↓(Ψ x);c←↓(Υ x);ret(b∧c)};
                                                    ret(a,x,d)}] (a⟶d)"
    by (rule postOp)
  moreover
  have "∀x. (do {b←↓Ψ x; c←↓Υ x; ret (b ∧ c)}) ∈ Dsef"
    proof -
      have "∀b c. dsef (ret(b∧c::bool))"
        by (simp add: dsef_ret)
      from this have "∀b c. (ret(b∧c::bool)) ∈ Dsef"
        by (simp add: Dsef_def)
      from Dsef_Ψ Dsef_Υ this show ?thesis
        by (simp add: weak_Dsef2seq_exp)
    qed
  from this
  have
    "∀x. ((↓↑(do {b←Rep_Dsef (Ψ x);c←Rep_Dsef (Υ x);ret (b ∧ c)}))
          = do {b←Rep_Dsef (Ψ x); c←Rep_Dsef (Υ x); ret (b ∧ c)})"
    by simp
  ultimately show ?thesis
    by (simp add: hoare_def condConj_def)
qed


lemma "if_True":
  assumes "{|Φ∧_D⇑v|} x←p {|Ψ x|}"
  shows "{|Φ∧_D⇑v|} x←(if v then p else q){|Ψ x|}"
proof (cases v)
  case True
  from this prems show "{|Φ∧_D ⇑v|}x←if v then p else q{|Ψ x|}"
    by (simp add: hoare_def condConj_def)
next
  case False
  have Dsef_ret: "(ret False) ∈ Dsef"
    by (simp add: dsef_ret Dsef_def)
  have f1: "{|Φ∧_D ⇑v|}{|⇑False|}"
    proof -
      have Dsef_Φ: "↓Φ ∈ Dsef"
        by (simp add: Rep_Dsef)
      moreover
      from Dsef_ret have "(ret False) ∈ Dsef"
        by simp
      moreover have "∀x y. ret (x ∧ y) ∈ Dsef"
        by (simp add: dsef_ret Dsef_def)
      ultimately have "do {x←↓Φ; y←ret False; ret (x ∧ y)} ∈ Dsef"
```

```
        by (rule weak_Dsef2seq_exp)
      from prems this Dsef_ret show ?thesis
        by (simp add: hoare_Tupel_def condConj_def gdj_def)
    qed
  from Dsef_ret have "{|⇑False|}x←if v then p else q{|Ψ x|}"
    by (simp add: hoare_def gdj_def)
  from this f1 show "{|Φ∧_D ⇑v|}x←if v then p else q{|Ψ x|}"
    by (rule wk_pre)
qed

lemma "if_False":
  assumes "{|Φ∧_D⇑¬v|} x←q {|Ψ x|}"
  shows "{|Φ∧_D⇑¬v|} x←(if v then p else q){|Ψ x|}"
proof (cases v)
  case True
  have Dsef_ret: "(ret False) ∈ Dsef"
    by (simp add: dsef_ret Dsef_def)
  have f1: "{|Φ∧_D ⇑¬v|}{|⇑False|}"
    proof -
      have Dsef_Φ: "↓Φ ∈ Dsef"
        by (simp add: Rep_Dsef)
      moreover
      from Dsef_ret have "(ret False) ∈ Dsef"
        by simp
      moreover have "∀x y. ret (x ∧ y) ∈ Dsef"
        by (simp add: dsef_ret Dsef_def)
      ultimately have "do {x←↓Φ; y←ret False; ret (x ∧ y)} ∈ Dsef"
        by (rule weak_Dsef2seq_exp)
      from prems this Dsef_ret show ?thesis
        by (simp add: hoare_Tupel_def condConj_def gdj_def)
    qed
  from Dsef_ret have "{|⇑False|}x←if v then p else q{|Ψ x|}"
    by (simp add: hoare_def gdj_def)
  from this f1 show "{|Φ∧_D ⇑¬v|}x←if v then p else q{|Ψ x|}"
    by (rule wk_pre)
next
  case False
  from this prems show "{|Φ∧_D ⇑¬v|}x←if v then p else q{|Ψ x|}"
    by (simp add: hoare_def condConj_def split_def)
qed

lemma if:
  assumes a: "{|Φ∧_D b|} x←p {|Ψ x|}" and b: "{|Φ∧_D ¬_Db|} y←q {|Ψ y|}"
  shows "{|Φ|} x←(if_D b then p else q){|Ψ x|}"
proof -


  have "{|Φ∧_D b|} x←(if_D b then p else q){|Ψ x|}"
    proof -
```

```
      have "⦃Φ∧_D b⦄ v←↓b⦃(Φ∧_D b)∧_D ⇑v⦄"
        proof -
          from Dsef_ret Dsef_b Dsef_Φ Dsef1 Dsef2 Dsef3 cp_b show ?thesis
            apply (simp add: hoare_def condConj_def)
            apply (simp add: cp_ret2seq)
            apply (rule double2)
            apply (simp add: Dsef_def)
            apply (simp add: Dsef_def)
            by (simp add: gdj_def Dsef_def)
        qed
      moreover from a have
        "∀v. ⦃(Φ∧_D b)∧_D ⇑v⦄x←p⦃Ψ x⦄"
        proof -
          from Dsef1 Dsef3 dsef_Φ dsef_b have "∀v. ⦃(Φ∧_D b)∧_D ⇑v⦄⦃Φ∧_D b⦄"
            apply (simp add: hoare_Tupel_def condConj_def dsef_ret Dsef_def)
            apply auto
            apply (rule double)
            apply auto
            by (simp add: gdj_def)
          from a this have "∀v. ⦃(Φ∧_D b)∧_D ⇑v⦄x←p⦃Ψ x⦄"
            apply auto
            apply (rule wk_pre)
            apply simp
            by simp
          from this show ?thesis
            by (simp add: hoare_def condConj_def)
        qed
      from this have "∀v. ⦃(Φ∧_D b)∧_D ⇑v⦄x←if v then p else q⦃Ψ x⦄"
        by (simp add: if_True)
      ultimately have seq:
        "⦃Φ∧_D b⦄ v←↓b;x←if v then p else q⦃Ψ x⦄"
        by (rule seq)
      from this have "⦃Φ∧_D b⦄ x←do{v←↓b;if v then p else q}⦃Ψ x⦄"
        by (rule hoare_ctr)
      from this show ?thesis
        by (simp add: if_D_def  if_T_def)
    qed
  moreover
  have "⦃Φ∧_D ¬_D b⦄ x←(if_D b then p else q)⦃Ψ x⦄"
    proof -
      have "⦃Φ∧_D ¬_D b⦄ v←↓b⦃(Φ∧_D ¬_D b)∧_D ⇑¬v⦄"
        proof -
          from Dsef_ret Dsef_b Dsef_Φ Dsef4 Dsef5 Dsef6 Dsef7 cp_b show
?thesis
            apply (simp add: hoare_def condConj_def condNot_def)
            apply (simp add: cp_ret2seq)
            apply (rule double2)
            apply (simp add: Dsef_def)
            apply (simp add: Dsef_def)
```

```
            by (simp add: gdj_def Dsef_def)
        qed
      moreover have
        "∀v. ⦃(Φ∧_D ¬_D b)∧_D ⇑¬v⦄x←q⦃Ψ x⦄"
        proof -
          from Dsef2 Dsef2 Dsef3 Dsef4 Dsef5 Dsef6 Dsef7 Dsef8 dsef_Φ
dsef_b have
            "∀v. ⦃(Φ∧_D ¬_D b)∧_D ⇑¬v⦄⦃Φ∧_D¬_D b⦄"
            apply (simp add: hoare_Tupel_def condConj_def dsef_ret Dsef_def
condNot_def)
            apply auto
            apply (rule double)
            apply auto
            by (simp add: gdj_def)
          from b this have "∀v. ⦃(Φ∧_D¬_D b)∧_D ⇑¬v⦄x←q⦃Ψ x⦄"
            apply auto
            apply (rule wk_pre)
            apply simp
            by simp
          from this show ?thesis
            by (simp add: hoare_def condConj_def)
        qed
      from this have "∀v. ⦃(Φ∧_D¬_D b)∧_D ⇑¬v⦄x←if v then p else q⦃Ψ x⦄"
        by (simp add: if_False)
      ultimately have seq:
        "⦃Φ∧_D ¬_D b⦄ v←↓b;x←if v then p else q⦃Ψ x⦄"
        by (rule seq)
      from this have "⦃Φ∧_D ¬_D b⦄ x←do{v←↓b;if v then p else q}⦃Ψ x⦄"
        by (rule hoare_ctr)
      from this show ?thesis
        by (simp add: if_D_def)
    qed
  ultimately have
    "⦃(Φ∧_D b)∨_D(Φ∧_D ¬_D b)⦄ x←(if_D b then p else q)⦃Ψ x⦄"
    by (rule disj)
  moreover
  from Dsef1 Dsef3 Dsef4 Dsef5 Dsef6 cp_Φ dsef_Φ dsef_b have
    "⦃Φ⦄⦃(Φ∧_D b)∨_D(Φ∧_D ¬_D b)⦄"
    apply (simp add: condConj_def condDisj_def condNot_def hoare_Tupel_def)
    apply (simp only: cp_ret2seq)
    apply (rule double)
    apply auto
    by (simp add: gdj_def)
  ultimately show ?thesis
    by (rule wk_pre)
qed
```

The rule for iter is not prooven yet. To demonstrate the use of Hoare-Triples we axiomatize it because there are only a few „real“ algorithms without iteration

axioms

```
  iter: "{|(Φ x) ∧_D (b x)|} y←(p x) {|Φ y|} ⟹
          {|Φ x|} y←(iter_D b p x){|(Φ y) ∧_D (¬_D(b y))|}"
```

end

Literaturverzeichnis

[And02] Peter B. Andrews. *An Introduction to Mathematical Logic: To Truth Through Proof*. Number 27 in Applied Logic Series. Kluwer Academic Publishers, 2002.

[Asp05] David Aspinall. Proof General. 2005. `http://proofgeneral.inf.ed.ac.uk/` [Stand 22.06.05].

[Bac04] Danial Bachfeld. Software-Fehler verursachte US-Stromausfall 2003. *heise online news*, 13.02.2004, 2004. `http://www.heise.de/newsticker/meldung/44621` [Stand 03.09.05].

[FM] Emacs Folding Mode. `http://www.chrislott.org/geek/emacs/n2n_folding_mode.php` [Stand 15.07.05].

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[isa05] Isabelle Homepage. 2005. `http://www.cl.cam.ac.uk/Research/HVG/Isabelle/` [Stand 03.09.05].

[Jon01] Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell, 2001.

[Jon03] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, Apr 2003.

[Kur01] Michael Kurzidim. Software-Fehler legt Geldautomaten lahm (Update). *heise online news*, 03.07.2001, 2001. `http://www.heise.de/newsticker/meldung/18957` [Stand 03.09.05].

[Mog91] Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991.

[Nip03] Tobias Nipkow. Structured Proofs in Isar/HOL. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs (TYPES 2002)*, volume 2646, pages 259–278, 2003.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[Pau04] Lawrence C. Paulson. *The Isabelle Reference Manual*. 2004. `http://www.cl.cam.ac.uk/Research/HVG/Isabelle/dist/packages/Isabelle/doc/isar-ref.pdf` [Stand 03.09.05].

[SM03] Lutz Schröder and Till Mossakowski. Monad-independent hoare logic in HasCASL. In Mauro Pezze, editor, *Fundamental Approaches to Software Engineering (FASE 2003)*, volume 2621 of *Lecture Notes in Computer Science*, pages 261–277. Springer; Berlin; http://www.springer.de, 2003.

[SM04] Lutz Schröder and Till Mossakowski. Monad-independent dynamic logic in HasCASL. *Journal of Logic and Computation*, 14(4):571–619, 2004. Earlier version appeared in Martin Wirsing, Dirk Pattinson, and Rolf Hennicker (eds.), Recent Trends in Algebraic Development Techniques, 16th International Workshop (WADT 2002), LNCS vol. 2755, Springer, Berlin, 2003, pp. 425-441.

[Vog02] Uwe Vogel. Software-Fehler beim 7er-BMW. *heise online news*, 28.05.2002, 2002. `http://www.heise.de/newsticker/meldung/27727` [Stand 03.09.05].

[Wal05] Dennis Walter. *Monadic Dynamic Logic: Application and Implementation*. 2005.

[WB04] Markus Wenzel and Stefan Berghofer. *The Isabelle System Manual*. 2004. `http://www.cl.cam.ac.uk/Research/HVG/Isabelle/dist/packages/Isabelle/doc/system.pdf` [Stand 03.09.05].

[Wed03] Christoph Wedler. X-Symbol for WYSIWYG in Emacs. 2003. `http://x-symbol.sourceforge.net/` [Stand 03.09.05].

[Wen04] Markus Wenzel. *The Isabelle/Isar Reference Manual*. 2004. `http://www.cl.cam.ac.uk/Research/HVG/Isabelle/dist/packages/Isabelle/doc/isar-ref.pdf` [Stand 03.09.05].