Technische Universität Braunschweig

Carl-Friedrich-Gauß-Fakultät für Mathematik und Informatik
Institut für Software Systems Engineering

# Isabelle/HOL-Umsetzung strombasierter Definitionen zur Verifikation von verteilten, asynchron kommunizierenden Systemen

Informatik-Bericht 2006-03

Borislav Gajanovic
Bernhard Rumpe



Braunschweig, den 20. Juli 2006

Dieser technische Bericht beschreibt grundlegende Datenstrukturen, Funktionen und Prinzipien zur Verifikation des Verhaltens verteilter, asynchron kommunizierender Systeme wie etwa Bussysteme im Auto, Telekommunikationsnetzen oder dem Internet.

Dazu werden die im Focus-Ansatz [BS01] eingeführten Ströme im Theorembeweiser Isabelle [NPW02] in komfortabler Weise so umgesetzt, dass sowohl verteilte Komponenten als auch Protokolle zwischen Komponenten einfach und elegant definiert und ihre Eigenschaften untersucht werden können. Zur Verfügung stehen Techniken zur Definition von Liveness- und Fairness-Eigenschaften, schnittstellenbasierte Komposition und Verfeinerungstechniken, die miteinander kompatibel sind.

Die Focus-Theorie basiert im Wesentlichen auf der Formalisierung von Kommunikationshistorien und von Komponenten als mathematische Funktion, die Eingabe- in Ausgabehistorien abbildet.

Das hier vorgestellte Modell für potentiell unendliche Ströme nutzt einen grundlegenden neuen Datentypkonstruktor `fstream`, der zwar auf HOLCF beruht, in dessen Umgang aber der Anwender nicht explizit auf HOLCF angewiesen ist. Dadurch ist HOLCF vor dem Anwender verborgen.

# Inhaltsverzeichnis

# 1 Motivation und Einleitung

Die Sicherstellung der Korrektheit von Software basierten Systemen ist heute mehr denn je eine große Herausforderung. Dies gilt insbesondere für die korrekte Zusammenarbeit von Komponenten in verteilten, asynchron kommunizierenden Systemen wie etwa Bussystemen im Auto (siehe z.B. [BT06]), Telekommunikationsnetzen (siehe z.B. [Krü00]) oder dem Internet (z.B. Web Engineering). Die Kosten einer Verifikation liegen heute immer noch um einiges höher als die Kosten von Softwareentwicklung und Testen zusammen. Kritisch ist dabei auch die zeitliche Verzögerung bei der Einführung von neuer Software in den Markt. Um diese Kosten zu reduzieren, werden seit einigen Jahren für Theorembeweiser wie Isabelle [Mün06, Pau03a, Pau03b, NPW02, Pau94] immer ausgefeiltere Theorien entwickelt, die auch Verifikationsunterstützung für spezifische Domänen bieten.

Für verteilte, asynchron kommunizierende Systeme bietet sich der Focus-Ansatz [BS01, Rum96, BDD+92a, BDD+92b, DW92, HSSS96] besonders an, weil er nicht nur eine vollständig formalisierte Grundlage darstellt, sondern es auch erlaubt, alle relevanten Phänomene derartiger Systeme zu modellieren und analysieren. Dazu gehören Liveness, Fairness, schnittstellenbasierte und zustandsbasierte Komposition und Verfeinerungstechniken, die mit der Komposition kompatibel sind. Deshalb ist es nahe liegend, für die Verifikation dieser Systemklasse eine Codierung des Focus-Ansatzes in Isabelle/HOL vorzunehmen. Dabei ist besonders darauf zu achten, dass die Theorie adäquat umgesetzt wird und komfortable Funktionalitäten zur Verfügung gestellt werden, die eine kompakte und effektive Definition und Verifikation von Eigenschaften erlauben.

Die Focus-Theorie basiert im Wesentlichen auf der Formalisierung von Kommunikationshistorien in Form von Strömen. Dies sind endliche oder unendliche Sequenzen von Nachrichten auf einem Kommunikationskanal. Eine Komponente wird dann durch eine mathematische Funktion repräsentiert, die Eingabe- in Ausgabeströme abbildet. Die Spezifikation einer solchen Komponente besteht aus einer Menge von möglichen Verhalten, also einer Menge von Funktionen.

Eine Umsetzung dieses Ansatzes in einem Verifikationswerkzeug wie Isabelle ist bisher noch nicht optimal gelungen. In Isabelle/HOLCF ist der Datentypkonstruktor `stream` für allgemeine Strombereiche (engl. general stream domain) als einer der Basisdatentypkonstruktoren mit rudimentären Eigenschaften in Form von Lemmata und wenigen Funktionen definitorisch vorhanden. Streams sind bereichstheoretisch partielle oder unendliche Sequenzen auf einem zugrunde liegenden Datentyp, der eine cpo mit kleinstem Element sein muss.

Es zeigt sich jedoch, dass in HOLCF keine vernünftige Definition der Konkatenation auf Streams zu finden ist. Die Schwierigkeit liegt darin, dass die Konkatenation wegen der potentiellen Unendlichkeit der Argumente auf solchen Typen nicht stetig bzw. keine HOLCF-Funktion ist. Sie muss aber auch auf unendlichen Streams definiert sein. Alternativ ist dies im Rahmen von HOL nur mit Fallunterscheidung über die Länge des ersten Stream möglich.

Es wird daher zusätzlich eine explizite Längenfunktion notwendig oder es müssen die eher unbequemen HOL-Deskriptionsoperatoren verwendet werden. Bei ähnlichen Funktionen, wie der unendlichen Konkatenation oder dem fairen Merge auf Streams beliebiger Länge, entsteht immer analoger Aufwand. Eine einheitliche Definitionsmöglichkeit mit Hilfe mächtiger HOLCF-Konstrukte wäre deutlich angenehmer.

Um ein passendes, formales Modell für potentiell unendliche Sequenzen bzw. Ströme zu erhalten, entwickeln wir in dieser Arbeit den grundlegenden Datentypkonstruktor `fstream`, der einerseits auf dem besprochenen Konstruktor `stream` beruht, aber andererseits `stream` vollständig kapselt. Mit dem Konstruktor `fstream` können Ströme (in der Literatur $M^{\omega}$) auf beliebigen Typen (nicht nur pcpo's) erzeugt werden. Die so konstruierten Typen entsprechen deshalb dem Begriff der Ströme im Sinne von Focus. Für alternative Sequenzenmodelle und einen umfassenden Vergleich siehe [DGM97].

Durch geschickte Anwendung von konstruktiven Prinzipien der Definition von Strömen entfallen jegliche Fallunterscheidungen nach endlichen oder unendlichen Strömen genau so wie notwendige Stetigkeitsbeweise bei vielen Funktionsdefinitionen bzw. Lemmata. Dazu wird eine geeignete Infrastruktur aus grundlegenden, für die Konstruktion von Strömen einsetzbaren Funktionen und Rekursionsprinzipien entwickelt. Im Normalfall reichen diese Techniken auf `fstream` zur Funktionsdefinition vollkommen aus, sodass ein Rückgriff auf den zugrunde liegenden HOLCF-Typ `stream` nicht notwendig wird. Die entstehenden Funktionen werden, soweit sinnvoll, auch in eine entsprechende Erweiterung des Isabelle-Simplifikators eingebunden.

Die oben angesprochenen Lösungen bieten also die Möglichkeit die Vorteile und Fähigkeiten beider Logiken einheitlich zu nutzen und damit Isabelle/HOL und Isabelle/HOLCF zur Grundlage einer komfortablen Infrastruktur für die Verifikation von strombasierten Spezifikationen zu vereinen. Die Implementierung ist an die theoretischen Grundlagen von HOLCF angelehnt, bietet dem Benutzer jedoch in der letzten Instanz eine flexible, d.h. von strengen HOLCF-Konstrukten unabhängige, Theorie für Ströme an. Die Infrastruktur kann damit auch unabhängig vom HOLCF-Hintergrund und in den anderen, auf Strömen (also potentiell unendlichen Sequenzen) basierenden Anwendungsbereichen verwendet werden. So können z.B. mit der hier vorliegenden Infrastruktur funktionale Programme auf unendlichen Listen formalisiert bzw. verifiziert werden (siehe z.B. [Tho99, Bir98, Pau96] ). Ferner eignet sich der vorliegende Bericht auch zum kompakten Einstieg in die Bereichstheorie.

## 1.1 Überblick

**Kapitel 2** In diesem Kapitel werden Isabelle und die beiden Instanzen Isabelle/HOL und Isabelle/HOLCF kurz vorgestellt. Insbesondere wird das Zusammenwirken beider Logiken näher beleuchtet. Es werden die Grundlagen von Isabelle/HOLCF, insbesondere Stetigkeit von HOLCF-Funktionen sowie stromverarbeitende HOLCF-Funktionen behandelt.

**Kapitel 3** Hier wird der Datentypkonstruktor `stream` aus HOLCF vorgestellt. Anhand der Definition von `stream` wird gezeigt wie Streams auf pcpo's gebildet werden. Es werden die grundlegenden Eigenschaften auf der Basis des Konstruktionsoperators und

das Take-Lemma als das grundlegende Beweisprinzip für Streams vorgestellt. Ergänzend und abschließend werden hier weitere Basiseigenschaften abgeleitet.

**Kapitel 4** Auf der Grundlage des Datentypkonstruktors `stream` wird in diesem Kapitel der abgeleitete Konstruktor `fstream` eingeführt. Anschließend wird das für weitere Definitionen nützliche Take-Funktional auf Fstreams übertragen. Im Weiteren wird die Konkatenation als die zentrale Konstruktionsoperation für Fstreams aufgebaut. Ferner werden wesentliche Funktionen auf Fstreams definiert wie etwa die Längenfunktion. Im Weiteren werden die Induktionsregeln für Fstreams geschaffen. Abschließend werden weitere charakteristische Eigenschaften von hier eingeführten Konstrukten behandelt.

**Kapitel 5** Anhand des Konkatenationsoperators wird hier ein kompaktes Definitionsmuster beschrieben, um in einfacher Weise rekursive Funktionen auf Fstreams zu definieren. Mit diesem Definitionsmuster können beliebige rekursive Funktionen auf Fstreams, insbesondere auch die stromverarbeitenden Funktionen, systematisch und elegant definiert werden. Ferner wird der HOLCF-Ansatz für die Definition von automatisch erkennbaren stetigen HOLCF-Funktionen auf der Basis von bereits vorhandenen stetigen Funktionen und des Fixpunktoperators vorgestellt.

**Kapitel 6** In diesem Kapitel wird eine grundlegende Bibliothek von rekursiven und stromverarbeitenden Funktionen auf der Basis des Typkonstruktors `fstream` definiert und die jeweiligen charakterisierenden Entfaltungslemmata dieser Funktionen eingeführt. Durch die Komposition dieser Funktionen lassen sich weitere rekursive bzw. stromverarbeitende Funktionen, wie sie typischerweise bei der strombasierten Verifikation vorkommen, noch einfacher einführen.

**Kapitel 7** Der Bericht schließt mit einer Zusammenfassung und Diskussion ab.

## 1.2 Lesehinweise

Bei dieser Umsetzung wird die Isabelle-Version *Isabelle2005* eingesetzt, so wie sie auf der offiziellen Webseite [Mün06] des Theorembeweisers zum Zeitpunkt der Erstellung dieser Arbeit zur Verfügung stand.

In den meisten Lemmata wird auf die Darstellung des äusseren universellen Quantors (Allquantor der Metalogik, symbolisch $\bigwedge$) aus Übersichtlichkeitsgründen verzichtet. Auf diese Art der Variablenbindung wird auch in der umliegenden textuellen Beschreibung solcher Lemmata nicht eingegangen. Alle freien Variablen werden in Isabelle automatisch so behandelt und sind beim Lesen des Berichts leicht zu berücksichtigen.

Alle literaturüblichen, mathematischen oder informellen Beschreibungen als auch die Isabelle-Quellcode-Ausschnitte in dieser Arbeit werden durch eine Marke an der rechten Seite der Darstellung eines solchen Konstrukts abgeschlossen. Sie ist strukturiert in eine Bezeichnung **B**, **Def**, **TS**, **N** oder **S** deren Bedeutung unten erklärt wird und einer Nummerierung in Form von c.s.n für Kapitel (c), Abschnitt (s) und die laufende Nummer (n).

(**B** c.s.n) Bezeichnet das Ende eines hier vollständig dargestellten Beweises.

(**Def** c.s.n) Bezeichnet das Ende einer hier dargestellten Definition einer Funktion in Isabelle inklusive der dazugehörigen Funktionstypvereinbarung. Die syntaktisch notwendigen Schlüsselwörter `constdefs` vor den üblichen Definitionen bzw. `consts` vor den Funktionstypvereinbarungen auf der Basis des `primrec`-Konstrukts (siehe [NPW02] ) werden hier aus Übersichtlichkeitsgründen ausgelassen.

(`lemma`) Schließt ein links stehendes Lemma (bzw. einen Satz in der Isabelle-Syntax ohne den zugehörigen Beweis[1]) ab, das den Namen `lemma` trägt. Wir halten uns bei der Benennung von den meisten hier eingeführten Lemmata an bestimmte Vereinbarungen, die im Folgenden beschrieben werden. Falls der Name mit dem Suffix `_simp` endet, wird dieses Lemma dem Simplifikator hinzugefügt bzw. es befindet sich bereits in der Menge der Simplifikationsregeln. Endet der Name mit dem Suffix `_simps` dann handelt es sich dabei um einen Lemmatablock, der in den Simplifikator hinzugefügt wird, falls nicht geschehen. Die einzelnen Namen der jeweiligen Lemmata aus einem solchen Lemmatablock tragen hier nicht weiter zum Verständniss bei, sodass sie der Übersichtlichkeit wegen hier ausgelassen wurden. Insbesondere wird die Anwendung solcher Lemmata in den Beweisen vom Simplifikator übernommen. Endet der Name andererseits mit `_unfolds`, so bezeichnet dieser Name mehrere Lemmata, die der Aufschreibungsreihenfolge nach in den jeweiligen Namen eine Nummerierung statt dem letzten Zeichen `s` im Namenssuffix `_unfolds` erhalten können.

(**N** c.s.n) Nicht-Isabelle Syntax, wie z.B. Sachverhalte in der allgemeinen, literaturüblichen Notation.

(**S** c.s.n) Sonstige Isabelle-Syntax, wie z.B. Simplifikatoranweisungen usw.

(**TS** c.s.n) Isabelle-Syntax für Typvereinbarung bzw. Typanweisungen.

## 1.3 Eine kurze Begriffs- und Notationsübersicht

Um die vorliegende Arbeit von Anfang an verständlicher und übersichtlicher zu machen stellen wir hier eine kurze Begriffs- und Notationseinführung der in dieser Arbeit behandelten Konzepte dar:

`stream` Der HOLCF-Datentypkonstruktor für partielle oder unendliche Sequenzen auf beliebigen Typen der `pcpo`-Typklasse (deshalb auch allgemeine Streams bzw. engl. general stream domains im Unterschied zum deutschen „Ströme“). Die Instanzen von

[1] In Isabelle wird mit Lemma ein Satz inklusive des zugehörigen Beweises bezeichnet.

`stream` nennen wir Streams. Er ist definitorisch in HOLCF vorhanden.

**Ströme** — Die Ströme im Sinne der Focus-Entwicklungsmethodik ([BS01]). In Focus modellieren die Ströme den an einem Kanalende beobachtbaren gerichteten Zeichen- bzw. Nachrichtenfluss ohne Berücksichtigung der Zeit, wobei die Kanäle der Kommunikation zwischen Systemkomponenten dienen und unidirektional sind.

**`fstream`** — Der HOLCF-Datentypkonstruktor für Ströme, der in dieser Arbeit eingeführt wird. Die Instanzen von `fstream` nennen wir Fstreams, Ströme oder potentiell unendliche Sequenzen.

**HOLCF-Funktion** — Eine stetige (Abschnitt 2.3) HOL-Funktion oder eine Funktion, deren Typ durch die Verwendung des HOLCF-Funktionstypkonstruktors ($\rightarrow$) in HOLCF eingeführt wurde. (das Konzept von HOLCF-Funktionen wird ausführlich in Kapitel 2 behandelt)

**Operation** — Eine HOLCF-Funktion, deren Werte- und Definitionsbereich jeweils Typen der HOLCF-Typklasse `pcpo` sind.

**cpo** — Die Bezeichnung für eine vollständige partielle Ordnungsrelation oder auch für eine Menge mit einer impliziten, vollständigen partiellen Ordnungsrelation (engl. **c**omplete **p**artial **o**rder). In HOLCF ist `cpo` das Schlüsselwort für die dementsprechende HOLCF-Typklasse. In dieser Arbeit nennen wir die Typen dieser Typklasse auch kurz cpo's. Sowohl nach der obigen Definition als auch in HOLCF enthalten cpo's nicht notwendigerweise auch ein kleinstes Element.

**pcpo** — Die Bezeichnung für eine vollständige partielle Ordnung mit kleinstem Element (engl. a **p**ointed cpo). Das kleinste Element wird allgemein mit $\bot$ symbolisiert. In HOLCF ist `pcpo` das Schlüsselwort für die dementsprechende HOLCF-Typklasse. In dieser Arbeit nennen wir die Typen dieser Typklasse auch kurz pcpo's.

**$\sqsubseteq^c_X$** — Unsere Notation für eine binäre, vollständige ($c$ für complete) partielle Ordnungsrelation ($\sqsubseteq$) mit der Trägermenge $X$ (siehe auch cpo). $X$ entfällt, wenn aus dem Kontext erschließbar.

**$X^{\sqsubseteq^c}$** — Die Menge $X$ ist durch die Relation $\sqsubseteq^c$ vollständig partiell geordnet.

**$X \xrightarrow{s} Y$** — Notation für die Spezifikation der Signatur einer stetigen (siehe Abschnitt 2.3) Funktion mit dem Definitionsbereich $X$ und dem Wertebereich $Y$. Dabei sind die Mengen $X$ und $Y$ cpo's.

| | |
|---|---|
| stromverarbeitende Funktion | Eine stromverarbeitende Funktion modelliert das Verhalten einer interaktiven, deterministischen Systemkomponente in Focus. Eine einkanälige, stromverarbeitende Funktion $f$ besitzt die Signatur: $f : M_{in}^{\omega} \xrightarrow{s} M_{out}^{\omega}$. Die Mengen $M_{in}$ bzw. $M_{out}$ sind die Mengen aller Zeichen, die die Funktion bzw. Komponente einlesen bzw. ausgeben kann. Der Definitions- und Wertebereich einer stromverarbeitenden Funktion sind bezüglich der Anfangsstück-Relation auf Strömen pcpo's. Formal ist eine stromverarbeitende Funktion eine HOLCF-Funktion des HOLCF-Funktionstyps `'a fstream` $\rightarrow$ `'b fstream` wobei die Typ-Parameter `'a` bzw. `'b` die jeweiligen Eingabe- bzw. Ausgabezeichentypen modellieren. |
| $X^*$ | Menge aller endlichen Sequenzen auf der Menge $X$. |
| $X^{\infty}$ | Menge aller unendlichen Sequenzen auf der Menge $X$. |
| $X^{\omega}$ | Menge aller potentiell unendlichen Sequenzen bzw. Ströme auf der Menge $X$. Es gilt: $X^{\omega} = X^* \cup X^{\infty}$ |

# 2 Isabelle

In diesem Kapitel werden Isabelle und die beiden Instanzen Isabelle/HOL und Isabelle/HOLCF kurz vorgestellt. Insbesondere wird das Zusammenwirken beider Logiken näher beleuchtet. Es werden die Grundlagen von Isabelle/HOLCF, insbesondere Stetigkeit von HOLCF-Funktionen sowie stromverarbeitende HOLCF-Funktionen behandelt.

## 2.1 Isabelle/HOL und Isabelle/HOLCF in Kürze

Isabelle [Mün06, Pau03a, Pau03b, NPW02, Pau94] ist ein generisches Logiksystem bzw. Theorembeweiser. Mit Isabelle/HOL [NPW02, Mün06] , im Weiteren einfach HOL, wird die umfangreiche Isabelle-Ausprägung der Prädikatenlogik höherer Stufe (Higher Order Logic) bezeichnet. Eine gute Einführung in die mathematische Logik bieten [Sch00, Fit96, EFT96, And02] . Isabelle/HOLCF [Mün06, Reg94, Reg95, MNvOS99, Pau87, Plo77] (im Weiteren kurz HOLCF) ist eine konservative Erweiterung von HOL um bereichstheoretische Konstrukte u.a. für die Darstellung von Ketten, Stetigkeit und Zulässigkeit sowie um einen Fixpunktoperator.

In HOL werden Terme durch Anwendung von Funktionen auf Argumente gebildet. Die Termsyntax ist weitgehend an die herkömmliche mathematische Syntax und die funktionalen programmiersprachlichen Konstrukte (siehe z.B. [Tho99, Bir98, Pau96] ) angelehnt. Alle Terme des Typs `bool` heißen Formeln. Die gängigsten Basistypen in HOL sind `bool` und `nat`. `set` ist einer der Basistypkonstruktoren und dient zur Bildung von Mengen. Funktionstypen werden mit dem rechtsassoziativen Typkonstruktor $\Rightarrow$ eingeführt und dienen der Konstruktion von totalen Funktionen. Komplexe Typen können mit Hilfe von Typvariablen, Basistypen und Typkonstruktoren konstruiert werden. HOL übernimmt das Typsystem von der Metalogik in Isabelle, sodass die Typinferenz für Variablen automatisch erfolgt. [NPW02] bietet eine sehr gute Einführung in HOL. Darin sind auch die Unterschiede zwischen Objektlogik und Metalogik erklärt. So sind z.B. zwei Arten von Implikation vorhanden: Die objektlogische Implikation $\longrightarrow$ und die metalogische Inferenz $\Longrightarrow$. Der Unterschied kann hier beim zügigen Durchlesen vernachlässigt werden. Ferner gibt es auch die objektlogische Gleichheit $=$ und die metalogische, definitorische Gleichheit $\equiv$. Die Prämissen einer Inferenzregel werden in $[\![\ ]\!]$ eingeschloßen und durch ; getrennt. Der universelle Quantor wird mit $\bigwedge$ symbolisiert.

HOLCF erweitert HOL durch die zentrale Typklasse `pcpo`, die für jede Instanz das Relationssymbol $\sqsubseteq$, entsprechend einer partiellen Ordnungsrelation, die Existenz der kleinsten oberen Schranke für jede Kette (siehe Abschnitt 2.2) und das kleinste Element (symbolisch $\bot$) axiomatisch fordert. Die Erweiterung erfolgt schrittweise auf der Basis der HOLCF-Theorie *Porder*, in der zuerst die Typklasse `po` für partiell geordnete Typen eingeführt wird. Diese Typklasse wird anschließend durch die Forderung der Existenz der kleinsten oberen Schranke für jede Kette zu der Typklasse `cpo` in der HOLCF-Theorie *Pcpo* weiter angereichert. In

diesem Bericht werden wir im Weiteren für die Typen der `cpo`-Typklasse die allgemeine Bezeichnung HOLCF-Typ verwenden, soweit keine weiteren Details notwendig sind. Schließlich wird in der Theorie *Pcpo* die Typklasse `pcpo` eingeführt, die ergänzend zu den vorherigen `cpo`-Anforderungen für jede Instanz die Existenz des kleinsten Elements fordert. Die HOLCF-Fixpunkttheorie *Fix* nutzt die Typklasse *pcpo* als Grundlage und implementiert im Wesentlichen den Fixpunktoperator (`fix`) (siehe Abschnitt 5.3).

Durch die Lambda-Abstraktion wird in HOL der Umgang mit Funktionsnamen bzw. Funktionen analog den funktionalen Programmiersprachen gehandhabt. Das folgende Beispiel der Nachfolgerfunktion auf natürlichen Zahlen in der literaturüblichen Notation spiegelt die zentrale Rolle des Lambda-Operators ($\lambda$) wieder:

$$Suc(x) = x + 1 \quad \stackrel{!}{=} \quad Suc = \lambda x.\ (x + 1) \qquad (\mathbf{N}\ 2.1.0)$$

Mit dem Zeichen $\stackrel{!}{=}$ symbolisieren wir die informelle Äquivalenz zweier Konzepte. Durch den $\lambda$-Operator kann also ein Name (Funktionsname bzw. Bezeichner) an eine Rechenvorschrift gebunden werden bzw. je nach Bedarf entweder der Name (symbolisch) oder die Rechenvorschrift (auswertungsorientiert) verwendet werden. Zwei Funktionen sind identisch falls die jeweiligen Rechenvorschriften für gleiche Argumente gleiche Ergebnisse liefern. Dank der Lambda-Abstraktion und der automatischen Typinferenz in Isabelle können elegant und dynamisch neue, namenslose Funktionen spezifiziert werden (z.B. direkt in Beweiskommandos). Die Lambda-Abstraktion in HOL wird durch den Operator $\lambda$ ermöglicht. Die oben erwähnte Funktionsgleichheit wird in HOL durch die folgende Regel (das Extensionalitätsprinzip für HOL-Funktionen) festgelegt:

$$(\bigwedge \texttt{x.}\ \texttt{f x} = \texttt{g x}) \implies \texttt{f} = \texttt{g} \qquad (\texttt{ext})$$

HOLCF stellt im Gegensatz dazu einen eigenen Typkonstruktor ($\rightarrow$) für die Konstruktion von stetigen Funktionen auf HOLCF-Typen bereit, die wir hier kurz HOLCF-Funktionen nennen. Das Konzept der Stetigkeit einer Funktion $f$ (`cont f`) wird in Abschnitt 2.3 erklärt. Für den Einsatz dieses Funktionstypkonstruktors werden zusätzlich eine eigene Lambda-Abstraktion ($\Lambda$ - der HOLCF-Operator für die HOLCF-Lambda-Abstraktion) und dementsprechend die eigene, stetige Funktionsanwendung („$\cdot$“- der HOLCF-Operator für die HOLCF-Funktionsanwendung) eingeführt. HOLCF stellt ergänzend eine eigene Definitionssyntax für HOLCF-Funktionen, wie etwa den Fixpunktoperator, zur Verfügung. Auf die Definitionsprinzipien wird ausführlich in Kapitel 5 eingegangen. Ferner werden einige HOLCF-Funktionen in Kapitel 6 eingeführt.

Die HOL-Charakterisierung der obigen HOLCF-Operatoren erfolgt als $\beta$-Reduktion für stetige HOL-Funktionen in HOLCF mit folgendem Satz:

$$\texttt{cont f} \implies \texttt{(}\Lambda\texttt{x. f x)}\cdot\texttt{a} = \texttt{(}\lambda\texttt{x. f x) a} = \texttt{f a} \qquad (\texttt{beta\_cfun})$$

Ist $f$ nicht stetig, so ist die Auswertung von `(`$\Lambda$`x. f x)`$\cdot$`a` nicht möglich und damit auch keine sinnvolle Verwendung im Sinne dieser Operatoren (Der Operator $\Lambda$ ist auf beliebige HOL-Funktionen auf cpo's anwendbar).

Im Prinzip wird also die Stetigkeit von $f$ in der Regel `beta_cfun` durch den Funktionstypkonstruktor $\rightarrow$ bzw. den Operator $\Lambda$ in dem zugehörigen HOLCF-Funktionstyp gekapselt. HOLCF bietet also eine syntaktische Unterstützung beim Umgang mit stetigen Funktionen.

Die syntaktisch erzwungene Trennung von HOL-Funktionsklassen ermöglicht einen effizienten Einsatz des Simplifikators, aber auch eine übersichtlichere Beweisführung seitens des Anwenders sowie einen wesentlich übersichtlicheren Umgang mit zusammengesetzten stetigen Funktionen.

Analog der obigen HOL-Regel `(ext)` sieht die Introduktionsregel für die Gleichheit zweier HOLCF-Funktionen in HOLCF wie folgt aus:

$$(\bigwedge \texttt{x. f·x = g·x}) \implies \texttt{f = g} \qquad (\texttt{ext\_cfun})$$

Jede stetige HOL-Funktion kann also entsprechend dem Lemma `beta_cfun` durch die Vertauschung von Lambda-Operatoren (also $\Lambda$ für $\lambda$), durch die entsprechende Funktionstypkonversion ($\rightarrow$ für $\Rightarrow$) und der anschließenden Beachtung der Funksionsanwendungssyntax von HOLCF-Funktionen ($\cdot$) an allen Anwendungsstellen in eine äquivalente HOLCF-Funktion transformiert werden. Dies ist meistens dann der Fall, wenn die Funktion zunächst als eine HOL-Funktion definiert und deren Stetigkeit anschließend durch den Benutzer bewiesen wurde. Ergänzend gibt es für jede HOLCF-Funktion eine äquivalente stetige HOL-Funktion [Reg94, Reg95, MNvOS99]. Entsprechend dieser Äquivalenz können wir ohne Einschränkung der Allgemeinheit jede stetige HOL-Funktion auch als eine HOLCF-Funktion bezeichnen, bzw.:

$$\text{Menge aller stetigen HOL-Funktionen} \stackrel{!}{=} \text{Menge aller HOLCF-Funktionen} \qquad (\mathbf{N}\ 2.1.1)$$

## 2.2 Isabelle/HOLCF im Überblick

Die Isabelle-Logik HOLCF ist eine Formalisierung der Scottschen Bereichstheorie (engl. Domain Theory), die in [Sco82, Sco76] als eine formale Infrastruktur für die Beschreibung der denotationellen Semantik von Programmiersprachen eingeführt wurde. Ausführliche Einführungen in die Bereichstheorie sowie die denotationelle Semantik finden sich in [Win93, Gun92, LS84] . Ferner bietet [DP02] eine gute Einführung in die ordnungstheoretischen Konzepte. In diesem Abschnitt soll nur die Notation für die später verwendeten Konstrukte eingeführt und auf einige realisierungsbezogene Besonderheiten kurz eingegangen werden.

Eines der zentralen Konzepte in HOLCF ist das Konzept der abzählbaren Ketten. Eine abzählbare Kette $K$ ist eine geordnete Teilmenge[1] $K = (C_K, O_K)$ einer partiell geordneten Menge $M$ mit $M = (C_M, O_M)$, deren Trägermenge $C_K$ abzählbar und die korrespondierende Ordnung $O_K$ bezüglich der Trägermenge total sind.

Gewöhnlich werden in HOLCF die Elemente einer Kette auf einem HOLCF-Datentyp (bzw. einem allgemeineren Datentyp der `po`-Typklasse) punktweise durch eine mit natürlichen Zahlen parametrisierte Funktion erzeugt. Durch die lineare Ordnung der natürlichen Zah-

[1] Eine geordnete Menge $M$ wird als ein geordnetes Paar $(C_M, O_M)$ definiert, wobei $C_M$ eine Trägermenge und $O_M$ eine Ordnungsrelation auf der Trägermenge darstellen. Eine Teilmenge einer geordneten Menge $M$ mit $M = (C_M, O_M)$ ist jede geordnete Menge $B = (C_B, O_B)$ für die gilt: $C_B \subseteq C_M$ und $O_B \subseteq O_M$. Ferner gilt $(a, b) \in O_B \implies a \in C_B \wedge b \in C_B$. Falls die Trägermenge einer geordneten Menge durch die korrespondierende Ordnungsrelation total geordnet wird, so kann von ihr abstrahiert werden.

len im erwähnten Parameter wird auch die Ordnung der Kettenelemente impliziert. Somit stellt eine solche Funktion eine äquivalente Darstellungsform zu der obigen mathematischen Definition einer Kette dar. Im Weiteren werden wir solche Funktionen dementsprechend auch als Ketten bezeichnen. Die Ketteneigenschaft als auch das Wesentliche der HOLCF-Kettendarstellung wird hier anhand der Introduktionsregel für die Ketteneigenschaft aus der HOLCF-Theorie *Porder* veranschaulicht:

`(⋀i. K i ⊑ K (Suc i)) ⟹ chain K` (`chainI`)

Die Umkehrung der obigen Implikation wird im Lemma `chainE` festgehalten. Für die Erzeugung von Kettenelementen werden oft auch passende Funktionale herangezogen. Ein Funktional ist eine Bezeichnung für eine Funktion die Funktionen als Argument(e) bzw. als Ergebnis haben. So ist z.B. `iterate` (bzw. `iterate n`) ein typisches Funktional in HOLCF des Funktionstyps `nat ⇒ ('a → 'a) ⇒ 'a ⇒ 'a`. `iterate` wendet $n$-Mal eine HOLCF-Funktion $f$ des Typs `'a → 'a` auf einen Wert des Typs `'a` an und liefert als Ergebnis einen dementsprechenden Wert des Typs `'a` zurück.

Die obige Definition von Ketten schließt insbesondere auch die Ketten von Funktionen auf HOLCF-Typen ein, weil die Ordnungsrelation für Funktionen in HOLCF nämlich durch die folgende Regel festgelegt wird:

`(⋀x. f x ⊑ g x) ⟹ f ⊑ g` (`less_fun_ext`)

Somit sind Funktionen und die Funktionswerte in Ketten „vertauschbar“. In cpo's besitzt jede Kette im obigen Sinne eine kleinste obere Schranke als das kleinste Element der Menge aller oberen Schranken der Kette. Für eine Kette $K$ wird sie in HOLCF mit `λx. ⨆i. K i x` spezifiziert, wobei der formale Parameter `x` die Basis für die Konstruktion der jeweiligen Kettenelemente (als Werte) darstellt.

Die Zulässigkeit (`adm`) wurde in der HOLCF-Theorie *Adm* formalisiert und der Fixpunktoperator (`fix`) in *Fix*. Die Zulässigkeit wird näher in Kapitel 4.6 und der Fixpunktoperator in Kapitel 5.3 beschrieben. Eine grundlegende Einführung der erwähnten bereichstheoretischen Begriffe im Kontext der Verifikation findet sich in [LS84]. Eine weitere Einführung ist in [Win93] zu finden.

Um HOLCF-Funktionalitäten auf HOL-Typen anwenden zu können, wird von HOLCF als Schnittstelle die Theorie *Lift* zur Verfügung gestellt. Der Typkonstuktor `lift` erweitert beliebige Typen insbesondere HOL-Typen zu flachen HOLCF-Typen (flachen cpo's, siehe Abschnitt 4.1) und macht sie damit für HOLCF zugänglich. Ein HOL-Typ `'a` wird durch das Liften zu einem HOLCF-Typ `'a lift` der neben allen Elementen von `'a` zusätzlich $\bot$ als das kleinste Element enthält. Im Weiteren verwenden wir übersichtshalber die Notation $\uparrow v$ für eine geliftete Instanz von $v$ aus dem entsprechend gelifteten HOL-Typ. Z.B. ist $\uparrow 0$ eine Instanz von `nat lift`. Die Beziehung zur HOLCF-Notation ist: $\uparrow v \equiv$ `Def v`. Ferner gilt: $\uparrow v_1 \sqsubseteq \uparrow v_2 \Longrightarrow v_1 = v_2$.

## 2.3 Stetigkeit von HOLCF-Funktionen

Eine stetige Funktion $f$ in der bereichstheoretischen bzw. literaturüblichen Notation ist eine Funktion, deren Werte- und Definitionsbereich jeweils Mengen die vollständig partiell

geordnet also cpo's sind:

$$f : D^{\sqsubseteq^c} \to W^{\sqsubseteq^c} \qquad (\mathbf{N}\ 2.3.0)$$

Diese Notation wurde bereits in Abschnitt 1.3 eingeführt. In HOL sprechen wir von Funktionstypvereinbarung anstatt von Funktionssignatur. Der HOL-Funktionstyp von $f$ wird für den HOLCF-Typ-Parameter `'d` für den Definitionsbereich und den HOLCF-Typ-Parameter `'w` für den Wertebereich von $f$ wie folgt vereinbart:

```
f  :: "'d  ⇒  'w"                                    (TS 2.3.0)
```

$f$ ist darüberhinaus bezüglich der obigen Ordnungen monoton (ordnungserhaltend), d.h.:

$$\forall x\ y.\ x \sqsubseteq^c_D y \implies f(x) \sqsubseteq^c_W f(y) \qquad (\mathbf{N}\ 2.3.1)$$

In HOLCF wird die Monotonieeigenschaft einer HOL-Funktion $f$ durch die folgende Definition (kurz bzw. ohne die begleitende Typvereinbarung!) festgelegt:

```
monofun f  ≡  ∀x y. x ⊑ y ⟶ f x ⊑ f y                (monofun)
```

Alle Konstrukte bzw. notwendigen Eigenschaften aus N 2.3.1, die in der obigen HOLCF-Definition von `monofun` nicht vorkommen, werden durch das Typklassensystem von Isabelle impliziert. Die wahrheitswertige Funktion `monofun` fordert eine Funktion auf HOLCF-Typen im Argument[2] entsprechend einer Typvereinbarung analog der in N 2.3.0. Die Monotonie sichert in Focus, dass einmal verursachte Nachrichten nicht zurück genommen werden können. Abschließend ist $f$ kompatibel mit der Konstruktion der kleinsten oberen Schranke für jede Kette[3] $K$, d.h.:

$$\forall K.\ \mathit{chain}\ K \implies f(\bigsqcup K) = \bigsqcup f(K) \qquad (\mathbf{N}\ 2.3.2)$$

In HOLCF wird die obige Eigenschaft für eine HOL-Funktion $f$ durch die folgende Definition (auch hier ohne die begleitende Typvereinbarung) festgelegt:

```
contlub f  ≡  ∀K. chain K ⟶ f (⨆i. K i) = ⨆i. f (K i)   (contlub)
```

Der Funktionstyp des Funktionals `contlub` ist dem oben beschriebenen Funktionstyp der Funktion `monofun` gleich. Die folgende Funktion auf der Menge aller Ströme über $M$ mit der Signatur $h : M^\omega \to M^\omega$ ist monoton, aber sie ist mit der Konstruktion von kleinsten oberen Schranken nicht kompatibel:

$$h(s) = \begin{cases} \varepsilon & \text{falls } s \text{ endlich} \\ s & \text{sonst} \end{cases} \qquad (\mathbf{N}\ 2.3.3)$$

Mit der Anfangsstück-Relation auf Strömen ist $M^\omega$ eine cpo im obigen Sinne. Die Funktion

[2] Sowohl der Definitionsbereich als auch der Wertebereich der Funktion im Argument von `monofun` können eigentlich Typen der HOLCF-Typklasse `po` sein, also allgemeiner als cpo's (der Typ von `monofun` in HOLCF lautet: `('a::po ⇒ 'b::po) ⇒ bool`). Dies kann in dieser Arbeit vernachlässigt werden also statt der Typklasse `po` werden wir aus Übersichtlichkeitsgründen nur den Spezialfall von `po` nämlich `cpo` betrachten.

[3] In der bereichstheoretischen Notation wird die Ketteneigenschaft für eine Kette $K$ mit *chain K* und die kleinste obere Schranke von $K$ mit $\bigsqcup K$ bezeichnet. Es wird im Gegensatz zur HOLCF-Syntax (siehe auch Abschnitt 2.2) von der zugrunde liegenden linearen Ordnung abstrahiert.

$h$ ist monoton. Sie ist z.B. nicht kompatibel mit der Konstruktion der kleinsten oberen Schranke für die Kette aller endlichen Anfangsstücke eines unendlichen Stromes aus $M^\omega$.

Die Eigenschaft `contlub` sichert bei einer Funktion, dass eine (insbesondere unendliche) Ausgabe der Funktion immer anhand der Ausgaben für eine Menge bzw. Kette der endlichen Eingaben (im Sinne des jeweiligen HOLCF-Datentyps) approximiert werden kann (die realisierbaren Systemkomponenten in Focus können nur endlich viele Speicherplätze bzw. Zeit für die Berechnung eines Zwischenergebnises zur Verfügung stellen, siehe z.B. das obige Beispiel).

Stetige Funktionen über Strömen erfassen damit die fundamentalen Charakteristika der verteilten, asynchron kommunizierenden Systeme. Unstetige Funktionen sind entsprechend den obigen Ausführungen nicht implementierbar. Zusammenfassend kann die Stetigkeit (dabei wird der Funktionstyp von $f$ durch das Isabelle-Typsystem anhand der Typen von `monofun`, `contlub` und `cont` impliziert) schließlich wie folgt in HOLCF kurz charakterisiert werden:

`cont f` $\Longleftrightarrow$ `monofun f` $\wedge$ `contlub f` (`contIFFmonoAcontlub`)

Die oben eingeführten Eigenschaften sind in verschiedenen Ausprägungen in der HOLCF-Theorie *Cont* formalisiert. So gibt es neben den obigen Definitionen die jeweiligen Introduktions- (`monofunI`, `contlubI`, `contI`) und die entsprechende Eliminationsregeln (`monofunE`, `contlubE`, `contE`) für HOL-Funktionen.

Falls eine Funktion $f$ von Anfang an stetig ist (z.B. durch die Verwendung einer geeigneten Defnitionssyntax) so wird dies literaturüblich bereits in deren Signatur wie folgt symbolisiert:

$f : D^{\sqsubseteq^c} \xrightarrow{s} W^{\sqsubseteq^c}$ (**N** 2.3.4)

Dieses Konzept entspricht dem Konzept der HOLCF-Funktionen. Entsprechend der obigen literaturüblichen Spezifikationssyntax wird der HOLCF-Funktionstypkonstruktor für die Vereinbarung des Funktionstyps einer HOLCF-Funktion $f$ wie folgt verwendet:

`f :: "'d → 'w"` (**TS** 2.3.1)

Die HOLCF-Theorie *Cfun* bietet die obige Eigenschaften angepasst an die HOLCF-Funktionen, wie z.B. das Lemma `monofun_cfun_arg` für die Monotonie einer HOLCF-Funktion oder analog `contlub_cfun_arg` usw.

Besitzt eine HOL-Funktion die Eigenschaften `monofun` und `contlub` und die oben besprochenen HOLCF-Typen im Definitions- und Wertebereich, so kann sie mit der Transformationsvorschrift aus dem Abschnitt 2.1 in eine äquivalente HOLCF-Funktion transformiert werden, nachdem die Eigenschaften `monofun` und `contlub` vom Benutzer bewiesen worden sind. Wird im Gegensatz dazu eine HOLCF-Funktion durch eine Fixpunktgleichung definiert, wobei innerhalb der Gleichung nur HOLCF-Konstrukte vorkommen, so ist sie per se stetig und darüber hinaus automatisch von HOLCF als stetig erkennbar bzw. auswertbar. Die entsprechende Definitionssyntax wird in Kapitel 5 ausführlicher beschrieben. In Kapitel 5 werden ergänzend drei weitere Definitionsmethoden für stromverarbeitende Funktionen (siehe den nächsten Abschnitt) eingeführt. Ferner wird in Kapitel 5 eine systematische Definitionsmethode für beliebige rekursive Funktionen auf HOLCF-Typen vorgestellt.

Für die Stetigkeitsbeweise führen wir ergänzend zu den obigen Eigenschaften und Lemmata in HOLCF den für die Funktionen auf cpo's wohlbekannten Sachverhalt (siehe z.B. [LS84]) ein:

```
⟦monofun (f::'a::cpo ⇒ 'b::cpo);
    (∀Y. chain Y ⟶ f (⨆i. Y i) ⊑ (⨆i. f (Y i)))⟧ ⟹ cont f
                                                        (cpo_contI)
```

Die Eigenschaft `contlub` kann also als Voraussetzung für Stetigkeit abgeschwächt werden. Sie wird im Folgenden bewiesen.

```
lemma cpo_contI:
  "⟦monofun (f::'a::cpo ⇒ 'b::cpo);
      (∀Y. chain Y ⟶ f (⨆i. Y i) ⊑ (⨆i. f (Y i)))⟧ ⟹ cont f"
apply (rule monocontlub2cont, assumption)
apply (rule contlubI)
apply (erule_tac x="Y" in allE, drule mp, assumption)
apply (subst po_eq_conv, rule conjI, assumption)
apply (frule cpo, erule exE)
apply (frule is_lubD1)
apply (frule ub2ub_monofun)
apply assumption
apply (drule thelubI)
apply (rule is_lub_thelub)
apply (drule ch2ch_monofun)
apply assumption+
apply (drule sym, erule subst)
apply assumption
done
```

(**B** 2.3.0)

Das Lemma `cpo_contI` ist insbesondere deshalb vorteilhaft, weil für die Funktionen, bei denen der Wertebereich ein `pcpo`-Typ ist, bei der Auswertung der trivialen Fälle zu $\bot$ nur die linke Seite der Formel `f (⨆i. Y i) ⊑ (⨆i. f (Y i))` in der zweiten Prämisse des Lemmas ausgewertet werden muss bzw. $\bot$ immer das kleinste Element ist. Dies ist insbesondere bei stromverarbeitenden Funktionen (siehe unten) oft der Fall.

## 2.4 Stromverarbeitende HOLCF-Funktionen

Eine stromverarbeitende Funktion, als das zentrale Konzept bei strombasierten Spezifikations- und Verifikationstechniken, ist eine stetige Funktion deren Definitionsbereich die Menge aller Ströme über einer Menge der Eingabezeichen $M_{in}$ und deren Wertebereich die Menge aller Ströme über einer Menge der Ausgabezeichen $M_{out}$ ist:

$$f : M_{in}^{\omega} \xrightarrow{s} M_{out}^{\omega} \qquad (\mathbf{N}\ 2.4.0)$$

Anhand der impliziten Anfangsstück-Relation auf Strömen sind die Definitions- und Wer-

tebereiche vollständig partiell geordnet und der leere Strom ist das kleinste Element. Die Mengen von Eingabe- bzw. Ausgabezeichen modellieren dabei die Nachrichten, die über die gerichteten Kommunikationskanäle der jeweiligen Komponenten ausgetauscht werden können. Damit modelliert eine stromverarbeitende Funktion das Verhalten einer interaktiven, deterministischen Komponente ([Rum96, BS01]), deren Eingabe und Ausgabe über die, der Komponente zugewiesenen Eingabe- und Ausgabekanäle abgewickelt werden. Vertiefende Einblicke in das Konzept der stromverarbeitenden Funktionen bieten auch [Ste97, BS96].

Für die Konstruktion der obigen Definitions- und Wertebereiche von stromverarbeitenden Funktionen auf der Basis von Nachrichtentypen bzw. Nachrichtenmengen wird der Typkonstruktor `fstream` in Kapitel 4 eingeführt. Damit wird es möglich den Typ einer stromverarbeitenden HOLCF-Funktion $f$ in HOLCF wie folgt zu vereinbaren:

```
f  :: "'i fstream  →  'o fstream"                    (TS 2.4.0)
```

wobei der Typ-Parameter `'i` für den Typ der Zeichen in der Menge $M_{in}$ und `'o` analog für den Typ der Zeichen in $M_{out}$ stehen. Falls die Mengen $M_{in}$ bzw. $M_{out}$ mit dem jeweiligen Typ äquivalent sind, dann gibt es auch keinen Unterschied zwischen dem jeweiligen Typ und der jeweiligen Zeichenmenge, sonst muss eine stromverarbeitende Funktion den Unterschied entsprechend berücksichtigen. Im Gegensatz dazu wird die folgende Typvereinbarungssyntax für allgemeinere rekursive Funktionen (z.B. die nicht monotonen) auf Fstreams verwendet:

```
f  :: "'a fstream  ⇒  'b fstream"                    (TS 2.4.1)
```

# 3 Allgemeine Streams in HOLCF

Hier wird der Datentypkonstruktor `stream` aus HOLCF vorgestellt. Anhand der Definition von `stream` wird gezeigt wie Streams auf pcpo's gebildet werden. Es werden die grundlegenden Eigenschaften auf der Basis des Konstruktionsoperators und das Take-Lemma als das grundlegende Beweisprinzip für Streams vorgestellt. Ergänzend und abschließend werden hier weitere Basiseigenschaften abgeleitet.

## 3.1 Konstruktion von Streams

Die folgende wohlbekannte Definition stellt den allgemeinen Datentypkonstruktor `stream` mit Hilfe des `domain`-Konstrukts in HOLCF zur Verfügung:

```
domain 'a stream = "&&" (ft::'a) (lazy rt::'a stream)          (TS 3.1.0)
```

Mit dieser Gleichung werden für den polymorphen Datentypkonstruktor `stream` die Symbole $\bot$ und `&&` als Konstruktoren und `ft` sowie `rt` als Selektoren eingeführt. Die Definition ist rekursiv: Ein nichtleerer Stream auf dem Typ `'a` wird durch den Konstruktor `&&` erzeugt, indem ein neues, von $\bot$ verschiedenes, Element `ft` des Typs `'a` zu einem existierenden Stream `rt` aus Elementen desselben Typs am linken Ende hinzugefügt wird. Die unendlichen Streams sind anhand der Axiome der `pcpo`-Typklasse als kleinste obere Schranken von Ketten auf Streams axiomatisch inbegriffen. Sie können dementsprechend deskriptiv oder konstruktiv durch die Anwendung des $\bigsqcup$-Operators auf eine passende Kette (z.B. die Menge aller endlichen Anfangsstücke des gewünschten Streams, die konstruiert werden kann) spezifiziert werden.

Zusätzlich wird für den Konstruktor `&&` die übliche Infixnotation vereinbart, was in der obigen Definition übersichtshalber ausgelassen wurde. `'a` ist eine Typvariable, die für einen beliebigen HOLCF-Typ der Klasse `pcpo` steht. Der Konstruktionsoperator `&&` ist strikt im ersten und lazy im zweiten Argument. Wegen der Striktheit im ersten Argument kollabiert ein nichtleerer Stream zu $\bot$ sobald ihm $\bot$ als Element hinzugefügt wird. Der Konstruktionsoperator ist rechtsassoziativ. Die Elemente von `'a stream` werden üblicherweise wie folgt interpretiert:

| | | |
|---|---|---|
| $\bot$ | : | der leere Stream |
| $x_0$ `&&` $x_1$ `&&` ... `&&` $\bot$ | : | ein nichtleerer, partieller Stream von Elementen des Typs `'a` wobei $x_0 \neq \bot \wedge \exists n > 0.\ x_n = \bot$ |
| $x_0$ `&&` $x_1$ `&&` ... | : | ein unendlicher Stream wobei gilt: $\forall n.\ x_n \neq \bot$ |

Der leere Stream sowie die restlichen partiellen Streams aus der obigen Interpretation werden bei der Modellierung von verteilten, asynchron kommunizierenden Systemen ohne Be-

rücksichtigung der Zeit nichts weiter als partielle oder vollständige Beobachtungen eines Kanalgeschehens darstellen.

Der Konstruktionsoperator ist dank dem `domain`-Konstrukt eine HOLCF-Funktion mit der internen Signatur, die abgesehen von Notationsvereinbarungen wie folgt veranschaulicht werden kann:

```
&& :: "'a → 'a stream → 'a stream"
```
(**TS** 3.1.1)

Die obige Datentypdefinition stellt ergänzend zur Stetigkeit des Konstruktionsoperators (`&&`) die üblichen Lemmata, wie z.B. die Fallunterscheidung, Striktheitseigenschaften, Injektivität, Gleichheitsrelation und die Entfaltungslemmata für den Kopf- und den Restoperator für den Datentypkonstruktor `stream` bereit.

Zusätzlich wird mit der Definition automatisch die Funktion `stream_take` zur Selektion des höchstens $n$-elementigen Anfangsstücks eines Streams eingeführt. Sie ist aufgrund des internen Aufbaus, der bei dieser Umsetzung ohne weiteres vernachlässigt werden kann, ein Beispiel eines Funktionals für die in Abschnitt 2.2 besprochene Kettenbildung. Alle angesprochenen bzw. für die Umsetzung notwendigen Eigenschaften von Streams werden in diesem Kapitel beschrieben.

## 3.2 Basiseigenschaften

Sowohl die Striktheitseigenschaft des Konstruktionsoperators als auch die Entfaltungslemmata für den Kopf- und den Restoperator sind die grundlegende Eigenschaften die auf der Basis der obigen Definition zusammengefasst unter dem Namen `stream.rews` von HOLCF zur Verfügung gestellt werden. `stream.rews` beinhaltet ferner die Charakterisierung des Take-Funktionals, die im nächsten Abschnitt beschrieben wird.

Die folgenden Regeln beschreiben die Striktheit des Konstruktionsoperators im ersten Argument:

$$\left.\begin{array}{l} \bot \texttt{ \&\& s} = \bot \\ \texttt{a} \neq \bot \implies \texttt{a \&\& s} \neq \bot \end{array}\right\} \quad (\texttt{stream.rews})$$

Die Entfaltungsregeln für den Kopfoperator werden in den folgenden Lemmata dargestellt:

$$\left.\begin{array}{l} \texttt{ft}\cdot\bot = \bot \\ \texttt{ft}\cdot\texttt{(a \&\& s)} = \texttt{a} \\ \texttt{x} \neq \bot \implies \texttt{ft}\cdot\texttt{x} \neq \bot \end{array}\right\} \quad (\texttt{stream.rews})$$

Die Entfaltungsregeln für den Restoperator werden in den folgenden Lemmata dargestellt:

$$\left.\begin{array}{l} \texttt{rt}\cdot\bot = \bot \\ \texttt{a} \neq \bot \implies \texttt{rt}\cdot\texttt{(a \&\& s)} = \texttt{s} \end{array}\right\} \quad (\texttt{stream.rews})$$

Mit der `declare`-Anweisung aus HOL fügen wir die obigen Regeln dem Simplifikator wie folgt hinzu:

```
declare stream.rews [simp add]                                    (S 3.2.0)
```

Nach der obigen Anweisung sind die vorhergehenden Lemmata dem Simplifikator bekannt und müssen deshalb in den darauf aufbauenden Beweisen bzw. Theorien nicht explizit berücksichigt werden. Deshalb wurden hier aus Übersichtlichkeitsgründen die jeweiligen Namen ausgelassen.

Das nachfolgende Lemma mit dem Namen `stream.inverts` charakterisiert die $\sqsubseteq$-Relation auf Streams. Bei Streams auf flachen HOLCF-Typen wird sie zur Anfangsstück-Relation (siehe auch Abschnitt 4.4).

$$[\![ \texttt{x \&\& xs} \sqsubseteq \texttt{y \&\& ys;}\ \texttt{x} \neq \bot ]\!] \Longrightarrow \texttt{x} \sqsubseteq \texttt{y} \wedge \texttt{xs} \sqsubseteq \texttt{ys} \qquad (\texttt{stream.inverts})$$

Wenn im vorhergehenden Satz das Symbol $\sqsubseteq$ durch das Gleichheitszeichen ersetzt wird, so entsteht dadurch das interne `stream.injects` Lemma, das die Gleichheitsrelation auf Streams charakterisiert:

$$[\![ \texttt{x \&\& xs = y \&\& ys;}\ \texttt{x} \neq \bot ]\!] \Longrightarrow \texttt{x = y} \wedge \texttt{xs = ys} \qquad (\texttt{stream.injects})$$

Abschließend wird die Struktur von Streams durch das Lemma `stream.exhaust` beschrieben:

$$\bigwedge \texttt{s. s} = \bot \vee (\exists \texttt{x xs. s = x \&\& xs} \wedge \texttt{x} \neq \bot) \qquad (\texttt{stream.exhaust})$$

## 3.3 Take-Funktional und Take-Lemma

In der oben besprochenen Menge der Basiseigenschaften, die mit der Definition unter dem Namen `stream.rews` entsteht, sind auch die Charakterisierungslemmata für das Take-Funktional vorhanden. Die einfachen Fälle bei denen eines der Argumente trivial ist werden durch die zwei folgenden Lemmata beschrieben:

$$\left.\begin{array}{l} \texttt{stream\_take 0·x} = \bot \\ \texttt{stream\_take n·}\bot = \bot \end{array}\right\} \qquad (\texttt{stream.rews})$$

Die Entfaltung des Take-Funktionals im Fall der nichttrivialen Argumente beschreibt das folgende Lemma:

$$\texttt{stream\_take (Suc n)·(x\&\&xs) = x \&\& stream\_take n·xs} \qquad (\texttt{stream.rews})$$

Für jeden HOLCF-Datentypkonstruktor der mit dem `domain`-Konstrukt eingeführt wurde, wird ein Take-Lemma direkt per Definition geliefert. Bei Streams ist es:

$$(\bigwedge \texttt{n. stream\_take n·s1 = stream\_take n·s2}) \Longrightarrow \texttt{s1 = s2} \qquad (\texttt{take\_lemmas})$$

Sind alle endlichen Anfangsstücke gleicher Länge zweier Streams gleich, so sind diese Streams identisch. Mit diesem Lemma wird die Gleichheit zweier Streams auf den Beweis der Gleichheit all deren endlichen Anfangsstücke gleicher Länge per Induktion über die natürli-

chen Zahlen zurückgeführt. Das Lemma realisiert die Grundannahme, dass sich ein unendlicher Stream eindeutig durch die Menge aller seinen endlichen Anfangstücke darstellen lässt.

## 3.4 Weitere abgeleitete Basiseigenschaften

Aus dem Lemma `stream.exhaust` (Abschnitt 3.2) leiten wir hier ergänzend das Lemma `scase` ab, das bei dieser Umsetzung für die Fallunterscheidungen bezüglich der Konstruktion nützlich sein wird:

$$\texttt{s} \neq \bot \Longrightarrow \exists \texttt{x xs. s = x \&\& xs} \wedge \texttt{x} \neq \bot \qquad (\texttt{scase})$$

Die grundlegende Eigenschaft des Take-Funktionals ist es, dass es die Ketteneigenschaft bzw. das Lemma `chainI` (Abschnitt 2.2) erfüllt:

$$\forall \texttt{s. stream\_take i·s} \sqsubseteq \texttt{stream\_take (Suc i)·s} \qquad (\texttt{chain\_take\_lemmas})$$

Das Lemma wird wie folgt bewiesen:

```
lemma chain_take_lemmas:
  "∀s. stream_take i·s ⊑ stream_take (Suc i)·s"
apply (induct_tac i, auto)
apply (case_tac "s=⊥", auto, drule scase, auto)
by (rule monofun_cfun_arg, auto)
```

(**B** 3.4.0)

In der ersten Beweiszeile wird die natürliche Induktion auf $i$ aufgerufen. Der Induktionsanfang wird automatisch durch den Simplifikator bewiesen (`auto`). Danach wird in der zweiten Zeile eine Fallunterscheidung auf $s$ mit der HOL-Anweisung `case_tac` vorgenommen. Falls $s = \bot$ gilt, wird dieser Fall vom Simplifikator gehandhabt. Für einen zusammengesetzten Stream verwenden wir das obige Lemma `scase` und anschließend wieder den Simplifikator. In beiden Fällen verwendet der Simplifikator die Eigenschaften des Take-Funktionals aus dem Abschnitt 3.3. In der letzten Zeile nutzen wir schließlich die Monotonieeigenschaft des Konstruktionsoperators (`&&`) im ersten Argument aus. Der Simplifikator schließt den Beweis mit Hilfe der vorhandenen Induktionsannahme ab.

Der obige Beweis veranschaulicht auch wie die natürliche Induktion zusammen mit dem Take-Lemma als ein Beweisprinzip für Streams verwendet wird. Ferner impliziert das Take-Lemma zwei äquivalente Sichten auf einen Stream $s$, was sich zunächst informell wie folgt ausdrücken lässt:

$$\{p \mid \texttt{take}\ n{\cdot}s = p \wedge n \in \mathbb{N}\} \stackrel{!}{=} s \qquad (\textbf{N}\ 3.4.0)$$

Eine Formalisierung dieses Sachverhaltes für Streams auf flachen Typen (sie kann einfach durch Typerweiterung verallgemeinert werden) wird im folgenden Kapitel eingeführt.

# 4 Fstreams - Ein formales Modell für Ströme

Auf der Grundlage des Datentypkonstruktors `stream` wird in diesem Kapitel der abgeleitete Konstruktor `fstream` eingeführt. Anschließend wird das für weitere Definitionen nützliche Take-Funktional auf Fstreams übertragen. Im Weiteren wird die Konkatenation als die zentrale Konstruktionsoperation für Fstreams aufgebaut. Ferner werden wesentliche Funktionen auf Fstreams definiert wie etwa die Längenfunktion. Im Weiteren werden die Induktionsregeln für Fstreams geschaffen. Abschließend werden weitere charakteristische Eigenschaften von hier eingeführten Konstrukten behandelt.

## 4.1 Grundlage für die Konstruktion von Fstreams

Für die Erzeugung von Streams auf beliebigen HOL-Typen ist es notwendig diese Typen zuvor zu einer `pcpo` zu liften. Vom Liften (siehe Abschnitt 2.2) abgesehen bekommen wir auf diese Weise Streams auf beliebigen HOL-Typen. Die zugrunde liegende flache Ordnung, die sich aus dem Liften ergibt, impliziert die Spezialisierung von $\sqsubseteq$ zur Anfangsstück-Relation auf den zugehörigen Streams.

In HOLCF ist die Typklasse `pcpo` die Standardeinstellung (siehe Abschnitt 2.1). Weil wir später die Ströme auf beliebigen Typen erzeugen möchten, ist es sinvoll diese Voreinstellung möglichst früh zu ändern, um die allgemeine Typen ohne die explizite Typanweisung verwenden zu können. Anschließend fassen wir den Typkonstruktor für das Liften und den Typkonstruktor `stream` im Folgenden syntaktisch zum zusammengesetzten Typkonstruktor `fstream` zusammen. Insgesamt ergibt sich:

```
defaultsort type
types 'a fstream = "('a lift) stream"
```

(**TS** 4.1.0)

In der ersten Zeile wird die Standardtypeinstellung von HOLCF (`pcpo`) durch den HOL-Befehl `defaultsort` auf den allgemeinsten Typ, nämlich `type`, zurückgesetzt. Anschließend wird die Bedeutung des Schlüsselwortes `fstream` festgelegt. Mit `fstream` erhalten wir damit einen Datentypkonstruktor für Ströme. Dabei werden der leere Strom, in der Literatur $\varepsilon$, hier durch $\bot$ und die endlichen Ströme durch die partiellen Fstreams vertreten. In den folgenden Abschnitten werden Lemmata und Funktionalitäten auf `fstream` eingeführt, die das obige Liften der zugrunde liegenden Zeichenmenge verstecken, die vorhandenen Eigenschaften und Lemmata von dem allgemeineren Datentypkonstruktor `stream` auf `fstream` anpassen und abschließend die neuen `fstream`-spezifischen Eigenschaften und Funktionalitäten darstellen. Insbesondere wird ein `fstream`-spezifischer Konstruktionsoperator mit Hilfe der Konkatenation entwickelt. Damit wird uns also der Datentypkonstruktor

fstream als ein formales Modell für Ströme auf beliebigen Mengen bzw. Typen im Sinne der Focus-Entwicklungsmethodik zur Verfügung stehen. Im Weiteren werden wir also die Begriffe der Ströme und Fstreams entsprechend den obigen Ausführungen als Synonyme verwenden. Mit der obigen Standardtypeinstellung und der anschließenden Typvereinbarung für fstream rücken die HOLCF-Typen und damit auch HOLCF syntaktisch in den Hintergrund.

Entsprechend der obigen Vereinbarung wird das Lemma scase (Abschnitt 3.4) auf Fstreams unter dem Namen fscase1 übertragen:

(s::'a fstream) $\neq$ $\bot$ $\Longrightarrow$ $\exists$x xs. s = $\uparrow$x && xs (fscase1)

Analog wird das Lemma stream.inverts (Abschnitt 3.2) unter dem Namen fstream_inverts1 übernommen:

$\uparrow$x && xs $\sqsubseteq$ $\uparrow$y && ys $\Longrightarrow$ x = y $\wedge$ xs $\sqsubseteq$ ys (fstream_inverts1)

Die Eigenschaften der Klasse der flachen HOLCF-Typen sind in der HOLCF-Theorie *Pcpo* durch das Axiom ax_flat festgelegt. Falls zwei Instanzen eines solchen Typs in der Relation $\sqsubseteq$ stehen, so sind sie entweder identisch oder die erste Instanz ist gleich dem kleinsten Element ($\bot$). Dies gilt insbesondere für beliebige geliftete HOL-Typen, so dass sie zu der Typklasse der flachen HOLCF-Typen automatisch gehören.

Abschließend wird das Lemma stream.injects (Abschnitt 3.2) unter dem Namen fstream_injects1 auf Fstreams übertragen:

($\uparrow$x && xs = $\uparrow$y && ys) = (x = y $\wedge$ xs = ys) (fstream_injects1)

Alle in diesem Abschnitt eingeführten Lemmata dienen nur als ein Zwischenschritt zu einer konstruktiven, typspezifischen und komfortablen Charakterisierung von Fstreams die in Abschnitt 4.4 anhand des in Abschnitt 4.3 eingeführten Konkatenationsoperators dargestellt wird.

## 4.2 Take-Funktional auf Fstreams und seine kleinste obere Schranke

Mit Hilfe des Take-Funktionals auf Streams aus dem vorhergehenden Kapitel definieren wir hier ein Take-Funktional für Fstreams (im Weiteren nur das Take-Funktional) wie folgt:

```
take   :: "nat ⇒ 'a fstream → 'a fstream
take_def: "take n ≡ Λs. stream_take n·s
```

(**Def** 4.2.0)

Im Prinzip ist das Take-Funktional bis auf die Typeinschränkung identisch dem Take-Funktional für Streams.

Anhand der obigen Definition und den im Simplifikator vorhandenen Regeln über stream_take aus dem Abschnitt 3.3 werden die folgenden Entfaltungslemmata bewiesen:

$$\left.\begin{array}{l}\texttt{take 0·x = ⊥}\\ \texttt{take n·⊥ = ⊥}\\ \texttt{take (Suc n)·(↑x \&\& xs) = ↑x \&\& take n·xs}\end{array}\right\} \quad \texttt{(take\_simps)}$$

Die obigen Lemmata werden dem Simplifikator hinzugefügt. Analog wird auch das Take-Lemma für Fstreams unter dem Namen `take_lemma` auf Fstreams übertragen.

$$\texttt{(}\bigwedge\texttt{n. take n·s1 = take n·s2)} \Longrightarrow \texttt{s1 = s2} \qquad \texttt{(take\_lemma)}$$

Die Ketteneigenschaft des Take-Funktionals wird durch Einsetzen der obigen Definition, der HOLCF-Introduktionsregel `chainI` (Abschnitt 2.2) und dem Lemma `chain_take_lemmas` (Abschnitt 3.4) unter dem Namen `chain_take` formalisiert:

$$\texttt{chain (}\lambda\texttt{i. take i·s)} \qquad \texttt{(chain\_take)}$$

Mit den vorhergehenden Konstrukten wird die für die Umsetzung fundamentale Eigenschaft des Take-Funktionals hier bewiesen:

$$\texttt{(}\bigsqcup\texttt{i. take i·s) = s} \qquad \texttt{(fstream\_reach)}$$

Die Kette $\lambda$`n. take n·s` beschreibt somit durch die Bildung der kleinsten oberen Schranke eindeutig den Strom `s` und erlaubt uns das Take-Funktional als eine Approximationsfunktion bzw. eine äquivalente Darstellungsform für Ströme (als Ketten) anzusehen. Entsprechend den Ausführungen in Kapitel 2.2 beschreibt das Take-Funktional andererseits approximativ die Identitätsfunktion. Durch die flache Ordnung des zugrunde liegenden Alphabets (siehe Anfang von Abschnitt 4.1) besitzt jede nicht-stationäre[1] Kette sogar genau eine obere Schranke die damit automatisch die kleinste obere Schranke ist.

Die obige Eigenschaft wird wie folgt bewiesen:

```
lemma fstream_reach: "(⨆i. take i·s) = s"
apply (rule take_lemma [OF spec [where x=s]])
apply (induct_tac n, auto)
apply (case_tac "x=⊥", auto)
apply (simp add: lub_const [THEN thelubI])
apply (drule fscase1, auto)
apply (insert chain_take)
apply (subst lub_range_shift [where j="Suc 0", THEN sym], auto)
by (subst contlub_cfun_arg [THEN sym], auto)
```

(**B** 4.2.0)

Die Beweisführung stützt sich auf das Take-Lemma und die natürliche Induktion. Insbesondere ist hier das Lemma `lub_range_shift` aus der HOLCF-Theorie *Pcpo* nützlich:

$$\texttt{chain Y} \Longrightarrow \texttt{(}\bigsqcup\texttt{i. Y (i + j)) = (}\bigsqcup\texttt{i. Y i)} \qquad \texttt{(lub\_range\_shift)}$$

Wenn in der obigen Regel $j$ mit $Suc\ 0$ instantiiert wird, so bekommen wir mit der anschließenden Symetrieanwendung und der Simplifikation von $i + (Suc\ 0)$ zu $Suc\ i$ eine Möglichkeit zur Entfaltung der kleinsten oberen Schranken von Ketten:

[1] Eine Kette nennen wir stationär, falls sie ein maximales Element enthält.

```
chain Y ⟹ (⨆i. Y i) = (⨆i. Y (Suc i))                    (lub_range_shift_Suc)
```

Ist die obige Kette nichttrivial (nicht konstant) und so beschaffen, dass der $\bigsqcup$-Operator auf der rechten Seite der Gleichheit im vorhergehenden Satz nach der Entfaltung von `Y (Suc i)` zu `Y i` eingerückt werden kann, so wird dadurch eine rekursive Charakterisierung von `⨆i. Y i` gewonnen. Die Einrückung von $\bigsqcup$ ist aber nur dann möglich, wenn die entsprechenden Operatoren nach der Entfaltung der Kette den Satz `contlub` (Abschnitt 2.3) erfüllen. Danach kann die Induktionsannahme im obigen Beweis angewandt werden und dadurch das obige Lemma schließlich bewiesen werden.

In Verbindung mit den grundlegenden Eigenschaften von stetigen Funktionen (Abschnitt 2.3) und der bereichstheoretischen Zulässigkeit von Eigenschaften, die in Abschnitt 4.6 ausführlich behandelt wird, ermöglicht das Lemma `fstream_reach` also den Sprung vom Endlichen ins Unendliche bezüglich der Stromlänge.

Abschließend wird mit Hilfe des obigen Satzes das folgende Hilfslemma bewiesen und dem Simplifikator hinzugefügt:

```
take n·s ⊑ s                                              (ub_of_take_simp)
```

## 4.3 Konkatenation

Um einen Operator für die Konkatenation (das „Zusammenkleben“) zweier Ströme einzuführen, benötigen wir die folgende Hilfsfunktion:

```
rec_fsconc :: "nat ⇒ 'a fstream ⇒ 'a fstream ⇒ 'a fstream"
primrec
 rec_fsconc_0   : "rec_fsconc 0 s1 s2 = ⊥"
 rec_fsconc_Suc : "rec_fsconc (Suc n) s1 s2 =
                     if (s1 = ⊥)
                        then ft·s2 && rec_fsconc n ⊥ (rt·s2)
                        else ft·s1 && rec_fsconc n (rt·s1) s2"
```

(**Def** 4.3.0)

Die Funktion `rec_fsconc` wird mit dem HOL-Konstrukt `primrec` definiert. Sie enthält ergänzend eine Parametrisierung durch eine natürliche Zahl. Zwei Ströme werden entsprechend diesem Parameter punktweise und nacheinander durchlaufen und ausgegeben. Ist das Ende des ersten Stromes erreicht (im Sinne des kleinsten Elements ($\bot$)) setzt also `rec_fsconc` mit dem zweiten Strom fort.

Mit Hilfe der obigen Funktion `rec_fsconc` führen wir schließlich mit der folgenden Definition den Konkatenationsoperator auf Fstreams ein:

```
fsconc     :: "'a fstream ⇒ 'a fstream ⇒ 'a fstream"
fsconc_def : "fsconc ≡ λx y. (⨆i. rec_fsconc i x y)"  (_⌢_)²
```

(**Def** 4.3.1)

Für die Konkatenation zweier Ströme $x$ und $y$ wird in der Definition die Infixnotation $x^\frown y$

vereinbart. Mit der obigen rekursiven Hilfsdefinition wurde also eine zur Konkatenationsoperation passende Kette (Beweis - s.u.) aus zwei Strömen gebildet. Das Ergebnis der Konkatenation zweier Ströme (Def 4.3.1) wird somit durch die Bildung der kleinsten oberen Schranke dieser Kette erhalten.

Ist das zweite Argument von `rec_fsconc` der leere Strom, so wird die folgende Charakterisierung von `rec_fsconc` per Induktion und anschließende Fallunterscheidung bewiesen:

```
rec_fsconc n ⊥ y = take n·y                                    (rfsconc_fst_bot2take)
```

Analog wird die Eigenschaft von `rec_fsconc` bewiesen, bei der das dritte Argument der leere Strom ist. Damit werden direkt durch die Entfaltung der obigen Definition `fsconc_def` und das anschließende Einsetzen von `fstream_reach` (Abschnitt 4.2) zwei einfachere Fälle der Konkatenation bewiesen:

```
⊥⌢y = y   }
x⌢⊥ = x   }                                                    (fsconc_simps)
```

Der wesentliche Punkt ist, dass `rec_fsconc` auf zwei Strömen $x$ und $y$ eine Kette erzeugen soll (`chainI` in Abschnitt 2.2), d.h.:

```
chain (λi. rec_fsconc i x y)                                   (chain_rfsconc)
```

Die Ketteneigenschaft und die Eigenschaften des Konstruktionsoperators (`&&`) als eine stetige Funktion erlauben es die Definition der Konkatenation mit Hilfe des Lemmas `lub_range_shift` (Abschnitt 4.2) zu entfalten. Die gewünschte Ketteneigenschaft wird durch die natürliche Induktion bewiesen. Damit wird anschließend die übliche, rekursive Beschreibung der Konkatenation für das nichttriviale erste Argument bewiesen:

```
(↑x && y)⌢z = ↑x && (y⌢z)                                      (fsconc2scons_simp)
```

Der Beweis des obigen Lemmas sieht wie folgt aus:

```
lemma fsconc2scons: "(↑x && y)⌢z = ↑x && (y⌢z)"
apply (unfold fsconc_def, insert chain_rfsconc)
apply (subst lub_range_shift [where j="Suc 0", THEN sym], auto)
by (rule contlub_cfun_arg [THEN sym], auto)
```

(**B** 4.3.0)

Die formale Beweisführung stützt sich auf die Ketteneigenschaft der Hilfsfunktion `chain_rfsconc`.

## 4.4 Eine konstruktive Charakterisierung von Fstreams

Mit den bisherigen Konstrukten wird hier eine bequeme Charakterisierung von Fstreams auf der Basis des Konkatenationsoperators, also einheitlich im Bezug auf die Konkatenation,

[2] Diese Infixnotationsvereinbarung entspricht nicht der exakten Isabelle-Syntax für Notationsvereinbarungen. An dieser Stelle wurde aus Übersichlichkeitsgründen auf eine solche verzichtet. Dies gilt auch für alle nachfolgenden Notationsvereinbarungen für Operatoren in Isabelle.

formalisiert. Zunächst führen wir zusätzlich einen Operator für die Bildung von einelementigen Fstreams wie folgt ein:

```
fstream_pt    :: "'a ⇒ 'a fstream"  ("<_>")
fstream_pt_def: "<a> ≡ ↑a && ⊥"
```

(**Def** 4.4.0)

Insbesondere gilt mit der obigen Definition trivialerweise die folgende Gleichung:

```
<x>⌢xs = ↑x && xs
```

(fscons2scons)

Einelementige Fstreams werden also die Grundbausteine für die Bildung der restlichen Fstreams. Dies hat zwei wesentliche Vorteile: Zum Einen wird das Liften verdeckt und zum Anderen gibt es keinen neuen Konstruktionsoperator bzw. die Konkatenation kann gleichzeitig auch als der Konstruktionsoperator für jeden Fstream verwendet werden. Weil wir die einelementigen Fstreams nach der vollständigen Charakterisierung der Konstruktion in diesem Abschnitt als Grundbausteine betrachten werden, darf die obige Regel nicht den Simplifikationsregeln hinzugefügt werden. Mit dem obigen Konstrukt für die einelementigen Fstreams wird schließlich die Zusammensetzung von Fstreams auf der Basis des Konkatenationsoperators und der Bildung von einelementigen Fstreams im Folgenden Lemma formalisiert:

```
⋀s. s = ⊥ ∨ (∃x xs. s = <x>⌢xs)
```

(fstream_exhaust)

Analog werden die folgenden Lemmata eingeführt:

```
s ≠ ⊥ ⟹ ∃x xs.  s = <x>⌢xs
```

(fscase)

```
<x>⌢xs ⊑ <y>⌢ys ⟹ (x = y ∧ xs ⊑ ys)
```

(fstream_inverts)

```
<x>⌢xs = <y>⌢ys) ⟹ (x = y ∧ xs = ys)
```

(fstream_injects)

Die Beweise der vorhergehenden Lemmata werden durch die entsprechenden Definitionen geführt. Das entsprechende Entfaltungslemma für das Take-Funktional wird auch formalisiert und dem Simplifikator hinzugefügt:

```
take (Suc n)·(<x>⌢xs) = <x>⌢take n·xs
```

(take_Suc_fscons_simp)

Anschließend werden der Kopf- und der Restoperator analog der obigen Vorgehensweise auf die Fstreams übertragen:

```
ft·(<x>⌢xs) = ↑x
rt·(<x>⌢xs) = xs
```

(sel_fscons_simps)

Beide obige Regeln werden dem Simplifikator hinzugefügt. Für die Beweise auf der Basis der Präfixrelation auf Strömen sind insbesondere die folgenden Lemmata nützlich:

```
x ⊑ y ⟹ (x=⊥) ∨ (∃a s t. s ⊑ t ∧ x = <a>⌢s ∧ y = <a>⌢ t)
```

(le_unfold)

```
x ⊑ <a>⌢s ⟹ (x=⊥) ∨ (∃t. x = <a>⌢t ∧ t ⊑ s)
```

(le_fscons_unfold_fst)

```
<a>⌢s ⊑ x  ⟹  ∃t. x = <a>⌢t  ∧  s ⊑ t                    (le_fscons_unfold_snd)
```

Die vorhergehenden Lemmata werden durch Fallunterscheidungen und das Lemma `fstream_inverts` bewiesen.

## 4.5 Längenfunktion

Mit der Längenfunktion wird für die endlichen Ströme die Längeninformation bestimmt. Außerdem wird durch die externe HOLCF-Theorie *inat* die Behandlung von beliebigen Stromlängen, inklusive der Länge von unendlichen Strömen, algebraisch vereinheitlicht. Mit `inat` wurde deshalb ein neuer Zahlentyp eingeführt, der neben allen natürlichen Zahlen ergänzend das Symbol $\infty$ für die unendliche Zahl enthält. Für jede Instanz $i$ von *inat* gilt: Entweder gibt es eine natürliche Zahl $n$, sodass $i = Fin\ n$ oder $i = \infty$ gilt. Dabei steht `Fin` als Konstruktor auf der Basis des Typs `nat` für die Klasse aller `inat`-Zahlen außer $\infty$.

Mit der folgenden Definition führen wir die Längenfunktion auf Fstreams ein:

```
fslen   :: "'a fstream ⇒ inat"  ("#_")
fslen_def: "#s  ≡  if (∃n. take n·s = s)
                      then Fin (LEAST n. take n·s = s)
                      else ∞"
```

(**Def** 4.5.0)

In der obigen Definition wird der HOL-Operator `LEAST` eingesetzt, der die kleinste natürliche Zahl auswählt für die die jeweilige Eigenschaft erfüllt wird. In unserem Fall soll das kleinste $n$ so ausgewählt werden, dass `take n·s = s` gilt. Falls es überhaupt kein solches $n$ gibt, so ist der Strom unendlich. Das folgende Lemma stellt die grundlegende Charakterisierung des Längenoperators auf der Basis der Konkatenation dar:

```
#(<x>⌢xs) = iSuc (#xs)                                  (fslen_fscons_simp)
```

Der Operator `iSuc` ist eine passende Erweiterung des Operators `Suc` aus der HOL-Theorie *Nat* um die abstrakte unendliche Zahl. Das Lemma wird mit den hier eingeführten Konstrukten und dem Lemma `Least_Suc2` für den `LEAST`-Operator aus der HOL-Theorie *Nat* bewiesen. Aus der obigen Definition folgt mit Induktion sofort der folgende Satz über die Beziehung des Längenoperators und des Take-Funktionals:

```
∀x k. #x = Fin k  ∧  k<=i  ⟶  take i·x = x                (fslen_take)
```

Für den Monotoniebeweis des Längenoperators (bezüglich der `inat`-Zahlen) beweisen wir mit Induktion über $k$ das folgende Hilfslemma:

```
∀x y. #x = Fin k  ∧  y ⊑ x  ⟶  #y <= Fin k              (fin_fslen_mono)
```

Mit dem vorhergehenden Lemma ergibt sich anhand der Fallunterscheidung über die Länge von $y$:

```
x ⊑ y  ⟹  #x <= #y                                      (fslen_mono)
```

Das folgende Hilfslemma wird bewiesen um anschließend die Äquivalenz von zwei gleich langen Strömen, die in der Präfixrelation zueinander stehen, zu folgern:

```
∀s t. s ⊑ t ∧ #s = #t ⟶ take n·s = take n·t          (leAfslen_eq2take_eq)
```

Das obige Lemma wird hauptsächlich mit der natürlichen Induktion über $n$ bewiesen. Abschließend ergibt sich mit dem Take-Lemma und dem obigen Hilfslemma der folgende Satz:

```
⟦s ⊑ t; #s = #t⟧ ⟹ s = t                              (leAfslen_eq2eq)
```

## 4.6 Grundlegende Induktionsregeln für Fstreams

Mit Hilfe der bisher entwickelten Charakterisierung von Fstreams lassen sich die grundlegenden Induktionsregeln für Fstreams einführen. Zunächst wird das folgende Hilfslemma durch die natürliche Induktion über $n$ und die anschließende Fallunterscheidung über $x$ bewiesen:

```
∀x. (P ⊥ ∧ (∀a s. P s ⟶ P (<a>⌢s))) ⟶ P (take n·x)   (fscons_take_ind)
```

Mit dem obigen Hilfslemma `fscons_take_ind`, dem Lemma `fslen_take` (Abschnitt 4.5) bzw. dessen Ausprägung `#s = Fin k ⟹ take k·s = s`, der Definition Def 4.4.0 und den Lemmata in der Regelmenge des Simplifikators (insbesondere das Lemma `fsconc2scons_simp` in Abschnitt 4.3) wird die folgende grundlegende Regel für die Induktion über die endlichen Fstreams bewiesen:

```
⟦#x = Fin k; P ⊥; ⋀a s. P s ⟹ P (<a>⌢s)⟧ ⟹ P x        (fscons_fin_ind)
```

Die allgemeine Induktionsregel, also die Induktion über die Ströme beliebiger Länge, wird durch den folgenden Beweis bzw. Lemma eingeführt:

```
⟦adm (P); P ⊥; ⋀a s. P s ⟹ P (<a>⌢s)⟧ ⟹ P x          (fscons_ind)
```

Das obige Lemma wird im Folgenden bewiesen:

```
lemma fsconc_ind:
  "⟦adm (P); P ⊥; ⋀a s. P s ⟹ P (<a>⌢s)⟧ ⟹ P x"
apply (unfold adm_def)
apply (erule_tac x="λn. take n·x" in allE, auto)
apply (simp add: chain_take)
apply (rule fscons_take_ind [rule_format], auto)
by (simp add: fstream_reach)
```

(**B** 4.6.0)

Der obige Beweis ist bis auf die Zulässigkeit (`adm`) analog dem besprochenen Beweis der vorhergehenden Induktionsregel. Die Zulässigkeit wird in der HOLCF-Theorie *Adm* wie folgt definiert:

```
adm   :: "('a::cpo ⇒ bool) ⇒ bool"
adm_def: "adm P ≡ ∀Y. chain Y ⟶ (∀i. P (Y i)) ⟶ P (⨆i. Y i)"
```
(**Def** 4.6.0)

Der obige Beweis von `fscons_ind` wird durch die Entfaltung der Definition der Zulässigkeit, das Einsetzen der Kette $\lambda$`n. take n·x` in die vorhin entfaltete Definition, die Anwendung von `fscons_take_ind` und den anschließenden Einsatz von `fstream_reach` geführt. Mit Hilfe der Zulässigkeit konnten wir also im obigen Beweis von allen endlichen Präfixen eines unendlichen Stromes auf die kleinste obere Schranke dieser Präfixe bzw. diesen unendlichen Strom „springen“.

Die explizite Behandlung der Zulässigkeit für Eigenschaften verschiedener Art durch die Verallgemeinerung auf alle Ketten ist im Gegensatz zum obigen Beweis, bei dem die Zulässigkeitseigenschaft nur für die Take-Kette benötigt wurde, bezüglich der Beweisführung meistens einfacher. Deshalb wurde die Zulässigkeit als ein eigenständiges, bereichstheoretisches Konstrukt eingeführt. Ferner kann die Zulässigkeit einer komlexen Eigenschaft in vielen Fällen auf die Komposition der Zulässigkeiten für einfachere Teil-Eigenschaften zurückgeführt werden. Viele grundlegende Lemmata bezüglich der Zulässigkeit finden sich in der HOLCF-Theorie *Adm*.

## 4.7 Weitere Eigenschaften der Konkatenation

### 4.7.1 Monotonie

Mit der Induktion über $n$ wird zunächst das folgende Lemma bewiesen:

```
∀x y. #x = ∞ ⟶ rec_fsconc n x y = take n·x
```
(`rfsconc2fst`)

Daraus folgt sofort mit der Definition des Konkatenationsoperators (Def 4.3.0/Def 4.3.1) und dem Lemma `fstream_reach` (Abschnitt 4.2) der folgende Satz:

```
#x = ∞ ⟹ x⌢y = x
```
(`fsconc2fst`)

Abschließend wird die Monotonie des Konkatenationsoperators im zweiten Argument mit dem folgenden Lemma eingeführt:

```
monofun (λy. x⌢y)
```
(`monofun_fsconc`)

Zunächst wird die Fallunterscheidung auf der Länge von $x$ vorgenommen. Der unendliche Fall wird sofort mit dem Satz `fsconc2fst` bewiesen. Im endlichen Fall führen wir die endliche Induktion (`fscons_fin_ind` aus dem vorhergehenden Abschnitt) auf dem Strom $x$ durch. Die Konkatenation wird schließlich wegen der Monotonie im ersten Argument auf den Konstruktionsoperator für Streams (`&&`) anhand der Definition in Def 4.4.0 zurückgeführt. Mit der Ausnutzung der Induktionsannahme wird der Beweis abgeschlossen.

### 4.7.2 Stetigkeit

Mit der Monotonie der Konkatenation im zweiten Argument aus dem vorhergehenden Abschnitt lässt sich sofort das folgende Lemma beweisen:

```
∀s. chain Y ⟶ chain (λi. s⌢(Y i))                    (chain_fsconc)
```

Anschließend wird das folgende Lemma eingeführt:

```
contlub (λy. x⌢y)                                    (contlub_fsconc)
```

Das Lemma wird wie folgt bewiesen:

```
lemma contlub_fsconc: "contlub (λy. x⌢y)"
apply (rule contlubI)
apply (case_tac "#x")
apply (rule fscons_fin_ind [of x], auto)
apply (simp add: fstream_pt_def)
apply (rule contlub_cfun_arg)
apply (simp add: chain_fsconc)
by (rule sym, rule lub_const [THEN thelubI])
```

(**B** 4.7.0)

Der obige Beweis stützt sich also hauptsächlich auf die endliche Induktion über Fstreams (`fscons_fin_ind`), die Zurückführung der Konkatenation auf den Konstruktionsoperator für Streams (`&&`) und die obige Ketteneigenschaft für das zweite Argument (`chain_fsconc`).

Abschließend wird mit Hilfe der Monotonie im zweiten Argument und des obigen Lemmas die Stetigkeit der Konkatenation im zweiten Argument bewiesen:

```
cont (λy. x⌢y)                                       (cont_fsconc)
```

Zur automatischen Konstruktion stetiger Funktionen ist es nützlich die Stetigkeit im Funktionstyp verankert zu haben, deshalb wird hier durch die folgende Definition eine bis auf den Funktionstyp identische Funktion eingeführt wird:

```
cfsconc   :: "'a fstream ⇒ 'a fstream → 'a fstream"  ("_ ++")
cfsconc_def: "cfsconc s1 ≡ Λs2. s1⌢s2"
```

(**Def** 4.7.0)

Dieser Operator ist insbesondere in Verbindung mit den HOLCF-Simplifikationsregeln nützlich z.B. bei automatischen Zulässigkeitsüberprüfungen oder bei der Einführung von neuen Operatoren durch die Fixpunktgleichungen. Es gilt anhand der oben bewiesenen Stetigkeit der Konkatenation im zweiten Argument die folgende Gleichheit:

```
s1 ++ ·s2 = s1⌢s2                                    (cfsconc2fsconc)
```

Sobald also der Zweck der im Funktionstyp gekapselten Stetigkeit erfüllt wird kehren wir mit dem obigen Satz auf die normale Konkatenationsoperation zurück. Diese Methode wird in Kapitel 6 bei der Definition von HOLCF-Funktionen eingesetzt.

### 4.7.3 Assoziativität

Mit der Fallunterscheidung auf der Länge von $x$ und der anschließenden endlichen Induktion über $x$ wird die Assoziativität des rechtsassoziativen Konkatenationsoperators bewiesen:

```
(x⌢y)⌢z = x⌢y⌢z                                    (fsconc_assoc_simp)
```

### 4.7.4 Länge der Konkatenation zweier endlichen Ströme

Anhand der Assoziativität und mit der natürlichen Induktion über $n$ wird der folgende Satz über die Länge der Konkatenation zweier endlichen Ströme bewiesen:

```
∀x y. Fin n = #x ∧ Fin m = #y ⟶ #(x⌢y) = Fin (n + m)          (fsconc_fin)
```

### 4.7.5 Injektivität

Mit der endlichen Induktion über $x$ wird sofort die Injektivität der Konkatenation im zweiten Argument bewiesen:

```
#x = Fin k ⟹ (x⌢y = x⌢z) = (y = z)                          (fsconc_inject)
```

## 4.8 Weitere Eigenschaften des Take-Funktionals

### 4.8.1 Idempotenz

Eine Funktion ist genau dann idempotent, wenn die mehrfache Komposition der Funktion mit sich selbst das Ergebnis einer einmaligen Funktionsanwendung nicht ändert. Informell kann dieser Sachverhalt wie folgt beschrieben werden:

$$f\ (f\ x)\ =\ f\ x \qquad (\mathbf{N}\ 4.8.0)$$

Das Take-Funktional ist idempotent für ein festes, erstes Argument $n$:

```
∀n. take n·(take n·s) = take n·s                          (take_idempotent_simp)
```

Der Idempotenzbeweis für das Take-Funktional wird sofort mit der Induktion über $s$ und der anschließenden Fallunterscheidung über $n$ beweisen. Der Zulässigkeitsbeweis erfolgt automatisch durch den Simplifikator, weil `take n` stetig ist.

### 4.8.2 Komposition

Für die Komposition des Take-Funktionals mit sich selbst werden neben dem obigen Idempotenzbeweis zwei folgende Sätze eingeführt:

```
∀k n. n<k  ⟶  take n·(take k·s) = take n·s                    (take_comp_out_le)
```

```
∀k n. k<n  ⟶  take n·(take k·s) = take k·s                    (take_comp_in_le)
```

Die Beweise der beiden obigen Sätze sind dem Idempotenzbeweis im vorhergehenden Abschnitt sehr ähnlich. Es wird in beiden Beweisen zusätzlich eine Fallunterscheidung auf $k$ vorgenommen.

## 4.9 Die verallgemeinerte Induktion

Die grundlegenden Induktionsprinzipien für Ströme (Abschnitt 4.6) können mit Hilfe der obigen Eigenschaften der Konkatenation dazu verwendet werden, die allgemeine Induktion auf Fstreams einzuführen.

Die folgenden Regeln formalisieren die Induktion am üblichen linken Ende eines Stromes.

```
⟦#w = Fin k; P ⊥; ⋀s t. P s ⟹ P (t⌢s)⟧ ⟹ P w                  (fsconc_fin_ind)
```

```
⟦adm (P); P ⊥; ⋀s t. P s ⟹ P (t⌢s)⟧ ⟹ P w                    (fsconc_ind)
```

Ergänzend wird von der Konkatenation die Möglichkeit gegeben auf beliebige Ströme auf ihr rechtes Ende, falls sie endlich sind, zuzugreifen. Dementsprechend werden zwei weitere Induktionsmöglichkeiten eingeführt.

```
⟦#w = Fin k; P ⊥; ⋀s t. P s ⟹ P (s⌢t)⟧ ⟹ P w                  (r_fsconc_fin_ind)
```

```
⟦adm (P); P ⊥; ⋀s t. P s ⟹ P (s⌢t)⟧ ⟹ P w                    (r_fsconc_ind)
```

## 4.10 Löschen beliebig langer Anfangsstücke und der punktweise Zugriff

Um einen Operator für das Löschen eines endlich langen Anfangsstücks eines Stromes unter dem Namen `drop` zu formalisieren wird der Iterationsoperator aus HOLCF genutzt. Der Iterationsoperator `iterate` mit den zugehörigen Lemmata befindet sich in der HOLCF-Theorie *Fix* (siehe auch Abschnitt 2.2).

```
drop   :: "nat  ⇒  'a fstream  →  'a fstream"
drop_def: "drop i  ≡  Λs. iterate i rt s"
```

(**Def** 4.10.0)

Der Operator `drop` ist anhand des Typkonstruktors → stetig im zweiten Argument. Die Eigenschaften von `drop` lassen sich hauptsächlich mit natürlicher Induktion und den Eigenschaften von `iterate` beweisen. Insbesondere sind die zwei folgenden Lemmata für den Umgang mit `drop` nützlich:

```
drop (Suc n)·(<a>⌢s) = drop n·s                               (drop_forw)
```

```
drop (Suc n)·s = rt·(drop n·s)                                    (drop_back)
```

Die obigen Lemmata werden die Vorwärts- und die Rückwärtsentfaltung entsprechend dem jeweiligen Namen genannt. Mit dem drop Operator lässt sich ergänzend der Operator für den direkten, punktweisen Zugriff auf Fstreams wie folgt formalisieren:

```
pt   :: "nat ⇒ 'a fstream → 'a fstream"
pt_def: "pt i ≡ Λs. take (Suc 0)·(drop i·s)"
```

(**Def** 4.10.1)

Mit der natürlichen Induktion über $n$ ergibt sich die folgende Entfaltungseigenschaft für den pt-Operator für den Fall, bei dem die Argumente nichttrivial sind:

```
pt (Suc n)·(<a>⌢y) = pt n·y                                  (pt_Suc_fscons_simp)
```

Ferner lässt sich mit Hilfe des obigen Operators ein weiterer Operator für den direkten Zugriff auf die Zeichen eines Stromes wie folgt einführen:

```
nth   :: "nat ⇒ 'a fstream ⇒ 'a"
nth_def: "nth n s ≡ if (Fin n<#s) then (THE a. pt n·s = <a>) else arbitrary"
```

(**Def** 4.10.2)

## 4.11 Weitere grundlegende Lemmata für Fstreams

### 4.11.1 Zerlegungslemma

Die Beziehung zwischen dem Take-Funktional und dem drop-Operator beschreibt das Stromzerlegungslemma:

```
(take n·s)⌢(drop n·s) = s                                      (split_fstream)
```

Das Lemma wird mit der natürlichen Induktion über $n$ bewiesen. Jeder Strom kann für eine beliebige natürliche Zahl $n$ in zwei Teile zerlegt werden, die sich entsprechend den obigen Operatoren ergeben.

### 4.11.2 Lemma für den punktweisen Vergleich

Mit dem Operator pt wird der folgende Satz für den punktweisen Vergleich zweier Ströme formalisiert:

```
(⋀n. pt n·s1 = pt n·s2) ⟹ s1 = s2                              (ptwise_comp)
```

Der Beweis des Lemmas wird mit Hilfe des Take-Lemmas auf die natürliche Induktion zurückgeführt. Danach werden die Entfaltungseigenschaften des pt-Operators, die Fallunterscheidungen auf den Strömen $s1$ und $s2$ und die passenden Spezialisierungen angewandt. Mit dieser Regel können die Beweise über Ströme beliebiger Länge analog dem Take-

Lemma auf die natürliche Induktion zurückgeführt werden. Darüber hinaus ermöglicht der punktweise Zugriff eine elegante Unterscheidungsmöglichkeit für Ströme.

### 4.11.3 Approximationslemma

Das folgende Lemma beschreibt die Tatsache, dass die Relation $\sqsubseteq$ auf Strömen die Anfangsstück-Relation ist bzw., dass jedes Anfangsstück eines Stromes zum ursprünglichen Strom verlängert werden kann:

$$\texttt{s1} \sqsubseteq \texttt{s2} \implies \exists \texttt{t.}\ \texttt{s1}^\frown\texttt{t} = \texttt{s2} \qquad (\texttt{approx})$$

Der Beweis des Approximationslemma basiert auf der Fallunterscheidung über die Länge des ersten Stromes. Ist der erste Strom unendlich, dann sind beide Ströme identisch (siehe die Lemmata `fslen_mono` und `leAfslen_eq2eq`). Im Fall des endlichen ersten Stromes werden das Lemma `split_fstream` und die folgende Eigenschaft des Take-Funktionals angewandt:

$$\forall \texttt{s t k.}\ \texttt{\#s = Fin k} \wedge \texttt{s} \sqsubseteq \texttt{t} \wedge \texttt{n<=k} \longrightarrow \texttt{take n·s = take n·t} \qquad (\texttt{le2take\_eq})$$

Das obige Hilfslemma wird hauptsächlich über die natürliche Induktion über $n$ und den Simplifikator bewiesen.

# 5 Die Definitionsprinzipien für rekursive Funktionen auf Fstreams

Anhand des Konkatenationsoperators wird hier ein kompaktes Definitionsmuster beschrieben, um in einfacher Weise rekursive Funktionen auf Fstreams zu definieren. Mit diesem Definitionsmuster können beliebige rekursive Funktionen auf Fstreams, insbesondere auch die stromverarbeitenden Funktionen, systematisch und elegant definiert werden. Ferner wird der HOLCF-Ansatz für die Definition von automatisch erkennbaren stetigen HOLCF-Funktionen auf der Basis von bereits vorhandenen stetigen Funktionen und des Fixpunktoperators vorgestellt.

## 5.1 Ein allgemeines Definitionsmuster

### 5.1.1 Rekursive Funktionen auf Strömen

Die im vorhergehenden Kapitel eingesetzte Methode für die Definition des Konkatenationsoperators wird im Folgenden Lemma auf der Basis des Konkatenationsoperators verallgemeinert eingeführt und formalisiert:

```
⟦chain K; ∀i. K (Suc i) in = out⌢(K i (h in))⟧
                          ⟹ (⨆i. K i in) = out⌢(⨆i. K i (h in))
                                                        (LubRecLemma)
```

Das Lemma wird im Folgenden bewiesen:

```
lemma LubRecLemma:
  "⟦chain K; ∀i. K (Suc i) in = out⌢(K i (h in))⟧
                          ⟹ (⨆i. K i in) = out⌢(⨆i. K i (h in))"
apply (subst contlub_fsconc [THEN contlubE])
apply (rule chainI, drule ch2ch_fun [THEN chainE], assumption)
apply (subst lub_range_shift [where j="Suc 0", THEN sym], auto)
by (drule ch2ch_fun, assumption)
```

(**B** 5.1.0)

Dieses Lemma stellt eine Basis zur Definition von rekursiven Funktionen auf Strömen dar. Um eine rekursive Funktion auf Fstreams zu definieren, reicht es eine passende Kette von Funktionen $K\ i$ zu finden, die einen Eingabestrom jeweils nur abschnittsweise verarbeiten. Eine solche Kette kann in HOL mit dem Konstrukt `primrec` eingeführt werden. Die so erzeugte Kette besitzt eine kleinste obere Schranke $\lambda x.\bigsqcup i.\ K\ i\ x$ (siehe Abschnitt 2.2), die die gesuchte Funktion darstellt. Diese Funktion verarbeitet schließlich einen Teil der

Eingabe *in* produziert eine Ausgabe *out* und arbeitet dann auf der veränderten Eingabe $h(in)$ weiter.

Diese Methode ermöglicht uns beliebige rekursive Funktionen auf Fstreams, also insbesondere auch nicht monotone, zu definieren. Einige Anwendungsbeispiele werden im folgenden Kapitel angegeben (siehe z.B. die Definition der Funktion `iCycle_def` in Def 6.12.0 bzw. Def 6.12.1).

### 5.1.2 Der Spezialfall: Stromverarbeitende Funktionen

Für die Definition von stromverarbeitenden Funktionen auf der Basis des obigen Konstrukts führen wir das folgende Lemma ein:

```
⟦chain K; ∀x i.∃n. K i (take n·in) = K i in; ∀i. monofun (K i)⟧
                                        ⟹ cont (λin. ⨆i. K i in)
                                                        (contLubFunChainLemma)
```

Dieses Lemma stellt eine wesentliche Basis zur Definition von stromverarbeitenden Funktionen dar. Der Beweis wird hier aus Platzgründen ausgelassen.

Insgesamt ergibt sich die folgende Regel für die Definition von stromverarbeitenden Funktionen:

```
⟦chain K; ∀i. K (Suc i) in = out⌢(K i (h in));
    ∀in i.∃n. K i (take n·in) = K i in; ∀i. monofun (K i)⟧
                          ⟹ cont (λin. ⨆i. K i in) ∧
                              (⨆i. K i in) = out⌢(⨆i. K i (h in))
                                                        (spfAsLubLemma)
```

Jede Funktion *K i* benötigt bei jedem Eingabestrom *in* nur einen endlichen Abschnitt von *in* um das entsprechende Ergebnis für *in* zu berechnen. Dieser wesentliche Abschnitt des Eingabestromes *in* kann also mit dem Take-Funktional begrenzt werden. Ergänzend muss *K i* für jedes *i* monoton sein. Die kleinste obere Schranke all dieser Funktionen ist die gesuchte stromverarbeitende Funktion, die auf allen Eingabeströmen (also auch den unendlichen) definiert ist.

Es gilt auch die Umkehrung der obigen Implikation: Jede stromverarbeitende Funktion $f$ lässt sich durch die obige Konstruktion darstellen, indem für die obige Kette `λi x. K i x = f·(take i·x)` eingesetzt wird.

Der wesentliche Vorteil dieser Methode besteht darin, dass die Stetigkeitsbeweise im Wesentlichen auf die einfache natürliche Induktion zurückgeführt werden. Ein Anwendungsbeispiel für die obige Methode findet sich im folgenden Kapitel (siehe die Funktionsdefinition `scanl_def` in Def 6.10.0 bzw. Def 6.10.1).

## 5.2 Stromverarbeitende Funktionen als Zustandsautomaten

Eine intuitive und graphisch orientierte Darstellung von stromverarbeitenden Funktionen stellen die vollständigen, deterministischen, buchstabierenden Automaten [Rum96, RK99]

dar. In einer Fortsetzung dieses Berichts wird dieser Konstrukt formalisiert. Damit wird eine weitere bequeme Definitionssyntax für stromverarbeitende Funktionen zur Verfügung stehen. Die Stetigkeit einer Funktion die durch einen solchen Automaten eingeführt wird, folgt aus der Struktur des Automaten bzw. der Vollständigkeit und des Determinismus des Automaten.

Die folgende Abbildung stellt beispielhaft einen buchstabierenden Automaten für die Eingabezeichenmenge $\{a, b\}$ dar.

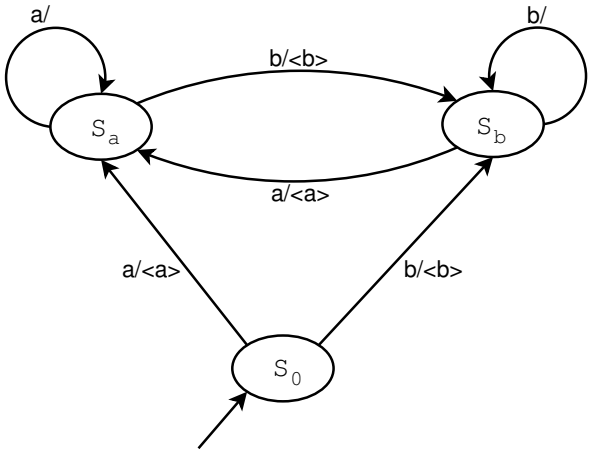


Abbildung 5.1: Graphische Darstellung eines Automaten auf dem Alphabet $M_{in} = \{a, b\}$

Durch die Verallgemeinerung des obigen Automaten auf beliebige Zeichenmengen wird eine alternative Definition für den Operator `nub` (Abschnitt 6.8) erhalten.

## 5.3 Stromverarbeitende Funktionen als HOLCF-Fixpunktgleichungen

In diesem Abschnitt wird zunächst das Konzept der Fixpunkte auf cpo's kompakt, informell und bezüglich der stromverarbeitenden Funktionen beschrieben. Anschließend wird die built-in HOLCF-Definitionssyntax für die Funktionen kurz vorgestellt.

Ein Fixpunkt einer Funktion $h : X \to X$ ist ein Wert $x \in X$ für den gilt:

$$h\,(x) = x \qquad (\mathsf{N}\ 5.3.0)$$

Im Allgemeinen ist für die Bestimmung eines Fixpunktes einer Funktion je nach Anwendungsgebiet ein spezieller Operator (z.B. hier für einen zunächst beliebigen Fixpunkt kurz $fix$, falls existent) vorhanden.

Eine rekursive Funktion ist eine Funktion die sich selbst aufruft bzw. in der Funktionsvereinbarung der Funktion kommt der Name der zu vereinbarenden Funktion auch in der Rechenvorschrift vor, die an diesen Funktionsnamen gebunden werden soll. Dies wird in der bezüglich des Arguments von $f$ abstrakten Notation wie folgt ausgedrückt:

$$f \;:=\; g\,(f) \qquad (\mathbf{N}\ 5.3.1)$$

Die stromverarbeitende Funktion $f$ ist also ein Fixpunkt eines Funktionals $G$. So besitzt z.B. für eine stromverarbeitende Funktion mit dem Funktionstyp $M_{in}^{\omega} \xrightarrow{s} M_{out}^{\omega}$ das entsprechende Funktional den Funktionstyp $(M_{in}^{\omega} \xrightarrow{s} M_{out}^{\omega}) \to (M_{in}^{\omega} \xrightarrow{s} M_{out}^{\omega})$. Es ergäbe sich also mit Hilfe eines zur Zeit fiktiven Fixpunktoperators $fix$ die folgende Definitionsmöglichkeit für die obige stromverarbeitende Funktion $f$:

$$f \;:=\; fix\ G \qquad (\mathbf{N}\ 5.3.2)$$

falls der Operator $fix$ genau den Fixpunkt von $G$ liefert, der mit $f$ identisch ist. In diesem Fall wäre die Definition von $f$ nicht mehr rekursiv. Die Bereichstheorie sichert zunächst die Existenz eines passenden und (in jedem Argument) stetigen Funktionals $G$ für jede stromverarbeitende Funktion $f$ im obigen Sinne. Auch die Existenz eines solchen Fixpunktoperators, der auf $G$ genau die Funktion $f$ liefert (per Extensionalitätsprinzip bzw. die Regel `ext` in Abschnitt 2.1 anhand von N 5.3.1 und N 5.3.2), ist in der Bereichstheorie durch Stetigkeit von $G$ gesichert. Dies wird im Folgenden veranschaulicht.

Anhand von $G$ kann die folgende Menge von Funktionen gebildet werden:

$$f\,0 \;=\; \bot, \quad f\,(\mathtt{Suc}\ n) \;=\; G\,(f\ n) \qquad \text{bzw.} \qquad f\ i = G^i\ \bot \qquad (\mathbf{N}\ 5.3.3)$$

Die Komposition von stetigen Funktionen ist immer stetig (siehe z.B. [Win93]). Ferner ist $\bot$ eine Funktion die für jede Eingabe $\bot$ (als Wert) im Ergebnis liefert und somit auf jedem beliebigen HOLCF-Typ stetig ist. Dementsprechend ist jedes Element der obigen Menge stetig bzw. befindet sich in der Menge aller stetigen Funktionen des Typs $M_{in}^{\omega} \xrightarrow{s} M_{out}^{\omega}$. Eine solche Menge ist ferner eine pcpo dank der Ordnungsrelation die mit der Regel `less_fun_ext` in Abschnitt 2.2 vorgestellt wurde (siehe [Win93, Gun92]). Eine Funktion die für beliebige Eingaben stets $\bot$ als Ergebnis liefert, ist die kleinste Funktion bezüglich der obigen Ordnungsrelation und wird deshalb oben durch $\bot$ symbolisiert. Die Elemente der obigen Menge entstehen also durch die i-fache Anwendung von $G$ auf die kleinste Funktion $\bot$. Weil $G$ monoton ist, folgt aus $\bot \sqsubseteq G\ \bot$ die Ketteneigenschaft für die obige Menge und damit auch die Existenz der kleinsten oberen Schranke dieser Kette. Es ergibt sich dank der Stetigkeit von $G$ (die Regel `contlub` in Abschnitt 2.3) und den Eigenschaften des $\bigsqcup$-Operators (siehe die Regel `lub_range_shift_Suc` in Abschnitt 4.2) schließlich die folgende Gleichheit:

$$G\,(\bigsqcup i.\ G^i\ \bot) \;=\; \bigsqcup i.\ G^i\ \bot \qquad (\mathbf{N}\ 5.3.4)$$

Damit ist $\bigsqcup i.\ G^i\ \bot$ ein Fixpunkt von $G$. Bei stetigen Funktionen ist dieser Fixpunkt sogar der kleinste (nach Kleene). Somit wird der gesuchte Operator $fix$ schließlich wie folgt verwirklicht:

$$fix\ G \;:=\; \bigsqcup i.\ G^i\ \bot \qquad (\mathbf{N}\ 5.3.5)$$

Damit gilt sofort die gewünschte Äquivalenz (N 5.3.2):

$$f \;=\; fix\ G \qquad (\mathbf{N}\ 5.3.6)$$

$f\ x$ kann für jede beliebige endliche Eingabe $x$ so berechnet werden, dass ein passendes Kettenelement aus der obigen Kette der Funktionen ausgewählt und anschließend auf $x$ angewendet wird. Für eine unendliche Eingabe kann die Ausgabe dank der Stetigkeit durch analogen Prozess beliebig (also insbesondere auch unendlich oft, wodurch eine unendliche Ausgabe erzeugt wird) echt angenähert werden.

Der gesamte obige Sachverhalt wird in HOLCF ausgenutzt, um eine eigene Syntax für die Einführung von HOLCF-Funktionen zur Verfügung zu stellen. Die obige literaturübliche Abbildung $fix$ realisiert in HOLCF der Fixpunktoperator `fix`, der in der HOLCF-Theorie *Fix* entwickelt wird. Dort sind auch die wesentlichen charakterisierenden Lemmata bezüglich von `fix` eingeführt, so z.B. das folgende Lemma:

```
f ≡ fix·G ⟹ f = G·f                                    (fix_eq2)
```

das eine Entfaltung einer solchen HOLCF-Funktionsdefinition ermöglicht und damit auch die gewünschte rekursive Charakterisierung der jeweiligen Funktion liefert.

Ferner müssen alle in einer solchen Definition beteiligten Konstrukte HOLCF-Konstrukte sein, wie etwa HOLCF-Typen und HOLCF-Funktionen, falls die Stetigkeitsbehandlung automatisch erfolgen soll. Deshalb enthält HOLCF z.B. eigene Konstrukte, wie etwa den dreiwertigen Datentyp `tr` mit den Werten `TT`, `FF` und $\bot$ als eine HOLCF-Ausprägung der (gelifteten) booleschen Werte. Das Konstrukt `If ... then ... else ... fi` ist ein HOLCF-Konstrukt für das HOL-sche `if ... then ... else ...`, bei dem anstatt von HOL-Typen an den jeweilligen Stellen die entsprechenden HOLCF-Typen eingesetzt werden müssen. Einige Anwendungsbeispiele dieser HOLCF-Syntax im Bezug auf die stromverarbeitende Funktionen finden sich in Kapitel 6. Die obige Definitionssyntax wird ausführlich in [Reg94, Reg95, MNvOS99] beschrieben. Eine sehr gute Einführung in die Fixpunkttheorie im Kontext der Informatik findet sich in [Bro98]. Ferner finden sich die Grundlagen der Fixpunkttheorie in [Tar55, Kle52].

Für eine übersichtlichere Definition von stromverarbeitenden Funktionen führen wir ergänzend das folgende syntaktische Konstrukt ein, das die Fallunterscheidung auf Fstreams auf der Basis unserer Charakterisierung über die einelementige Fstreams vereinfacht:

```
scase_short   :: "'a fstream ⇒ ('a ⇒ 'b fstream) ⇒ 'b fstream"  (σ_. _)
scase_short_def: "scase_short s eq ≡ (case ft·s of ⊥ ⇒ ⊥ | ↑a ⇒ eq a)"
                                                        (Def 5.3.0)
```

Mit der obigen Definition führen wir den binären Operator $\sigma$ in Präfixnotation ein, der eine Fallunterscheidung auf dem ersten Argument veranlasst. Falls der übergebene Strom $s$ leer ist, dann wird $\sigma$ zu $\bot$ ausgewertet (strikt). Falls $s$ nicht leer ist, dann wird das Ergebnis entsprechend der Funktion im zweiten Argument geliefert. Dabei nimmt eine solche Funktion das erste Zeichen des Stromes $s$ als Argument. Es ergibt sich das folgende Lemma:

```
σs. eq = (case ft·s of ⊥ ⇒ ⊥ | ↑a ⇒ eq a)              (scase_short_simp)
```

Das Lemma wird der Menge der Simplifikationsregeln hinzugefügt. Die Anwendungsbeispiele finden sich in Abschnitt 6.

## 5.4 Stromverarbeitende Funktionen mit Hilfe der stetigen HOL-Funktionen

Eine weitere Möglichkeit eine stromverarbeitende Funktion auf der Basis der vorliegenden Umsetzung zu definieren, ist die, die Funktion mit dem HOL-Funktionstypkonstruktor ($\Rightarrow$) auf HOLCF-Datentypen einzuführen. Die Stetigkeit einer solchen Funktion muss in einem solchen Fall bewiesen und anschließend explizit berücksichtigt werden (siehe auch die Abschnitte 2.1 und 2.3). Ein Beispiel liefert der Konkatenationsoperator durch die Stetigkeit im zweiten Argument, die aber im zugehörigen Funktionstyp nicht berücksichtigt wird. (siehe die Abschnitte 4.7.1 und 4.7.2).

Ferner kann eine stromverarbeitende Funktion mit dem HOLCF-Funktionstypkonstruktor ($\rightarrow$) eingeführt werden, dabei jedoch die HOLCF-Definitionssyntax aus dem vorhergehenden Abschnitt nur zum Teil beachtet werden bzw. es werden auch HOL-Konstrukte in der entsprechenden Definition eingesetzt. Auch bei einer solchen Definition sind die Stetigkeitseigenschaften mindestens für die HOL-Teilkonstrukte der Definition explizit vom Benutzer zu beweisen. Diese Möglichkeit wurde auch in den Abschnitten 2.1 und 2.3 ausführlich beschrieben.

# 6 Eine Bibliothek von Funktionen für Fstreams

In diesem Kapitel wird eine grundlegende Bibliothek von rekursiven und stromverarbeitenden Funktionen auf der Basis des Typkonstruktors `fstream` definiert und die jeweiligen charakterisierenden Entfaltungslemmata dieser Funktionen eingeführt. Durch die Komposition dieser Funktionen lassen sich weitere rekursive bzw. stromverarbeitende Funktionen, wie sie typischerweise bei der strombasierten Verifikation vorkommen, noch einfacher einführen.

## 6.1 Punktweise Abbildung von Strömen

Für die punktweise Abbildung eines Eingabestromes auf einen Ausgabestrom auf der Basis einer passenden Abbildung $f$ wird mit der folgenden Fixpunktgleichung die Funktion `map` definiert:

```
map   :: "('a ⇒ 'b) ⇒ 'a fstream → 'b fstream"
map_def: "map f ≡ fix·(Λh s. σs. (λa. (<(f a)>)⌢(h·(rt·s))))"
```

(**Def** 6.1.0)

Die in der obigen Definition eingesetzten Konstrukte wurden ausführlich in Abschnitt 5.3 erläutert. Die obige Fixpunktgleichung wird mit dem Lemma `fix_eq2` (Abschnitt 5.3) wie folgt in ein rekursives Entfaltungslemma transformiert:

```
map f = (Λs. σs. (λa. (<(f a)>)⌢(map f·(rt·s))))
```

(`map_def_unfold`)

Das obige Entfaltungslemma wird wie folgt bewiesen:

```
lemma map_def_unfold:
  "map f = (Λs. σs. (λa. (<(f a)>)⌢(map f·(rt·s))))"
apply (subst map_def [THEN fix_eq2])
by (simp add: cfsconc_lemma [THEN sym])
```

(**B** 6.1.0)

Die Funktion $f$ wird auf jedes Element des Eingabestromes der Reihe nach angewandt. Der jeweilige Funktionswert wird in den Ausgabestrom geschrieben. Diese Funktion ist insbesondere für Typkonversionen nützlich darunter auch für Projektionen von Strömen, deren Elementstruktur einem Tupel entspricht. Mit dem obigen Entfaltungslemma ergibt sich die folgende rekursive Charakterisierung von `map`:

```
map f·⊥ = ⊥
map f·(<x>⌢xs) = <(f x)>⌢(map f·xs)
```

(`map_simps`)

## 6.2 Projektionen

Mit Hilfe der vorhergehenden Funktion lässt sich die punktweise erste Projektion für Ströme deren Elemente geordnete Paare sind wie folgt definieren:

```
Proj1   :: "('a * 'b) fstream → 'a fstream"
Proj1_def: "Proj1 ≡ Λx. map fst·x"
```

(**Def** 6.2.0)

Die Funktion `fst` ist eine HOL-Funktion. Sie selektiert die erste Komponente eines geordneten Paares. Es ergibt sich die folgende rekursive Gleichung für `Proj1`:

```
Proj1·⊥ = ⊥
Proj1·(<(a,b)>⌢s) = <a>⌢(Proj1·s)
```

(Proj1_simps)

Analog zur obigen Definition lässt sich die zweite Projektion zur Selektion der zweiten Komponente eines geordneten Paares einführen:

```
Proj2   :: "('a * 'b) fstream → 'b fstream"
Proj2_def: "Proj2 ≡ Λx. map snd·x"
```

(**Def** 6.2.1)

Es ergibt sich die folgende rekursive Charakterisierung:

```
Proj2·⊥ = ⊥
Proj2·(<(a,b)>⌢s) = <b>⌢(Proj2·s)
```

(Proj2_simps)

## 6.3 Filterung von Strömen auf der Basis einer Zeichenmenge

Die folgende Definition stellt die sogenannte Filterfunktion dar, die für eine Menge $M$ und einen Eingabestrom $s$ für jedes Zeichen in $s$ prüft, ob das Zeichen in $M$ enthalten ist. Falls das aktuelle Zeichen ein Element von $M$ ist dann verbleibt es im Ausgabestrom, sonst kommt es im Ausgabestrom nicht vor.

```
filter   :: "'a set ⇒ 'a fstream → 'a fstream"
filter_def: "filter M ≡ fix·(Λh s.
       σs. (λa. (If ↑(a ∈ M) then (<a>)⌢(h·(rt·s)) else h·(rt·s) fi)))"
```

(**Def** 6.3.0)

Durch die der Entwicklung der Funktion `map` analoge Vorgehensweise wird die rekursive Charakterisierung von `filter` erreicht:

```
filter M·⊥ = ⊥
a ∈ M  ⟹  filter M·(<a>⌢s) = <a>⌢(filter M·s)
a ∉ M  ⟹  filter M·(<a>⌢s) = filter M·s
```

(filter_unfolds)

## 6.4 Zippen zweier Ströme

Um aus zwei Strömen einen zu bekommen dessen Elementstruktur einem geordneten Paar mit dem ersten Element aus dem ersten und dem zweiten Element aus dem zweiten Argument entspricht, wird die dementsprechende Funktion `zip` mit der folgenden Fixpunktgleichung eingeführt:

```
zip   :: "'a fstream → 'b fstream → ('a * 'b) fstream"
zip_def: "zip ≡ fix·(Λh s1 s2.
                  σs1. (λa. (σs2. (λb. (<(a,b)>)⌢h·(rt·s1)·(rt·s2)))))"
```
(**Def** 6.4.0)

Die Funktion wird wie folgt rekursiv charakterisiert:

```
zip·s·⊥ = ⊥
zip·⊥·s = ⊥
zip·(<x>⌢xs)·(<y>⌢ys) = <(x,y)>⌢zip·xs·ys
```
(zip_simps)

## 6.5 Reissverschlussartiges Mergen zweier Ströme

Unter dem reissverschlussartigen Mergen zweier Ströme verstehen wir die punktweise Verzahnung (engl. interleaving) dieser Ströme, sodass im Ergebnisstrom nach dem $n$-ten Element aus dem ersten Strom immer das $n$-te Element aus dem zweiten Strom vorkommt. Sobald das Ende eines Argumentstroms erreicht wird, wird die weitere Verzahnung abgebrochen. Die Funktion `uMerge` wird wie folgt formalisiert:

```
uMerge   :: "'a fstream → 'a fstream → 'a fstream"
uMerge_def: "uMerge ≡ fix·(Λh s1 s2.
                  σs1. (λa. (σs2. (λb. <a>⌢<b>⌢h·(rt·s1)·(rt·s2)))))"
```
(**Def** 6.5.0)

Es ergibt sich die folgende rekursive Charakterisierung von `uMerge`:

```
uMerge·⊥·s = ⊥
uMerge·s·⊥ = ⊥
uMerge·(<x>⌢xs)·(<y>⌢ys) = <x>⌢<y>⌢uMerge·xs·ys
```
(uMerge_unfolds)

## 6.6 Selektion eines Anfangsstücks

Die Funktion `tWhile` selektiert das längste Anfangsstück eines Stromes auf dem das als Argument übergebene Prädikat punktweise erfüllt wird:

```
tWhile   :: "('a ⇒ bool) ⇒ 'a fstream → 'a fstream"
tWhile_def: "tWhile p ≡ fix·(Λh s.
                σs. (λa. If ↑(p a) then (<a>)⌢h·(rt·s) else ⊥ fi))"
```
(**Def** 6.6.0)

tWhile wird wie folgt rekursiv charakterisiert:

```
tWhile p·⊥ = ⊥
¬(p x) ⟹ tWhile p·(<x>⌢xs) = ⊥
p x ⟹ tWhile p·(<x>⌢xs) = <x>⌢(tWhile p·xs)
```
(tWhile_unfolds)

## 6.7 Löschen eines Anfangsstücks

Die Funktion dWhile löscht das längste Anfangsstück eines Stromes auf dem das als Argument übergebene Prädikat punktweise erfüllt wird. Sie ist also das Gegenstück der im vorhergehenden Abschnitt eingeführten Funktion tWhile.

```
dWhile   :: "('a ⇒ bool) ⇒ 'a fstream → 'a fstream"
dWhile_def: "dWhile f ≡ fix·(Λh s.
               σs. (λa. If ↑(f a) then h·(rt·s) else s fi))"
```
(**Def** 6.7.0)

dWhile wird wie folgt rekursiv charakterisiert:

```
dWhile f·⊥ = ⊥
¬(f x) ⟹ dWhile f·(<x>⌢xs) = <x>⌢xs
f x ⟹ dWhile f·(<x>⌢xs) = dWhile f·xs
```
(dWhile_unfolds)

## 6.8 Entfernung von unmittelbaren Zeichenwiederholungen

Die folgende Funktion entfernt die unmittelbaren Zeichenwiederholungen aus einem Strom. In Anlehnung an ähnliche Funktionen in funktionalen Programmiersprachen nennen wir sie nub.

```
nub   :: "'a fstream → 'a fstream"
nub_def: "nub ≡ fix·(Λh s. σs. (λa. (<a>)⌢h·(dWhile (λz. z = a)·(rt·s))))"
```
(**Def** 6.8.0)

nub wird wie folgt rekursiv charakterisiert:

```
nub·⊥ = ⊥
nub·(<x>) = <x>
nub·(<x>⌢<x>⌢s) = nub·(<x>⌢s)
x ≠ y ⟹ nub·(<x>⌢<y>⌢s) = <x>⌢(nub·(<y>⌢s))
```
(nub_unfolds)

## 6.9 Verflachung von Strömen auf Strömen

Im Folgenden wird die Funktion flatten eingeführt. Der Definitionsbereich von flatten ist die Menge aller Ströme auf einem Alphabet, das aus Strömen auf einem HOL-Typ besteht.

Die Funktion entliftet die erwähnten Ströme und führt anschließend die Konkatenation dieser Ströme durch.

```
flatten   :: "'a fstream fstream → 'a fstream"
flatten_def: "flatten ≡ fix·(Λh s. σs. (λfs. fs⌢h·(rt·s)))"
```

(**Def** 6.9.0)

`flatten` wird wie folgt rekursiv charakterisiert:

```
flatten·⊥ = ⊥
flatten·(<x>⌢xs) = x⌢(flatten·xs)
```

(flatten_simps)

## 6.10 Rekursionsmuster für Ströme

Die im Folgenden definierte Funktion `scanl` kann für eine zweistellige Funktion (bzw. Funktionssymbol) $\oplus$, einen Startwert $q$ und einen festen Strom $<x_0>^\frown<x_1>^\frown<x_2>^\frown \cdots$ informell wie folgt veranschaulicht werden:

$$\texttt{scanl} \oplus q\ (<x_0>^\frown<x_1>^\frown<x_2>^\frown \cdots) = <q>^\frown<q \oplus x_0>^\frown<(q \oplus x_0) \oplus x_1>^\frown<((q \oplus x_0) \oplus x_1) \oplus x_2>^\frown \cdots$$

(**N** 6.10.0)

`scanl` wird mit der in Abschnitt 5.1 eingeführten Definitionsmethode wie folgt formalisiert:

```
rec_scanl:: "nat ⇒ ('a ⇒ 'b ⇒ 'a) ⇒ 'a ⇒ 'b fstream → 'a fstream"
primrec
 rec_scanl_0:  "rec_scanl 0 f q s = <q>"
 rec_scanl_Suc:"rec_scanl (Suc i) f q s =
                  (case ft·s of
                        ⊥  ⇒ <q> |
                        ↑a ⇒ <q>⌢(rec_scanl i f (f q a) (rt·s)))"
```

(**Def** 6.10.0)

Schließlich definieren wir mit Hilfe der obigen Funktion:

```
scanl    :: "('a ⇒ 'b ⇒ 'a) ⇒ 'a ⇒ 'b fstream → 'a fstream"
scanl_def : "scanl f q ≡ Λs. (⨆i. rec_scanl i f q s)"
```

(**Def** 6.10.1)

Die im Lemma `spfAsLubLemma` spezifizierten Eigenschaften werden einfach mit der natürlichen Induktion bewiesen bzw. sehen wie folgt aus:

```
∀f a x. rec_scanl n f a x ⊑ rec_scanl (Suc n) f a x
```

(rscanl_chain)

```
∀f q s. rec_scanl n f q s = rec_scanl n f q (take n·s)
```

(rscanl_contlub)

```
∀f q x y. x ⊑ y ⟶ rec_scanl n f q x ⊑ rec_scanl n f q y
```

(rscanl_mono)

Mit dem Lemma spfAsLubLemma ergibt sich die folgende Charakterisierung von scanl:

```
scanl f q·⊥ = <q>
scanl f q·(<x>⌢xs) = <q>⌢(scanl f (f q x)·xs)
```
(scanl_simps)

## 6.11 Iterierte Bildung von Strömen

Unter der iterierten Bildung von Strömen verstehen wir die Bildung eines Stromes auf der Basis einer Funktion $f$ und eines Startwertes $q$. Eine solche Funktion, die wir im Weiteren eiterate nennen, wird wie folgt informell beschrieben:

$$\texttt{eiterate}\ f\ q\ =\ <q>^\frown<f\ q>^\frown<f\ (f\ q)>^\frown<f\ (f\ (f\ q))>^\frown\cdots \qquad (\textbf{N}\ 6.11.0)$$

Die Funktion eiterate wird wie folgt formalisiert:

```
rec_eiterate :: "nat ⇒ ('a ⇒ 'a) ⇒ 'a ⇒ 'a fstream"
primrec
 rec_eiterate_0:  "rec_eiterate 0 f a = ⊥"
 rec_eiterate_Suc:"rec_eiterate (Suc i) f a = <a>⌢(rec_eiterate i f (f a))"
```
(**Def** 6.11.0)

Schließlich definieren wir mit Hilfe der obigen Funktion:

```
eiterate     :: "('a ⇒ 'a) ⇒ 'a ⇒ 'a fstream"
eiterate_def  : "eiterate f x ≡ ⨆i. rec_eiterate i f x"
```
(**Def** 6.11.1)

Die Ketteneigenschaft von rec_eiterate wird mit der natürlichen Induktion bewiesen. Es ergibt sich mit dem Lemma LubRecLemma (Abschnitt 5.1.1) die folgende Entfaltungsregel:

```
eiterate f x = <x>⌢(eiterate f (f x))
```
(eiterate_unfold)

## 6.12 Bildung periodischer Ströme

Für die Bildung eines unendlichen, zyklischen Stromes mit der Struktur $s^\frown s^\frown s^\frown s\cdots$ auf der Basis eines Argumentstromes $s$ führen wir die Funktion iCycle ein:

```
rec_iCycle :: "nat ⇒ 'a fstream ⇒ 'a fstream"
primrec
 rec_iCycle_0:  "rec_iCycle 0 s = ⊥"
 rec_iCycle_Suc:"rec_iCycle (Suc i) s = s⌢(rec_iCycle i s)"
```
(**Def** 6.12.0)

Schließlich definieren wir mit der obigen Hilfsfunktion:

```
iCycle     :: "'a fstream ⇒ 'a fstream"
iCycle_def  : "iCycle s ≡ ⨆i. rec_iCycle i s"
```

(**Def** 6.12.1)

Die Ketteneigenschaft von *rec_iCycle* wird mit der natürlichen Induktion bewiesen. `iCycle` ist nicht monoton und deshalb auch nicht stetig. Es ergibt sich mit dem Lemma `LubRecLemma` (Abschnitt 5.1.1) die folgende rekursive Charakterisierung:

```
iCycle ⊥ = ⊥
iCycle s = s⌢(iCycle s)
```

(iCycle_unfolds)

## 6.13 Mengen von Strömen

Für die Bildung von Strömen auf einer gegebenen Zeichenmenge werden abschließend noch die aus der Literatur üblichen Operatoren eingeführt. Für eine Zeichen- bzw. Nachrichtenmenge $M$ wird die Menge aller Ströme über $M$ mit der Bezeichnung $M^\omega$ wie folgt definiert:

```
allFstreams   :: "'a set ⇒ 'a fstream set"  ("_^ω")
allFstreams_def: "allFstreams M ≡ {s. filter M·s = s}"
```

(**Def** 6.13.0)

Die Zerlegung von $M^\omega$ in die Menge aller endlichen Ströme $M^*$ über $M$ und die Menge aller unendlichen Ströme $M^\infty$ über $M$ wird auf der Basis der Menge $M^\omega$ durchgeführt, indem das Prädikat `#s=∞` bzw. `#s<∞` in der jeweiligen Mengenbeschreibung durch Konjunktion hinzugenommen wird. Für die Menge aller endlichen Fstreams auf einer gegebenen Menge $M$ ergibt sich:

```
finFstreams   :: "'a set ⇒ 'a fstream set"  ("_^∞")
finFstreams_def: "finFstreams M ≡ {s. s ∈ M^ω ∧ #s<∞}"
```

(**Def** 6.13.1)

Für die Menge aller unendlichen Fstreams auf einer gegebenen Menge $M$ ergibt sich:

```
infFstreams   :: "'a set ⇒ 'a fstream set"  ("_^*")
infFstreams_def: "infFstreams M ≡ {s. s ∈ M^ω ∧ #s=∞}"
```

(**Def** 6.13.2)

# 7 Schlussbemerkungen

## 7.1 Zusammenfassung und Ausblick

In der vorliegenden Arbeit wurde eine komfortable Infrastruktur für die Verifikation von strombasierten Spezifikationen von verteilten, asynchron kommunizierenden, Software basierten Systemen auf der Basis der formalen Logiken Isabelle/HOL und Isabelle/HOLCF bzw. als deren Erweiterung eingeführt.

Auf der Basis des Datentypkonstruktors `stream` wurde der Datentypkonstruktor `fstream` als ein formales Modell für Ströme auf beliebigen Typen entwickelt. Danach wurde die Konkatenation auf Fstreams eingeführt und als der zentrale Konstruktionsoperator implementiert.

Auf der Basis der Konkatenation für Fstreams wurden dann die weiteren Eigenschaften bzw. Lemmata und Funktionen für Fstreams entwickelt. Insbesondere zeigte sich in der vorliegenden Arbeit, dass das Take-Lemma das hinreichende Beweisprinzip für die Entwicklung war. Die Induktionsprinzipien konnten aus dem Take-Lemma abgeleitet werden. Die HOLCF built-in Funktion `stream_finite` zeigte sich neben der expliziten Längenfunktion auch als überflüssig und wurde deshalb von Anfang an nicht berücksichtigt.

Ferner wurde anhand des Konkatenationsoperators ein neues Verfahren formalisiert, mit dem die rekursiven Funktionen auf Fstreams also auch die stromverarbeitenden Funktionen in HOLCF systematisch eingeführt werden können. Insbesondere ermöglicht uns diese Formalisierung eine weitere, dementsprechende HOLCF-Automatisierung der Definitionssyntax für diese Klasse von HOLCF-Funktionen. Neben dieser neuen Definitionssyntax wurden hier die deterministischen, buchstabierenden Automaten als eine weitere und bequeme Definitionssyntax vorgeschlagen. Diese Erweiterungen sind insbesondere deshalb wichtig, weil die stromverarbeitenden Funktionen das semantische Modell strombasierter Spezifikationen sind.

Im Gegensatz zu der strengen HOLCF-Syntax sind die hier eingeführten, alternativen Definitionskonstrukte generell sehr handhabbar, übersichtlich und HOL-bezogen. In [MNvOS99] wurde diese Problematik allgemeiner und ausführlich behandelt und dort durch die folgenden Sätze auf den Punkt gebracht worden:

> *Although HOLCF cannot overcome the complications introduced by domains and partial functions, it can delay the point where they rear their ugly head. The underlaying philosophy is to express as much as possible in the HOL basis and as much as necessary in the LCF extension ...*

Abschließend wurde in Kapitel 6 eine Bibliothek von Funktionen auf Strömen entwickelt. Für jede Funktion wurden die entsprechenden Entfaltungslemmata bewiesen bzw. aus der jeweiligen Definition abgeleitet.

Die weiteren Optimierungen der obigen Konstrukte sind Gegenstand der weiteren Entwicklung. So kann demnächst versucht werden, die Beweise der Stetigkeitseigenschaften mit Hilfe der oben vorgestellten Syntax-Erweiterungen soweit als möglich zu automatisieren. Demnächst wird ergänzend eine Formalisierung der buchstabierenden Automaten eingeführt. Ein interessanter Punkt der beim Einsatz der Infrastruktur zum Vorschein kam, ist die fehlende einheitliche Behandlung der Beweisführung für die Idempotenz der einzelnen oder der zusammengesetzten Konstrukte. Auch die Spezialisierung von Funktionen bzw. stromverarbeitenden Funktionen auf Teilmengen von korrespondierenden Typen im Definitions- bzw. Wertebereich wurde noch nicht explizit behandelt.

Die Infrastruktur wird erweitert werden, indem Konstrukte für die Beschreibung nichtdeterministischen Komponentenverhaltens eingeführt werden. Alle geeigneten Konstrukte sind ferner auf Stromtupel zu verallgemeinern. Schließlich sind auch gezeitete Ströme auf der Basis von `fstream` einzuführen. Unser Ziel ist es schließlich ein allgemeines Rahmenwerk zur formalen Spezifikation und Verifikation von verteilten, asynchron kommunizierenden Systemen zu schaffen. Mit diesem Rahmenwerk soll also die spätere Implementierung der konkreten Syntax bzw. Semantik von strombasierten Spezifikationssprachen wie Focus oder seiner Derivate auf einfache Weise möglich sein.

Die hier vorgestellte Infrastruktur bietet entscheidende Vorteile für den Aufbau des oben beschriebenen Rahmenwerks. Auf der einen Seite steht uns die umfassende formale Logik HOL zur Verfügung, die insbesondere auf der Spezifikationsseite des Rahmenwerks nützlich ist. Auf der anderen Seite bietet HOLCF mächtige Konstrukte u.a. für die Behandlung von Stetigkeit, Zulässigkeit und der potentiellen Unendlichkeit der Ströme. Dadurch werden wesentliche Automatisierungswerkzeuge für den Semantik- und den Verifikationsteil des Rahmenwerks zur Verfügung gestellt.

## 7.2 Diskussion

Das tragende Konstrukt dieser Umsetzung ist der Datentypkonstruktor `stream`, der Konstruktor für partielle oder unendliche Sequenzen auf beliebigen pcpo's in HOLCF. Die grundsätzlichen Eigenschaften dieses Konstruktors führten uns zu der Annahme, dass die Schnittstelle zwischen HOL und HOLCF als der Schwerpunkt der Umsetzung eine sehr flexible und fruchtbare Basis bieten würde. Dies war insbesondere deshalb möglich, weil HOLCF die Logik HOL konservativ erweitert. Der Bericht bietet deshalb ergänzend einen tiefen und praktischen Einblick in die Schnittstelle zwischen HOL und HOLCF bzw. der Logik höherer Stufe und des Kalküls für berechenbare Funktionen.

Am Beispiel des nichtstetigen Konkatenationsoperators wurden die Eigenschaften von `fstream` bzw. `stream` deutlich. Zum abschließenden Vergleich sei hier eine HOL-Definition der Konkatenation (siehe auch z.B. [SS95]) mit der in der Definition Def 4.3.1 bereits angegebenen Signatur wie folgt angegeben:

```
hol_fsconc_def:
  "hol_fsconc s1 s2 ≡ if (∃n. take n·s1 = s1)
                         then  (SOME s. ∃n. (take n·s = s1) ∧
                                            (drop n·s = s2))
                         else s1"
```

(**Def** 7.2.0)

Die Existenz eines entsprechenden Ergebnisstromes im endlichen Fall kann durch Induktion bewiesen werden. Abschließend ergibt sich die Äquivalenz der beiden Funktionen. Für die Implementierung sind hier zusätzlich eine vorherige Entwicklung des Operators `drop` und ein geübter Umgang mit dem HOL-Deskriptionsoperator `SOME` notwendig.

Unsere Implementierung der Konkatenation in Def 4.3.0 bzw. Def 4.3.1 als auch das obige Beispiel sollten im vorliegenden Bericht einen anschaulichen Charakter haben, deshalb wurde die Stetigkeit im zweiten Argument nicht durch die dementsprechende HOLCF-Typkonstruktion für stetige Funktionen begleitet. Dies ist aus Übersichtlichkeitsgründen hier durch das neue Konstrukt `cfsconc_def` (Def 4.7.0) und die Äquivalenz im Lemma `cfsconc2fsconc` (Abschnitt 4.7.2) erreicht worden. Eine solche bringt den Vorteil der automatischen Behandlung der Stetigkeit durch HOLCF, weil HOLCF die Stetigkeit im jeweiligen Typ verankert.

## 7.3 Danksagung

Das Entstehen des Berichts wäre ohne das Isabelle-System nicht möglich, deshalb bedanken wir uns bei Isabelle-Entwicklungsteams von der Technischen Universität München und der University of Cambridge für die Entwicklung dieses großartigen Theorembeweisers. Bei Maria Spichkova bedanken wir uns für Diskussion in Zusammenhang mit Focus und die konstruktiven Kommentare auf die Urversion der vorliegenden Arbeit. Bei David von Oheimb bedanken wir uns für Zusammenarbeit an einer Urversion der Infrastruktur. Bei unseren Studenten Tim Gülke und Jan Oliver Ringert bedanken wir uns für das Probelesen und die anschließende Kritik der vorliegenden Arbeit.

# Literaturverzeichnis


[And02] P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*, volume 27 of *Applied Logic Series*. Springer, 2002.

[BDD+92a] M. Broy, F. Dederich, C. Dendorfer, M. Fuchs, T. Gritzner, and R. Weber. The Design of Distributed Systems - An Introduction to FOCUS. Technical Report TUM-I9202, Technische Universität München, 1992.

[BDD+92b] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T.F. Gritzner, and R. Weber. Summary of Case Studies in FOCUS - a Design Method for Distributed Systems. Technical Report TUM-I9203, Technische Universität München, 1992.

[Bir98] R. Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall Series in Computer Science. Prentice Hall, second edition, 1998.

[Bro98] M. Broy. *Informatik. Eine grundlegende Einführung. Band 1. Programmierung und Rechnerstrukturen. 2. Auflage*. Springer Verlag, 1998.

[BS96] M. Broy and G. Stefanescu. The Algebra of Stream Processing Functions. Technical Report TUM-I9620, Technische Universität München, 1996.

[BS01] M. Broy and K. Stoelen. *Specification and Development of Interactive Systems. Focus on Streams, Interfaces and Refinement*. Springer Verlag Heidelberg, 2001.

[BT06] R. Bogenberger and D. Trachtenherz. Qualitätssteigerung der Automotive-Software durch formale Spezifikation funktionaler Eigenschaften auf der Abstraktionsebene des Modellentwurfs. In H. C. Mayr and R. Breu, editors, *Modellierung 2006*, volume P-82 of *LNI*, pages 35–49. BMW Group Forschung und Technik, 2006.

[DGM97] M. Devillers, W. O. D. Griffioen, and O. Müller. Possibly Infinite Sequences in Theorem Provers: A Comparative Study. In E. L. Gunter and A. P. Felty, editors, *TPHOLs*, volume 1275 of *Lecture Notes in Computer Science*, pages 89–104. Springer, 1997.

[DP02] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, Cambridge, 2002.

[DW92] C. Dendorfer and R. Weber. Development and Implementation of a Communication Protocol - An Exercise in FOCUS. Technical Report TUM-I9205, Technische Universität München, 1992.

[EFT96] H. D. Ebbinghaus, J. Flum, and W. Thomas. *Einführung in die mathematische Logik*. Spektrum, 1996.

[Fit96] M. Fitting. *First-Order Logic and Automated Theorem Proving*. Springer, second edition, 1996.

[Gun92] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing. MIT Press, 1992.

[HSSS96] F. Huber, B. Schätz, A. Schmidt, and K. Spies. AutoFocus - A Tool for Distributed Systems Specification. In B. Jonsson and J. Parrow, editors, *Proceedings FTRTFT'96 - Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 467–470. LNCS 1135, Springer Verlag, 1996.

[Kle52] S. C. Kleene. *Introduction to Metamathematics*. Van Nostrand, 1952.

[Krü00] I. Krüger. *Distributed System Design with Message Sequence Charts*. Doktorarbeit, Technische Universität München, 2000.

[LS84] J. Loeckx and K. Sieber. *The Foundations of Program Verification*. Wiley-Teubner series in computer science, 1984.

[MNvOS99] O. Müller, T. Nipkow, D. von Oheimb, and O. Slotosch. HOLCF = HOL + LCF. *Journal of Functional Programming*, 9(2):191–223, 1999.

[Mün06] Lehrstuhl Software & Systems Engineering. Technische Universität München. Isabelle-Homepage at http://isabelle.in.tum.de/, 2006.

[NPW02] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. LNCS 2283, Springer Heidelberg, 2002.

[Pau87] L. C. Paulson. *Logic and computation: interactive proof with Cambridge LCF*. Cambridge University Press, New York, NY, USA, 1987.

[Pau94] L. C. Paulson. *Isabelle - A Generic Theorem Prover (with a contribution by T. Nipkow)*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.

[Pau96] L. C. Paulson. *ML for the working programmer (2nd ed.)*. Cambridge University Press, New York, NY, USA, 1996.

[Pau03a] L. C. Paulson. *Introduction to Isabelle*. Computer Laboratory, University of Cambridge, 2003.

[Pau03b] L. C. Paulson. *The Isabelle Reference Manual. With Contributions by Tobias Nipkow and Markus Wenzel*. Computer Laboratory, University of Cambridge, 2003.

[Plo77] G. D. Plotkin. LCF Considered as a Programming Language. *Theoretical Computer Science*, 5(3):225–255, 1977.

[Reg94] F. Regensburger. *HOLCF: Eine konservative Erweiterung von HOL um LCF*. Doktorarbeit, Technische Universität München, 1994.

[Reg95] F. Regensburger. HOLCF: Higher Order Logic of Computable Functions. In E. T. Schubert, P. J. Windley, and J. Alves-Foss, editors, *TPHOLs*, volume 971 of *Lecture Notes in Computer Science*, pages 293–307. Springer, 1995.

[RK99] B. Rumpe and C. Klein. Automata Describing Object Behavior. In *Object-Oriented Behavioral Specifications*, pages 265–287. Kluwer Academic Publishers, Norwell, Massachusetts, 1999.

[Rum96] B. Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Herbert Utz Verlag Wissenschaft, 1996.

[Sch00] U. Schöning. *Logik für Informatiker, 5. Auflage*. Spektrum Akademischer Verlag, 2000.

[Sco76] D. Scott. Data types as lattices. *SIAM Journal of Computing*, 34(5):522–587, 1976.

[Sco82] D. Scott. Domains for denotational semantics. In *Proceedings of the 9th Colloquium on Automata, Languages and Programming*, pages 577–613, London, UK, 1982. Springer-Verlag.

[SS95] B. Schätz and K. Spies. Formale Syntax zur logischen Kernsprache der FOCUS-Entwicklungsmethodik. Technical Report TUM-I9529, Technische Universität München, 1995.

[Ste97] R. Stephens. A Survey of Stream Processing. *Acta Informatica*, 34(7):491–541, 1997.

[Tar55] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[Tho99] S. Thompson. *The Haskell: The Craft of Functional Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[Win93] G. Winskel. *The Formal Semantics of Programming Languages*. Foundations of Computing. The MIT Press, Cambridge, Massachusetts, 1993.

Technische Universität Braunschweig
Informatik-Berichte ab Nr. 2002-06

| | | |
|---|---|---|
| 2002-06 | H. G. Matthies, J. Steindorf | Strong Coupling Methods |
| 2002-07 | H. Firley, U. Goltz | Property Preserving Abstraction for Software Verification |
| 2003-01 | M. Meyer, H. G. Matthies | Efficient Model Reduction in Non-linear Dynamics Using the Karhunen-Loève Expansion and Dual-Weighted-Residual Methods |
| 2003-02 | C. Täubner | Modellierung des Ethylen-Pathways mit UML-Statecharts |
| 2003-03 | T.-P. Fries, H. G. Matthies | Classification and Overview of Meshfree Methods |
| 2003-04 | A. Keese, H. G. Matthies | Fragen der numerischen Integration bei stochastischen finiten Elementen für nichtlineare Probleme |
| 2003-05 | A. Keese, H. G. Matthies | Numerical Methods and Smolyak Quadrature for Nonlinear Stochastic Partial Differential Equations |
| 2003-06 | A. Keese | A Review of Recent Developments in the Numerical Solution of Stochastic Partial Differential Equations (Stochastic Finite Elements) |
| 2003-07 | M. Meyer, H. G. Matthies | State-Space Representation of Instationary Two-Dimensional Airfoil Aerodynamics |
| 2003-08 | H. G. Matthies, A. Keese | Galerkin Methods for Linear and Nonlinear Elliptic Stochastic Partial Differential Equations |
| 2003-09 | A. Keese, H. G. Matthies | Parallel Computation of Stochastic Groundwater Flow |
| 2003-10 | M. Mutz, M. Huhn | Automated Statechart Analysis for User-defined Design Rules |
| 2004-01 | T.-P. Fries, H. G. Matthies | A Review of Petrov-Galerkin Stabilization Approaches and an Extension to Meshfree Methods |
| 2004-02 | B. Mathiak, S. Eckstein | Automatische Lernverfahren zur Analyse von biomedizinischer Literatur |
| 2005-01 | T. Klein, B. Rumpe, B. Schätz (Herausgeber) | Tagungsband des Dagstuhl-Workshop MBEES 2005: Modellbasierte Entwicklung eingebetteter Systeme |
| 2005-02 | T.-P. Fries, H. G. Matthies | A Stabilized and Coupled Meshfree/Meshbased Method for the Incompressible Navier-Stokes Equations — Part I: Stabilization |
| 2005-03 | T.-P. Fries, H. G. Matthies | A Stabilized and Coupled Meshfree/Meshbased Method for the Incompressible Navier-Stokes Equations — Part II: Coupling |
| 2005-04 | H. Krahn, B. Rumpe | Evolution von Software-Architekturen |
| 2005-05 | O. Kayser-Herold, H. G. Matthies | Least-Squares FEM, Literature Review |
| 2005-06 | T. Mücke, U. Goltz | Single Run Coverage Criteria subsume EX-Weak Mutation Coverage |
| 2005-07 | T. Mücke, M. Huhn | Minimizing Test Execution Time During Test Generation |
| 2005-08 | B. Florentz, M. Huhn | A Metamodel for Architecture Evaluation |
| 2006-01 | T. Klein, B. Rumpe, B. Schätz (Herausgeber) | Tagungsband des Dagstuhl-Workshop MBEES 2006: Modellbasierte Entwicklung eingebetteter Systeme |
| 2006-03 | B. Gajanovic, B. Rumpe | Isabelle/HOL-Umsetzung strombasierter Definitionen zur Verifikation von verteilten, asynchron kommunizierenden Systemen |