Verfeinerung in der Objektorientierten Programmkonstruktion

Emil Sekerinski

Verfeinerung in der objektorientierten Programmkonstruktion

Zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

von der Fakultät für Informatik

an der Universität Karlsruhe (Technische Hochschule)

genehmigte

Dissertation

von

Emil Sekerinski

aus Belgrad

Tag der mündlichen Prüfung: 19. Dezember 1994

Erster Gutachter: Prof. Dr. G. Goos

Zweiter Gutachter: Prof. Dr. P. Deussen

Kurzfassung

Die vorliegende Arbeit gibt eine formale Fundierung objektorientierter Entwurfsprinzipien. Hierzu wird eine Notation definiert, die zum einen alle wesentlichen Elemente objektorientierter Programmiersprachen enthält, und zum anderen Spezifikationskonstrukte bietet. In dieser Notation werden folgende typische objektorientierte Entwurfsprinzipien betrachtet:

- Implementieren von Klassen
- Inkrementelles Modifizieren von Klassen durch Vererbung
- Herausfaktorisieren von Gemeinsamkeiten von Klassen

Es werden Beweis- und Konstruktionsregeln für die korrekte Anwendung dieser Entwurfsprinzipien aufgestellt. Die Regeln tragen auch zu einem tieferen Verständnis der Entwurfsprinzipien bei.

Spezifikationen haben die Form von indeterministischen Anweisungen, wie sie aus dem Verfeinerungskalkül bekannt sind, bei denen das Resultat durch ein Prädikat spezifiziert wird. An objektorientierten Konzepten werden Objekte mit Attributen und Methoden, Kapselung privater Attribute, Klassen mit Vererbung, Untertyp-Polymorphie, parametrische Polymorphie und Objektidentitäten betrachtet. Für die Semantik der Notation wird das Typsystem $F^{\omega}_{\leq}$, eine Erweiterung des typisierten λ-Kalküls, zugrunde gelegt. Anweisungen werden durch Prädikatentransformer definiert. Damit wird gezeigt, wie eine Notation, die bezüglich der Spezifikationsmächtigkeit und der Flexibilität der Typisierung über existierende typisierte objektorientierte Programmier- und Spezifikationssprachen hinausgeht, formal definiert werden kann. Die Flexibilität beruht insbesondere auf der Unterscheidung von Objekttypen und Klassen.

Das zentrale methodische Konzept ist die Klassenverfeinerung. Sie erlaubt es, abstrakte, spezifizierende Klassen durch konkretere, implementierungsnähere zu ersetzen. Es wird gezeigt, daß Klassenverfeinerung durch die aus dem Verfeinerungskalkül bekannte Technik der Datenverfeinerung nachgewiesen werden kann. Für die Verfeinerung von inkrementell modifizierten Klassen werden spezielle, vereinfachte Beweisregeln aufgestellt. Das Herausfaktorisieren von Gemeinsamkeiten von Klassen wird durch zwei neuartige Operatoren, Klassenschnitt und Klassenvereinigung, unterstützt.

Danksagung

Als erstes möchte ich mich bei Prof. Dr. G. Goos, meinem Betreuer über viele Jahre hinweg, bedanken. Seine fortwährenden Kommentare zur Arbeit haben entscheidend zur klareren Strukturierung und einsichtigeren Präsentation des Ansatzes geführt. Die Ausrichtung der Arbeit, eine Balance zwischen Theorie und Praxis zu halten, wurde von ihm geprägt. Prof. Dr. P. Deussen hat dankenswerterweise das Korreferat übernommen und auf eine Reihe von Unzulänglichkeiten in der Darstellung hingewiesen.

Bei Prof. Dr. M. Odersky möchte ich mich für sein Interesse an der Arbeit und seine spontane Bereitschaft, sich intensiv mit ihr zu beschäftigen, bedanken. Er hat mich auch auf einige verwandte Arbeiten aufmerksam gemacht. Während eines 3-monatigen Aufenthaltes bei Prof. Dr. R. Back in Turku war es mir möglich, mehr über den Verfeinerungskalkül zu lernen, über meinen Ansatz zu diskutieren und Unterstützung für meine Ideen zu finden. An dieser Stelle möchte ich mich insbesondere bei C. Lewerentz und R. Back bedanken, daß sie diesen Aufenthalt finanziell unterstützt haben.

Meine Kollegen am FZI, insbesondere meine Mitarbeiter Andreas Rüping, Claus Lewerentz, Franz Weber und Thomas Lindner aus dem Korso Projekt waren über Jahre hinweg geduldige Diskussionspartner beim Vorstellen zahlreicher unreifer Ideen. Viele Einsichten zur objektorientierten Programmierung und deren Formalisierung sind aus der Zusammenarbeit mit ihnen entstanden. Die detaillierten Kommentare von Walter Zimmer zu einer früheren Fassung haben zu vielen Verbesserungen geführt.

Inhaltsverzeichnis

1 Einleitung

Die vorliegende Arbeit gibt eine formale Fundierung objektorientierter Entwurfsprinzipien. Es werden *Beweis- und Konstruktionsregeln* aufgestellt, mit denen die *Korrektheit* von objektorientierten Entwürfen sichergestellt werden kann. Damit wird zur *Zuverlässigkeit* objektorientierter Programme beigetragen. Die Regeln tragen auch zu einem tieferen Verständnis der Entwurfsprinzipien bei.

Objektorientierte Programmkonstruktion

Die objektorientierte Programmkonstruktion beginnt typischerweise mit der Analyse des Problemgebietes. Darin werden *Objekte* identifiziert und später auf Objekte im Programm abgebildet. Das Verhalten von gleichartigen Objekten wird durch *Klassen* beschrieben. Dieses Vorgehen hat seinen Ursprung in der *Simulation* (Dahl & Nygaard, 1966), bei der eine 1:1 Korrespondenz von Objekten der realen Welt mit Objekten im Programm natürlich ist. Dieses Vorgehen hat sich auch bei Informationssystemen, Steuerungen, Benutzerschnittstellen, Betriebssystemen etc. als nützlich erwiesen. (Objekte entsprechen dann nicht mehr notwendigerweise Dingen der realen Welt, sind aber hilfreiche *Abstraktionen*.) Typische Prinzipien für den Entwurf von Klassen sind:

Implementieren von Klassen. Eine problemnahe Spezifikation von Klassen ist oft nicht (effizient) ausführbar. Dann ist es notwendig, diese effizient zu implementieren.

Inkrementelle Modifikation von Klassen. Neue Klassen, die verwandt zu existierenden sind, können durch Modifikation von diesen mittels Vererbung beschrieben werden.

Herausfaktorisieren von Gemeinsamkeiten zu neuen Klassen. Objekte verschiedener Klassen können ein ähnliches, aber nicht identisches Verhalten aufweisen. In diesem Fall kann es zweckmäßig sein, das gemeinsame Verhalten in einer eigenen Klasse zu beschreiben.

Ein objektorientierter Verfeinerungskalkül

In der vorliegenden Arbeit wird für diese Entwurfsprinzipien eine theoretische Fundierung entwickelt, die das korrekte Anwenden aufzeigt. Dazu ist es notwendig, eine Notation zugrunde zu legen, in der Spezifikationen und Programme gleichermaßen formuliert werden können. Die Arbeit gliedert sich thematisch in zwei Teile: im ersten wird eine *objektorientierte Entwurfsnotation* definiert, im zweiten damit die Entwurfsprinzipien formalisiert.

Der Ansatz für eine objektorientierte Entwurfsnotation ist, in einer objektorientierten Programmiersprache als Anweisungen (z. B. in Methodenrümpfen) auch *Spezifikationen* zuzulassen, die das Resultat implizit mittels eines Prädikates spezifizieren.

Im *sequentiellen Verfeinerungskalkül* (Back & Wright, 1989; Morgan, 1990; Morris, 1987) wird die Semantik von Anweisungen und Spezifikationen auf einheitliche Weise durch Prädikatentransformer definiert. Ausführbare Anweisungen (mit Kontrollstrukturen ähnlich zu Dijkstras guarded commands) werden eingebettet in einem größeren Raum nicht-ausführbarer Anweisungen.

Das hier vorgestellte *objektorientierte Verfeinerungskalkül* ist eine Erweiterung des sequentiellen Verfeinerungskalküls um charakteristische Elemente objektorientierter Sprachen (Cardelli, et al., 1989; Cox, 1986; Goldberg & Robson, 1983; Meyer, 1988; Stroustrup, 1986; Tessler, 1985):

- Objekte mit Attributen und Methoden
- Kapselung privater Attribute
- Klassen mit Vererbung

- Untertypisierung
- Parametrisierung (Generizität)
- Objektidentitäten.

Für die Semantik wird das Typsystem $F^{\omega}_{\leq}$ (Bruce & Mitchell, 1992; Pierce & Turner, 1994) zugrunde gelegt. Das Typsystem $F^{\omega}_{\leq}$ ist eine Erweiterung des einfachen typisierten λ-Kalküls. Es wurde entwickelt, um in einem einfachen funktionalen Rahmen die Semantik von Untertypisierung, Kapselung und Polymorphie zu definieren und Typisierungsregeln zu studieren. Die Polymorphie kommt in $F^{\omega}_{\leq}$ in zwei Ausprägungen, der Parametrisierung von Funktionen mit Typen und der Parametrisierung von Typen mit Typen.

Die Verbindung des Verfeinerungskalküls mit $F^{\omega}_{\leq}$ erfolgt durch die Definition von Prädikaten als boolesche Funktionen und Prädikatentransformern als Funktionen von Prädikaten nach Prädikaten in $F^{\omega}_{\leq}$. Anweisungen werden mit Prädikatentransformern gleichgesetzt, wobei die ausführbaren Anweisungen eine Teilmenge bilden. Objekten werden so definiert, daß Methoden beliebige (ausführbare oder nicht ausführbare) Anweisungen sein können. Durch diese Verbindung geht die Spezifikationsmächtigkeit und die Flexibilität der Typisierung (bei statisch überprüfbarer Typisierung) über existierende objektorientierte Programmier- und Spezifikationssprachen hinaus.

Für die Formalisierung von Entwurfsprinzipien wird als zentrales methodisches Konzept die *Klassenverfeinerung* zugrunde gelegt. Analog zu der Verfeinerungsrelation zwischen Anweisungen besagt die Klassenverfeinerung, daß eine verfeinerte Klasse durch eine verfeinernde Klasse ersetzt werden kann, und daß dabei die Korrektheit des gesamten Programmes gewahrt bleibt.

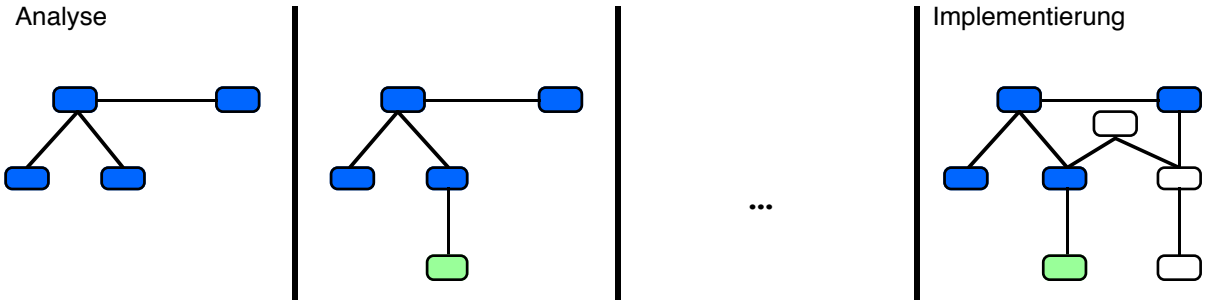
In dieser Arbeit wird eine Notation und keine Sprache vorgestellt, in dem Sinne, daß keine (konkrete oder abstrakte) Syntax definiert wird. Die gewählte Notation abstrahiert von den Eigenarten und Unzulänglichkeiten existierender Programmiersprachen. Die methodischen Erkenntnisse lassen sich aber auf diese anwenden. Die vorgestellte Semantik objektorientierter Konstrukte läßt sich auch für die Entwicklung neuer Sprachen einsetzten.

Insbesondere wird gezeigt, daß es nützlich ist, Objekttypen von Klassen zu unterscheiden. Der Typ eines Objektes bestimmt seine Schnittstelle, während die

Klasse eines Objektes sein Verhalten bestimmt. Durch die Unterscheidung wird die maschinell überprüfbare Typisierung von unentscheidbaren Fragen des Verhaltens von Objekten getrennt.

Die Rolle einer Theorie der objektorientierter Programmkonstruktion

Eine Eigenart der objektorientierten Programmkonstruktion ist, daß zu einmal gefundenen Klassen, immer nur weitere Klassen hinzukommen. Die Klassen der Problemanalyse und aus nachfolgenden Entwurfsschritten werden somit auch zu Klassen der Implementierung. Klassen werden während des Entwurfes i. d. R. nicht durch andere Klassen ersetzt oder gelöscht.



Ein Vorteil der objektorientierten Programmkonstruktion ist die *einheitliche Begriffswelt*. Eine Transformation der Begriffe in unterschiedliche Sprachen für Analyse, …, Implementierung findet nicht statt. Dies macht den Entwurf großer Programme übersichtlicher. Es wird auch die *Wartung* von Programmen vereinfacht: das Resultat der Programmentwicklung ist eine Sammlung von Klassen, die die Entwurfsentscheidungen widerspiegeln. Entwurfsentscheidungen sind somit dokumentiert und nachvollziehbar.

Ein weiterer Vorteil ist das entstehende Potential für die *Wiederverwendung*. Klassen, die bei der Analyse eines Problembereiches entstehen, beschreiben das Verhalten von Objekten der „realen Welt“. Sie sind nicht notwendigerweise gebunden an das Programm, für das sie ursprünglich entworfen worden sind. Sie sind deshalb in anderen Programmen wiederverwendbar.

Diese Vorteile haben objektorientierten Programmiersprachen zu einer weiten Verbreitung verholfen. Gleichzeitig zu der weiten Verbreitung kann eine starke Vielfalt beim Anwenden der sprachlichen Konzepte und der Entwurfsmethodik

beobachtet werden. Die Bibliotheken der objektorientierten Programmiersprachen Smalltalk, Eiffel, C++, usw. sind nach unterschiedlichen Prinzipien aufgebaut. Insbesondere wird Vererbung verschiedenartig eingesetzt. Beispielsweise werden in (LaLonde & Pugh, 1991) drei und in (Budd, 1991) acht typische Arten der Benutzung aufgezählt (und gleichzeitig von einigen abgeraten).

Von einer Theorie der objektorientierten Programmierung wird erwartet, daß sie zu einer Klarstellung der „korrekten“ Benutzung von Konzepten wie der Vererbung führt, analog zu der Rolle von Theorien für andere Programmiersprachen (siehe Tabelle 1).

Programmierparadigma	typische Sprachen	theoretische Grundlagen
strukturiertes Programmieren	PASCAL	schrittweise Verfeinerung
modulares Programmieren	Modula-2, Ada	Datenabstraktion
funktionales Programmieren	ML	λ-Kalkül, axiomat. Spez.
logisches Programmieren	Prolog	Logik, Resolution
paralleles Programmieren	occam, …	CSP, …
objektorientiertes Prog.	Smalltalk-80, …	?

Tabelle 1: Klassifikation sequentieller Programmiersprachen

Aufbau der Arbeit

Kapitel 2 gibt einen Überblick über verwandte Arbeiten und den Ansatz.

Kapitel 3 definiert die Semantik elementarer Programme und gibt die wichtigsten Gleichheits- und Verfeinerungsregeln. Die Definitionen werden in einer Teilmenge von $F^{\omega}_{\leq}$ gegeben.

In Kapitel 4 definiert die Semantik objektorientierten Konzepte: Objekte mit Attributen und Methoden, Kapselung privater Attribute, Attributselektion und Methodenaufrufe, Untertypisierung, Parametrisierung, Klassen und Instanzen, Vererbung. Die Definitionen werden in $F^{\omega}_{\leq}$ gegeben.

Die folgenden Kapitel behandeln, aufbauend auf der Notation aus Kapiteln 3 und 4, methodische Aspekte.

In Kapitel 5 wird in einer verhaltensorientierten Sicht Äquivalenz und Verfeinerung von Klassen definiert. Es wird gezeigt, daß von zwei gegebenen Klassen die Verfeinerungsrelation durch die Beweistechnik mittels Simulation nachgewiesen werden kann.

In Kapitel 6 geht es um den korrekten Entwurf von verfeinernden Klassen per Konstruktion. Dazu werden zwei Techniken unterschiedlicher Allgemeinheit benutzt: die Klassenverfeinerung mittels Relation und die Klassenverfeinerung mittels Abstraktionsfunktion und Invariante.

Kapitel 7 behandelt den Spezialfall des Entwurfs einer verfeinernden Klasse, die durch inkrementelle Modifikation mittels Vererbung entsteht.

Kapitel 8 geht auf das Herausfaktorisieren von Gemeinsamkeiten von Klassen beim Entwurf ein. Es wird gezeigt, wie mittels den zwei Operatoren Schnitt und Vereinigung von zwei Klassen eine gemeinsam verfeinerte bzw. verfeinernde systematisch konstruiert werden kann.

Kapitel 9 faßt zusammen und ordnet den Ansatz ein. Es werden die Praktikabilität der Benutzung und Implementierungsaspekte diskutiert und einen Ausblick auf Erweiterungen gegeben.

Der Anhang erklärt das Typsystem $F^{\omega}_{\leq}$ und listet die Regeln von $F^{\omega}_{\leq}$ auf.

2 Verwandte Arbeiten und Ansatz

Dieses Kapitel gibt einen Überblick über Ansätze zur Fundierung von Objektorientierung: Theorien für die Typisierung und Untertypisierung von Objekten (2.1), Theorien für die Spezifikation von Klassen (2.2) und Ansätze für Verfeinerungskalküle mit Objekten (2.3). Daraus wird der gewählte Ansatz begründet (2.4).

2.1 Typisierung und Untertypisierung

Bei Programmiersprachen (und Spezifikationssprachen) unterscheidet man typisierte von untypisierten Sprachen. Die Typisierung führt zu einer Reihe von Vorteilen:

- Die korrekte Typisierung bestimmt die Ausdrücke, deren Semantik definiert ist. Somit ist sichergestellt, daß ein wohl-typisierter Ausdruck eine Semantik hat.
- Die Typisierung führt eine Redundanz ein, die eine mechanische Überprüfung der konsistenten Benutzung zuläßt. Durch die Typisierung können Werte unterschieden werden, die im semantischen Modell gleich sind (wie die Zahl 0 und die leere Menge) oder im Rechner gleich dargestellt werden (wie *Int* und Zeiger).
- Für typisierte Programmiersprachen kann effizienterer Kode erzeugt werden weil der Speicherbedarf einer Variablen oft statisch bestimmbar ist und weil bestimmte Laufzeitüberprüfungen wegfallen.

Diese Vorteile beruhen auf dem Prinzip, daß durch ein *Typinferenzsystem* jedem Ausdruck ein Typ zugewiesen werden kann. Die Typ kann entweder statisch bestimmbar (*statische Typisierung*) oder dynamisch sein, aber trotzdem obige Vorteile ermöglichen (*strenge Typisierung*).

Im folgenden wird auf Ansätze für typisierte objektorientierte Sprachen eingegangen. Untertypisierung wird im Sinne der strengen Typisierung als eine Flexibilisierung des Typsystems unter Beibehaltung obiger Vorteile gesehen. Die Flexibilisierung beruht darauf, daß ein Wert nicht nur einen sondern mehrere Typen haben kann (wobei es einen kleinsten gibt). In (Cardelli, 1984; Cardelli & Wegner, 1985) werden hierzu Objekte durch *Verbunde* (*records*) beschrieben. Beispielsweise ist

$$type\ Vehicle = record\ age : Int, speed : Int\ end$$

der Typ der Verbunde mit Feldern *age* und *speed*. Durch den binären Operator $\oplus$ können Felder redefiniert und angehängt werden, ähnlich wie bei Vererbung:

$$type\ Car = Vehicle \oplus record\ fuel : String\ end$$

Eine dazu äquivalente Definition ist:

$$type\ Car = record\ age : Int, speed : Int, fuel : String\ end$$

Die Reihenfolge der Felder in einem Verbund ist irrelevant. Auf Verbundtypen ist eine *Untertyprelation* definiert: ein Untertyp muß mindestens die Felder des Obertyps haben und die Felder mit gleichem Namen müssen ebenfalls in einer Untertyprelation stehen. Beispielsweise ist Car ein Untertyp von $Vehicle$, geschrieben:

$$Car \leq Vehicle$$

Die Untertyprelation ist unabhängig davon, welche syntaktischen Mittel zur Konstruktion der Typen verwendet werden. Im obigen Beispiel gilt die Untertyprelation mit beiden Definitionen von *Car*.

Ein Wert eines Verbundtyps weist den Feldnamen entsprechende Werte zu. Beispielsweise ist

$$val\ mycar = (age \mapsto 4, speed \mapsto 120, fuel \mapsto \text{"}gasoline\text{"})$$

ein Verbund vom Typ *Car*. Der Verbund *mycar* ist auch vom Typ *Vehicle* und von weiteren Typen, *Car* ist aber sein kleinster Typ.

Ein Feld eines Verbundes wird mit dem Punkt-Operator selektiert:

$$mycar.age = 4$$

Eine Funktion, die bestimmt, ob ein Fahrzeug alt ist, wird definiert durch:

$$val\ isOld = (\lambda\, v : Vehicle \bullet v.age \geq 12)$$

Die Funktion *isOld* kann auf alle Verbunde vom Typ *Vehicle* angewandt werden. Weil *mycar* auch vom Typ *Vehicle* ist, läßt sich *isOld* somit auch auf *mycar* anwenden:

$$isOld\ mycar = false$$

Beginnend mit (Cardelli, 1984) wird Untertypisierung in einem einfachen funktionalem Rahmen durch Erweiterung des einfachen typisierten λ-Kalküls studiert. Das Modell von Typen sind dabei Mengen bestimmter Bauart. Die Zugehörigkeit eines Wertes x zu einem Typ T, geschrieben $x : T$ entspricht $x \in T$ und Untertypisierung $S \leq T$ entspricht $S \subseteq T$. Um die Fragen der Typisierung und der Untertypisierung entscheidbar zu machen (mit anfangs genannter Motivation), können neue Typen nur aus gegebenen Konstruktoren für Verbunde, Varianten, Mengen, etc. und elementaren Typen wie *Int*, *Bool*, *String*, etc. definiert. Es können keine neu axiomatisierte Typen, wie aus algebraischen Spezifikationssprachen bekannt, hinzugefügt werden. Auf konstruierten Typen (wie *Car*) gibt es vordefinierte Operationen. Für Verbunde sind dies die Konstruktion (wie bei *mycar*) und die Selektion (z. B. *mycar.age*).

In (Cardelli & Wegner, 1985) wird die Nützlichkeit der Erweiterung des typisierten λ-Kalküls mit Untertypisierung um parametrische Polymorphie gezeigt. Darunter sind Funktionen zu verstehen, die auch Typen als Parameter haben, wie aus dem λ-Kalkül zweiter Ordnung bekannt. Eine naive Definition einer Funktion, die das Alter eines Fahrzeuges erhöht, ist:

$$val\ increaseAge = (\lambda\, v : Vehicle \bullet v \oplus (age \mapsto v.age{+}1))$$

Der Operator ⊕ überschreibt die Felder des ersten Operanden mit denen des zweiten. Das Resultat dieser Funktion ist immer vom Typ *Vehicle*, auch wenn sie auf einen Untertyp angewandt wird:

$$increaseAge\ mycar = (age \mapsto 5, speed \mapsto 120)$$

Alternativ hierzu läßt sich eine Funktion definieren, die mit einem Typ parametrisiert ist, der Untertyp von *Vehicle* sein muß:

$$val\ IncreaseAge = (\lambda\ T <: Vehicle \bullet (\lambda\ v : T \bullet v \oplus (age \mapsto v.age{+}1)))$$

Die Funktion *IncreaseAge* kann angewandt werden auf *Car*, weil *Car* Untertyp von *Vehicle* ist. Das Resultat ist:

$$IncreaseAge\ Car = (\lambda\ v : Car \bullet v \oplus (age \mapsto v.age{+}1))$$

Diese Funktion kann wiederum auf *mycar* angewandt werden, mit dem Resultat:

$$IncreaseAge\ Car\ mycar = (age \mapsto 5, speed \mapsto 120, fuel \mapsto \text{"}gasoline\text{"})$$

Ein weiteres Beispiel für parametrische Polymorphie ist eine Sortierfunktion, die mit dem Typ der zu sortierenden Elemente parametrisiert ist,

$$type\ Ordered = record\ key : Int\ end$$
$$val\ Sort = (\lambda\ T <: Ordered \bullet (\lambda\ s : seq\ T \bullet \ \ldots))\ .$$

Die bisher betrachteten Verbunde (Objekte) bestehen nur aus Feldern, die die Rolle von Attributen haben. Falls Felder Methoden sein sollen, ergibt sich eine gewisse Zirkularität dadurch, daß eine selektierte Methode auf einem Verbund operiert, von dem sie selbst Teil ist. Diese Abhängigkeit wird üblicherweise durch Rekursion ausgedrückt (Amadio & Cardelli, 1993; Canning, et al., 1989; Cardelli & Wegner, 1985). Weil eine funktionale Interpretation zugrunde liegt, kann eine Methode nicht die Attribute „in situ“ ändern, sondern erzeugt einen neuen Verbund.

Es sei $T[S]$ ein Typausdruck, in dem die Typvariable S vorkommen kann. Es sei $Rec\ S \bullet T[S]$ der kleinste Typ, der die Gleichung $S = T[S]$ erfüllt. Der Typ eines Kellers von Zahlen mit den Methoden *push*, *pop* und *top* ist dann:

$$type\ IntStack =$$
$$\quad Rec\ S \bullet record\ push : Int \to S, pop : S, top : Int\ end$$

Ist beispielsweise $s : IntStack$, dann erzeugt der Aufruf (die Feldselektion) $s.push$ (3) ein neues Objekt vom Typ *IntStack*. Der Typ *IntStack* legt die *Schnittstelle* oder *Signatur* der Objekte fest.

Objekte vom Typ *IntStack* verhalten sich unterschiedlich, je nach dem ob der Keller leer ist oder nicht. Um das Verhalten von Kellerobjekten vollständig zu beschreiben ist es notwendig, zwei Objekte zu definieren. Das Objekt *empty* entspricht einem leeren Keller, das Objekt $cons(x, l)$ einem Keller, der aus dem

Keller l durch hinzufügen von x entstanden ist. Typischerweise initialisiert man ein Objekt vom Typ *IntStack* mit *empty*.

Es sei $e\ (s)$ ein (Wert-) Ausdruck, in dem die Variable s vorkommen kann. Es sei $rec\ s \bullet e\ (s)$ der „kleinste“ Wert, der die Gleichung $s = e\ (s)$ erfüllt. Ein Objekt vom Typ *IntStack* ist von der Form $rec\ self \bullet e\ (self)$. Die Idee dabei ist, durch *self* innerhalb des Objektes auf das ganze Objekt zuzugreifen:

$$
\begin{array}{l}
val\ empty = \\
\quad rec\ self \bullet \\
\qquad (push \mapsto (\lambda\, x : Int \bullet cons\ (x, self)), \\
\qquad push \mapsto error, \\
\qquad top \mapsto error)
\end{array}
$$

$$
\begin{array}{l}
val\ cons\ (x : Int, l : IntStack) = \\
\quad rec\ self \bullet \\
\qquad (push \mapsto (\lambda\, x : Int \bullet cons\ (x, self)), \\
\qquad pop \mapsto l, \\
\qquad top \mapsto x)
\end{array}
$$

Die oben diskutierte Untertyprelation wird analog auf rekursive Typen erweitert. Beispielsweise ist

$$
\begin{array}{l}
type\ IntStackEmpty = \\
\quad Rec\ S \bullet record\ push : Int \to S, pop : S, top : Int, empty : Bool\ end
\end{array}
$$

ein Untertyp von *IntStack*, d.h. $IntStackEmpty \leq IntStack$.

In (Reynolds, 1978) wird zwischen *prozeduraler Datenabstraktion* und *Typabstraktion* als zwei komplementären Arten der *Datenabstraktion* unterschieden. Bei der prozeduralen Abstraktion ist die Repräsentation eines Typs nach außen nicht sichtbar. Auf sie kann nur durch eine prozedurale Schnittstelle zugegriffen werden kann. Die prozedurale Datenabstraktion ist beispielsweise bei der algebraischen Spezifikation von Datentypen gegeben. Bei der Typabstraktion ist die Existenz einer Repräsentation nach außen sichtbar, jedoch wird der direkte Zugriff auf sie verboten. Beispiele für die Typabstraktion sind modellorientierte Spezifikation von Datentypen in Z oder VDM.

In (Cook, 1990) wird argumentiert, daß die Kodierung von Objekten durch rekursive Typen der prozeduralen Datenabstraktion entspricht. Die Typabstraktion wird durch *existentiell quantifizierte Typen* erreicht (Mitchell & Plotkin, 1985). Ein existentiell quantifizierter Typ $(\sum S \bullet T[S])$ steht intuitiv für eine Menge von Typen $T[S]$, die durch die Typvariable S indiziert sind. In (Pierce & Turner, 1994) wird ein Modell von Objekten durch existentiell quantifizierte Typen vorgeschlagen. Die Idee dabei ist, daß Objekte aus Attributen und Methoden bestehen, wobei der Typ der Attribute im Typ des Objektes nicht sichtbar sein soll. Dies wird erreicht, indem der Typ der Attribute existentiell gebunden wird. Die Kapselung der Attribute ist also gewährleistet, indem zwar die Existenz der Attribute (im Typ der Objekte) sichtbar ist, der Typ der Attribute aber unbekannt ist. Der Vorteil dieses Modells von Objekten ist, daß auf Rekursion im obigen Sinne verzichtet werden kann, was zu einem einfacheren Modell führt.

Ein Element vom Typ $(\sum S \bullet T[S])$ kann intuitiv verstanden werden als ein Paar, dessen erste Komponente ein Typ ist, hier mit S bezeichnet. Die zweite Komponente ist ein Wert (der *Inhalt*), dessen Typ von S abhängt. Der Typ $(\sum S \bullet T[S])$ ist dann die Menge aller solcher Paare.

Beispielsweise wird der Typ *IntStack* definiert durch einen existentiell quantifizierten Typ, dessen *Inhalt* aus den Attributen und den Methoden *push*, *pop* und *top* besteht, wobei die Methoden auf den Attributen operieren. Der Typ der Attribute wird durch den Quantor versteckt:

$$\begin{aligned} &\mathit{type}\ \mathit{IntStack} = \\ &\quad (\sum A \bullet A \times \mathit{record}\ \mathit{push} : A \times \mathit{Int} \to A, \mathit{pop} : A \to A, \mathit{top} : A \to \mathit{Int}\ \mathit{end}) \end{aligned}$$

Der Typ *IntStack* legt nur die Schnittstelle bzw. Signatur fest. Ein Element dieses Typs spezifiziert den gekapselten Typ A als auch den Inhalt bestehend aus dem Zustand (vom Typ A) und den Methoden. Es bietet sich an, den Zustand eines Kellers zu repräsentieren durch den Typ (in ML-ähnlicher Schreibweise)

$$\mathit{IntStackRep} = \mathit{empty} \mid \mathit{cons}\ (\mathit{Int}, \mathit{IntStackRep})\ .$$

Elemente von existentiell quantifizierten Typen können nur durch *Packen* konstruiert werden. Dabei wird der Inhalt, die Instantierung der Typvariablen und der resultierende Typ spezifiziert. Ein leerer Keller ist beispielsweise definiert durch:

```
pack
    (empty,
        (push ↦ (λ l: seq Int, x : Int • cons (x, l)),
        pop ↦ (λ l: seq Int • case l of
            empty -> error
            cons (x, l') -> l'),
        top  ↦ (λ l: seq Int • case l of
            empty -> error
            cons (x, l') -> x)))
by IntStackRep as IntStack
```

Dieses Objekt kann zur Initialisierung von Objekten vom Typ *IntStack* benutzt werden. Ein Nachteil der Kodierung von Objekten durch existentielle Typen ist, daß auf die Felder nicht direkt zugegriffen werden kann. Dazu muß ein Objekt zuerst *geöffnet* werden. Beim Öffnen wird dem gebundenen Typ ein Name gegeben. Falls *s* vom Typ *IntStack*, wird die Methode *top* aufgerufen durch

```
open s as A by
    state : A, oper : record push : A × Int → A, pop : A → A, top : A → Int end
in
    oper.top (state)
end  .
```

Die Untertyprelation wird auch auf existentiell quantifizierte Typen erweitert: Zwei existentiell quantifizierte Typen stehen in einer Untertypbeziehung, falls ihre Inhalte (Rümpfe) in einer Untertypbeziehung stehen. Beispielsweise ist

```
type IntStackEmpty =
    (∑ A • A × record push : A × Int → A, pop : A → A,
        top : A → Int, empty : A → Int end)
```

ein Untertyp von *IntStack*, d.h. $IntStackEmpty \leq IntStack$.

Interessant an beiden Ansätzen (Kodierung von Objekten durch rekursive Typen oder existentielle Typen) ist die Erweiterung auf die Parametrisierung von Typen mit Typen, analog zum λ-Kalkül höherer Stufe. Ein Beispiel hierfür ist die Parametrisierung des obigen Kellers mit dem Typ der Kellerelemente:

```
type Stack [T]=
    (∑ A • A × record push : A × T → A, pop : A → A, top : A → T end)
```

Die genannten Ansätze zur Typisierung und Untertypisierung in objektorientierten Sprachen sind entstanden als Erweiterung des typisierten λ-Kalküls. Die Problematik der korrekten Typisierung in imperativen Sprachen läßt sich auch in diesen funktionalen Ansätzen modellieren. Beispielsweise wurden in (Cook, 1989) Fehler im Typsystem der Programmiersprache Eiffel gefunden und Vorschläge zur Korrektur gemacht.

In (Bruce & Gent, 1993; Bruce, et al., 1994; Eifrig, et al., 1993) werden objektorientierte Typsysteme auf imperative Sprachen angewandt. Dem imperativen Teil wird eine denotationale oder operationale Semantik gegeben. Damit werden alle wesentlichen Elemente objektorientierter Sprachen formal definiert; die Flexibilität der Typisierung und Parametrisierung geht über übliche Programmiersprachen sogar hinaus. Methodische Aspekte wie die Spezifikation oder Verifikation von objektorientierten Programmen werden in keinem dieser (funktionalen und imperativen) Ansätze betrachtet.

2.2 Spezifikation von Klassen mit Zusicherungen

Ein wesentlicher Beitrag zum methodischen Verständnis von Objektorientierung liefert die Spezifikation von Anweisungen durch Zusicherungen (America, 1987; Leavens, 1991; Meyer, 1988; Wills, 1991). Eine Zusicherung der Form

$$\{pre\}\ stat\ \{post\}$$

bedeutet, daß, falls vor Ausführung der Anweisung *stat* die Bedingung *pre* gilt, die Ausführung von *stat* terminiert und danach die Bedingung *post* gilt. Die Vor- und Nachbedingung bezieht sich auf die Programmvariablen in *stat*. Mit den Regeln des Hoare-Kalküls (Hoare, 1969) läßt sich diese Beziehung nachweisen. Durch Zusicherung wird die Sichtweise der Implementierung von der Sichtweise der Benutzung getrennt. Für die Benutzung reicht die Kenntnis des Verhaltens wie durch die Vor- und Nachbedingung gegeben.

Zusicherungen lassen sich auch zur *modellorientierten* Spezifikation von Datentypen einsetzen. In (Hoare, 1972) wird dies mit Datentypen gezeigt, die als Simula-Klassen geschrieben werden. Für jede Operation des Datentyps wird eine Vor- und Nachbedingung angegeben. Diese Zusicherungen werden mit Hilfe *abstrakter Variablen* ausgedrückt, d.h. Variablen, die nicht Teil der Implementie-

rung sind, sondern nur dazu dienen, das Verhalten des Datentypes zu definieren. Damit wird ebenfalls eine Trennung der Sichtweise der Implementierung von der Sichtweise der Benutzung erreicht. Für die Benutzung ist nur das Verhalten der Operationen wie durch die Vor- und Nachbedingungen auf den abstrakten Variablen gegeben interessant.

Unter der Analogie, daß eine Klasse einem Datentyp entspricht, läßt sich dieses Prinzip direkt auf Klassen in üblichen Programmiersprachen übertragen. Folgendes Beispiel aus (America, 1990) gibt die Spezifikation eines Kellers und die Implementierung durch ein dynamisches Feld. Dabei ist s die abstrakte Variable der Spezifikation und s_0 eine Hilfsvariable. In der Spezifikation steht [] für die leere Sequenz, $[x, \ldots, y]$ für die Sequenz aus $x, \ldots, y$ und $s \circ t$ für die Konkatenation von Sequenzen s und t. Die Indizierung wird durch $s[i]$ geschrieben:

$s : seq\ of\ Int := []$
$\{s = s_0\}\ put\ (n)\ \{s = s_0 \circ [n]\}$
$\{s = s_0 \wedge s \neq []\}\ get\ ()\ \{s_0 = s \circ [res]\}$

```
class ArrayStack
    var t : Int := 0
        a : Array(Int)
    method put (n : Int)
        begin t := t + 1 ; a[t] := n end
    method get : Int
        begin
            if t = 0 then res := NIL
            else res := a[t] ; t := t - 1
            end
        end
end
```

In der Terminologie von (America, 1990) ist die Spezifikation der Klasse auf der linken Seite ein *Typ*. Die Klasse auf der rechten Seite *implementiert* den Typ[1]. Mit den Regeln aus (Hoare, 1972) läßt sich diese Implementierungsbeziehung nachweisen. Dazu muß eine *Abstraktionsfunktion* aufgestellt werden, die die Variablen der Implementierung auf die abstrakten Variablen abbildet. Im obigen Fall ist eine adäquate Abstraktionsfunktion:

$$s = abs\ (a, t) \quad \text{mit} \quad abs\ (a, t) = [a[0], \ldots, a[t\text{-}1]]$$

[1]Dies unterscheidet sich von dem Typbegriff der Typtheorien in 2.1. In der Terminologie der Typtheorie (und dem Hauptteil dieser Arbeit) beschreibt ein Typ eines Objektes lediglich seine Schnittstelle bzw. Signatur, nicht aber sein Verhalten.

Laut (Hoare, 1972) ergeben sich eine Beweisbedingung für die Initialisierung von s bzw. t und a und je eine Beweisbedingung für die Methoden *put* und *get*:

$$[] = abs\ (a,0)$$
$$\{abs\ (a, t) = s_0\}\ put\ (n)\ \{abs\ (a, t) = s_0 \circ [n]\}$$
$$\{abs\ (a, t) = s_0 \wedge abs\ (a, n) \neq []\}\ get\ ()\ \{s_0 = abs\ (a, t) \circ [res]\}$$

Die beiden letzteren lassen sich wiederum mit den Regeln des Hoare-Kalküls nachweisen.

Die Klassenspezifikationen können eingesetzt werden, um die Ersetzbarkeit (substitutability) einer Klasse bezüglich des Verhaltens sicherzustellen. Der Terminologie von (America, 1990) weiter folgend, ist die *Untertyprelation* eine Beziehung zwischen den Typen (Spezifikationen) von zwei Klassen. Untertypisierung besagt, daß sich Instanzen eines Untertyps auch entsprechend des Obertyps verhalten. Beispielsweise ist der Typ eines ungeordneten Puffers (*Buffer*) gegeben durch folgende Spezifikation mit einer abstrakten Variablen b. Es sei $\langle\rangle$ die leere Multimenge, $\langle x, \ldots, y\rangle$ die Multimenge aus $x, \ldots, y$ und $b + c$ die Vereinigung von Multimengen b und c.

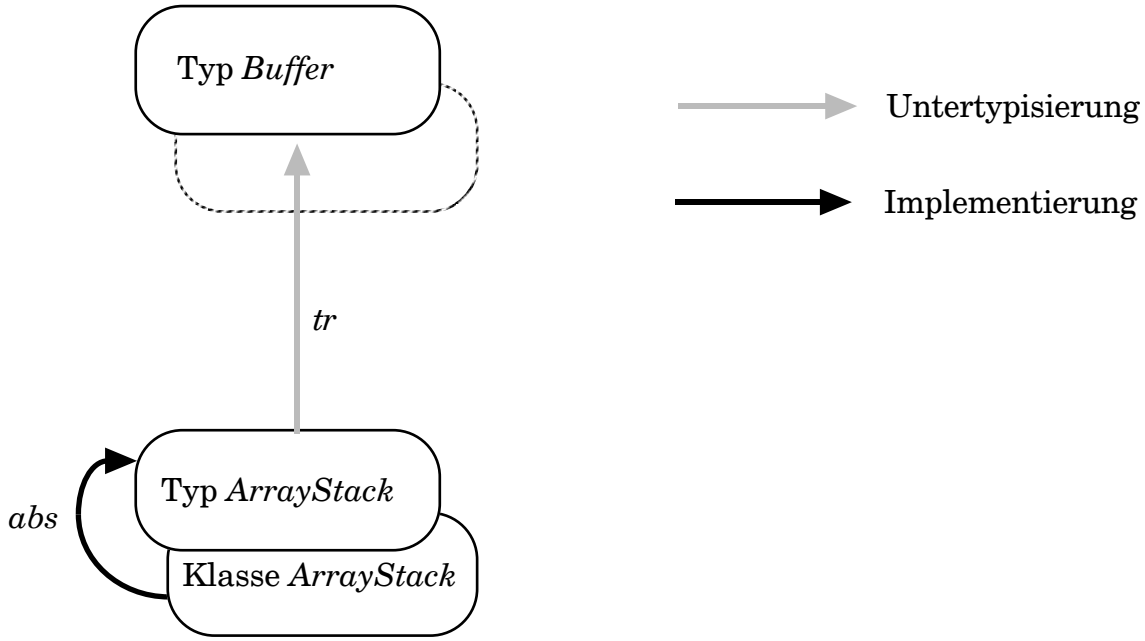
$$b : bag\ of\ Int := \langle\rangle$$
$$\{b = b_0\}\ put\ (n)\ \{b = b_0 + \langle n\rangle\}$$
$$\{b = b_0 \wedge b \neq \langle\rangle\}\ get\ ()\ \{b_0 = b + \langle res\rangle\}$$

Der Typ von *ArrayStack* ist ein Untertyp von *Buffer* in dem Sinne, daß sich Instanzen von *ArrayStack* auch entsprechend des Typs *Buffer* verhalten. Untertypisierung zwischen Typen wird mittels einer *Transferfunktion*, die die (abstrakten) Variablen des Untertyps auf die des Obertyps abbildet, nachgewiesen. Es sei *#s* die Länge der Sequenz s. Eine Transferfunktion von *ArrayStack* nach *Buffer* ist

$$b = tr\ (s) \quad \text{mit} \quad tr\ (s) = \langle s[0], \ldots, s[\#s - 1]\rangle\ .$$

Die Untertyprelation wird dann durch einfache Implikationsbeweise der entsprechenden Vor- und Nachbedingungen nachgewiesen.

Für die entstehenden Beziehungen ergibt sich folgendes Bild. Die Klasse *Buffer* ist spezifiziert, aber nicht implementiert, die Klasse *ArrayStack* ist sowohl spezifiziert als auch implementiert.



Der praktische Nutzen der Spezifikation von Klassen durch Zusicherungen wurde in (Cook, 1992) mit einer kompakten Spezifikation der *Smalltalk Collection-Klassen* demonstriert (Goldberg & Robson, 1983); es wurden Inkonsistenzen aufgezeigt und daraus Verbesserungsvorschläge abgeleitet.

Der prinzipielle Nachteil dieses Ansatzes ist, daß die Spezifikation einer Klasse *nicht* Teil der Klasse ist, sondern zur Metasprache gehört. Das bedeutet, daß sich die objektorientierte Strukturierung durch Klassen und Vererbung nicht auf Spezifikationen anwenden läßt. Diese Zweistufigkeit ist nicht im Sinne des objektorientierten Entwurfs. Vielmehr ist es natürlicher, die Spezifikation als eine abstrakte Klasse anzusehen und einen fließenden Übergang zwischen abstrakten und konkreten Klassen zu haben. Dies ist auch die Art wie Zusicherungen in Eiffel (Meyer, 1988) in der Praxis benutzt werden. In Eiffel können Vor- und Nachbedingungen statt Methodenrümpfe geschrieben werden. Die Vor- und Nachbedingung wird als ein abstrakter Methodenrumpf verstanden. Trotzdem ist damit die Zweistufigkeit nicht aufgehoben: in Eiffel kann eine Klasse *nicht* zur Spezifikation einer anderen Klasse benutzt werden, beispielsweise als Attribut- oder als Parametertyp. Auf ein Attribut, das von einer Klasse ist, kann nur, durch Methodenaufrufe zugegriffen werden. Methodenaufrufe können aber nicht in Zusicherungen benutzt werden, sie beschreiben eine Zustandstransformation und nicht einen booleschen Wert. Das bedeutet, daß es nicht möglich ist, Klassen (in der Rolle von Attribut- oder Parametertypen) für die *Spezifikation* von anderen Klassen durch Vor- und Nachbedingungen zu verwenden. Es ist nur

möglich, Klassen für die *Implementierung* anderer Klassen zu benutzen. Die Skalierbarkeit des Ansatzes ist nicht gegeben.

2.3 Verfeinerungskalküle mit Objekten

Die oben diskutierte Zweistufigkeit wäre behoben, wenn Spezifikations- und Implementierungssprache angeglichen wären. Dies ist genau die Grundidee des Verfeinerungskalküls (Back, 1978; Back, 1980), bei dem Spezifikationen als abstrakte Anweisungen gesehen werden. In diesem Abschnitt wird zunächst das sequentielle Verfeinerungskalkül vorgestellt und dann objektorientierte Erweiterungen.

Der Verfeinerungskalkül ist als eine Weiterentwicklung des Hoare-Kalküls entstanden, um das Prinzip der schrittweisen Verfeinerung direkter auszudrücken. Der Verfeinerungskalkül vereinigt die Programmentwicklung durch Zusicherungen (Hoare, 1969) mit dem transformationellen Ansatz (Bauer, et al., 1985). Statt von einer Anweisung *stat* nachzuweisen, daß sie zu der Nachbedingung *post* unter Vorbedingung *pre* führt,

$$\{pre\}\ stat\ \{post\}$$

wird das Paar $[pre, post]$ (in der Notation von (Morgan, 1988c)) als eine *Spezifikationsanweisung* (*specification statement*) verstanden, die von *stat* verfeinert wird:

$$[pre, post] \sqsubseteq stat$$

Die Rolle der Bedingungen (Prädikate über Programmvariablen) *pre* und *post* hat sich dabei geändert: Im Hoare-Kalkül gehören sie zur Metasprache, im Verfeinerungskalkül sind sie Teil der (erweiterten) Programmiersprache.

Als Semantik von Anweisungen werden in den Verfeinerungskalkülen (Back & Wright, 1989; Morgan, 1990; Morris, 1987) *Prädikatentransformer* genommen. In E. W. Dijkstras ursprünglicher Notation (Dijkstra, 1976) ist

$$wp\ (stat, post)$$

die *schwächste Vorbedingung* (*weakest precondition*), so daß *stat* terminiert und danach *post* gilt. Der wp-Kalkül legt eine Programmentwicklung von „hinten nach vorne“ nahe: Man startet mit der Nachbedingung und der letzten Anwei-

sung und berechnet dann die schwächste Vorbedingung für diese Anweisung unter der Nachbedingung. Diese ist dann die Nachbedingung für die vorhergehende Anweisungen, mit der dann gleichermaßen fortgesetzt wird.

Das Ein/Ausgabeverhalten einer Anweisung ist eindeutig durch den Zusammenhang von Vor- und Nachbedingung charakterisiert. Deshalb kann die wp-Notation dahingehend vereinfacht werden, daß Anweisungen identifiziert werden mit Funktionen, die Nachbedingungen auf Vorbedingungen abbilden. Dies setzt voraus, daß nur das *funktionale* Verhalten einer Anweisung interessiert. Nichtfunktionale Aspekte wie Zeit- und Speicheraufwand werden damit nicht berücksichtigt. Anweisungen sind damit Funktionen von Prädikaten nach Prädikaten. Die Menge *Tran* der Prädikatentransformer wird definiert als die Menge von Funktionen der Prädikatenmenge *Pred* in sich:

$$Tran = Pred \to Pred$$

Die Anwendung eines Prädikatentransformers (einer Anweisung) p auf ein Prädikat b, geschrieben $p\ b$, ist äquivalent zu Dijkstras $wp\ (p, b)$. Mit Prädikatentransformern lassen sich alle üblichen sequentiellen Programmkonstrukte wie sequentielle Komposition, Verzweigung, Iteration, Rekursion, Prozeduren als auch Spezifikationsanweisungen definieren. Beispielsweise sind für Prädikatentransformer p, q und Prädikat b:

$skip\ b$	$= b$	*„leere" Anweisung*
$(p\ ;\ q)\ b$	$= p\ (q\ b)$	*sequentielle Komposition*
$[pre, post]\ b$	$= pre \wedge (post \Rightarrow b)$	*Spezifikationsanweisung*

In (Dijkstra, 1976) erfüllen die implementierbaren Prädikatentransformer (Anweisungen) sog. *„healthiness conditions"*. Es seien $a\ i$ für $i \geq 0$ und c Prädikate:

$p\ false$	$= false$	*Gesetz des ausgeschlossenen Wunders*
$(p\ b) \wedge (p\ c)$	$= p\ (b \wedge c)$	*Konjunktivität*
$p\ (\exists\ i \geq 0 \bullet a\ i)$	$= (\exists\ i \geq 0 \bullet p\ (a\ i))$	*Stetigkeit*
falls $a\ i \Rightarrow a\ (i + 1)$ für alle $i \geq 0$		

Das Gesetz des ausgeschlossenen Wunders (*law of the excluded miracle*) schließt Prädikatentransformer aus, die jede Nachbedingung automatisch erfüllen. Die Konjunktivität schließt Prädikatentransformer aus, die gutartig indeterministisch sind. Die Stetigkeit schließt Anweisungen aus, die unbeschränkt indeterministisch sind (Dijkstra, 1982), beispielsweise die Spezifikationen

$[true, x \geq 0]$.

Um Spezifikationen dieser Art als Anweisungen zuzulassen, werden alle diese Einschränkungen im Verfeinerungskalkül fallen gelassen, d. h. es werden wundersame, gutartig indeterministische und unbeschränkt indeterministische Anweisungen zugelassen. Somit werden ausführbare Anweisungen eingebettet in einen größeren Raum *abstrakter Anweisungen*. Als einzige Einschränkung wird die Monotonie beibehalten. Sie besagt, daß die Verstärkung einer Nachbedingung auch die Verstärkung der schwächsten Vorbedingung zu Folge hat, d. h. für Prädikatentransformer p und Prädikate b, c gilt:

aus $b \Rightarrow c$ folgt $p\ b \Rightarrow p\ c$ *Monotonie*

Anweisung p wird verfeinert durch Anweisung p bedeutet, daß q nur zu Nachbedingungen führt, zu denen auch p führt:

$p \sqsubseteq q$ gdw für alle Prädikate b gilt: $p\ b \Rightarrow q\ b$

Programmentwicklung von einer abstrakten Anweisung p zu einer Implementierung q kann schrittweise erfolgen:

$p = p_1 \sqsubseteq p_2 \sqsubseteq \ldots \sqsubseteq p_n = q$

Dies ist durch die Transitivität von $\sqsubseteq$ sichergestellt. Die Zwischenschritte p_2, …, $p_{n\text{-}1}$ können abstrakte Anweisungen sein, konkrete Anweisungen, oder Kombinationen von abstrakten und konkreten Anweisungen. Es können auch Teile von Anweisungen verfeinert werden (*partwise refinement*). Falls $p\ [q]$ eine Anweisung ist, in der q vorkommt, gilt:

aus $q \sqsubseteq r$ folgt $p\ [q] \sqsubseteq p[r]$

(Dies ist durch die Einschränkung auf monotone Prädikatentransformer sichergestellt.)

Für die Semantik von (abstrakten) Anweisungen existieren eine Reihe alternativer Modelle. Beispielsweise können Anweisungen durch Relationen (z. B. in (Hoare, et al., 1987)) oder durch mengenwertige Funktionen (z.B. in (Apt & Plotkin, 1986)). In diesen Modellen gibt es nur eine Art von Indeterminismus, der dann als bösartiger Indeterminismus ausgelegt wird. Die Semantik durch Prädikatentransformer läßt sowohl bösartigen als auch gutartigen Indeterminismus zu. Prädikatentransformer bilden somit das umfassendste Modell. Prädikatentransformer sind auch das einzige Modell, in dem Anweisungen einen vollständi-

gen Verband bilden mit der bösartigen Auswahl als Schnitt und der gutartigen Auswahl als Vereinigung.

In (Utting, 1992) wird eine objektorientierte Erweiterung des Verfeinerungskalküls untersucht, mit dem Ziel abstrakte Anweisungen als Methodenrümpfe zuzulassen. Hierzu wird folgendes Gleichungssystem aufgestellt:

$$\begin{aligned} Val &= Int + Bool + \ldots + Obj \\ State &= Var \to Val \\ Pred &= State \to Bool \\ Stat &= Pred \xrightarrow[m]{} Pred \\ Obj &= State \times Stat \end{aligned}$$

Werte (*Val*) sind entweder Werte von Basistypen oder Objekte. Ein Zustand ist eine Abbildung der Variablen auf Werte. Prädikate werden semantisch definiert als Funktionen von Zuständen nach booleschen Werten. Anweisungen sind monotone Funktionen von Prädikaten nach Prädikaten, symbolisiert durch $\xrightarrow[m]{}$. Objekte bestehen aus einem Zustand und (vereinfacht) einer Methode. Es wird gezeigt, daß dieses Gleichungssystem i. A. keine Lösung besitzt. Lösungen existieren für den Spezialfall, daß Anweisungen *stetig* sind. Dann lassen sich Modelle der denotationalen Semantik (die ja gerade auf stetigen Funktionen basiert) konstruieren. Das bedeutet, daß Anweisungen nur beschränkt indeterministisch sind, also keine Spezifikationen der Art [*pre*, *post*] sind. Aus diesem Grund werden in (Utting, 1992) hauptsächlich „multiple dispatch“ Methoden betrachtet. Das sind Methoden, die nicht Teil eines Objektes sind, sondern eher Prozeduren gleichen. Je nach Typ des übergebenen Parameters, wird ein unterschiedlicher Methodenrumpf ausgeführt. Der Methodenrumpf kann dabei von dem Typ eines (des ersten) oder mehrerer Parameter abhängen, daher die Bezeichnung „multiple dispatch“. Der Nachteil ist, daß die Attribute der Objekte für alle Methoden zugänglich sein müssen und deshalb nicht gekapselt sind. Die Sicht eines Objektes als abstrakte Datenstruktur geht verloren.

In (Naumann, 1994) wird die Problematik dadurch umgangen, daß Prädikate und Prädikatentransformer als *typisierte* Funktionen definiert werden. Der Typ eines Prädikates bzw. Prädikatentransformers bestimmt die Variablen, auf die sich das Prädikat bzw. der Prädikatentransformer bezieht. Es wird also *kein* Zustandsraum mit allen Variablen angenommen. Damit wird die Zirkularität im obigen Gleichungssystem so weit reduziert, daß Objekten mit beliebigen Prädika-

tentransformern als Methoden eine Semantik gegeben werden kann. Die Objekte sind dabei Verbunde im Stil von Oberon (Reiser & Wirth, 1992). Eine Kapselung von privaten Attributen, wie in Abschnitt 2.1 diskutiert, ist in dieser Semantik nicht möglich. Untertypisierung ist auf *Projektion* bei Zuweisungen beschränkt. Eine Subsumptionsregel, die besagt, daß ein Wert eines Untertyps auch ein Wert eines Obertyps ist, gibt es nicht.

2.4 Ansatz

Der Ansatz dieser Arbeit ist ein objektorientierter Verfeinerungskalkül auf typtheoretischer Basis. Wie im folgenden gezeigt wird, kann durch die typtheoretische Basis eine reichhaltige Entwurfsnotation definiert werden, mit der objektorientierte Entwurfsprinzipien formalisiert werden können.

Als Basis wird das Typsystem $F^{\omega}_{\leq}$ genommen (Bruce & Mitchell, 1992). Das Typsystem $F^{\omega}_{\leq}$ ist eine Erweiterung des einfachen typisierten λ-Kalküls um (1) parametrische Polymorphie (Funktionen auf Werten unterschiedlicher Typen), (2) Untertyp-Polymorphie (ein Wert eines Untertyps kann eingesetzt werden wo immer ein Wert eines Obertyps erwartet wird) und (3) Typoperatoren (Funktionen von Typen nach Typen). Die parametrische Polymorphie wird durch universelle Quantifizierung ausgedrückt. Zusätzlich ist es durch existentielle Quantifizierung möglich, Kapselung auszudrücken. Die Syntax und die Regeln von $F^{\omega}_{\leq}$ sind im Anhang aufgeführt.

Anweisungen werden durch Prädikatentransformer, d. h. Funktionen von Prädikaten nach Prädikaten definiert. Prädikate sind Funktionen von Zuständen nach booleschen Werten. Das bedeutet, daß Anweisungen Funktionen höherer Stufe sind, wie sie sich im typisierten λ-Kalkül und damit auch in $F^{\omega}_{\leq}$ ausdrücken lassen. Zustände werden mit Verbunden (partiellen Abbildungen von Namen nach Werten) gleichgesetzt. Falls S, T Zustandsräume (Verbundtypen) sind, ist der Typ der Prädikate und der Prädikatentransformer definiert durch:

$$\begin{aligned} \mathit{Pred}\ S &= S \to \mathit{Bool} \\ \mathit{Tran}\ S\ T &= \mathit{Pred}\ T \to \mathit{Pred}\ S \end{aligned}$$

(Man beachte, daß Prädikatentransformer kontravariant Prädikate über dem finalen Zustandsraum auf Prädikate über dem initialen Zustandsraum abbilden,

deshalb die Umkehrung der Reihenfolge. Initialer und finaler Zustandsraum müssen nicht identisch sein.)

Objekte werden durch existentielle Typen wie folgt kodiert. Die Attribute eines Objektes bilden zusammen einen Verbund. Methoden sind, sofern sie keine Parameter haben, Prädikatentransformer über den Attributen. Sie bilden zusammen ebenfalls einen Verbund. Ein Objekt besteht aus einem Paar von Attribut- und Methodenverbunden, wobei der Typ des Attributverbundes versteckt ist. Ein Objekt mit einer parameterlosen Methode m ist demnach vom Typ

$$(\sum Attr \bullet Attr \times (m : Tran\ Attr\ Attr)) \ ,$$

hier geschrieben als

$$object\ m()\ end \ .$$

Falls die Attribute teilweise öffentlich sind, wird dies durch eine Schranke bei dem existentiellen Quantor ausgedrückt. Falls $p : P$ ein öffentliches Attribut sein soll, wird gefordert, daß der Typ der Attribute ein Untertyp des Verbundtyps $(p : P)$ sein muß, d. h.

$$(\sum Attr <: (p : P) \bullet Attr \times (m : Tran\ Attr\ Attr)) \ ,$$

hier geschrieben als

$$object\ p : P, m()\ end \ .$$

Die Übergabe von Wert- bzw. Resultatparametern wird durch Vergrößern der initialen bzw. finalen Zustandsräume der Methoden erreicht. Für zwei Verbundtypen S und T mit unterschiedlichen Feldnamen sei $S \oplus T$ der Verbundtyp mit den Feldern von S und T. Hat beispielsweise die Methode m einen Wertparameter $val : V$ und Resultatparameter $res : R$, ist der Typ eines Objektes mit Methode m

$$(\sum Attr \bullet Attr \times (m : Tran\ (Attr \oplus (val : V))\ (Attr \oplus (res : R)))) \ ,$$

hier geschrieben als

$$object\ m(val : V)\ res : R\ end \ .$$

Die Methode m kann bei Objekten von diesem Typ somit sowohl die Attribute ändern als auch ein Resultat liefern. Es wird nicht unterschieden zwischen rein lesenden und evtl. ändernden Methoden. (Wie bereits in (Hoare, 1972) beobachtet ist eine solche Unterscheidung nicht sinnvoll weil eine lesende Operation durch

eine mit einem nicht beobachtbaren Seiteneffekt implementiert werden kann.) Aus diesem Grund kann ein Methodenaufruf nicht in einem Ausdruck auf der rechten Seite einer Zuweisung oder einer Bedingung stehen. Ein Methodenaufruf ist immer vom Typ Anweisung. Öffentliche Attribute hingegen können hingegen in beliebigen Ausdrücken gelesen werden. Dies ist der Grund, weshalb es nützlich ist, öffentliche Attribute zu betrachten.

Ein Verfeinerungskalkül mit typtheoretischer Basis führt zu einer Ausdrucksmächtigkeit bei der Spezifikation und einer Flexibilität bei der Typisierung, die über existierenden objektorientierten Spezifikations- und Programmiersprachen liegen:

(1) Anweisungen, insbesondere Methodenrümpfe, sind ausführbar oder abstrakt. Abstrakte Anweisungen sind unbeschränkt indeterministisch, gutartig indeterministisch oder wundersam.

(2) Es wird zwischen Objekttypen und Klassen unterschieden. Der Typ eines Objektes beschreibt seine Schnittstelle, d. h. die Signatur der Methoden und die Typen der öffentlichen Attribute, nicht aber sein Verhalten. Klassen dienen als Vorlage beim Erzeugen von Objekten und können aus bestehenden Klassen durch Vererbung entstehen. Klassen beschreiben das Verhalten von Objekten. Objekte, die von unterschiedlichen Klassen erzeugt wurde, können vom gleichen Typ sein.

(3) Auf Typen ist eine strukturelle Untertyprelation gegeben. Ein Wert eines Untertyps kann immer eingesetzt werden, wenn ein Wert eines Obertyps erwartet wird. Für Objekttypen bedeutet dies, daß ein Untertyp zusätzliche öffentliche Attribute und Methoden haben kann. Ein Untertyp muß nicht dieselben privaten Attribute haben, wie der Obertyp.

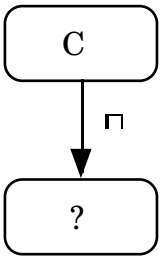
(4) Anweisungen (Prozeduren), Klassen und Typen können mit Typen parametrisiert werden und selbst als Parameter übergeben werden. Typparameter können beschränkt werden.

Die *Klassenverfeinerung* ist das zentrale methodische Konzept für die Formalisierung von Entwurfsprinzipien. Klassenverfeinerung bedeutet:

> Eine Klasse *D* verfeinert eine Klasse *C*, falls alle beobachtbare Eigenschaften von Objekten der Klasse D auch beobachtbare Eigenschaften der Objekte der Klasse C sind. Beobachtbare Eigenschaften eines Objekte sind solche, die sich anhand von globalen Variablen und Resultatparametern nach einer Sequenz von Methodenaufrufen feststellen lassen.

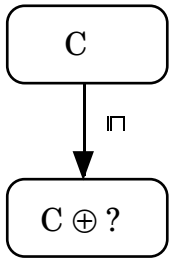
Klassenverfeinerung hat eine ähnliche Rolle und ähnliche Eigenschaften wie die Verfeinerung von Anweisungen, was motiviert, sie als $C \sqsubseteq D$ zu schreiben (gelesen: *C* wird verfeinert durch *D*). Bei der Verfeinerung von Klassen können sich die privaten Attribute ändern, solange das nach außen beobachtbare Verhalten gleich bleibt. Dies geht aber nur, wenn nicht alle Attribute öffentlich sind (wie beispielsweise bei Oberon (Reiser & Wirth, 1992)). Aus diesem Grund wird die Diskussion von Klassenverfeinerung auf Klassen beschränkt, die nur private Attribute haben.

Hier wird eine *verhaltensorientierte* oder *observationelle* Definition der Klassenverfeinerung gegeben, die obigen Sachverhalt direkt wiedergibt. Aus dieser Definition werden Beweisregeln abgeleitet, ähnlich wie sie aus (America, 1987; Lano & Haughton, 1992) bekannt ist. Der Unterschied ist, daß die Abstraktionsfunktion zu einer Relation verallgemeinert werden kann, was in einigen Fällen notwendig ist, um Verfeinerung nachzuweisen. Das Prinzip einer verhaltensorientierten Definition von Verfeinerung wurde für Datentypen (die eine gewisse Verwandtschaft mit Klassen haben) in (He, et al., 1986; Nipkow, 1986) gegeben und wird hier erstmalig auf Klassen angewandt. Klassenverfeinerung wird eingesetzt für folgende drei Entwurfsprinzipien:

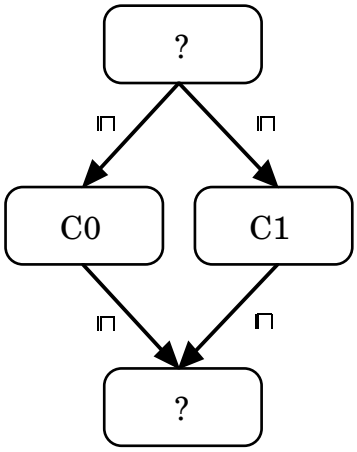


Implementierung von Klassen. Klassen, die bei der Problemanalyse entstehen, können das Verhalten der Objekte auf eine abstrakte, problemnahe Art beschreiben unter Benutzung von abstrakten Anweisungen. Für diese Klassen müssen dann (effiziente) Implementierungen angegeben werden. Dies wird unmittelbar durch Klassenverfeinerung erreicht: Zu einer abstrakten Klasse *C* muß eine Klasse *C*' gefunden werden, so daß $C \sqsubseteq C'$. Die Klasse *C*' kann dabei ausführbar sein oder noch immer abstrakte Anweisungen enthalten, die erst in weiteren Verfeinerungsschritten eliminiert werden. (Aus diesem Grund wird der Begriff Verfeinerung dem Begriff Implementierung vorgezogen.) Verfeinerung kann prinzipiell auf zwei Arten geschehen: Es wird eine verfeinernde Klasse *C*'

vorgeschlagen und dann die Verfeinerungsrelation nachgewiesen, oder es wird eine Klasse *C'* abgeleitet, die per Konstruktion korrekt ist. Die angegebenen Regeln lassen sich für beide Vorgehensweisen einsetzen.



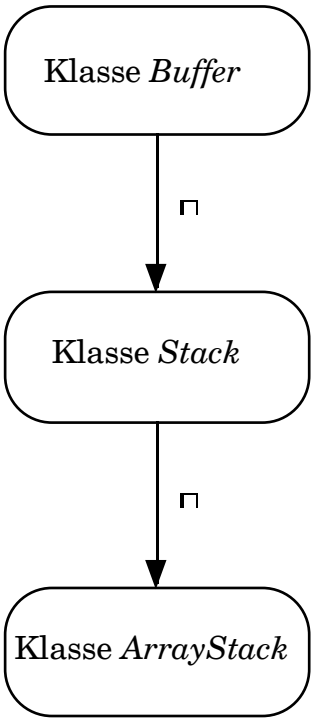
Inkrementelle Modifikation von Klassen. Eine Klasse, die Ähnlichkeiten zu einer bestehenden Klasse hat, kann durch Vererbung kompakter definiert werden. Dabei können Attribute und Methoden hinzukommen oder geändert werden. Für die Situation, daß die neue Klasse die bestehende verfeinern soll, wird eine vereinfachte Beweistechnik gegeben. Sie beruht auf der Idee, lediglich die Differenz zu verifizieren; das Prinzip des „programming by difference“ (Cox, 1986; Johnson & Foote, 1988) wird bei der Vererbung zu „verifying the difference“ fortgesetzt. Die Beweistechnik ist anwendbar, falls die gemeinsamen Attribute der beiden Klassen in der neuen Klasse auf gleiche Art benutzt werden wie in der bestehenden. Sie kann eingesetzt werden, um vorgeschlagene Änderungen nachträglich als korrekt zu beweisen oder um notwendige Änderungen abzuleiten.



Herausfaktorisieren von Gemeinsamkeiten zu neuen Klassen. Falls Klassen *C0* und *C1* ähnliches, aber nicht identisches Verhalten beschreiben, ist es oft zweckmäßig die Gemeinsamkeit in einer neuen Klasse festzuhalten, die durch die beiden Klassen verfeinert wird. Dual kann nach einer Klasse gesucht werden, die sowohl das Verhalten von *C0* als auch das Verhalten von *C1* aufweist. Hierzu werden zwei Operatoren eingeführt, Klassenschnitt und Klassenvereinigung, mit denen ausgehend von zwei (oder mehreren) Klassen eine gemeinsam verfeinerte oder verfeinernde Klassen konstruiert werden können.

Bei diesem Ansatz ergibt sich gegenüber der Spezifikation von Klassen durch Zusicherungen (Abschnitt 2.3) sich ein modifiziertes Bild. Die Spezifikation einer Klasse (dort Typ genannt) wird selbst wieder als Klasse beschrieben. Sowohl das Implementieren einer Spezifikation als auch die Untertypbeziehung zwischen Spezifikationen werden hier durch die Klassenverfeinerung ausgedrückt. Für das gegebene Beispiel könnte dies wie folgt aussehen. Die Klassen *Buffer* und *Stack* haben die Rolle von *Schnittstellenklassen* während die Klasse *ArrayStack* die

Rolle einer *Implementierungsklasse* hat. Das bedeutet, daß alle interessanten Eigenschaften für die Benutzung der Klasse *ArrayStack* bereits durch die Klasse *Stack* beschrieben sind.



Wie bereits in Abschnitt 2.3 diskutiert, gibt es für die Semantik von Anweisungen eine Reihe von Modellen, von denen Prädikatentransformer das einzige Modell sind, in dem Anweisungen einen vollständigen Verband bilden mit der bösartigen Auswahl als Schnitt und der gutartigen Auswahl als Vereinigung. Diese Verbandseigenschaften werden hier bei der Definition von Klassenschnitt und Klassenvereinigung ausgenutzt, so daß andere Modelle für Anweisungen für diesen Zweck ungeeignet wären. Die bekannten Regeln für die prozedurale Verfeinerung (Verfeinerung von Anweisungen) als auch Datenverfeinerung bleiben gültig und können somit übernommen werden.

Für das zugrunde liegende Typsystem $F_{\leq}^{\omega}$ ist das einzig bekannte Modell das von Typen als partiellen Äquivalenzrelationen. (Es ist bekannt, daß das in (Cardelli & Wegner, 1985) vorgeschlagene Modell von Typen als Idealen inkonsistent ist.) Partielle Äquivalenzrelationen sind Relationen, die symmetrisch und transitiv, aber nicht notwendigerweise reflexiv sind. Die Idee dabei ist, daß ein Typ nicht nur eine Menge von Werten definiert, sondern über die Relation auch definiert, welche Werte als gleich zu betrachten sind (Bruce & Longo, 1990). Gleichheit

hängt von dem Typ ab, unter dem die Werte betrachtet werden. Zwei Werte können unter einem Typ betrachtet unterschiedlich sein, unter einem Obertyp dieses Typs betrachtet aber gleich sein: für die Gleichheit von zwei Werten sind nur die „sichtbaren Merkmale“ (z. B. Attribute) relevant und ein Obertyp definiert i. a. weniger sichtbare Merkmale. Dadurch, daß hier auf $F^{\omega}_{\leq}$ aufgesetzt wird, kann von diesem Modell abstrahiert werden. In dem Anhang werden die Regeln von $F^{\omega}_{\leq}$, wie sie sich aus diesem Modell ableiten, aufgeführt.

Die Kodierung von Objekten basierend auf existentiellen Typen ist vergleichbar zu der von (Pierce & Turner, 1994) ebenfalls in $F^{\omega}_{\leq}$. Der Unterschied ist, daß hier Methoden aus imperativen Konstrukten bestehen, in (Pierce & Turner, 1994) sind Methoden nur reine Funktionen. Diese Kodierung durch existentielle Typen erscheint insbesondere in einem imperativen Rahmen natürlicher als die Kodierung durch rekursive Typen. Bei der Kodierung durch rekursive Typen kann weitere Komplexität dadurch entstehen, daß für die Kapselung von Attributen diese zusätzlich mit existentiellen Typen kombiniert werden (Bruce, 1992). Der Ansatz hier beruht ausschließlich auf existentiellen Typen.

Die hier vorgestellte Entwurfsnotation ist gegenüber üblichen typisierten Programmiersprachen (durch die Unterscheidung von Objekttypen und Klassen) flexibler. Trotzdem ist die korrekte Typisierung entscheidbar. Dies ist sichergestellt dadurch, daß alle Definitionen im Typsystem $F^{\omega}_{\leq}$ gegeben werden und die korrekte Typisierung in $F^{\omega}_{\leq}$ entscheidbar ist. Das Ableiten von effizienten Typüberprüfungsalgorithmen ist allerdings nicht Thema dieser Arbeit.

Die methodischen Erkenntnisse bei der Formalisierung o. g. Entwurfsprinzipien sind weitgehend unabhängig davon, ob Objekte eine „Identität“ haben oder nicht. Deshalb werden Identitäten bzw. Referenzen hier optional behandelt, d. h. das Erzeugen eines Objektes liefert das Objekt selbst oder eine Referenz darauf, wie z. B. bei Oberon. Die Aliasing-Problematik kann damit vermieden werden kann, wenn Referenzen von der Sicht der Modellierung aus nicht notwendig sind. Dies weicht von Programmiersprachen wie Smalltalk und Eiffel ab, bei denen nur Objekte per Referenz erzeugt werden können.

3 Der Verfeinerungskalkül

Die ausführbaren Anweisungen der Entwurfsnotation enthalten folgenden Konstrukte, die im wesentlichen Dijkstras „guarded commands“ entsprechen (Dijkstra, 1976).

abort	*chaotische Anweisung*
skip	*leere Anweisung*
$v := e$	*(Mehrfach-) Zuweisung*
$p\ ;\ q$	*sequentielle Komposition*
$p \sqcap q$	*indeterministische Auswahl*
if b then p else q end	*Verzweigung*
while b do p end	*Iteration*

Diese Anweisungen werden eingebettet in einen größeren Raum *abstrakter Anweisungen* mit

(1) unbeschränkt indeterministischen Anweisungen,

(2) wundersamen Anweisungen,

(3) gutartigem Indeterminismus,

(4) Anweisungen mit unterschiedlichem initialen und finalen Zustandsraum.

Nach Vorbemerkungen zur Notation und einer Einführung in die für diese Kapitel benötigten Konzepte von $F^{\omega}_{\leq}$ in 3.1 wird der typisierte Verfeinerungskalkül hier weitgehend so dargestellt, wie aus der Literatur bekannt (Back, 1993); In 3.2 werden Prädikate und Prädikatentransformer eingeführt, in 3.3 elementare Gleichheits- und Verfeinerungsregeln gegeben und in 3.4 drei wichtige Teil-

mengen der Prädikatentransformer diskutiert, monotone, konjunktive und disjunktive Prädikatentransformer.

In 3.5 werden abweichend zu (Back, 1993) Zustände durch Verbunde statt durch Tupel beschrieben, d. h. Programmvariablen werden durch ihren Namen statt durch ihre Position in einem Tupel referenziert. Die Motivation hierfür ist, Programmvariablen und Attribute von Objekten, die immer durch ihren Namen referenziert werden, gleich zu behandeln. Für Prädikate, die als Funktionen von Zuständen nach *Bool* definiert sind, wird eine vereinfachte Schreibweise als boolesche Ausdrücke zugelassen.

Das Formalisieren von Zuständen durch Verbunde ermöglicht, eine Einbettung von Anweisungen zu betrachten, um, ähnlich wie in untypisierten Ansätzen (Morgan, 1990), die Komposition von Anweisungen auf disjunkten Programmvariablen zuzulassen (Abschnitt 3.6). Dies ermöglicht, Kommutativitätseigenschaften von Anweisungen auf disjunkten Programmvariablen zu formulieren (Abschnitt 3.7).

In 3.8 werden Variablen- und Prozedurdeklarationen eingeführt. In 3.9 wird Datenverfeinerung definiert und Verifikationstechniken abgeleitet. Beide Abschnitte nutzen die Eigenschaft von Anweisungen, unterschiedliche initiale und finale Zustandsräume haben zu können. Die Datenverfeinerung basiert auf konjunktiven und disjunktiven Prädikatentransformern.

Die Sätze in 3.2 bis 3.4 werden direkt aus der Literatur übernommen. Sätze in 3.5 bis 3.9, die sich gegenüber der Darstellung in der Literatur wesentlich geändert haben, werden bewiesen.

3.1 Bemerkungen zur Schreibweise

Die Anwendung der Funktion f auf Argument a wird durch $f\,a$ geschrieben. Der Ausdruck $e[f/v]$ steht für die Substitution der freien Vorkommen der Variablen v durch f im Ausdruck e. Dabei können v und f gleich lange Listen von Variablen bzw. Ausdrücken sein. Folgende Aufreihung gibt die Bindungsstärke der Operatoren in absteigender Priorität:

- Substitution
- Applikation
- unäre Operatoren ($\neg$, #, …)
- binäre Operatoren ($\in$, $\cup$, …)
- Konjunktion ($\wedge$), Disjunktion ($\vee$)
- Implikation ($\Rightarrow$), Konsequenz ($\Leftarrow$), Äquivalenz ($\Leftrightarrow$)
- Relationen ($=$, $\sqsubseteq$, $\sqsupseteq$, $\leq$, $\geq$)

Alle vorkommenden Ausdrücke sind typisiert. Typen sind Mengen von Werten, so daß $x : T$ zu verstehen ist als $x \in T$. Bei der Bindung von Variablen können die Typen weggelassen werden, falls sie aus dem Kontext ersichtlich sind, z. B. $(\forall\, x \bullet b)$ statt $(\forall\, x : T \bullet b)$. Typen werden grundsätzlich mit einem Großbuchstaben beginnend geschrieben.

Definitionen werden als $D \mathrel{\widehat{=}} E$ geschrieben, mit der Bedeutung daß im nachfolgenden Text D durch E textuell ersetzt werden kann.

Die Definitionen dieses Kapitels werden in einer Teilmenge von $F^{\omega}_{\leq}$ gegeben, die hier kurz erklärt wird. Die komplette Syntax von $F^{\omega}_{\leq}$ mit allen Regeln ist im Anhang gegeben.

Die *Abstraktion* $(\lambda\, x : T \bullet \mathrm{e})$ steht für eine Funktion, die ein Argument vom Typ T nimmt und x durch dieses in dem Ausdruck e substituiert. Falls e vom Typ U ist, ist $(\lambda\, x : T \bullet \mathrm{e})$ eine Funktion von T nach U:

$$(\lambda\, x : T \bullet e) : T \rightarrow U$$

Dieses Beispiel zeigt die Parametrisierung eines *Wertausdruckes* mit einem Wert. Ein Wertausdruck kann auch mit einem Typ parametrisiert werden. Die *Typabstraktion* $(\lambda\, A \bullet e)$ steht für eine Funktion, die einen Typ als Argument nimmt und A durch diese in dem Ausdruck e substituiert. Falls e vom Typ U ist, lautet die Typisierung:

$$(\lambda\, A \bullet e) : (\prod A \bullet U)$$

$(\prod A \bullet U)$ ist der Typ der Funktionen, die einen beliebigen Typ als Parameter nehmen und einen Wert vom Typ U liefern. Dabei kann U von dem Typparame-

ter abhängen. Statt $(\prod A \bullet U)$ wird in der Literatur auch $(\forall A \bullet U)$ geschrieben, wovon hier wegen der Unterscheidung zu dem logischen Quantor abgesehen wird. Beispielsweise ist

$$\begin{aligned} Id &\mathrel{\widehat{=}} (\lambda A \bullet (\lambda x : A \bullet x)) \\ Id &: (\prod A \bullet A \to A) \end{aligned}$$

die polymorphe Identitätsfunktion. Sie kann mit jedem beliebigen Typ instanziert werden, beispielsweise mit dem Typ *Int*:

$$IntId \mathrel{\widehat{=}} Id\ Int$$

Mit der Regel *Typ–Reduktion* (siehe Anhang) folgt, daß *IntId* äquivalent ist zu:

$$IntId = (\lambda x : Int \bullet x)$$

Mit der Regel :–*Typapplikation* folgt für den Typ von *IntId*:

$$IntId : Int \to Int.$$

Als Basistypen werden einfache Typen wie *Int* (die ganzen Zahlen) und *Bool* (mit den Werten *true* und *false*) verwendet, als auch zusammengesetzte Typen wie Mengen *set of T*, Sequenzen *seq of T*, Multimengen *bag of T* und Kreuzprodukte $T_1 \times \ldots \times T_n$, jeweils mit den üblichen Operationen. Insbesondere werden Verbunde (records) eingesetzt. Es seien $l_1, \ldots, l_n$ Namen von *Feldern*:

$(l_1 : T_1, \ldots, l_n : T_n)$	*Verbundtyp*
$(l_1 \mapsto e_1, \ldots, l_n \mapsto e_n)$	*Verbund*
$r.l$	*Feldselektion*
$R \oplus S$	*Überschreiben von Verbundtypen*
$r \oplus s$	*Überschreiben von Verbunden*
$dom\ R$	*Feldnamen*
$r \mid \{l_1, \ldots, l_n\}$	*Einschränkung*

Eine alternative Schreibweise für Verbundtypen und Verbunde ist

$$\begin{aligned} &(l_1, \ldots, l_n : T_1, \ldots, T_n)\ , \\ &(l_1, \ldots, l_n \mapsto e_1, \ldots, e_n)\ . \end{aligned}$$

oder, falls $l = l_1, \ldots, l_n$ eine Liste von Namen, $T = T_1, \ldots, T_n$ eine Liste von Typen und $e = e_1, \ldots e_n$ eine Liste von Werten ist, kurz

$$\begin{aligned} &(l : T)\ , \\ &(l \mapsto e)\ . \end{aligned}$$

Es werden auch Mischformen beider Schreibweisen zugelassen, beispielsweise falls $k = k_1, \ldots, k_m$ eine Liste von Namen, $S = S_1, \ldots, S_m$ eine Liste von Typen und $f = f_1, \ldots f_m$ eine Liste von Werten ist:

$$(l : T, k : S)$$
$$(l \mapsto e, k \mapsto f)$$

Das Überschreiben von Verbunden und Verbundtypen ist asymmetrisch: Der zweite Operand überschreibt die Inhalte der gleichnamigen Felder des ersten Operanden und hängt neue Felder an.

Diese Basistypen können im „reinen“ λ-Kalkül ohne Basistypen kodiert werden (Mitchell, 1990). Im folgenden wird davon ausgegangen, daß sie mit der üblichen Semantik zur Verfügung stehen.

In $F^{\omega}_{\leq}$ werden Beweise mit den Inferenzregeln für Äquivalenz geführt. Diese Regeln sind im Anhang aufgelistet. Beispielsweise kann mit der Äquivalenzregel *=–Symmetrie* abgeleitet werden, daß für Zahlen x, y aus $x + 3 = y$ folgt $y = x + 3$, geschrieben:

$$\frac{x : Int, y : Int \vdash x + 3 = y}{x : Int, y : Int \vdash y = x + 3}$$

Um Beweise kompakter darzustellen, werden sie im sog. „calculational style“ geführt, etwa wie in (Dijkstra & Scholten, 1990). Obiger Beweis wird geschrieben als:

$$\begin{array}{ll} & x + 3 = y \\ \Rightarrow & \text{«=–Symmetrie»} \\ & y = x + 3 \end{array}$$

Das Symbol ⇒ steht dabei für die Inferenz. Die Typisierungen der vorkommenden Variablen werden weggelassen und müssen aus dem Kontext hervorgehen. Der Name der Regeln wird in «…» Klammern geschrieben. Häufig werden mehrere Inferenzschritte zu einem zusammengefaßt. Einfache Regeln wie Assoziativität und Symmetrie werden oft implizit angewandt. Falls die Inferenz in beiden Richtungen möglich ist, wie in diesem Beispiel, wird ⇔ statt ⇒ geschrieben. Falls aus der unteren Zeile die obere folgt wird sinngemäß ⇐ geschrieben.

Beweise gewisser Bauart können weiter vereinfacht werden. Falls beispielsweise in der Umgebung (Liste von Typisierungen) E gezeigt werden soll, daß $b = true$ gilt und bereits gezeigt wurde, daß $b = c$, $c = d$ und $d = true$ gilt, geht der Beweis mit zweimaliger Anwendung von $=$–*Transitivität*:

$$\dfrac{\dfrac{E \vdash b = c \qquad E \vdash c = d}{E \vdash b = d} \qquad E \vdash d = true}{E \vdash b = true}$$

Dies wird „linear“ ausgedrückt durch:

$$\begin{array}{ll} & b \\ = & \text{«wegen } b = c\text{»} \\ & c \\ = & \text{«wegen } c = d\text{»} \\ & d \\ = & \text{«wegen } d = true\text{»} \\ & true \end{array}$$

Die Regel $=$–*Transitivität* wird nicht explizit erwähnt. Diese Art von Beweisen kann mit jeder transitiven, binären Relation geführt werden.

3.2 Prädikate und Prädikatentransformer

Im folgenden seien R, S, T Typen, die die Rolle von *Zustandsräumen* haben. (Zustandsräume können für die Zwecke dieses und folgender Abschnitte beliebige Typen sein; sie werden in Abschnitt 3.5 eingeschränkt). Ein *Zustandsprädikat* über einem Zustandsraum S ist eine Funktion von S nach *Bool*, also vom Typ[2]

$$Pred\ S \ \widehat{=}\ S \to Bool \qquad \textit{Zustandsprädikat}$$

Konstante Zustandsprädikate sind:

$$\begin{array}{ll} True\ S \ \widehat{=}\ (\lambda\, \sigma : S \bullet true) & \textit{True} \\ False\ S \ \widehat{=}\ (\lambda\, \sigma : S \bullet false) & \textit{False} \end{array}$$

[2]In $F^{\omega}_{\leq}$ läßt sich *Pred* präziser definieren durch $Pred \mathrel{\widehat{=}} (\Lambda\ S\ ^{\cdot}\ S \to Bool)$ als eine Funktion von Typen nach Typen, d. h. von der Art $* \to *$. Weil *Pred* nie alleine, sondern immer nur mit einem Typargument S vorkommt, wird die obige, einfachere Schreibweise vorgezogen.

Der Typ von *True S* und *False S* ist *Pred S*.

Auf Zustandsprädikaten sind die üblichen booleschen Operatoren durch die *punktweise Erweiterung* der entsprechenden Operatoren auf *Bool* definiert, also für $\sigma : S$ und $\alpha, \beta : Pred\ S$ gilt:

$$\begin{array}{lcll}
\neg\alpha & \widehat{=} & (\lambda\ \sigma : S \bullet \neg(\alpha\ \sigma)) & \textit{Negation} \\
\alpha \wedge \beta & \widehat{=} & (\lambda\ \sigma : S \bullet \alpha\ \sigma \wedge \beta\ \sigma) & \textit{Konjunktion} \\
\alpha \vee \beta & \widehat{=} & (\lambda\ \sigma : S \bullet \alpha\ \sigma \vee \beta\ \sigma) & \textit{Disjunktion} \\
\alpha \Rightarrow \beta & \widehat{=} & (\lambda\ \sigma : S \bullet \alpha\ \sigma \Rightarrow \beta\ \sigma) & \textit{Implikation} \\
\alpha \Leftarrow \beta & \widehat{=} & (\lambda\ \sigma : S \bullet \alpha\ \sigma \Leftarrow \beta\ \sigma) & \textit{Konsequenz} \\
\alpha \Leftrightarrow \beta & \widehat{=} & (\lambda\ \sigma : S \bullet \alpha\ \sigma \Leftrightarrow \beta\ \sigma) & \textit{Äquivalenz}
\end{array}$$

Der Typ von $\neg\alpha$, $\alpha \wedge \beta$, ... ist *Pred S*.

Die *universelle Implikation*, *universelle Konsequenz* und *Gleichheit* (*universelle Äquivalenz*) von Zustandsprädikaten $\alpha, \beta : Pred\ S$ ist definiert durch:

$$\begin{array}{lcll}
\alpha \preceq \beta & \widehat{=} & (\forall\ \sigma : S \bullet \alpha\ \sigma \Rightarrow \beta\ \sigma) & \textit{universelle Implikation} \\
\alpha \succeq \beta & \widehat{=} & (\forall\ \sigma : S \bullet \alpha\ \sigma \Leftarrow \beta\ \sigma) & \textit{universelle Konsequenz} \\
\alpha = \beta & \widehat{=} & (\forall\ \sigma : S \bullet \alpha\ \sigma \Leftrightarrow \beta\ \sigma) & \textit{Gleichheit}
\end{array}$$

Falls α durch $i \in I$ indiziert ist (I beliebig), ist die *Quantifizierung* von Zustandsprädikaten definiert durch:

$$\begin{array}{lcll}
(\forall\ i \in I \bullet \alpha) & \widehat{=} & (\lambda\ \sigma : S \bullet (\forall\ i \in I \bullet \alpha\ \sigma)) & \textit{universelle Quantifizierung} \\
(\exists\ i \in I \bullet \alpha) & \widehat{=} & (\lambda\ \sigma : S \bullet (\exists\ i \in I \bullet \alpha\ \sigma)) & \textit{existentielle Quantifizierung}
\end{array}$$

Eine Anweisung p mit initialem Zustandsraum S und finalem Zustandsraum T wird identifiziert mit einer Funktion, die Zustandsprädikate über T abbildet auf Zustandsprädikate über S. Es sei $\beta : Pred\ T$ ein Zustandsprädikat. Dann ist $p\ \beta$ die schwächste Vorbedingung, so daß p terminiert und danach β gilt. Der Typ *Tran S T* der Prädikatentransformer von Zustandsraum S nach Zustandsraum T ist definiert durch:

$$Tran\ S\ T \quad \widehat{=} \quad Pred\ T \rightarrow Pred\ S \qquad \textit{Prädikatentransformer}$$

Es werden einige elementare Anweisungen vom Typ *Tran S T* definiert. Die Anweisung *Miracle S T* führt zu jeder gewollten Nachbedingung. Die Anweisung *Abort S T* führt zu keiner (kontrollierbaren) Nachbedingung. Die Anweisung *Skip T* vom Typ *Tran T T* läßt den Zustand unverändert:

$$\begin{array}{lcll} \mathit{Miracle}\ S\ T & \widehat{=} & (\lambda\ \alpha : \mathit{Pred}\ T \bullet \mathit{True}\ S) & \textit{wundersame Anweisung} \\ \mathit{Abort}\ S\ T & \widehat{=} & (\lambda\ \alpha : \mathit{Pred}\ T \bullet \mathit{False}\ S) & \textit{chaotische Anweisung} \\ \mathit{Skip}\ T & \widehat{=} & (\lambda\ \alpha : \mathit{Pred}\ T \bullet \alpha) & \textit{leere Anweisung} \end{array}$$

Es seien β vom Typ $\mathit{Pred}\ T$. Die Annahme $[\beta]$ verhält sich wie die leere Anweisung falls β gilt, sonst wie wundersam. Die Zusicherung $\{\beta\}$ verhält sich ebenfalls wie die leere Anweisung falls β gilt, sonst chaotisch:

$$\begin{array}{lcll} [\beta] & \widehat{=} & (\lambda\ \alpha : \mathit{Pred}\ T \bullet \beta \Rightarrow \alpha) & \textit{Annahme} \\ \{\beta\} & \widehat{=} & (\lambda\ \alpha : \mathit{Pred}\ T \bullet \beta \wedge \alpha) & \textit{Zusicherung} \end{array}$$

Es sei $\phi : S \to T$ eine *Zustandsfunktion* vom Zustandsraum S nach T. Die Anweisung $\langle\phi\rangle : \mathit{Tran}\ S\ T$ führt die durch ϕ gegebene Zustandstransformation aus. Sie ist ähnlich einer Mehrfachzuweisung, verallgemeinert sie aber dahingehend, daß initialer und finaler Zustandsraum nicht identisch sein müssen:

$$\langle\phi\rangle \quad \widehat{=} \quad (\lambda\ \alpha : \mathit{Pred}\ T \bullet (\lambda\ \sigma : S \bullet \alpha\ (\phi\ \sigma))) \qquad \textit{Zustandstransformation}$$

Es sei $p : \mathit{Tran}\ R\ S$ und es seien $q, r : \mathit{Tran}\ S\ T$. Die sequentielle Komposition $p\ ;\ q$ entspricht der Hintereinanderausführung von p und q. Die Auswahl $q \sqcap r$ führt zu einer Nachbedingung, falls *sowohl q als auch r* zu dieser Nachbedingung führen. Die Auswahl wird als *bösartig* (*demonic*, *malvolent*) bezeichnet. Die Auswahl $q \sqcup r$ führt zu einer Nachbedingung falls *q oder r* zu dieser Nachbedingung führen. Die Auswahl wird als *gutartig* (*angelic*, *clairvoyant*) bezeichnet:

$$\begin{array}{lcll} p\ ;\ q & \widehat{=} & (\lambda\ \alpha : \mathit{Pred}\ T \bullet p\ (q\ \alpha)) & \textit{sequentielle Komposition} \\ q \sqcap r & \widehat{=} & (\lambda\ \alpha : \mathit{Pred}\ T \bullet q\ \alpha \wedge r\ \alpha) & \textit{binäre bösartige Auswahl} \\ q \sqcup r & \widehat{=} & (\lambda\ \alpha : \mathit{Pred}\ T \bullet q\ \alpha \vee r\ \alpha) & \textit{binäre gutartige Auswahl} \end{array}$$

Falls q durch $i \in I$ indiziert ist, sind die entsprechenden allgemeinen Formen der Auswahl definiert durch:

$$\begin{array}{lcll} (\sqcap\ i \in I \bullet q) & \widehat{=} & (\lambda\ \alpha : \mathit{Pred}\ T \bullet (\forall\ i \in I \bullet q\ \alpha)) & \textit{bösartige Auswahl} \\ (\sqcup\ i \in I \bullet q) & \widehat{=} & (\lambda\ \alpha : \mathit{Pred}\ T \bullet (\exists\ i \in I \bullet q\ \alpha)) & \textit{gutartige Auswahl} \end{array}$$

3.3 Gleichheit und Verfeinerung von Prädikatentransformern

Die Anweisungen q und r sind gleich, falls sie zu gleichen Nachbedingungen unter gleichen Vorbedingungen führen. Anweisung q wird verfeinert durch

Anweisung r, falls die Vorbedingungen, unter denen q zu einer Nachbedingung führt, auch Vorbedingungen von r für diese Nachbedingung sind. Es seien q, r vom Typ $Tran\ S\ T$:

$$q = r \quad \widehat{=} \quad (\forall\ \alpha : Pred\ T \bullet q\ \alpha = r\ \alpha) \qquad \textit{Gleichheit von Anweisungen}$$
$$q \sqsubseteq r \quad \widehat{=} \quad (\forall\ \alpha : Pred\ T \bullet q\ \alpha \leq r\ \alpha) \qquad \textit{Verfeinerung von Anweisungen}$$
$$q \sqsupseteq r \quad \widehat{=} \quad (\forall\ \alpha : Pred\ T \bullet q\ \alpha \geq r\ \alpha)$$

Ein Hoare-Triple $\{pre\}\ q\ \{post\}$ ist definiert durch:

$$\{pre\}\ q\ \{post\} \quad \widehat{=} \quad pre \leq q\ post \text{ für alle } pre, post$$

Die Verfeinerung von Anweisungen erhält totale Korrektheit:

$$q \sqsubseteq r \quad \text{gdw für alle } pre, post\text{: } \{pre\}\ q\ \{post\} \Rightarrow \{pre\}\ r\ \{post\}$$

Folgende vier Sätze veranschaulichen die Möglichkeiten der Verfeinerung:

$$[\beta] \sqsubseteq [\gamma] \quad \text{gdw} \quad \beta \geq \gamma \qquad \sqsubseteq\textit{–Annahme}$$
$$\{\beta\} \sqsubseteq \{\gamma\} \quad \text{gdw} \quad \beta \leq \gamma \qquad \sqsubseteq\textit{–Zusicherung}$$
$$q \sqcap r \sqsubseteq q \qquad \sqsubseteq\text{–}\sqcap$$
$$q \sqsubseteq q \sqcup r \qquad \sqsubseteq\text{–}\sqcup$$

Die Verfeinerung kann den Bereich des wundersamen Verhaltens vergrößern, den Bereich des chaotischen Verhaltens verkleinern, bösartigen Indeterminismus reduzieren und gutartigen Indeterminismus hinzufügen.

Die Prädikatentransformer eines Typs bilden mit der Verfeinerungsrelation $\sqsubseteq$ als Ordnung, mit der bösartigen Auswahl $\sqcap$ als Schnitt und mit der gutartigen Auswahl $\sqcup$ als Vereinigung einen vollständigen Verband:

- Die Verfeinerungsrelation ist eine partielle Ordnung, d. h. reflexiv, transitiv und antisymmetrisch

$$q \sqsubseteq q \qquad \sqsubseteq\textit{–reflexiv}$$
$$(q \sqsubseteq r) \wedge (r \sqsubseteq s) \Rightarrow (q \sqsubseteq s) \qquad \sqsubseteq\textit{–transitiv}$$
$$(q \sqsubseteq r) \wedge (r \sqsubseteq q) \Rightarrow (q = r) \qquad \sqsubseteq\textit{–antisymmetrisch}$$

- Für zwei Prädikatentransformer gibt es immer eine kleinste obere und größte untere Schranke:

$$(q \sqsubseteq r) \wedge (q \sqsubseteq s) \Leftrightarrow (q \sqsubseteq r \sqcap s) \qquad \sqsubseteq\textit{–größte untere Schranke}$$
$$(r \sqsubseteq q) \wedge (s \sqsubseteq q) \Leftrightarrow (r \sqcup s \sqsubseteq q) \qquad \sqsubseteq\textit{–kleinste obere Schranke}$$

- Weiterhin gibt es für jede beliebige Menge von Prädikatentransformer eine größte untere und kleinste obere Schranke, d.h. der Verband ist vollständig.

In vollständigen Verbänden gibt es ein kleinstes und ein größtes Element:

$$Abort\ S\ T \sqsubseteq q \qquad \sqsubseteq\text{–}abort$$
$$q \sqsubseteq Miracle\ S\ T \qquad \sqsubseteq\text{–}miracle$$

Die sequentielle Komposition ist assoziativ, hat die leere Anweisung als Einselement und die chaotische und wundersame als Linksnull. Es sei $q : Tran\ S\ T$:

$$\begin{array}{lcll} (p\ ;\ q)\ ;\ r & = & p\ ;\ (q\ ;\ r) & ;\text{–}assoziativ \\ Skip\ S;\ q & = & q & ;\text{–}skip \\ & = & q\ ;\ Skip\ T & \\ Abort\ R\ S\ ;\ q & = & Abort\ R\ T & ;\text{–}abort\text{–}linksnull \\ Miracle\ R\ S\ ;\ q & = & Miracle\ R\ T & ;\text{–}miracle\text{–}linksnull \end{array}$$

Die bösartige Auswahl ist idempotent, symmetrisch und assoziativ. Sie hat die chaotische Anweisung als Nullelement und die wundersame als Einselement. Es seien p, q, r vom Typ $Tran\ S\ T$:

$$\begin{array}{lcll} q \sqcap q & = & q & \sqcap\text{–}idempotent \\ q \sqcap r & = & r \sqcap q & \sqcap\text{–}symmetrisch \\ (q \sqcap r) \sqcap s & = & q \sqcap (r \sqcap s) & \sqcap\text{–}assoziativ \\ q \sqcap Abort\ S\ T & = & Abort\ S\ T & \sqcap\text{–}abort \\ q \sqcap Miracle\ S\ T & = & q & \sqcap\text{–}miracle \end{array}$$

Die gutartige Auswahl ist idempotent, symmetrisch und assoziativ. Sie hat aber die chaotische Anweisung als Einselement und die wundersame als Nullelement:

$$\begin{array}{lcll} q \sqcup q & = & q & \sqcup\text{–}idempotent \\ q \sqcup r & = & r \sqcup q & \sqcup\text{–}symmetrisch \\ (q \sqcup r) \sqcup s & = & q \sqcup (r \sqcup s) & \sqcup\text{–}assoziativ \\ q \sqcup Abort\ S\ T & = & q & \sqcup\text{–}abort \\ q \sqcup Miracle\ S\ T & = & Miracle\ S\ T & \sqcup\text{–}miracle \end{array}$$

Die bösartige (gutartige) Auswahl ist distributiv über die binäre gutartige (bösartige) Auswahl. Sowohl die bösartige als auch gutartige Auswahl sind linksdistributiv über die sequentielle Komposition. Es seien p und q durch $i \in I$ indiziert:

$$(\sqcap i \in I \bullet q) \sqcup r = (\sqcap i \in I \bullet q \sqcup r) \qquad \sqcap\text{–}\sqcup\text{–distributiv}$$
$$(\sqcup i \in I \bullet q) \sqcap r = (\sqcup i \in I \bullet q \sqcap r) \qquad \sqcup\text{–}\sqcap\text{–distributiv}$$
$$(\sqcap i \in I \bullet p) ; q = (\sqcap i \in I \bullet p ; q) \qquad \sqcap\text{–;–linksdistributiv}$$
$$(\sqcup i \in I \bullet p) ; q = (\sqcup i \in I \bullet p ; q) \qquad \sqcup\text{–;–linksdistributiv}$$

Alle bisher definierten Operatoren sind monoton bezüglich Verfeinerung:

$$(q \sqsubseteq r) \Rightarrow (p ; q \sqsubseteq p ; r) \qquad ;\text{–monoton}$$
$$\wedge \quad (q ; s \sqsubseteq r ; s)$$
$$(q \sqsubseteq r) \Rightarrow (\sqcap i \in I \bullet q) \sqsubseteq (\sqcap i \in I \bullet r) \qquad \sqcap\text{–monoton}$$
$$(q \sqsubseteq r) \Rightarrow (\sqcup i \in I \bullet q) \sqsubseteq (\sqcup i \in I \bullet r) \qquad \sqcup\text{–monoton}$$

Eine wahre Zusicherung von ist die leere Anweisung, eine falsche Zusicherung ist die chaotische Anweisung. Die sequentielle Komposition von Zusicherungen entspricht der Konjunktion der Prädikate. Eine gutartige Auswahl von Zusicherungen ist die leere Anweisung, falls eine der Zusicherungen wahr ist. Sie ist chaotisch, falls alle Zusicherungen falsch sind:

$$\{True\ T\} = Skip\ T \qquad true\text{–Zusicherung}$$
$$\{False\ T\} = Abort\ T\ T \qquad true\text{–Zusicherung}$$
$$\{\beta\} ; \{\gamma\} = \{\beta \wedge \gamma\} \qquad ;\text{–Zusicherung}$$
$$(\sqcup i \in I \bullet \{\beta\}) = \{\exists i \in I \bullet \beta\} \qquad \sqcup\text{–Zusicherung}$$

Eine wahre Annahme ist ebenfalls die leere Anweisung, eine falsche Annahme ist die wundersame Anweisung. Die sequentielle Komposition von Annahmen entspricht der Konjunktion der Prädikate. Eine bösartige Auswahl von Annahmen ist die leere Anweisung, falls eine der Annahmen wahr ist. Sie ist wundersam, falls alle Annahmen falsch sind:

$$[True\ T] = Skip\ T \qquad true\text{–Annahme}$$
$$[False\ T] = Miracle\ T\ T \qquad false\text{–Annahme}$$
$$[\beta] ; [\gamma] = [\beta \wedge \gamma] \qquad ;\text{–Annahme}$$
$$(\sqcap i \in I \bullet [\beta]) = [\exists i \in I \bullet \beta] \qquad \sqcap\text{–Annahme}$$

Die Zustandstransformation mit der Identität ist die leere Anweisung. Die sequentielle Komposition von Zustandstransformation entspricht ihrer umgekehrten Hintereinanderausführung:

$$\langle Id\ T\rangle = Skip\ T \qquad Id\text{–Zustandstransformation}$$
$$\langle \phi \rangle ; \langle \psi \rangle = \langle \psi \circ \phi \rangle \qquad ;\text{–Zustandstransformation}$$

Die sequentielle Komposition einer Zustandstransformation mit einer Zusicherung (Annahme) läßt sich in der Reihenfolge vertauschen, falls sich dabei die Zusicherung (Annahme) auf den finalen Zustand bezieht:

$$\langle \phi \rangle \,;\, \{\beta\} \quad = \quad \{\beta \circ \phi\} \,;\, \langle \phi \rangle \qquad \textit{Zustandstransformation–Zusicherung}$$
$$\langle \phi \rangle \,;\, [\beta] \quad = \quad [\beta \circ \phi] \,;\, \langle \phi \rangle \qquad \textit{Zustandstransformation–Annahme}$$

Die Verzweigung läßt sich mit den bisherigen Operatoren direkt definieren durch:

$$\textit{if } \beta \textit{ then } q \textit{ else } r \textit{ end} \;\; \widehat{=} \;\; ([\beta] \,;\, q) \sqcap [\neg\beta] \,;\, r) \qquad \textit{Verzweigung}$$

Eine dazu äquivalente Definition ist:

$$\textit{if } \beta \textit{ then } q \textit{ else } r \textit{ end} \;\; \widehat{=} \;\; (\{\beta\} \,;\, q) \sqcup \{\neg\beta\} \,;\, r)$$

Für Verzweigungen gelten eine Reihe abgeleiteter Gesetze:

$$[\alpha] \,;\, \textit{if } \alpha \vee \beta \textit{ then } q \textit{ else } r \textit{ end} \quad = \quad [\alpha] \,;\, q \qquad \textit{if–true–Zweig}$$
$$[\neg\alpha] \,;\, \textit{if } \alpha \wedge \beta \textit{ then } q \textit{ else } r \textit{ end} \quad = \quad [\neg\alpha] \,;\, r \qquad \textit{if–false–Zweig}$$
$$\textit{if } \alpha \textit{ then } q \textit{ else } r \textit{ end} \quad = \quad \textit{if } \neg\alpha \textit{ then } r \textit{ else } q \textit{ end} \qquad \textit{if–Negation}$$
$$\textit{if } \beta \textit{ then } q \textit{ else } r \textit{ end} \quad = \quad \textit{if } \beta \textit{ then } [\beta] \,;\, q \textit{ else } r \textit{ end} \qquad \textit{if–true–Annahme}$$
$$\textit{if } \beta \textit{ then } q \textit{ else } r \textit{ end} \quad = \quad \textit{if } \beta \textit{ then } q \textit{ else } [\neg\beta] \,;\, r \textit{ end} \qquad \textit{if–false–Annahme}$$
$$\textit{if } \beta \textit{ then } q \textit{ else } q \textit{ end} \quad = \quad q \qquad \textit{if–idempotent}$$
$$\textit{if } \beta \textit{ then } q \textit{ else } r \textit{ end} \,;\, s \quad = \quad \textit{if } \beta \textit{ then } q \,;\, s \textit{ else } r \,;\, s \textit{ end} \qquad \textit{if–linksdistributiv}$$

Es sei q : *Tran T T*. Die Verzweigung ohne *else* ist definiert durch:

$$\textit{if } \beta \textit{ then } q \textit{ end} \;\; \widehat{=} \;\; \textit{if } \beta \textit{ then } q \textit{ else Skip } T \textit{ end}$$

Die Verzweigung ist aus monotonen Operatoren definiert, also gilt wegen der Transitivität entsprechend Monotonie:

$$(q \sqsubseteq r) \;\Rightarrow\; \textit{if } \beta \textit{ then } q \textit{ else } p \textit{ end} \sqsubseteq \textit{if } \beta \textit{ then } r \textit{ else } p \textit{ end} \qquad \textit{if–monoton}$$
$$\wedge \quad \textit{if } \beta \textit{ then } p \textit{ else } q \textit{ end} \sqsubseteq \textit{if } \beta \textit{ then } p \textit{ else } r \textit{ end}$$

Es sei F eine Funktion von Prädikatentransformern nach Prädikatentransformern. Der Operator $(\mu\, X \bullet F\, X)$ steht für eine Rekursion mit gebundenem Namen X und Rumpf $F\, X$, in dem X aufgerufen wird. Die Rekursion kann explizit definiert werden durch Schnitte (bösartige Auswahl):

$$(\mu\, X \bullet F\, X) \;\; \widehat{=} \;\; (\sqcap\, q \in \{X \mid X = F\, X\} \bullet q) \qquad \textit{Rekursion}$$

Die Verbandseigenschaften ermöglichen die Anwendung des Satzes von Knaster-Tarski: Falls F monoton ist, ist $(\mu\, X \bullet F\, X)$ der kleinste Fixpunkt von F, d. h. die kleinste Lösung der Gleichung $X = F\, X$:

$$(\mu\ X \bullet F\, X) \ = \ F\,(\mu\ X \bullet F\, X) \qquad \mu\text{–Fixpunkt}$$
$$(F\, Y \sqsubseteq Y) \ \Rightarrow \ ((\mu\ X \bullet F\, X) \sqsubseteq Y) \qquad \mu\text{–minimal}$$

Die Iteration wird definiert durch eine Schwanzrekursion:

$$\textit{while}\ \beta\ \textit{do}\ p\ \textit{end} \ \widehat{=} \ (\mu X \ \bullet \textit{if}\ \beta\ \textit{then}\ p\ ;X\ \textit{end}) \qquad \textit{Iteration}$$

Für die Iteration gilt entsprechend Monotonie:

$$(q \sqsubseteq r) \ \Rightarrow \ \textit{while}\ \beta\ \textit{do}\ q\ \textit{end} \sqsubseteq \textit{while}\ \beta\ \textit{do}\ r\ \textit{end} \qquad \textit{while–monoton}$$

Unter der zusätzlichen Annahme von *Stetigkeit* läßt sich Rekursion und Iteration alternativ definieren. Eine Funktion F von der Menge $Tran\ S\ T$ in sich heißt *stetig*, falls für alle Prädikatentransformer $r\, i$, mit $r\, i \sqsubseteq r\,(i{+}1)$ für $i \geq 0$ gilt:

$$F\,(\sqcup\, i \geq 0 \bullet r\, i) \ = \ (\sqcup\, i \geq 0 \bullet F\,(r\, i))$$

In diesem Fall kann die Rekursion durch eine Folge von Approximationen definiert werden. Es sei

$$F^0\, q \ = \ q \ ,$$
$$F^{i+1}\, q \ = \ F\,(F^i\, q) \ .$$

Es sei $\bot = \textit{Abort}\ S\ T$. Dann entspricht $F^i\, \bot$ dem Verhalten einer Rekursion, deren Tiefe i nicht überschreitet. Weil F monoton ist, gilt $F^i\, \bot \sqsubseteq F^{i+1}\, \bot$, d. h. $F^{i+1}\, \bot$ approximiert das Verhalten der Rekursion besser als $F^i\, \bot$. Ist die Funktion F stetig, ist die Rekursion gleich dem Grenzwert dieser Folge:

$$(\mu\, X \bullet F\, X) \ = \ (\sqcup\, i \geq 0 \bullet F^i\, \bot)$$

Die sequentielle Komposition $q\ ;r$, die binäre gutartige $q \sqcup r$ und die binäre bösartige Auswahl $q \sqcap r$ sind jeweils in beiden Operanden q und r stetig. Die unbeschränkte gutartige Auswahl $(\sqcup\, i \in I \bullet q)$ ist ebenfalls stetig, die unbeschränkte bösartige Auswahl $(\sqcap\, i \in I \bullet q)$ ist es nicht (Dijkstra, 1982), d. h. Rekursion kann nicht über den Grenzwert definiert werden falls unbeschränkte bösartige Auswahl (z. B. in Form von Spezifikationsanweisungen) zugelassen wird. Hierzu ist es notwenig, Rekurison und Iteration über Schnitte zu definieren. In (Dijkstra & Gasteren, 1986) wird gezeigt, wie ein Iterationstheorem (mit einer Terminierungsfunktion über eine wohlgeordnete Menge) unter dieser schwächeren Voraussetzung abgeleitet werden kann.

3.4 Monotone, konjunktive und disjunktive Prädikatentransformer

Ein Prädikatentransformer $q : \mathit{Tran}\ S\ T$ heißt *monoton* genau dann, wenn für alle $\alpha, \beta : \mathit{Pred}\ T$ gilt:

$$(\alpha \leq \beta) \ \Rightarrow \ (q\ \alpha \leq q\ \beta)$$

(Dies unterscheidet sich von dem Monotoniebegriff des letzten Abschnittes, bei dem es um monotone Operatoren ging, d. h. solche für die $(p \sqsubseteq q) \Rightarrow (F\ p \sqsubseteq F\ q)$). Die Menge der monotonen Prädikatentransformer läßt sich auch syntaktisch charakterisieren.

Definition (*Anweisungen* $\mathcal{A}$). Es sei $\mathcal{A}$ die kleinste Menge von Termen bestehend aus

- $[\beta]$, $\{\beta\}$, $\langle\phi\rangle$,
- $p\ ;\ q$ falls $p, q \in \mathcal{A}$,
- $(\sqcap\ i \in I \bullet p)$, $(\sqcup\ i \in I \bullet p)$ falls $p \in \mathcal{A}$ für alle $i \in I$.

Die Elemente von $\mathcal{A}$ heißen Anweisungen.

Satz (*Darstellung monotoner Prädikatentransformer*). Jede Anweisung ist ein monotoner Prädikatentransformer und jeder monotone Prädikatentransformer läßt sich schreiben als eine Anweisung. Die Menge $\mathcal{A}$ und die Menge der monotonen Prädikatentransformer sind identisch.

Der Beweis dieses Satzes ist in (Back & Wright, 1990) gegeben.

Ein Prädikatentransformer $p : \mathit{Tran}\ S T$ heißt *universell konjunktiv* (oder nur *konjunktiv*) bzw. *universell disjunktiv* (oder nur *disjunktiv*) falls für eine beliebige Menge durch $i \in I$ indizierter Prädikate $\alpha : \mathit{Pred}\ T$ gilt:

$$p\ (\forall\ i \in I \bullet \alpha) \ = \ (\forall\ i \in I \bullet p\ \alpha) \quad \text{bzw.} \quad p\ (\exists\ i \in I \bullet \alpha) \ = \ (\exists\ i \in I \bullet p\ \alpha)$$

So ist *Miracle S T* konjunktiv aber nicht disjunktiv, *Abort S T* ist disjunktiv aber nicht konjunktiv. Die Zustandstransformation $\langle\phi\rangle$ ist sowohl konjunktiv als auch disjunktiv. Die sequentielle Komposition $p\ ;\ q$ ist konjunktiv bzw. disjunktiv falls

sowohl p als auch q konjunktiv bzw. disjunktiv sind. Die folgenden Sätze charakterisieren die konjunktiven und disjunktiven Prädikatentransformer.

Satz (*Darstellung konjunktiver Prädikatentransformer*). Die konjunktiven Prädikatentransformer entsprechen genau den Termen, die aus $[\beta]$, $\langle\phi\rangle$, $p\,;q$, $(\sqcap\, i \in I \bullet p)$ zusammengesetzt sind.

Satz (*Darstellung disjunktiver Prädikatentransformer*). Die disjunktiven Prädikatentransformer entsprechen genau den Termen, die aus $[\beta]$, $\langle\phi\rangle$, $p\,;q$, $(\sqcup\, i \in I \bullet p)$ zusammengesetzt sind.

Die Beweise sind in (Back & Wright, 1992) gegeben. Im allgemeinen gilt für Prädikatentransformer $p : \mathit{Tran}\ R\ S$ und $q : \mathit{Tran}\ S\ T$:

$$
\begin{array}{lcl}
p\,;\mathit{Miracle}\ S\ T & \sqsubseteq & \mathit{Miracle}\ R\ T \\
p\,;\mathit{Abort}\ S\ T & \sqsupseteq & \mathit{Abort}\ R\ T \\
p\,;(\sqcap\, i \in I \bullet q) & \sqsubseteq & (\sqcap\, i \in I \bullet p\,;q) \\
p\,;(\sqcup\, i \in I \bullet q) & \sqsupseteq & (\sqcup\, i \in I \bullet p\,;q)
\end{array}
$$

Ein Beispiel für die Ungleichheit ist $p = \mathit{Abort}\ R\ S$ im ersten, $p = \mathit{Miracle}\ R\ S$ im zweiten, $I = \{\,\}$ im dritten und vierten Fall (weil $(\sqcap\, i \in \{\,\} \bullet p\,;q) = \mathit{Miracle}\ R\ T$ und $(\sqcup\, i \in \{\,\} \bullet p\,;q) = \mathit{Abort}\ R\ T$). Falls cp ein konjunktiver Prädikatentransformer und dp ein disjunktiver Prädikatentransformer ist, gilt zusätzlich:

$$
\begin{array}{lcll}
cp\,;\mathit{Miracle}\ S\ T & = & \mathit{Miracle}\ R\ T & ;\text{–}\mathit{miracle}\text{–}\mathit{rechtsnull} \\
dp\,;\mathit{Abort}\ S\ T & = & \mathit{Abort}\ R\ T & ;\text{–}\mathit{abort}\text{–}\mathit{rechtsnull} \\
cp\,;(\sqcap\, i \in I \bullet q) & = & (\sqcap\, i \in I \bullet cp\,;q) & \sqcap\text{–};\text{–}\mathit{rechtsdistributiv} \\
dp\,;(\sqcup\, i \in I \bullet q) & = & (\sqcup\, i \in I \bullet dp\,;q) & \sqcup\text{–};\text{–}\mathit{rechtsdistributiv}
\end{array}
$$

3.5 Programmvariable

Ein *Zustand* bildet eine Menge von Programmvariablen nach Werten ab. Zustände sind Verbunde. Ein Verbund

$$(v_1 \mapsto e_1, \ldots, v_n \mapsto e_n) : (v_1 : V_1, \ldots, v_n : V_n)$$

entspricht einem Zustand über dem *Zustandsraum* $(v_1 : V_1, \ldots, v_n : V_n)$. *Programmvariablen* werden mit Namen von Verbundfeldern gleichgesetzt. Beispielsweise ist mit $T = (x : Int, y : Int)$

$$(\lambda\, \tau : T \bullet \tau.x + c \geq \tau.y) : Pred\ T$$

ein Zustandsprädikat mit den Programmvariablen (Feldnamen) x und y. Um die übliche Schreibweise von Prädikaten in Annahmen und Zusicherungen zuzulassen, wird folgende Konvention getroffen. Es sei b ein *boolescher Ausdruck* mit freien *Programmvariablen* $w : W$. (Dabei ist w eine Liste von Variablennamen und W eine gleich lange Liste von Typen.) Es sei $T = (w : W)$ und $\tau : T$. Die Funktion $eval\ b\ \tau$ wertet den Ausdruck b im Zustand τ aus:

$$\{b\} \mathrel{\widehat{=}} \{\lambda\, \tau : T \bullet eval\ b\ \tau\} \qquad \textit{Zusicherung mit booleschem Ausdruck}$$
$$[b] \mathrel{\widehat{=}} [\lambda\, \tau : T \bullet eval\ b\ \tau] \qquad \textit{Annahme mit booleschem Ausdruck}$$

Die Funktion $eval\ b\ \tau$ ist definiert über die Struktur des Ausdrucks b. Weil für die Ausdrücke keine feste Syntax zugrunde gelegt wurde, wird die exakte Definition von *eval* offen gelassen. Folgendes Beispiel soll aber zur Illustration dienen. Es sei $b = (x + c \geq y)$ und es seien x, y die Programmvariablen in b. Dann gilt mit $T = (x : Int, y : Int)$:

$$\begin{aligned} & [x + c \geq y] \\ = \quad & \text{«}x, y \text{ Programmvariable»} \\ & [\lambda\, \tau : T \bullet eval\ (x + c \geq y)\ \tau] \\ = \quad & \text{«}eval\text{»} \\ & [\lambda\, \tau : T \bullet \tau.x + c \geq \tau.y] \end{aligned}$$

Zu beachten ist, daß die freien Variablen c, x, y in b die Rolle von Programmvariablen oder die Rolle von *Konstanten* („logischen“ Variablen) haben. Die jeweilige Rolle muß aus dem Kontext hervorgehen. Diese Konvention erlaubt es auch, Verzweigungen und Iterationen in der üblichen Notation zu schreiben, z.B.:

$$if\ x + c \geq y\ then\ q\ else\ r\ end = ([x + c \geq y]\ ;\ q) \sqcap [\neg(x + c \geq y)]\ ;\ r)$$

Mit Hilfe dieser Konvention läßt sich auch die verallgemeinerte (Mehrfach-) Zuweisung definieren. Es seien v, w Listen von Programmvariablen. Die Zuweisung $v \to w := e$ eliminiert die Variablen v und führt gleichzeitig Variablen w mit initialen Werten e ein. Es seien $u, v : U, V$ die freien Programmvariablen in $e : W$ und $S = (u, v : U, V)$ sowie $T = (u, w : U, W)$:

$$v \rightarrow w := e \;\widehat{=}\; \langle \lambda\, \sigma : S \bullet (u, w \mapsto u, eval\ e\ \sigma) \rangle$$
$$v \rightarrow w := e \;:\; Tran\ S\ T \qquad \textit{verallgemeinerte Zuweisung}$$

Dabei sind $u : U$ die Programmvariablen in e, die unverändert bleiben. Dies kann durch folgende Expansion explizit gemacht werden:

$$v \rightarrow w := e \;=\; u, v \rightarrow u, w := u, e \qquad :=\textit{–Expansion}$$

Die normale (Mehrfach-) Zuweisung $v := e$ ergibt sich als ein Spezialfall bei identischen initialen und finalen Variablen:

$$v := e \;\widehat{=}\; v \rightarrow v := e \qquad \textit{Zuweisung}$$

Die indeterministische Zuweisung $v \rightarrow w :[w' \bullet b]$ führt neue Programmvariablen w ein, wählt bösartig Werte für w' aus, so daß b gilt, und eliminiert gleichzeitig Programmvariablen v. Dual dazu wählt $v \rightarrow w :\{w' \bullet b\}$ die neuen Wert für w' gutartig aus:

$$v \rightarrow w :[w' \bullet b] \;\widehat{=}\; (\sqcap\, w' \in W \bullet [b] \,;\, v \rightarrow w := w')$$
verallgemeinerte bösartige Zuweisung
$$v \rightarrow w :\{w' \bullet b\} \;\widehat{=}\; (\sqcup\, w' \in W \bullet \{b\} \,;\, v \rightarrow w := w')$$
verallgemeinerte gutartige Zuweisung

Auch hier können die Programmvariablen v und w teilweise gleich sein. Im Spezialfall von identischen initialen und finalen Variablen führt dies zur normalen bösartigen bzw. gutartigen Zuweisung:

$$v :[v' \bullet b] \;\widehat{=}\; v \rightarrow v :[v' \bullet b] \qquad \textit{bösartige Zuweisung}$$
$$v :\{v' \bullet b\} \;\widehat{=}\; v \rightarrow v :\{v' \bullet b\} \qquad \textit{gutartige Zuweisung}$$

Die bösartige Zuweisung hat eine ähnliche Rolle wie die Spezifikationsanweisung in (Morgan, 1988c). Falls *pre*, *post* boolesche Ausdrücke über die Programmvariablen v sind, entspricht die Spezifikationsanweisung v:[*pre*, *post*] der Anweisung

$$\{pre\} \,;\, v{:}[v' \bullet post[v'/v]] \;.$$

(Falls der Ausdruck *post* initiale und finale Werte in Beziehung setzen soll, werden für die initiale Werte in der Spezifikationsanweisung v:[*pre*, *post*] mit 0 indizierte Variable genommen und für die finalen Werte normale Variablen. In der bösartigen Zuweisung oben beziehen sich normale Variablen auf die initialen Werte und gestrichene Variablen auf die finalen Werte.)

Die verallgemeinerte (deterministische) Zuweisung ist konjunktiv und disjunktiv. Die verallgemeinerte bösartige Zuweisung ist konjunktiv aber nicht disjunktiv. Dual dazu ist die verallgemeinerte gutartige Zuweisung disjunktiv aber nicht konjunktiv. Dies folgt unmittelbar aus den Sätzen über die Darstellung konjunktiver bzw. disjunktiver Prädikatentransformer.

Die deterministische Zuweisung ergibt sich auch als Spezialfall der bösartigen und gutartigen Zuweisung:

$$v :[v' \bullet v' = e] \quad = \quad v := e$$
$$v :\{v' \bullet v' = e\} \quad = \quad v := e$$

Im folgenden werden verschiedene Formen der sequentiellen Komposition mit verallgemeinerten Zuweisungen untersucht.

Satz (*Sequentielle Komposition von Zuweisungen*). Es sei b ein boolescher Ausdruck, e eine Liste von Ausdrücken und u, v, w Listen von Variablen. Dann gilt:

(1) $u \to v := e \,;\, [b] \quad = \quad [b[e/v]] \,;\, u \to v := e$

(2) $u \to v := e \,;\, v \to w := f \quad = \quad u \to w := f[e/v]$

(3) $u \to v := e \,;\, v \to w :[w' \bullet b] \quad = \quad u \to w :[w' \bullet b[e/v]]$

Beweis. (1) und (2) folgen aus *Zustandstransformation–Annahme* und *;–Zustandstransformation*.

Für (3) gilt:

$u \to v := e \,;\, v \to w :[w' \bullet b]$

= «bösartige Zuweisung»

$u \to v := e \,;\, (\sqcap\, w' \bullet [b] \,;\, v \to w := w')$

= «$u \to v := e$ konjunktiv, $\sqcap$–;–rechtsdistributiv»

$(\sqcap\, w' \bullet u \to v := e \,;\, [b] \,;\, v \to w := w')$

= «mit (1)»

$(\sqcap\, w' \bullet [b[e/v]] \,;\, u \to v := e \,;\, v \to w := w')$

= «mit (2)»

$(\sqcap\, w' \bullet [b[e/v]] \,;\, u \to w := w')$

= «bösartige Zuweisung»

$u \to w :[w' \bullet b[e/v]]$

Für den Spezialfall von identischem initialen und finalen Zustandsraum gilt:

$$\begin{aligned} v := e \,;\, [b] &= [b[e/v]] \,;\, v := e \\ v := e \,;\, v := f &= v := f[e/v] \\ v := e \,;\, v :[v' \bullet b] &= v :[v' \bullet b[e/v]] \end{aligned}$$

Satz (*Sequentielle Komposition von bösartigen Zuweisungen mit Annahmen*). Es seien b, c boolesche Ausdrücke und v, w Listen von Programmvariablen. Dann gilt:

(1) $[b] \,;\, v \rightarrow w :[w' \bullet c] = v \rightarrow w :[w' \bullet b \wedge c]$

(2) $v \rightarrow w :[w' \bullet c] \,;\, [b] = v \rightarrow w :[w' \bullet c \wedge b[w'/w]]$

Beweis. Für (1) gilt:

$[b] \,;\, v \rightarrow w :[w' \bullet c]$

= «verallgemeinerte bösartige Zuweisung»

$[b] \,;\, (\sqcap w' \bullet [c] \,;\, v \rightarrow w := w')$

= «$[b]$ konjunktiv, $\sqcap$–;–rechtsdistributiv»

$(\sqcap w' \bullet [b] \,;\, [c] \,;\, v \rightarrow w := w')$

= «;–Annahme»

$(\sqcap w' \bullet [b \wedge c] \,;\, v \rightarrow w := w')$

= «verallgemeinerte bösartige Zuweisung»

$v \rightarrow w :[w' \bullet b \wedge c]$

Für (2) gilt:

$v \rightarrow w :[w' \bullet c] \,;\, [b]$

= «verallgemeinerte bösartige Zuweisung»

$(\sqcap w' \bullet [c] \,;\, v \rightarrow w := w') \,;\, [b]$

= «$\sqcap$–;–linksdistributiv»

$(\sqcap w' \bullet [c] \,;\, v \rightarrow w := w' \,;\, [b])$

= «sequentielle Komposition von Zuweisungen(1)»

$(\sqcap w' \bullet [c] \,;\, b[w'/w]] \,;\, v \rightarrow w := w')$

= «;–Annahme»

$(\sqcap w' \bullet [c \wedge b[w'/w]] \,;\, v \rightarrow w := w')$

= «verallgemeinerte bösartige Zuweisung»

$v \rightarrow w :[w' \bullet c \wedge b[w'/w]]$

Satz (*Sequentielle Komposition von gutartigen Zuweisungen mit Zusicherungen*). Es seien b, c boolesche Ausdrücke und v, w Listen von Programmvariablen. Dann gilt:

(1) $\{b\} ; v \to w :\{w' \bullet c\} \quad = \quad v \to w :\{w' \bullet b \wedge c\}$
(2) $v \to w :\{w' \bullet c\} ; \{b\} \quad = \quad v \to w :\{w' \bullet c \wedge b[w'/w]\}$

Beweis. Aus Dualitätsgründen geht der Beweis analog zum vorangegangenen Beweis.

Satz (*Sequentielle Komposition von bösartigen Zuweisungen*). Es seien b, c boolesche Ausdrücke und u, v, w Listen von Programmvariablen. Dann gilt:

(1) $u \to v :[v' \bullet b] ; v \to w :[w' \bullet c] \quad = \quad u \to w :[w' \bullet (\exists\, v' \bullet b \wedge c[v'/v])]$

Es sei zusätzlich u disjunkt zu v, w. Dann gilt:

(2) $u :[u' \bullet b] ; v \to w :[w' \bullet c] \quad = \quad u, v \to u, w :[u', w' \bullet b \wedge c]$
(3) $v \to w :[w' \bullet c] ; u :[u' \bullet b] \quad = \quad u, v \to u, w :[u', w' \bullet c \wedge b[w'/w]]$

Beweis. Für (1) gilt:

$u \to v :[v' \bullet b] ; v \to w :[w' \bullet c]$
$=$ «verallgemeinerte bösartige Zuweisung»
$u \to v :[v' \bullet b] ; (\sqcap w' \bullet [c] ; v \to w := w')$
$=$ «$u \to v :[v' \bullet b]$ konjunktiv, $\sqcap$–;–rechtsdistributiv»
$(\sqcap w' \bullet u \to v :[v' \bullet b] ; [c] ; v \to w := w')$
$=$ «verallgemeinerte bösartige Zuweisung; $\sqcap$–;–linksdistributiv»
$(\sqcap w' \bullet (\sqcap v' \bullet [b] ; u \to v := v' ; [c] ; v \to w := w'))$
$=$ «sequentielle Komposition von Zuweisungen (1), (2)»
$(\sqcap w' \bullet (\sqcap v' \bullet [b \wedge c[v'/v]] ; u \to w := w'))$
$=$ «$\sqcap$–;–linksdistributiv, $\sqcap$–Annahme»
$(\sqcap w' \bullet [\exists\, v' \bullet b \wedge c[v'/v]] ; u \to w := w')$
$=$ «verallgemeinerte bösartige Zuweisung»
$u \to w :[w' \bullet (\exists\, v' \bullet b \wedge c[v'/v])]$

Die Punkte (2) und (3) folgen als Spezialfälle aus (1) wegen:

$u :[u' \bullet b] \quad = \quad u, v :[u', v' \bullet b \wedge (v' = v)]$
$v \to w :[w' \bullet c] \quad = \quad u, v \to u, w :[u', w' \bullet c \wedge (u' = u)]$

$$u :[u' \bullet b] \quad = \quad u, w :[u', w' \bullet b \wedge (w' = w)]$$

3.6 Einbettung von Prädikatentransformern

Ein Prädikatentransformer $q : Tran\ S\ T$ kann so in einen größeren Zustandsraum eingebettet werden, daß er die zusätzlichen Variablen unverändert läßt. Falls die Variablen (Felder) von R disjunkt sind zu denen von S und T, hat die *Einbettung* $q{\uparrow}R$ den Typ

$$q{\uparrow}R : Tran\ (R \oplus S)\ (R \oplus T) \ .$$

Es wird folgende abkürzende Schreibweise für Variablenbindungen für eine λ-Abstraktion mit zusammengesetztem Zustandsraum eingeführt:

$$(\lambda\ \rho \oplus \sigma : R \oplus S \bullet e) \ \widehat{=}\ (\lambda\ \psi : R \oplus S \bullet e\ [(\psi \mid (dom\ R)/\rho), (\psi \mid (dom\ S)/\sigma)])$$

Es sei $q : Tran\ S\ T$. Die Einbettung ist definiert durch:

$$q{\uparrow}R \ \widehat{=}\ (\lambda\ \gamma : Pred\ (R \oplus T) \bullet (\lambda\ \rho \oplus \sigma : R \oplus S \bullet q\ (\lambda\ \tau : T \bullet \gamma\ (\rho \oplus \tau))\ \sigma))$$

Einbettung

Für die Einbettung elementarer Anweisungen ergeben sich folgende Vereinfachungen:

Satz. Es sei $\beta : Pred\ T$, es sei $\gamma : Pred\ (R \oplus T)$ und es sei $\phi : S \to T$. Dann gilt:

(1) $([\beta] \uparrow R)\ \gamma \ = \ (\lambda\ \rho \oplus \tau : R \oplus T \bullet \beta\ \tau \Rightarrow \gamma\ (\rho \oplus \tau))$

Einbettung von Annahmen

(2) $(\{\beta\} \uparrow R)\ \gamma \ = \ (\lambda\ \rho \oplus \tau : R \oplus T \bullet \beta\ \tau \wedge \gamma\ (\rho \oplus \tau))$

Einbettung von Zusicherungen

(3) $(\langle \phi \rangle \uparrow R)\ \gamma \ = \ (\lambda\ \rho \oplus \sigma : R \oplus S \bullet \gamma\ (\rho \oplus \phi\ \sigma))$

Einbettung von Zustandstransformationen

Beweis. Für (1) gilt:

$$\begin{array}{ll} & ([\beta] \uparrow R)\ \gamma \\ = & \text{«Einbettung»} \\ & (\lambda\ \rho \oplus \tau : R \oplus T \bullet [\beta]\ (\lambda\ \tau : T \bullet \gamma\ (\rho \oplus \tau))\ \tau) \\ = & \text{«Annahme»} \\ & (\lambda\ \rho \oplus \tau : R \oplus T \bullet (\beta \Rightarrow (\lambda\ \tau : T \bullet \gamma\ (\rho \oplus \tau)))\ \tau) \end{array}$$

$=$ «Implikation, Reduktion»
$(\lambda\, \rho \oplus \tau : R \oplus T \bullet \beta\, \tau \Rightarrow \gamma\, (\rho \oplus \tau))$

Der Beweis von (2) verläuft analog. Für (3) gilt:

$(\langle \phi \rangle \uparrow R)\ \gamma$
$=$ «Einbettung»
$(\lambda\, \rho \oplus \sigma : R \oplus S \bullet\ \langle \phi \rangle\, (\lambda\, \tau : T \bullet \gamma\, (\rho \oplus \tau))\ \sigma)$
$=$ «Zustandstransformation»
$(\lambda\, \rho \oplus \sigma : R \oplus S \bullet\ (\lambda\, \sigma : S \bullet (\lambda\, \tau : T \bullet \gamma\, (\rho \oplus \tau))\ (\phi\, \sigma))\ \sigma)$
$=$ «2× Reduktion»
$(\lambda\, \rho \oplus \sigma : R \oplus S \bullet \gamma\, (\rho \oplus \phi\, \sigma))$

Satz (*Einbettung von Zuweisungen*). Es seien die drei Zuweisungen $v \rightarrow w := e$, $v \rightarrow w :[w' \bullet c]$ und $v \rightarrow w :\{w' \bullet c\}$ vom Typ $Tran\ S\ T$ und es sei $R = (u : U)$. Dann gilt:

(1) $v \rightarrow w := e \uparrow R \quad = \quad u, v \rightarrow u, w := u, e$
(2) $v \rightarrow w :[w' \bullet c] \uparrow R \quad = \quad u, v \rightarrow u, w :[u', w' \bullet (u' = u) \wedge c]$
(3) $v \rightarrow w :\{w' \bullet c\} \uparrow R \quad = \quad u, v \rightarrow u, w :\{u', w' \bullet (u' = u) \wedge c\}$

Beweis. (1) bis (3) folgen aus dem vorangegangenem Satz und den Definitionen der Zuweisungen.

Einbettungen werden implizit verwendet, um die Komposition von Anweisungen mit unterschiedlichen Zustandsräumen zuzulassen. Es seien beispielsweise x, y, z Programmvariablen vom Typ *Int*. Dann gilt:

$x := y \quad : \quad Tran\ (x, y : Int, Int)\ (x, y : Int, Int)$
$z := x \quad : \quad Tran\ (x, z : Int, Int)\ (x, z : Int, Int)$

Die sequentielle Komposition $p\ ;\ q$ erfordert, daß der finale Zustandsraum von p identisch ist mit dem initialen Zustandsraum von q. Die bösartige Auswahl von $q \sqcap r$ bzw. die gutartige Auswahl $q \sqcup r$ erfordern ebenfalls identische Zustandsräume. Mit impliziten Einbettungen gilt aber:

$x := y\ ;\ z := x$
$=$ «:=–Expansion»
$x, y := y, y\ ;\ x, z := x, x$

$=$ «implizite Einbettung»
$(x, y := y, y) \uparrow (z : Int)\ ;\ (x, z := x, x) \uparrow (y : Int)$
$=$ «Einbettung von Zuweisungen (1)»
$x, y, z := y, y, z \quad ; \quad x, y, z := x, y, x$
$=$ «Sequentielle Komposition von Zuweisungen (2)»
$x, y, z := y, y, y$

Das Beispiel verdeutlicht das Zusammenspiel von Programmvariablen in Ausdrücken mit der impliziten Einbettung: Mit der Konvention des letzten Abschnittes werden den Zuweisungen *minimale Zustandsräume* zugewiesen; die implizite Einbettung läßt trotzdem ihre Komposition zu.

Die implizite Einbettung läßt die übliche Schreibweise für *miracle*, *abort* und *skip* ohne Parametrisierung durch den Zustandsraum zu: der minimale Zustandsraum für diese Anweisungen ist der leere Zustandsraum, d. h. der Verbundtyp ():

miracle	$\widehat{=}$	*Miracle* () ()	*wundersame Anweisung*
abort	$\widehat{=}$	*Abort* () ()	*chaotische Anweisung*
skip	$\widehat{=}$	*Skip* ()	*leere Anweisung*

Für indeterministische Zuweisungen gilt beispielsweise:

$$
\begin{array}{lcl}
v :[v' \bullet \mathit{false}] & = & \mathit{miracle} \\
v :\{v' \bullet \mathit{false}\} & = & \mathit{abort} \\
v :[v' \bullet v' = v] & = & \mathit{skip} \\
v :\{v' \bullet v' = v\} & = & \mathit{skip}
\end{array}
$$

3.7 Kommutativität von Prädikatentransformern

Zwei Prädikatentransformer $p : Tran\ Q\ R$ und $q : Tran\ S\ T$ heißen *unabhängig*, falls die Felder von Q und R alle verschieden sind sowohl von S als auch von T. Für unabhängige Prädikatentransformer p, q wird im folgenden die Beziehung von $p\ ;\ q$ und $q\ ;\ p$ untersucht.

Satz. Es seien $[\beta]$, $\{\beta\}$ und $\langle\phi\rangle$ unabhängig vom Prädikatentransformer p. Dann gilt:

(1) $\quad p\ ;\ [\beta] \ \sqsubseteq\ [\beta]\ ;\ p$ *Annahme–kommutativ*

(2) $p\,;\{\beta\} \sqsupseteq \{\beta\}\,;p$ *Zusicherung–kommutativ*

(3) $p\,;\langle\phi\rangle = \langle\phi\rangle\,;p$ *Zustandstransformation–kommutativ*

Beweis. Es sei $\beta : Pred\ R$, $\phi : Q \to R$ und $p : Tran\ S\ T$. Es sei γ ein beliebiges Zustandsprädikat vom Typ $Pred\ (R \oplus T)$. Für die linke Seite von (1) gilt:

$(p\,;[\beta])\,\gamma$
$=$ «implizite Einbettung»
$(p \uparrow R\,;[\beta] \uparrow T)\,\gamma$
$=$ «sequentielle Komposition, Einbettung von Annahmen»
$(p \uparrow R)\,(\lambda\,\rho \oplus \tau : R \oplus T \bullet \beta\,\rho \Rightarrow \gamma\,(\rho \oplus \tau))$
$=$ «Einbettung, Reduktion»
$(\lambda\,\rho \oplus \sigma : R \oplus S \bullet p\,(\lambda\,\tau : T \bullet \beta\,\rho \Rightarrow \gamma\,(\rho \oplus \tau))\,\sigma)$

Für die rechte Seite von (1) gilt:

$([\beta]\,;p)\,\gamma$
$=$ «implizite Einbettung»
$([\beta] \uparrow S\,;p \uparrow R)\,\gamma$
$=$ «sequentielle Komposition, Einbettung»
$([\beta] \uparrow S)\,(\lambda\,\rho \oplus \sigma : R \oplus S \bullet p\,(\lambda\,\tau : T \bullet \gamma\,(\rho \oplus \tau))\,\sigma)$
$=$ «Einbettung von Annahmen, Reduktion»
$(\lambda\,\rho \oplus \sigma : R \oplus S \bullet \beta\,\rho \Rightarrow p\,(\lambda\,\tau : T \bullet \gamma\,(\rho \oplus \tau))\,\sigma)$
$\leq$ «Fallunterscheidung mit $\beta\,\rho = true$ und $\beta\,\rho = false$»
$(\lambda\,\rho \oplus \sigma : R \oplus S \bullet p\,(\lambda\,\tau : T \bullet \beta\,\rho \Rightarrow \gamma\,(\rho \oplus \tau))\,\sigma)$

Die Beweise von (2) und (3) verlaufen ähnlich.

Die Verallgemeinerung dieses Satzes lautet:

Satz. Es sei cq ein konjunktiver und dq ein disjunkter Prädikatentransformer. Falls p ein von cq und dq unabhängiger Prädikatentransformer ist, gilt:

$p\,;cq \sqsubseteq cq\,;p$ *konjunktiv–kommutativ*

$p\,;dq \sqsupseteq dq\,;p$ *disjunktiv–kommutativ*

Beweis. Jeder konjunktive (disjunktive) Prädikatentransformer läßt sich schreiben als eine Kombination von Zustandstransformationen, Annahmen, sequentieller Komposition und bösartiger Auswahl (Zustandstransformationen, Zusiche-

rungen, sequentieller Komposition und gutartiger Auswahl). Der Beweis erfolgt dann mit Induktion über den Aufbau von cq (dq). Weil der Beweis lang aber einfach ist, werden die Details hier weggelassen.

3.8 Variablen- und Prozedurdeklaration

Es sei v eine Liste von Programmvariablen und es sei e eine Liste von Ausdrücken. Die *Variableneinführung* $enter\ v := e$ erweitert den Zustandsraum um v und initialisiert diese mit e. Die Variablenelimination $exit\ v$ entfernt die Variablen v aus dem Zustandsraum. Beide sind Spezialfälle der verallgemeinerten Zuweisung. Es sei ε die leere Liste:

$$enter\ v := e \quad \widehat{=} \quad \varepsilon \rightarrow v := e \qquad \textit{Variableneinführung}$$
$$exit\ v \quad \widehat{=} \quad v \rightarrow \varepsilon := \varepsilon \qquad \textit{Variablenelimination}$$

Bei der sequentiellen Komposition mit $exit\ v$ kann die vorangehende Anweisung gelöscht werden, falls es nur die Variablen v ändert.

Satz (*Sequentielle Komposition mit exit*). Es sei $cp : Tran\ (v : V)\ (v : V)$ bzw. $dp : Tran\ (v : V)\ (v : V)$ ein konjunktiver bzw. disjunktiver Prädikatentransformer. Dann gilt:

(1) $exit\ v \ \sqsubseteq \ cp\ ;\ exit\ v$
(2) $dp\ ;\ exit\ v \ \sqsubseteq \ exit\ v$

Beweis. Der Beweis erfolgt analog zum Beweis von *konjunktiv–kommutativ* bzw. *disjunktiv–kommutativ* durch Induktion über den Aufbau von cp (dp).

Die lokale *Variablendeklaration* ($var\ v := e \bullet p$) erweitert den Zustandsraum um Programmvariablen v mit initialen Werten e, führt p aus und eliminiert wieder v:

$$(var\ v := e \bullet p) \quad \widehat{=} \quad enter\ v := e\ ;\ p\ ;\ exit\ v \qquad \textit{Variablendeklaration}$$

Eine *Prozedur* ist eine Anweisung mit Wert- und Resultatparameter. Die Parameterübergabe erfolgt durch Vergrößern bzw. Verkleinern des Zustandsraumes: Falls n deklariert ist durch ($proc\ n\ (val : V)\ res : R = r \bullet q$), werden bei einem

Prozeduraufruf $z := n\ (e)$ die aktuellen Wertparameter e nach val kopiert, der *Rumpf* r aufgerufen und die Resultate *res* nach z kopiert:

$$(proc\ n\ (val : V)\ res : R = r \bullet q) \ \widehat{=}\ q[enter\ res : R\ ;\ r\ ;\ exit\ val/n] \qquad \textit{Prozedurdeklaration}$$

Es sei p vom Typ $Tran\ (val : V, w : W)\ (res : R, w : W)$:

$$z := p\ (e) \ \widehat{=}\ enter\ val := e\ ;\ p\ ;\ res := z\ ;\ exit\ res \qquad \textit{Prozeduraufruf}$$

Der Rumpf r kann die Resultatparameter, die lokalen Kopien der Wertparameter und die globalen Variablen modifizieren. Die Resultatparameter haben initial einen unbekannten Wert. Der Ausdruck $enter\ res : R$ in der Definition der Prozedurdeklaration ist definiert durch

$$enter\ v : V \ \widehat{=}\ enter\ v := v0 \quad \text{wobei} \quad v0 \in V \quad \text{beliebig} \ .$$

(Mit dieser Definition wird die Notwendigkeit von indeterministischen Initialisierungen oder Initialisierungen mit undefiniertem Wert $\perp$ vermieden.)

Mit der obigen Definition der Prozedurdeklaration wird ein Prozedurname an jeder Stelle, an der die Prozedur aufgerufen wird, durch *einen* Ausdruck ersetzt, unabhängig davon welche Programmvariablen an der Aufrufstelle existieren. Der Ausdruck muß dazu in den jeweiligen Kontext *eingebettet* werden.

Prozeduren sind hier nicht rekursiv, ließen sich aber einfach um Rekursion erweitern (Back, 1987; Morgan, 1988b). In der Tat ist es so, daß üblicherweise mit Prozeduren drei unabhängige Dimensionen verbunden sind: die Bennenung einer Anweisung, die Parameterübergabe und die Rekursion. Im Hinblick auf die Definition von Methoden sind hier nur die ersten zwei relevant, weil sich eine rekursive Prozedur (bzw. Methode) als eine nicht-rekursive mit lokaler Rekursion definieren läßt (Morgan, 1988b).

3.9 Prozedurale Verfeinerung und Datenverfeinerung

Die Grundidee des Verfeinerungskalküls ist die Herleitung von effizienten Programmen aus kompakten, abstrakten Spezifikationen. *Abstraktion* in der Spezifikation wird auf unterschiedliche Weise erreicht:

- durch die implizite, indeterministische Spezifikation von Resultaten (z. B. mittels Prädikate in indeterministischen Zuweisungen);
- durch Verwendung von abstrakten Datentypen wie Mengen, Relationen usw. in Programmen.

Der Verfeinerungskalkül ist eine „Breitbandsprache“, in der abstrakte Anweisungen und Datentypen mit ausführbaren kombiniert werden können. Was genau ausführbar ist wird – bewußt – offen gelassen. Dies hängt von der Programmiersprache ab, die beispielsweise keine Mengen kennt, nur eingeschränkte Mengen zuläßt (wie PASCAL) oder beliebige Mengen. Die Theorie bleibt davon unberührt. Die Begriffe konkret und abstrakt sind deshalb *relativ*.

Die Transformation von abstrakten Anweisungen in konkrete wird *prozedurale Verfeinerung* genannt, die Transformation von abstrakten Datenstrukturen in konkretere die *Datenverfeinerung*. Typische Regeln, für diese zwei Arten der Verfeinerung werden im folgenden erläutert.

Satz (*Verfeinerung von Zuweisungen*). Es seien b, c boolesche Ausdrücke, v, w Listen von Programmvariablen und e eine Liste von Ausdrücken. Dann gilt:

(1) $v \rightarrow w :[w' \bullet b] \sqsubseteq v \rightarrow w :[w' \bullet c]$ falls $b \Leftarrow c$ gilt
(2) $v \rightarrow w :\{w' \bullet b\} \sqsubseteq v \rightarrow w :\{w' \bullet c\}$ falls $b \Rightarrow c$ gilt
(3) $v \rightarrow w :[w' \bullet b] \sqsubseteq v \rightarrow w := e$ falls $b[e/w']$ gilt

Beweis. Die Punkte (1) und (2) folgen direkt aus der Definition der verallgemeinerten Zuweisung und $\sqsubseteq$–*Annahmen* bzw. $\sqsubseteq$–*Zusicherungen*. Punkt (3) folgt aus (1).

Für den häufigen Spezialfall von identischem initialen und finalen Zustandsraum gilt offensichtlich:

$v :[v' \bullet b] \sqsubseteq v :[v' \bullet c]$ falls $b \Leftarrow c$ gilt

$v :\{v' \bullet b\} \quad \sqsubseteq \quad v :\{v' \bullet c\} \qquad$ falls $b \Rightarrow c$ gilt
$v :[v' \bullet b] \quad \sqsubseteq \quad v := e \qquad$ falls $b[e/v']$ gilt

Eine Art die bösartige Zuweisung zu konkretisieren ist die Verzweigung.

Satz (*bösartige Zuweisung–if*). Es sei v eine Liste von Programmvariablen und es seien $b, c, c0, c1$ boolesche Ausdrücke. Dann gilt:

$v :[v' \bullet c] \;=\; \mathit{if}\ b\ \mathit{then}\ v :[v' \bullet c0]\ \mathit{else}\ v :[v' \bullet c1]\ \mathit{end}$
falls $b \Rightarrow (c \Leftrightarrow c0)$ und $\neg b \Rightarrow (c \Leftrightarrow c1)$ gilt

Beweis. Der Beweis erfolgt durch Auflösen der Verzweigung auf der rechten Seite und Anwendung von *Sequentielle Komposition von bösartigen Zuweisungen mit Annahmen* und ⊓⊢*Annahme*.

Beispiel (*Maximum zweier Zahlen*). Das Maximum *max* zweier Zahlen a, b hat die Eigenschaft

$$((max = a) \vee (max = b)) \wedge (a \leq max) \wedge (b \leq max) \ .$$

Ein Programm, das das Maximum von a und b der Programmvariablen x zuweist, wird wie folgt hergeleitet.

$x :[x' \bullet ((x' = a) \vee (x' = b)) \wedge (a \leq x') \wedge (b \leq x')]$
$=$ «bösartige Zuweisung–if (*)»
$\mathit{if}\ a \leq b\ \mathit{then}\ x :[x' \bullet x' = b]\ \mathit{else}\ x :[x' \bullet x' = a]\ \mathit{end}$
$=$ «Vereinfachung von bösartigen Zuweisungen»
$\mathit{if}\ a \leq b\ \mathit{then}\ x := b\ \mathit{else}\ x := a\ \mathit{end}$

Auch wenn in diesem Beispiel das Ergebnis eindeutig bestimmt ist, läßt es sich einfacher durch eine implizite, indeterministische Zuweisung spezifizieren. Der Schritt (*) beinhaltet die Entwurfsentscheidung, eine Verzweigung mit der Bedingung $a \leq b$ einzusetzen. Für die beiden Zweige entstehen für die booleschen Ausdrücke $c0$ bzw. $c1$ in den Zuweisungen die Beweisverpflichtungen

$a \leq b \Rightarrow (((x' = a) \vee (x' = b)) \wedge (a \leq x') \wedge (b \leq x') \Leftrightarrow c0)$ bzw.
$a > b \Rightarrow (((x' = a) \vee (x' = b)) \wedge (a \leq x') \wedge (b \leq x') \Leftrightarrow c1)$,

aus denen sich ableiten läßt, daß

$$c0 \;\Leftrightarrow\; (x' = b) \text{ bzw.}$$
$$c1 \;\Leftrightarrow\; (x' = a) \;.$$

In der ursprünglichen Technik der Datenverfeinerung (Hoare, 1972) wird der konkrete und abstrakte Zustandsraum durch eine *Abstraktionsfunktion* in Beziehung gesetzt. Hierbei kann der konkrete Zustandsraum zusätzlich durch eine *Invariante* eingeschränkt sein. In neueren Arbeiten (Back, 1989; Gardiner & Morgan, 1993; He, et al., 1986; Morgan, 1988a; Morris, 1989) wird statt einer partiellen Abstraktionsfunktion verallgemeinernd eine Relation genommen. Zentral hierfür ist die Notation

$$p \sqsubseteq_{rel} q \;.$$

Sie besagt, daß p durch q so verfeinert wird, daß die initialen und finalen Zustandsräume von p und q jeweils in Beziehung *rel* stehen. Beispielsweise gilt

$$i := i + 1 \sqsubseteq_{rel} b := \neg b \qquad \text{falls} \qquad rel\ (i, b) \;=\; (b = odd\ i) \;.$$

Weil Anweisungen mit unterschiedlichem initialem und finalem Zustandsraum zugelassen werden, läßt sich Datenverfeinerung als eine spezielle Art der prozeduralen Verfeinerung ausdrücken (Back & Wright, 1989). Dazu wird eine Anweisung definiert (*Simulation* oder *Kodierung* genannt), die entsprechend der Relation *rel* den abstrakten Zustandsraum in den konkreten Zustandsraum abbildet. Alternativ dazu wird eine Anweisung definiert (*Cosimulation* oder *Dekodierung* genannt), die entsprechend der Relation *rel* den konkreten in den abstrakten Zustandsraum abbildet.

Definition (*Prädikatentransformer über einem Zustandsraum*). Ein Prädikatentransformer p *ist über dem Zustandsraum* S falls initialer und finaler Zustandsraum von p identisch sind und S Teil von diesen ist. D. h. p ist vom Typ:

$$p : Tran\ (R \oplus S)\ (R \oplus S) \;.$$

Definition (*Datenverfeinerung mittels Relation,* $\sqsubseteq_{rel}$). Es sei p ein Prädikatentransformer über $(v : V)$ und q ein Prädikatentransformer über $(w : W)$. Es sei $rel : V \times W \to Bool$ und es seien *sim* und *cos* gegeben durch:

$$sim \;=\; v \to w :[w' \bullet rel\ (v, w')] \quad \text{bzw.} \quad cos \;=\; w \to v :\{v' \bullet rel\ (v', w)\}$$

Prädikatentransformer p mit Variablen v wird verfeinert durch q mit Variablen w mittels rel ist definiert durch eine der folgenden äquivalenten Ungleichungen:

(1) $p \,;\, sim \sqsubseteq sim \,;\, q$
(2) $cos \,;\, p \,;\, sim \sqsubseteq q$
(3) $cos \,;\, p \sqsubseteq q \,;\, cos$
(4) $p \sqsubseteq sim \,;\, q \,;\, cos$

Falls v und w aus dem Kontext hervorgehen, wird dies abkürzend geschrieben als $p \sqsubseteq_{rel} q$.

Zu beachten ist, daß die *Simulation sim* konjunktiv und die *Cosimulation cos* disjunktiv ist. Die Äquivalenz der vier Definitionen ergibt sich daraus, daß *sim* und *cos Semiinverse* sind (Back, 1993) in dem Sinn, daß

$$cos \,;\, sim \;\sqsubseteq\; skip \qquad \text{und} \qquad skip \;\sqsubseteq\; sim \,;\, cos \;.$$

Der Beweis von (1) $\Rightarrow$ (2) lautet:

$p \,;\, sim \;\sqsubseteq sim \,;\, q$
$\Rightarrow$ « ;–monoton»
$cos \,;\, p \,;\, sim \sqsubseteq cos \,;\, sim \,;\, q$
$\Rightarrow$ «wegen $cos \,;\, sim \sqsubseteq skip$, Transitivität»
$cos \,;\, p \,;\, sim \sqsubseteq q$

Der Beweis läßt sich analog zyklisch fortsetzen, (2) $\Rightarrow$ (3), (3) $\Rightarrow$ (4), (4) $\Rightarrow$ (1), wodurch die Äquivalenz gezeigt ist.

Verfeinerung mittels Relation ermöglicht das Ersetzen von lokalen Variablen im folgenden Sinne:

Satz (*Variablenverfeinerung mittels Relation*). Es sei p ein Prädikatentransformer über $(v : V)$ und q über $(w : W)$ und es sei $rel : V \times W \to Bool$. Dann gilt:

$$(var\ v := i \bullet p) \sqsubseteq (var\ w := j \bullet q) \quad \text{falls} \quad rel\ (i, j) \quad \text{und} \quad p \sqsubseteq_{rel} q$$

Beweis. Der Beweis verläuft nach demselben Muster wie in (Back, 1993):

$(var\ v := i \bullet p)$
$=$ «*var*»

$\quad$ *enter* $v := i$; p ; *exit* v

$\sqsubseteq$ «mit $sim = v \rightarrow w$:[$w' \bullet rel\ (v, w')$], Sequentielle Komposition mit exit»

$\quad$ *enter* $v := i$; p ; sim ; *exit* v

$\sqsubseteq$ «wegen $p \sqsubseteq_{rel} q$»

$\quad$ *enter* $v := i$; sim ; q ; *exit* v

$\sqsubseteq$ «Seq. Komp. von Zuweisungen (3), $rel\ (i, j)$, Verf. von Zuweisungen (2)»

$\quad$ *enter* $w := j$; q ; *exit* v

$=$ «*var*»

$\quad$ $(var\ w := j \bullet q)$

Eine wesentliche Eigenschaft der Datenverfeinerung ist ihre Distributivität bezüglich der Operatoren von $\mathcal{A}$. Das bedeutet, daß sich Datenverfeinerungen *kompositional* berechnen lassen:

Satz. Es sei p ein Prädikatentransformer über $(v : V)$ und q ein Prädikatentransformer über $(w : W)$ und es sei $rel : V \times W \rightarrow Bool$. Dann gilt:

(1) $p\ ;\ p' \sqsubseteq_{rel} q\ ;\ q'$ falls $p \sqsubseteq_{rel} q$ und $p' \sqsubseteq_{rel} q'$ $\quad \sqsubseteq_{rel}$–;–*subdistributiv*

(2) $(\sqcap i \in I \bullet p) \sqsubseteq_{rel} (\sqcap i \in I \bullet q)$

falls $p \sqsubseteq_{rel} q$ für alle $i \in I$ $\quad \sqsubseteq_{rel}$–$\sqcap$–*subdistributiv*

(3) $(\sqcup i \in I \bullet p) \sqsubseteq_{rel} (\sqcup i \in I \bullet q)$

falls $p \sqsubseteq_{rel} q$ für alle $i \in I$ $\quad \sqsubseteq_{rel}$–$\sqcup$–*subdistributiv*

Beweis. Es sei $sim = v \rightarrow w$:[$w' \bullet r\ (v, w')$]. Nach der Variante (1) der Definition von $\sqsubseteq_{rel}$ gilt für (1):

$\quad$ $p\ ;\ p'\ ;\ sim$

$\sqsubseteq$ «wegen $p' \sqsubseteq_{rel} q'$»

$\quad$ $p\ ;\ sim\ ;\ q'$

$\sqsubseteq$ «wegen $p \sqsubseteq_{rel} q$»

$\quad$ $sim\ ;\ q\ ;\ q'$

Für (2) gilt:

$\quad$ $(\sqcap\ i \in I \bullet p)\ ;\ sim$

$=$ «$\sqcap$–;–linksdistributiv»

$\quad$ $(\sqcap\ i \in I \bullet p\ ;\ sim)$

$\sqsubseteq$ «wegen $p \sqsubseteq_{rel} q$ für alle $i \in I$»

$\quad$ $(\sqcap\ i \in I \bullet sim\ ;\ q)$

$=$ «*sim* konjunktiv, $\sqcap$–;–rechtsdistributiv»
$sim\ ;\ (\sqcap\ i \in I \bullet q)$

Der Beweis von (3) verläuft analog.

Eine Anweisung, die unabhängig von den zu verfeinernden Variablen ist, bleibt von einer Datenverfeinerung unberührt.

Satz. Es sei $S = (v : V)$, $T = (w : W)$ und es sei $rel : V \times W \to Bool$. Falls r ein von S und T unabhängiger Prädikatentransformer ist, gilt:

$$r \sqsubseteq_{rel} r \qquad \sqsubseteq_{rel}\text{–}unabhängig$$

Beweis. Es sei $sim = v \to w\ :[w' \bullet rel\ (v, w')]$. Nach der Variante (1) der Definition von $\sqsubseteq_{rel}$ gilt:

$r\ ;\ sim$
$\sqsubseteq$ «konjunktiv–kommutativ»
$sim\ ;\ r$

Mit den letzten beiden Sätzen läßt sich eine Anweisung über abstrakten Variablen in eine strukturgleiche Anweisung so weit transformieren, daß nur noch die eigentlichen Operationen auf den Variablen zu verfeinern sind. Sind die abstrakte Operation, die konkrete Operation und die Relation gegeben, kann die Datenverfeinerung überprüft werden. Alternativ dazu kann die konkrete Operation aus der abstrakten Operation und der Relation abgeleitet werden. Für deterministische und indeterministische Zuweisungen an abstrakte und globale Variablen geht dies wie folgt.

Satz (*Datenverfeinerung von Zuweisungen mittels Relation*). Es seien $u : U$, $v : V$, $w : W$ Listen von Programmvariablen, es seien e, f Listen von Ausdrücken, es seien b, c boolesche Ausdrücke und es sei $rel : V \times W \to Bool$. Dann gilt:

(1) $v\ :[v' \bullet b] \sqsubseteq_{rel} w\ :[w' \bullet (\forall\ v \bullet rel\ (v, w) \Rightarrow (\exists\ v' \bullet b \wedge rel\ (v', w')))]$
(2) $v := e \sqsubseteq_{rel} w\ :[w' \bullet (\forall\ v \bullet rel\ (v, w) \Rightarrow rel\ (e, w'))]$
(3) $u\ :[u' \bullet c] \sqsubseteq_{rel} u\ :[u' \bullet (\forall\ v \bullet rel\ (v, w) \Rightarrow c)]$
(4) $u := f \sqsubseteq_{rel} u\ :[u' \bullet (\forall\ v \bullet rel\ (v, w) \Rightarrow (u' = f))]$

Beweis. Es sei $sim = v \rightarrow w :[w' \bullet rel\ (v, w')]$. Nach Variante (1) der Definition von $\sqsubseteq_{rel}$ gilt für (1):

$sim\ ;\ w :[w' \bullet (\forall\ v \bullet rel\ (v, w) \Rightarrow (\exists\ v' \bullet b \wedge rel\ (v', w')))]$

$=$ «*sim*, Sequentielle Komposition von bösartigen Zuweisungen (1)»

$v \rightarrow w :[w' \bullet (\exists\ w \bullet rel\ (v, w) \wedge\ (\forall\ v \bullet rel\ (v, w) \Rightarrow (\exists\ v' \bullet b \wedge rel\ (v', w'))))]$

$\sqsupseteq$ «Verfeinerung von Zuweisungen»

$v \rightarrow w :[w' \bullet (\exists\ v' \bullet b \wedge rel\ (v', w'))]$

$=$ «Sequentielle Komposition von bösartigen Zuweisungen (1), *sim*»

$v :[v' \bullet b]\ ;\ sim$

(2) folgt direkt aus (1) wegen $v := e\ \ =\ \ v :[v' \bullet v' = e]$. Für (3) gilt:

$sim\ ;\ u :[u' \bullet (\forall\ v \bullet rel\ (\ v, w') \Rightarrow c)]$

$=$ «*sim*, Sequentielle Komposition von bösartigen Zuweisungen (3)»

$u, v \rightarrow u, w :[u', w' \bullet rel\ (u, w') \wedge (\forall\ v \bullet rel\ (v, w') \Rightarrow c)]$

$\sqsupseteq$ «Verfeinerung von Zuweisungen (1)»

$u, v \rightarrow u, w :[u', w' \bullet rel\ (u, w') \wedge c]$

$=$ «Sequentielle Komposition von bösartigen Zuweisungen (2), *sim*»

$u :[u' \bullet c]\ ;\ v \rightarrow w :[w' \bullet rel\ (v, w')]$

(4) folgt direkt aus (3) wegen $(u := f)\ =\ u :[u' \bullet u' = f]$.

Für Annahmen, Zuweisungen, Verzweigungen und Iterationen ergeben sich folgende Datenverfeinerungsregeln.

Satz. Es seien $R = (u : U)$, $S = (v : V)$, $T = (w : W)$ und $rel : V \times W \rightarrow Bool$. Es seien b, c boolesche Ausdrücke. Falls p, p' vom Typ $Tran\ (R \oplus S)\ (R \oplus S)$ und q, q' vom Typ $Tran\ (R \oplus T)\ (R \oplus T)$, gilt:

(1) $[b] \sqsubseteq_{rel} [\forall\ v \bullet rel\ (v, w) \Rightarrow b]$ — $\sqsubseteq_{rel}$*–Annahme*

(2) $\{b\} \sqsubseteq_{rel} \{\exists\ v \bullet rel\ (v, w) \wedge b\}$ — $\sqsubseteq_{rel}$*–Zusicherung*

(3) *if b then p else p' end* $\sqsubseteq_{rel}$ *if c then q else q' end* — $\sqsubseteq_{rel}$*–if*

falls $p \sqsubseteq_{rel} q$ und $p' \sqsubseteq_{rel} q'$

und $(\exists\ v \bullet rel\ (v, w) \wedge b) \leq c \leq (\forall\ v \bullet rel\ (v, w) \Rightarrow b)$

(4) *while b do p end* $\sqsubseteq_{rel}$ *while c do q end* — $\sqsubseteq_{rel}$*–while*

falls $p \sqsubseteq_{rel} q$

und $(\exists\ v \bullet rel\ (v, w) \wedge b) \leq c \leq (\forall\ v \bullet rel\ (v, w) \Rightarrow b)$

Beweis. Es sei

$$sim = v \rightarrow w :[w' \bullet rel\ (v, w')] \quad \text{und} \quad cos = w \rightarrow v :\{v' \bullet rel\ (v', w)\}\ .$$

Dann folgt (1) aus der Datenverfeinerung von Zuweisungen (1) wegen

$$[b] = v :[v' \bullet b \wedge (v' = v)]\ .$$

Für (2) gilt nach der Variante (3) der Definition von $\sqsubseteq_{rel}$:

$\{\exists\ v \bullet rel\ (v, w) \wedge b\}\ ;\ cos$
$=$ «*cos*, Sequentielle Komposition mit Zusicherungen (1)»
$w \rightarrow v :\{v' \bullet (\exists\ v \bullet rel\ (v, w) \wedge b) \wedge rel\ (v', w)\}$
$\sqsupseteq$ «Logik, Verfeinerung von Zuweisungen (2)»
$w \rightarrow v :\{v' \bullet rel\ (v', w) \wedge b\ [v'/v]\}$
$=$ «Sequentielle Komposition mit Zusicherungen (2), *cos*»
$cos\ ;\ \{b\}$

Für (3) gilt:

if b then p else p' end $\sqsubseteq$ *if c then q else q' end*
$\Leftrightarrow$ «Verzweigung»
$([b]\ ;\ p) \sqcap ([\neg b]\ ;\ p') \sqsubseteq ([c]\ ;\ q) \sqcap ([\neg c]\ ;\ q)$
$\Leftarrow$ «Voraussetzung: $p \sqsubseteq_{rel} q$ und $p' \sqsubseteq_{rel} q'$, Monotonie»
$([b] \sqsubseteq_{rel} [c]) \wedge ([\neg b] \sqsubseteq_{rel} [\neg c])$
$\Leftarrow$ «$\sqsubseteq_{rel}$–Annahme»
$([(\forall\ v \bullet rel\ (v, w) \Rightarrow b)] \sqsubseteq [c]) \wedge ([(\forall\ v \bullet rel\ (v, w) \Rightarrow \neg b)] \sqsubseteq [\neg c])$
$\Leftrightarrow$ «$\sqsubseteq$–Annahme»
$(c \leq (\forall\ v \bullet rel\ (v, w) \Rightarrow b)) \wedge (\neg c \leq (\forall\ v \bullet rel\ (v, w) \Rightarrow \neg b))$
$\Leftrightarrow$ «Logik»
$(c \leq (\forall\ v \bullet rel\ (v, w) \Rightarrow b)) \wedge ((\exists\ v \bullet rel\ (v, w) \wedge b) \leq c)$

Die letzte Zeile entspricht genau der zweiten Voraussetzung. Der Beweis von (4) verläuft analog.

Die Technik der Datenverfeinerung eignet sich in den Fällen, in denen ein abstraktes Programm in ein konkretes *strukturgleiches* verfeinert wird. Wie bei der prozeduralen Verfeinerung, werden nicht-funktionale Aspekte wie Zeit- und Speicheraufwand nicht (im Verfeinerungskalkül) formalisiert, auch wenn diese die Motivation für Verfeinerung sind. Ein Beispiel hierfür wäre die Implementierung einer Menge mit einer Hash-Funktion.

4 Objektorientierung im Verfeinerungskalkül

Im vorangegangenem Kapitel wurde ein Modell von Anweisungen durch Prädikatentransformer gegeben. Dieses Modell wird jetzt um objektorientierte Elemente erweitert:

- Objekte mit Methoden und (gekapselten) Attributen (4.1)
- Attributselektion und Methodenaufrufe (4.2)
- Untertypisierung (4.3)
- beschränkte Parametrisierung (4.4)
- Klassen und Klasseninstanzen, Vererbung (4.5)

Außerdem werden parametrisierte Konstanten- und Typdeklaration eingeführt. Die Definitionen werden im Typsystem $F^{\omega}_{\leq}$ gegeben (siehe Anhang).

4.1 Objekte

Objekte haben einen *Zustand* und bieten *Operationen* auf diesem Zustand an. Ein Zustand ist ein Verbund, dessen Felder *Attribute* sind. So ist beispielsweise der Zustand eines zweidimensionalen Punktes:

$$(color \mapsto blue, xcoord \mapsto 17, ycoord \mapsto 3)$$

Mögliche Operationen auf diesem Zustand sollen – der Schreibweise von Prozeduren folgend – sein:

$$set\ (x : Int, y : Int)$$
$$get\ (\ )\ x : Int, y : Int$$

Die Operationen eines Objektes bilden ebenfalls einen Verbund, dessen Felder *Methoden* sind, z. B.:

$$(set \mapsto xcoord, ycoord := x, y\ ;\ exit\ x, y,$$
$$get \mapsto enter\ x, y\ ;\ x, y := xcoord, ycoord)$$

Die Zustandsräume der Methoden *set* und *get* umfassen Attribute, Wert- und Resultatparameter. Der nach außen sichtbare Typ der Methoden soll aber die privaten Attribute verstecken. Deshalb wird von den Attributen abstrahiert und diese durch das Symbol *Attr* ersetzt:

$$(set : Tran\ (Attr \oplus (x : Int, y : Int))\ Attr,$$
$$get : Tran\ Attr\ \ (Attr \oplus (x : Int, y : Int)))$$

Dieser Verbund wird mit dem Typ *Attr* parametrisiert und *PointMeth* genannt:

$$PointMeth \mathrel{\widehat{=}} (\Lambda\ Attr \bullet$$
$$\quad (set : Tran\ (Attr \oplus (x : Int, y : Int))\ Attr,$$
$$\quad get : Tran\ Attr\ \ (Attr \oplus (x : Int, y : Int)))$$

PointMeth ist eine Funktion von Typen nach Typen, d. h. der Art:

$$PointMeth : * \to *$$

Von den Attributen eines Punktes soll das Attribut *color* öffentlich sein:

$$ColPub \mathrel{\widehat{=}} (color : Int)$$

Die öffentlichen Attribute und die Methoden bilden die *Schnittstelle* bzw. die *Signatur* eines Objektes. Die privaten Attribute sollen gegen Zugriffe von außen gekapselt sein. Der Typ der farbigen Punkte muß also eine Schnittstelle bestehend aus *set*, *get* und *color* beschreiben und private Attribute verstecken. Dies wird durch einen existentiellen Quantor erreicht:

$$ColPoint \mathrel{\widehat{=}} (\sum Attr <: ColPub \bullet Attr \times PointMeth\ Attr)$$

Der Typ *ColPoint* umfaßt alle Objekte mit einem privaten Typ *Attr* und einem *Inhalt Attr* × *PointMeth Attr*, wobei *Attr* ein Untertyp von *ColPub ist*. Die Untertypisierung bedeutet, daß *Attr* mindestens das Feld *color* : *Int* hat. *Attr* kann aber auch weitere Felder haben.

Dieses Konstruktionsprinzip kann allgemein durch eine Funktion *Object* ausgedrückt werden, die aus dem Typ der öffentlichen Attribute und dem Typ der Methoden einen *Objekttyp* erzeugt.

Definition (*Objekttypkonstruktor*). Es sei Pub ein Verbundtyp und *Meth* ein Typoperator der Art $* \to *$.

$$Object\ Pub\ Meth \quad \widehat{=} \quad (\sum Attr <: Pub \bullet Attr \times Meth\ Attr)$$

Object ist eine Funktion, die einen Typ und eine Funktion von Typen nach Typen als Parameter nimmt und einen Typ liefert, d. h. von der Art:

$$Object : * \to (* \to *) \to *$$

Der Typ der farbigen Punkte kann damit kompakter ausgedrückt werden durch

$$ColPoint \mathrel{\widehat{=}} Object\ ColPub\ PointMeth \ .$$

Eine konventionellere Schreibweise für Objekttypen, wie sie künftig benutzt wird, ist:

$ColPoint \mathrel{\widehat{=}}$
  object
    $color : Int;$
    $set\ (x : Int, y : Int),$
    $get\ ()\ x : Int, y : Int$
  end

Ein Objekt vom Typ *ColPoint* hat beispielsweise Attribute

$$pt1state \mathrel{\widehat{=}} (color \mapsto blue, xcoord \mapsto 7, ycoord \mapsto 13)$$

vom Typ

$$pt1Attr \mathrel{\widehat{=}} (color : Int, xcoord : Int, ycoord : Int) \ .$$

Die Methoden seien gegeben durch:

$pt1oper \mathrel{\widehat{=}}$
  $(set \mapsto xcoord, ycoord := x, y\ ;\ exit\ x, y,$
  $get \mapsto enter\ x, y\ ;\ x, y := xcoord, ycoord)$
$pt1oper : PointMeth\ pt1Attr$

Punkte können, als Werte eines existentiellen Typs, nur durch *Packen* konstruiert werden (siehe Anhang). Der Ausdruck

$$pack\ e\ by\ S\ as\ T$$

steht für einen Wert vom Typ T mit gekapseltem Typ S und Inhalt e. Ein Objekt vom Typ *ColPoint* ist

$$pt1 \mathrel{\widehat{=}} pack\ (pt1state, pt1oper)\ by\ pt1Attr\ as\ ColPoint\ .$$

Dieses Konstruktionsprinzip kann allgemein durch eine Funktion *MakeObject* ausgedrückt werden.

Definition (*Objektkonstruktor*). Es sei *Pub* ein Verbundtyp, $Attr \trianglelefteq Pub$ und *Meth* ein Typoperator der Art $* \to *$.

$$\begin{array}{l} MakeObject\ Pub \mathrel{\widehat{=}} \\ \quad (\lambda\ state, oper : Attr, Meth\ Attr \bullet \\ \qquad pack\ (state, oper)\ by\ Attr\ as\ (Object\ Pub\ Meth)) \end{array}$$

MakeObject ist eine Funktion, die einen Typ (Pub) und ein Paar von Werten ($state$, $attr$) als Parameter nimmt und einen (gepackten) Wert liefert:

$$MakeObject\ : (\prod Pub \bullet Attr \times Meth\ Attr \to Object\ Pub\ Meth)$$

Obiges Objekt *pt1* kann kompakter definiert werden durch:

$$pt1 \mathrel{\widehat{=}} MakeObject\ ColPub\ (pt1state, pt1oper)$$

Eine einfachere Schreibweise für Objekte, wie sie künftig benutzt wird, ist

$$pt1 = \langle\!| xcoord \mapsto 7, ycoord \mapsto 13 \bullet color \mapsto blue; set \mapsto \ldots, get \mapsto \ldots |\!\rangle\ .$$

Die privaten Attribute werden von den öffentlichen Attributen und den Methoden durch einen Punkt getrennt. Die Schreibweise soll andeuten, daß es sich bei den Namen der privaten Attribute um gebundene Namen handelt, die durch andere entsprechend ersetzt werden können.

Zu bemerken ist, daß die Objekte eines Typs nicht alle die gleichen Methodenrümpfe und die gleichen privaten Attribute zu haben brauchen. Dies ist eine Verallgemeinerung gegenüber üblichen Programmiersprachen. So können Punkte vom Typ *Point* erzeugt werden, die beispielsweise Polarkoordinaten benutzen.

4.2 Attributselektion und Methodenaufruf

Um auf den Inhalt eines Objektes zugreifen zu können muß dieses als Wert eines existentiellen Typs *geöffnet* werden. In

$$open\ e\ as\ A <: T\ by\ c : S\ in\ f\ end$$

steht e für den zu öffnenden Wert und T für die Schranke des gekapselten Typs. Die gebundene Variable A steht für den gepackten Typ und die gebundene Variable c für den gepackten Inhalt. Der Gültigkeitsbereich von A ist der Typausdruck S und der Ausdruck f, der Gültigkeitsbereich von c ist der Ausdruck f.

Beispielsweise wird das öffentliche Attribut *color* von *pt1* selektiert durch:

$$\begin{array}{ll} & open\ pt1\ as\ Attr <: ColPub\ by\ state, oper\ : Attr, PointMeth\ Attr \\ & in\ state.color\ end \\ = & \text{«pack-open-Reduktion»} \\ & blue \end{array}$$

Die verwendete Regel ist im Anhang gegeben. Dieser Selektionsmechanismus kann allgemein durch eine Funktion $e.a$ ausgedrückt werden.

Definition (*Selektion öffentlicher Attribute*). Es sei Pub ein Verbund mit Feld a und e vom Typ *Object Pub Meth*.

$$e.a \ \widehat{=}\ open\ e\ as\ Attr <: Pub\ by\ state, oper : Attr, Meth\ Attr\ in\ state.a\ end$$

Falls T der Typ des Feldes a in Pub ist, gilt $e.a : T$. Mit der Selektion $e.a$ können keine privaten Attribute selektiert werden. In $e.a$ muß a ein Feld von Pub sein, damit der Ausdruck wohltypisiert ist. Obige Selektion von *color* aus *pt1* kann damit kompakter ausgedrückt werden durch

$$pt1.color = blue\ .$$

Es können auch keine Methoden auf ähnliche Weise selektiert werden, auch wenn der Name der Methoden (im Gegensatz zu dem der privaten Attribute) nach außen sichtbar ist. Der Grund hierfür ist, daß der Typ der Methoden nicht nach außen bekannt ist, weil er von den gekapselten Attributen abhängt. Methoden als Teile von Objekten können nur durch Methodenaufrufe benutzt werden. Diese werden im folgenden definiert.

Die *Repräsentation* eines Objektes besteht aus dem Typ der Attribute und dem Verbund von Methoden. Die Motivation für diese Festlegung ist, daß sich die Attribute eines Objektes durch Methodenaufrufe ändern können, nicht aber seine Repräsentation. Das Konstrukt

$$rep\ x\ as\ Attr, oper\ in\ p\ end$$

öffnet die Programmvariable x vom Typ $Object\ Pub\ Meth$ im initialen Zustand und weist $Attr, oper$ die Repräsentation von x mit Sichtbarkeitsbereich p zu.

Definition (*Repräsentation*). Es sei Pub ein Verbundtyp, $Meth : * \rightarrow *$, x eine Programmvariable vom Typ $Object\ Pub\ Meth$ und $p : Tran\ S\ T$ ein Prädikatentransformer über x.

$$\begin{aligned}&rep\ x\ as\ Attr, oper\ in\ p\ end \mathrel{\widehat{=}}\\ &\quad (\lambda\ \beta : Pred\ T \bullet \lambda\ \sigma : S \bullet\\ &\qquad open\ s.x\ as\ Attr <: Pub\ by\ state, oper : Attr, Meth\ Attr\\ &\qquad in\ p\ \beta\ \sigma\ end)\end{aligned}$$

Ein Unterschied von *rep* zu *open* ist, daß das erste Argument von *rep* der Name eine Programmvariable ist, während es bei *open* ein Ausdruck ist. Ein weiterer Unterschied ist, daß in *rep* nur der Methodenverbund benannt werden, während in *open* der gesamte Inhalt bestehend aus Attributverbund und Methodenverbund benannt wird. Der Typ von $rep\ x\ as\ Attr, oper\ in\ p\ end$ ist derselbe wie von p:

$$rep\ x\ as\ Attr, oper\ in\ p\ end : Tran\ S\ T$$

Ist die Repräsentation $Attr, oper$ eines Objektes x bekannt, werden mit *push x as Attr, oper* seine Attribute dem aktuellen Zustandsraum hinzugefügt. Mit *pop x as* $Attr, oper$ wird ein neues Objekt mit der gegebenen Repräsentation und den Attributen aus dem aktuellen Zustandsraum erzeugt.

Definition (*push und pop von Attributen*). Es sei Pub ein Verbundtyp, $Attr = (attr : A)$ ein Untertyp von Pub und $Meth : * \rightarrow *$. Es sei x eine Programmvariable vom Typ $Object\ Pub\ Meth$.

$$\begin{aligned}&push\ x\ by\ Attr, oper \mathrel{\widehat{=}}\\ &\quad x \rightarrow attr :[attr' \bullet x = MakeObject\ Pub\ ((attr \mapsto attr'), oper)]\end{aligned}$$

$$pop\ x\ by\ Attr, oper \mathrel{\widehat{=}}$$
$$attr \rightarrow x := MakeObject\ Pub\ ((attr \mapsto attr), oper)$$

Die Anweisung *push x by Attr*, *oper* erwartet, daß x tatsächlich die Repräsentation *Attr*, *oper* hat. Falls dies nicht der Fall ist, ist sie *miracle*. Für die Typen von *push* und *pop* wie oben gilt:

$$push\ x\ by\ Attr, oper \ :\ Tran\ (x : Object\ Pub\ Meth)\ Attr$$
$$pop\ x\ by\ Attr, oper \ :\ Tran\ Attr\ (x : Object\ Pub\ Meth)$$

Satz (*Aufheben von push und pop*). Es sei x eine Programmvariable vom Typ *Object Pub Meth*, es sei $Attr \leq Pub$ und $oper : Meth\ Attr$. Dann gilt:

$$pop\ x\ by\ Attr, oper\ ;\ push\ x\ by\ Attr, oper \ = \ skip$$

Beweis.

$$pop\ x\ by\ Attr, oper\ ;\ push\ x\ by\ Attr, oper$$
$$= \quad \text{«pop, push»}$$
$$attr \rightarrow x := MakeObject\ Pub\ ((attr \mapsto attr), oper)\ ;$$
$$x \rightarrow attr :[attr' \bullet x = MakeObject\ Pub\ ((attr \mapsto attr'), oper)]$$
$$= \quad \text{«Sequentielle Komposition von Zuweisungen, Logik»}$$
$$attr \rightarrow attr :[attr' \bullet attr = attr']$$
$$= \quad \text{«skip»}$$
$$skip$$

Ein Methodenaufruf läuft in mehreren Schritten ab:

(1) Das Objekt wird geöffnet und seine Repräsentation bestimmt.
(2) Der Zustandsraum wird um die Attribute des Objektes erweitert.
(3) Die Methode wird im erweiterten Zustandsraum aufgerufen, Wert- und Resultatparameter werden wie bei Prozeduren übergeben.
(4) Der geänderten Attribute aus dem Zustandsraum werden wieder in das Objekt gepackt.

Definition (*Methodenaufruf*). Es sei *Pub* ein Verbundtyp, $Meth : * \rightarrow *$ und x eine Programmvariable vom Typ *Object Pub Meth*.

$x.m\ () \mathrel{\widehat{=}}$
rep x as Attr, oper in
push x by Attr, oper ;
oper.m ;
pop x by Attr, oper
end

Der Typ von $x.m\ ()$ ist der Typ des Feldes m in $Meth\ ()$. Zu beachten ist, daß der Typ von $oper.m$ von der Form $Tran\ (Attr \oplus (val : V))\ (Attr \oplus (res : R))$ ist. Die Attribute *Attr* existieren also nach dem eigentlichen Aufruf der Methoden und können somit durch *pop* wieder in das Objekt x gepackt werden. (Es wären aber auch Verallgemeinerungen dahingehend denkbar, daß ein Methodenaufruf die Methoden oder auch den Typ der Attribute und Methoden ändert.) Um die Schreibweise von Methodenaufrufen von der Feldselektion zu unterscheiden, werden Klammern hinter dem Methodenaufruf geschrieben.

Parametrisierte Methodenaufrufe werden gleich behandelt wie parametrisierte Prozeduraufrufe. Beispielsweise ist

$pt1.set\ () : Tran\ (pt1 : ColPoint, x : Int, y : Int)\ (pt1 : ColPoint)$,
$pt1.get\ () : Tran\ (pt1 : ColPoint)\ (pt1 : ColPoint, x : Int, y : Int)$.

Es gilt:

pt1.set (8,9)
= «parametrisierter Prozeduraufruf»
enter $x, y : Int, Int$; $x, y := 8, 9$; *pt1.set*()

Außerdem gilt:

$a, b :=$ *pt1.get* ()
= «parametrisierter Prozeduraufruf»
pt1.get() ; $a, b := x, y$; *exit* x, y

Eine Attributselektion $x.a$ ist von dem Typ des selektierten Attributes. Sie kann deshalb überall stehen, wo ein Wert dieses Typs erwartet wird, beispielsweise auf der rechten Seite einer Zuweisung oder in der Bedingung einer Verzweigung:

if pt1.color = blue then ... else ... end

Ein Methodenaufruf $x.m()$ ist vom Typ eines Prädikatentransformers. Eine Verzweigung der Art *if pt1.get*() = (7,13) *then … else … end* ist deshalb nicht möglich. Für eine Verzweigung mit dieser Intention müßte zuerst das Resultat von *pt1.get*() Hilfsvariablen zugewiesen werden, die dann in der Bedingung der Verzweigung vorkommen können. Um in diesen Situationen eine einfache Formulierung zu ermöglichen, werden öffentliche Attribute zugelassen.

Der Methodenaufruf *pt1.get*() ist vom Typ eines Prädikatentransformers, weil er auch die privaten Attribute ändern kann. Eine Alternative wäre, zwei Arten von Methoden zu betrachten: rein lesende Methoden und Methoden, die sowohl lesen als auch ändern können. Dann könnten rein lesende Methoden in Bedingungen aufgerufen werden. Diese Alternative wird bereits in (Hoare, 1972) verworfen, weil es auch sinnvoll ist, lesende Methoden in solche, die sowohl lesen als auch schreiben, zu verfeinern (und bei der Verfeinerung der Typ der Methoden gleich bleiben muß). Ein Beispiel hierfür ist das Suchen eines Elementes in einer einfach verketteten Liste, bei dem das gefundene Element an den Anfang der Liste gestellt wird, damit das Suchen häufig benutzter Elemente beschleunigt wird.

4.3 Untertypisierung von Objekten

Die Untertyprelation von $F^{\omega}_{\leq}$ führt auch zu Monotonie-Eigenschaften bei Objekttypen: beispielsweise führt eine Untertypisierung der öffentlichen Attribute zu einer Untertypisierung der Objekttypen.

Satz (*Untertypisierung von Attributen*). Es seien Pub und Pub' Verbundtypen und $Meth : * \to *$. Dann gilt:

$$\frac{E \vdash Pub' \leq Pub}{E \vdash Object\ Pub'\ Meth \leq Object\ Pub\ Meth}$$

Beweis. Mit der Regel *≤–existentieller Typ* (siehe Anhang) und der Definition von *Object* gilt:

$$\frac{\dfrac{E \vdash Pub' \leq Pub}{E \vdash (\sum Attr <: Pub' \bullet Attr \times Meth\ Attr) \leq (\sum Attr <: Pub \bullet Attr \times Meth\ Attr)}}{E \vdash Object\ Pub'\ Meth \leq Object\ Pub\ Meth}$$

Es sei beispielsweise

$$Point \mathrel{\widehat{=}} Object\ ()\ PointMeth \quad .$$

Wegen $ColAttr \leq ()$ gilt

$$ColPoint \leq Point \quad .$$

Anweisungen sind, analog zu Funktionen, bezüglich Untertypisierung kontravariant im ersten Argument und kovariant im zweiten Argument.

Satz (*Untertypisierung von Prädikatentransformern*). Es seien S, S', T, T' Typen. Dann gilt:

$$\frac{E \vdash S \leq S' \quad E \vdash T' \leq T}{E \vdash Tran\ S'\ T' \leq Tran\ S\ T}$$

Beweis. Mit den Regeln ≤*–Reflexivität* und ≤*–Funktionstyp* und der Definition von *Pred* und *Tran* gilt:

$$\frac{E \vdash S \leq S' \quad E \vdash T' \leq T}{\dfrac{E \vdash S \leq S' \quad E \vdash Bool \leq Bool \quad E \vdash T' \leq T}{\dfrac{E \vdash S' \to Bool \leq S \to Bool \quad E \vdash T \to Bool \leq T' \to Bool}{\dfrac{E \vdash Pred\ S' \leq Pred\ S \quad E \vdash Pred\ T \leq Pred\ T'}{\dfrac{E \vdash Pred\ T' \to Pred\ S' \leq Pred\ T \to Pred\ S}{E \vdash Tran\ S'\ T' \leq Tran\ S\ T}}}}}$$

Die Untertypisierung von Kreuzprodukten ist durch die Untertypisierung der Argumente gegeben, d. h.:

$$\frac{E \vdash S' \leq S \quad E \vdash T' \leq T}{E \vdash S' \times T' \leq S \times T}$$

(Diese Regel läßt sich aus der Kodierung von Kreuzprodukten in „minimalem“ $F^{\omega}_{\leq}$ ableiten, siehe z. B. (Curien & Ghelli, 1992)).

Die Untertypisierung von Typoperatoren ist definiert durch die punktweise Erweiterung, d. h. für K, $K' : * \to *$ bedeutet $K' \leq K$, daß $K'\ T \leq K\ T$ für alle Typen T.

Die Untertypisierung der Methoden eines Objekttypes führt zu der Untertypisierung des Objekttypes in folgendem Sinne:

Satz (*Untertypisierung von Methoden*). Es sei Pub ein Verbundtyp und *Meth'*, *Meth* Typoperatoren der Art $* \to *$. Dann gilt:

$$\frac{E \vdash Meth' \leq Meth}{E \vdash Object\ Pub\ Meth' \leq Object\ Pub\ Meth}$$

Beweis. Mit der obigen Untertypisierungsregel für Kreuzprodukte, der Regeln ≤– *existentieller Typ* und der Definition von *Object* gilt:

$$\frac{\dfrac{\dfrac{\dfrac{\dfrac{E \vdash Meth' \leq Meth}{E \vdash Meth'\ Attr \leq Meth\ Attr}}{E \vdash Attr \times Meth'\ Attr \leq Attr \times Meth\ Attr}}{E \vdash (\sum Attr <: Pub \bullet Attr \times Meth'\ Attr) \leq (\sum Attr <: Pub \bullet Attr \times Meth\ Attr)}}{E \vdash Object\ Pub\ Meth' \leq Object\ Pub\ Meth}$$

Es sei beispielsweise:

$$\begin{aligned}
&MoveablePointMeth \mathrel{\widehat{=}} (\Lambda\ Attr \bullet \\
&\quad (set : Tran\ (Attr \oplus (x : Int, y : Int))\ Attr, \\
&\quad get : Tran\ Attr\ (Attr \oplus (x : Int, y : Int)), \\
&\quad move : Tran\ (Attr \oplus (dx : Int, dy : Int))\ Attr)) \\
&MoveablePoint \mathrel{\widehat{=}} Object\ ()\ MoveablePointMeth
\end{aligned}$$

Offensichtlich gilt $MoveablePointMeth \leq PointMeth$. Daraus folgt mit dem obigen Satz:

$$MoveablePoint \leq Point$$

Verknüpft mit der Untertyprelation ist die *Subsumptionsregel*. Für Objekttypen S, T besagt sie, daß ein Objekt vom Typ S auch ein Objekt vom Typ T ist, falls S ein Untertyp von T ist. So gilt mit *pt1* wie oben:

$$\begin{aligned}
&Colored \mathrel{\widehat{=}} object\ color : Int\ end \\
&pt1 : Colored
\end{aligned}$$

Ein Objekt eines Untertyps kann immer als Parameter übergeben werden, wenn ein Objekt eines Objekttyps erwartet wird. Ein Beispiel hierfür ist:

$$isblack \;\widehat{=}\; (\lambda\, cl : Colored \bullet cl.color = black)$$
$$isblack\ pt1 \;=\; false$$

Dies bedeutet insbesondere, daß ein Objekt eines Untertyps an eine Programmvariable eines Obertyps zugewiesen werden kann. Es sei *pt* eine Programmvariable vom Typ *Point*:

$$pt := pt1 \;:\; Tran\ (pt : Point)\ (pt : Point)$$

Zu bemerken ist, daß in $F^{\omega}_{\leq}$ nicht zwischen „dynamischem“ und „statischem“ Typ unterschieden wird. Der (kleinste) Typ ist immer statisch bestimmbar. Im obigen Beispiel besagt dies, daß das Attribut *color* von *pt* nach der Zuweisung nicht selektierbar ist, auch wenn *pt* dieses Attribut besitzt.

Die allgemeine Untertypisierungsregel für Objekte ergibt sich aus der Kombination der Regeln für die Untertypisierung von Attributen, Prädikatentransformern und Methoden. Sie besagt, daß zwei Objekttypen in einer Untertyprelation stehen, falls

(1) alle öffentlichen Attribute des Obertyps auch öffentliche Attribute des Untertyps sind;

(2) die Typen der gemeinsamen Attribute in einer Untertyprelation stehen;

(3) alle Methoden des Obertyps auch Methoden des Untertyps sind mit den gleichen Anzahl und gleichen Namen der Wert- und Resultatparameter;

(4) die Typen der sich entsprechenden Wertparamter in einer Obertyprelation stehen (*Kontravarianz*);

(5) die Typen der sich entsprechenden Resultatparameter in einer Untertyprelation stehen (*Kovarianz*).

Dies wird durch folgenden Satz ausgedrückt. Zur einfacheren Schreibweise steht für gleich lange Listen S, T von Typen der Ausdruck $S \leq T$ für die Untertypisierung der entsprechenden Elemente der Listen.

Satz (*Untertypisierung von Objekten*). Es seien *m1*, ...*mi*, ...*mj* Namen, es seien *pub*, *pub*', *v1*, ...*vi*, ...*vj*, *r1*, ...*ri*, ...*rj* Listen von Namen und *V1*, ...*Vi*, *R1*, ...*Ri*, *W1*, ...*Wi*, ..., *Wj*, S1, ..., *Si*, ...*Sj* Listen von Typen. Dann gilt:

$$\frac{E \vdash P \le Q \quad E \vdash V1 \le W1 \quad \ldots \quad E \vdash Vi \le Wi \quad E \vdash S1 \le R1 \quad \ldots \quad E \vdash Si \le Ri}{\begin{array}{l} E \vdash \mathit{object}\ \mathit{pub} : Q, \mathit{pub}' : Q', \mathit{m1}\ (\mathit{v1} : W1)\ \mathit{r1} : S1, \ldots, \\ \qquad \mathit{mi}\ (\mathit{vi} : Wi)\ \mathit{ri} : Si, \ldots, \mathit{mj}\ (\mathit{vj} : Wj)\ \mathit{rj} : Sj\ \mathit{end} \le \\ \quad \mathit{object}\ \mathit{pub} : Q, \mathit{m1}\ (\mathit{v1} : V1)\ \mathit{r1} : R1, \ldots, \mathit{mi}\ (\mathit{vi} : Vi)\ \mathit{ri} : Ri\ \mathit{end} \end{array}}$$

Beweis. Der Satz folgt unmittelbar aus den vorangegangenen drei Sätzen.

Es ist bekannt, daß die Kontravarianz-Regel in vielen praktischen Fällen zu restriktiv ist. Sie ist jedoch notwendig, um statische Typisierung zu erreichen. In (Weber, 1992) wird gezeigt, wie sie in gewissen Fällen gelockert werden kann. In Abschnitt 4.5 wird diskutiert, daß dem Kontravarianz-Problem hier Rechnung getragen wird, indem nur für Untertypisierung nicht aber bei Vererbung Kontravarianz gefordert wird.

4.4 Parametrisierung

In diesem Abschnitt wird gezeigt, welche Arten der Parametrisierung von Funktionen, Prozeduren und Typen in Verbindung mit Untertypisierung ausdrückbar sind und wie diese eingesetzt werden können.

Funktionen, die Werte liefern, können sowohl mit Werten als auch mit Typen parametrisiert sein. Zur einfacheren Schreibweise werden lokale Funktionsdeklarationen eingeführt:

Definition (*Funktionsdeklaration mit Wert- und Typparameter*). Es seien e, f Ausdrücke und V, T Typen.

$$(\mathit{val}\ h\ [A <: T]\ (v : V) = e \bullet f) \ \widehat{=}\ f\,[(\lambda\, A <: T \bullet (\lambda\, v : V \bullet e))/h]$$

Dabei ist e der *Rumpf* der Deklaration und f der *Gültigkeitsbereich*. Die Schranke T kann weggelassen werden, falls sie der größte Typ *TYPE* ist.

Beispiele für die Parametrisierung mit Werten sind (unter Weglassung des Gültigkeitsbereiches der Deklaration):

$$val\ isblack\ (cl : Colored) \ = \ (cl.color = black)$$
$$val\ darker\ (cl0 : Colored, cl1 : Colored) \ =$$
$$\quad if\ cl0.color \leq cl1.color\ then\ cl0\ else\ cl1\ end$$

Die Subsumptionsregel läßt die Anwendung dieser Funktion auf Objekte vom Typ *ColPoint* zu:

$$pt0 \mathrel{\widehat{=}} \langle\!| xcoord \mapsto 1, ycoord \mapsto 9 \bullet color \mapsto black; set \mapsto \ldots, get \mapsto \ldots |\!\rangle$$
$$isblack\ (pt0) = true$$
$$darker\ (pt0, pt1) = \langle\!| color \mapsto black |\!\rangle$$

Das Resultat der letzten Zeile kommt zustande, weil der Resultattyp von *darker* gleich dem der formalen Parameter *cl0* und *cl1* ist, also *Colored*. Der Typ von *pt0* und *pt1* geht bei der Anwendung „verloren“. Alternativ dazu kann die Funktion zusätzlich mit einem Typ parametrisiert werden:

$$val\ Darker\ [C <: Colored]\ (cl0 : C, cl1 : C) \ =$$
$$\quad if\ cl0.color \leq cl1.color\ then\ cl0\ else\ cl1\ end$$

Dann gilt:

$$Darker\ [ColPoint]\ (pt0, pt1) = pt0$$

Die eckige Klammer um *ColPoint* dient nur der besseren Lesbarkeit, sie hat keine Bedeutung. Analog zur Parametrisierung von Funktionen mit Typen lassen sich auch Prozeduren mit Typen parametrisieren.

Definition (*Prozedurdeklaration mit Typ-, Wert- und Resultatparameter*). Es seien q, r Prädikatentransformer, *val*, *res* Listen von Programmvariablen, T ein Typ und V, R Listen von Typen.

$$(proc\ n\ [A <: T]\ (val : V)\ res : R = r \bullet q) \mathrel{\widehat{=}}$$
$$\quad q[(\lambda\, A <: T \bullet enter\ res : R\ ;\ r\ ;\ exit\ val : V)/n]$$

Die Schranke T wird weggelassen, falls sie der größte Typ *TYPE* ist.

Es sei beispielsweise der Typ der grafischen Objekte, die sich verschieben lassen, gegeben durch:

$$Moveable \;\widehat{=}\; object\ move\ (dx\colon Int, dy\colon Int)\ end$$

Falls die Prozedur *Duplicate* deklariert ist mit

$$proc\ Duplicate\ [M <\colon Moveable]\ (m : M, dx : Int, dy : Int)\ d : M =$$
$$d := m\ ;\ d.move\ (dx, dy)$$

und *mpt*, *mpt*' Programmvariablen vom Typ *MoveablePoint* sind, ist eine zulässige Anwendung von *Duplicate*:

$$mpt' := Duplicate\ [MoveablePoint]\ (mpt, 20, 20)$$

Schließlich ist die Parametrisierung von Typen durch Typen möglich.

Definition (*Typdeklaration mit Typparameter*). Es sei K eine Art, T ein Typ und e ein Ausdruck.

$$(type\ B\ [A : K] = T \bullet e) \;\widehat{=}\; e[(\Lambda\ A : K \bullet T)/B]$$

Falls die Art des Typparameters $*$ ist, kann sie weggelassen werden. Ein Beispiel hierfür ist:

$$type\ Labelled\ [E] \;=\; (cont : E, label : String)$$
$$type\ LPoint \;=\; Labelled\ [Point]$$

4.5 Klassen

Eine *Klasse* beschreibt Objekte, die die gleiche interne Struktur haben, d. h. Objekte eines Typs, mit den gleichen Methodenrümpfen und Attributen. Klassen werden auf zwei Arten verwendet:

(1) Eine Klasse dient als Vorlage zum Erzeugen von Objekten;

(2) Eine Klasse kann aus einer bestehenden mittels Vererbung entstehen.

Der erste Punkt legt nahe, eine Klasse als ein initiales Objekt aufzufassen. Das initiale Objekt induziert eine Menge von Objekten, die mittels Methodenaufruf von dem initialen Objekt erreichbar sind. (Intuitiv wird oft unter einer Klasse

diese Menge verstanden). Der zweite Punkt macht es notwendig, die Attribute und Methoden einer Klassen offenzulegen, damit sie bei Vererbung benutzt und geändert werden können. Eine Klasse ist deshalb gegeben durch das Paar der initialen Werte der Attribute und der Methodenrümpfe. Somit ist eine Klasse ein Wert. Beispielsweise ist die Klasse *PointClass* von Punkten, die initial im Ursprung liegen, gegeben durch:

$$
\begin{array}{l}
\mathit{PointClass} \mathrel{\widehat{=}} \\
\quad ((\mathit{xcoord} \mapsto 0, \\
\quad \mathit{ycoord} \mapsto 0), \\
\quad (\mathit{set} \mapsto \mathit{xcoord}, \mathit{ycoord} := x, y \ ; \ \mathit{exit}\ x, y, \\
\quad \mathit{get} \mapsto \mathit{enter}\ x, y \ ; \ x, y := \mathit{xcoord}, \mathit{ycoord}))
\end{array}
$$

Ein konventionellere Schreibweise für diese Klasse ist:

$$
\begin{array}{l}
\mathit{PointClass} \mathrel{\widehat{=}} \\
\quad \mathit{class} \\
\qquad \mathit{xcoord} : \mathit{Int} := 0, \\
\qquad \mathit{ycoord} : \mathit{Int} := 0; \\
\qquad \mathit{set}\ (x : \mathit{Int}, y : \mathit{Int}) \quad = \quad \mathit{xcoord}, \mathit{ycoord} := x, y, \\
\qquad \mathit{get}\ ()\ x : \mathit{Int}, y : \mathit{Int} \quad = \quad x, y := \mathit{xcoord}, \mathit{ycoord} \\
\quad \mathit{end}
\end{array}
$$

Klassen haben, genau wie alle anderen Werte einen Typ. Die Funktion *Class* erzeugt einen Klassentyp aus den Typen der Attribute und Methoden.

Definition (*Klassentypkonstruktor*).

$$\mathit{Class} \quad \widehat{=} \quad (\Lambda\, \mathit{Attr} \bullet (\Lambda\, \mathit{Meth} : * \to * \bullet \mathit{Attr} \times \mathit{Meth}\ \mathit{Attr}))$$

Die Funktion *Class* ist von der Art $* \to (* \to *) \to *$. Beispielsweise gilt:

$$\mathit{PointClass} : \mathit{Class}\ \mathit{PointAttr}\ \mathit{PointMeth}$$

Mit der Funktion *make* lassen sich Objekte einer Klasse erzeugen. Es werden nur Klassen betrachtet, bei denen alle Attribute zu privaten Attributen der Objekte

werden. D. h. die Funktion *make* entspricht der Funktion *MakeObject*, nur daß die öffentlichen Attribute vom leeren Verbundtyp () sind [3].

Definition (*Objektkonstruktor von Klasse*). Es sei c : *Class Attr Meth* eine Klasse.

$$make\ c \ \widehat{=}\ MakeObject\ ()\ c$$

Der Typ von *make* ist *Class Attr Meth* → *Object* () *Meth*. Falls *pt* eine Programmvariable vom Typ *Point* ist, läßt sich ein neues Objekt der Klasse *PointClass* erzeugen durch:

$$pt := make\ PointClass$$

Eine Klasse kann durch *Vererbung* inkrementell modifiziert werden. Dazu wird eine Funktion *modify* definiert. Sie nimmt als Parameter die ursprüngliche Klasse und ein *Inkrement*. Das Inkrement gibt die neuen Attribute und die neuen Methoden an, wobei die neuen Methoden über den neuen *und* ursprünglichen Attributen sein können.

Definition (*Inkrementelle Modifikation*). Es sei $c = (init, oper)$ eine Klasse und $m = (init', oper')$ ein Inkrement mit den Feldern $init'$ disjunkt zu denen von $init$.

$$modify\ c\ m \ \widehat{=}\ (init \oplus init', oper \oplus oper')$$

Der Typ von *modify* ist:

$$\begin{array}{l} modify : \\ \quad Class\ Attr\ Meth \rightarrow \\ \quad\quad Class\ Attr'\ Meth' \rightarrow Class\ (Attr \oplus Attr')\ (Meth \oplus Meth') \end{array}$$

Die Klasse *MoveablePointClass* der Punkte, die eine zusätzliche Methode zum Verschieben hat, läßt sich durch inkrementelle Modifikation aus der Klasse *PointClass* ableiten.

---

[3] Dies ist keine prinzipielle Einschränkung, weil ein öffentliches Attribut immer durch ein privates Attribut mit entsprechenden *set* und *get* Methoden umschrieben werden kann. Diese Umschreibung ist für die Verfeinerung von Klassen (Kapitel 5) sinnvoll, denn ein privates Attribut kann in einer verfeinerten Klasse anders repräsentiert werden, während ein öffentliches Attribut nicht verfeinert werden könnte.

$$
\begin{array}{l}
\mathit{MoveablePointClass} \mathrel{\widehat{=}} \\
\quad \mathit{modify}\ \mathit{PointClass} \\
\quad\quad ((), (\mathit{move} \mapsto \\
\quad\quad\quad \mathit{xcoord}, \mathit{ycoord} := \mathit{xcoord} + \mathit{dx}, \mathit{ycoord} + \mathit{dy}\ ;\ \mathit{exit}\ \mathit{dx}, \mathit{dy}))
\end{array}
$$

Eine konventionellere Schreibweise hierfür ist:

$$
\begin{array}{l}
\mathit{MoveablePointClass} \mathrel{\widehat{=}} \\
\quad \mathit{class}\ (\mathit{PointClass}) \\
\quad\quad \mathit{move}\ (\mathit{dx} : \mathit{Int}, \mathit{dy} : \mathit{Int}) = \\
\quad\quad\quad \mathit{xcoord}, \mathit{ycoord} := \mathit{xcoord} + \mathit{dx}, \mathit{ycoord} + \mathit{dy} \\
\quad \mathit{end}
\end{array}
$$

Beim Erben von einer Klasse können die Attribute und Methoden beliebig geändert werden. Objekte der entstehenden Klasse sind nicht notwendigerweise von einem Untertyp der ursprünglichen Klasse. Ein Beispiel hierfür ist die Modifikation der Klasse *PointClass* zu 3D Punkten:

$$
\begin{array}{l}
\mathit{Point3DClass} \mathrel{\widehat{=}} \\
\quad \mathit{class}\ (\mathit{PointClass}) \\
\quad\quad \mathit{zcoord} : \mathit{Int} := 0; \\
\quad\quad \mathit{set}\ (x : \mathit{Int}, y : \mathit{Int}, y : \mathit{Int}) \quad = \quad \mathit{xcoord}, \mathit{ycoord}, \mathit{zcoord} := x, y, z, \\
\quad\quad \mathit{get}\ ()\ x : \mathit{Int}, y : \mathit{Int}, y : \mathit{Int} \quad = \quad x, y, z := \mathit{xcoord}, \mathit{ycoord}, \mathit{zcoord} \\
\quad \mathit{end}
\end{array}
$$

Vererbung kann also liberaler als in den meisten Programmiersprachen zur Konstruktion von Klassen eingesetzt werden. Trotzdem ist Typsicherheit garantiert. Dies ist durch die Trennung von Vererbung und Untertypisierung möglich. Wie in (Weber, 1992) beobachtet wird, können zwei Arten der Benutzung von Vererbung in Programmiersprachen unterschieden werden: Die eine dient der Integration von Subsystemen (Sammlung von abhängigen Klassen) in eine bestehende Umgebung; die andere dient der Adaption einer generellen Problemlösung für spezielle Bedürfnisse. In dem Ansatz dieser Arbeit dient die Untertypisierung (von Typen) dem ersten Zweck und die Vererbung (von Klassen) dem zweiten Zweck.

Ein Objekt heißt *Instanz* einer Klasse c, falls das Objekt die gleichen Methoden und Attributtypen wie c hat. Der Klassentest e *is* c ist wahr, falls e Instanz der Klasse c ist.

Definition (*Klassentest*). Es sei $c = (init, oper)$ eine Klasse vom Typ *Class Attr Meth* und e ein Ausdruck vom Typ *Object* () *Meth*.

$$(e \; is \; c) \;\widehat{=}\; (\exists \; state : Attr \bullet e = make \; (state, oper))$$

Es sei zum Beispiel ein Objekt *mpt* gegeben durch:

$$\begin{aligned} mpt \;\widehat{=}\; \langle\!| & xcoord \mapsto 9, ycoord \mapsto 4 \bullet \\ & set \mapsto xcoord, ycoord := x, y \; ; exit \; x, y, \\ & get \mapsto enter \; x, y \; ; x, y := xcoord, ycoord, \\ & move \mapsto xcoord, ycoord := xcoord + dx, ycoord + dy \; ; exit \; dx, dy |\!\rangle \end{aligned}$$

Das Objekt *mpt* ist vom Typ *MoveablePoint* und aufgrund der Subsumptionsregel auch vom Typ *Point*, weil $MoveablePoint \leq Point$. Das Objekt *mpt* ist eine Instanz der Klasse *MoveablePointClass*:

$$mpt \; is \; MoveablePointClass$$

Es gilt aber nicht, daß *mpt* eine Instanz von *PointClass* ist. Eine Instanz einer Klasse muß von genau dieser Klasse sein, eine Subsumptionsregel auf Klassen gibt es nicht. (In Sprachen wie Oberon ist der Test $x \; is \; c$ ein *Typtest*, kein *Klassentest*.)

Klassen lassen sich, wie alle Werte, mit Werten und Typen parametrisieren.

Beispiel (*Warteschlangen*). In einer Warteschlange lassen sich Elemente einfügen und wieder entnehmen. Die einfachste Form der Warteschlange garantiert keine Fairneß bei der Reihenfolge des Ein- und Ausganges sondern lediglich, daß keine Elemente verloren gehen und dupliziert werden. Sie wird definiert als eine Klasse mit einer Multimenge („*bag*“) als Attribut. Dabei ist

$\langle\,\rangle$	die leere Multimenge,
$\langle x \rangle$	die Multimenge bestehend aus x,
$b + c$	die Addition der Multimengen b und c,
$b - c$	die Subtraktion der Multimenge c von b,
$x \in b$	das Enthaltensein von x in der Multimenge b,
$\#b$	die Anzahl der Elemente in der Multimenge b.

Die Klasse *Queue* ist eine abstrakte Klasse mit einem generischen Parameter. Die Anweisung *error* steht für eine geeignete Fehlerbehandlung:

val Queue[*Item*] =
  class
    q : *bag of Item* := ⟨ ⟩;
    enqueue (i : *Item*) = $q := q + \langle i \rangle$,
    dequeue () i : *Item* = *if* $q \neq \langle\,\rangle$ *then* $i :[i' \bullet i' \in q]$; $q := q - \langle i \rangle$
      else error end,
    empty () e : *Bool* = $e := (q = \langle\,\rangle)$
  end

Die Klasse *PriorityQueue* stellt eine Variante von Warteschlangen dar, bei der die Elemente nach einer fixen Priorität entnommen werden. Diese Priorität wird als öffentliches Attribut der Elemente angegeben:

type OrderedItem = *object key* : *Int end*

val PriorityQueue [*Item* <: *OrderedItem*] =
  class (*Queue* [*Item*])
    dequeue () i : *Item* =
      if $q \neq \langle\,\rangle$ *then*
        $i :[i' \bullet (i' \in q) \wedge (\forall\, x : Item \bullet (x \in q) \Rightarrow (i'.key \leq x.key))]$;
        $q := q - \langle i \rangle$
      else error end
  end

Die Klasse *BoundedPriorityQueue* ist eine Warteschlange mit begrenzter Anzahl von Elementen:

val BoundedPriorityQueue [*Item* <: *OrderedItem*] (*max* : *Int*) =
  class (*PriorityQueue* [*Item*])
    enqueue (i : *Item*) = $\{\#q < max\}$; $q := q + \langle i \rangle$
  end

Eine Warteschlange mit begrenzter Anzahl von Elementen kann mit Hilfe eines Feldes implementiert werden. Ein Feld

a : *array n of T*

ist eine indizierte Sequenz fester Länge mit folgenden Konstruktoren und Selektoren:

$[e_1, \ldots, e_n]$	Feld mit Komponente $e_1, \ldots, e_n$
$[e]^n$	Feld der Länge n mit allen Komponenten e

$a[i]$	selektiert Komponente i vom Typ T falls $0 \le i < n$
$a[i \mapsto e]$	modifiziert a an Stelle i zu e falls $0 \le i < n$

Zuweisungen von Feldkomponenten und an Feldkomponenten haben folgende spezielle Semantik:

$$x := a[i] \quad = \quad x :[x' \bullet (0 \le i < n) \wedge (x' = a[i])]$$

Zuweisung von Feldkomponente

$$a[i] := e \quad = \quad a :[a' \bullet (0 \le i < n) \wedge (a' = a[i \mapsto e])]$$

Zuweisung an Feldkomponente

Die Elemente werden in dem Feld in der Reihenfolge des Einfügens abgelegt. Beim Entnehmen wird das Element mit der kleinsten Priorität gesucht und durch den letzten Eintrag im Feld ersetzt (die Initialisierung des Feldes wird weggelassen, weil sie irrelevant ist):

```
val ArrayPriorityQueue [Item <: OrderedItem] (max : Int)  =
    class
        a : array max of Item,
        n : Int := 0;
        enqueue (i : Item)  =  a[n] := i ; n := n + 1,
        dequeue () i : Item  =
            if n ≠ 0 then
                var h, p := 0, 1 •
                while p < n do
                    if a[h].key < a[p].key then h := p end; p := p + 1
                end
                i := a[h] ; n := n - 1 ; a[h] := a[n]
            else error end,
        empty () e : Bool    =  e := (n = 0)
    end
```

Zu beachten ist, daß für beliebigen Typ *Item* ≤ *OrderedItem* und ganzer Zahl *max* die Instanzen der Klassen

- *Queue*[*Item*]
- *PriorityQueue* [*Item*]
- *BoundedPriorityQueue* [*Item*] (*max*)
- *ArrayPriorityQueue* [*Item*] (*max*)

alle vom gleichen Typ sind, nämlich:

object enqueue (*i* : *Item*), *dequeue* () *i* : *Item*, *empty* () *e* : *Bool end*

Dies ist unabhängig davon, wie die Klassen durch Vererbung konstruiert sind. Das bedeutet, daß sich Instanzen dieser Klassen aneinander zuweisen lassen. Es kann dabei nie zu Fehlern wie „message not understood“ führen. Trotzdem ist dabei zu beachten, daß Objekte ein „kompatibles“ Verhalten haben können oder nicht. Beispielsweise verhalten sich Instanzen der Klasse *ArrayPriorityQueue* so wie Instanzen der Klasse *BoundedPriorityQueue*, aber nicht wie *PriorityQueue*. Das bedeutet, daß die Benutzung einer Instanz von *ArrayPriorityQueue* statt einer Instanz von *PriorityQueue* zwar typkorrekt ist aber im folgenden zu einem Überlauf führen kann. Dies führt zu der Forderung der Klassenverfeinerung, wie in Kapitel 5 definiert.

In diesem Beispiel haben, der Terminologie von (Rüping, et al., 1993) folgend *Queue*, *PriorityQueue* und *BoundedPriorityQueue* die Rolle von (parametrisierten) *Schnittstellenklassen*, *ArrayPriorityQueue* hat die Rolle einer *Implementierungsklasse*. Die Rolle von *OrderedItem*, in (Rüping, et al., 1993) als Konzeptklasse bezeichnet, würde man hier *Konzepttyp* nennen. Die Unterscheidung der Rollen sind ein Kriterium beim Entwurf von Klassen bzw. Typen.

4.6 Transformation von Methodenaufrufen

Das Problem bei der Verifikation und Verfeinerung von Anweisungen mit Methodenaufrufen, ist daß ein Methodenaufruf $z := x.m$ (e) nicht transformiert werden kann, wenn nur der *Typ* von x bekannt ist. Hierzu muß zusätzlich bekannt sein, von welcher *Klasse* das Objekt x Instanz ist. Im folgenden wird eine Möglichkeit gezeigt, Methodenaufrufe aufzulösen, falls die Klasse des Objektes durch eine Annahme gegeben ist.

Satz (*Auflösen von rep*). Es sei $c = (init, oper)$ eine Klasse vom Typ *Class Attr Meth* und x eine Programmvariable vom Typ *Object* () *Meth* und p ein Prädikatentransformer. Dann gilt:

$$[x \textit{ is } c] \,;\, \textit{rep } x \textit{ as } At, op \textit{ in } p \textit{ end} = [x \textit{ is } c] \,;\, p[Attr/At, oper/op]$$

Beweis. Es sei *Tran S T* der Typ von p. Für beliebiges Zustandsprädikat $\beta : Pred\ T$ und beliebigen Zustand $\sigma : S$ gilt:

$$
\begin{array}{ll}
 & ([x\ is\ c]\ ;\ rep\ x\ as\ At, op\ in\ p\ end)\ \beta\ \sigma \\
= & \text{«Sequentielle Komposition, Annahme mit booleschem Ausdruck»} \\
 & [\lambda\ \sigma : S \bullet eval\ (x\ is\ c)\ \sigma]\ (rep\ x\ as\ At, op\ in\ p\ end\ \beta)\ \sigma \\
= & \text{«Annahme, } eval\text{, Implikation»} \\
 & (\sigma.x\ is\ c) \Rightarrow rep\ x\ as\ At, op\ in\ p\ end\ \beta\ \sigma \\
= & \text{«}is, rep, make, st\text{ neuer Bezeichner»} \\
 & (\exists\ state : Attr \bullet \sigma.x = pack\ (state, oper)\ by\ Attr\ as\ Object\ ()\ Meth) \\
 & \qquad \Rightarrow open\ \sigma.x\ as\ Attr\ by\ st, op : Attr, Meth\ Attr\ in\ p\ \beta\ \sigma\ end \\
= & \text{«Logik, pack-open-Reduktion»} \\
 & (\forall\ state : Attr \bullet \sigma.x = pack\ (state, oper)\ by\ Attr\ as\ Object\ ()\ Meth \\
 & \qquad \Rightarrow (p\ \beta\ \sigma)[Attr/A,\ state/st, oper/op]) \\
= & \text{«Logik, } make, is, st\text{ nicht frei in } p\ \beta\ \sigma\text{»} \\
 & (\sigma.x\ is\ c) \Rightarrow (p\ \beta\ \sigma)[Attr/A, oper/op] \\
= & \text{«Implikation, } eval\text{, Annahme»} \\
 & ([\lambda\ \sigma : S \bullet eval\ (x\ is\ c)\ \sigma]\ (p\ \beta)[Attr/A, oper/op])\ \sigma \\
= & \text{«Annahme mit booleschem Ausdruck, Sequentielle Komposition»} \\
 & ([x\ is\ c]\ ;\ p[Attr/A, oper/op])\ \beta\ \sigma
\end{array}
$$

Satz. Es sei x eine Programmvariable vom Typ *Object* () *Meth* und $c = ((attr \mapsto init), oper)$ eine Klasse vom Typ *Class Attr Meth*. Dann gilt:

(1) $push\ x\ by\ Attr, oper = [x\ is\ c]\ ;\ push\ x\ by\ Attr, oper$ *is–push*

(2) $pop\ x\ by\ Attr, oper = pop\ x\ by\ Attr, oper\ ;\ [x\ is\ c]$ *is–pop*

Beweis. Der Satz folgt direkt aus den Definitionen von *push* bzw. *pop* und aus *Sequentielle Komposition von bösartigen Zuweisungen mit Annahmen*.

Satz (*Auflösen von Methodenaufrufen*). Es sei e eine Liste von Ausdrücken, z eine Liste von Programmvariablen, x eine Programmvariable vom Typ *Object Pub Meth* und es sei $c = (state, oper)$ vom Typ *Class Attr Meth*. Dann gilt:

$$
\begin{array}{l}
[x\ is\ c]\ ;\ z := x.m\ (e)\ = \\
\qquad push\ x\ as\ Attr, oper\ ;\ z := (oper.m)\ (e)\ ;\ pop\ x\ as\ Attr, oper
\end{array}
$$

Beweis.

$[x\ is\ c]\ ;z := x.m\ (e)$
= «Methodenaufruf»
$[x\ is\ c]\ ;rep\ x\ as\ At, op\ in\ push\ x\ by\ A, op\ ;op.m\ ;pop\ x\ by\ A, op\ end$
= «Auflösen von *rep*, Substitution»
$[x\ is\ c]\ ;push\ x\ by\ Attr, oper\ ;oper.m\ ;pop\ x\ by\ Attr, oper$
= «*is–push*»
$push\ x\ by\ Attr, oper\ ;oper.m\ ;pop\ x\ by\ Attr, oper$

Für die praktische Anwendung dieses Satzes ist es notwendig, die Klassenzugehörigkeit $[x\ is\ c]$ an den jeweiligen Stellen im Programm zu kennen. Folgender Satz zeigt, wie diese Information erzeugt und weitergeleitet werden kann.

Satz (*Propagieren von Klassenzugehörigkeit*). Es sei x eine Programmvariable vom Typ $Object\ ()\ Meth$ und $c : Class\ Attr\ Meth$. Dann gilt:

(1) $enter\ x := make\ c \quad = \quad enter\ x := make\ c\ ;[x\ is\ c]$
(2) $[x\ is\ c]\ ;z := x.m\ (e) \ = \ [x\ is\ c]\ ;z := x.m\ (e)\ ;[x\ is\ c]$

Beweis. Für (1) gilt:

$enter\ x := make\ c\ ;[x\ is\ c]$
= «Sequentielle Komposition von Zuweisungen (1)»
$[(make\ c)\ is\ c]\ ;enter\ x := make\ c$
= «*make, is*, *true–Annahme*, ;*–skip*»
$enter\ x := make\ c$

Für (2) gilt:

$[x\ is\ c]\ ;z := x.m\ (e)$
= «Auflösen von Methodenaufrufen»
$push\ x\ as\ Attr, oper\ ;z := (oper.m)\ (e)\ ;pop\ x\ as\ Attr, oper$
= «*is–pop*, Auflösen von Methodenaufrufen»
$[x\ is\ c]\ ;z := x.m\ (e)\ ;[x\ is\ c]$

Beispielsweise sei mpt eine Programmvariable vom Typ *MoveablePoint* wie früher. Dann gilt für beliebige $x1, x2, y1, y2$ vom Typ *Int*,

$$[mpt\ is\ MoveablePointClass]\ ;\ mpt.move\ (x1 + y1)\ ;\ mpt.move\ (x2 + y2) =$$
$$[mpt\ is\ MoveablePointClass]\ ;\ mpt.move\ (x1 + x2, y1 + y2)\ .$$

(Ohne die Annahmen über die Klasse von *mpt* gilt die Gleichung nicht!)

Es sei *mptoper* der Verbund der Methoden der Klasse *MoveablePointClass*:

$$mptoper =$$
$$(set \mapsto xcoord, ycoord := x, y\ ;\ exit\ x, y,$$
$$get \mapsto enter\ x, y\ ;\ x, y := xcoord, ycoord,$$
$$move \mapsto xcoord, ycoord := xcoord + dx, ycoord + dy\ ;\ exit\ dx, dy)\ .$$

Obige Gleichung wird wie folgt nachgewiesen:

$[mpt\ is\ MoveablePointClass]\ ;\ mpt.move\ (x1 + y1)\ ;\ mpt.move\ (x2 + y2)$
$=$ «Propagieren von Klassenzugehörigkeit (2)»
$[mpt\ is\ MoveablePointClass]\ ;\ mpt.move\ (x1 + y1)\ ;$
$[mpt\ is\ MoveablePointClass]\ ;\ mpt.move\ (x2 + y2)$
$=$ «Auflösen von Methodenaufrufen, Aufheben von *push* und *pop*»
$push\ mpt\ as\ PointAttr, mptoper\ ;$
$xcoord, ycoord := xcoord + x1, ycoord + y1\ ;$
$xcoord, ycoord := xcoord + x2, ycoord + y2\ ;$
$pop\ mpt\ as\ PointAttr, mptoper$
$=$ «Zusammenfassen von Zuweisungen»
$push\ mpt\ as\ PointAttr, mptoper\ ;$
$xcoord, ycoord := xcoord + x1 + x2, ycoord + y1 + y2\ ;$
$pop\ mpt\ as\ PointAttr, mptoper$
$=$ «Prozeduraufruf, Auflösen von Methodenaufrufen»
$[mpt\ is\ MoveablePointClass]\ ;\ mpt.move\ (x1 + x2, y1 + y2)$

4.7 Objektidentitäten

Objekte können beim Erzeugen mit einer eindeutigen *Identität* oder *Referenz* versehen werden, unter der auf sie zugegriffen werden kann. Dies ist bei der objektorientierten Modellierung eine natürliche Situation: Beispielsweise kann sich der Name oder die Adresse einer Person ändern, nicht aber ihre Identität. Das Referenzieren eines Objektes erfolgt immer bezüglich eines (ggf. impliziten) *Universums*, in dem sich das Objekt befindet.

In (Utting, 1992) werden Referenzen mit Zeigern (Luckham & Suzuki, 1979) gleich gesetzt. Der Ansatz hier basiert auf einer einfachen, abstrakten Formalisierung durch Funktionen höherer Stufe. Referenzen sind dabei Werte, die, wie andere Werte, als Parameter übergeben und kopiert werden können.

Für jeden Typ T wird dazu die Existenz einer unendlichen Menge von Referenzen auf Objekte des Typs T angenommen. Diese Menge wird als $Ref\ T$ bezeichnet. Die Funktion $\mathcal{U}\ T$ bildet Referenzen von Objekten des Typs T auf die Objekte ab:

$$\mathcal{U}\ T : Ref\ T \to T$$

Die Funktion $\mathcal{U}\ T$ stellt das Universum der Objekte des Typs T. Initial existieren keine referenzierbaren Objekte des Typs T. Dies wird modelliert, indem die Funktion $\mathcal{U}\ T$ so initialisiert wird, daß sie „undefinierte“ Objekte liefert. Es sei T_0 ein ausgezeichnetes Objekt vom Typ T, z. B. bei dem alle Methoden *miracle* sind. Die Initialisierung von $\mathcal{U}\ T$ ist dann

$$\mathcal{U}\ T := (\lambda\ id : Ref\ T \bullet T_0)\ .$$

Die Einträge in $\mathcal{U}\ T$, die nicht auf T_0 abbilden, stellen die *Extension* der Objekte des Typs T dar, d. h. die Menge aller referenzierbaren Objekte vom Typ T.

Die Anweisung $x := new\ c$ erzeugt ein neues Objekt der Klasse c und weist x seine Referenz zu. Dazu wählt die Anweisung aus den noch nicht benutzen Referenzen eine aus – die Auswahl eines Elementes aus einer nicht-leeren Menge ist durch das Auswahlaxiom sichergestellt. Sodann wird die Funktion $\mathcal{U}\ T$ so modifiziert, daß sie unter x das neue Objekt liefert. In der Definition von *new* wird der bedingte Ausdruck *if b then e else f end* mit der üblichen Bedeutung verwendet:

Definition (*Objekterzeugung*). Es sei c eine Klasse mit Instanzen vom Typ T und es sei x eine Programmvariable vom Typ T:

$$\begin{aligned} x := new\ c \quad \widehat{=}\ & \\ & x := some\ \{id : Ref\ \ T \mid (\mathcal{U}\ T)\ id \neq T_0\}\ ; \\ & \mathcal{U}\ T := (\lambda\ id : Ref\ T \bullet if\ id = x\ then\ make\ c\ else\ (\mathcal{U}\ T)\ id\ end) \end{aligned}$$

$\mathcal{U}\ T$ ist eine (implizite) Programmvariable in jedem Programm, das Objektreferenzen vom Typ $Ref\ \ T$ erzeugt, d. h.:

$$x := new\ c \quad : \quad Tran\ (x : T,\ \mathcal{U}\ T : Ref\ T \to T)\ (x : T,\ \mathcal{U}\ T : Ref\ T \to T)$$

Falls x eine Referenz auf ein Objekte ist, ist $x{\rightarrow}m()$ ein (parameterloser) Methodenaufruf, bei dem zuerst das Objekt unter der Referenz x bestimmt wird bevor die Methoden m aufgerufen wird.

Definition (*Methodenaufruf mit Referenz*). Es sei x eine Programmvariable vom Typ *Ref T* und m eine Methode des Objekttyps T:

$$
\begin{aligned}
&x{\rightarrow}m() \;\widehat{=}\\
&\quad (var\ h := (\mathcal{U}\ T)\ x \bullet\\
&\quad h.m();\\
&\quad \mathcal{U}\ T := (\lambda\ id : Ref\ \ T \bullet if\ id = x\ then\ h\ else\ (\mathcal{U}\ T)\ id\ end))
\end{aligned}
$$

Parametrisierte Methodenaufrufe mit Referenz werden analog auf parameterlose Methodenaufrufe zurückgeführt wie bei normalen Methodenaufrufen.

$\mathcal{U}\ T$ ist eine (implizite) Programmvariable in jedem Programm mit Methodenaufrufen von Objektreferenzen vom Typ *Ref T*, d. h. falls die Methode m auf keine globalen Variablen zugreift, ist der Typ von $x{\rightarrow}m()$:

$$x{\rightarrow}m() \;:\; Tran\ (x : T,\ \mathcal{U}\ T : Ref\ \ T \rightarrow T)\ (x : T,\ \mathcal{U}\ T : Ref\ \ T \rightarrow T)$$

Die Konsequenz dieser Typisierung ist, daß für zwei Objektreferenzen x, y vom gleichen Typ im allgemeinen

$$x{\rightarrow}m();\ y{\rightarrow}m() \;=\; y{\rightarrow}m();\ x{\rightarrow}m()$$

nicht gilt, weil $x{\rightarrow}m()$ und $y{\rightarrow}m()$ nicht unabhängig sind (beide haben $\mathcal{U}\ T$ in ihren Zustandsräumen), ein Problem das als *Aliasing* bekannt ist. Eine weitere Konsequenz ist, daß $\mathcal{U}\ T$ (implizite) globale Variable in jeder Prozedur und jeder Methode ist, die Referenzen auf T erzeugt oder aufruft.

5 Verhaltensorientierte Sicht auf Klassen

Eine Klasse legt die interne Struktur von Objekten fest. Objekte weisen auch ein nach außen beobachtbares Verhalten auf. In diesem Kapitel werden hierzu *verhaltensorientierte* (*observationelle*) Definitionen von *Klassenäquivalenz* und *Klassenverfeinerung* gegeben. Diese Relationen erlauben es, eine Klasse durch eine Klasse mit einer unterschiedlichen internen Struktur aber äquivalentem bzw. verfeinerndem beobachtbaren Verhalten zu ersetzen. Damit kann das Verhalten von Objekten der „realen Welt“ zunächst durch abstrakte Klassen problemnah beschrieben werden. Diese können in späteren Entwicklungsschritten durch effizient implementierte Klassen ersetzt werden.

Für den Nachweis der Klassenverfeinerung wird eine allgemeine Beweistechnik mittels Simulation eingeführt. Diese wird in späteren Kapiteln weiter spezialisiert.

5.1 Äquivalenz und Verfeinerung von Klassen

Auf ein Objekt kann auf zwei Arten gesehen werden: In der „glas box view“ sieht man die Interna eines Objektes und schließt daraus auf sein Verhalten. In der „black box view“ betrachtet man ein Objekt nur durch seine Schnittstelle (Signatur) und kann über sie sein Verhalten beobachten. Die Klasse eines Objektes bestimmt die Interna, d. h. die Sicht eines Objektes als Instanz einer Klasse entspricht der „glas box view“. Der Typ eines Objektes macht nur die Schnitt-

stelle bekannt, durch die ein Objekte beobachtet werden kann. Die Sicht eines Objektes als Element eines Typs entspricht der „black box view“. Diese Unterscheidung beruht darauf, daß die Interna eines Objektes, die Attribute und die Methodenrümpfe, im Typ durch einen existentiellen Quantor versteckt sind.

Diese Entkopplung der beiden Sichtweisen macht es möglich, die Interna eines Objektes zu ändern, ohne daß dies nach außen beobachtbar ist. In der Praxis läßt dies zu, Objekte durch Klassen auf eine problemnahe aber abstrakte (d. h. nicht ausführbare oder nicht effizient ausführbare) Art zu spezifizieren. In der Implementierung können Objekte dieser Klasse durch Objekte einer anderen Klasse ersetzt werden, falls sie das gleiche beobachtbare Verhalten haben.

Die verhaltensorientierte Sicht auf Klassen hängt von der Art der Beobachtungen der Objekte ab. Eine Beobachtungen kann z. B. darin bestehen, auf einem Objekt eine Reihe von ändernden Operationen durchzuführen und sein Verhalten dann durch inspizierende Operationen beobachten. Sowohl ändernde als auch inspizierende Operationen auf Objekten durch ihre Schnittstelle werden durch Methodenaufrufe getätigt. (Falls die Schnittstelle auch öffentliche Attribute umfaßt, kann ein Objekt auch durch das Lesen dieser inspiziert werden. Der Einfachheit halber werden in diesem Kapitel nur Methoden in Schnittstelle von Objekten betrachtet, ohne Beschränkung der Allgemeinheit.) Eine Beobachtung eines Objektes entspricht also einer Sequenz von Methodenaufrufen an dieses Objekt. Um von einem definierten Anfangszustand des Objektes ausgehen zu können, muß vor der Beobachtung das Objekt initialisiert, d. h. erzeugt werden.

Beispiel (*Mittelwertberechnung*). Es wird ein einfache Klasse zur Mittelwertberechnung spezifiziert, ähnlich zu den Datentypen in (Hoare & He, 1986; Morgan & Gardiner, 1990), mit zwei Methoden:

- füge eine Zahl einer Reihe von Zahlen hinzu, deren Mittelwert berechnet werden soll;
- berechne den Mittelwert aller bisher eingefügten Zahlen.

Der Zustand dieser Klasse ist durch eine Multimenge (*bag*) spezifiziert. Die Funktion $sum\ b$ auf Multimengen b, c von ganzen Zahlen ist axiomatisiert durch:

$$\begin{aligned} sum\ \langle\rangle &= 0 \\ sum\ \langle i\rangle &= i \\ sum\ (b + c) &= sum\ b + sum\ c \end{aligned}$$

Die Klasse besteht aus einem privaten Attribut und zwei öffentlichen Methoden:

$$
\begin{array}{l}
\mathit{MeanCalc} \mathrel{\widehat{=}} \\
\quad \mathit{class} \\
\qquad b : \mathit{bag\ of\ Int} := \langle\rangle; \\
\qquad \mathit{add}\ (i : \mathit{Int}) \quad = \ b := b + \langle i \rangle, \\
\qquad \mathit{mean}\ ()\ m : \mathit{Int} \ = \ m := \mathit{sum}\ b\ \mathit{div}\ \#b \\
\quad \mathit{end}
\end{array}
$$

Die Klasse *MeanCalc*' mit derselben Schnittstelle besteht aus zwei privaten Attributen, der ganzen Zahl s (für die Summe) und der ganzen Zahl n (für die Anzahl):

$$
\begin{array}{l}
\mathit{MeanCalc}' \mathrel{\widehat{=}} \\
\quad \mathit{class} \\
\qquad s : \mathit{Int} := 0, \\
\qquad n : \mathit{Int} := 0; \\
\qquad \mathit{add}\ (i : \mathit{Int}) \quad = \ s, n := s + i, n + 1, \\
\qquad \mathit{mean}\ ()\ m : \mathit{Int} \ = \ m := s\ \mathit{div}\ n \\
\quad \mathit{end}
\end{array}
$$

Eine Beobachtung eines Objektes der Klasse *MeanCalc* ist beispielsweise:

$$(\mathit{var}\ x := \mathit{make}\ \mathit{MeanCalc} \bullet x.\mathit{add}\ (3)\ ;\ x.\mathit{add}\ (7)\ ;\ a := x.\mathit{mean}\ ())$$

Mit den Transformationstechniken aus 4.6 läßt sich einfach zeigen, daß diese Beobachtung der Anweisung $a := 5$ entspricht. Verwendet man statt *MeanCalc* die Klasse *MeanCalc*',

$$(\mathit{var}\ x := \mathit{make}\ \mathit{MeanCalc}' \bullet x.\mathit{add}\ (3)\ ;\ x.\mathit{add}\ (7)\ ;\ a := x.\mathit{mean}\ ())\ ,$$

entspricht dies ebenfalls der Anweisung $a := 5$. Das bedeutet, daß die beiden Klassen *MeanCalc* und *MeanCalc*' für diese spezielle Beobachtung verhaltensäquivalent sind. Der Name der lokalen Programmvariable spielt in der beobachtenden Anweisung keine Rolle weil sich Variablen in lokalen Deklarationen beliebig umbenennen lassen.

Es liegt somit nahe zu definieren, daß zwei Klassen verhaltensäquivalent sind, falls alle Beobachtungen gleich sind. Beobachtungen können im allgemeinen mit beliebigen Programmkonstrukten zusammengesetzte Methodenaufrufe sein. Die Definition der ein Objekt x beobachtenden Anweisungen $\mathcal{B}[x]$ ist wie folgt moti-

viert. Nach dem Satz über die Darstellung monotoner Prädikatentransformer (Abschnitt 3.4) läßt sich jeder monotone Prädikatentransformer schreiben als eine Anweisung aus $\mathcal{A}$, wobei die Anweisungen aus $\mathcal{A}$ zusammengesetzt sind aus:

- $[\beta]$, $\{\beta\}$, $\langle\phi\rangle$
- $p\,;q$ falls $p, q \in \mathcal{A}$
- $(\sqcap\, i \in I \bullet p)$, $(\sqcup\, i \in I \bullet p)$ falls $p \in \mathcal{A}$ für alle $i \in I$

Eine nicht erlaubte Beobachtung eines Objektes x ist beispielsweise die Verzweigung

$$\textit{if } x \textit{ is } c \textit{ then } \ldots \textit{ else } \ldots \textit{ end} \;=\; ([x \textit{ is } c]\,;\ldots) \sqcap ([\neg x \textit{ is } c]\,;\ldots)$$

Damit wäre es nämlich möglich, Programme zu formulieren, die davon abhängen, daß ein Objekt von einer bestimmten Klasse ist. Solche Objekte lassen sich dann nicht durch verhaltensäquivalente oder verfeinernde ersetzen. Aus diesem Grund dürfen beobachtete Objekte nicht in Annahmen und (aus Dualitätsgründen) in Zusicherungen vorkommen. Dies ließe sich aber umlaufen, wenn das beobachtete Objekt an ein anderes zugewiesen wird und dieses dann in einer Annahme oder Zusicherung vorkommt. Deshalb wird die Zuweisung des beobachteten Objektes an andere Programmvariable verboten. Die einzigen erlaubten Beobachtungen an ein Objekt x sind Methodenaufrufe der Form $z := x.m(e)$, wobei x selbst nicht aktueller Parameter des Methodenaufrufes sein darf (was ja einer Zuweisung entsprechen würde).

Definition (*Beobachtende Anweisungen*). Es sei x eine Programmvariable vom Typ *Object* () *Meth*. Die Menge $\mathcal{B}[x]$ ist induktiv definiert durch

- $z := x.m(e)$, falls m Methode von x und x nicht in e und z vorkommt,
- $\langle\phi\rangle$, falls $\phi : S \to T$ und x nicht Feld von S, T,
- $\{\beta\}$, $[\beta]$, falls $\beta : S \to \mathit{Bool}$ und x nicht Feld von S,
- $q\,;r$, falls $q, r \in \mathcal{B}[x]$,
- $(\sqcap\, i \in I \bullet q)$, $(\sqcup\, i \in I \bullet q)$, falls $q \in \mathcal{B}[x]$ für alle $i \in I$.

Es wird auch der Einfachheit halber die Zuweisung von anderen Objekten an die beobachtete Programmvariable verboten. Nach einer solchen Zuweisung verhält

sich das beobachtete Objekte ohnehin unabhängig davon, welchen Wert es zuvor hatte, eine Verfeinerung dieses ist also belanglos.

Definition (*Verhaltensäquivalenz von Klassen*). Es seien c, d Klassen mit Instanzen vom Typ T. Klassen c und d sind verhaltensäquivalent, geschrieben $c \equiv d$, gdw. für alle Anweisungen $p \in \mathcal{B}[x]$ mit $x : T$,

$$(var\ x := make\ c \bullet p) \quad = \quad (var\ x := make\ d \bullet p) \ .$$

Die Gleichheit von Klassen ist komponentenweise definiert, d. h. zwei Klassen sind gleich wenn sie die gleichen Methoden und Attribute haben. Die Verhaltensäquivalenz ist eine schwächere Relation als die Gleichheit. Zwei Klassen können verhaltensäquivalent aber nicht gleich sein, wie beispielsweise die Klassen *MeanCalc* und *MeanCalc*'.

Ob zwei Klassen gleich oder verhaltensäquivalent sind hat unterschiedliche Auswirkungen, je nach dem ob die Klassen zum Erzeugen von Objekten benutzt werden oder um von ihnen zu erben. Falls von der Klasse c oder d geerbt wird, müssen c und d gleich sein, damit die entstehenden Klassen gleich sind. Falls ein Objekte der Klasse c oder d erzeugt wird, müssen c und d verhaltensäquivalent sein, damit die entstehenden Programme gleich sind.

Für die Definition von Verhaltensäquivalenz wird hier die größte sinnvolle Menge an Beobachtungen zugrunde gelegt. In (Nipkow, 1986) wird anhand von (bösartig) indeterministischen Datentypen gezeigt (Datentypen sind ähnlich zu den Klassen, nur daß keine Untertyp-Polymorphie betrachtet wird), daß in einer Programmiersprache ohne Rekursion mehr Datentypen verhaltensäquivalent sind. Verhaltensquivalenz hängt also von der Mächtigkeit der Beobachtungen ab. Die hier gewählten Beobachtungen gehen sogar über Programmiersprachen dahingehend hinaus, daß sie auch gutartig indeterministisch und wundersam sein können. (Es stellt sich aber heraus, daß damit nicht noch mehr Objekte unterscheidbar werden.)

Bei der Entwicklung von konkreten Klassen aus abstrakten Klassen reicht es im allgemeinen aus, eine abstrakte Klasse durch eine verfeinernde statt einer verhaltensäquivalenten Klasse zu ersetzen. Verfeinerung von Klassen ist definiert

analog zu der Verhaltensäquivalenz von Klassen, nur daß das die entstehenden Anweisungen in einer Verfeinerungsrelation stehen.

Definition (*Verfeinerung von Klassen*). Es seien c, d Klassen mit Instanzen vom Typ T. Klasse c wird verfeinert durch d, geschrieben $c \sqsubseteq d$, gdw. für alle Anweisungen $p \in \mathcal{B}[x]$ mit $x : T$,

$$(var\ x := make\ c \bullet p) \quad \sqsubseteq \quad (var\ x := make\ d \bullet p) \ .$$

Für die Verfeinerungsrelation auf Klassen gelten einige einfache Ordnungseigenschaften. Die Ordnung ist reflexiv, transitiv und antisymmetrisch (partielle Ordnung). Es seien b, c, d Klassen mit Instanzen vom selben Typ:

$$c \sqsubseteq c \qquad \textit{Klassen–}\sqsubseteq\textit{–reflexiv}$$
$$(b \sqsubseteq c) \wedge (c \sqsubseteq d) \ \Rightarrow \ (b \sqsubseteq d) \qquad \textit{Klassen–}\sqsubseteq\textit{–transitiv}$$
$$(c \sqsubseteq d) \wedge (d \sqsubseteq c) \ \Rightarrow \ (c \equiv d) \qquad \textit{Klassen–}\sqsubseteq\textit{–antisymmetrisch}$$

Diese Eigenschaften folgen alle aus der Definition der Klassenverfeinerung und den entsprechenden Eigenschaften von Prädikatentransformern.

Die Transitivität ermöglicht die schrittweise Verfeinerung von Klassen: Ein Programm kann zunächst mit Hilfe einer abstrakten Klasse formuliert werden. Später kann diese Klasse durch eine verfeinernde ersetzt werden. Die Transitivität ermöglicht, diesen Schritt solange fortzusetzen, bis das Programm effizient genug ist.

Die Verfeinerungsordnung hat ein kleinstes und ein größtes Element. Es sei *Any Meth* die Klasse mit Methoden *Meth* (), die alle *abort* sind (mit $Meth : * \to *$). Es sei *None Meth* die Klasse mit Methoden *Meth* (), die alle *miracle* sind:

$$Any\ Meth \ = \ class\ m1\ (v1 : V1)\ r1 : R1 = abort, \ldots\ end$$
$$None\ Meth \ = \ class\ m1\ (v1 : V1)\ r1 : R1 = miracle, \ldots\ end$$

Dann gilt für beliebige Klasse c : *Class Attr Meth*

$$Any\ Meth \ \sqsubseteq \ c \qquad \sqsubseteq\textit{–Any}$$
$$c \ \sqsubseteq \ None\ Meth \qquad \sqsubseteq\textit{–None}$$

Verfeinerung und Verhaltensäquivalenz von Klassen wie sie hier definiert sind, sind schwierig nachzuweisen: sie würden eine Induktion über die Struktur der

beobachtenden Programme erfordern. Der nächste Abschnitt gibt, als erster Schritt, eine einfachere Möglichkeit mittels einer Simulation. Weil die Verfeinerung von Klassen zu einer schwächeren Beweisobligation führt als die Verhaltensäquivalenz, wird im folgenden nur noch Verfeinerung betrachtet.

5.2 Klassenverfeinerung mittels Simulation

Satz (*Klassenverfeinerung mittels Simulation*). Es seien Klassen c, d gegeben durch

$$
\begin{array}{ll}
c = class & d = class \\
\quad cattr : C := cinit; & \quad dattr : D := dinit; \\
\quad m1\ (v1 : V1)\ r1 : R1 = c1, & \quad m1\ (v1 : V1)\ r1 : R1 = d1, \\
\quad \ldots & \quad \ldots \\
\quad mn\ (vn : Vn)\ rn : Rn = cn & \quad mn\ (vn : Vn)\ rn : Rn = dn \\
end & end
\end{array}
$$

Falls $sim : Tran\ (cattr : C)\ (dattr : D)$ ein konjunktiver Prädikatentransformer ist mit

(1) $enter\ cattr := cinit\ ;\ sim \sqsubseteq enter\ dattr := dinit$

(2) $c1\ ;\ sim \sqsubseteq sim\ ;\ d1,$
$\ldots$
$cn\ ;\ sim \sqsubseteq sim\ ;\ dn$

dann gilt $c \sqsubseteq d$.

Punkt (1) besagt, daß die initialen Werte der privaten Attribute der beiden Klassen durch *sim* in Beziehung zu stehen haben. Punkt (2) besagt, daß die entsprechenden Methoden der beiden Klassen die Beziehung *sim* zwischen ihren Attributen aufrecht erhalten müssen.

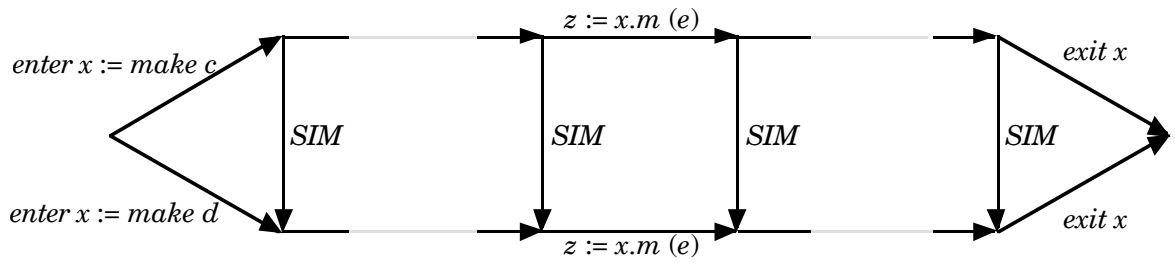
Beweis. Es seien *coper*, *doper*, *CAttr*, *DAttr*, *Meth* so, daß

$$
\begin{array}{ll}
c = ((cattr \mapsto cinit), coper)\ , & d = ((dattr \mapsto dinit), doper)\ , \\
c : Class\ CAttr\ Meth\ , & d : Class\ DAttr\ Meth\ .
\end{array}
$$

Weiterhin sei *SIM* ein konjunktiver Prädikatentransformer, definiert durch:

$$SIM = push\ x\ by\ CAttr, coper\ ;\ sim\ ;\ pop\ x\ by\ DAttr, doper$$

Die Rolle von *SIM* wird durch folgendes Diagramm veranschaulicht:



Die oberen Pfeile von der linken zur rechten Ecke symbolisieren die Ausführung von $(var\ x := make\ c \bullet p)$ für $p \in \mathcal{B}[x]$. Die unteren Pfeile symbolisieren die Ausführung von $(var\ x := make\ d \bullet p)$ für dasselbe p. Mit der Anweisung *SIM* kann x als Instanz von c zu jedem Zeitpunkt der Ausführung in eine Instanz von d transformiert werden. Der Beweis des Satzes gliedert sich in zwei Teile:

(a) $enter\ x := make\ c\ ;\ SIM \sqsubseteq enter\ x := d$

(b) $p\ ;\ SIM \sqsubseteq SIM\ ;\ p$ für alle $p \in \mathcal{B}[x]$

Damit gilt für alle $p \in \mathcal{B}[x]$:

$(var\ x := make\ c \bullet p)$

$=$ «Variablendeklaration»

$enter\ x := make\ c\ ;\ p\ ;\ exit\ x$

$\sqsubseteq$ «Sequentielle Komposition mit exit»

$enter\ x := make\ c\ ;\ p\ ;\ SIM\ ;\ exit\ x$

$\sqsubseteq$ «(b)»

$enter\ x := make\ c\ ;\ SIM\ ;\ p\ ;\ exit\ x$

$\sqsubseteq$ «(a)»

$enter\ x := make\ d\ ;\ p\ ;\ exit\ x$

$=$ «Variablendeklaration»

$(var\ x := make\ d \bullet p)$

Laut der Definition der Klassenverfeinerung bedeutet dies $c \sqsubseteq d$.

Beweis von (a).

$enter\ x := make\ c\ ;\ SIM$

$=$ «*SIM*»

enter x := *make c* ; *push x by CAttr*, *coper* ; *sim* ; *pop x by DAttr*, *doper*
= *push*, Sequentielle Komposition von Zuweisungen (3)»
enter cattr := *cinit* ; *sim* ; *pop x by DAttr*, *doper*
= «(1)»
enter dattr := *dinit* ; *pop x by DAttr*, *doper*
⊑ «*pop*, *make*»
enter x := *make d*

Beweis von (b). Der Beweis erfolgt per Induktion über den Aufbau von *p*.

- $p = z := x.m\ (e)$ falls *m* Methode von *x* und *x* nicht in *e* und *z* vorkommt.

Zu zeigen ist:

$$z := x.m\ (e)\ ;\ SIM \ \sqsubseteq\ SIM\ ;\ z := x.m\ (e)$$

Die folgenden Schritte dienen dazu, die Methodenaufrufe in die Form $[x\ is\ c]\ ;\ z := x.m\ (e)$ zu transformieren um ihre Auflösung zu ermöglichen:

$z := x.m\ (e)\ ;\ SIM \ \sqsubseteq\ SIM\ ;\ z := x.m\ (e)$
⇔ «*SIM,* is–push »
$z := x.m\ (e)\ ;\ SIM \ \sqsubseteq\ [x\ is\ c]\ ;\ SIM\ ;\ z := x.m\ (e)$
⇔ «für beliebige *b, q, r* : $q \sqsubseteq [b]\ ;\ r \quad gdw \quad [b]\ ;\ q \sqsubseteq [b]\ ;\ r$»
$[x\ is\ c]\ ;\ z := x.m\ (e)\ ;\ SIM \ \sqsubseteq\ [x\ is\ c]\ ;\ SIM\ ;\ z := x.m\ (e)$
⇔ «*SIM,* is–push, is–pop»
$[x\ is\ c]\ ;\ z := x.m\ (e)\ ;\ SIM \ \sqsubseteq\ SIM\ ;\ [x\ is\ d]\ ;\ z := x.m\ (e)$

Der Beweis wird durch Transformation der linken Seite weitergeführt:

$[x\ is\ c]\ ;\ z := x.m\ (e)\ ;\ SIM$
= «Auflösen von Methodenaufrufen»
push x by CAttr, *coper* ; $z := (coper.m)\ (e)$; *pop x by DAttr*, *doper* ; *SIM*
= «*SIM,* Aufheben von push und pop»
push x by CAttr, *coper* ; $z := (coper.m)\ (e)$; *sim* ; *pop x by DAttr*, *doper*
⊑ «Lemma unten: $z := (coper.m)\ (e)\ ;\ sim \sqsubseteq sim\ ;\ z := (doper.m)\ (e)$ »
push x by CAttr, *coper* ; *sim* ; $z := (doper.m)\ (e)$; *pop x by DAttr*, *doper*
= «Aufheben von push und pop, *SIM*»
SIM ; *push x by DAttr*, *doper* ; $z := (doper.m)\ (e)$; *pop x by DAttr*, *doper*
= «Auflösen von Methodenaufrufen»
SIM ; $[x\ is\ d]\ ;\ z := x.m\ (e)$

Das oben benutzte Lemma wird wie folgt bewiesen:

$z := (coper.m)\ (e)\ ;\ sim$
$=$ «Prozeduraufruf»
$enter\ v := e\ ;\ coper.m\ ;\ z := r\ ;\ exit\ r\ ;\ sim$
$=$ «x kommt nicht in z, r vor, Zustandstransformation–kommutativ»
$enter\ v := e\ ;\ coper.m\ ;\ sim\ ;\ z := r\ ;\ exit\ r$
$\sqsubseteq$ «(2)»
$enter\ v := e\ ;\ sim\ ;\ doper.m\ ;\ z := r\ ;\ exit\ r$
$=$ «x kommt nicht in v, e vor, Zustandstransformation–kommutativ»
$sim\ ;\ enter\ v := e\ ;\ doper.m\ ;\ z := r\ ;\ exit\ r$
$=$ «Prozeduraufruf»
$sim\ ;\ z := (doper.m)\ (e)$

Dies beendet den ersten Fall des Induktionsbeweises.

- $p = \langle\phi\rangle$ für $\phi : S \to T$ und x nicht Feld von S, T

Zu zeigen ist $\langle\phi\rangle\ ;\ SIM \ \sqsubseteq\ SIM\ ;\ \langle\phi\rangle$:

$\langle\phi\rangle\ ;\ SIM$
$=$ «$\langle\phi\rangle : Tran\ S\ T$, Zustandstransformation–kommutativ»
$SIM\ ;\ \langle\phi\rangle$

- $p = \{\beta\}$ oder $p = [\beta]$ für $\beta : S \to Bool$ und x nicht Feld von S

Zu zeigen ist $p\ ;\ SIM \ \sqsubseteq\ SIM\ ;\ p$:

$p\ ;\ SIM$
$\sqsubseteq$ «SIM konjunktiv, konjunktiv–kommutativ»
$SIM\ ;\ p$

- $p = q\ ;\ r$

Zu zeigen ist $q\ ;\ r\ ;\ SIM \ \sqsubseteq\ SIM\ ;\ q\ ;\ r$ mit Hilfe der Induktionsannahmen $q\ ;\ SIM \ \sqsubseteq\ SIM\ ;\ q$ und $r\ ;\ SIM \ \sqsubseteq\ SIM\ ;\ r$:

$q\ ;\ r\ ;\ SIM$
$\sqsubseteq$ «Induktionsannahme»
$q\ ;\ SIM\ ;\ r$
$\sqsubseteq$ «Induktionsannahme»
$SIM\ ;\ q\ ;\ r$

- $p = (\sqcap\, i \in I \bullet q)$ oder $p = (\sqcup\ i \in I \bullet q)$ für durch $i \in I$ indiziertes q.

Zu zeigen ist $(\sqcap\, i \in I \bullet q)\ ;\ SIM \sqsubseteq SIM\ ;\ (\sqcap\, i \in I \bullet q)$ und Analoges für $(\sqcup\, i \in I \bullet q)$ mit der Induktionsannahme $q\ ;\ SIM \sqsubseteq SIM\ ;\ q$:

$(\sqcap\, i \in I \bullet q)\ ;\ SIM$
$=$ «$\sqcap$–;–linksdistributiv»
$(\sqcap\, i \in I \bullet q\ ;\ SIM)$
$\sqsubseteq$ «Induktionsannahme»
$(\sqcap\, i \in I \bullet SIM\ :\ q)$
$=$ «*sim* konjunktiv, $\sqcap$–;–rechtsdistributiv»
$SIM\ ;\ (\sqcap\, i \in I \bullet q)$

bzw.

$(\sqcup\, i \in I \bullet q)\ ;\ SIM$
$=$ «$\sqcup$–;–linksdistributiv»
$(\sqcup\, i \in I \bullet q\ ;\ SIM)$
$\sqsubseteq$ «Induktionsannahmen»
$(\sqcup\, i \in I \bullet SIM\ ;\ q)$
$\sqsubseteq$ «$\sqcup$–;–rechtsdistributiv»
$SIM\ ;\ (\sqcup\, i \in I \bullet q)$

Dies beendet den Beweis der Klassenverfeinerung durch Simulation.

6 Methodik der Klassenverfeinerung

Das Thema dieses Kapitels ist die systematische Konstruktion von verfeinernden Klassen. Dazu wird Klassenverfeinerung auf Datenverfeinerung der einzelnen Methoden zurückgeführt. Die Techniken der Datenverfeinerung lassen sich dann zur Ableitung der Methoden der verfeinernden Klassen einsetzen. Es werden zwei Techniken von unterschiedlicher Allgemeinheit und Komplexität betrachtet. Dies sind die Klassenverfeinerung mittels Relation und Klassenverfeinerung mittels Abstraktionsfunktion mit Invariante.

6.1 Klassenverfeinerung mittels Relation

Bei der Klassenverfeinerung mittels Relation werden die Attribute der beiden Klassen durch eine (binäre) Relation in Beziehung gesetzt.

Satz (*Klassenverfeinerung mittels Relation*). Es seien Klassen c, d gegeben durch:

$$
\begin{array}{ll}
c = class & d = class \\
\quad cattr : C := cinit; & \quad dattr : D := dinit; \\
\quad m1\ (v1 : V1)\ r1 : R1 = c1, & \quad m1\ (v1 : V1)\ r1 : R1 = d1, \\
\quad \ldots & \quad \ldots \\
\quad mn\ (vn : Vn)\ rn : Rn = cn & \quad mn\ (vn : Vn)\ rn : Rn = cn \\
end & end
\end{array}
$$

Falls $rel : C \times D \to Bool$ eine Relation ist mit

(1) $rel\ (cinit, dinit)$

(2) $c1 \sqsubseteq_{rel} d1$
$\ldots$
$cn \sqsubseteq_{rel} dn$

gilt $c \sqsubseteq d$.

Beweis. Der Beweis erfolgt durch Zurückführen auf Klassenverfeinerung mittels Simulation. Es sei *sim* definiert durch

$$sim = cattr \rightarrow dattr \,:[dattr' \bullet rel\ (cattr, dattr')] \ .$$

Offensichtlich ist *sim* konjunktiv. Punkt (1) der Klassenverfeinerung mittels Simulation gilt wegen:

enter cattr := *cinit* ; *sim*
= «*sim*, Sequentielle Komposition mit Zuweisungen (3)»
enter dattr :[*dattr*' • *rel* (*cinit*, *dattr*')]
= «Verfeinerung von Zuweisungen (3), (1)»
enter dattr := *dinit*

Punkt (2) der Klassenverfeinerung durch Simulation folgt unmittelbar aus der Definition von *sim* und $\sqsubseteq_{rel}$. Damit folgt $c \sqsubseteq d$.

Beispiel (*Mittelwertberechnung, Fortsetzung*). Es soll die Klasse *MeanCalc* verfeinert werden.

MeanCalc =
class
b : *bag of Int* := $\langle\rangle$;
add (i : *Int*) = $b := b + \langle i \rangle$,
mean () m : *Int* = $m :=$ *sum b div #b*
end

Statt eine Verfeinerung vorzuschlagen und diese dann zu verifizieren, soll *MeanCalc*' systematisch hergeleitet werden. Dazu wird festgelegt, daß die Attribute s : *Int* und n : *Int* sind, die der laufenden Summe bzw. Anzahl entsprechen. Die Relation zwischen diesen Attributen und dem Attribut b von *MeanCalc* ist:

$$rel\ (b, s, n) \ \Leftrightarrow \ (s = sum\ b) \wedge (n = \#b)$$

(1) Für die Initialisierung von s, n gilt:

$$
\begin{array}{ll}
 & rel\ (\langle\rangle, s, n) \\
\Leftrightarrow & \text{«}rel\text{»} \\
 & (s = sum\ \langle\rangle) \wedge (n = \#\langle\rangle) \\
\Leftrightarrow & \text{«}sum,\ \#\text{»} \\
 & (s = 0) \wedge (n = 0)
\end{array}
$$

(2) Die Datenverfeinerung von *add* mittels *rel* ist:

$$
\begin{array}{ll}
 & b := b + \langle i\rangle \\
\sqsubseteq_{rel} & \text{«Datenverfeinerung von Zuweisungen (2)»} \\
 & s, n :[s', n' \bullet (\forall\ b \bullet rel\ (b, s, n) \Rightarrow rel\ (b + \langle i\rangle, s', n'))] \\
= & \text{«}rel, sum, \#\text{»} \\
 & s, n :[s', n' : (\forall\ b \bullet rel\ (b, s, n) \Rightarrow (s' = sum\ b + i) \wedge (n' = \#b + 1)] \\
\sqsubseteq & \text{«}rel\text{, Verfeinerung von Zuweisungen (1)»} \\
 & s, n :[s', n' \bullet (s' = s + i) \wedge (n' = n + 1)] \\
= & \text{«Verfeinerung von Zuweisungen (3)»} \\
 & s, n := s + i,\ n + 1
\end{array}
$$

(3) Die Datenverfeinerung von *mean* mittels *rel* ist:

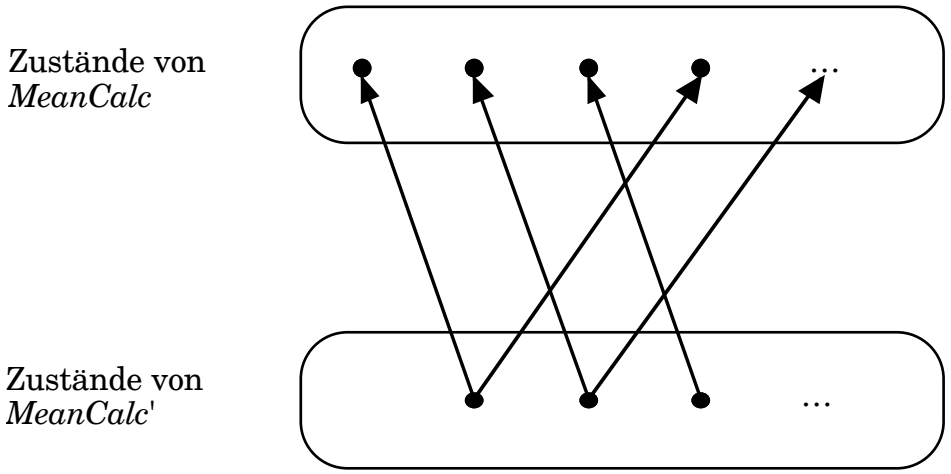
$$
\begin{array}{ll}
 & m := sum\ b\ div\ \#b \\
\sqsubseteq_{rel} & \text{«Datenverfeinerung von Zuweisungen (4)»} \\
 & m :[m' \bullet (\forall\ b \bullet rel\ (b, s, n) \Rightarrow (m' = sum\ b\ div\ \#b))] \\
= & \text{«}rel\text{»} \\
 & m :[m' \bullet (\forall\ b \bullet (s = sum\ b) \wedge (n = \#b) \Rightarrow (m' = sum\ b\ div\ \#b))] \\
\sqsubseteq & \text{«Verfeinerung von Zuweisungen (3)»} \\
 & m := s\ div\ n
\end{array}
$$

Zusammengefaßt ergibt dies die Klasse *MeanCalc*', die *MeanCalc* verfeinert:

$$
\begin{array}{l}
MeanCalc' \mathrel{\widehat{=}} \\
\quad class \\
\qquad s : Int := 0, \\
\qquad n : Int := 0; \\
\qquad add\ (i : Int) \quad = \quad s, n := s + i, n + 1, \\
\qquad mean\ ()\ m : Int \quad = \quad m := s\ div\ n \\
\quad end
\end{array}
$$

Die Eigenschaft dieser Verfeinerung (und die Idee für die Verbesserung der Effizienz) ist, daß die verfeinernde Klasse nicht alle Informationen der verfeinerten Klasse mit sich führt, sondern nur so viel Information, um korrekte Beobachtun-

gen zu ermöglichen. Dies äußert sich dadurch, daß mehrere Zustände der Klasse *MeanCalc* einem Zustand der Klasse *MeanCalc*' entsprechen, wie durch nachfolgendes Diagramm symbolisiert. Diese Art von Verfeinerungen ist mit dem Ansatz in (America, 1987) nicht möglich. Dort muß ein Zustand der verfeinernden Klasse höchstens einem Zustand der verfeinerten Klassen entsprechen.



Im allgemeinen ist eine beliebige Relation zwischen den konkreten und abstrakten Zuständen der Klassen möglich.

6.2 Klassenverfeinerung mittels Abstraktionsfunktion und Invariante

In vielen Fällen ist die Relation für die Datenverfeinerung *funktional* in dem Sinne, daß ein konkreter Zustand höchstens einem abstrakten Zustand entspricht. Dies ist auch der Fall, der in (America, 1987; Hoare, 1972; Meyer, 1988) betrachtet wird. Hierbei ergeben sich vereinfachende Regeln zum Ableiten der konkreten Operationen aus den abstrakten Operationen.

Definition (*Funktionale Datenverfeinerung,* $\sqsubseteq_{abs,inv}$). Es sei p ein Prädikatentransformer über $(v : V)$ und q ein Prädikatentransformer über $(w : W)$ und es sei $abs : W \to U$, $inv : W \to Bool$. Dann ist $p \sqsubseteq_{abs,inv} q$ definiert durch

$$p \sqsubseteq_{abs,inv} q \quad \widehat{=} \quad p \sqsubseteq_{rel} q \quad \text{mit} \quad rel\ (v, w) \Leftrightarrow (w = abs\ w) \wedge inv\ w\ .$$

Für $\sqsubseteq_{abs,inv}$ gelten analoge Distributivregeln bezüglich ; , $\sqcap$ und $\sqcup$ wie für $\sqsubseteq_{rel}$; sie werden hier nicht gesondert aufgeführt. Die interessanten Vereinfachungen ergeben sich für die Datenverfeinerung von Zuweisungen.

Satz (*Funktionale Datenverfeinerung von Zuweisungen*). Es sei $abs : W \to U$ und $inv : W \to Bool$. Dann gilt:

(1) $v :[v' \bullet b] \quad \sqsubseteq_{abs,inv} \quad w :[w' \bullet inv\ w \Rightarrow (b[abs\ w/v, abs\ w'/v'] \wedge inv\ w')]$

(2) $v := e \quad \sqsubseteq_{abs,inv} \quad w :[w' \bullet inv\ w \Rightarrow (abs\ w' = e[abs\ w/v'] \wedge inv\ w')]$

(3) $u :[u' \bullet c] \quad \sqsubseteq_{abs,inv} \quad u :[u' \bullet inv\ w \Rightarrow c[abs\ w/v]]$

(4) $u := f \quad \sqsubseteq_{abs,inv} \quad u :[u' \bullet inv\ w \Rightarrow u' = f[abs\ w/v])]$

Beweis. Dieser Satz folgt unmittelbar aus *Datenverfeinerung von Zuweisungen mittels Relation*.

Satz. Es seien p, p' Prädikatentransformer über $(u : U, v : V)$ und q, q' Prädikatentransformer über $(u : U, w : W)$. Es sei $abs : W \to U$ und $inv : W \to Bool$. Dann gilt:

(1) $[b] \quad \sqsubseteq_{abs,inv} \quad [inv\ w \Rightarrow b[abs\ w/v]]$ $\qquad \sqsubseteq_{abs,inv}$–*Annahme*

(2) $\{b\} \quad \sqsubseteq_{abs,inv} \quad \{inv\ w \wedge b[abs\ w/v]\}$ $\qquad \sqsubseteq_{abs,inv}$–*Zusicherung*

(3) *if b then p else p' end* $\sqsubseteq_{abs,inv}$ *if c then q else q' end* $\qquad \sqsubseteq_{abs,inv}$–*if*
falls $p \sqsubseteq_{abs,inv} q$ und $p' \sqsubseteq_{abs,inv} q'$
und $inv\ w \wedge b[abs\ w/v] \leq c \leq inv\ w \Rightarrow b[abs\ w/v]$

(4) *while b do p end* $\sqsubseteq_{abs,inv}$ *while c do q end* $\qquad \sqsubseteq_{abs,inv}$–*while*
falls $p \sqsubseteq_{abs,inv} q$
und $inv\ w \wedge b[abs\ w/v] \leq c \leq inv\ w \Rightarrow b[abs\ w/v]$

Beweis. Dieser Satz folgt unmittelbar aus dem entsprechenden Satz für $\sqsubseteq_{rel}$.

Satz (*Funktionale Klassenverfeinerung*). Es seien die Klassen c, d gegeben durch:

$c = class$	$d = class$
$\quad cattr : C := cinit;$	$\quad dattr : D := dinit;$
$\quad m1\ (v1 : V1)\ r1 : R1 = c1,$	$\quad m1\ (v1 : V1)\ r1 : R1 = d1,$
$\quad \ldots$	$\quad \ldots$
$\quad mn\ (vn : Vn)\ rn : Rn = cn$	$\quad mn\ (vn : Vn)\ rn : Rn = cn$
end	end

Falls $abs : D \to C$ und $inv : D \to Bool$ Funktionen sind mit

(1) $(cinit = abs\ dinit) \wedge inv\ \ dinit$

(2) $c1 \sqsubseteq_{abs,inv} d1$

$\ldots$

$cn \sqsubseteq_{abs,inv} dn$

gilt $c \sqsubseteq d$.

Beweis. Der Satz folgt mit der Definition von $\sqsubseteq_{abs,inv}$ direkt aus *Klassenverfeinerung mittels Relation*.

Beispiel (*Menge als Bitvektoren*). Es werden Mengen von ganzen Zahlen aus dem Intervall [0, *max*) betrachtet. Operationen darauf sind

- Einfügen einer Zahl zwischen 0 und *max* - 1;
- Testen, ob eine Zahl zwischen 0 und *max* - 1 enthalten ist;
- Bestimmen der Anzahl der Elemente der Menge.

Dies wird durch die Klasse *SmallIntSet* spezifiziert:

$$
\begin{array}{l}
val\ SmallIntSet\ (max : Int) = \\
\quad class \\
\qquad s : set\ of\ Int := \{\}; \\
\qquad insert\ (i : Int) \quad = \quad [0 \le i < max]\ ;\ s := s \cup \{i\}, \\
\qquad has\ (i : Int)\ r : Bool \quad = \quad [0 \le i < max]\ ;\ r := i \in s, \\
\qquad size\ ()\ r : Int \quad = \quad r := \#s \\
\quad end
\end{array}
$$

Diese Klasse soll verfeinert werden durch die Technik der *Bitvektoren*. Für die verfeinernde Klasse wird das Attribut, die Abstraktionsfunktion und die Invariante festgelegt:

$$
\begin{array}{l}
v : array\ max\ of\ Bool \\
abs\ v \ = \ \{0 \le i < max \mid v\ [i]\} \\
inv\ v \ = \ true
\end{array}
$$

(1) Die Bedingung für die Initialisierung von v ist:

$(\{\} = abs\ v) \wedge inv\ v$

$\Leftrightarrow$ «abs, inv»

$\{\} = \{0 \le i < max \mid v\ [i]\}$

$\Leftarrow$ «Logik»

$v = [false]^{max}$

(2) Die Datenverfeinerung von *insert* mittels abs, inv ist:

$[0 \le i < max]\ ;\ s := s \cup \{i\}$

$\sqsubseteq_{abs,inv}$ «Distributivität von $\sqsubseteq_{abs,inv}$, Funktionale Datenverfeinerung von Zuweisungen (2)»

$[0 \le i < max]\ ;\ v :[v' \bullet abs\ v' = abs\ v \cup \{i\}]$

$=$ «Sequentielle Komposition mit Annahmen (1), abs»

$v :[v' \bullet (0 \le i < max) \wedge (\{0 \le j < max \mid v'[j]\} = \{0 \le j < max \mid v'[j]\} \cup \{i\})]$

$\sqsubseteq$ «Verfeinerung von Zuweisungen (1)»

$v :[v' \bullet (0 \le i < max) \wedge (v' = v[i \mapsto true])]$

$=$ «Zuweisung an Feldkomponenten»

$v[i] := true$

(3) Die Datenverfeinerung von *has* mittels abs, inv ist:

$[0 \le i < max]\ ;\ r := i \in s$

$\sqsubseteq_{abs,\ inv}$ «Distributivität von $\sqsubseteq_{abs,\ inv}$, Funktionale Datenverfeinerung von Zuweisungen (4)»

$[0 \le i < max]\ ;\ r :[r' \bullet r' = i \in abs\ v]$

$=$ «Sequentielle Komposition mit Annahmen (1), abs»

$r :[r' \bullet (0 \le i < max) \wedge (r' = v[i])]$

$=$ «Zuweisung an Feldkomponenten»

$r := v[i]$

(4) Die Datenverfeinerung von *size* mittels abs, inv ist:

$r := \#s$

$\sqsubseteq_{abs,\ inv}$ «Funktionale Datenverfeinerung von Zuweisungen (4)»

$r :[r' \bullet r' = \#(abs\ v)]$

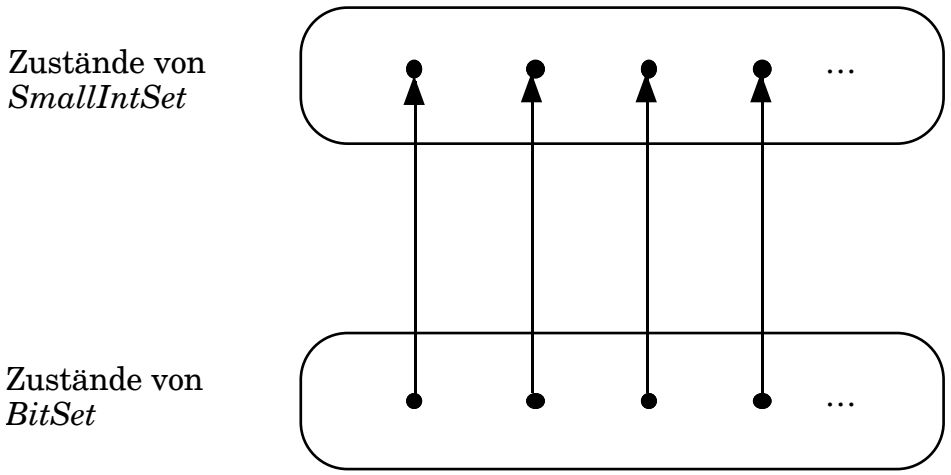
$=$ «abs»

$r := \#\{0 \le j < max \mid v[j]\}$

Das Ergebnis wird zu der Klasse *BitSet* zusammengefaßt:

```
val BitSet (max : Int) =
    class
        v : array max of Bool := [false]^max;
        insert (i : Int)          =  v[i] := true,
        has (i : Int) r : Bool    =  r := v[i],
        size () r : Bool          =  r := #{0 ≤ i < max | v[i]}
    end
```

Bei der Verfeinerung von *SmallIntSet* durch *BitSet* entspricht ein Zustand der einen Klasse genau einem Zustand der anderen Klasse. In diesem Fall liefert die Herleitung der Methoden ein eindeutiges Ergebnis. Im allgemeinen kann ein Zustand der abstrakten Klasse mehreren Zustände der konkreten Klasse entsprechen.



7 Inkrementelle Klassenverfeinerung

Typisch für den objektorientierten Entwurf ist es, daß eine Klasse aus einer gegebenen durch inkrementelle Modifikation mittels Vererbung entsteht. Beispiele hierfür sind:

(1) eine neue Klasse *BoundedQueue* entsteht aus *Queue* (Abschnitt 4.5) durch Ändern der Methode *enqueue*;
(2) die Klasse *BitSet* (Abschnitt 6.2) wird so erweitert, daß zur Beschleunigung der Methoden die Anzahl der Elemente in einem zusätzlichen Attribut festgehalten wird;
(3) die Klasse *PriorityQueue* (Abschnitt 4.5) entsteht aus der Klasse *Queue* durch Ändern der Methode *dequeue*

In den letzten beiden Fällen ist die modifizierte Klasse eine Verfeinerung der ursprünglichen Klasse, in Fall (1) nicht. Für die *inkrementelle Klassenverfeinerung* in den Fällen (2) und (3) lassen sich spezielle, vereinfachte Konstruktionstechniken anwenden. In diesem Kapitel werden zwei Arten der inkrementellen Klassenverfeinerung vorgestellt: Bei der ersten Art hat die verfeinerte Klasse sowohl zusätzliche Attribute, als auch geänderte Methoden (Fall 2); bei der zweiten hat die verfeinerte Klasse nur geänderte Methoden (Fall 3).

7.1 Klassenverfeinerung mittels Invariante

Vorbereitend wird die *Datenverfeinerung mittels Invariante* eingeführt und Rechenregeln abgeleitet. Formal gesehen ist die Datenverfeinerung mittels Inva-

riante ein Spezialfall der Datenverfeinerung mittels Abstraktionsfunktion und Invariante, bei der die Abstraktionsfunktion eine *Projektion* ist.

Definition (*Datenverfeinerung mittels Invariante,* $\sqsubseteq_{inv}$). Es sei p ein Prädikatentransformer über $(u : U, v : V)$ und q ein Prädikatentransformer über $(u : U, v : V, w : W)$. Falls $inv : V \times W \to Bool$, ist $p \sqsubseteq_{inv} q$ definiert durch

$$p \sqsubseteq_{inv} q \;\widehat{=}\; p \sqsubseteq_{abs,inv} \quad \text{mit } abs\,(v, w) = v \;\;.$$

Für $\sqsubseteq_{inv}$ gelten analoge Distributivgesetze bezüglich ; , $\sqcap$ und $\sqcup$ wie für $\sqsubseteq_{rel,}$ und $\sqsubseteq_{abs,\,inv}$. Sie werden hier nicht gesondert aufgeführt.

Satz (*Datenverfeinerung von Zuweisungen mittels Invariante*). Es seien $u : U$, $v : V$, $w : W$ Programmvariable, es seien e, f Listen von Ausdrücken, b, c boolesche Ausdrücke und es sei $inv : V \times W \to Bool$. Dann gilt:

(1) $v :[v' \bullet b] \quad \sqsubseteq_{inv} \quad v, w :[v', w' \bullet inv\,(v, w) \Rightarrow b \wedge inv\,(v', w')]$
(2) $v := e \quad \sqsubseteq_{inv} \quad v, w :[v', w' \bullet inv\,(v, w) \Rightarrow (v' = e) \wedge inv\,(v', w')]$
(3) $u :[u' \bullet c] \quad \sqsubseteq_{inv} \quad u :[u' \bullet inv\,(v, w) \Rightarrow c]$
(4) $u := f \quad \sqsubseteq_{inv} \quad u :[u' \bullet inv\,(v, w) \Rightarrow (u' = f)]$

Beweis. Dieser Satz folgt unmittelbar aus *Funktionale Datenverfeinerung von Zuweisungen*.

Satz. Es seien p, p' Prädikatentransformer über $(u : U, v : V)$ und q, q' Prädikatentransformer über $(u : U, v : V, w : W)$. Es sei $inv : V \times W \to Bool$. Dann gilt:

(1) $[b] \quad \sqsubseteq_{inv} \quad [inv\,(v, w) \Rightarrow b]$ — $\sqsubseteq_{inv}$*–Annahmen*
(2) $\{b\} \quad \sqsubseteq_{inv} \quad \{inv\,(v, w) \wedge b\}$ — $\sqsubseteq_{inv}$*–Zusicherung*
(3) *if b then p else p' end* $\sqsubseteq_{inv}$ *if c then q else q' end* — $\sqsubseteq_{inv}$*–if*
falls $p \sqsubseteq_{inv} q$ und $p' \sqsubseteq_{inv} q'$
und $inv\,(v, w) \wedge b \leq c \leq inv\,(v, w) \Rightarrow b$
(4) *while b do p end* $\sqsubseteq_{inv}$ *while c do q end* — $\sqsubseteq_{inv}$*–while*
falls $p \sqsubseteq_{inv} q$
und $inv\,(v, w) \wedge b \leq c \leq inv\,(v, w) \Rightarrow b$

Beweis. Dieser Satz folgt unmittelbar aus dem entsprechenden Satz für $\sqsubseteq_{abs,\,inv}$.

Bei der Klassenverfeinerung mittels Invariante werden die zu einer Klasse hinzugefügten Attribute mit den bestehenden durch eine Invariante in Beziehung gesetzt. Die Initialisierung der hinzugekommenen Attribute muß die Invariante erfüllen. Die Methoden müssen so geändert werden, daß sie die Invariante aufrechterhalten.

Satz (*Klassenverfeinerung mittels Invariante*). Es seien $m1$, … mi …mn Methodennamen und es seien Klassen c, d gegeben durch:

$$
\begin{array}{ll}
c = class & d = class\ (c) \\
\quad cattr : C := cinit; & \quad dattr : D := dinit; \\
\quad m1\ (v1 : V1)\ r1 : R1 = c1, & \quad m1\ (v1 : V1)\ r1 : R1 = d1, \\
\quad \ldots & \quad \ldots \\
\quad mn\ (vn : Vn)\ rn : Rn = cn & \quad mi\ (vi : Vi)\ ri : Ri = di \\
\quad end & \quad end
\end{array}
$$

Falls $inv : C \times D \to Bool$ mit

(1) $inv\ (cinit, dinit)$

(2) $c1 \sqsubseteq_{inv} d1$

$\ldots$

$ci \sqsubseteq_{inv} di$

$\ldots$

$cn \sqsubseteq_{inv} cn$

gilt $c \sqsubseteq d$.

Beweis. Der Satz folgt nach Auflösen der Vererbung direkt aus *Klassenverfeinerung mittels Abstraktionsfunktion und Invariante*.

Der Satz führt zu Beweisverpflichtungen für die geänderten und gleich gebliebenen Methoden. Umgekehrt läßt sich der Satz anwenden, um zu bestimmen, welche Methoden geändert werden müssen, und welche gleich bleiben können. Dies zeigt das nachfolgende Beispiel.

Beispiel (*Mengen als Bitvektoren, Fortsetzung*). Es soll die Klasse *BitSet* so modifiziert werden, daß zur Beschleunigung der Methode *size* die laufende Anzahl

der Elemente in einem zusätzlichen Attribut festgehalten wird. Das Attribut und die Invariante werden festgelegt durch:

$$s : Int$$
$$inv\ (v, s) \;=\; (s = \#\{0 \le i < max \mid v[i]\})$$

(1) Für die Initialisierung von s gilt:

$$inv\ ([false]^{max}, s)$$
$\Leftrightarrow$ «inv»
$$s = \#\{0 \le i < max \mid false\}$$
$\Leftrightarrow$ «Logik»
$$s = 0$$

(2) Verfeinerung von *insert*:

$$v[i] := true$$
$=$ «Zuweisung an Feldkomponenten»
$$v :[v' \bullet (0 \le i < max) \wedge (v' = v[i \mapsto true])]$$
$\sqsubseteq_{inv}$ «Datenverfeinerung von Zuweisungen mittels Invariante (1)»
$$v, s :[v', s' \bullet inv\ (v, s) \Rightarrow (0 \le i < max) \wedge (v' = v[i \mapsto true]) \wedge inv\ (v', s')]$$

Zur weiteren Vereinfachung der Anweisung wird eine Fallunterscheidung mit der Bedingung $v[i]$ gemacht. Der Ausdruck in der Zuweisung vereinfacht sich unter der Annahme $v[i]$ wie folgt:

$$inv\ (v, s) \Rightarrow (0 \le i < max) \wedge (v' = v[i \mapsto true]) \wedge inv\ (v', s')$$
$\Leftarrow$ «Logik, $v[i]$ impliziert $(0 \le i < max)$»
$$(v' = v) \wedge (s' = s)$$

Unter der Annahme $\neg v[i]$ gilt:

$$inv\ (v, s) \Rightarrow (0 \le i < max) \wedge (v' = v[i \mapsto true]) \wedge inv\ (v', s')$$
$\Leftarrow$ «Logik»
$$(0 \le i < max) \wedge (v' = v[i \mapsto true]) \wedge (s' = s + 1)$$

Damit wird die Ableitung fortgesetzt:

...
$\sqsubseteq$ «bösartige Zuweisung–if, Verfeinerung von Zuweisungen (1)»
$$if\ v[i]\ then\ v, s :[v', s' \bullet (v' = v) \wedge (s' = s)]$$
$$else\ v, s :[v', s' \bullet (0 \le i < max) \wedge (v' = v[i \mapsto true]) \wedge (s' = s + 1)]\ end$$
$\sqsubseteq$ «*skip*, Seq. Komposition von bösartigen Zuweisungen (1)»

$$
\begin{array}{ll}
 & \mathit{if}\ v[i]\ \mathit{then}\ \mathit{skip} \\
 & \mathit{else}\ v :[v' \bullet (0 \le i < \mathit{max}) \wedge (v' = v[i \mapsto \mathit{true}])]\ ;\ s :[s' \bullet s' = s + 1]\ \mathit{end} \\
= & \text{«Zuweisung an Feldkomponenten, if–Negation»} \\
 & \mathit{if}\ \neg v[i]\ \mathit{then}\ v[i] := \mathit{true}\ ;\ s := s + 1\ \mathit{end}
\end{array}
$$

(3) Verfeinerung von *has*:

$$
\begin{array}{ll}
 & r := v[i] \\
\sqsubseteq_{inv} & \text{«Datenverfeinerung von Zuweisungen mittels Invariante (4)»} \\
 & r :[r' \bullet \mathit{inv}\ (v, s) \Rightarrow (r' = v[i]) \wedge (0 \le i < \mathit{max})] \\
\sqsubseteq & \text{«Verfeinerung von Zuweisungen (3)»} \\
 & r := v[i]
\end{array}
$$

Bei der Verfeinerung von *has* wird von der Invariante kein Gebrauch gemacht.

(4) Verfeinerung von *size*:

$$
\begin{array}{ll}
 & r := \#\{0 \le i < \mathit{max} \mid v[i]\} \\
\sqsubseteq_{inv} & \text{«Datenverfeinerung von Zuweisungen mittels Invariante (4)»} \\
 & r :[r' \bullet \mathit{inv}\ (v, s) \Rightarrow (r' = \#\{0 \le i < \mathit{max} \mid v[i]\})] \\
= & \text{«Logik, } \mathit{inv}\text{»} \\
 & r :[r' \bullet \mathit{inv}\ (v, s) \Rightarrow (r' = s)] \\
\sqsubseteq & \text{«Verfeinerung von Zuweisungen (3)»} \\
 & r := s
\end{array}
$$

Die Datenverfeinerung der Methoden zeigt, daß sich *insert* und *size* ändern und *has* gleich bleibt. Das Ergebnis wird zu der Klasse *BitSetCount* zusammengefaßt:

$$
\begin{array}{l}
\mathit{val}\ \mathit{BitSetCount} = \\
\quad \mathit{class}\ (\mathit{BitSet}) \\
\qquad s : \mathit{Int} := 0: \\
\qquad \mathit{insert}\ (i : \mathit{Int}) \quad = \quad \mathit{if}\ \neg v[i]\ \text{then}\ v[i] := \mathit{true}\ ;\ s := s + 1\ \mathit{end}, \\
\qquad \mathit{size}\ ()\ r : \mathit{Bool} \quad = \quad r := s \\
\quad \mathit{end}
\end{array}
$$

7.2 Klassenverfeinerung durch Methodenverfeinerung

Der einfachste Fall ist der, daß Methoden einzeln verfeinert werden und die Attribute gleich bleiben.

Satz (*Klassenverfeinerung durch Methodenverfeinerung*). Es seien $m1$, ..., mi, ...mn Methodennamen und es seien Klassen c, d gegeben durch:

$$
\begin{array}{ll}
c = class & d = class\,(c) \\
\quad cattr : C := cinit; & \quad m1\ (v1 : V1)\ r1 : R1 = d1, \\
\quad m1\ (v1 : V1)\ r1 : R1 = c1, & \quad \ldots \\
\quad \ldots & \quad mi\ (vi : Vi)\ ri : Ri = di \\
\quad mn\ (vn : Vn)\ rn : Rn = cn & \ end \\
\ end &
\end{array}
$$

Falls $c1 \sqsubseteq d1$, ... $ci \sqsubseteq di$ gilt $c \sqsubseteq d$.

Beweis. Der Satz folgt unmittelbar aus *Klassenverfeinerung mittels Invariante* mit Invariante *true*.

Beispiel (*Warteschlangen, Fortsetzung*). Es soll gezeigt werden, daß

$$Queue[Item] \quad \sqsubseteq \quad PriorityQueue[Item] \quad \text{für alle } Item \leq OrderedItem \ .$$

Laut obigem Satz ist für *dequeue* zu zeigen:

$$
\begin{array}{l}
i :[i' \bullet i' \in q]\ ;\ q := q \text{ - } \langle i \rangle \sqsubseteq \\
\quad i :[i \bullet (i' \in q) \wedge (\forall\ x \in q \bullet i'.key \leq x.key)]\ ;\ q := q \text{ - } \langle i \rangle
\end{array}
$$

Dies folgt aus ;–*monoton* und der Verfeinerung von Zuweisungen (1).

8 Klassenschnitt und Klassenvereinigung

Häufig werden beim objektorientierten Entwurf Klassen gefunden, die ähnliches, aber kein identisches Verhalten beschreiben. In diesen Fällen kann das gemeinsame Verhalten zu einer abstrakten Klasse *herausfaktorisiert* werden, die von den gefundenen verfeinert wird. Hierfür gibt es mehrere Gründe: (1) die Ähnlichkeit der gefundenen Klassen wird explizit dokumentiert; (2) die Beschreibung der Klassen wird kompakter, falls von der abstrakten Klasse geerbt wird; (3) die abstrakte Klasse kann z.B. zur Spezifikation von Parametern verwendet werden, wenn nur die Eigenschaften dieser Klasse benötigen benötigt — somit wird eine *Überspezifikation* vermieden und *Wiederverwendung* gefördert.

Eine duale Situation ist, daß von zwei oder mehreren gegebenen Klassen eine konkretere Klasse gesucht wird, die diese verfeinert. In dieser, wie in der vorhergehenden Situation, stellt sich die Frage nach der Existenz gemeinsamer verfeinerter oder verfeinernden Klassen.

In diesem Kapitel werden zwei Operationen auf Klassen definiert, Schnitt und Vereinigung. Damit können gemeinsam verfeinerte bzw. verfeinernde Klassen konstruiert werden. An einem Beispiel von Bankkonten wird deren Anwendung demonstriert.

8.1 Definition und Eigenschaften von Schnitt und Vereinigung

Definition (*Klassenschnitt und Klassenvereinigung*). Es seien Klassen c, d gegeben durch:

$$
\begin{array}{ll}
c = class & d = class \\
\quad attr : A := init, & \quad attr : A := init, \\
\quad cattr : C := cinit; & \quad dattr : D := dinit; \\
\quad m1\ (v1 : V1)\ r1 : R1 = c1, & \quad m1\ (v1 : V1)\ r1 : R1 = d1, \\
\quad \ldots & \quad \ldots \\
\quad mn\ (vn : Vn)\ rn : Rn = cn & \quad mn\ (vn : Vn)\ rn : Rn = dn \\
end & end
\end{array}
$$

Die Attribute *attr* sind die gemeinsamen Attribute von c und d, die Attribute *cattr*, und *dattr* sind von diesen verschieden. Der Schnitt $c \sqcap d$ und die Vereinigung $c \sqcup d$ besitzen die Attribute von sowohl c als auch d. Beim Schnitt werden die Methoden durch die bösartige Auswahl der entsprechenden Methoden von c und d gebildet, bei der Vereinigung durch die gutartige Auswahl:

$$
\begin{array}{ll}
c \sqcap d \mathrel{\widehat{=}} class & c \sqcup d \mathrel{\widehat{=}} class \\
\quad attr : A := init, & \quad attr : A := init, \\
\quad cattr : C := cinit, & \quad cattr : C := cinit, \\
\quad dattr : D := dinit; & \quad dattr : D := dinit; \\
\quad m1\ (v1 : V1)\ r1 : R1 = c1 \sqcap d1, & \quad m1\ (v1 : V1)\ r1 : R1 = c1 \sqcup d1, \\
\quad \ldots & \quad \ldots \\
\quad mn\ (vn : Vn)\ rn : Rn = cn \sqcap dn & \quad mn\ (vn : Vn)\ rn : Rn = cn \sqcup dn \\
end & end
\end{array}
$$

Zwei Klassen mit Instanzen vom gleichen Typ können immer in eine Form gebracht werden, so daß ihr Schnitt und ihre Vereinigung gebildet werden kann. Attribute mit gleichem Namen, aber unterschiedlichem Typ oder unterschiedlicher Initialisierung müssen zuvor in einer der beiden Klassen systematisch umbenannt werden.

Klassenschnitt und Klassenvereinigung sind idempotent, symmetrisch und assoziativ. Es seien b, c, d Klassen mit Instanzen vom gleichen Typ:

$$
\begin{array}{ll}
c \sqcap c = c & c \sqcup c = c
\end{array}
$$

$c \sqcap d = d \sqcap c$ $\qquad$ $c \sqcup d = d \sqcup c$

$(b \sqcap c) \sqcap d = b \sqcap (c \sqcap d)$ $\qquad$ $(b \sqcup c) \sqcup d = b \sqcup (c \sqcup d)$

Diese Eigenschaften folgen unmittelbar aus der Definition von Klassenschnitt bzw. -vereinigung und den entsprechenden Eigenschaften für Prädikatentransformer. Die letzten beiden Gleichungen bedeuten, daß die Reihenfolge bei der Bildung von Klassenschnitt und -vereinigung irrelevant ist. Die Ausführungen dieses Kapitels zu Schnitt und Vereinigung von zwei Klassen lassen sich somit auf mehrere Klassen übertragen.

Satz. Es seien c, d Klassen mit Instanzen vom gleichen Typ. Dann gilt:

(1) $c \sqcap d \sqsubseteq c$ und $c \sqcap d \sqsubseteq d$ *Klassen–⊓–untere Schranke*

(2) $c \sqsubseteq c \sqcup d$ und $d \sqsubseteq c \sqcup d$ *Klassen–⊔–obere Schranke*

Beweis. Es wird jeweils nur der erste Teil der Behauptung gezeigt, der zweite folgt aus Symmetriegründen. Der Beweis wird geführt durch Klassenverfeinerung mittels Simulation. Es seien die Klassen c, d, $c \sqcap d$ und $c \sqcup d$ wie oben. Für (1) sei $sim = exit\ dattr$. Zu zeigen ist:

(a) $enter\ attr, cattr, dattr := init, cinit, dinit\ ;\ sim \sqsubseteq$
$\qquad enter\ attr, cattr := init, cinit$

(b) $(c1 \sqcap d1)\ ;\ sim \sqsubseteq sim\ ;\ c1$
$\qquad \ldots$
$\qquad (cn \sqcap dn)\ ;\ sim \sqsubseteq sim\ ;\ cn$

Punkt (a) folgt aus *Sequentielle Komposition mit exit*. Für (b) gilt:

$\qquad (ci \sqcap di)\ ;\ sim$
$\sqsubseteq$ «⊓–monoton, ;–monoton»
$\qquad ci\ ;\ sim$
$=$ «Zustandstransformation–kommutativ»
$\qquad sim\ ;\ ci$

Für (2) sei $sim = enter\ dattr := dinit$. Zu zeigen ist:

(c) $enter\ attr, cattr := init, cinit\ ;\ sim \sqsubseteq$
$\qquad enter\ attr, cattr, dattr := init, cinit, dinit$

(d) $c1\ ;\ sim \sqsubseteq sim\ ;\ (c1 \sqcup d1)$
$\qquad \ldots$
$\qquad cn\ ;\ sim \sqsubseteq sim\ ;\ (cn \sqcup dn)$

Punkt (c) folgt aus *Sequentielle Komposition von Zuweisungen* (3). Für (d) gilt:

$$
\begin{array}{ll}
 & ci \,;\, sim \\
= & \text{« Zustandstransformation–kommutativ»} \\
 & sim \,;\, ci \\
\sqsubseteq & \text{« } \sqcup\text{–monoton, ;–monoton»} \\
 & sim \,;\, (ci \sqcup di)
\end{array}
$$

8.2 Ableiten von verfeinerten und verfeinernden Klassen

Klassenschnitt und Klassenvereinigung können beim Entwurf als Hilfsmittel zum Auffinden verfeinerter bzw. verfeinernder Klassen eingesetzt werden.

Beispiel (*Bankkonten*). Es soll die Verwaltung von Konten einer Bank spezifiziert werden. Grundlegende Operationen auf Bankkonten sind das Öffnen eines Bankkontos, das Ein- und Ausbezahlen von Beträgen (amount) und die Auszahlung von Zinsen. Zwei prinzipielle Arten von Bankkonten sind Sparkonten und Girokonten. Sparkonten können nicht überzogen werden, haben aber einen günstigen Zinssatz. Girokonten können überzogen werden, haben aber (vereinfacht) keinen Zins. Es sei $Rate \geq 1$ der Sparzinssatz:

$$
\begin{array}{l}
SavingAccount \mathrel{\widehat{=}} \\
\quad class \\
\qquad bal : Real := 0.0; \\
\qquad transact\ (a : Real) \;=\; \{bal + a \geq 0\} \,;\, bal := bal + a, \\
\qquad balance\ ()\ a : Real \;=\; a := bal, \\
\qquad payinterest\ () \qquad\;\; = \; bal := bal * Rate \\
\quad end \\
CurrentAccount \mathrel{\widehat{=}} \\
\quad class \\
\qquad bal : Real := 0.0; \\
\qquad transact\ (a : Real) \;=\; bal := bal + a, \\
\qquad balance\ ()\ a : Real \;=\; a := bal, \\
\qquad payinterest\ () \qquad\;\; = \; skip \\
\quad end
\end{array}
$$

Die Datenbasis der Bank besteht u. a. aus der Mengen aller Spar- und Girokonten. Es soll die Klasse *Account* abgeleitet werden, die die gemeinsamen Eigenschaften von Spar- und Girokonten hat. Sie kann dann zur Spezifikation von gemeinsamen Operationen auf Spar- und Girokonten benutzt werden, wie z. B. Drucken des Kontostandes. Dazu wird der Schnitt von *SavingAccount* und *CurrentAccount* gebildet:

$$
\begin{array}{ll}
 & \mathit{SavingAccount} \sqcap \mathit{CurrentAccount} \\
= & \text{«Klassenschnitt»} \\
 & \mathit{class} \\
 & \quad \mathit{bal} : \mathit{Real} := 0.0; \\
 & \quad \mathit{transact}\ (a : \mathit{Real}) = (\{\mathit{bal} + a \geq 0\}\ ;\ \mathit{bal} := \mathit{bal} + a) \sqcap (\mathit{bal} := \mathit{bal} + a), \\
 & \quad \mathit{balance}\ ()\ a : \mathit{Real} = (a := \mathit{bal}) \sqcap (a := \mathit{bal}), \\
 & \quad \mathit{payinterest}\ () \quad = (\mathit{bal} := \mathit{bal} * \mathit{Rate}) \sqcap \mathit{skip} \\
 & \mathit{end} \\
= & \text{«Verfeinerungskalkül»} \\
 & \mathit{class} \\
 & \quad \mathit{bal} : \mathit{Real} := 0.0; \\
 & \quad \mathit{transact}\ (a : \mathit{Real}) = \{\mathit{bal} + a \geq 0\}\ ;\ \mathit{bal} := \mathit{bal} + a, \\
 & \quad \mathit{balance}\ ()\ a : \mathit{Real} = a := \mathit{bal}, \\
 & \quad \mathit{payinterest}\ () \quad = (\mathit{bal} := \mathit{bal} * \mathit{Rate}) \sqcap \mathit{skip} \\
 & \mathit{end}
\end{array}
$$

Der Schnitt von *SavingAccount* und *CurrentAccount* ist also ein Konto, das kein Überziehen erlaubt und entweder Zinsen bezahlt oder nicht. Im Hinblick auf eine *Generalisierung* (es könnten im Laufe der Zeit weitere Kontenarten hinzukommen) und im Hinblick auf eine *Vereinfachung* ist es ratsam, für *Account* eine kleinere (verfeinerte) Klasse zu wählen, z. B.:

$$
\begin{array}{ll}
\mathit{Account} \mathrel{\widehat{=}} & \\
 & \mathit{class} \\
 & \quad \mathit{bal} : \mathit{Real} := 0.0; \\
 & \quad \mathit{transact}\ (a : \mathit{Real}) = \{\mathit{bal} + a \geq 0\}\ ;\ \mathit{bal} := \mathit{bal} + a, \\
 & \quad \mathit{balance}\ ()\ a : \mathit{Real} = a := \mathit{bal}, \\
 & \quad \mathit{payinterest}\ () \quad = \mathit{bal} :[\mathit{bal}' \bullet \mathit{bal}' \geq \mathit{bal}] \\
 & \mathit{end}
\end{array}
$$

Es gilt

$$\mathit{Account} \sqsubseteq \mathit{SavingAccount} \quad \text{und} \quad \mathit{Account} \sqsubseteq \mathit{CurrentAccount}\ .$$

Dual dazu läßt sich ein Konto bestimmen, das sowohl *SavingAccount* als auch *CurrentAccount* verfeinert. Dazu wird die Vereinigung gebildet:

$$
\begin{array}{ll}
 & \mathit{SavingAccount} \sqcup \mathit{CurrentAccount} \\
= & \text{«Klassenvereinigung»} \\
 & \mathit{class} \\
 & \quad \mathit{bal} : \mathit{Real} := 0.0; \\
 & \quad \mathit{transact}\ (a : \mathit{Real}) = (\{\mathit{bal} + a \geq 0\}\ ;\ \mathit{bal} := \mathit{bal} + a) \sqcup (\mathit{bal} := \mathit{bal} + a), \\
 & \quad \mathit{balance}\ ()\ a : \mathit{Real} = (a := \mathit{bal}) \sqcup (a := \mathit{bal}), \\
 & \quad \mathit{payinterest}\ () \quad = (\mathit{bal} := \mathit{bal} * \mathit{Rate}) \sqcup \mathit{skip} \\
 & \mathit{end} \\
= & \text{«Verfeinerungskalkül»} \\
 & \mathit{class} \\
 & \quad \mathit{bal} : \mathit{Real} := 0.0; \\
 & \quad \mathit{transact}\ (a : \mathit{Real}) = \mathit{bal} := \mathit{bal} + a, \\
 & \quad \mathit{balance}\ ()\ a : \mathit{Real} = a := \mathit{bal}, \\
 & \quad \mathit{payinterest}\ () \quad = (\mathit{bal} := \mathit{bal} * \mathit{Rate}) \sqcup \mathit{skip} \\
 & \mathit{end}
\end{array}
$$

Die Vereinigung von *SavingAccount* und *CurrentAccount* ist ein Konto, daß sich überziehen läßt und das gutartig auswählt, ob Zinsen bezahlt werden oder nicht. Offensichtlich ist dieses Konto nicht implementierbar.

Das Beispiel zeigt, wie sich Klassenschnitt und -vereinigung im objektorientierten Entwurf einsetzen lassen, um eine gemeinsame Abstraktion bzw. Konkretisierung von zwei Klassen zu finden.

Eine der Eigenschaften von Klassenschnitt und -vereinigung ist, daß sie sich immer bilden lassen, sofern die Klasse die gleiche Schnittstelle haben. (Falls Methoden mit unterschiedlichen Parametern definiert sind, macht dies keinen Sinn). Die Möglichkeit, immer Klassenschnitt und -vereinigung bilden zu können, geht zurück auf die Einbettung von Anweisungen als Prädikatentransformer in einem Verband, in dem kleinste obere und größte untere Schranken immer existieren. Der praktische Nutzen hiervon zeigt sich im Fall, wenn man unsicher ist, ob von zwei (oder mehreren) Klassen eine verfeinerte oder verfeinernde Klasse existiert: man bildet lediglich den Schnitt bzw. die Vereinigung und vereinfacht das Ergebnis. Dieses ist dann entweder implementierbar

wie bei *SavingAccount* ⊓ *CurrentAccount* oder nicht implementierbar wie bei *SavingAccount* ⊔ *CurrentAccount*.

Die Einbettung einer Struktur in einen reichhaltigeren Bereich folgt einem allgemeinen Prinzip, das aus der Mathematik bekannt ist: Aus den ganzen Zahlen sind die negativen, gebrochenen, reellen, komplexen Zahlen entstanden, um jeweils das Rechnen zu vereinfachen, dadurch daß mehr Gleichungen lösbar werden. Beispielsweise wird mit negativen Zahlen in der Architektur gerechnet, auch wenn alle Maße positive Zahlen sind. Zinseszinsrechnung wird mit reellen Zahlen gemacht, auch wenn ein Kontostand immer nur eine ganze Zahl ist. Mit dieser Motivation werden im Verfeinerungskalkül gutartig indeterministische und wundersame Programme betrachtet, auch sie machen mehr (Un-) Gleichungen lösbar.

Dieses Prinzip wird hier erstmalig auf objektorientierten Entwurf angewandt, indem gutartig indeterministische und wundersame Klassen (d. h. Klassen mit solchen Methoden) betrachtet werden und damit Klassenschnitt und Klassenvereinigung definierbar werden. Die Konsequenz dessen ist, daß das Ergebnis der Berechnung einer Klasse nicht notwendigerweise implementierbar sind. In der Analogie mit der Architektur entspricht dies dem Fall, daß die Länge einer Wand negativ ist. Beides kann nicht realisiert werden. Trotzdem ist es in beiden Fällen wichtig, nachdenken zu können, ob etwas realisierbar ist und wie die Voraussetzungen zu ändern sind, damit es realisierbar wird. Nicht implementierbare Anweisungen und Klassen können auch als Zwischenergebnis bei einer Berechnung auftreten (wie auch in der Architektur negative Zahler als Zwischenergebnis auftreten können).

9 Zusammenfassung und Ausblick

Abschließend soll der vorgestellte objektorientierte Verfeinerungskalkül eingeordnete und kritisch betrachtet werden:

- Wie praktikabel ist es, damit zu spezifizieren und verifizieren?
- Wie praktikabel ist es, Teile davon zu implementieren?

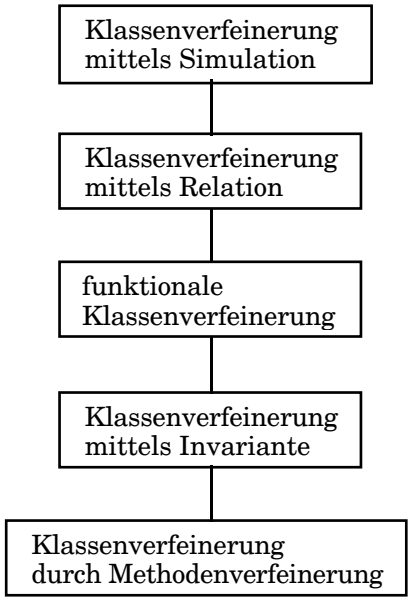
Außerdem wird ein Ausblick auf mögliche Erweiterungen gegeben.

9.1 Zusammenfassung und Einordnung

Es wurde eine objektorientierte Entwurfsnotation definiert, die es erlaubt, alle Klassen von der Analyse bis zur Implementierung zu spezifizieren (Kapitel 4). Im Gegensatz zu den üblichen Programmiersprachen können damit abstrakte Klassen vollständig spezifiziert werden.[4] Im Gegensatz zu vielen objektorientierten Spezifikationssprachen (Carrington, et al., 1990; Lano, 1991) ist eine freie Kombination von Spezifikations- mit Programmkonstrukten möglich. Der methodische Beitrag liegt in der verhaltensorientierten Definition von Klassenverfeinerung (Kapitel 5), der Hierarchie von Beweistechniken für Klassenverfeinerung mit Gewicht auf der Herleitung verfeinernder Klassen (Kapitel 6), der inkrementellen Klassenverfeinerung (Kapitel 7) und dem Herausfaktorisieren von Klassen (Kapitel 8).

[4] In der Terminologie objektorientierter Programmiersprachen ist eine abstrakte Klasse eine teilweise *undefinierte* Klasse.

Die Kombination von Verfeinerungskalkül und objektorientierter Entwurfsmethodik vereinigt die Anwendungsbereiche beider Vorgehensweisen:



- Im Verfeinerungskalkül wird zwischen der Spezifikation eines Problems und seiner Implementierung unterschieden. Die Spezifikation ist auf Kompaktheit und Verständlichkeit ausgelegt, die Implementierung auf Effizienz. Die Struktur der Implementierung ist i. d. R. verschieden von der Struktur der Spezifikation (z. B. bei Sortierverfahren). Weil die Verfeinerung der Spezifikation zu einer Implementierung ein strukturverändernder Prozeß ist, eignet sich der Ansatz für kleine, komplexere Programme, d. h. zum *Programmieren–im–Kleinen* (*programming–in–the–small*).

- In der objektorientierten Entwurfsmethodik werden die Klassen der Analyse bis zur Implementierung beibehalten. Die Struktur der Analyse ist Teil der Struktur der Implementierung. Der Ansatz eignet sich deshalb zum *Programmieren–im–Großen* (*programming–in–the–large*).

Die Kombination erlaubt, objektorientierte Entwurfsmethodik einzusetzen, um Klassen zu finden, und mit dem Verfeinerungskalkül die interne Struktur von Klassen zu manipulieren.

Die Definition der Semantik durch $F_{\leq}^{\omega}$ führt auch zu Einsichten für den Entwurf objektorientierter Sprachen. Die Motivation, ein Typsystem wie $F_{\leq}^{\omega}$ einzusetzen (im Vergleich zu untypisierten Ansätzen), ist, korrekte Typisierung entscheidbar und damit maschinell überprüfbar zu machen. Die Typisierung von Variablen wird als eine partielle Spezifikation angesehen, die eine Überprüfung auf konsistente Benutzung zuläßt. Dies hat sich in der Praxis als hilfreich bei Programmiersprachen gezeigt und setzt sich bei (maschinell unterstützten) Spezifikationssprachen wie Z und RAISE durch.

Diese Motivation für Typisierung führt zu einer Entkopplung von Objekttypen und Klassen. Folgende Tabelle faßt die wesentlichen Unterschiede zusammen:

	Objekttypen	Klassen
Rolle	Schnittstelle	Verhalten
Kapselung	ja	nein
Enthaltensein	$x : T$ (entscheidbar)	x is C (unentscheidbar)
Ordnung	$\leq$ Untertypisierung (entscheidbar)	$\sqsubseteq$ Verfeinerung (unentscheidbar)
Typparameter	ja, Art als Schranke	ja, Typ als Schranke
Wertparameter	nein	ja

Die meisten Programmiersprachen machen keine Unterscheidung zwischen Objekttypen und Klassen. Dies führt zu unnötigen Einschränkungen, z. B. lassen sich Klassen in Programmiersprachen nicht mit Werten parametrisieren, um den Test *x is C* entscheidbar zu machen. Die Unterscheidung von Objekttypen und Klassen bringt eine zusätzliche Komplexität mit sich, scheint aber inhärent notwendig zu sein: Essentiell für Klassen ist, daß sie die privaten Attribute sichtbar machen. Nur dadurch ist es möglich, das Verhalten zu definieren. Essentiell für Objekttypen ist, daß sie die privaten Attribute verstecken (um so eine flexible Untertypisierung zu ermöglichen). In Sprachen wie Oberon definiert ein Objekttyp (bzw. Verbundtyp) auch alle Attribute. Ein Untertyp muß dann alle Attribute des Obertyps haben. Eine Verfeinerung, die die Repräsentation ändert wie in Kapitel 5 und 6 diskutiert ist dann nicht möglich.

9.2 Praktikabilität der Implementierung und Benutzung

Die Ausdrucksmächtigkeit der Entwurfsnotation geht bezüglich Spezifikation und Typisierung in ihrer Kombination über übliche objektorientierte Spezifikations- und Programmiersprachen hinaus. Für die Implementierung läßt sich daraus eine effiziente, übersetzbare Teilsprache selektieren. Diese kann ggf. weitere Restriktionen haben, z. B. Namensgleichheit von Typen. Mit einer prototypischen Implementierung von (Erweiterung von) $F^{\omega}_{\leq}$ und $F_{\leq}$ in (Cardelli, 1993; Matthes, 1993; Pierce & Turner, 1994) wurde eine prinzipielle Implementierbarkeit von mächtigen Typsystemen gezeigt.

Die Objekte können andersartig implementiert sein als sie hier auf eine abstrakte Art definiert werden. Beispielsweise können alle Objekte einer Klasse wie

üblich auf dieselben Methodenrümpfe zugreifen, und sich lediglich einen Zeiger auf sie halten. Denkbar ist auch eine Programmierumgebung, in der Spezifikationen auf korrekte Typisierung überprüft werden, denn bereits die Typisierungsregeln sind so komplex, daß eine mechanische Überprüfung der Typisierung hilfreich ist.

Ein Teil der Komplexität kommt durch die Kodierung von Objekten in $F^{\omega}_{\leq}$. Eine Vereinfachung für den Anwender wäre, auf eine Typtheorie mit „eingebauten" Objekten aufzusetzen. Damit ließen sich Konzepte, wie existentielle Quantoren, Packen und Öffnen, die nur zur Definition von Objekten benötigt werden, vermeiden. Ein neuerer Ansatz hierzu ist in (Cardelli, 1994) vorgestellt. Die anderen Konzepte von $F^{\omega}_{\leq}$ wie Untertypisierung und Parametrisierung mit Typen sind der Objektorientierung inhärent und nicht zu vermeiden. Inkonsistenzen bei der Untertypisierung und Parametrisierung können aber immer mechanisch aufgedeckt werden.

Eine weitere Komplexität kommt von den vielfältigen Konzepten des Verfeinerungskalküls. Viele dieser Konzepte, wie Anweisungen mit unterschiedlichen initialen und finalen Zustandsräumen, konjunktive und disjunktive Anweisungen, dienen nur der Herleitung von Rechenregeln wie den Datenverfeinerungsregeln. Sie sind für praktische Spezifikation und Verifikation irrelevant. Durch die Angabe von genügend vielen prozeduralen und Datenerfeinerungsregeln kann von vielen Konzepten abstrahiert werden. Prädikatentransformer und Zustandsprädikate wären dann nur noch ein Modell für Anweisungen und boolesche Ausdrücke. Interessant ist die Angabe eines *vollständigen* Satzes von Gleichheits- und Verfeinerungsregeln etwa wie in (Hoare, et al., 1987) für sequentielle Programme.

9.3 Ausblick

In dieser Arbeit wurde zur Kapselung existentielle Quantifizierung statt Rekursion wie in den meisten Ansätzen genommen. Dies vereinfacht das Modell von Objekten. Trotzdem gibt es Fälle, in denen Rekursion unvermeidbar ist. Rekursion kann in Form rekursiver Typen auftreten (z. B. ein Typ *Point* mit Methode *distance* (*other*: *Point*) oder in Form rekursiver Werte (z. B. eine Klasse *TreeNode* mit den Attributen *left*, *right* : *TreeNode*). Vererbung wird dann als *inkrementelle*

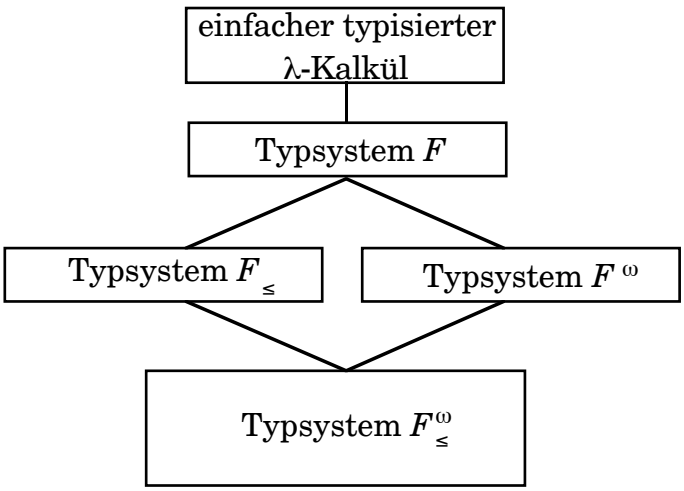
Modifikation selbstreferenzierender Strukturen verstanden (Wegner & Zdonik, 1988). Rekursiven Typen und Werten können in $F^{\omega}_{\leq}$ mit Rekursion (Bruce & Mitchell, 1992) definiert werden. Eine sinnvolle Definition der Verfeinerung rekursiver Klassen bleibt aber ein offenes Problem.

Einige objektorientierte Sprachen bieten *Module* zur Gruppierung von Klassen an (Nelson, 1991; Reiser & Wirth, 1992; Wills, 1991). Sie dienen u. a. der syntaktischen Einschränkung der Sichtbarkeit von Klassen. Interessant ist aber die rekursive Abhängigkeit von Klassen innerhalb eines Moduls. Dies tritt typischerweise bei *Frameworks* auf. Ein Framework ist eine Sammlung von abstrakten und konkreten Klassen zu einem Anwendungsgebiet. Frameworks legen den Zustandsraum und Methoden der Klassen teilweise fest; für eine vollständige Anwendung müssen die Lücken gefüllt werden. Frameworks dienen der *Wiederverwendung von Entwürfen* (Deutsch, 1989). Ein typisches Beispiel ist das Smalltalk Model / View / Controller Framework (Goldberg & Robson, 1983). Eine typtheoretische Behandlung von gegenseitig rekursiven Typen und Werten ist ein offenes und interessantes Problem.

Objektidentitäten können zum Aufbau graphenartige Objektgeflechte benutzt werden. Beispielsweise kann eine Menge implementiert werden durch ein Objekt, das die Wurzel eines Baumes ist, der die Menge repräsentiert. Dies wird in (Jones, 1992) zusammen mit der *Nebenläufigkeit* von Objekten betrachtet (die für jedes Objekt eine Identität voraussetzt, unter der Nachrichten geschickt werden können). Die Spezifikation ist ein (sequentielles) Objekt, daß sukzessive zerlegt wird in Objekte, die parallel ausgeführt werden können. Verifikationsregeln für Nebenläufigkeit im Hoare-Stil sind in (America & Boer, 1994) gegeben. In beiden Ansätzen dient die Strukturierung mit Objekten zur Einschränkung der Parallelität: es kann immer nur eine Methode in einem Objekt aktiv sein.

Anhang: Das Typsystem $F^{\omega}_{\leq}$

Verschiedene Varianten des λ–Kalküls sind entstanden, um in einem einfachen funktionalen Rahmen die Typisierung und Semantik von objektorientierten Programmen zu studieren. Die Grundlage dafür liefert der Ansatz, Objekte als Verbunde aufzufassen (Cardelli, 1984). Dieser wurde in (Cardelli & Wegner, 1985) zu der Sprache *Bounded Fun* erweitert, die dem Typsystem $F_{\leq}$ entspricht. Das Typsystem $F_{\leq}$ entsteht aus dem einfachen typisierten λ-Kalkül von A. Church durch zwei Erweiterungen. Die eine Erweiterung ist die Parametrisierung von Funktionen mit Typen. Dieser polymorphe λ-Kalkül wurde von Y. Girard System F genannt. Die andere Erweiterung ist eine Untertyprelation ≤ zwischen Typen. Das Typsystem F^{ω} ist eine Erweiterung von F um Typoperatoren, d. h. Funktionen von Typen nach Typen. Das Typsystem $F^{\omega}_{\leq}$ entsteht durch die Kombination von $F_{\leq}$ und F^{ω}.



In diesem Anhang werden die Syntax und die Regeln von $F^{\omega}_{\leq}$ (Pierce & Turner, 1994) folgend vorgestellt. Die Semantik von $F^{\omega}_{\leq}$ ist in (Cardelli & Longo, 1991) gegeben, die Semantik einer erweiterten Version mit rekursiven Werten und

Typen in (Bruce & Mitchell, 1992). Die Äquivalenzregeln sind aus (Cardelli, et al., 1991; Hofmann & Pierce, 1994) entnommen.

Das Prinzip der Semantik von $F^{\omega}_{\leq}$ ist, daß Typen Mengen bestimmter Bauart sind, daß die Zugehörigkeit zu einem Typ, $x : T$, bedeutet $x \in T$ und daß die Untertypisierung $S \leq T$ bedeutet $S \subseteq T$. Hierzu ist es notwendig, Typen als partielle Äquivalenzrelationen (partial equivalence relation, PER) zu definieren (Bruce & Mitchell, 1992). Das sind Relationen, die symmetrisch und transitiv, aber nicht notwendigerweise reflexiv sind. Das Typsystem $F^{\omega}_{\leq}$ ist das flexibelste bekannte Typsystem, das eine statische Überprüfung der Typisierung zuläßt. Weitere Ausdrucksmächtigkeit könnte dadurch entstehen, daß Typen selbst als Werte betrachtet werden. Auch wenn es möglich ist, für solche Kalküle eine Semantik zu geben (u.a. in (Cardelli, 1986)), muß die intuitive Definition von Typen durch Mengen aufgegeben werden. Zudem ist die korrekte Typisierung nicht mehr entscheidbar, was entgegen der Motivation für Typisierung ist. Aus diesem Grund ist es sinnvoll, sich auf $F^{\omega}_{\leq}$ zu beschränken.

Abschnitt A.1 stellt den einfachen typisierten λ-Kalkül vor, A.2 die Erweiterung zu $F_{\leq}$ und A.3 die Erweiterung zu $F^{\omega}_{\leq}$.

A.1 Der einfache typisierte λ-Kalkül

Das einfache typisierte λ-Kalkül ist von „höherer Ordnung“ in dem Sinne, daß Argumente von Funktionen wieder Funktionen sein können. Die Konsistenz des Kalküls ist dadurch gewährleistet, daß jedem Ausdruck genau ein *Typ* zugeordnet wird. Typen entweder Basistypen wie *Int*, *Bool* oder Funktionstypen wie z. B. $Int \rightarrow Int$, der Typ der Funktionen von *Int* nach *Int*. Ausdrücke bestehen aus elementaren Operationen auf den Basistypen, Variablen, Funktionsabstraktionen und Funktionsapplikationen. Die Syntax hat folgende Form. Es gibt zwei syntaktische Kategorien, eine für Typen (T) und eine für Werte (e):

T	::=	$Bool \mid Int \mid \ldots$	*Basistypen*
	$\mid$	$T \rightarrow T$	*Funktionstyp*
e	::=	x	*Variable*
	$\mid$	$false \mid true \mid 0 \mid 1 \mid \ldots$	*Konstante*
	$\mid$	$e + e \mid e \wedge e \mid \ldots$	*Basisoperationen*

|	$(\lambda\, x : T \bullet e)$	*Abstraktion*
|	$e\ e$	*Applikation*

Typ- und Wertausdrücke können geklammert werden, um Mehrdeutigkeiten zu vermeiden.

Als Basistypen werden einfache Typen wie *Int* und *Bool* (mit den Werten *true* und *false*) verwendet, sowie zusammengesetzte Typen wie Mengen *set of T*, Sequenzen *seq of T*, Multimengen *bag of T* und Kreuzprodukte $T_1 \times \ldots \times T_n$, jeweils mit den üblichen Operationen. Als abkürzende Schreibweise kann die Funktion

$$(\lambda\, x_1, \ldots, x_n : T_1, \ldots, T_n \bullet e)$$

auf ein Tupel $(e_1, \ldots, e_n)$ vom Typ $T_1 \times \ldots \times T_n$ angewendet werden.

Ein Ausdruck e hat genau einen Typ T, geschrieben $e : T$. Eine *Umgebung* E ist eine Liste von Variablentypisierungen $x_1 : T_1, \ldots, x_n : T_n$. „Aus der Umgebung E folgt $e : T$“ wird geschrieben als $E \vdash e : T$. Mit folgenden *Inferenzregeln* läßt sich der Typ eines (Wert-) Ausdrucks bestimmen.

$$E, x : T \vdash x : T \qquad \text{:–Variable}$$

$$\frac{E, x : T \vdash e : U}{E \vdash (\lambda\, x : T \bullet e) : T \to U} \qquad \text{:–Abstraktion}$$

$$\frac{E \vdash e : T \to U \quad E \vdash f : T}{E \vdash e\ f : U} \qquad \text{:–Applikation}$$

Neben diesen Typisierungsregeln gibt es *Äquivalenzregeln* $E \vdash e = f$, die das „Rechnen“ ermöglichen:

$$E \vdash e = e \qquad \text{=–Reflexivität}$$

$$\frac{E \vdash e = f}{E \vdash f = e} \qquad \text{=–Symmetrie}$$

$$\frac{E \vdash e = f \quad E \vdash f = g}{E \vdash e = g} \qquad \text{=–Transitivität}$$

$$\frac{E, x : T \vdash e = f}{E \vdash (\lambda\, x : T \bullet e) = (\lambda\, x : T \bullet f)} \qquad =\text{–}\textit{Abstraktion}\ (\xi)$$

$$\frac{E \vdash e = e' \quad E \vdash f = f'}{E \vdash e\, f = e'\, f'} \qquad =\text{–}\textit{Applikation}\ (\mu)$$

$$\frac{y \text{ nicht frei in } e}{E \vdash (\lambda\, x : T \bullet e) = (\lambda\, y : T \bullet e[y/x\,])} \qquad \textit{Umbenennung}\ (\alpha)$$

$$E \vdash (\lambda\, x : T \bullet e)\, f = e[f/x\,] \qquad \textit{Reduktion}\ (\beta)$$

$$E \vdash (\lambda\, x : T \bullet e\, x) = e \qquad \textit{Konversion}\ (\eta)$$

A.2 Parametrische Polymorphie und Untertyp-Polymorphie

Sei $l_1, \ldots, l_n$ eine Liste von *Feldnamen*. Der Typ $(l_1 : T_1, \ldots, l_n : T_n)$ steht für Verbunde mit durch $l_1, \ldots, l_n$ benannten *Feldern* der Typen $T_1, \ldots, T_n$. Die Reihenfolge der Felder ist irrelevant. Die Syntax wird wie folgt erweitert:

$$\begin{array}{lll} T ::= & \ldots & \\ \quad | & (l_1 : T_1, \ldots, l_n : T_n) & \textit{Verbundtyp} \end{array}$$

$$\begin{array}{lll} e ::= & \ldots & \\ \quad | & (l_1 \mapsto e_1, \ldots, l_n \mapsto e_n) & \textit{Verbund} \\ \quad | & e.l & \textit{Feldselektion} \end{array}$$

Die Typisierungsregeln für Verbunde sind:

$$\frac{E \vdash e_1 : T_1 \quad \ldots \quad E \vdash e_n : T_n}{E \vdash (l_1 \mapsto e_1, \ldots, l_n \mapsto e_n) : (l_1 : T_1, \ldots, l_n : T_n)} \qquad :\text{–}\textit{Verbund}$$

$$\frac{E \vdash (l_1 \mapsto e_1, \ldots, l_n \mapsto e_n) : (l_1 : T_1, \ldots, l_n : T_n)}{E \vdash (l_1 \mapsto e_1, \ldots, l_n \mapsto e_n).l_i : T_i} \qquad :\text{–}\textit{Feldselektion}$$

Die Äquivalenzregeln für Verbunde sind:

$$\frac{E \vdash e_1 = e_1' \quad \ldots \quad E \vdash e_n = e_n'}{E \vdash (l_1 \mapsto e_1, \ldots, l_n \mapsto e_n) = (l_1 \mapsto e_1', \ldots, l_n \mapsto e_n')} \qquad =\text{–}\textit{Verbund}$$

$$E \vdash (l_1 \mapsto e_1, \ldots, l_n \mapsto e_n).l_i = e_i \qquad =\text{–}\textit{Feldselektion}$$

„Typ T ist Untertyp von Typ U“ wird geschrieben als $T \leq U$. Für Verbundtypen bedeutet dies, daß der Untertyp mehr Felder haben kann. Ist $f : U \to S$ eine Funktion und e von einem Untertyp von U, dann ist die Applikation von $f\,e$ zulässig: der aktuelle Parameter hat alle vom formalen Parametertyp geforderten „Eigenschaften“. Dieses Prinzip heißt *Untertyp-Polymorphie* (*„subtype polymorphism“*).

Die folgenden *Untertypisierungsregeln* sind von der Form $E \vdash T \leq U$:

$$E \vdash T \leq T \qquad \leq\text{–}\textit{Reflexivität}$$

$$\frac{E \vdash S \leq T \quad E \vdash T \leq U}{E \vdash S \leq U} \qquad \leq\text{–}\textit{Transitivität}$$

$$\frac{E \vdash T \leq T' \quad E \vdash U' \leq U}{E \vdash T' \to U' \leq T \to U} \qquad \leq\text{–}\textit{Funktionstyp}$$

$$\frac{E \vdash (l_1 : S_1, \ldots, l_m : S_m) \leq (l_1 : T_1, \ldots, l_m : T_m)}{E \vdash (l_1 : S_1, \ldots, l_m : S_m, \ldots, l_n : S_m) \leq (l_1 : T_1, \ldots, l_n : T_n)} \qquad \leq\text{–}\textit{Verbundtyp}$$

Es kommt folgende *Subsumptionsregel* hinzu, die den Zusammenhang zwischen Typisierung und Untertypisierung beschreibt: Ist ein Wert e von einem Typ T und ist T ein Untertyp von U, dann ist auch e vom Typ U:

$$\frac{E \vdash e : T \quad E \vdash T \leq U}{E \vdash e : U} \qquad \textit{Subsumption}$$

Ein Wert kann also mehrere Typen haben, im Gegensatz zum einfachen typisierten λ-Kalkül. Für jeden Wert gibt es aber immer einen wohldefinierten kleinsten Typ. In der Untertyprelation gibt es einen größten Typ, mit *TYPE* bezeichnet. Jeder Wert ist vom Typ *TYPE*.

Falls $l = l_1, \ldots, l_n$ eine Liste von Namen $T = T_1, \ldots, T_n$ eine Liste von Typen und $e = e_1, \ldots e_n$ eine Liste von Werten ist, steht

$$(l : T) \quad \text{für} \quad (l_1 : T_1, \ldots, l_n : T_n) \ ,$$
$$(l \mapsto e) \quad \text{für} \quad (l_1 \mapsto e_1, \ldots, l_n \mapsto e_n) \ .$$

Das *Überschreiben* $R \oplus S$ von zwei Verbundtypen R und S steht für den Verbundtyp mit den Feldern von sowohl R als auch S, wobei die letzteren die ersteren mit dem gleichen Namen überschreiben. Das *Überschreiben* $r \oplus s$ von zwei Verbunden r und s steht für den Verbund mit den Feldern von sowohl r als auch s. Die Menge *dom* R von Feldnamen eines Verbundes R steht für

$$dom\ (l_1 : T_1, \ldots, l_n : T_n) \ = \ \{l_1, \ldots, l_n\} \ .$$

Die Einschränkung $r \mid \{l_1, \ldots, l_m\}$ eines Verbundes r auf Felder $l_1, \ldots l_m$ steht für

$$(l_1 \mapsto e_1, \ldots, l_m \mapsto e_m, \ldots, l_n \mapsto e_n) \mid \{l_1, \ldots, l_m\} \ = \ (l_1 \mapsto e_1, \ldots, l_m \mapsto e_m) \ .$$

Universell quantifizierte Typen erlauben *parametrische Polymorphie* zu beschreiben. Beispielsweise ist

$$Id \ \widehat{=} \ (\lambda A \bullet (\lambda x : A \bullet x))$$

die polymorphe Identitätsfunktion und

$$Id\ Int \ = \ (\lambda x : Int \bullet x)$$

die Identitätsfunktion auf *Int*. Der Typ von *Id* ist

$$(\prod A \bullet A \to A) \ ,$$

der Typ der Funktionen, die zuerst eine Typparameter nehmen und dann einen Wert dieses Typs. Typparameter können auch beschränkt sein. Die Funktion $(\lambda A <: T \bullet e)$ nimmt als Parameter einen Untertyp von T. Der Typ dieser Funktion ist $(\prod A <: T \bullet S)$, falls e vom Typ S unter der Annahme daß $A \leq T$.

Mit *existentiell quantifizierten Typen* wird *Kapselung* ausgedrückt (Mitchell & Plotkin, 1985). Beispielsweise wird der Typ

$$(\sum S \bullet S \times (S \to Bool))$$

als ein Objekttyp verstanden, dessen Elemente aus einem Wert vom Typ S, dem „Zustand“, und einer Funktion von S nach *Bool* bestehen, einer „Methode“. Der Typ S des Zustandes ist unbekannt, man weiß lediglich, daß es einen gibt: Den durch $\sum$ geschriebenen existentiellen Quantor kann man sich als eine Vereinigung vorstellen. Mit

$$p \mathrel{\widehat{=}} pack\ (5, (\lambda\, x : Int \bullet odd\ x))\ by\ Int\ as\ (\sum S \bullet S \times (S \to Bool))$$

wird ein Wert dieses Typs erzeugt bei dem S zu *Int* gesetzt wird. Es ist möglich, Werte (Objekte) des gleichen Typs (Signatur) mit unterschiedlicher Repräsentation zu erzeugen. Mit

$$open\ p\ as\ S\ by\ state, oper : S, S \to Bool\ in\ oper\ state\ end$$

wird ein gepacktes Element dieses Typs „geöffnet", indem der Zustand durch *state* und die Operation durch *oper* benannt werden. Sodann wird die Operation, eine Funktion von S nach *Bool*, auf die erste Komponente vom Typ S angewandt. Das Resultat ist vom Typ *Bool*. Mit p wie oben ergibt dies *true*.

In (Cardelli & Wegner, 1985) wurden *beschränkte* existentielle Typen vorgeschlagen. Dies sind Typen, die ihre Repräsentation teilweise nach außen bekannt geben. Die Syntax wird folgendermaßen erweitert:

T ::=	…	
	$\mid A$	*Typvariable*
	$\mid (\prod A <: T \bullet T)$	*universeller Typ*
	$\mid (\sum A <: T \bullet T)$	*existentieller Typ*
e ::=	…	
	$\mid (\lambda\, A <: T \bullet e)$	*Typabstraktion*
	$\mid e\ T$	*Typapplikation*
	$\mid pack\ e\ by\ T\ as\ T$	*Packen*
	$\mid open\ e\ as\ A <: T\ by\ x : T\ in\ e\ end$	*Öffnen*

Sei *TYPE* der größte Typ. Es gelten folgende Abkürzungen:

$$(\prod A \bullet T) = (\prod A <: TYPE \bullet T)$$
$$(\sum A \bullet T) = (\sum A <: TYPE \bullet T)$$
$$(\lambda\, A \bullet e) = (\lambda\, A <: TYPE \bullet e)$$
$$open\ e\ as\ A\ by\ x : T\ in\ e\ end = open\ e\ as\ A <: TYPE\ by\ x : T\ in\ e\ end$$

Eine Umgebung besteht zusätzlich zu Typisierungen $x : T$ aus Untertypisierungen $A <: T$. Die Typisierungsregeln für quantifizierte Typen sind:

$$\frac{E, A <: T \vdash e : U}{E \vdash (\lambda\, A <: T \bullet e) : (\prod A <: T \bullet U)} \quad :\text{–}Typabstraktion$$

$$\frac{E \vdash e : (\prod A <: T \bullet U) \quad E \vdash S \leq T}{E \vdash e\ S : U[S/A]} \qquad :\text{–Typapplikation}$$

$$\frac{E \vdash S \leq T \quad E \vdash e : U[S/A]}{E \vdash pack\ e\ by\ S\ as\ (\sum A <: T \bullet U) : (\sum A <: T \bullet U)} \qquad :\text{–Packen}$$

$$\frac{E \vdash e : (\sum A <: T \bullet U) \quad E, A <: T, x : U \vdash f : S}{E \vdash open\ e\ as\ A <: T\ by\ x : U\ in\ f\ end : S} \qquad :\text{–Öffnen}$$

Die Äquivalenzregeln für quantifizierte Typen sind:

$$\frac{E, A <: T \vdash e = f}{E \vdash (\lambda A <: T \bullet e) = (\lambda A <: T \bullet f)} \qquad =\text{–Typabstraktion}$$

$$\frac{E \vdash e = e' \quad E \vdash T = T'}{E \vdash e\ T = e'\ T'} \qquad =\text{–Typapplikation}$$

$$\frac{E \vdash e = e' \quad E \vdash T = T' \quad E \vdash S = S'}{E \vdash pack\ e\ by\ S\ as\ T = pack\ e'\ by\ S'\ as\ T'} \qquad =\text{–Packen}$$

$$\frac{\begin{array}{l}E \vdash e = e' \quad E \vdash T = T' \\ E, A <: T \vdash U = U' \\ E, A <: T, x : U \vdash f = f'\end{array}}{\begin{array}{l}E \vdash open\ e\ as\ A <: T\ by\ x : U\ in\ f\ end = \\ \quad open\ e'\ as\ A <: T'\ by\ x : U'\ in\ f'\ end\end{array}} \qquad =\text{–Öffnen}$$

$$\frac{B \text{ nicht frei in } e}{E \vdash (\lambda A <: T \bullet e) = (\lambda B <: T \bullet e[B/A])} \qquad Typ\text{–Umbenennung}$$

$$\frac{E \vdash U \leq T}{E \vdash (\lambda A <: T \bullet e)\ U = e[U/A]} \qquad Typ\text{–Reduktion}$$

$$E \vdash (\lambda A <: T \bullet e\ A) = e \qquad Typ\text{–Konversion}$$

$$\frac{B \text{ nicht frei in } U, f \quad y \text{ nicht frei in } f}{\begin{array}{l}E \vdash open\ e\ as\ A <: T\ by\ x : U\ in\ f\ end = \\ \quad open\ e\ as\ B <: T\ by\ y : U[B/A]\ in\ f[y/x][B/A]\ end\end{array}} \qquad open\text{–Umbenennung}$$

$$\frac{E \vdash S \leq T \quad E \vdash e : U[S/A]}{E \vdash \mathit{open}\ (\mathit{pack}\ e\ \mathit{by}\ S\ \mathit{as}\ (\sum A <: T \bullet U))\ \mathit{as}\ B <: T\ \mathit{by}\ x : U\ \mathit{in}\ f\ \mathit{end} = f[e/x][S/B]} \quad \textit{pack–open–Reduktion}$$

$$E \vdash \mathit{open}\ e\ \mathit{as}\ A <: T\ \mathit{by}\ x : U\ \mathit{in}\ \mathit{pack}\ e\ \mathit{by}\ A\ \mathit{as}\ (\sum A <: T \bullet U)\ \mathit{end} = e \quad \textit{pack–open–Konversion}$$

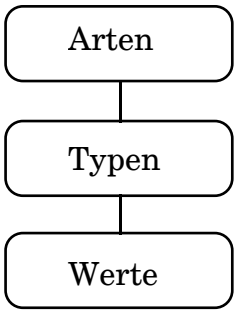
Die Untertypisierungsregeln für quantifizierte Typen sind:

$$E, A <: T \vdash A \leq T \quad \textit{≤–Typvariable}$$

$$\frac{E \vdash T \leq T' \quad E, A <: T \vdash U' \leq U}{E \vdash (\prod A <: T' \bullet U') \leq (\prod A <: T \bullet U)} \quad \textit{≤–Universeller Typ}$$

$$\frac{E \vdash T \leq T' \quad E, A <: T \vdash U' \leq U}{E \vdash (\sum A <: T' \bullet U') \leq (\sum A <: T \bullet U)} \quad \textit{≤–Existentieller Typ}$$

A.3 Typoperatoren



Die letzte Erweiterung entsteht, indem man für Typen Ausdrücke zuläßt ähnlich denen für Werte, nur eine Stufe höher. Weil die Funktionen von Werten nach Werten selbst keine Werte, sondern von einem Typ sind, sind auch die Funktionen von Typen nach Typen selbst keine Typen sondern von einer *Art* (*kind*). Die Syntax ergibt sich mit folgenden Erweiterungen:

$K ::=$	$*$	*Art der Typen*
	$\mid K \to K$	*Art der Typoperatoren*
$T ::=$	$\ldots$	
	$\mid \mathit{Top}\ K$	*Top*
	$\mid (\Lambda A : K \bullet T)$	*Typoperatorabstraktion*
	$\mid T\ T$	*Typoperatorapplikation*

Dabei ist *Top K* der größte Typ der Art *K*. Es gilt:

$$\mathit{TYPE} \mathrel{\widehat{=}} \mathit{Top}\ *$$

Beispielsweise läßt sich der generische Typ $Stack[X]$ auffassen als eine Funktion, die einen (beliebigen) Typ als Argument nimmt und eine Typ liefert. Sie ist also von der Art $* \to *$, geschrieben $Stack : * \to *$.

Es seien im folgenden A, B, ... Typvariablen, S, T, U, ... Typausdrücke und K, L, ... Artausdrücke. Umgebungen enthalten zusätzlich Deklarationen der Form $A : K$. Die Art der Typausdrücke wird durch folgende Regeln bestimmt:

$$E \vdash Top\ K : K \qquad :\!-Top$$

$$E, A : K \vdash A : K \qquad :\!-Typvariablen$$

$$\frac{E, A : Top\ K \vdash U : L}{E \vdash (\Lambda\, A : K \bullet U) : K \to L} \qquad :\!-Typoperatorabstraktion$$

$$\frac{E \vdash S : K \to L \quad E \vdash T : K}{E \vdash S\ T : L} \qquad :\!-Typoperatorapplikation$$

Folgende Untertypisierungsregeln kommen hinzu:

$$\frac{E \vdash T : K}{E \vdash T \leq Top\ K} \qquad \leq\!-Top$$

$$\frac{E, \mathrm{A} \leq Top\ K \vdash T' \leq T}{E \vdash (\Lambda\, A : K \bullet T') \leq (\Lambda\, A : K \bullet T)} \qquad \leq\!-Typoperatorabstraktion$$

Verzeichnis der Definitionen und Sätze

Literaturverzeichnis

Amadio, R. M., Cardelli, L. (1993). Subtyping Recursive Types. *ACM Transactions on Programming Languages and Systems*, 15(4), S. 575-631.

America, P. (1987). Inheritance and subtyping in a parallel object-oriented language. In J. Bézivin, J.-M. Hullot, P. Cointe, H. Lieberman (Hrsg.), *ECOOP'87: European Conference on Object-Oriented Programming* (S. 234-242). Paris, France, Springer-Verlag.

America, P. (1990). A Behavioral Approach to Subtyping in Object-Oriented Programming Languages. In J. W. deBakker, W. P. deRoever, G. Rozenberg (Hrsg.), *REX School / Workshop on the Foundations of Object-Oriented Languages* (S. 60-90). Noordwijkerhout, The Netherlands, Springer-Verlag.

America, P., Boer, F. d. (1994). Reasoning about Dynamically Evolving Process Structures. *Formal Aspect of Computing*, erscheint noch, S. 1-49.

Apt, K. R., Plotkin, G. D. (1986). Countable Nondeterminism and Random Assignment. *Journal of the ACM*, 33(4), S. 724-767.

Back, R. J. R. (1978). *On the Correctness of Refinement Steps in Program Development*. Ph. D. Thesis, Department of Computer Science, University of Helsinki.

Back, R. J. R. (1980). *Correctness Preserving Program Refinements: Proof Theory and Applications*. Mathematical Centre Tracts 131, Amsterdam, Mathematisch Centrum.

Back, R. J. R. (1987). *Procedural Abstraction in the Refinement Calculus* (Ser. A No. 55). Departments of Computer Science and Mathematics, Åbo Akademi.

Back, R. J. R. (1989). Data refinement in the refinement calculus. In *22nd Hawaii International Conference of System Sciences*. Kailua-Kona, January 1989.

Back, R. J. R. (1993). Refinement Calculus, Lattices and Higher Order Logic. In M. Broy (Hrsg.), *Program Design Calculi* (S. 53-72). Springer-Verlag.

Back, R. J. R., Wright, J. v. (1989). Refinement Calculus, Part I: Sequential Nondeterministic Programs. In J. W. deBakker, W.-P. deRoever, G. Rozenberg (Hrsg.), *REX Workshop on Stepwise Refinement of Distributed Systems - Models, Formalisms, Correctness* (S. 42-66). Mook, The Netherlands, Springer-Verlag.

Back, R. J. R., Wright, J. v. (1990). Dualitiy in Specification Languages: A Lattice-theoretical Approach. *Acta Informatica*, 27, S. 583-625.

Back, R. J. R., Wright, J. v. (1992). Combining Angels, Demons and Miracles in Program Specifications. *Theoretical Computer Science*, 100, S. 365-383.

Bauer, F. L., Berghammer, R., Broy, M., Dosch, W., Geiselbrechtinger, F., Gnatz, R., Hangel, E., Hesse, W., Krieg-Brückner, B., Laut, A., Matzner, T. A., Möller, B., Nickl, F., Partsch, H., Pepper, P., Samelson, K., Wirsing, M., Wössner, H. (1985). *The Munich Project CIP. Volume I: The wide spectrum language CIP-L*. Lecture Notes in Computer Science 183, Springer-Verlag.

Bruce, K. B. (1992). *A Paradigmatic Object-Oriented Language: Design, Static Typing and Semantics* (Techincal Report No. CS-92-01). Williams College.

Bruce, K. B., Gent, R. v. (1993). *TOIL: A new Type-safe Object-oriented Imperative Language*. Department of Computer Science, Williams College.

Bruce, K. B., Longo, G. (1990). A Modest Model of Records, Inheritance, and Bounded Quantification. *Information and Computation*, 87, S. 196-240.

Bruce, K. B., Mitchell, J. (1992). PER models of subtyping, recursive types and higher-order polymorphism. In *Nineteenth ACM Symposium on Principles of Programming Languages*. Albequerque, New Mexico.

Bruce, K. B., Schuett, A., Gent, R. v. (1994). *PolyTOIL: A type-safe polymorphic object-oriented language*. Department of Computer Science, Williams College.

Budd, T. (1991). *An Introduction to Object Oriented Programming*. Addision-Wesley.

Canning, P. S., Cook, W., Hill, W. L., Mitchell, J. C., Olthoff, W. G. (1989). F-Bounded Quantification for Object-Oriented Programming. In *Proceedings of the ACM Conference on Functional Programming and Computer Architecture* (S. 273-280).

Cardelli, L. (1984). A Semantics of Multiple Inheritance. In G. Kahn, D. MacQueen, G. Plotkin (Hrsg.), *International Symposium on the Semantics of Data Types* (S. 51-67). Springer-Verlag.

Cardelli, L. (1986). *A Polymorphic λ-calculus with Type:Type* (Research Report No. 10). DEC System Research Center.

Cardelli, L. (1993). *An implementation of* $F_{<:}$ (Research Report No. 97). DEC Systems Research Center.

Cardelli, L. (1994). A Semantics of Object Types. In *Logic in Computer Science*, erscheint demnächst.

Cardelli, L., Donahue, J., Glassman, L., Jordan, M., Kalsow, B., Nelson, G. (1989). *Modula-3 Report (revised)* (SRC Report No. 52). DEC Systems Research Center.

Cardelli, L., Longo, G. (1991). A semantic basis for Quest. *Journal of Functional Programming*, 1(4), S. 417-458.

Cardelli, L., Mitchell, J. C., Martini, S., Scedrov, A. (1991). An extension of system F with subtyping. In *Theoretical Aspects of Computer Software*. Sendai, Japan, Springer-Verlag.

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

Carrington, D. A., Duke, D., Duke, R., King, P., Rose, A., Smith, G. (1990). Object-Z: An object-oriented extension to Z. In S. Vuong (Hrsg.), *Formal Description Techniques FORTE'89*. North-Holland.

Cook, W. R. (1989). A Proposal for Making Eiffel Type-Safe. *The Computer Journal*, 32(4), S. 305-310.

Cook, W. R. (1990). Object-Oriented Programming Versus Abstract Data Types. In J. W. deBakker, W. P. deRoever, G. Rozenberg (Hrsg.), *REX School / Workshop on Foundations of Object-Oriented Languages* (S. 151-178). Noordwijkerhout, The Netherlands, Springer-Verlag.

Cook, W. R. (1992). Interfaces and Specifications for the Smalltalk-80 Collection Classes. *SIGPLAN Notices (Special Issue on the Conf. on Object-Oriented Programming Systems, Languages and Applications)*, 27(10), S. 1-15.

Cox, B. J. (1986). *Object-Oriented Programming, An Evolutionary Approach*. Reading (Mass.), Addison-Wesley.

Curien, P.-L., Ghelli, G. (1992). Coherence of Subsumption, Minimum Typing and Type-Checking in $F_{\leq}$. *Mathematical Structures in Computer Science*, 2, S. 55-91.

Dahl, O. J., Nygaard, K. (1966). SIMULA - An ALGOL-based simulation language. *Communications of the ACM*, 9(9), S. 671-678.

Deutsch, L. P. (1989). Design Reuse and Frameworks in the Smalltalk-80 System. In T. J. Biggerstaff, A. J. Perlis (Hrsg.), *Software Reusability*. (S. 57-72). ACM Press.

Dijkstra, E. W. (1976). *A Discipline of Programming*. Prentice-Hall.

Dijkstra, E. W. (1982). The Equivalence of Bounded Nondeterminacy and Continuity. In *Selected Writing in Computing*. Springer-Verlag.

Dijkstra, E. W., Gasteren, A. J. M. v. (1986). A Simple Fixpoint Argument Without the Restriction to Continuity. *Acta Informatica*, 23(1-7).

Dijkstra, E. W., Scholten, C. S. (1990). *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science, Springer-Verlag.

Eifrig, J., Smith, S., Trifonov, V., Zwarico, A. (1993). A Simple Interpretation of OOP with State. In *SIPL'93 – ACM SIGPLAN Workshop on State in Programming Languages,* YALEU/DCS/RR-968 (S. 1-16). Copenhagen, Denmark, Yale University, Department of Computer Science.

Gardiner, P. H. B., Morgan, C. (1993). A Single Complete Rule for Data Refinement. *Formal Aspects of Computing*, 5, S. 367-382.

Goldberg, A., Robson, D. (1983). *Smalltalk-80: The Language and its Implementation*. Addison-Wesley.

He, J., Hoare, C. A. R., Sanders, J. W. (1986). Data Refinement Refined. In B. Robinet, R. Wilhelm (Hrsg.), *European Symposium on Programming*. Springer-Verlag.

Hoare, C. A. R. (1969). An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12, S. 576-581.

Hoare, C. A. R. (1972). Proof of Correctness of Data Representation. *Acta Informatica*, 1(4).

Hoare, C. A. R., Hayes, I. J., Jifeng, H., Morgan, C. C., Roscoe, A. W., Sanders, J. W., Sørensen, I. H., Spivey, J. M., Sufrin, B. A. (1987). Laws of Programming. *Communications of the ACM*, 30(9).

Hoare, C. A. R., He, J. (1986). The weakest prespecification. *Fundamenta Informaticae*, IX, S. 81-84.

Hofmann, M., Pierce, B. (1994). *A Unifying Type-Theoretic Framework for Objects* (Research Report). University of Edinburgh.

Johnson, R. E., Foote, B. (1988). Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1(2), S. 22-35.

Jones, C. B. (1992). *An Object-Based Design Method for Concurrent Programs* (Report No. 1992-12-04). University of Manchester, Department of Computer Science.

LaLonde, W., Pugh, J. (1991). Subclassing ≠ subtyping ≠ Is-a. *Journal of Object Oriented Programming*, (January 1991).

Lano, K. (1991). Z++, An Object-Oriented Extension To Z. In J. E. Nichols (Hrsg.), *Z User Workshop: Proceedings of the Forth Annual Z User Meeting*. Oxford, Springer-Verlag.

Lano, K., Haughton, H. (1992). Reasoning and Refinement in Object-Oriented Specification Languages. In O. L. Madsen (Hrsg.), *European Conference on Object-Oriented Programming ´92*. , Springer-Verlag.

Leavens, G. T. (1991). Modular Specification and Verification of Object-Oriented Programs. *IEEE Software*, July, S. 72-80.

Luckham, D. C., Suzuki, N. (1979). Verification of array, record, and pointer operations in Pascal. *ACM Transactions on Programming Languages and Systems*, 1(2), S. 226-244.

Matthes, F. (1993). *Persistente Objektsysteme - Integrierte Datenbankentwicklung und Programmerstellung*. Springer-Verlag.

Meyer, B. (1988). *Object-Oriented Software Construction*. Prentice Hall.

Mitchell, J. C. (1990). Type Systems for Programming Languages. In J. v. Leeuwen (Hrsg.), *Handbook of Theoretical Computer Science*. Elsevier Science Publisher.

Mitchell, J. C., Plotkin, G. D. (1985). Abstract types have existential type. In *12th Annual Symposium on Principles of Programming Languages* (S. 37-51). New Orleans, La., ACM, New York.

Morgan, C., Gardiner, P. H. B. (1990). Data Refinement by Calculation. *Acta Informatica*, 27, S. 481-503.

Morgan, C. C. (1988a). Data Refinement by Miracles. *Information Processing Letters*, 26 (January 1988).

Morgan, C. C. (1988b). Procedures, parameters, and abstraction: Separate concerns. *Science of Computer Programming*, 11(1), S. 17-28.

Morgan, C. C. (1988c). The Specification Statement. *ACM Transactions on Programming Languages and Systems*, 10(3), S. 403-419.

Morgan, C. C. (1990). *Programming from Specifications*. Prentice Hall.

Morris, J. M. (1987). A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3).

Morris, J. M. (1989). Laws of data refinement. *Acta Informatica*, 9(3), S. 287-306.

Naumann, D. A. (1994). Predicate Transformer Semantics of an Oberon-Like Language. In E.-R. Olderog (Hrsg.), *Programming Concepts, Methods and Calculi*(S. 460-480). San Miniato, Italy, International Federation for Information Processing.

Nelson, G. (Hrsg.). (1991). *Systems Programming with Modula-3*. Prentice-Hall.

Nipkow, T. (1986). Nondeterministic Data Types: Models and Implementations. *Acta Informatica*, 22.

Pierce, B. C., Turner, D. N. (1994). Simple Type-Theoretic Foundations For Object-Oriented Programming. *Journal of Functional Programming*, erscheint noch.

Reiser, M., Wirth, N. (1992). *Programming In Oberon - Steps Beyond Pascal and Modula*. Addison-Wesley.

Reynolds, J. C. (1978). User-Defined Types and Procedural Data Structures as Complementary Approaches to Data Abstraction. In D. Gries (Hrsg.), *Programming Methodology, A Collection of Papers by Members of IFIP WG 2.3*. (S. 309-317). New York, Springer Verlag.

Rüping, A., Weber, F., Zimmer, W. (1993). Demonstrating Coherent Design: A Data Structure Catalogue. In R. Ege, M. Singh, B. Meyer (Hrsg.), *11th Conference on Technology of Object-Oriented Languages and Systems* (S. 363-377). Santa Barbara, Prentice-Hall.

Stroustrup, B. (1986). *The C++ Programming Language*. Addison-Wesley.

Tessler, L. (1985). Object Pascal Report. *Structured Language World*, 9(3).

Utting, M. (1992). *An Object-Oriented Refinement Calculus with Modular Reasoning*. Doctoral Thesis, University of New South Wales, Kensington.

Weber, F. (1992). Getting Class Correctness and System Correctness Equivalent - How to get the Covariance Right. In *Proceeding of the TOOLS ´92 Conference*. St. Barbara, USA, Prentice-Hall.

Wegner, P., Zdonik, S. B. (1988). Inheritance as an Incremental Modification Mechanism or What Like Is and Isn‘t Like. In S. Gjessing, K. Nygaard (Hrsg.), *European Conference on Object Oriented Programming* (S. 55-77). Springer-Verlag.

Wills, A. (1991). Capsules and Types in Fresco. In P. America (Hrsg.), *European Conference on Object-Oriented Programming*. Springer-Verlag.