# Algebraic Foundations of the Unifying Theories of Programming

Walter Guttmann

Institut für Programmiermethodik und Compilerbau, Universität Ulm
walter.guttmann@uni-ulm.de

**Abstract:** Bei der Entwicklung von Software ist es erforderlich sich mit der Bedeutung von Programmen und Spezifikationen auseinanderzusetzen, etwa um bestehende Programme zu verstehen, die Korrektheit sicherheitskritischer Systeme zu gewährleisten oder eine eindeutige Interpretation von Modellen zu erreichen. Eine formale Semantik hilft dabei Fehler zu vermeiden, eröffnet den Zugang zu maschineller Unterstützung und schafft Klarheit im Zweifelsfall.

Die *Unifying Theories of Programming* von Hoare und He beschreiben die Semantik von Spezifikationen und Programmen unterschiedlicher Sprachparadigmen in einem gemeinsamen Formalismus. Im Mittelpunkt steht die Modellierung nichtdeterministischer, imperativer Programme durch spezielle Relationen, sogenannte *Designs*. Sie bilden eine solide Grundlage für den Aufbau weiterer Theorien, schränken aber den Anwendungsbereich unnötig ein.

Diese Arbeit verbessert die Unifying Theories of Programming auf zwei Arten. Einerseits verallgemeinern wir die Designs und klären ihre algebraische Struktur. Dies macht bestehende Mathematik auf die Unifying Theories of Programming anwendbar, integriert weitere Ansätze zur Semantik und vereinfacht Rechnungen und Beweise. Andererseits erweitern wir die Theorie um undefinierte Werte und nichtstrikte Programme. Dies ermöglicht das kontrollierte Auffangen von Berechnungen mit undefiniertem Ergebnis und die Übertragung der aus funktionalen Programmiersprachen bekannten verzögerten Auswertung auf imperative Programme.

## 1 Einführung

Ein Programm ist korrekt, wenn es leistet was sein Benutzer erwartet. Um diese Erwartungen zu präzisieren, werden sie bei einer methodischen Softwareentwicklung als Anforderungen formuliert und in einer Spezifikation festgelegt. Sodann gilt ein Programm als korrekt, wenn es seine Anforderungen erfüllt. Für eine derartige Aussage ist es aber nötig die Bedeutung des Programms und der Anforderungen zu erfassen. Dazu muß die Semantik der verwendeten Programmier- und Spezifikationssprachen beschrieben sein. Sofern überhaupt vorhanden, sind solche Semantikbeschreibungen in Textform und mehrere hundert Seiten lang, siehe etwa [ISO02, GJSB05, Obj07]. Das führt häufig dazu, daß sie Fehler enthalten oder in Übersetzern und anderen Werkzeugen fehlerhaft implementiert werden. Eine *formale* Spezifikation ist zwar aufwendig, trägt aber bei diese Fehler aufzudecken, wie etwa Arbeiten zur Semantik von Java und UML-Aktivitätsdiagrammen mit Abstract State Machines zeigen [Sch00, Sar06, SG07]. Eine formale Semantik ermöglicht zudem

Rechnerunterstützung, präzise Aussagen und Beweise, und sie findet Anwendung bei der Entwicklung korrekter Software durch Programmtransformation oder -verifikation.

Die hier vorgestellte Arbeit [Gut07] verwendet zur Formalisierung der Semantik die von Hoare und He in [HH98] eingeführten *Unifying Theories of Programming* (UTP). Dieser Ansatz versucht verschiedene Programmiersprachparadigmen semantisch unter einen Hut zu bringen. Im Kern geht es um imperative Programme, also solche mit Zustand und Anweisungen. Der Zustand ist gegeben durch die Werte der vorkommenden Variablen und eine Anweisung transformiert einen Zustand in einen neuen Zustand. Demnach wird unterschieden zwischen dem Wert einer Variablen vor Ausführung der Anweisung und dem Wert danach. Eine Anweisung oder Berechnung setzt diese Werte miteinander in Beziehung, ist also mathematisch gesehen eine Relation. Programme sind ebenfalls Relationen und werden aus elementaren Anweisungen, zum Beispiel Zuweisungen, mit Hilfe von Konstrukten wie Fallunterscheidung und Wiederholung zusammengesetzt.

Dieses einfache Modell reicht nur für terminierende Programme. Um Nichtterminierung darzustellen führt die UTP die sogenannten *Designs* ein (die als Begriff der UTP nicht etwa mit „Entwurf“ in der Softwaretechnik verwechselt werden sollten). Jedes Design beschreibt ein Programm und besteht aus zwei Komponenten, die beide Relationen sind. Eine charakterisiert die Startzustände, von denen aus das Programm terminiert und die andere Relation beschreibt die dort möglichen Zustandsübergänge. In einem Design werden diese beiden Arten von Informationen in geschickter Weise kombiniert. Dadurch sind Designs selber Relationen, aber nicht mehr beliebige wie im einfachen Modell, sondern speziellen Bedingungen unterworfene. Sämtliche Programme müssen diese Einschränkungen, die sogenannten *Healthiness Conditions*, erfüllen. Sie sind der Preis, den die Theorie zahlen muß, um die Wirklichkeit genauer wiederzugeben.

Die vorliegende Arbeit beschäftigt sich mit der Frage was geschieht, wenn derartige Einschränkungen fallengelassen werden. Davon erhofft man sich eine weitergehende Anwendbarkeit der Theorie in vergleichbaren oder auf neuen Gebieten. Die gewählte Vorgehensweise ist algebraisch: Aus der Mathematik ist bekannt, wie Relationen durch Axiome beschrieben werden können, siehe etwa [Mad96, SHW97]. Die Healthiness Conditions charakterisieren Designs und können ebenfalls als Axiome aufgefaßt werden. Beides zusammengenommen ergibt eine axiomatische Beschreibung der durch die UTP modellierten Programme. Diese kann nun untersucht werden, etwa um Axiome, die unnötige Einschränkungen verursachen, aufzuspüren und wegzulassen. Als ein Ergebnis dieser Arbeit wird die so gewonnene, den Designs zugrundeliegende Struktur klar dargelegt. Die erzielte Verallgemeinerung der UTP offenbart Zusammenhänge zu bestehender Mathematik und weiteren Ansätzen zur Semantik, unter anderem zu [Nel89, Wri04, MS06]. Zudem vereinfacht die neue, abstrahierende Darstellung Beweise und Rechnungen mit Programmen.

Die algebraische Sichtweise hat weitere Vorteile: Bei Bedarf kann man zusätzliche Struktur auferlegen indem man neue Operationen mit entsprechenden Axiomen einführt. Die Arbeit macht ausgiebig Gebrauch davon und beschreibt auf diese algebraische Art unter anderem endliche und unendliche Iterationen, Definitionsbereich, Urbild, Determiniertheit und Invarianten. Außerdem kann man Erweiterungen der UTP untersuchen, die mit Designs nicht möglich sind. Ein Ergebnis dieser Arbeit ist eine derartige Erweiterung um nichtstrikte Programme. Sie beschreiben Berechnungen, die trotz einer undefinierten

Eingabe ein definiertes Ergebnis produzieren. Diese nichtstrikten Berechnungen ermöglichen eine verzögerte Auswertung, wie aus funktionalen Programmiersprachen bekannt, auch im Kontext einer imperativen Sprache. Es ist dann ausreichend, die zur Berechnung der Endergebnisse notwendigen Programmteile auszuführen, was zu einer Verbesserung der Laufzeit führen kann.

## 2 Relationen und Designs

Um zu verstehen, weshalb eine algebraische Sicht und eine Verallgemeinerung der Designs sinnvoll sind, beschreiben wir ihren Aufbau genauer. Ein Design besteht aus zwei Relationen, die Zustandsübergänge und Terminierungseigenschaften modellieren.

Betrachten wir zunächst die Zustandsübergänge. Jeder Zustand im Ablauf eines imperativen Programms ist durch die Werte der vorkommenden Variablen charakterisiert. Der einfacheren Erklärung halber nehmen wir im folgenden an, daß es nur eine Variable $x$ gibt. Ein Zustand ist dann etwa durch $x = 2$ gegeben. Durch die Ausführung einer Anweisung wie $x := 3x + 1$ erreichen wir einen neuen Zustand, im Beispiel $x = 7$. Zur Unterscheidung schreiben wir $x$ und $x'$ für den Wert der Variablen $x$ vorher und nachher. Diese Werte werden durch die Anweisung in Beziehung gesetzt: Im Beispiel gilt $x' = 3x+1$. Das kann mathematisch als Relation, also Teilmenge eines kartesischen Produkts, aufgefaßt werden. Die angegebene Zuweisung *ist* eine Relation, formal gilt

$$(x := 3x + 1) \;\equiv\; \{(x, x') \mid x' = 3x + 1\}\,.$$

Nehmen wir etwa die natürlichen Zahlen als Wertebereich von $x$, dann ist das die Menge der Paare $\{(0, 1), (1, 4), (2, 7), (3, 10), \ldots\}$. Um Programme zusammenzusetzen, sieht die UTP außer Zuweisungen unter anderem die folgenden Konstrukte vor. Die Hintereinanderausführung der Programme $Q$ und $R$ ist mathematisch die Komposition zweier Relationen, geschrieben $Q \;;\; R$. Ihr neutrales Element, die Identitätsrelation, modelliert die leere Anweisung. Im speziellen Fall, daß die Relationen sogar Funktionen sind, erhalten wir die Funktionskomposition mit der Identitätsfunktion als neutralem Element. Es gibt eine Fallunterscheidung, deren Bedingung als spezielle Relation gesehen wird. Die Semantik rekursiver Programme ist durch den größten Fixpunkt gegeben.

Damit die UTP auch als Spezifikationssprache eingesetzt werden kann, gibt es das folgende Konstrukt, das wir etwas näher beschreiben. Die nichtdeterministische Auswahl, mathematisch die Vereinigung zweier Relationen, modelliert Unterspezifikation. Intuitiv gesehen ist das Paar $(x, x')$ in der Relation $R$ falls die durch $R$ dargestellte Berechnung bei Eingabe $x$ zur Ausgabe $x'$ führen *kann*. Wir erhalten zum Beispiel

$$(x := 3x + 1) \cup (x := 4x + 1) \;\equiv\; \{(0, 1), (1, 4), (1, 5), (2, 7), (2, 9), \ldots\}\,.$$

Bei Eingabe $x = 1$ sind die Ergebnisse $x' = 4$ und $x' = 5$ möglich. Gilt $Q \subseteq R$ für zwei Relationen $Q$ und $R$, so ist jede Beobachtung $(x, x') \in Q$ eines Ablaufs von $Q$ auch eine zulässige Beobachtung von $R$. In diesem Fall sagt man: Das Programm $Q$ erfüllt

die Spezifikation $R$. Die Teilmengenbeziehung ermöglicht also eine Entwicklung durch schrittweise Verfeinerung.

Kommen wir nun zu den Terminierungseigenschaften. Sie werden durch spezielle Relationen, sogenannte *Vektoren* beschrieben, die Mengen von Zuständen darstellen. Dabei umfaßt eine solche Menge genau die Startzustände, aus denen die Ausführung des Programms garantiert terminiert. Um auszudrücken, daß ein Zustand $x$ in der Menge ist, enthält die Relation sämtliche Paare $(x, x')$ mit beliebigem $x'$. Vektoren können durch Prädikate, die nur von $x$ aber nicht von $x'$ abhängen, charakterisiert werden.

Die UTP faßt nun beide Relationen, also die Terminierungsinformation $P$ und die Zustandsübergänge $Q$, als Prädikate auf und kombiniert sie zum Design

$$(P \vdash Q) \;\equiv\; (ok \wedge P) \Rightarrow (ok' \wedge Q)\,.$$

Zwei künstliche Boolesche Variablen $ok$ und $ok'$ modellieren Start und Ende des Programms. Das Design $(P \vdash Q)$ ist folgendermaßen zu interpretieren: Wenn das Programm in einem Zustand startet der die Vorbedingung $P$ erfüllt, stoppt es und zwar in einem Zustand der $Q$ erfüllt. Das spezielle Design $(\mathit{false} \vdash \mathit{false})$ beschreibt demnach ein Programm, das nie terminiert, also eine Endlosschleife. Eine der Einschränkungen, die sämtliche Designs erfüllen, ist

$$(\mathit{false} \vdash \mathit{false})\;;\,(P \vdash Q) \;\equiv\; (\mathit{false} \vdash \mathit{false})\,. \tag{1}$$

In Worten heißt das, ein Programm *nach* einer Endlosschleife auszuführen hat keine Auswirkung. Diese Aussage steht in Einklang mit vielen bekannten Programmiersprachen. Wie wir in Abschnitt 4 erörtern, kann es trotzdem sinnvoll sein, auf diese Einschränkung zu verzichten. Zuvor beschäftigen uns aber die Einschränkungen, die implizit durch die Verwendung von Relationen gegeben sind.

## 3 Algebraische Struktur der Designs

Relationen erlauben einige Operationen, deren Einsatz in der Informatik fraglich ist. Ein Beispiel ist die Transposition von Relationen, also das Vertauschen der Elemente in jedem Paar. Das entspricht dem „Rückwärtslaufenlassen“ von Programmen, womit man zum Beispiel eine Zahl sehr einfach faktorisieren könnte. Ein anderes Beispiel ist das Komplement einer Relation, also genau die Paare, die nicht in der Relation sind. Damit könnte man etwa einfach formulieren, daß ein Programm *nicht* abstürzen soll. Wenn man sich also die Frage stellt, welche Operationen man tatsächlich braucht, erreicht man einerseits eine Annäherung an die Informatik. Andererseits hat man mathematisch gesehen weniger Axiome, also weniger Einschränkungen, und damit mehr Strukturen die darauf passen. Relevante Aussagen der UTP sollten wir aber weiterhin zeigen können.

Bei Anwendung der UTP stellt man außerdem fest, daß komplizierte Rechnungen mit Substitutionen entstehen. Der Grund dafür liegt in der Verwendung von Prädikaten, um Designs und ihre Operationen zu beschreiben. Dadurch entstehen auch solche Artefakte,

wie die bereits erwähnten künstlichen Variablen $ok$ und $ok'$, die wir gleichermaßen beseitigen sollten, vergleiche [Gut06]. Die Idee ist nun, die Prädikate durch Elemente von Halbringen [HW93] zu ersetzen. Dadurch können wir Designs algebraisch beschreiben.

Wir verwenden Halbringe, weil sie genau zu den imperativen, nichtdeterministischen Programmen passen. Ein Halbring ist nämlich eine algebraische Struktur mit zwei Operationen $+$ und $*$ und dazugehörigen neutralen Elementen. Im Vergleich zu Ringen braucht die Addition $+$ keine inversen Elemente zu besitzen. Die Operation $+$ modelliert die nichtdeterministische Auswahl. Die Operation $*$ modelliert die sequentielle Komposition mit der leeren Anweisung als neutralem Element. Halbringe besitzen also nur einen Teil der Struktur der Relationen, insbesondere fehlen die oben erwähnten Operationen, deren Anwendung auf Programme nicht sinnvoll ist. Mit Halbringelementen modellieren wir die Zustandsübergänge von Designs.

Für die Terminierungsinformation benötigen wir etwas mehr Struktur. Dazu bestimmen wir eine Teilmenge des Halbrings, aus der wir die Vorbedingungen nehmen. Diese Teilmenge muß eine Boolesche Algebra sein, damit wir mit Bedingungen rechnen können wie gewohnt. Außerdem muß sie ein Linksideal des Halbrings sein, damit wir Bedingungen darstellen können wie in der UTP.

Unsere abstrakten Designs sind $(2 \times 2)$-Matrizen mit Elementen aus dem Halbring und dem Halbringideal. Die Hilfsvariablen $ok$ und $ok'$ sind in den Zeilen und Spalten der Matrix kodiert. Sämtliche Operationen, die Programmkonstrukte beschreiben und auf den Designs definiert sind, verallgemeinern wir auf die neue Darstellung. Durch die Vermeidung von Prädikaten wird zum Beispiel die sequentielle Komposition von Designs zur bekannten Matrixmultiplikation vereinfacht.

Die Verallgemeinerung der Bestandteile der Designs überträgt sich auf die Designs selber. Sie sind nun nicht mehr Relationen, sondern Elemente allgemeinerer Halbringe. Von Nachteil ist, daß wir deshalb weniger spezifische Aussagen zeigen können. Dies fällt aber nicht stark ins Gewicht, da die verringerten Eigenschaften immer noch ausreichen, um die Theorie der Designs zu rekonstruieren, unter anderem die Healthiness Conditions und einen Satz über Fixpunkte [HH98, Theorem 3.1.6]. Ein Vorteil der Verallgemeinerung ist, daß man Verbindungen zu verwandten Ansätzen zur Semantikbeschreibung herstellen kann, etwa um diese miteinander zu vergleichen [GM06].

Durch die Offenlegung der tatsächlich benötigten Axiome wird die wesentliche Struktur der Designs klar dargestellt. Außerdem erkennt man algebraische Eigenschaften besser und kann sie einfacher verwerten. Ein konkretes Ergebnis ist, daß die Designs eine Kleene-Algebra [Koz94] und eine Omega-Algebra [Coh00] bilden, mit denen endliche und unendliche Iterationen modelliert werden. Bei dem entsprechenden Beweis können wir dank der Darstellung von Designs als Matrizen auf bekannte Aussagen aus der Mathematik zurückgreifen [Con71]. Um zusätzliche Aspekte von Programmen zu beschreiben, führen wir zahlreiche neue Operationen mit Axiomen ein. Zum Beispiel beschreibt eine der Modallogik entlehnte, abstrakte Formulierung von Determiniertheit jene Zustände aus denen höchstens ein Übergang existiert [DM01].

Als Anwendung erlauben Kleene- und Omega-Algebren zum Beispiel Aussagen über die Äquivalenz verschieden strukturierter Rekursionen. Unter anderem zeigen wir algebraisch

wie man eine symmetrische lineare Rekursion durch zwei aufeinanderfolgende Schleifen ersetzen kann. Dies geschieht durch getrennte Betrachtung der terminierenden und nichtterminierenden Zustände unter Verwendung der axiomatisch beschriebenen Operationen. Derartige Äquivalenzbeweise sind für effizienzsteigernde Programmtransformationen erforderlich und wurden in der UTP bislang nicht durchgeführt. Dank der algebraischen Herangehensweise sind unsere Rechnungen im Vergleich zur UTP kompakter darstellbar und weniger fehleranfällig.

## 4 Nichtstrikte Berechnungen

Unser nächstes Ziel ist die Modellierung von nichtstrikten Berechnungen in der UTP. Eine nichtstrikte Berechnung kann trotz einer undefinierten Eingabe $\bot$ ein definiertes Ergebnis erzeugen. Der spezielle Wert $\bot$ beschreibt dabei Ausdrücke, deren Auswertung zum Programmabbruch führt, etwa eine Division durch Null. Betrachten wir das folgende Beispiel einer nichtstrikten Berechnung. Der Wert, den die Variable $x$ unmittelbar vor der Zuweisung $x := 2$ hat, wird nicht mehr verwendet und spielt daher keine Rolle. Wir können also die Gleichung

$$(x := \bot)\ ;\ (x := 2)\ \equiv\ (x := 2) \tag{2}$$

fordern. Die Zuweisung $x := \bot$ steht dabei für eine Berechnung, die abbricht, falls sie tatsächlich durchgeführt wird. Wie eben dargelegt, ist das in diesem Fall aber nicht nötig. Nichtstrikte Berechnungen ermöglichen eine solche verzögerte Auswertung, mit den aus funktionalen Programmiersprachen bekannten Vor- und Nachteilen, siehe etwa [PJ87].

Eine Umsetzung für imperative Sprachen wäre interessant, geht aber mit Designs nicht, und zwar aus dem folgenden Grund. In der UTP ist Undefiniertheit semantisch gleichgesetzt mit Nichtterminierung, formal gilt

$$(x := \bot)\ \equiv\ (\mathit{false} \vdash \mathit{false})\,.$$

Zusammen mit der für Designs gültigen Einschränkung (1), wonach insbesondere die Zuweisung $x := 2$ nach einer Endlosschleife keinen Effekt hat, folgt

$$(x := \bot)\ ;\ (x := 2)\ \equiv\ (\mathit{false} \vdash \mathit{false})\ ;\ (x := 2)\ \equiv\ (\mathit{false} \vdash \mathit{false})\,. \tag{3}$$

Die Gleichungen (2) und (3) stehen in Widerspruch zueinander, denn $x := 2$ terminiert, aber $(\mathit{false} \vdash \mathit{false})$ nicht. Die Gleichung (3) spricht für eine Auswertung der sequentiellen Komposition von links nach rechts, um die Endlosschleife zu berücksichtigen. Um den Programmabbruch zu vermeiden, spricht (2) dagegen für eine Auswertung von rechts nach links. Das Problem ist, wir müßten feststellen ob der vordere Programmteil terminiert, was im allgemeinen nur durch Ausführung möglich ist. Um die nichtstrikte Berechnung zu erhalten, müssen wir auf die Gleichung (3), also auch auf (1) und damit auf Designs verzichten und einen neuen Ansatz entwickeln.

Zunächst erweitern wir die UTP um die Fähigkeit, Variablen und Ausdrücke mit undefiniertem Wert unabhängig von Nichtterminierung darzustellen. Eine derartige Unterscheidung ist zum Beispiel für das kontrollierte Auffangen von Berechnungen mit undefiniertem Ergebnis notwendig. Sie kann herbeigeführt werden, indem man die Wertebereiche der Variablen und die Übergangsrelationen anpaßt. Zwei spezielle Elemente $\perp$ und $\infty$ der Wertebereiche beschreiben nun Ausdrücke, deren Auswertung undefiniert abbricht beziehungsweise nicht terminiert. Diese Unterscheidung zwischen Undefiniertheit und Nichtterminierung ist auch beim Ablauf von Programmen beobachtbar.

Statt Designs definieren wir dann neue Relationen, die nichtstrikte Berechnungen modellieren. Demnach sind die speziellen Werte $\perp$ und $\infty$ intuitiv wie folgt zu interpretieren. Hat eine Variable $x$ den Wert $\perp$ bedeutet das: Wenn der Wert von $x$ benötigt wird, führt das zu einem Abbruch. Hat $x$ den Wert $\infty$ bedeutet das: Wenn der Wert von $x$ benötigt wird, führt das zu Nichtterminierung. Wird der Wert von $x$ nicht benötigt, hat das keine weiteren Auswirkungen.

Die Schwierigkeit besteht darin, die Modelle so zu wählen, daß einerseits diese speziellen Werte bei ihrer Verwendung propagiert werden und andererseits die Programmkonstrukte *stetig* sind. Letzteres ist erforderlich, um die Lösungen von Rekursionsgleichungen iterativ zu berechnen, was dem tatsächlichen Ablauf rekursiver Programme entspricht. Wir lösen dieses Problem durch die Einführung einer partiellen Ordnung auf den Wertebereichen und dem beidseitigen Abschluß der Relationen bezüglich dieser Ordnung. Konstrukte wie die Zuweisung und die Fallunterscheidung werden dahingehend angepaßt. Unverändert bleiben hingegen die Konstrukte für Hintereinanderausführung, nichtdeterministische Auswahl und Rekursion. Das ist wichtig, damit man weiterhin mit den bekannten Axiomen und Gesetzen rechnen kann.

Anschließend leiten wir unter anderem her, daß die neu modellierten Berechnungen die gewünschten Stetigkeits- und Totalitätseigenschaften besitzen. Sind sämtliche Berechnungen definiert, stimmen die neuen Relationen mit den Designs der UTP überein. Andernfalls haben Undefiniertheit und Nichtterminierung keine Auswirkungen, sofern die entsprechenden Werte nicht benötigt werden. Insgesamt erhalten wir eine Theorie, die nichtstrikte Berechnungen in einem imperativen, nichtdeterministischen Kontext beschreibt [Gut08].

Die neue Theorie gestattet die Übertragung der aus funktionalen Programmiersprachen bekannten verzögerten Auswertung. Zum Beispiel kann die Laufzeit eines Programms verbessert werden, indem nur die für das Ergebnis benötigten Teile ausgeführt werden. Eine entsprechende operationale Semantik wird in der Arbeit skizziert. Nachteilig sind der mögliche Mehraufwand durch die Implementierung und die verminderte Vorhersagbarkeit von Laufzeit, Speicherverbrauch und Ausführungsreihenfolge. Dagegen bietet verzögerte Auswertung als Vorteil die Möglichkeit, Programme freier und damit klarer zu beschreiben, da man an weniger Stellen auf Terminierung achten muß.

Bei einer verzögerten Ausführung ist es notwendig, die Abhängigkeiten zwischen den einzelnen Berechnungen zu berücksichtigen. Solche Abhängigkeiten spielen unter anderem auch in optimierenden Programmtransformationen bei Übersetzern eine Rolle. Wir untersuchen ihre Struktur ausgehend von der Beobachtung, daß nichtstrikte Berechnungen mit definierten Ergebnissen nicht von undefinierten Eingaben abhängen. Zwei Eigenschaften, die Abhängigkeiten in Berechnungen modellieren, werden hergeleitet und mit Hilfe einer

zusätzlichen partiellen Ordnung auf den Wertebereichen in eleganter Weise algebraisch beschrieben. Sämtliche behandelten Programmkonstrukte haben diese Eigenschaften. Sie sind aber auch auf Relationen, die neue Konstrukte modellieren, anwendbar. Dies beinhaltet die parallele Komposition von Relationen, siehe etwa [BBH+92], die wir in die UTP einführen, um eine algebraische Behandlung von lokalen Variablen zu ermöglichen.

## 5 Folgen

Abschließend beschreiben wir die Auswirkungen dieser Arbeit auf die UTP selber, auf Werkzeuge und in einem allgemeineren Kontext. Als unmittelbare Folge können wir in der UTP einfacher mit Programmen rechnen sowie Aussagen und Beweise kompakt formulieren. Durch die Bestimmung der Struktur von Designs erhalten wir Querbezüge zur Mathematik und zu weiteren, bereits untersuchten Formalismen für Semantik. Ergebnisse jener Theorien sind damit auf die UTP anwendbar. Die Erweiterung um undefinierte Werte ermöglicht die Modellierung von Ausnahmebehandlung wie sie in vielen Programmiersprachen verfügbar ist. Zu einer effizienteren Ausführung der Programme kann die nun erlaubte verzögerte Auswertung beitragen.

Bei Anwendung der UTP ist eine Rechnerunterstützung erwünscht, etwa um Aussagen teilweise automatisch zu zeigen. Indem wir die Theorie verallgemeinern und dadurch weniger Axiome haben, können solche Werkzeuge effizientere Entscheidungsverfahren einsetzen. Durch die Verwendung spezieller Beschreibungsmittel, etwa der Modallogik, ist der Einsatz bislang nicht anwendbarer Werkzeuge möglich. Werkzeuge können ebenfalls eingesetzt werden, um die Analyse der Abhängigkeiten in verzögerten Berechnungen durchzuführen und, etwa durch die Umsetzung stärkerer Typsysteme, die Verwendung undefinierter Werte zu erkennen.

Aus der Literatur ist bekannt, daß sich viele Eigenschaften von Programmen algebraisch beschreiben lassen, siehe unter anderem [Koz97, ÉL05, DMS06]. Die vorliegende Arbeit ermöglicht das auch in der UTP und belegt, daß der algebraische Ansatz sinnvoll ist und in weiteren Theorien Anwendung finden sollte. Mittelbar kann man von dieser Herangehensweise erhoffen, daß sie zu einer Programmentwicklung auf höherer Abstraktionsebene führt und so der zunehmenden Komplexität von Programmen begegnet. Schließlich können die nichtstrikten Berechnungen zu neuen Konstrukten und Methoden führen und weitere Konzepte funktionaler Programmiersprachen, etwa unendliche Datenstrukturen, für imperative Programme verfügbar machen.

## Literatur

[BBH+92] R. C. Backhouse, P. J. de Bruin, P. Hoogendijk, G. Malcolm, E. Voermans und J. van der Woude. Polynomial Relators (Extended Abstract). In M. Nivat, C. Rattray, T. Rus und G. Scollo, Herausgeber, *Algebraic Methodology and Software Technology*, Seiten 303–326. Springer-Verlag, 1992.

[Coh00] E. Cohen. Separation and Reduction. In R. Backhouse und J. N. Oliveira, Herausgeber, *Mathematics of Program Construction*, Band 1837 von *Lecture Notes in Computer Science*, Seiten 45–59. Springer-Verlag, 2000.

[Con71] J. H. Conway. *Regular Algebra and Finite Machines*. Chapman and Hall, 1971.

[DM01] J. Desharnais und B. Möller. Characterizing determinacy in Kleene algebras. *Information Sciences*, 139(3):253–273, Dezember 2001.

[DMS06] J. Desharnais, B. Möller und G. Struth. Kleene algebra with domain. *ACM Transactions on Computational Logic*, 7(4):798–833, Oktober 2006.

[ÉL05] Z. Ésik und H. Leiß. Algebraically complete semirings and Greibach normal form. *Annals of Pure and Applied Logic*, 3(1–3):173–203, Mai 2005.

[GJSB05] J. Gosling, B. Joy, G. Steele und G. Bracha. *The Java Language Specification*. Addison-Wesley, 3. Auflage, 2005.

[GM06] W. Guttmann und B. Möller. Modal Design Algebra. In S. Dunne und W. Stoddart, Herausgeber, *Unifying Theories of Programming*, Band 4010 von *Lecture Notes in Computer Science*, Seiten 236–256. Springer-Verlag, 2006.

[Gut06] W. Guttmann. Non-termination in Unifying Theories of Programming. In W. MacCaull, M. Winter und I. Düntsch, Herausgeber, *Relational Methods in Computer Science 2005*, Band 3929 von *Lecture Notes in Computer Science*, Seiten 108–120. Springer-Verlag, 2006.

[Gut07] W. Guttmann. *Algebraic Foundations of the Unifying Theories of Programming*. Dissertation, Universität Ulm, 2007.

[Gut08] W. Guttmann. Lazy Relations. In R. Berghammer, B. Möller und G. Struth, Herausgeber, *Relations and Kleene Algebra in Computer Science*, Band 4988 von *Lecture Notes in Computer Science*, Seiten 138–154. Springer-Verlag, 2008.

[HH98] C. A. R. Hoare und J. He. *Unifying theories of programming*. Prentice Hall Europe, 1998.

[HW93] U. Hebisch und H. J. Weinert. *Halbringe*. Teubner, 1993.

[ISO02] ISO/IEC. Information technology: Z formal specification notation: Syntax, type system and semantics. ISO/IEC 13568:2002(E), Juli 2002.

[Koz94] D. Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Information and Computation*, 110(2):366–390, Mai 1994.

[Koz97] D. Kozen. Kleene algebra with tests. *ACM Transactions on Programming Languages and Systems*, 19(3):427–443, Mai 1997.

[Mad96] R. D. Maddux. Relation-algebraic semantics. *Theoretical Computer Science*, 160(1–2):1–85, Juni 1996.

[MS06] B. Möller und G. Struth. WP is WLP. In W. MacCaull, M. Winter und I. Düntsch, Herausgeber, *Relational Methods in Computer Science 2005*, Band 3929 von *Lecture Notes in Computer Science*, Seiten 200–211. Springer-Verlag, 2006.

[Nel89] G. Nelson. A Generalization of Dijkstra's Calculus. *ACM Transactions on Programming Languages and Systems*, 11(4):517–561, Oktober 1989.

[Obj07] Object Management Group. *OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2*, November 2007.

[PJ87] S. L. Peyton Jones. *The implementation of functional programming languages*. Prentice Hall, 1987.

[Sar06] S. Sarstedt. *Semantic Foundation and Tool Support for Model-Driven Development with UML 2 Activity Diagrams*. Dissertation, Universität Ulm, 2006.

[Sch00] W. Schulte. *High-Integrity Compilation and Secure Execution of Java*. Habilitationsschrift, Universität Ulm, 2000.

[SG07] S. Sarstedt und W. Guttmann. An ASM Semantics of Token Flow in UML 2 Activity Diagrams. In I. Virbitskaite und A. Voronkov, Herausgeber, *Perspectives of System Informatics: 6th International Andrei Ershov Memorial Conference, PSI 2006*, Band 4378 von *Lecture Notes in Computer Science*, Seiten 349–362. Springer-Verlag, 2007.

[SHW97] G. Schmidt, C. Hattensperger und M. Winter. Heterogeneous Relation Algebra. In C. Brink, W. Kahl und G. Schmidt, Herausgeber, *Relational Methods in Computer Science*, Kapitel 3, Seiten 39–53. Springer-Verlag, Wien, 1997.

[Wri04] J. von Wright. Towards a refinement algebra. *Science of Computer Programming*, 51(1–2):23–45, Mai 2004.

**Walter Guttmann** wurde 1977 in Großwardein (Rumänien) geboren und kam 1985 nach Deutschland. Nach dem Abitur am Maria-Theresia-Gymnasium Augsburg begann er 1996 das Informatikstudium an der Universität Ulm. Seinen Zivildienst leistete er 1997–1998 beim Blutspendedienst des Bayerischen Roten Kreuzes im Institut Augsburg. 2001 erhielt er das Diplom in Informatik. Seit 2002 ist er akademischer Mitarbeiter bei Prof. Dr. Helmuth Partsch am Institut für Programmiermethodik und Compilerbau der Universität Ulm. Mit der hier vorgestellten Arbeit promovierte er 2007. Im Mittelpunkt seiner Forschung stehen algebraische Strukturen und Methoden in der Informatik, die Semantik von Programmier- und Spezifikationssprachen, sowie die Transformation funktionaler Programme. Als Teilnehmer und Coach beteiligt er sich seit 1997 bei Europa- und Weltmeisterschaften des ACM-Programmierwettbewerbs.