Mehrpässige Compilation mit finalen Coalgebren

Peter Padawitz
Universität Dortmund, Germany

4. September 2007

1 Einleitung

Prinzipiell lässt sich jedes Modell eines Softwaresystems einer von zwei Klassen zuordnen: den konstruktorbasierten white-box-Modellen oder den destruktorbasierten black-box-Modellen. Ein größeres System setzt sich konstruktor- bzw. destruktorbasierter Teilen zusammen. Endliche, durch kontextfreie Grammatiken beschriebene Datenstrukturen gehören in die erste Klasse, unendliche Datenstrukturen (z.B. Ströme) sowie Automaten, Transitionssysteme und alle Arten von zustands- und objektorientierten Modellen in die zweite Klasse. Konstruktor- wie destruktorbasierte Modelle verwenden Funktionsterme. Im ersten Fall liefern diese die Bausteine des Modells selbst, im zweiten Fall beschreiben sie "Versuchsaufbauten", mit deren Hilfe das Modell und sein Verhalten beobachtet wird und somit seine Bausteine identifiziert werden.

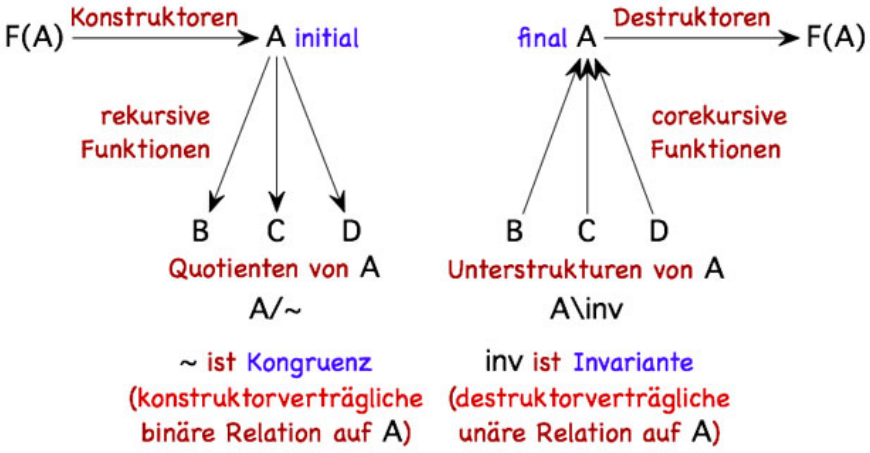

Figure 1. *Das Tai Chi der Systemmodellierung*

Die Dualität zwischen konstruktor- und destruktorbasierten Modellen zieht sich durch die gesamte, für die beiden Klassen jeweils typische Methodik, von den in konkreten Entwürfen verwendeten mathematischen Konzepten über die Verifikationsverfahren bis hin zu den für Implementierungen der Modelle charakteristischen Datenstrukturen, Algorithmen und Programmierstilen. So sind zum Beispiel Funktionsdefinitionen mit Hilfe von *Rekursionsschemata* und Abstraktion durch *Quotientenbildung* typisch für konstruktorbasierte Entwürfe, während die, als Attribute oder Methoden bekannten, Funktionen eines destruktorbasierten Entwurfs *corekursiv* definiert werden und die Einführung von Invarianten zur *Unterstrukturbildung* führt (siehe Fig. 1).

Methoden der konstruktorbasierten Modellierung werden seit 30 Jahren entwickelt und haben inzwischen, mehr oder weniger explizit, Eingang in die Softwaretechnik gefunden (siehe z.B. [7, 17, 3]). Destruktorbasierte Modellierung hat zwar, vor allem in der Automaten- und Systemtheorie, noch ältere Wurzeln. Die

Dualität zwischen beiden Ansätzen wurde jedoch erst mit den vor gut 10 Jahren im Rahmen der Kategorientheorie begonnenen Forschung über *Coalgebren* deutlich (siehe z.B. [2, 6, 22, 12, 11, 23, 9, 15, 10]). Hier gibt's aber noch viel zu tun, was die Prägnanz und Universalität der Grundkonzepte, das adäquate Rechnen und (halb)automatische Beweisen sowie die Integration beider Ansätze anbelangt.[1] Die Säulen der konstruktorbasierten Modellierung in Gestalt von Termmodellen und rekursiven Funktionsdefinitionen haben im dualen Ansatz noch kein ähnlich anschauliches Gegenstück gefunden. Das liegt vor allem an der vermutlich größeren Bandbreite destruktorbasierter gegenüber konstruktorbasierten Modellen, die wiederum damit zusammenhängt, dass ihre Objekte prinzipiell unsichtbar sind und nur indirekt über die Anwendung von Attribut- und Transitionsfunktionen identifizierbar sein müssen. Fig. 2 zeigt links einen Term, bestehend aus Konstruktoren, also das typische Element einer *initialen* Algebra, während rechts das typische Element einer *finalen* Coalgebra zu sehen ist: eine Folge von "Experimenten" in Form von Destruktortermen, die auf das durch diese beschriebene Objekt (die kleine Box) angewendet werden und durch die in ihren Blättern (einem Blatt pro Term) angezeigten Attributwerte das Objekt identifizieren. Der Produktbildung im Konstruktorterm entspricht die Summenbildung (Auswahl) in den Destruktortermen. Die dick gezeichneten Pfade sollen den jeweiligen "Datenfluss" andeuten.

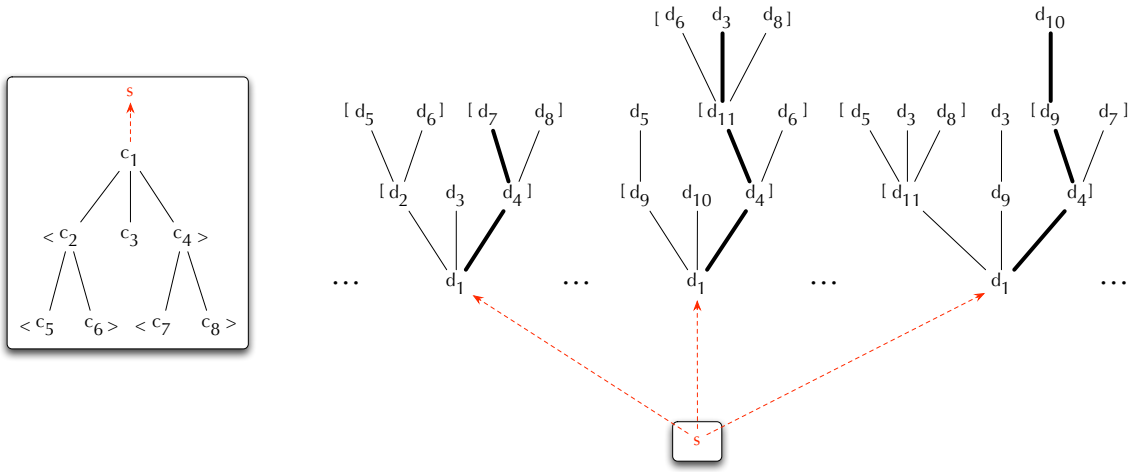

Figure 2. *Zwei Objekte eines konstruktorbasierten initialen bzw. destruktorbasierten finalen Modells*

Vieles wird schon jetzt klarer. Alles, was sich im Laufe der letzten 30 Jahre im "Teilchenzoo" formaler Methoden angesammelt hat, verspricht unter dem Gesichtspunkt der Dualität von Algebra und Coalgebra in seinem jeweiligen Kern besser verstanden, abgrenzbar und gegebenfalls entwicklungsfähiger zu werden.

2 Grammatiken und Algebren

Ich möchte in diesem Vortrag eine Klasse von Programmieraufgaben herausgreifen, zu deren Lösung konstruktorbasierte Modellierung (und Implementierung!) besonders geeignet ist: die Übersetzung oder Interpretation einer kontextfreien Sprache in eine beliebige Zielsprache. In seiner Grundstruktur ist das Vorgehen denkbar einfach: Eine CF-Grammatik $G = (N, T, P, S)$ wird zur mehrsortigen Signatur $\Sigma(G)$, in dem N als Sortenmenge betrachtet wird und jede Produktion $p = (s \rightarrow w_0 s_1 w_1 \ldots s_n w_n)$ von G mit $w_i \in T^*$ und $s_i \in N$ zum Funktionssymbol (Konstruktor) $f : s_1 \times \ldots \times s_n \rightarrow s$. $\Sigma(G)$ nennt man üblicherweise die *abstrakte Syntax* von G und die Menge $T_{\Sigma(G)}$ der $\Sigma(G)$-Grundterme *Syntaxbäume* von G. $T_{\Sigma(G)}$ ist die Zwischensprache auf dem Weg von der Quellsprache $L(G)$ der von G erzeugten Worte in die gewünschte Zielsprache A. Ein Übersetzer von $L(G)$ nach A ist nichts anderes als die Komposition einer

[1] [16] und [19, 20] arbeiten in Richtung konstruktor- und destruktorbasierte Modelle umfassender Spezifikationssprachen.

Syntaxanalyse-Funktion $parse : L(G) \to T_{\Sigma(G)}$ mit der Funktion $eval^A : T_{\Sigma(G)} \to A$, die Syntaxbäume in der zur $\Sigma(G)$-Algebra erweiterten Zielsprache A auswertet. $eval^A$ existiert, weil $T_{\Sigma(G)}$ selbst eine $\Sigma(G)$-Algebra ist (mit Elementen wie dem Konstruktorterm in Fig. 2). Darüberhinaus ist $T_{\Sigma(G)}$ die *initiale* $\Sigma(G)$-Algebra, d.h. $eval^A$ existiert nicht nur, sondern ist der einzige $\Sigma(G)$-Homomorphismus (d.i. eine mit den Interpretationen von Σ in $T_{\Sigma(G)}$ bzw. A verträgliche Abbildung) von $T_{\Sigma(G)}$ nach A. Daraus ergeben sich *rekursive* Definitionen von $eval^A$ und auch von *parse*, denn auch die Quellsprache $L(G)$ lässt sich zur $\Sigma(G)$-Algebra machen, womit *parse* zur *Retraktion* bzgl. $eval^{L(G)}$ wird. (G is genau dann *eindeutig*, wenn *parse* auch eine Coretraktion, also bijektiv ist.)

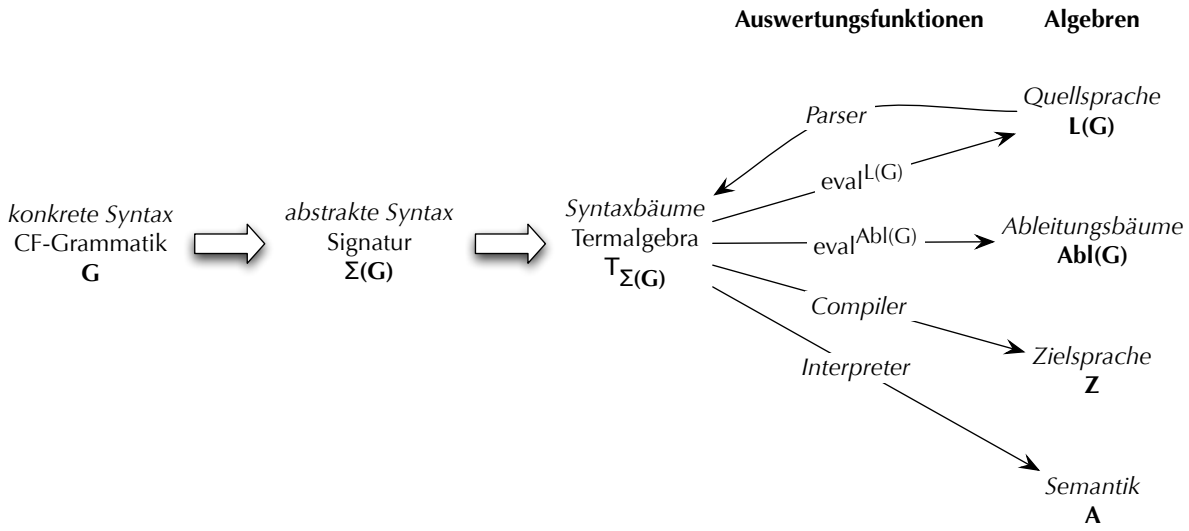

Figure 3. *Von Grammatiken zu Algebren*

Ein aus rekursiven Definitionen über der abstrakten Syntax von G zusammengesetzter Compiler hat den großen Vorteil, dass er modular aufgebaut ist: jede Regel von G liefert einen Konstruktor f von $\Sigma(G)$ und damit eine Komponente von *parse* mittels der Definition von $eval^{L(G)}$ auf Syntaxbäumen mit Wurzel f sowie eine Komponente von $eval^A$, nämlich die Definition von $eval^A$ auf Syntaxbäumen mit Wurzel f).

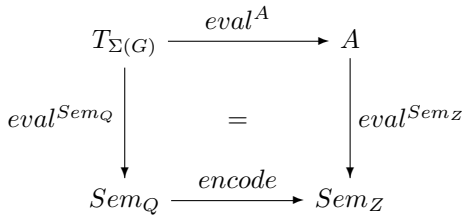

Figure 4. *Korrektheit eines Compilers von der Zwischen- in die Zielsprache*

Formuliert man nicht nur die Zielsprache A, sondern auch deren Semantik Sem_Z sowie die Semantik Sem_Q der Quellsprache als $\Sigma(G)$-Algebren, dann sind – wegen der Initialität von $T_{\Sigma(G)}$ – der Übersetzer $eval^A$ und die Interpretation von I_Q von $\Sigma(G)$-Termen in Sem_Q eindeutig bestimmt (siehe Fig. 4). Zur Korrektheit des Compilers bliebe noch zu zeigen, dass auch *encode* und $eval^{Sem_Z}$ $\Sigma(G)$-Homomorphismen sind. Dann wären auch die Kompositionen $eval^{Sem_Z} \circ comp$ und $encode \circ eval^{Sem_Q}$ homomorph und – wieder wegen der Initialität von $T_{\Sigma(G)}$ – identisch, d.h. das Diagramm von Fig. 4 wäre kommutativ.

Die algebraische Sicht auf Compiler und die Vorteile, die man daraus ziehen kann (z.B. bei ihrer Verifikation) ist beileibe nicht neu (siehe z.B. [7, 18, 24]), scheint aber in den letzten 10 Jahren ziemlich in

Vergessenheit geraten zu sein. Schaut man sich beispielsweise die zum Teil sehr umständlichen Ansätze zur Verarbeitung von XML-Dateien an, dann sucht man vergeblich nach einer Begründung, warum das o.g. klassische Vorgehen bei der Compiler-Entwicklung für XML und deren Anfragesprachen nicht geeignet sein soll.

3 Attributierte Übersetzung

Auch attributierte Syntaxbäume fügen sich zunächst gut in das konstruktorbasierte Schema ein. Zwar scheint die Verwendung von Attributen nach dem in §1 Gesagten Destruktoren zu erfordern. Akzeptiert man jedoch Algebren mit Trägermengen, die aus Funktionen bestehen, dann ist – zumindest bei einpässiger Übersetzung – keine destruktorbasierte Modellierung erforderlich. Bei mehrpässiger Übersetzung kommt allerdings ein Datentyp ins Spiel, der sich als finale Coalgebra entpuppt. Mehrere Pässe führen die Übersetzung nämlich aus dem einfachen Schema der (bottom-up-)Auswertung von Syntaxbäumen (Anwendung von $eval^A$; s.o.) heraus und verlangen die explizite Einführung attributierter, also mit Elementen einer potenziell unendlichen Menge markierter Syntaxbäume in das Modell. Will man die ursprünglichen Konstruktoren von $\Sigma(G)$ beibehalten und Attributwerte ohne Veränderung der Baumstruktur hinzufügen, dann kann man nicht mehr von einem Termmodell sprechen. An die Stelle der Auswertungsfunktion tritt die homomorphe (d.h. hier: destruktorverträgliche) *Färbung* der Baumknoten. Für jede Sorte s von $\Sigma(G)$ (Nichtterminal von G) gibt es einen Destruktor $d_s : s \to \coprod_{f:w \to s \in \Sigma(G)} w$ und eventuell weitere Destruktoren a mit Typen der Form $s \to s'$, wobei s' eine Sorte ist, die nicht zu $\Sigma(G)$ gehört, sondern durch die Menge der Werte des durch a repräsentierten Attributs interpretiert wird.

In Anlehnung an Automaten (die bei fast allen coalgebraischen Begriffen eine gute Intuition bieten) nennen wir die Funktionen d_s (aus $\Sigma(G)$ gebildete) *Transitionsfunktionen* und die Attribute repräsentierenden Destruktoren *Ausgabefunktionen*. Die inneren Knoten der in Fig. 2 als Bäume dargestellten Destruktorterme wären hier stets mit Transitionsfunktionen, die Blätter immer mit Ausgabefunktionen markiert. Aus den o.g. Typen der Destruktoren ergibt sich, dass jedes Element der finalen Coalgebra (gemäß Fig. 2) eindeutig durch einen mit typkonformen Attributwerten knotenmarkierten Syntaxbaum (also einem Term der ursprünglichen Konstruktorsignatur!) repräsentieren lässt. In dieser Coalgebra ist d_s als Umkehrfunktion *aller* Konstruktoren mit Zielsorte s interpretiert. Sie entfernt die Wurzel ihres jeweiligen Argumentbaums und gibt einen (!) Unterbaum zurück.

Das adäquate Modell für attributierte Syntaxbäume ist also eine aus $\Sigma(G)$ und den Ausgabefunktionen gebildete finale Coalgebra. Tatsächlich enthält sie auch alle unendlichen Syntaxbäume. Da die Transitionsfunktionen d_s aber aus endlichen Bäumen nur endliche Bäume machen können (sie geben ja nur Unterbäume zurück), ist die Untercoalgebra der endlichen Syntaxbäume selbst final.

Welchen Nutzen man aus der Finalität des Modells ziehen kann, ist noch nicht klar. Ich vermute, dass sie bisher unbekannte, aber möglicherweise adäquatere, vielleicht sogar effizientere, Realisierungen nahelegt als diejenige, die Bäume direkt zu implementieren und Attributwerte in Knotenzellen abzulegen. Im Rahmen von XML tauchen attributierte Syntaxbäume übrigens unter dem Begriff *data trees* auf (siehe z.B. [4]).

In §2 haben wir gezeigt, dass ein Compiler neben dem Parser, der Wörter in Syntaxbäume übersetzt, aus der Erweiterung der Zielsprache A zu einer $\Sigma(G)$-Algebra besteht. Attributierte Übersetzung[2] heißt im algebraischen Kontext, dass die Trägermengen von A aus Funktionen von den Wertebereichen *vererbter* Attribute in die Wertebereiche *abgeleiteter* Attribute bestehen. Sei also $A_s = B_s \to C_s$. Der allgemeine Ansatz für die Interpretation eines Konstruktors $f : s_1 \times \ldots \times s_n \to s$ von $\Sigma(G)$ in A ist eine bedingte Gleichung der Form

$$f^A(g_1, \ldots, g_n)(a_0) = e_n \text{ where } \begin{aligned} a_1 &= g_1(e_0) \\ a_2 &= g_2(e_1) \end{aligned}$$

[2] Das Wesentliche darüber steht bereits in [14].

$$\begin{aligned} &\vdots \\ &a_n = g_n(e_{n-1}) \end{aligned} \tag{3.1}$$

wobei für alle $1 \leq i \leq n$ $g_i \in A_{s_i} = B_{s_i} \to C_{s_i}$ ist. B_s und C_s sind i.d.R. mehrstellige Produkte, deren Komponenten die Wertebereiche mehrerer Attribute bilden. Mehrpässige Übersetzung wird erforderlich, wenn (3.1) zyklische Abhängigkeiten aufweist. Dann liefert (3.1) nämlich noch gar keine Definition von f^A. Solche Zyklen lassen sich jedoch automatisch ermitteln und mit einem einfachen Algorithmus in eine Zerlegung der gesamten Attributmenge überführen (siehe z.B. [13, 21]). Die Teilmengen der Zerlegung, sagen wir $At^1, \ldots, At^r$, können dann in r Pässen berechnet werden, d.h. anstelle einer Algebra A erhalten wir r Algebren $A^1, \ldots, A^r$ mit folgenden Trägermengen: Sei $1 \leq k \leq r$ und T^k die Menge der attributierten Syntaxbäume, deren Knoten mit Werten der im k-ten Pass berechneten Attribute markiert sind. Dann ist

$$A^k_s \;=\; B^k_s \to (T^1_s \to (\ldots (T^{k-1}_s \to T^k_s) \ldots)).$$

(Die vererbten Attribute von B^k_s bestimmen zusammen mit den in den Knoten der Syntaxbäume von $T^1, \ldots, T^{k-1}$ gespeicherten Attribute die Attributierung des nächsten Syntaxbaum, der dann zu T^k_s gehört.) Die Interpretation des Konstruktors $f : s_1 \times \ldots \times s_n \to s$ in A^k ist eine Projektion von (3.1) auf die Attribute von At^k: π^k projiziert jedes Attributwerttupel a auf das Tupel der Komponenten von a, die Werte von Attributen aus At^k sind. Für alle $1 \leq i \leq n$ sei $g_i \in A^k_{s_i} = B^k_{s_i} \to C^k_{s_i}$. Für alle $1 \leq i < k$ sei $t^i \in T^i_s$.

$$\begin{aligned} f^{A^k}(g_1, \ldots, g_n)(\pi^k(a_0))(t^1) \ldots (t^{k-1}) = \pi^k(e_n)[t^k_1, \ldots, t^k_n] \\ \text{where } t^k_1 = g_1(\pi^k(e_0))(t^1_1) \ldots (t^{k-1}_1) \\ \vdots \\ t^k_n = g_n(\pi^k(e_{n-1}))(t^1_n) \ldots (t^{k-1}_n) \end{aligned} \tag{3.2}$$

Der Ergebnisbaum $\pi^k(e_n)[t^k_1, \ldots, t^k_n]$ des k-ten Passes hat den Wurzeleintrag $\pi^k(e_n)$ und die Unterbäume $t^k_1, \ldots, t^k_n$. Die – im Gegensatz zu (3.1) – ausführbare Definition von f^A lautet schließlich wie folgt:

$$\begin{aligned} f^A(g_1, \ldots, g_n)(a_0) = (root(t^1), \ldots, root(t^r)) \\ \text{where } t^1 = f^{A^1}(g_1, \ldots, g_n)(\pi^1(a_0)) \\ t^2 = f^{A^2}(g_1, \ldots, g_n)(\pi^2(a_0)(t^1)) \\ \vdots \\ t^r = f^{A^r}(g_1, \ldots, g_n)(\pi^r(a_0))(t^1) \ldots (t^{r-1}) \end{aligned} \tag{3.3}$$

Unter Verwendung von Transitions- und Ausgabefunktionen der Darstellung von T^k als finale Coalgebra (s.o.) werden (3.2) und (3.3) unabhängig von der Baumdarstellung:

$$\langle at^k_s, d_s \rangle (f^{A^k}(g_1, \ldots, g_n)(\pi^k(a_0))(t^1) \ldots (t^{k-1})) = (\pi^k(e_n), \iota_f(t^k_1, \ldots, t^k_n)) \text{ where ... (s.o.) ...} \tag{3.2}$$

$$f^A(g_1, \ldots, g_n)(a_0) = (at^1_s(t^1), \ldots, at^r_s(t^r)) \text{ where ... (s.o.) ...} \tag{3.3}$$

$at^k_s : T^k_s \to B^k_s \cup C^k_s$ bezeichnet die Ausgabefunktion des k-ten Passes für Bäume der Sorte s und ι_f die Einbettung des Produktes $T^k_{s_1} \times \ldots \times T^k_{s_n}$ in die Summe $\coprod_{g:w \to s \in \Sigma(G)} w$ (s.o.).

Zusammen mit dem Zerlegungsalgorithmus liefern (3.2) und (3.3) eine Lösung des durch (3.1) gegebenen Gleichungssystems. Die Allgemeinheit von (3.1) lässt vermuten, dass es außer mehrpässiger Compilation weitere Anwendungen gibt für die Lösungssuche mit Hilfe attributierter Syntaxbäume.

Literatur

[1] J. Adamek, *Introduction to Coalgebra*, Theory and Applications of Categories 14 (2005) 157-199

[2] M.A. Arbib, E.G. Manes, *Parametrized Data Types Do Not Need Highly Constrained Parameters*, Information and Control 52 (1982) 139-158

[3] E. Astesiano, H.-J. Kreowski, B. Krieg-Brückner, eds., *Algebraic Foundations of Systems Specification*, IFIP State-of-the-Art Report, Springer 1999

[4] H. Björklund, M. Bojanczyk, *Bounded depth data trees*, ICALP'07, LNCS 4596 (2007) 862-874

[5] C. Cîrstea, *A Coalgebraic Equational Approach to Specifying Observational Structures*, Theoretical Computer Science 280 (2002) 35-68

[6] J. Goguen, G. Malcolm, *A Hidden Agenda*, Theoretical Computer Science 245 (2000) 55-101

[7] J.A. Goguen, J.W. Thatcher, E.G. Wagner, *An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types*, in: R. Yeh, ed., Current Trends in Programming Methodology 4, Prentice-Hall (1978) 80-149

[8] J.A. Goguen, J.W. Thatcher, E.G. Wagner, J.B. Wright, *Initial Algebra Semantics and Continuous Algebras*, J. ACM 24 (1977) 68-95

[9] H. P. Gumm, *Universelle Coalgebra*, in: Th. Ihringer, *Allgemeine Algebra*, Heldermann Verlag 2003

[10] I. Hasuo, *Modal Logics for Coalgebras - A Survey*, Report, Tokyo Institute of Technology (2003)

[11] B. Jacobs, *Exercises in Coalgebraic Specification*, in: R. Backhouse, R. Crole, J. Gibbons, eds., *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, Springer LNCS 2297 (2002) 237-280

[12] B. Jacobs, J. Rutten, *A Tutorial on (Co)Algebras and (Co)Induction*, EATCS Bulletin 62 (1997) 222-259

[13] U. Kastens, *Übersetzerbau*, Oldenbourg, 1990

[14] D. Knuth, *Semantics of Context-Free Languages*, Mathematical Systems Theory 2 (1968) 127-145; Correction: Math. Systems Theory 5 (1971) 95-96

[15] A. Kurz, *Coalgebras and Modal Logic*, Course Notes for ESSLLI 2001, CWI Amsterdam

[16] Till Mossakowski, Horst Reichel, Markus Roggenbach, Lutz Schröder, *Algebraic-coalgebraic specification in CoCASL*, to appear in: J. of Logic and Algebraic Programming 67 (2006) 121-143

[17] J. Meseguer, J.A. Goguen, *Initiality, Induction and Computability*, in: M. Nivat, J. Reynolds, eds., Algebraic Methods in Semantics, Cambridge University Press (1985) 459-541

[18] F.L. Morris, *Advice on Structuring Compilers and Proving Them Correct*, Proc. ACM POPL (1973) 144-152

[19] P. Padawitz, *Dialgebraic Specification and Modeling*, in Vorbereitung, fldit-www.cs.uni-dortmund.de/∼peter/Dialg.pdf, Dortmund 2007

[20] P. Padawitz, *Expander2: A Formal Methods Presenter and Animator*, fldit-www.cs.uni-dortmund.de/∼peter/Expander2.html, Dortmund 2007

[21] P. Padawitz, *Übersetzerbau*, Vorlesungsskript, Dortmund 2007

[22] H. Reichel, *An Approach to Object Semantics based on Terminal Coalgebras*, Math. Structures in Comp. Sci. 5 (1995) 129-152

[23] J.J.M.M. Rutten, *Universal Coalgebra: A Theory of Systems*, Theoretical Computer Science 249 (2000) 3-80

[24] J.W. Thatcher, E.G. Wagner, J.B. Wright, *More on Advice on Structuring Compilers and Proving Them Correct*, Theoretical Computer Science 15 (1981) 223-249