Prof. Dr. H.-D. Ehrich
Universität Dortmund, Abteilung Informatik

Zur Theorie abstrakter Datentypen

1. Einleitung

Unter einem *Datentyp* versteht man allgemein eine Menge von Datenobjekten mit zugehörigen Operationen auf diesen Objekten. Es liegt daher nahe, Datentypen mathematisch als *Algebren* aufzufassen, wie es erstmals in [10, 14] vorgeschlagen und in [1,8] präzisiert wurde. Wir folgen hier dieser Auffassung. Danach sind *abstrakte* Datentypen dann abstrakte Algebren, d. h. Isomorphieklassen von Algebren.

In der Informatik haben wir es sehr oft mit der *Beschreibung* von Dingen (Syntax) und der Beziehung zu den beschriebenen

2. *Wie* werden diese Klassen *beschrieben*?
3. Wie ist die *Klasse* der durch eine *Spezifikation* beschriebenen *Datentypen* definiert?
4. Wie müssen *Korrektheitskriterien* beschaffen sein?

So entspricht z. B. ein und dieselbe axiomatische Spezifikation S bei verschiedenen Autoren [1, 7] unterschiedlichen Klassen $K(S)$. Diese recht verwirrende Situation ergibt sich insbesondere daraus, daß nicht in jedem Fall praktische Bedingungen der Programmierung konsequent beachtet wurden. Insbesondere die Beantwortung der ersten Frage erscheint fundamental: Welche Klassen sind in der Programmierung relevant?

Da diese vier Fragestellungen gerade die praktisch interessierenden sind, führt eine Uneinheitlichkeit der theoretischen Untersuchungen zu diesen Problemen zu einer Verunsicherung potentieller Nutzer.

3. Abstrakte Datentypen

Ausgangspunkt der theoretischen Untersuchungen zu den im Punkt 2 genannten Fragen muß folgende Überlegung sein: An welcher praktischen Zielstellung muß eine Abstraktion (von einem konkreten Datentyp) orientiert sein? Diese Zielstellung ergibt sich aus dem entscheidenden Problem: Was muß ein Nutzer über die *Wirkung* eines Datentyps wissen, um ihn anwenden zu können?

Dementsprechend soll im folgenden die Wirkung eines Datentyps definiert werden, und zwar als spezielles Wechselverhältnis der Funktionen des Datentyps. Datentypen, die dieselbe Wirkung besitzen, werden dann als *äquivalent* bezeichnet. Äquivalenzklassen von Datentypen werden schließlich *abstrakte Datentypen* genannt. Abstrakte Datentypen sind demnach Klassen von Datentypen gleicher Wirkung und stellen somit für die Programmierung relevante Klassen von Datentypen dar.

Bevor näher auf den Wirkungsbegriff eingegangen wird, soll erläutert werden, wie dieses Herangehen zur Beantwortung der in Punkt 2 genannten Fragen führt:

1. Abstrakte Datentypen sind die interessierenden Klassen von Datentypen.
2. Da die Klassenbildung in bezug auf die Wirkung eines Datentyps und nicht in bezug auf eine bestimmte Spezifikationstechnik erfolgt ist, ist die Art der Spezifikationstechnik zunächst offen. Damit kann man die Spezifikationsmethode nach praktischen Kriterien auswählen bzw. mehrere Methoden gleichzeitig benutzen.
3. Das Verhältnis Spezifikation $S \to$ Klasse $K(S)$ ist so zu gestalten, daß einer Spezifikation S genau ein abstrakter Datentyp $K(S)$ zugeordnet wird.
4. Mit $C(S)$ sei die Klasse der mit dem Korrektheitskriterium C bzgl. S verifizierbaren Datentypen bezeichnet. Wichtige Eigenschaften von C sind dann seine Widerspruchsfreiheit $\big(C(S) \subseteq K(S)\big)$ und seine Vollständigkeit $\big(C(S) \supseteq K(S)\big)$.

Im folgenden soll der Begriff der *Wirkung* eines Datentyps näher erläutert werden. Dazu sei zunächst eine Klassifizierung der Funktionen eines Datentyps angegeben, die auf Guttag [7] zurückgeht.

Def. 2 Sei $d = (r, D, \Phi)$ ein Datentyp. $\varphi \in \Phi$ heißt *Konstruktor*, wenn ihr Wertevorrat r ist. Ansonsten heißt φ *Selektor*.

Konstruktoren sind somit Funktionen, mit deren Hilfe Elemente der Repräsentation erzeugt, aufgebaut und geändert werden (z. B. new1, append1, delete1). Ihre Wirkung ist zunächst nicht entscheidend, da die spezielle Repräsentation r nicht wesentlich ist. In unserem Beispiel queue1 und queue2 wurden ja gerade unterschiedliche Repräsentationen benutzt. Um nun Elemente der Repräsentation nach ihrem Informationsgehalt einschätzen zu können, ist es wichtig, über Selektoren zu verfügen (z. B. first1, isempty1). Mit Hilfe der Selektoren kann Information, die vorher durch Konstruktoren in Elementen der Repräsentation abgelegt wurde, abgefragt werden.

Entscheidend für die Wirkung eines Datentyps ist nunmehr das Wechselverhältnis seiner Konstruktoren und Selektoren:

Def. 3 a) Die *Wirkung eines Datentyps* $d = (r, D, \Phi)$ wird durch die Resultate der Selektoren $\varphi_s \in \Phi$ bei der Anwendung auf Elemente der Repräsentation $x \in r$ bestimmt, die aus endlich vielen Anwendungen von Konstruktoren $\varphi_k \in \Phi$ hervorgegangen sind.

b) Datentypen gleicher Wirkung heißen *äquivalent* (Notation $d \sim d'$).

c) Ein *abstrakter Datentyp* ist eine Äquivalenzklasse von Datentypen: $K(d) = \{d' \mid d' \sim d\}$.

Äquivalente Datentypen $d_1 = (r_1, D_2, \Phi_1)$ und $d_2 = (r_2, D_2, \Phi_2)$ besitzen demnach dasselbe „Konstruktur-Selektor-Verhalten". Sind $x_1 \in r_1$ und $x_2 \in r_2$ zwei Elemente, die in d_1 bzw. d_2 auf dieselbe Weise durch entsprechende Konstruktoren erzeugt wurden, so erhält man bei der Anwendung entsprechender Selektoren auf sie dieselben Resultate. Zunächst sei ein Beispiel für die Wirkung von queue1 gegeben:

$$x_1 = \text{delete1}(\text{append1}(\text{append1}(\text{append1}(\text{new1}, 'a'), 'b'), 'c')) = ('b', 'c') \in \text{sequence}$$

$\text{first1}(x_1) = 'b'$, $\text{isempty1}(x_1) = \text{false}$

Für die analogen Anwendungsfolge besitzt queue2 dieselbe Wirkung:

$$x_2 = \text{delete2}(\text{append2}(\text{append2}(\text{append2}(\text{new2}, 'a'), 'b'), 'c')) = (('a', 'b', 'c', \ldots), 2, 3) \in \text{record}$$

$\text{first2}(x_2) = 'b'$, $\text{isempty2}(x_2) = \text{false}$

Obwohl in der internen Realisierung verschieden (x_1 bzw. x_2), besitzen queue1 und queue2 dasselbe Konstruktor-Selektor-Verhalten und gehören demnach derselben abstrakten Datentypen an. Sie besitzen dieselbe Wirkung für einen Nutzer (first-in-first-out-Prinzip).

4. Spezifikation abtrakter Datentypen

Nachdem ein abstrakter Datentyp als Äquivalenzklasse von Datentypen definiert wurde, kann nun als gesondertes Problem die Frage nach seiner Beschreibung (*Spezifikation*) gestellt werden: Wie kann ein abstrakter Datentyp $K(d)$ spezifiziert werden?

Da die Klassenbildung bisher nur hinsichtlich der Wirkung von Datentypen und nicht in bezug auf eine a priori ausgezeichnete Beschreibungsmethode eingeführt wurde, ist eine Vereinheitlichung in der Literatur untersuchter Methoden erzielt worden: Abstrakte Datentypen können sowohl *axiomatisch*, als auch *operationell* beschrieben werden.

a) axiomatische Spezifikation [1, 7]: Eine Menge von Axiomen charakterisiert Eigenschaften der Funktionen. Für K(queue1) ergibt sich z. B. folgende Axiomatisierung:

isempty(new) = true

isempty(append(s, x)) = false

first(new) = ERROR

first(append(s, x)) = **if** isempty(s) **then** x **else** first(s)

Dingen (Semantik) zu tun. In der Theorie der abstrakten Datentypen sind insbesondere Spezifikationen durch *Gleichungen* ausgiebig diskutiert worden [1–9, 13], motiviert durch vielversprechende Anwendungen beim Entwurf von Programmen [10, 11, 14].

Diese Arbeit stellt i. w. die in [5] beschriebenen Ansätze und einige Ergebnisse im Überblick dar. Bezüglich der Details, insbesondere der Beweise zu den Sätzen, sei daher auf [5] verwiesen.

2. Spezifikation und Algebren

Eine *Gleichungs-Spezifikation* (im folgenden kurz *Spezifikation*) ist ein Tripel $D = \langle S, \Omega, E\rangle$, wobei S eine Menge von *Sorten* ist, $\Omega: \bar{\Omega} \to S^+$ ist eine *Signatur*, und $E \subset YT_\Omega \times YT_\Omega$ ist eine Menge von (Ω-)*Gleichungen*. Die Notation $(\omega \to xs) \in \Omega$, $x \in S^*$, $s \in S$, gibt zu jedem Operationssymbol ω die Argumentsorten x und die Wertsorte s an. Ist $x = s_1 s_2 \ldots s_n$, $s_i \in S$, so werden wir auch die Notation $\omega: s_1 \times s_2 \times \cdots \times s_n \to s$ anstelle von $(\omega \to xs)$ benutzen. Die Notation YT_Ω bezeichnet die (sortierte) Menge aller Ω-Terme über der (sortierten) Variablenmenge $Y: \bar{Y} \to S$, die wie üblich gebildet wird. Zum späteren Gebrauch geben wir zwei Beispiele.

Beispiel 1: $\mathrm{NAT} = \langle\{\mathrm{nat}\}, \{0: \to \mathrm{nat}, \mathrm{succ}: \mathrm{nat} \to \mathrm{nat}\}, \emptyset\rangle$ spezifiziert die natürlichen Zahlen.

Beispiel 2: $\mathrm{ALG} = \Big\langle\{\mathrm{alg}\}, \{\emptyset: \to \mathrm{alg}, \tau: \mathrm{alg} \to \mathrm{alg}, \cdot: \mathrm{alg} \times \mathrm{alg} \to \mathrm{alg}\} \quad \Big\{\emptyset \cdot \emptyset = \emptyset, a \cdot \emptyset = a, \emptyset \cdot a = a, \sigma(a) \cdot \sigma(b) = \sigma\big(\sigma(a \cdot b)\big)\Big\}\Big\rangle$ spezifiziert eine algebraische Struktur mit drei Operationen der Stelligkeiten 0, 1 und 2, die die gegebenen Gleichungen erfüllt.

Die Modelle einer Spezifikation D bilden eine gleichungsdefinierte Klasse von Algebren. Zusammen mit allen ihren Algebra-Morphismen bilden sie eine Kategorie **D-alg**. Unter den vielen Eigenschaften von **D-alg** ist die Existenz von initialen Objekten besonders wichtig. Sei **init** D diese Klasse (wir werden manchmal auch etwas ungenau einen ihrer Repräsentanten so bezeichnen). Die Idee der Initialen-Algebra-Semantik ist es, **init** D als abstraktes Standard-Modell zu D aufzufassen. Sind $D = \langle S, \Omega, E\rangle$ und $D^0 = \langle S, \Omega, \emptyset\rangle$, so nennt man **init** D^0 die *abstrakte Syntax*, und der (wegen der Initialität eindeutige) Morphismus SEM: **init** $D^0 \to$ **init** D gibt die Initiale-Algebra-Semantik von D (vgl. [1]).

3. Spezifikations-Morphismen

Um Beziehungen zwischen Spezifikationen zu studieren, führen wir einen Morphismenbegriff wie folgt ein. Seien $D_i = \langle S_i, \Omega_i, E_i\rangle$, $i = 0, 1$, Spezifikationen. Ein (Spezifikations-) Morphismus $f: D_0 \to D_1$ wird gegeben durch zwei Abbildungen $h: S_0 \to S_1$ und $g: \bar{\Omega}_0 \to \bar{\Omega}_1$, sofern sie den folgenden beiden Bedingungen genügen: (1) $\Omega_0 h = g\Omega_1$ (in unserer Notation ist z. B. $xfg := g\big(f(x)\big)$) und (2) $\mathring{E}_0\hat{f} \subset \mathring{E}_1$. In (1) haben wir die buchstabenweise Erweiterung von h aus S^* ebenfalls mit h bezeichnet. In (2) bezeichnen wir mit $\mathring{E}_i$ die Menge der konstanten Gleichungen (ohne Variable), die aus E_i herleitbar sind, und $\mathring{E}_0\hat{f}$ ist die Menge von Gleichungen, die wir aus $\mathring{E}_0$ dadurch erhalten, daß wir (in irgendeiner Darstellung) jedes Vorkommen eines Operationssymbols $\omega \in \bar{\Omega}_0$ durch $\omega g \in \bar{\Omega}_1$ und jedes Vorkommen eines Sortensymbols $s \in S_0$ durch $sh \in S_1$ ersetzen.

Die Spezifikationen und ihre so definierten Morphismen bilden eine Kategorie, die wir mit **spec** bezeichnen.

Für das Folgende benötigen wir einige spezielle Arten von Morphismen, die wir nun definieren. Sei $f = (h, g): D_0 \to D_1$. Sind h injektiv (bijektiv) und g injektiv, so heißt f eine (Ω-)*Einbettung*. Mit $T_i := \emptyset T_{\Omega_i}$, $i = 0, 1$, bezeichnen wir die konstanten Ω_i-Terme (ohne Variable). Sei $T_1{}^0$ die Teilmenge der Ω_1-Terme, deren äußere Operation eine Wertsorte in S_0 hat. Zur Vereinfachung der Notation nehmen wir an, daß $f = (h, g)$ von nun an eine (Ω-)Einbettung ist, bei der h und g Inklusionen sind.

Dann heißt f **voll** genau dann, wenn für jeden Term $t \in T_1{}^0$ ein Term $t' \in T_0$ existiert, so daß $E_1 \vdash t = t'$ gilt. f wird **treu** genannt genau dann, wenn für alle Terme $t, t' \in T_0$ gilt: $E_1 \vdash t = t'$ impliziert $E_0 \vdash t = t'$. Eine *Erweiterung* (*Anreicherung*) ist eine volle und treue (Ω-)Einbettung.

Jede Einbettung $f: D_0 \to D_1$ liefert auf kanonische Weise einen D_0-Algebra-Morphismus $\tilde{f}$: **init** $D_0 \to$ **init** $D_1{}^0$, wobei letzteres das Redukt von **init** D_1 bzgl. $S_0 \subset S_1$ und $\Omega_0 \subset \Omega_1$ bezeichnet. Für die speziellen Morphismen läßt sich folgendes zeigen.

Satz 3: (1) f ist voll $\langle=\rangle$ $\hat{f}$ ist surjektiv, (2) f ist treu $\langle=\rangle$ $\tilde{f}$ ist injektiv. (3) f ist eine Erweiterung $\langle=\rangle$ $\tilde{f}$ ist bijektiv; ist darüber hinaus f eine Anreicherung, so haben **init** $D_1{}^0$ und **init** D_1 dieselbe Trägermenge.

Einige sehr nützliche Eigenschaften unserer Kategorie **spec** sind in den folgenden Sätzen zusammengefaßt:

Satz 4: **spec** hat Pushouts.

Sei (f_3, f_4) das Pushout von f_1 und f_2. Dann gilt:

Satz 5: Sind f_1, f_2 voll, so sind auch f_3, f_4 voll.

Satz 6: Sind f_1, f_2 treu, so sind auch f_3, f_4 treu.

Korollar 7: Sind f_1, f_2 Erweiterungen (Anreicherungen) so haben auch f_3, f_4 diese Eigenschaft.

4. Implementierungen

Wir definieren den Begriff der Implementierung als eine Beziehung zwischen Spezifikationen. Zur Unterstützung der Intuition geben wir zunächst ein Beispiel.

Beispiel 8: Im Hinblick auf die Beispiele 1 und 2 wird ALG durch NAT wie folgt implementiert:

(1) Wir fügen die Operation $+: \mathrm{nat} \times \mathrm{nat} \to \mathrm{nat}$ zur Spezifikation NAT hinzu sowie die Bestimmungsgleichungen $n + 0 = n$ und $n + \mathrm{succ}(m) = \mathrm{succ}(n + m)$. Die so erweiterte Spezifikation heiße NAT+. Dann besteht eine volle Ω-Einbettung NAT $\to$ NAT+.

(2) Wir definieren $t = (h, g)$: ALG $\to$ NAT+ durch h: alg $\to$ nat und $g: \emptyset \to 0$, $\sigma \to \mathrm{succ}$, $\cdot \to +$. Dann ist t eine treue Einbettung.

Definition 9: Eine *Implementierung* von D_0 durch D_1 ist ein Tripel $I = (D_2, f, t)$, wobei $f: D_1 \to D_2$ eine volle Ω-Einbettung ist und $t: D_0 \to D_2$ eine treue Einbettung. D_1 *implementiert* D_0 genau dann, wenn es eine Implementierung I von D_0 durch D_1 gibt.

Wenn D_0 durch D_1 implementiert wird, so gibt Satz 3 unmittelbar die Beziehung zwischen ihren initialen Algebren. Ein Vergleich zeigt, daß unser Ansatz den von ADJ [1] verallgemeinert. Eine andere Verallgemeinerung des ADJ-Ansatzes findet sich in [7].

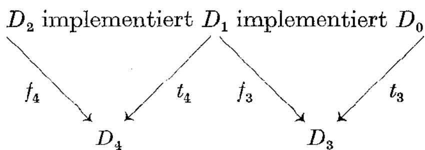
Aus Definition 9 können wir einige einfache Schlußfolgerungen ziehen. Um einige Beispiele anzugeben werde D_0 durch D_1 implementiert. Ist $f': D_3 \to D_1$ eine volle Ω-Einbettung, so wird D_0 auch durch D_3 implementiert. Ist t': $D_4 \to D_0$ eine treue Einbettung, so wird auch D_4 von D_1 (und D_3) implementiert. (Man zeigt leicht, daß die Komposition voller (bzw. treuer) Einbettungen wieder voll (bzw. treu) ist.)

Eine praktisch sehr wesentliche Frage ist es, ob sich Imple-

mentierungen zu „schrittweisem Vorgehen" zusammensetzen lassen. Der folgende Satz zeigt zunächst, daß „implementiert" eine transitive Relation ist.

Satz 10: Wird D_1 von D_2 sowie D_0 von D_1 implementiert, so wird auch D_0 von D_2 implementiert.

Der Beweis ist jedoch nicht konstruktiv, und es scheint auch keine allgemeine Kompositionsmethode für Implementierungen zu geben. Man betrachte das folgende Diagramm:



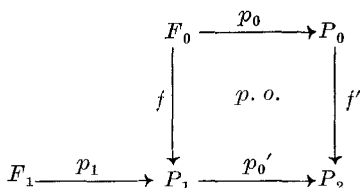
Sind f_3 und t_4 beides Erweiterungen, so liefert das Pushout von f_3 und t_4 nach Korollar 7 eine Gesamt-Implementierung von D_0 durch D_2. In [5] wird diskutiert, wie man auch in allgemeineren Fällen mit Hilfe von Pushouts Implementierungen konstruieren kann.

5. Parametrisierung

Da **spec** Pushouts hat, können Ersetzungsoperationen auf Spezifikationen (vgl. [6]) auf ähnliche Art und Weise behandelt werden wie Graph-Ersetzungen in der algebraischen Theorie der Graph-Grammatiken [12]. Einen wichtigen Spezialfall bilden parametrische Spezifikationen. Diese sind definiert als solche Spezifikationen P, welche eine Teilspezifikation F, den „formalen Teil", enthalten, die man durch verschiedene „aktuelle" Spezifikationen ersetzen möchte. So möchte man z. B. das Konzept eines Stack spezifizieren, ohne sich auf den Typ der Einträge festzulegen, um daraus dann Stacks zu gewinnen, deren Einträge z. B. ganze Zahlen, Zeichenketten, Arrays, etc. sind.

Definition 11: Eine *parametrische Spezifikation* ist eine Einbettung $p: F \to P$ in **spec**. F wird *formaler Teil* genannt.

Sei $p_0: F_0 \to P_0$ eine parametrische Spezifikation. Ein aktueller Parameter (der auch wieder parametrisch sein kann, z. B. Stacks von Arrays mit formalen Einträgen), etwa $p_1: F_1 \to P_1$, wird dem formalen Teil F_0 von p_0 durch einen Morphismus $f: F_0 \to P_1$ zugeordnet. Parameter-Einsetzung läßt sich dann durch das Pushout von p_0 und f präzisieren:



Dieses Pushout sei (f', p_0') und P_2 sei das Pushout-Objekt. Die resultierende (parametrische) Spezifikation ist dann $p_1p_0': F_1 \to P_2$. Der Fall $F_1 = \langle \emptyset, \emptyset, \emptyset \rangle$ gibt als Ergebnis dann i. w. Spezifikationen im Sinne der vorigen Abschnitte.

(Tatsächlich ist es nicht vernünftig, jeden beliebigen Morphismus $f: F_0 \to P_1$ als Parameter-Zuordnung zuzulassen. Einschränkungen werden in [5] diskutiert.)

Da aktuelle Parameter wieder parametrisch sein können, kann man den Vorgang der Parameter-Substitution iterieren. Der folgende Satz zeigt, daß das Resultat nicht von der Reihenfolge der Anwendungen abhängt.

Satz 12: Die Iteration der Parameter-Substitution ist assoziativ.

So kann man z. B. aus einem Stack- und einem Array-Konzept, jeweils mit formalen Einträgen, die Spezifikation eines Stacks mit Arrays als Einträgen, deren Einträge wiederum ganze Zahlen sind, eindeutig konstruieren.

Parametrische Spezifikationen lassen sich nach diesem Ansatz also als „Spezifikations-Konstruktoren" (und als Konstruktoren von Spezifikation-Konstruktoren) auffassen, wohingegen ADJ [2] die Auffassung als „Datentyp-Konstruktoren" vertritt. Ein Zusammenhang ergibt sich (für den Fall $F_1 = \langle \emptyset, \emptyset, \emptyset \rangle$), wenn man bei unserem Ansatz die initialen Algebren der aktuellen Parameter auf diejenigen der resultierenden Spezifikationen abbildet. Diese Fragestellung wird eingehender in [5] untersucht.

Literatur

[1] ADJ (GTWa): An initial algebra approach to the specification, correctness, and implementation of abstract data types. In: Current Trends in Programming Methodology IV: Data Structuring, ed. by R. Yeh, Prentice Hall, New Jersey 1978

[2] ADJ (TWaWr): Data type specification: parametrization and the power of specification techniques. Proc. SIGACT 10th Ann. Symp. on Theory of Computing, 1978

[3] *Burstall, R. M.; Goguen, J. A.:* Putting theories together to make specifications. Proc. 5th IJCAI – 77, MIT, Cambridge, Mass. 1977

[4] *Ehrich, H.-D.:* Extensions and implementations of abstract data type specifications. Proc. 7th Symp. MFCS '78, ed. by J. Winkowski. Lecture Notes in Computer Science 64, Springer-Verlag, Berlin (West) 1978, pp. 155–164

[5] *Ehrich, H.-D.:* On the theory of specification, implementation and parametrization of abstract data types. To be published

[6] *Ehrich, H.-D.; Lohberger, V. G.:* Parametric specification of abstract data types, parameter substitution, and graph replacements. To appear in the Proc. Workshop „Graphentheoretische Konzepte in der Informatik", Applied Computer Science, München–Wien: Carl Hanser Verlag 1978

[7] *Ehrig, H.; Kreowski, H.-J.; Padawitz, P.:* Some remarks concerning correct specification and implementation of abstract data types. Bericht Nr. 77–13, Techn. Universität Berlin (West), FB 20, Inst. für Software und Theoretische Informatik, 1977

[8] *Goguen, J. A.:* Correctness and equivalence of data types. In: Proc. 1975 Conf. on Algebraic Systems, Udine, Italy, Lecture Notes in Computer Science, Berlin (West): Springer-Verlag 1975

[9] *Goguen, J. A.:* Abstract errors for abstract data types. In: Proc. Conf. on Formal Description of Programming Languages. Amsterdam: ed. by E. J. Neuhold, North-Holland 1978

[10] *Guttag, J. V.:* The specification and application to programming of abstract data types. Tech. Report CSRG-59, Univ. of Toronto 1975

[11] *Liskov, B. H.; Zilles, S. N.:* Specification techniques for data abstractions. IEEE Transactions of Software Engineering Vol. SE-1 (1975), 7–19

[12] *Schneider, H.-J.; Ehrig, H.:* Grammars on partial graphs. Acta Informatica 6 (1976), 297–316

[13] *Wand, M.:* First-order identities as a defining language. Tech. Report No. 29, Computer Science Departement, Indiana University, Bloomington 1976

[14] *Zilles, S. N.:* Algebraic specification of data types. MIT Project MAC, Computation Structures Group Memo 119, Cambridge, Mass. 1975

Schlüsselwörter: Implementation — Parametrisierung abstrakter Datentypen

Ключевые слова: реализация — параметризация абстрактных типов данных

Key terms: implementation — parametrization — abstract data types

Mots-clés: impleméntation — paramétrisage de types de dates abstraits