Informatik Spektrum
Das aktuelle Schlagwort:
Datenbankprogrammiersprachen

Florian Matthes Joachim W. Schmidt

Universität Hamburg
Fachbereich Informatik
Arbeitsbereich DBIS
Vogt-Kölln Str. 30
2000 Hamburg 54
7. Februar 1992

Datenbankprogrammiersprachen entstehen durch die konzeptuelle Integration von Datenbankmodellen und algorithmisch vollständigen Programmiersprachen. Sie leisten damit einen Beitrag zur Realisierung adäquater Entwicklungsumgebungen für datenintensive Anwendungen.

Motiviert wird diese Integration einerseits durch die inzwischen allgemein klar erkannte Diskrepanz zwischen mengenorientierter, deklarativer Datenverarbeitung in Datenbanksystemen und dem elementorientierten, prozeduralen Paradigma konventioneller Programmiersprachen (*impedance mismatch*). Darüber hinaus gibt es jedoch noch weitere Unverträglichkeiten wesentlicher Konzepte in Datenbankmodellen und Sprachen zur Anwendungsprogrammierung; Beispiele hierfür sind Benennbarkeits- und Strukturierungsmechanismen sowie Bindungszeitpunkte und Methoden zur Lebensdauerkontrolle (*competence mismatch*). Diese Inkompatibilitäten führen nicht nur zu oft beklagten Performanzverlusten an der Schnittstelle zwischen Wirtsprache und Datenbanksystem, sondern erzwingen vor allem redundante Deklarationen und aufwendige repetitive Konvertierungsoperationen auf niedrigem Abstraktionsniveau, die hohe Folgekosten im gesamten Lebenszyklus der Applikationsprogramme nach sich ziehen.

Zur Überwindung dieser konzeptuellen Schwierigkeiten bieten Datenbankprogrammiersprachen (DBPLs) linguistische Unterstützung auf drei Ebenen:

Persistenzabstraktion durch die uniforme sprachliche Behandlung flüchtiger sowie langlebiger Datenobjekte;

Typvollständigkeit unter besonderer Berücksichtigung von Massendaten (*bulk data types*);

Iterationsabstraktion z.B. durch quantifizierte Prädikate, mengenwertige Ausdrücke, Funktoren oder unifikationsbasierte Datendeduktion.

Neben diesen konzeptuellen Anforderungen, die auf adäquatem sprachlichen Niveau zu erfüllen sind (*Orthogonalität*, *Generik*, *Formalität*) werden an Datenbanksprachen auch hohe *operationale* Anforderungen gestellt, wie z.B. optimierender Zugriff auf Sekundärspeicher, Unterstützung der Transaktionsprogrammierung sowei Verteilung und Kommunikation. Zunehmende Bedeutung gewinnen darüber hinaus solche Architekturen von DBPL Implementationen, die eine Eingliederung in bereits existierende Systemarchitekturen (Netzwerkdienste, Transaktionsmonitore, Benutzerschnitsstellen, Objektspeicher) unterstützen (*Offenheit*, *Interoperabilität*).

Ausgehend von ersten Arbeiten Ende der 70er Jahre [Schmidt 77] wird die rasche Entwicklung dieses Forschungsgebiets durch zahlreiche Veröffentlichungen, Workshops [Bancilhon, Buneman 87; Hull et al. 89; Kanellakis, Schmidt 91; Rosenberg, Koch 89; Dearle et al. 90] und nicht zuletzt durch evaluierbare Systeme und Produkte dokumentiert. An dieser Stelle wird lediglich eine grobe Klassifikation in drei Hauptentwicklungslinien von Datenbankprogrammiersprachen vorgenommen sowie eine Einführung in Konzepte und einige realisierte Sprachen gegeben. Die angegebene Literatur, insbesondere [Atkinson, Bunemann 87; Schmidt, Matthes 90], ermöglichen eine tiefergehende Auseinandersetzung mit dem Thema.

Typorientierte Datenbankprogrammiersprachen

Eine wesentliche Sprachfamilie läßt sich durch ihre Ausrichtung an wohlverstandenen Konzepten prozeduraler Programmiersprachen charakterisieren (statische, blockstrukturierte Sichtbarkeitsregeln, statische, strikte Typüberprüfung, Typkonstruktoren, Parametrisierung, Modularisierung)

Mitglieder dieser Klasse sind die *Relationalen Datenbankprogrammiersprachen* (z.B., Pascal/R, Plain, Rigel, Modula/R, Modulex, DBPL) mit Pascal-ähnlichen Spra-

chen als algorithmischen Kern, der um Relationstypen und mengenorientierte Operationen auf typisierten Relationenvariablen erweitert ist. Darüber hinaus existieren Mechanismen zur Deklaration von persistenten Daten und zur Definition von Transaktionen. Das folgende Beispiel in der Datenbankprogrammiersprache DBPL illustriert die Vorteile eines solchen Ansatzes:

```
from EmpDB import EmployeeRel;
transaction HireYoung(var E: EmployeeRelType);
begin
   if all e in E (e.age <25) then
      EmployeeRel:+ E
   else
      PrintEmployees({each e in E: e.age>=25});
   end
end HireYoung;
```

Für Programmierer sind die zugrundeliegenden Sprachkonzepte wie Typisierung, Modularisierung und Parametrisierung bereits bekannt, und die (meist kalkülorientierten) Anfragesprachen sind leicht zu erlernen. Gleichzeitig gestatten die relational vollständigen mengenorientierten Anfragesprachen, die eng mit den übrigen Ausdrücken der Sprache verwoben sind, eine effiziente, dynamisch optimierende Implementation.

Durch konsequente Anwendung des aus Programmiersprachen bekannten Orthogonalitätsprinzips bieten moderne Vertreter dieser Sprachfamilie *integrierte* Modellierungs- und Manipulationsmechanismen, wie sie beispielsweise in den historisch unabhängig voneinander entstandenen Datenmodellen für komplexstrukturierte Objekte und deduktive Datenbanken getrennt vorliegen. So werden etwa Strukturen wie die des NF^2-Modells durch das Prinzip der freien Schachtelung von Record-, Varianten-, Array- und Relationenkonstruktoren definierbar. Durch Lambda-Abstraktion und Parametrisierung von Anfrageausdrücken erreicht man die Ausdrucksmächtigkeit von Anfragesprachen mit stratifizierter Fixpunktsemantik. Entkoppelt man schließlich noch die Lebensdauer eines Datenobjektes von seinem Typ, so können auch nicht-relationale Strukturen (z.B. eine boolesche Variable oder eine Matrix) persistent gehalten werden.

Die Klasse der ebenfalls stark typorientierten *persistenten Programmiersprachen* (PS algol, Napier88, Amber, P-Quest) konzentrierte sich zuerst auf den Aspekt der Langlebigkeit beliebigstrukturierter Datenobjekte. Ausgehend von dem Modell eines *persistent heap* können in diesen Sprachen alle semantisch relevanten Objekte (Records, Funktionen, abstrakte Datentypen, Module, Zeiger, ...) dynamisch in einem globalen, programmübergreifenden Adressraum alloziert und manipuliert werden. Alle Objekte, die transitiv von einem ausgezeichneten Wurzelobjekt (*persistent root*) durch statische Sichtbarkeitsreglen oder dynamische Bindungen erreichbar sind, werden langlebig gespeichert und stehen anderen Programmen (durch Naviation ausgehend von dem Wurzelobjekt) zur Verfügung. Das Konzept der orthogonalen Persistenz erfordert einen generalisierten Bindungsbegriff, der es Anwendungsprogrammen gestattet, dynamisch Bindungen an bereits existierende Datenstrukturen auf der persistenten Halde zu erzeugen und bestehende Bindungen wieder zu lösen. Eine große konzeptuelle und technologische Herausforderung stellt in diesem Zusammenhang die Balance zwischen statischer Typsicherheit und dynamischer Bindungsflexibilität dar, ein Problem, das sich ebenfalls in modernen Systemprogrammiersprachen (Modula-3) und Sprachen für verteilte Anwendungen (Hermes) stellt.

Da auch Module und Anwendungsprogramme Bestandteil der persistenten Halde sein können, ist es möglich, Aufgaben von spezialisierten Werkzeuge, wie Schema- oder Formular-Editoren, Linker, Bibliotheks- oder Versionsmanager (*make*) durch die Mechanismen der Datenbankprogrammiersprache zu lösen.

Noch weitgehend unerforscht ist die Mächtigkeit der *reflektiven* Aspekte dieser Sprachfamilie. In der einfachsten Form bieten diese Systeme die Möglichkeit eines *callable compiler*, der Quelltexte (z.B. ad hoc Anfragen) in Funktionen übersetzt, die dynamisch zu dem in Ausführung befindlichen Programm gebunden werden. Dieser Mechanismus (*run time reflection*) wird z.B. erfolgreich zur Übersetzung von Kalkülanfragen in geschachtelte Schleifen eingesetzt und prototypisch für die typstrukturgesteuerte Generierung von Applikationsprogrammen benutzt. Eine andere Form der Reflektion (*compile time reflection*) erlaubt die Erweiterung der Syntaxanalysephase durch benutzerdefinierten Code, der z.B. generische Spracherweiterungen durch Abbildung in geschachtelte Ausdrücke einer reduzierten Kernsprache implementiert.

Eine wesentliche Einschränkung persistenter Programmiersprachen für konventionelle Datenbankanwendungen ist jedoch bisher das Fehlen einfach zu handhabender generischer Datenstrukturen für Massendaten und sprachlicher Mechanismen zur Iterationsabstraktion.

Logikbasierte und funktionale Datenbanksprachen

Die Strukturen und Iterationsabstraktionen des relationalen Datenmodells (Mengen und Prädikate) bilden den Ausgangspunkt logikbasierter und funktionaler Datenbanksprachen. Sprachen deduktiver Datenbanken (Wissensbanken, Expertensysteme) unifizieren extensionale und intensionale Datenbankaspekte (Fakten und Regeln) und streben im Gegensatz zu den imperativen typorientierten Sprachen eine rein deklarative Sprachsemantik an. In logikbasierten Sprachen kann sich der Anwender auf die Spezifikation der Eigenschaften der zu manipulierenden Strukturen beschränken und dem System die Auswahl einer optimierten Evaluationsreihenfolge überlassen.

Ausgehend von einfachen Horn-Klauseln wurden die Datenbanksprachen der *Datalog*-Familie schrittweise um Basisprädikate (Gleichheit, Ordnungsrelationen), Negation, Aggregatfunktionen und Funktionssymbole zur Repräsentation komplexer Objekte erweitert. Neben NAIL!, POSTGRES, ALGRES, PRISMA, educe und Megalog ist insbesondere die Sprache LDL zu erwähnen, die auch Operatoren für "deklarative" Änderungsoperationen auf Datenbanken umfaßt.

Die genannten Systeme bieten ebenfalls Persistenzabstraktion, indem alle erzeugten Fakten und Regeln persistent gespeichert werden. Der Preis für die potentiell exzellente Optimierbarkeit dieser Sprachen ist allerdings der Verlust der algorithmischen Vollständigkeit. Praktisch alle Sprachen bieten daher Schnittstellen zu imperativen Wirtsprachen für anspruchsvollere Datenmanipulationen (statistische Analysen, Dateneinund ausgabe) an, so daß der *impedance* und *competence mismatch* eventuell auf einer höheren Ebene wieder auftritt.

Die Notation der *list comprehensions* in Programmiersprachen wie Miranda oder Haskell hat wegen ihrer Nähe zum relationalen Kalkül das Interesse an der Verwendung funktionaler Sprachen für Datenbankanfragen wiederbelebt. Ein *join* in der Syntax einer *set comprehension*

$$[(name\ e,\ salary\ e,\ name\ o) \mid e \leftarrow employees;\ s \leftarrow orders;\ supplier\ o = name\ e]$$

ist wesentlich leichter lesbar als seine Formulierung mittels Funktoren in traditionellen funktionalen Datenbanksprachen wie FQL. Andererseits gibt es triviale Übersetzungsalgorithmen von *comprehensions* in Ausdrücke des reinen Lambda-Kalküls, auf denen Transformationen analog zu den bekannten algebraische Optimierungen des Relationalen Kalküls durchführbar sind. Zwar erlaubt die referentielle Transparenz rein funktionaler Sprachen ähnlich agressive Optimierungen wie in logikbasierten Formalismen, jedoch gibt es substantielle Unterschiede in der Semantik rekursiver Deklarationen. Die Abbildung relationaler Strukturen und Iterationsabstraktionen in den Lambda-Kalkül gestattet auch die Untersuchung verallgemeinerter Datenstrukturen (*quads*) für Massendaten (Listen, Bags, Multimengen, Bäume), die alle mit *comprehensions* als einer uniformen Anfragenotation ausgestattet sind. Der skizzierte Transformationsansatz wird ebenfalls für die prototypische Implementation funktionaler Datenbanksprachen eingesetzt, die zudem voll in das Typsystem und das Ausführungsparadigma moderner Programmiersprache (à la ML) integriert sind.

In der Sprache Machiavelli werden ebenfalls SQL-ähnliche Ausdrücke als "syntaktischer Zucker" in Applikationen einer einzelnen vordefinierter Funktionen höherer Ordnung (*hom*, homomorphe Extension) transformiert. Machiavelli löst darüber hinaus das Problem der Typisierung des *natural join* Operators ($Students \bowtie Employees$), indem die Sprache polymorphe Typregeln für konsistenzerhaltende Joinoperationen auf Records ($Student \bowtie Employee$) definiert, die gleichzeitig die Typisierung von allgemeinenen Vererbungsbeziehungen erlaubt, wie sie in objekt-orientierten Sprachen auftreten.

Ein weiterer wichtiger Beitrag funktionaler Datenbanksprachen liegt im Einsatz von *Typinferenzmechanismen*, die nicht nur dem Programmierer die Angabe expliziter Typinformationen z.B. bei der Formulierung von Anfragen oder bei Sichtendefinitionen ersparen, sondern auch Anfragen und Transaktion den "möglichst generischen" Typ (*principal type scheme*) zuordnen. Damit kann z.B. die Anwendung der folgenden Anfrage auf alle Relationen E zugelassen werden, sofern sie nur ein numerisches *status* Attribut besitzen.

select * **from** E **where** $status > 30$

Abschließend seien noch die Sprachen Life und COL erwähnt, die Konzepte der funktionalen Programmierung und Vererbungsbeziehungen durch polymorph typisierte Termkonstruktoren (Ψ-Terme in Life) in einen logikbasierten Rahmen integrieren.

Objekt-orientierte Systeme

Im Gegensatz zu de eher durch konzeptuelle und formale Vorteile motivierten funktionalen Datenbanksprachen wird der Entwurf objekt-orientierter Systeme sehr stark durch technologische Überlegungen geleitet, da viele der Sprachen im Zuge der Implementation prototypischer oder kommerzieller Datenbanksysteme entstanden sind (Adaplex für MULTIBASE, OPAL für GemStone, CO_2 für O_2, O++ für ODE). Obwohl sich z.B. Adaplex an einem funktionalen Datenmodell orientiert, bietet es bereits Objektidentität, Klassifikation und Mehrfachererbung sowie die für objekt-orientiere Anfragesprachen typische Iterationsabstraktion durch eine Mischung aus quantifizierten Mengenausdrücken und navigierendem Datenzugriff über Objektreferenzen (Punktnotation):

select *e.name, e.salary, o.name*
from *e* **in** *Employees, o* **in** *e.orders*

Objekt-orientierte Systeme bieten darüber hinaus Mechanismen zur Definition sogenannter komplexer Objekte und Methoden sowie Mechanismen zur späten Bindung und Methodenredefinition in Subklassen. Neben harten technischen Problemen der Anfrageoptimierung (z.B. dynamisch gebundene Methodenaufrufe), der Objektadressierung und Speichererverwaltung (*clustering*, *garbage collection*) stellen sich in diesen Systemen auch neuartige sprachliche Probleme. Biepsiele sind etwa die Semantik dynamischer Restrukturierungen der Klassenhierarchie oder die Frage nach Sichtbarkeitsregeln zwischen Super- und Subklassen, die eine effektive gemeinsame Nutzung von Methoden, Attributen und Integritätsbedingungen gestatten, ohne dabei ungewollte Namenskonflikte in Mehrbenutzersystemen zuzulassen.

Eine besondere Stärke objekt-orientierter Systeme ist die enge konzeptuelle und technologische Verflechtung zwischen grafischen Benutzerschnittstellen, modularer und erweiterbarer Datenabstraktion sowie effizienten Objektspeichern, wie sie z.B. durch die Datenbankbrowser und Debugger von O_2 und GemStone demonstriert wird (*objects on screen, in memory and on disk*).

Neuere Typsysteme für Datenbankapplikationssysteme

Zusammenfassend läßt sich heute eine erfreuliche Konvergenz der oben skizzierten drei Entwicklungslinien feststellen. Alle neueren Systeme und Sprachen betonen die *orthogonale* Kombinierbarkeit *elementarer* Basiskonzepte für Persistenzabstraktion, typvollständige Datenstrukturierung und Iterationsabstraktion.

Gegenstand aktueller Forschungsarbeiten (z.B. in den Esprit Projekten DAIDA, FIDE und dem DFG Programm *Objektbanken für Experten*) ist die formale Integration von Methoden, Sprachen und Systemen zur Entwicklung integrierter, effizienter Datenbankapplikationssysteme. Hohe Priorität besitzt dabei das Aufbrechen monolithischer Systemstrukturen in wohldefinierte funktionale Teilkomponenten wie z.B.

- Modellunabhängige Objektspeichersysteme, die hocheffizienten Zugriff auf einfach strukturierte Daten in logisch oder physisch partitionierten Bereichen (*repositories*) gestatten;
- Bibliotheken generischer, frei kombinierbarer (evtl. modellspezifischer) Datenstrukturen, wie *bang files* für logikbasierte Modelle, Bäume und Hashtabellen für relationale Modelle, Pfad-Indices für objektorientierte Modelle;
- Sammlungen parametrisierter Datenkonstruktoren zur implementationsunabhängigen Beschreibung von Massendaten (*bulk data types*) und der auf ihnen definierten mengenorientierten Operationen;
- Kanonische Zwischensprachen zur architekturunabhängigen Repräsentation, Analyse, Transformation und Übersetzung von Programmfragmenten (Anfragen) oder vollständigen Programmen.

So wird z.B. im FIDE Projekt [Matthes, Schmidt 91] speziell der Einsatz moderner Typkonzepte (parametrischer und Subtyp-Polymorphismus, Metatypisierung, *dependent types*) und Methoden algebraischer Spezifikationen zur Definition und kontrollierten Implementation dieser generischen Datenbankdienste in einem integrierten sprachlichen Rahmen untersucht.

Literatur

Atkinson, Bunemann 87: Atkinson, M.P. and Bunemann, P. „Types and Persistence in Database Programming Languages“. *ACM Computing Surveys*, 19(2), June 1987.

Bancilhon, Buneman 87: Bancilhon, F. and Buneman, P., Hrsg. *Proceedings of the First Workshop on Database Programming Languages*, Roscoff, Finistere, France, 1987. Altaïr, BP 105, Rocquencourt, 78153 Le Chesnay Cedex, France.

Dearle et al. 90: Dearle, A., Shaw, G., and Zdonik, S., Hrsg. *Implementing Persistent Object Bases: The Fourth International Workshop on Persistent Object Systems.* Morgan Kaufmann Publishers, 1990.

Hull et al. 89: Hull, R., Morrison, R., and Stemple, D., Hrsg. *Proceedings of the Second Workshop on Database Programming Languages*, Salishan, Oregon, 1989. Morgan Kaufmann Publishers.

Kanellakis, Schmidt 91: Kanellakis, P. and Schmidt, J.W., Hrsg. *Database Programming Languages: Bulk Types and Persistent Data*, Nafplion, Greece, 1991. Morgan Kaufmann Publishers.

Matthes, Schmidt 91: Matthes, F. and Schmidt, J.W. „Towards Database Application Systems: Types, Kinds and Other Open Invitations“. In: *Proceedings of the Kiev East/West Workshop on Next Generation Database Technology*, Band 504, *Lecture Notes in Computer Science*, April 1991.

Rosenberg, Koch 89: Rosenberg, J. and Koch, D., Hrsg. *Third International Workshop on Persistent Object Systems, Newcastle, Australia 1989.* Workshops in computing. Springer-Verlag, 1989.

Schmidt, Matthes 90: Schmidt, J.W. and Matthes, F. „Language Technology for Post-Relational Data Systems“. In: Blaser, A., Hrsg., *Database Systems of the 90s*, Band 466, *Lecture Notes in Computer Science*, Seite 81–114, November 1990.

Schmidt 77: Schmidt, J.W. „Some High Level Language Constructs for Data of Type Relation“. In: *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Toronto, Canada*, August 1977.