Algebraic Data Types for Language-Integrated Queries

George Giorgidze Torsten Grust Alexander Ulrich Jeroen Weijers

Department of Computer Science
Universität Tübingen
Tübingen, Germany
{george.giorgidze,torsten.grust,alexander.ulrich,jeroen.weijers}@uni-tuebingen.de

Abstract

The seamless integration of relational databases and programming languages remains a major challenge. Mapping rich data types featured in general-purpose programming languages to the relational data model is one aspect of this challenge. We present a novel technique for mapping arbitrary (nonrecursive) algebraic data types to a relational data model, based on previous work on the relational representation of nested tables. Algebraic data types may be freely constructed and deconstructed in queries and may show up in the result type. The number of relational queries generated is small and statically determined by the type of the query. We have implemented the mapping in the Database Supported Haskell (DSH) library.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Data types and structures; H.2.3 [*Languages*]: Query languages

General Terms Languages, design, theory

Keywords Programming language, database, algebraic data type

1. Introduction

Language-integrated queries allow relational database management systems (RDBMSs) to participate in the execution of program fragments written in high-level, general-purpose host programming languages. This approach to program execution is particularly beneficial for offloading data-intensive and data-parallel computations from the programming languages' runtime system to an RDBMS. The popular and widely used LINQ extension of the .NET family of languages [14], as well as the Links programming language [6] and the Database Supported Haskell (DSH) library [8] are examples of language-integrated query facilities.

These and other language-integrated query facilities differ in what types of computations they consider executable by the database backend. It is clear that when a larger set of the host language's constructs are faithfully and efficiently supported in a query context, more applications can benefit from the backend's processing capabilities. Some facilities, such as the Links language, only consider computations as database-executable if they have a flat bag type. LINQ supports a richer set of types, including more collection types, but many order-sensitive operations (e.g., indexing, zipping and splitting) are not considered database executable.

In this context, the Ferry framework [11] and its query compilation strategy deserve a particular mention. Ferry supports arbitrarily nested lists of tuples (or records) of basic types. Order-sensitive operations are faithfully implemented on the relational backend. Moreover, the number of relational queries generated by Ferry only depends on the static type of the compiled program's result. Ferry's compilation strategy, called *loop lifting*, has been used to develop language-integrated query facilities for the Haskell language in the form of DSH [8] and a number of other languages [10, 12, 16].

So far, work in the context of Ferry focused on supporting a rich set of operations for the supported types, especially for nested and ordered collections [12]. We now shift the focus towards extending the *set of types* which are supported in queries from the programmer's point of view. Algebraic data types (ADTs) are the essential data modelling tool of a number of functional programming languages like Haskell, OCaml and F#. We claim that faithful support for ADTs and operations on them in a query context considerably increases the expressiveness of language-integrated queries for these languages.

The key insight of this paper is that a facility that already supports arbitrarily nested lists of tuples of basic types can be extended to support (nonrecursive) ADTs without the need to modify its relational backend. This can be achieved by mapping ADTs to the already supported types of arbitrarily nested lists of tuples and realising operations on values of ADTs with already supported operations on lists and tuples. We demonstrate this by extending the DSH library. Specifically, DSH now considers construction and elimination of values of user-defined ADTs as database-executable computations. In addition, our approach benefits from earlier work on the generation of efficient relational code for the already supported types and operations [12].

A version of the Links language implemented by the third author [16], considered construction and elimination of values of ADTs as database-executable. This was achieved by developing a significantly extended variant of the loop lifting compilation strategy; that is, the functionality has been implemented in the backend of the compiler. In this paper we describe a simpler approach. By realising construction and elimination of ADTs in terms of the already supported operations, we reuse the original loop lifting compilation strategy without extending it; that is, we implement the functionality in DSH's frontend. As such, the present approach should carry over to similar language-integrated query facilities.

So far, significant amount of work has gone into realising object-relational mappings (ORMs) for object-oriented languages [5, 7], while relational mappings for ADTs, which are prevalent in functional languages, received relatively little attention. With this paper we aim to tackle this problem. In the following, we list our contributions.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DDFP'13, January 22, 2013, Rome, Italy.
Copyright © 2013 ACM 978-1-4503-1871-6/13/01. . . $15.00

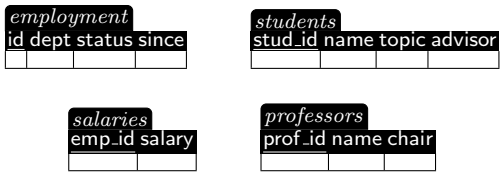

Figure 1. Table schemata for the university employees example.

- This paper presents a novel technique for mapping (nonrecursive) ADTs and operations on values of such types to the relational model and relational queries, respectively.
- The mapping is implemented in the DSH library. As a result, DSH now considers the construction and elimination of ADT values as database-executable computations. To the best of our knowledge, DSH is the first language-integrated query facility that supports arbitrarily nested lists, tuples and ADTs.
- The described DSH extension is implemented in its frontend without extending its relational backend. We think that this is a simpler approach that can be easily ported to other systems.
- The number of relational queries generated by DSH only depends on the static type of the compiled program's result. We describe how programmers can compute this number for program fragments whose type involves ADTs.
- The presented relational realisation of ADTs and operations on such types can be used in other integrations of relational databases in programming languages that support ADTs (e.g., OCaml, SML, and F#).

2. Queries and Algebraic Data Types

In this section, we give two examples to motivate the support for ADTs in DSH queries. Both examples consider university departments and use the table schemata shown in Figure 1.

As a first example, we fetch the information for all employees, sorted by seniority. Consider that there are multiple types of employees (e.g., *professors* and *PhD students*), each characterised by different properties. Naturally, in Haskell this would be modelled in terms of an ADT:

$$\begin{aligned}\textbf{data}\ \mathit{Employee} = \mathit{Prof}\ \{&\mathit{name} :: \mathit{String}, \mathit{chair} :: \mathit{String},\\ &\mathit{advisedStudents} :: [\mathit{String}]\}\\ \mid\ \mathit{Stud}\ \{&\mathit{name} :: \mathit{String}, \mathit{topic} :: \mathit{String},\\ &\mathit{advisor} :: \mathit{String}\}\end{aligned}$$

Instead of piecing together this heterogeneous result from multiple DSH queries, with support for ADTs we can write a single query which computes the desired answer:

```
employeesBySeniority :: Q [Employee]
employeesBySeniority = concat
  [ if eStatus ≡ "student"
    then [ stud sName sTopic sAdvisor
         | (sID, sName
           , sTopic, sAdvisor) ← table "students"
         , eID ≡ sID]
    else [ prof pName pChair sAdvised
         | (pID, pName, pChair) ← table "professors"
         , eID ≡ pID
         , let sAdvised =
             [ sName
             | (_, sName
               , _, sAdvisor) ← table "students"
             , sAdvisor ≡ pName]]
  | (eID, _, eStatus, _) ← sortWith (λ(_, _, _, d) → d)
                                    (table "employment")]
```

The query is formulated using a mixture of comprehensions [9] and combinators, thus resembling a vanilla Haskell list-processing program, with two notable differences: (1) the *table* function is used to reference database-resident tables and (2) the query's result type is wrapped in the Q type constructor. In DSH, the type $Q\ a$ describes a query that returns a value of type a. Lastly, the expressions *prof* and *stud* are not regular Haskell data constructors but *smart constructors* which are automatically generated by DSH for the type *Employee* (see Section 4).

At first sight, we could simply employ two queries to fetch students and professors separately and then use regular in-heap Haskell processing to append the two resulting lists. Observe however, that the outer comprehension orders the employees, a job that DSH dispatches to the database backend. Otherwise, to construct the result in the proper order from two separate queries, we would need to sort and/or merge the two intermediate lists in-memory. Not only does this violate the "*compute close to the data*" principle but it also incurs extra Haskell logic that is unnecessary in this case.

It is even worse to fetch general employment data first (the table `"employment"`) and then issue a separate query for each employee to read her information (leading to the well-known *n+1-queries problem* which is prevalent in language-integrated query facilities [12]). Instead, DSH issues exactly four relational queries to generate the result (see Section 3). The increased query expressiveness through ADTs allows us to specify a single DSH query which delivers the desired result and avoids a potentially costly post-processing step in main memory.

While the first example only *constructed* ADT values, the ability to *eliminate* such values (i.e., to perform case analysis) in queries is as crucial. As a second example, we are interested in minimum salaries (per department). We assume that there may be unpaid employees. Still, departments with unpaid members only (their associated list of salaries will be empty) shall be included in the result.

The Haskell list combinator $\mathit{minimum} :: \mathit{Ord}\ a \Rightarrow [a] \to a$ is a *partial function* that throws an exception when applied to the empty list. Exceptions are certainly not desirable, especially when sending code to remote execution facilities such as an RDBMS. With support for ADTs, we are able to provide a DSH function called $\mathit{safeMinimum}$. This function wraps its result in the *Maybe* type constructor:

$$\mathit{safeMinimum} :: (\mathit{Ord}\ a, \mathit{QA}\ a) \Rightarrow Q\ [a] \to Q\ (\mathit{Maybe}\ a)$$

The result is then $\mathit{Just}\ \mathit{min}$ for a nonempty list and $\mathit{Nothing}$ otherwise, making the error case explicit. The annotation $\mathit{QA}\ a$ restricts the type variable a to *queryable* types [8]. With this combinator, we are able to implement the desired query in a safe way:

```
salPerDept :: Q [(String, [Integer])]
salPerDept =
  [ (dept, [ salary
           | (sID, salary) ← table "salaries"
           , (dID, _, _, _) ← deptMembers
           , sID ≡ dID])
  | (dept, deptMembers) ← groupBy (λ(_, d, _, _) → d)
                                  (table "employment")]
minSalPerDept :: Q [(String, Integer)]
minSalPerDept =
  [(dept, case_ (safeMinimum sals)
                0
                (λmin → min))
  | (dept, sals) ← salPerDept]
```

The elimination function $case\_$ performs case distinction based on its first argument (a value of an ADT sum type)[1]. All remaining arguments handle the different cases. The second argument deals with the nullary $Nothing$ constructor by returning the salary 0 as a default value. Otherwise, $case\_$ applies a lambda expression (here: $\lambda min \rightarrow min$) to values tagged by $Just$, i.e., the actual minima of the non-empty lists. Note that—although not necessary in this example—the body of the lambda expression for the $Just$ case could be an arbitrary (database-executable) expression (see Section 4).

3. Representing Algebraic Data Types

Mapping nested data—non-first normal form (NF^2) data in which collections and tuples nest arbitrarily—onto the flat tables implemented by RDBMSs has been extensively studied by Van den Bussche [17] and in the context of Ferry [12]. Here, we present a technique to map arbitrary nonrecursive ADTs onto this nested data model. We provide an implementation of this technique in DSH [1, 8] which, in turn, is based on the Ferry work.

Consider the data type $Either$ with two constructors:

$$\textbf{data}\ Either\ a\ b = Left\ a \mid Right\ b$$

We map type $Either\ \ a\ \ b$ to its *representation type* $([a], [b])$, where the pair components represent the $Left$ and $Right$ constructors, respectively. A value of type $Either\ a\ b$, then, is represented as a pair of a singleton and an empty list: $Left\ e$ maps onto the pair $([e], [])$ while $Right\ e$ maps onto $([], [e])$.

In the following, we will consider algebraic data types as *sum-of-product types* as is common in the literature on data type generic programming [2]: an ADT with $n \geqslant 1$ constructors is an n-ary sum ($\oplus$) whose summands are product types ($\otimes$). The representation scheme for the binary $Either$ type sketched above generalises to general sum types with n constructors: an n-ary sum type is mapped to a representation type consisting of an n-tuple in which each of the tuple components correspond to one summand. Function rep derives the representation type—exclusively formed by list and tuple constructors—for this general case:

$$\begin{aligned}
rep\ (\tau_1\ \oplus\ ...\ \oplus\ \tau_n) &= ([rep\ \tau_1], ..., [rep\ \tau_n]) \\
rep\ (\tau_1\ \otimes\ ...\ \otimes\ \tau_n) &= (rep\ \tau_1, ..., rep\ \tau_n) \\
rep\ [\tau] &= [rep\ \tau] \\
rep\ \tau_{prim} &= \tau_{prim}
\end{aligned}$$

Applying this function to $Either$ (which has the generic type $a\ \oplus\ b$) indeed results in the representation type $([a], [b])$. Similarly, the sum-of-products representation of the $Employee$ data type from Section 2 is:

$$\begin{aligned}
&(String\ \otimes\ String\ \otimes\ [String]) \\
&\ \oplus\ (String\ \otimes\ String\ \otimes\ String)
\end{aligned}$$

Using function rep, the representation type for an $Employee$ value takes the form of a tuple-of-lists type:[2]

$$\begin{aligned}
&([(String, String, [String])] \\
&, [(String, String, String)])
\end{aligned}$$

In our earlier work on Ferry we have studied how values of such nested types efficiently map onto the flat relational data model [8, 12]. We do not repeat this here but illustrate with Figure 2, how a Haskell value (of type $[Employee]$), its tuple-of-lists equivalent, and its relational representation relate. A couple of points are worth noting:

- Columns named ref contain foreign keys: t_2.ref and t_3.ref refer to t_1 to supply the properties of professors and students, respectively. t_4.ref refers to t_2 and collects all students advised by a professor.
- Columns named pos encode list positions (e.g., `"A"` occurs at position 1 in its containing list, see t_4.pos).
- Looking at the key values p_i and s_i in t_1, either p_i or s_i is never referred to in the database (for any i). This implements the semantics of a (binary) ADT sum whose values are either of one or the other type. In Ferry, the absence of references is used to encode $[]$ which coincides with our tuple-of-lists encoding of sum types (see beginning of this section).

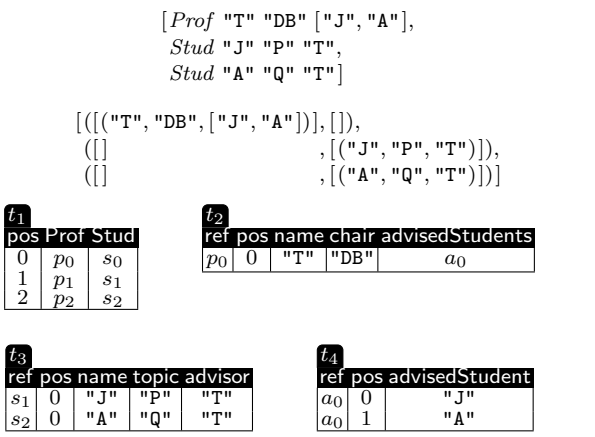

Figure 2. A value of type $[Employee]$ in three representations: ADT (top), generic tuple-of-lists (center), flat relational (bottom).

It is important to observe that knowing the *type* $Q\ \tau$ of a DSH expression suffices to predict the number of database tables $|\tau|$ that are required to hold its values: in a nutshell, this number depends on the number of list type constructors in $rep\ \tau$. With a look at rep, it is straightforward to define function $|\tau|$:

$$\begin{aligned}
|[\tau]| &= 1 + tab\ \tau \\
|\tau| &= 1 + tab\ \tau \\
tab\ (\tau_1\ \oplus\ ...\ \oplus\ \tau_n) &= n + tab\ \tau_1 + ... + tab\ \tau_n \\
tab\ (\tau_1\ \otimes\ ...\ \otimes\ \tau_n) &= tab\ \tau_1 + ... + tab\ \tau_n \\
tab\ [\tau] &= 1 + tab\ \tau \\
tab\ \tau_{prim} &= 0
\end{aligned}$$

As already shown in Figure 2, $|\cdot|$ indicates that four tables are required to encode a value of type $[Employee]$. In [12], we discuss how DSH will issue a bundle of exactly four database queries—each query yielding one of the t_i—to evaluate a DSH expression of type $Q\ [Employee]$. In other words, DSH does not suffer from the n+1-queries problem.

4. Construction and Elimination

In this section we describe how ADT values can be constructed and eliminated in DSH queries. Let us start again by considering the construction and elimination for the $Either$ data type. For this data type, DSH provides the following two (automatically generated) smart constructors:

$$\begin{aligned}
left\ &:: (QA\ a, QA\ b) \Rightarrow Q\ a \rightarrow Q\ (Either\ a\ b) \\
right\ &:: (QA\ a, QA\ b) \Rightarrow Q\ b \rightarrow Q\ (Either\ a\ b)
\end{aligned}$$

[1] Elimination via $case\_$ is only used for types with multiple constructors. For tuple matching as used in both examples, we use *View Patterns* [1] instead. See the DSH source code for the fully worked out example.

[2] Remember that these are internal types. Programmers continue to think in terms of their self-documenting Haskell types.

By exploiting the representation of ADTs described in Section 3, these two smart constructors are implemented in terms of the functions $\lambda e \rightarrow ([e], [])$ and $\lambda e \rightarrow ([], [e])$, respectively. Again, this internal representation of queries of type $Q\ (Either\ a\ b)$ is not exposed to the library users [1]. As we will see later in this section, this is important to provide the guarantee that the tuple-of-lists representation of sum ADTs contain exactly one singleton list.

The previous example has only dealt with the smart constructors for the $Either$ data type. In general, smart constructors for the ADT with the sums-of-products representation $\tau_1 \oplus \tau_2 \oplus \ ... \ \oplus \tau_n$ (with $n \geqslant 1$) can be conceptually defined as follows:

$$
\begin{aligned}
&c_1 = \lambda(a_1 :: \tau_1) \ \rightarrow ([a_1], [], ..., [])\\
&c_2 = \lambda(a_2 :: \tau_2) \ \rightarrow ([], [a_2], ..., [])\\
&...\\
&c_n = \lambda(a_n :: \tau_n) \rightarrow ([], [], ..., [a_n])
\end{aligned}
$$

In DSH, the names of the smart constructors are automatically derived from the constructor names of the original ADT. For example, the names of the *left* and *right* smart constructors are derived from the *Left* and *Right* constructors of the $Either$ data type.

We now turn to the elimination of ADT values in DSH queries. As an example, consider the following type signature of the eliminator (i.e., case distinction) for the $Either$ data type:

$$
\begin{aligned}
either\ ::\ &(Q\ a \rightarrow Q\ r)\\
&\rightarrow (Q\ b \rightarrow Q\ r)\\
&\rightarrow Q\ (Either\ a\ b)\\
&\rightarrow Q\ r
\end{aligned}
$$

For the $Left\ a$ case, $either$ applies its first argument to a. For the $Right\ b$ case, it applies its second argument to b. Internally, DSH exploits the representation of ADTs as a tuple of lists and implements this function by using the list processing combinators map, $+\!\!+$, and $head$, for all of which DSH provides efficient implementations on the relational backend [8]. More specifically, $either$ is implemented in terms of the following expression:

$$\lambda f\ g\ (a_1, a_2) \rightarrow head\ (map\ f\ a_1 +\!\!+ map\ g\ a_2)$$

Note that in this case and in the following general definition, the use of the $head$ function is safe because it is applied to a non-empty singleton list. As we have already mentioned, the smart constructors guarantee that exactly one field of the tuple (in the tuple-of-lists representation) is a singleton list. In DSH, the relational representation of a value e and a singleton list $[e]$ is exactly the same [11]. We can exploit this fact by removing the $head$ call just before query generation.

DSH compiles the map and $+\!\!+$ functions into relational queries involving projections and unions [16]. RDBMSs are heavily optimised for queries with these two relational operations.

Generalizing the example of the $Either$ data type, an eliminator for an ADT with the sums-of-products representation of $\tau_1 \oplus \tau_2 \oplus \ ... \ \oplus \tau_n$ (with $n \geqslant 1$) can be conceptually defined as follows:

$$
\begin{aligned}
elim\ &(a_1 :: [\tau_1], a_2 :: [\tau_2], ..., a_n :: [\tau_n])\\
&(f_1 :: \tau_1 \rightarrow r)\\
&(f_2 :: \tau_2 \rightarrow r)\\
&...\\
&(f_n :: \tau_n \rightarrow r) =\\
&head\ (map\ f_1\ a_1 +\!\!+ map\ f_2\ a_2 +\!\!+ ... +\!\!+ map\ f_n\ a_n)
\end{aligned}
$$

In DSH, the names of the eliminator functions are automatically derived from the type constructor of the original ADT. In addition to these type-specific eliminators, we provide overloaded, type-indexed eliminators (e.g., the $case\_$ function used in Section 2) that work with any nonrecursive ADT.

Currently, DSH makes use of Template Haskell [15] for compile-time generation of ADT constructors and eliminators. We are considering the Glasgow Haskell Compiler's new *generic deriving* mechanism [13] as an alternative to Template Haskell for this purpose. Generic deriving may provide for a simpler automatic derivation of ADT constructors and eliminators. For this and other implementation aspects of DSH we refer the interested reader to the DSH source code and documentation [1].

5. Future Work

We conclude this paper by outlining a number of items for future work.

- Although our initial results are encouraging, a comprehensive performance evaluation of language-integrated queries that operate on ADTs still lies ahead.
- In this work we have only dealt with nonrecursive ADTs. A natural next step would be to consider recursive types. We already support the recursive list type as a crucial, special case. Efficient relational support for arbitrary recursive ADTs (and recursive functions on them) is a harder problem. Although it is relatively straightforward to represent recursive data types relationally, compilation of various recursion schemes to efficient relational queries is challenging.
- The relational ADT encoding described in this paper bears strong similarities to the representation of ADTs in work on nested data parallelism in Data Parallel Haskell [4] which is based on the *flattening transformation* [3]. We currently explore harnessing the flattening transformation for query compilation.
- Ferry and DSH currently focus on the *query* aspect of database integration, we have not yet addressed the *update* aspect.

References

[1] Database Supported Haskell. `http://hackage.haskell.org/package/DSH`.

[2] Roland Backhouse, Patrik Jansson, Johan Jeuring, and Lambert Meertens. Generic Programming - An Introduction. In *Advanced Functional Programming*. Springer, 1999.

[3] Guy Blelloch and Gary Sabot. Compiling Collection-Oriented Languages onto Massively Parallel Computers. *J. Parallel Distrib. Comput.*, 8(2), 1990.

[4] Manuel Chakravarty and Gabriele Keller. More Types for Nested Data-Parallel Programming. In *Proc. ICFP*. ACM, 2000.

[5] William Cook and Ali Ibrahim. Integrating Programming Languages and Databases: What is the Problem? In *ODBMS.ORG, Expert Article*, 2005.

[6] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web Programming Without Tiers. In *Proc. FMCO*. Springer, 2007.

[7] George Copeland and David Maier. Making Smalltalk a Database System. In *Proc. SIGMOD*. ACM, 1984.

[8] George Giorgidze, Torsten Grust, Tom Schreiber, and Jeroen Weijers. Haskell Boards the Ferry: Database-Supported Program Execution for Haskell. In *Proc. IFL*. Springer, 2010.

[9] George Giorgidze, Torsten Grust, Nils Schweinsberg, and Jeroen Weijers. Bringing Back Monad Comprehensions. In *Proc. Haskell Symposium*. ACM, 2011.

[10] Torsten Grust and Manuel Mayr. A Deep Embedding of Queries into Ruby. In *Proc. ICDE*. IEEE, 2012.

[11] Torsten Grust, Manuel Mayr, Jan Rittinger, and Tom Schreiber. Ferry: Database-Supported Program Execution. In *Proc. SIGMOD*. ACM, 2009.

[12] Torsten Grust, Jan Rittinger, and Tom Schreiber. Avalanche-Safe LINQ Compilation. *PVLDB*, 3(1), 2010.

[13] José Pedro Magalhães, Atze Dijkstra, Johan Jeuring, and Andres Löh. A Generic Deriving Mechanism for Haskell. In *Proc. Haskell Symposium*. ACM, 2010.

[14] Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ: Reconciling Object, Relations and XML in the .NET Framework. In *Proc. SIGMOD*. ACM, 2006.

[15] Tim Sheard and Simon Peyton Jones. Template Meta-Programming for Haskell. In *Proc. Haskell Workshop*. ACM, 2002.

[16] Alexander Ulrich. A Ferry-Based Query Backend for the Links Programming Language. Master's thesis, University of Tübingen, 2011.

[17] Jan Van den Bussche. Simulation of the Nested Relational Algebra by the Flat Relational Algebra, with an Application to the Complexity of Evaluating Powerset Algebra Expressions. *Theor. Comput. Sci.*, 254(1-2), 2001.