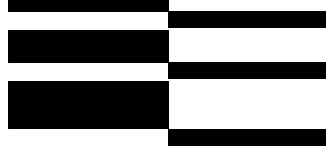
1



Implementation of the Object-Oriented Data Model TM

Hennie J. Steenhagen   Peter M.G. Apers

Abstract

Generally, one expects to find the solution to the growing need for database support in non-traditional application domains in object-oriented data models. Currently, at the University of Twente, work is being done on the high-level object-oriented data model **TM**. **TM** is an object-oriented data model, based on an extension of the type theory of Cardelli. **TM** includes a logical formalism for constraint specification, which is an important aspect of the data model, and the notion of predicative sets. Predicative set expressions provide for a high-level, descriptive specification mechanism. Topics of research are the theoretical foundations of **TM**, as well as implementation aspects. Just as in case of relational database systems, algebraic optimization is expected to be an important aspect of efficient implementation of the data model. With this expectation in mind, we have defined the language ADL, an algebraic database language based on the functional programming language FP, designed by Backus. Like FP, ADL is a language in which programs are functions, built up from primitive operators and functionals (higher order functions). In this paper, the main topic is the translation of **TM** to ADL. An algorithm is presented to translate (safe) **TM** expressions into ADL expressions. Furthermore, optimization in ADL is briefly discussed, and some equivalence rules are given.

1.1 Introduction

Nowadays, it has been widely accepted that the relational data model lacks certain modeling capabilities. The strength of the relational model, its simplicity, may at the same time be considered its weakness: in many new application domains, such as CAD/CAM and carthography, and even in the traditional business application domain, the requirement to store all sorts of data in tables causes inconvenience, as well as inefficiency.

Generally, one expects to find the solution to the growing need for database support in non-traditional application domains in object-oriented data models. Object-oriented data models provide for advanced modeling concepts with regard to the structure of the data, as well as the operations on it.

Though object-oriented database systems have been on the market for several years, there still seems to be no general consensus about what con-

stitutes an object-oriented data model. Several lists with 'necessary features,' 'desired features,' 'optional features,' and the like have been published [Atkinson et al. 89, Zdonik and Maier 90]. Moreover, little is known about the theoretical foundations of object-oriented data models and database management systems.

Currently, at the University of Twente, work is being done on the object-oriented data model **TM** [Balsters et al. 90B, Balsters and de Vreeze 91]. Foci of research are the theoretical foundations of **TM** as well as implementation aspects. **TM** is a strongly-typed, object-oriented data model supporting object-oriented concepts such as classes, object identity, complex objects, and multiple inheritance of data, methods, and constraints. As in O_2 [Lécluse and Richard 89], a distinction is made between objects (classes) and values (auxiliary classes). Constraint specification is an important aspect of the data model. In **TM**, it is possible to specify methods as well as constraints on three different levels: on the object, the class extension, and the database state level.

TM is based on the language **FM**, a strongly-typed, functional language, which, in turn, is based on the type system of Cardelli [Cardelli 84]. **FM**, supporting subtyping and inheritance, extends the Cardelli type theory by introducing set constructs and a logical formalism [Balsters et al. 90B]. Moreover, **FM** has been given a simple set-theoretical semantics [Balsters and Fokkinga 91, Balsters and de Vreeze 91, de Vreeze 91].

Refinement of the data model **TM**, still based on well-defined semantics, will take place guided by further study of user requirements in non-traditional application domains (carthography by name). To be able to judge functionality in an early stage, **TM** is implemented using ONTOS, the object-oriented database management system of Ontologic [ONTOS 90]. At the same time, research directed towards efficient implementation of **TM** is progressing. Because **TM** is a language with highly declarative language constructs, and also because **TM** is to be provided with a logical query language, algebraic optimization is expected to be an important aspect of efficient implementation of **TM**, just as in the case of the relational model.

In this paper, the main topic is the translation of **TM** to the language ADL (Algebraic Database Language). ADL is an algebraic language, based on the functional programming language FP, which was designed by Backus, and described in his famous Turing Award paper [Backus 78]. An important feature of FP is the emphasis on function-level programming. In FP, programming consists of nothing but the construction of functions from oth-

ers by means of functionals, which are higher-order functions. An important advantage of FP is the fact that, because FP is a variable-free language, it is possible to state clear, concise algebraic laws concerning program equivalence. Clearly, equivalence rules form the basis of program transformation and optimization.

The rest of the paper is organized as follows. In Section 2, we introduce the data model **TM** by means of an example. In Section 3, we informally present the language ADL, and in Section 4 the translation of **TM** to ADL is discussed. An algorithm for the translation of (a subset of) **TM** expressions is given. In Section 5 we present some equivalence rules, and finally, in Section 6, we outline future directions.

1.2
Introduction to TM

This section presents the data model **TM** by means of an example. It is not the intent to treat **TM** in full detail, but only to present those features that are interesting with regard to the translation of **TM** to ADL. For a more detailed overview of **TM** we refer to [Balsters et al. 90B], or the reference manual [Balsters et al. 90A].

In a nutshell, **TM** is a high-level, object-oriented data model, and as such it has the characteristic features of object-oriented data models: classes, object identity, complex objects, and (multiple) inheritance of data, methods, and constraints. Besides classes (objects), **TM** has auxiliary classes (values). The method specification language is essentially of a functional nature, but **TM** also has a general set notion. An important new feature of **TM** is the predicative set construct. **TM** includes a comprehensive constraint definition facility, and for constraint specification, a logical sublanguage based on first order predicate logic has been included.

Below we present the classical example to illustrate the main features of the model.

1.2.1 Example

Consider the following **TM** specification, modeling a database of persons, employees, managers, secretaries, and departments. (Explanation will follow in subsequent sections.)

```
begin specification

Class Person with extension Persons
  attributes
      name      :  string
      address   :  Address
      birthdate :  Date
      sex       :  string
      spouse    :  Person
  object constraints
      p1 :  sex = "M" or sex = "F"
  object methods
      update
      move (in newaddress:Address ) =
         self except (address = newaddress)
      retrieval
      age (out int) =
         system_date.year - birthdate.year
end Class Person

Class Employee ISA Person with extension Employees
  attributes
      salary :  int
  object methods
      update
      raise_salary (in amount:int ) =
         self except (salary = salary + amount)
  object constraints
      e1 :  salary > 30,000 and salary < 80,000
end Class Employee

Class Manager ISA Employee with extension Managers
  object constraints
      m1 :  salary > 60,000
end Class Manager

Class Secretary ISA Employee with extension Secretaries
  end Class Secretary

Class Department with extension Departments
  attributes
      name      :  string
```

```
      address   :  Address
      employees :  PEmployee
   object methods
      retrieval
      manager (out Manager) =
         unique  in {m:Manager | m sin self.employees}
   class methods
      retrieval
      emps_living_elsewhere (out PEmployee) =
         unnest (collect {e in d.employees | e.address.city ≠
         d.address.city} for d in self)
   object constraints
      de1  :  sum (collect e.salary for e  in self.employees) < 800,000
      de2  :  count {m:Manager | m sin self.employees} = 1
end Class Department

AuxClass Date
   type
      ⟨day:int,month:int,year:int⟩
   object constraints
      da1  :  1 ≤ day and day ≤ 31
      da2  :  1 ≤ month and month ≤ 12
      da3  :  month in {2,4,6,9,11} implies day ≠ 31
      da4  :  month = 2  implies day ≤ 29
      da5  :  month = 2 implies
         (year mod 4 ≠ 0 or
         (year mod 100 = 0 and year mod 400 ≠ 0) implies day ≠ 29)
end AuxClass Date

AuxClass Address
   type
      ⟨street:string,zipcode:string,city:string⟩
end AuxClass Address

Database specification
         system_date :  Date
   database constraints
         db1  :  forall m:Manager (forall s:Secretary
            (not exists e:Employee (m isa e and s isa e)))
end specification
```

1.2.2 Conceptual schema

A conceptual schema in **TM** consists of a collection of class and auxiliary class definitions and a database specification. In the *class definitions*, the structure of, the constraints holding for, and the operations on the entities of the universe of discourse are specified. In our example, entities of interest are persons, employees (which of course are persons), managers, secretaries (being employees), and departments. *Auxiliary class definitions* are useful to specify complex, dependent entity types. Dependent entities do not exist on their own, but are part of other entities. Dates and addresses are examples of dependent entities.

In the *database specification*, database attributes, database constraints, and database methods are specified. Database attributes are useful to store general database state information, such as the system date, or date of last change. Database constraints are useful to express constraints holding between classes. An example of a database constraint is constraint db_1, stating that a manager cannot be a secretary, and vice versa, i.e., the classes `Manager` and `Secretary` are disjoint. Database methods operate on the database as a whole, regarding the database as a record of classes, and are in fact transactions.

1.2.3 Classes

A class is denoted by its class name. Classes may have several superclasses, specified by the **ISA** clause, and the attributes, object constraints, and methods of the superclasses are inherited by the subclass. Inheritance in **TM** is based on an extension of Cardelli's subtyping relation. Class extensions are explicitely named (though there cannot be more than one).

Structurally class extensions are sets of records, and the names and domains of the record fields (*attributes*) are specified in the **attributes** clause. The **attributes** clause may be omitted in case one or more superclasses are defined. Notice the class `Secretary`, for which no attributes (nor methods nor constraints) are defined.

In **TM** *domains* are either basic domains or composite domains. Basic domains are basic types such as `int`, `real`, `bool`, `string` (and possibly others), class names, and auxiliary class names. Composite domains are record ($\langle\cdot\rangle$) and variant (or union) ($[\cdot]$) domains, and set ($\mathbf{P}\cdot$) and list ($\mathbf{L}\cdot$) domains. The following are examples of domains.

```
int
Employee
```

```
⟨str_nr:string, zipcode:string, city:string⟩
[home_address:⟨str_nr:string, city:string⟩, pobox:string]
PEmployee
⟨dept:Dept, emps:LEmployee⟩
```

The domain

```
[home_address:⟨str_nr:string, city:string⟩, pobox:string]
```

is an example of a variant domain: a mail address may be either a home address or a postoffice box.

The domains of the attributes of classes may be arbitrarily complex. Also recursive attribute domain specifications are allowed.

In **TM**, it is possible to specify *constraints* on three different levels: on the level of objects, on the level of class extensions, and on the level of database states [Balsters et al. 90B]. Constraints are labeled predicates. Object constraints should hold for each individual object in a class, class constraints should hold for each class extension, and database constraints (specified in the database specification) should hold for each database extension. Unlike object constraints, class constraints are *not* inherited by subclasses. In general, class constraints do not necessarily hold for subclasses. Consider for example a constraint on the class `Person`, stating that the percentage of female persons must lie between 40 and 60. Certainly in the Netherlands, this constraint does not hold for the subclass `Employee`.

The *methods* specified in a class definition are update or retrieval methods. A further distinction has been made between object and class methods. Subject of object methods are the individual members of a class, whereas in class methods sets of class members are the subject. Method definitions are of the form:

<method name> (**in** <variable-domain list> **out** <domain>) =
<method body>

The variable-domain list determines the names and domains of the formal input parameters. For retrieval methods the domain of the result of the method has to be specified. For update methods the domain of the result is known (the domain of **self**), so it may be omitted. The method body of a retrieval method and a class update method may be an arbitrary **TM** expression (to be described below); in object update methods, the **self except** construct is used to update attribute values. In the example some update and retrieval methods have been specified.

Besides classes, **TM** has auxiliary classes. Auxiliary classes differ from classes in that auxiliary classes do not have an extension, and instead of an attribute list a *type* is specified. Unlike auxiliary class names, class names may not occur in type specifications, and type specifications may not be recursive. As mentioned already, auxiliary classes are useful to specify complex dependent values (such as addresses and dates) with accompanying constraints and methods.

1.2.4 Expressions

TM expressions include the following: the special expression **self**, constants, variables, parenthesized expressions, and expressions having local definitions, defined with a **where** clause. For each composite domain there exists a collection of domain-specific language constructs to build up expressions of that domain (record, variant, set, and list expressions). Furthermore, **TM** has an if-then-else construct, arithmetical expressions, and aggregate expressions. Important types of expressions are the iteration expressions, for iterating over sets and lists, and predicates. And of course method calls belong to the expression language. Some of these expressions will be described in more detail below.

Self. When used in object constraints or object methods, **self** denotes the object at hand, when used in class constraints or class methods, **self** denotes the class extension.

Record expressions. A record expression is either an explicit record construction (such as ⟨`name="Jones",age=37,salary=50,000`⟩), a field selection (dot notation), or a record overwriting (used in object update methods). An example of a record overwriting is the expression **self except** `(address = newaddress)` (the body of the object update method `move` of the class `Person`), and the result is a record with the address field modified.

Set expressions. Set expressions are either enumerated set expressions, set expressions obtained through set operaters (**union**, **intersect**, **minus**), or predicative set expressions. An example of a predicative set expression is the following expression, delivering the set of employees not living in the city where the department `d` is located:

{`e` **in** `d.employees` | `e.address.city` ≠ `d.address.city`}

In fact, this expression is an abbreviation of the expression:

```
{e:Employee | e in d.employees and
                e.address.city ≠ d.address.city}
```

Note that predicative set expressions, when evaluated, have set values as a result, so predicative set expressions may, as any other expression, be used as part of other expressions.

List expressions. List expressions are either enumerated list expressions, or expressions built up by the operators **head**, **tail**, and **concat**.

Iteration expressions. Iteration expressions include expressions for selectively collecting or replacing elements of sets or lists, and for nesting and unnesting sets. Nesting is comparable to SQL's GROUP BY operation; unnesting a set of sets results in the union of those sets. We take the following collect expression, which contains a nested collect expression, as an example of an iteration expression:

```
collect (collect e
         for e in d.employees
         iff e.address.city ≠ d.address.city)
 for d in self
```

This expression determines, for each department d, the set of employees not living in the city where d is located. The general format of a collect expression is:

```
collect <result expression>
 for <variable> in <operand expression>
 iff <predicate>
```

The meaning of the collect expression is as follows. The operand expression is evaluated, a variable is iterated over the resulting set or list, for each value of the variable it is determined whether the predicate holds, if so, the result expression is evaluated, and this value is included into the resulting set or list. The result expression and the predicate are optional. If the result expression is missing, the set or list elements are included in the result unaltered; if the predicate is missing, the result expression is evaluated for each element of the operand.

Predicates. Predicates, always of type bool, are expressions built up by comparison operators, connectives **and**, **or**, **implies**, and **equiv**, negation **not**,

and quantifiers **forall** and **exists**. An example of a predicate is the database constraint concerning secretaries and managers:

```
forall m:Manager (forall s:Secretary
      (not exists e:Employee (m isa e and s isa e)))
```

stating that the classes `Manager` and `Secretary` are disjoint.

Some special comparison operators are **isa**, comparing objects of distinct types, such that one is a subtype of the other (equality modulo the subtype relation), $\simeq$ for shallow equality (where $=$ denotes deep equality), **in** and **subset** for element of and subset, and **sin** and **ssubset** for element of and subset modulo the subtype relation.

1.3 Introduction to ADL

ADL is an algebraic language for complex objects. The language is based on the functional programming language FP, designed by Backus [Backus 78]. Independently, Beeri and Kornatzky took a similar approach [Beeri and Kornatzky 90]. ADL and the language defined in [Beeri and Kornatzky 90] have very much in common. The main difference is that in [Beeri and Kornatzky 90] an abstract definition of the notion of data collection is used. Instead of having, for example, set, list, and array constructors, an abstract type 'constructor' is defined. However, a distinction is made between constructors that are permutable (e.g., the set constructor), constructors that eliminate duplicates (e.g., the set constructor), and constructors that eliminate nulls.

ADL is a typed language consisting of three layers:

- data objects – basic data objects (basic constants or persistent object names) and composite data objects (built up from other data objects by applying data constructors),
- operators – functions operating on data objects, delivering data objects, and
- functionals – higher order functions, having functions or data objects as parameters and delivering functions.

At present, ADL does not have subtyping nor parametric polymorphism.

1.3.1 Data objects

The data types of ADL are basic types or composite types. Basic types are int, real, bool, string, and oid, composite types are set types ($\{\cdot\}$), list types ($[\cdot]$), labeled tuple types ($< \cdot >$) and ordered tuple types ($(\cdot)$), and variant types ($<| \cdot |>$). Types can be arbitrarily nested. The data objects of ADL are either:

- basic data objects: basic constants and persistent object names
- composite data objects: data objects built up from other data objects by applying data constructors.

Let c denote basic constants, let v denote persistent variable names, and let a denote labels, then data objects o are given by the following syntax rule:

$$o ::= c \,|\, v \,|\, \{o, \ldots, o\} \,|\, [o, \ldots, o] \,|\, < a = o, \ldots, a = o > |\, (o, \ldots, o) \,|\, <| a = o |>$$

Examples of data objects are the following.

3, true, $\{1, 2, 3\}$, $\emptyset$, ("Doe", 25), <name="Doe", age=25>, Employees

We often refer to data objects as objects. Notice, however, that in ADL the term 'object' does not have the usual object-oriented connotation 'member of a class,' but instead stands for a value.

1.3.2 Operators

In the FP style of programming operators have only one argument. Multiple arguments are collected in sequences (the sequence constructor is the only data type constructor in FP). In ADL, we use ordered tuples for this purpose.

Operators are functions from data objects to data objects. For each data type a collection of useful operators is defined. Overloading is applied as much as possible, but we do not allow mixing of set and list type operands (e.g. appending a list to a set) for reasons of clarity.

ADL supports the following constructors as operators: the (unary) set constructor $\{\cdot\}$, the (unary) list constructor $[\cdot]$, the labeled tuple constructor $< a_1, \ldots, a_n >$ (where $a_1, \ldots, a_n$ are labels), and the variant constructor $<| a |>$ (a a label).

Important operators are selectors a (a a label), to select labeled tuple fields, and $s1, s2, \ldots$ (si for $i \in I\!N^+$), to select ordered tuple fields.

Furthermore, we have operators id (identity operator), : (prefix an element to a list or insert an element into a set), + (set union, list append and tuple concatenation), - (set difference and list subtraction), int (set and list

intersection), `prod` (Cartesian product of sets or lists), `hd` and `tl` (list head and tail), `flat` (union of set of sets and append of list of lists), `el` (taking the element from a singleton set), `mkset` (converting a list to a set) and `uniq` (removing duplicates from a list). In case of list operators, the order of the operands is preserved in obvious ways.

Aggregate operators supported are `min`, `max`, `cnt`, `sum`, and `avg`. Other operators included in the language are `sort`, for sorting a set or list, and `nest` and `unnest`, for both sets and lists of tuples, defined as in the nested relational model. Some of these operators have an extra parameter indicating the (possibly nested) attribute on which the operation has to take place.

Furthermore, we have arithmetical operators, comparison operators `<`, `<=`, `=`, `=>`, `>` (`=` also defined for objects of type `oid`), Boolean connectives `or` and `and`, and negation `not`.

1.3.3 Functionals

A functional is a higher order function: a function having functions and objects as parameters and delivering a function as result. Unlike operators, functionals have various arities. Roughly, functionals can be divided into two groups: the non-iteration functionals, and the iteration functionals.

Non-iteration functionals are functionals employed for control or construction. Iteration functionals are functionals mapping functions to elements of aggregate objects such as sets and lists. Iteration functionals correspond to the loop constructs from imperative programming languages.

Definitions are given below. In the definitions, : denotes function application. Notice, however, that : may also denote the prefix operator.

Non-iteration functionals

Constant. If o is an object, then $\mathtt{C}[\![o]\!]:x = o$. For example, $\mathtt{C}[\![\mathtt{1}]\!]:x = \mathtt{1}$, where x is an arbitrary object. This functional is useful to replace constants by functions.

Composition. Sequencing of functions is accomplished by function composition:

$$(f \circ g):x = f:(g:x)$$

Condition. The condition functional is a ternary functional:

$$(p \rightarrow f; g):x = \begin{cases} f:x & \text{if } p:x \\ g:x & \text{otherwise} \end{cases}$$

Construction. By means of construction it is possible to apply several functions to one object:

$$(f_1, \ldots, f_n) : x = (f_1 : x, \ldots, f_n : x)$$
$$\{f_1, \ldots, f_n\} : x = \{f_1 : x, \ldots, f_n : x\}$$
$$[f_1, \ldots, f_n] : x = [f_1 : x, \ldots, f_n : x]$$

Product. The product functional is used to distribute functions over a tuple of objects:

$$(f_1 \times \cdots \times f_n) : (x_1, \ldots, x_n) = (f_1 : x_1, \ldots, f_n : x_n)$$

The product functional can be expressed by means of construction and selector functions:

$$(f_1 \times \cdots \times f_n) = (f_1 \circ s1, \ldots, f_n \circ sn)$$

and is included in the language to avoid excessive use of selector functions.

Iteration functionals

Map. The functional `map` applies a function to every element of a set or list. For example, for a list:

$$\mathtt{map}[\![p]\!] : [\,] = [\,]$$
$$\mathtt{map}[\![f]\!] : [x_1, \ldots, x_n] = [f : x_1, \ldots, f : x_n]$$

Restrict. The functional `res` applies a predicate (a function having a Boolean result) to every element of a set or list and includes into the result those elements for which the predicate holds. For example, for a list:

$$\mathtt{res}[\![p]\!] : [\,] = [\,]$$
$$\mathtt{res}[\![p]\!] : [x_1, \ldots, x_n] = \begin{cases} x_1 :^{1} (\mathtt{res}[\![p]\!] : [x_2, \ldots, x_n]) & \text{if } p : x_1 \\ \mathtt{res}[\![p]\!] : [x_2, \ldots, x_n] & \text{otherwise} \end{cases}$$

Generate. The functional `gen` is a combination of the functionals `res` and `map`. The parameters of the functional `gen` are a predicate and a function. To each element of the operand set or list the function is applied, and then it is checked whether the predicate holds for the result of the function application. If so, the result of the function application is included in the result. The functional adds no power to the language, but it is included to be able to apply a function and a predicate to a set or list in succession, without having to create intermediate results (pipelining of operations [Beeri and Kornatzky 90]). For example, for a list:

[1]Note this : denotes the prefix operator, prefixing an element to a list (used in infix notation).

$$\texttt{gen}[\![p, f]\!] : [\,] = [\,]$$
$$\texttt{gen}[\![p, f]\!] : [x_1, \ldots, x_n] = \begin{cases} (f : x_1) :^1 (\texttt{gen}[\![p, f]\!] : [x_2, \ldots, x_n]) & \text{if } p : (f : x_1) \\ \texttt{gen}[\![p, f]\!] : [x_2, \ldots, x_n] & \text{otherwise} \end{cases}$$

Accumulate. The functional accumulate is a very powerful functional, which is also known as fold [Turner 85], or insert [Backus 78]. For lists, two different versions of the functional (right and left associative) exist:

$$\texttt{accr}[\![op, z]\!] : [x_1, \ldots, x_n] = x_1 \ op \ (x_2 \ op \ (\cdots (x_n \ op \ z) \cdots))$$
$$\texttt{accl}[\![op, z]\!] : [x_1, \ldots, x_n] = ((\cdots ((z \ op \ x_1) \ op \ x_2) \cdots) op \ x_n)$$

and, if applied to the empty list:

$$\texttt{accr}[\![op, z]\!] : [\,] = z$$
$$\texttt{accl}[\![op, z]\!] : [\,] = z$$

Note *op* is used in infix notation. The functional accumulate is also defined for sets, but then is it required, for the semantics to be well-defined, that the operator *op* is commutative and associative, because sets are unordered. The two versions of accumulate for lists then coincide (`acc`).

Forall and forsome. The forall functional tests whether a predicate holds for all elements of a set or list. For example, for a list:

$$\texttt{all}[\![p]\!] : [\,] = \texttt{true}$$
$$\texttt{all}[\![p]\!] : [x_1, \ldots, x_n] = \begin{cases} \texttt{all}[\![p]\!] : [x_2, \ldots, x_n] & \text{if } p : x_1 \\ \texttt{false} & \text{otherwise} \end{cases}$$

The forsome functional tests whether a predicate holds for at least one element of a set or list. For example, for a list:

$$\texttt{some}[\![p]\!] : [\,] = \texttt{false}$$
$$\texttt{some}[\![p]\!] : [x_1, \ldots, x_n] = \begin{cases} \texttt{true} & \text{if } p : x_1 \\ \texttt{some}[\![p]\!] : [x_2, \ldots, x_n] & \text{otherwise} \end{cases}$$

As is well-known from functional programming, many operators, and all other iteration functionals can be expressed in terms of `acc(l/r)`. This fact may be useful in generalizing equivalence rules and proofs. For lists, some examples using `accr` are given below.

$$\texttt{flat} = \texttt{accr}[\![+, [\,]]\!]$$
$$\texttt{cnt} = \texttt{accr}[\![+ \circ (1 \times \texttt{id}), 0]\!]$$
$$\texttt{sum} = \texttt{accr}[\![+, 0]\!]$$

$$\texttt{map}[\![f]\!] = \texttt{accr}[\![: \circ (f \times \texttt{id}), [\,]]\!]$$
$$\texttt{res}[\![p]\!] = \texttt{accr}[\![+\circ((p \rightarrow [\,]; \texttt{C}[\![[\,]]\!]) \times \texttt{id}), [\,]]\!]$$
$$\texttt{all}[\![p]\!] = \texttt{accr}[\![(p \circ s1 \rightarrow s2; \texttt{C}[\![\texttt{false}]\!]), \texttt{true}]\!]$$

1.3.4 Expressions

ADL expressions are of the form $f : o$, where f is a function, composed of functionals and operators, and o is an object.

1.4 Translation of TM to ADL

Mapping one typed language to another involves a mapping of types (or classes) and a mapping of expressions. In Section 4.1 we discuss the mapping of **TM** class definitions to ADL, in Section 4.2 we discuss the mapping of expressions. An example translation will be given in Section 4.3.

1.4.1 Translation of classes

Before we are in the position to discuss translation of **TM** expressions into ADL expressions, we have to decide how to translate the following (related) issues.

- Classes.
- Auxiliary classes.
- Object identity.
- Domains.
- Transparent references (see below).
- Inheritance of data, constraints, and methods.
- Constraints and methods.

Classes may be mapped to ADL object types in many different ways (possibly depending on performance characteristics). Class attributes may be distributed over several ADL object types, different classes may be brought together into one ADL object type, etc. Being the most obvious choice, in this paper we decide to map each **TM** class to one persistent object type in ADL. As a class extension structurally is a set of records, the corresponding persistent object in ADL is a set of labeled tuples.

Auxiliary classes have no extension; when used in object attribute definitions, an auxiliary class name is replaced by its (translated) type specification.

Object identity is implemented by means of object identifiers of type `oid`. Occurrences of class names in object attribute definitions are replaced by the type `oid`.

Mapping of basic or composite *domains* is straightforward: the collection of basic types of **TM** is a subset of the collection of basic types of ADL, and the collection of type (domain) constructors of both languages is identical. As an example we translate the following **TM** specification.

```
begin specification

AuxClass Date
  type
     ⟨day:int,month:int,year:int⟩
end AuxClass Date

Class Person with extension Persons
  attributes
      name      :  string
      birthdate :  Date
      spouse    :  Person
end Class Person

end specification
```

In ADL this is:

```
type Date = <day:int, month:int, year:int>
type Person = <name:string,
               birthdate:<day:int, month:int, year:int>,
               spouse:oid>
object Persons :  {Person} = ∅
```

In **TM**, relationships between classes are modeled by the use of class names in attribute domain specifications. By means of the dot notation we have direct access to objects being 'referenced' in attribute domain specifications, i.e. *references* are *transparent*. For example, in the **TM** specification above the name of the spouse of some person `p` is obtained with the expression `p.spouse.name`. Because class names are replaced by the type `oid`, 'dereferencing' must take place whenever attributes, of which the domain specification contains a class name (one or more), are accessed. In ADL this operation implies a 'join' between the two objects representing the class extensions.

Also *inheritance of data* can be mapped in many different ways. The mapping of inheritance of data in **TM** to ADL is comparable to the mapping of the EER concept of generalization to the relational model. The same options, as for example described in [Elmasri and Navathe 89], are applicable in this case. For example, one way to model inheritance of data on the ADL level is to include the attribute set of the superclass in the attribute set of the subclass(es), meaning that objects belonging to the superclass are distributed over several ADL persistent objects. Another way to model inheritance of data is to leave the attribute sets of sub- and superclasses as they are, and to use oid equality to implement the sub-superclass relationship. This means that attribute values of objects belonging to the superclass are distributed over several ADL persistent objects.

The specific choice of translation of inheritance of data of course influences the translation of constraints and methods. Translation of constraints and methods results in expressions $f:t$, where t, besides objects, also contains formal parameter names (the expression **self** is treated as a formal parameter name). When a method call or a constraint evaluation takes place, the actual parameters are substituted for the formal parameter names.

1.4.2 Translation of TM expressions

In this section we will give the outline of an algorithm to translate a subset of (safe) **TM** expressions to ADL expressions. In doing so, we assume there is no inheritance, and no class names occur in attribute specifications. Translation of inheritance and transparent references presents no fundamental problems, but this way we can concentrate on the translation algorithm itself, without having to take into consideration unnecessary details. Before giving the translation algorithm, we first discuss some general issues concerning safety and the translation of predicative set expressions.

General considerations

Safety

TM is a language with highly declarative language constructs. Predicative set expressions for instance are in general descriptive, not constructive: it is stated what the result of a query is, not how it should be obtained. In contrast, ADL is essentially a procedural language. ADL expressions represent a specific algorithm to find the answer to a query.

The declarative nature of **TM** causes problems with respect to termination, and even computability. For example, evaluation of the simple expression

{x:int |x > 0} will be non-terminating (if it even will get started computing). The expression {x:real|0 < x < 1} not only represents an infinite set, but also a set containing non-computable numbers. On the other hand, ADL expressions always represent finite and computable sets or lists.

To solve this problem we need a notion of safety: **TM** variables are allowed to range over finite sets only. A variable is safe if it ranges over a finite set; an expression is safe if the variables occurring in it are safe. Some examples of safe expressions are the following. Let c be a constant, e a safe set expression, and p a safe predicate.

1. {x:int|x = *c* }
2. {x:int|x **in** *e* **and** *p*(x)}
3. {x:int|**exists** v **in** *e* (x = v.a)}
4. {x:int|**exists** v **in** *e* (x = v.a **and** *p*(v))}
5. {x:int|**exists** v **in** *e* (x = v.a **and** *p*(x))}
6. {x:int|0 < x < 10}
7. {x:int|0 < x < 10 **and** x **mod** 2 = 0 }

The notion of safety is needed whenever in **TM** a clause of the form <variable>:<domain> is allowed, i.e. in predicative set expressions and quantified predicates. In [de By 91] a formal notion of safety is defined for a similar language. In the sequel, we assume all **TM** expressions to be translated are safe.

Predicative set expressions

Translation of most **TM** constructs and operators in itself is rather simple, because many **TM** constructs have ADL counterparts. Translation of the predicative set construct, however, is not so obvious, due to its possible highly descriptive nature. (Above all, predicative set expressions must be safe, as explained before.)

Considering the expressions listed above, each set is constructed differently. Evaluation of a predicative set expression almost always implies iterating over some set (the first expression being an exception to this rule). The set over which the iteration takes place either has already been constructed (the set e in expressions 2, 3, 4, and 5), or yet has to be constructed (expressions 6 and 7). For expression 6, the evaluation strategy may be to draw consecutive values from the domain of the positive integers, while the predicate holds. For expression 7, the strategy might be to do the same, but to stop when both

literals are false, and only to include those integers for which the conjunction is true. For the expressions 2, 3, 4, and 5, evaluation may come down to filtering an existing set (2), applying a function to an existing set (3), or both: filtering before application (4), or application before filtering (5).

There seems to be no general evaluation strategy for predicative sets; the choice of a specific strategy strongly depends on the form of the predicate. It is not clear yet, how, *in general*, predicative set expressions should be translated.

A restricted form of a predicative set expression which is easily translated is $\{\mathtt{v}:D \mid \mathtt{v}\ \mathbf{in}\ e\ \mathbf{and}\ p\}$, where D is a domain, e a set expression, and p a predicate. This may be abbreviated to $\{v\ \mathbf{in}\ e \mid \mathrm{p}\}$; its translation in ADL is an expression involving the restrict functional (see Section 4.2.2). Translation of predicative set expressions in general is a topic of further research.

Algorithm

We present the following simplified syntax for a subset of **TM** expressions. Among other things, variant and list expressions are not included for simplicity. Let v denote variables, c constants, a labels, and e expressions.

$$e ::= be \mid ie$$

$$\begin{aligned} be ::= \; & v \mid c \mid \mathbf{self} \mid \\ & \langle a = e, \ldots a = e \rangle \mid e.a \mid \\ & \{e, \ldots, e\} \mid \\ & \mathbf{if}\ e\ \mathbf{then}\ e\ \mathbf{else}\ e \mid \\ & unop(e) \mid e\ binop\ e \end{aligned}$$

$$\begin{aligned} ie ::= \; & \mathbf{collect}\ [e]\ \mathbf{for}\ v\ \mathbf{in}\ e\ [\mathbf{iff}\ e] \mid \\ & \{v\ \mathbf{in}\ e \mid e\} \end{aligned}$$

$$unop ::= \mathbf{not} \mid \mathbf{count} \mid \ldots$$

$$binop ::= = \mid + \mid - \mid / \mid * \mid \mathbf{and} \mid \mathbf{or} \mid \mathbf{union} \mid \ldots$$

Translation of a **TM** expression e (which is a constraint specification or a method body) proceeds bottom-up: subexpressions are translated first. Each subexpression of e is translated into an applicative expression $f:t$, where f is a function and t contains formal parameter names, class extension names, and iteration variables. Iteration variables are the bound variable names used

in iteration expressions, predicative set expressions, and quantified predicates. The expression **self**, which denotes an object (in object constraints or object method bodies), or a class extension (in class constraints or class method bodies), is treated as a formal parameter name. In the translation algorithm, formal parameter names, class extension names, and iteration variables are just called variables, or variable names. Ultimately, the expression e is translated into an applicative expression $f:t$, where t no longer contains iteration variables. Iteration variables are eliminated in the translation process.

We have the following translation for the expressions given in the syntax above.

Variables. Assuming identical name spaces, variables v (formal parameter names, class extension names, and iteration variables) are replaced by $\texttt{id}:v$. The function `id` is necessary because, as explained above, every subexpression is translated in an applicative expression $f:t$.

Constants. If a constant c occurs in a collect or predicative set expression iterating over variable v then c is replaced by $\texttt{C}[\![c]\!]:v$; if not, then c is replaced by $\texttt{C}[\![c]\!] : self$. Because every subexpression is translated in an applicative expression, the constant functional is given an artificial argument (the most obvious one in the context).

Self. The expression **self** is treated as a formal parameter name (a variable), so it is translated as $\texttt{id}:self$.

Record construction. Let $e = \langle a_1 = e_1, \ldots a_n = e_n \rangle$ be a record construction, and let $f_i : o_i$ be the translation of the expressions e_i for i within the range. The translation of e is then

$$< a_1, \ldots, a_n > \circ (f_1 \times \cdots \times f_n) : (o_1, \ldots, o_n)$$

Field selection. If $f:o$ is the translation of the expression e, then the translation of a field selection $e.a$, where a is an attribute name, is $(a \circ f):o$.

Set enumeration. Let $e = \{e_1, \ldots, e_n\}$ be an enumerated set expression, and let $f_i : o_i$ be the translation of the expressions e_i for i within the range. The translation of e is then

$$\{f_1 \circ s1, \ldots, f_n \circ sn\} : (o_1, \ldots, o_n)$$

The conditional. Let $f_i : o_i$ be the translation of the expression e_i for $1 \leq i \leq 3$, then the translation of the expression **if** e_1 **then** e_2 **else** e_3 is

$$(f_1 \circ s1 \rightarrow f_2 \circ s2; f_3 \circ s3) : (o_1, o_2, o_3)$$

Unary operations. If u is some unary operator, and $f : o$ is the translation of the expression e, then the translation of the expression $u(e)$ is $u' \circ f : o$, where u' is the ADL version of the operator u.

Binary operations. If b is some binary operator, and $f_i : o_i$ is the translation of the expression e_i for $1 \leq i \leq 2$, then the translation of the expression $e_1\ b\ e_2$ is $b' \circ (f_1 \times f_2) : (o_1, o_2)$, where b' is the ADL version of the operator b.

The collect expression. Let e be the expression **collect** e_1 **for** v **in** e_2 **iff** e_3, where e_2 is a set expression, and let

> $f_1 : o_1$ be the translation of the result expression e_1,
> $f_2 : o_2$ be the translation of the operand expression e_2,
> $f_3 : o_3$ be the translation of the predicate e_3.

Now, whenever a free variable name w occurs in o_1 or o_3, this variable is used in every step of the evaluation of the collect expression. By forming the product of the set $\{w\}$ and the operand object (tuples (w, v) for each v in the operand object), and applying the translated collect function f_1 and predicate f_3 to this product, using appropriate selector functions, we achieve a 'materialization' of iteration over a set or list with a free variable w. Of course o_1 or o_3 may contain several free variables.

So, let V be the set of variable names occurring in o_1 or in o_3. Let FV be the set $V \setminus \{v\}$, the set of free variable names occurring in o_1 or in o_3. Now two cases are distinguished: FV is not empty (free variables in o_1 or o_3, so a product has to be formed) or FV is empty (no free variables in o_1 or o_3; no product has to be formed).

1. $FV \neq \emptyset$. Let n be the cardinality of FV. Define a function $sel : FV \rightarrow \{s1, \ldots, s(n+1)\}$, with $sel(x) \in \{s1, \ldots, sn\}$, if $x \in FV$, and $sel(v) = s(n+1)$. The function sel binds each variable to a unique selector. Let o be the ordered tuple $(x_1, \ldots, x_n, o_2)$, where $x_i = x$ if $sel(x) = si$ for $1 \leq i \leq n$. The tuple o collects all variable names used in the expressions o_1, o_2, and o_3. Replace each $x \in V$ by $sel(x)$ in the expressions o_1 and o_3, and let the results be g_1 and g_3. In the expressions

o_1 and o_3, the variable names are replaced by selector functions, so that, instead of objects, we now have functions. Finally, the result of the translation of e is:

$$\mathtt{map}[\![f_1 \circ g_1]\!] \circ \mathtt{res}[\![f_3 \circ g_3]\!] \circ \mathtt{prod} \circ (\underbrace{\{\} \times \cdots \times \{\}}_{n\ times} \times f_2) : o$$

As explained, first tuples $(w_1, \ldots, w_n, v)$ are formed for each free variable w_i and each element v of the operand object, then a restriction on the product takes place, and finally the result is delivered by mapping $f_1 \circ g_1$ to the restricted product.

2. $FV = \emptyset$. Replace each $x \in V$ ($V = \{v\}$) by `id` in the expressions o_1 and o_3, resulting in g_1 and g_3. The expressions o_1 and o_3 cannot simply be omitted, because they may contain multiple occurrences of v. The result of translation of e is:

$$\mathtt{map}[\![f_1 \circ g_1]\!] \circ \mathtt{res}[\![f_3 \circ g_3]\!] \circ f_2 : o_2$$

If the result expression e_1 or the **iff** clause are missing, the translation can be easily adapted. If the result expression is missing, i.e., $e =$ **collect for** v **in** e_2 **iff** e_3, we omit the map functional; if the **iff** clause is missing, we omit the restrict functional. If both optional clauses are missing, then the collect expression is equivalent to the operand expression.

Predicative set expressions. Let e be the expression $\{v$ **in** $e_1 \mid e_2\}$, and let

$f_1 : o_1$ be the translation of the operand expression e_1,
$f_2 : o_2$ be the translation of the predicate e_2.

Let V be the set of variable names occurring in o_2. Let FV be the set $V \setminus \{v\}$, the set of free variable names occurring in o_2. Again, two cases are distinguished.

1. $FV \neq \emptyset$. Let n be the cardinality of FV. Define the function sel as above. Replace each $x \in V$ by $sel(x)$ in the expression o_2. Let the result be g_2. Let o be the ordered tuple $(x_1, \ldots, x_n, o_1)$, where $x_i = x$ if $sel(x) = si$ for $1 \leq i \leq n$. The result of the translation now is:

$$\mathtt{map}[\![s(n+1)]\!] \circ \mathtt{res}[\![f_2 \circ g_2]\!] \circ \mathtt{prod} \circ (\underbrace{\{\} \times \cdots \times \{\}}_{n\ times} \times f_1) : o$$

Whenever a predicative set expression is a subexpression, and its translation involves a product (that is, free variables occur), then the final result is achieved by a 'projection' on the product.

2. $FV = \emptyset$. Replace in the expression o_2 each $x \in V$ ($V = \{v\}$) by id, resulting in g_2. The result is:

$$\texttt{res}[\![f_2 \circ g_2]\!] \circ f_1 : o_1$$

As mentioned before, this form of predicative set expressions is a severely restricted form.

1.4.3 Example translation

To illustrate the algorithm we present the translation of the body of the class retrieval method `emps_living_elsewhere`, which was defined in the example specification given in Section 2.1. The result of the method is the set of employees who do not live in the same city as they work in.

In this example, the field selection `d.employees` occurs. Because each class is mapped to one persistent object type, and class names in attribute domain definitions are replaced by oid's, in the translation to ADL a dereference, or 'join,' should be inserted. However, for reasons of simplicity, in this case we translate the expression `d.employees` to the expression $\texttt{employees} \circ \texttt{id}{:}d$, without dereferencing, and we assume the selector `employees` delivers the desired result immediately.

In the translation below, translated subexpressions of intermediate forms have been underlined.

The initial expression is:

unnest (
  collect {e **in** d.employees |
      e.address.city ≠ d.address.city}
  for d **in self**)

The subexpressions of the nested predicative set expression are translated first. Before translating the entire predicate, the subexpressions of the binary operation are translated.

unnest (
  collect {e **in** $\underline{\texttt{employees} \circ \texttt{id}{:}d}$ |
      $\underline{\texttt{address} \circ \texttt{city} \circ \texttt{id}{:}e} \neq \underline{\texttt{address} \circ \texttt{city} \circ \texttt{id}{:}d}$ }
  for d **in self**)

Translation of the binary operation is easy:

$$\begin{array}{l}\textbf{unnest}\ (\\ \quad \textbf{collect}\ \{\mathtt{e}\ \ \textbf{in}\ \underline{\mathtt{employees} \circ \mathtt{id}\!:\!d}\ \mid \\ \qquad\qquad \underline{\neq \circ(\mathtt{address} \circ \mathtt{city} \circ \mathtt{id} \times \mathtt{address} \circ \mathtt{city} \circ \mathtt{id})\!:\!(e,d)}\ \}\\ \quad\ \textbf{for}\ \mathtt{d}\ \textbf{in}\ \textbf{self})\end{array}$$

We see that the expression $o_2 = (e, d)$ has a free variable d, so the first of the two options given above applies. We define $sel(d) = s1, sel(e) = s2$, and substitute:

$$\begin{array}{l}\textbf{unnest}\ (\\ \quad \textbf{collect}\ \underline{\mathtt{map}[\![s2]\!]\circ}\\ \qquad \underline{\mathtt{res}[\![\neq \circ(\mathtt{address} \circ \mathtt{city} \circ \mathtt{id} \times \mathtt{address} \circ \mathtt{city} \circ \mathtt{id}) \circ (s2, s1)]\!]\circ}\\ \qquad\quad \underline{\mathtt{prod} \circ (\{\} \times \mathtt{employees} \circ \mathtt{id})\!:\!(d,d)}\\ \quad\ \textbf{for}\ \mathtt{d}\ \textbf{in}\ \underline{\mathtt{id}\!:\!self)}\end{array}$$

For each department, the product is formed of the singleton set containing the department, and the set of employees working for the department. A projection on the desired employee-field follows the restriction on the product. Translation of the collect expression is simple: no free variables occur in $o_1 = (d, d)$, so the second option applies. The ADL version of the **unnest** operator is `flat`.

$$\begin{array}{l}\mathtt{flat} \circ \mathtt{map}[\![\mathtt{map}[\![s2]\!]\circ\\ \quad \mathtt{res}[\![\neq \circ(\mathtt{address} \circ \mathtt{city} \circ \mathtt{id} \times \mathtt{address} \circ \mathtt{city} \circ \mathtt{id}) \circ (s2, s1)]\!]\circ\\ \qquad \mathtt{prod} \circ (\{\} \times \mathtt{employees} \circ \mathtt{id}) \circ (\mathtt{id}, \mathtt{id})]\!] \circ \mathtt{id}\!:\!self\end{array}$$

The expression above can, by using algebraic rewrite rules (see Section 5), be simplified to:

$$\begin{array}{l}\mathtt{flat} \circ \mathtt{map}[\![\mathtt{map}[\![s2]\!] \circ \mathtt{res}[\![\neq \circ(\mathtt{address} \circ \mathtt{city} \times \mathtt{address} \circ \mathtt{city})]\!]\circ\\ \quad \mathtt{prod} \circ (\{\}, \mathtt{employees})]\!]\!:\!self\end{array}$$

The meaning of the result expression is as follows. For each department, the product is formed of the set of employees working for the department, and the singleton set containing the department. Then, for each tuple in the product (for each employee-department pair), it is checked whether the city the employee lives in is different from the city where the department is located. If so, the tuple is included in the result. After that, a field selection has to take place: we are not interested in the product tuples generated, but we only want the employee part (the second field). The final step consists of flattening the result (for each department a set of employees is delivered).

1.5 Optimization in ADL

Algebraic optimization involves the rewriting of an algebraic expression into a semantically equivalent (but syntactically different) expression, which can be evaluated at lower (or even minimal) cost. The rewrite process, being a difficult search problem, must be guided by a set of powerful heuristic rules. In this section, we will present some ADL equivalence rules, which may be useful in the rewriting process.

In [Backus 78] and [Beeri and Kornatzky 90] many examples of useful equivalence rules have been given already. However, in [Beeri and Kornatzky 90] rules are classified by the kinds of operators or functionals occurring in the rule. Here we will try to classify some rules by their intended purpose.

Heuristic optimization rules well-known from relational algebra optimization, such as:

- perform selections and projections as soon as possible,
- perform as many operations as possible during one access (grouping of operations),
- of a cascade of projections perform only the last one,

are just 'instantiations' of the more general optimization goals:

- reduce sizes of intermediate results,
- generate as few intermediate results as possible,
- avoid useless computations.

For these three optimization goals, which are generally valid, and possibly others, we have to find ADL analogues. In addition, we need a collection of equivalence rules that do not so much serve some specific optimization goal, but that are needed in the rewrite process.

Reduce sizes of intermediate results

Pushing restriction. The following equivalence rules are examples of the well-known selection rule, prescribing to perform selections as soon as possible. (Notice that, in general, equivalence rules must be read from left to right to achieve the desired effect.)

- Distributing restriction over binary operators is one example of the selection rule. For *op* $\in$ {`+`,`-`,`int`}, we have:

$$\mathtt{res}[\![p]\!] \circ op \equiv op \circ (\mathtt{res}[\![p]\!] \times \mathtt{res}[\![p]\!])$$

- For pushing a restriction beyond a product we have several rules:

$$\begin{array}{l} \mathtt{res}[\![p \circ q]\!] \circ \mathtt{prod} \circ (h_1, h_2) \equiv \mathtt{prod} \circ (\mathtt{res}[\![p]\!] \circ h_1, h_2) \\ \qquad \textbf{if } q \circ (h_1, h_2) \equiv h_1 \\ \mathtt{res}[\![p \circ q]\!] \circ \mathtt{prod} \circ (h_1, h_2) \equiv \mathtt{prod} \circ (h_1, \mathtt{res}[\![p]\!] \circ h_2) \\ \qquad \textbf{if } q \circ (h_1, h_2) \equiv h_2 \end{array}$$

  If a restriction follows a product operation, and the restriction predicate only involves one product field, then the restriction can be performed before the product is taken. Typically this is the case when the function q is a selector function.

  In the following rule the restriction predicate involves both fields of the product:

$$\mathtt{res}[\![\mathbf{and} \circ (p_1 \times p_2)]\!] \circ \mathtt{prod} \equiv \mathtt{prod} \circ (\mathtt{res}[\![p_1]\!] \times \mathtt{res}[\![p_2]\!])$$

- Pushing a restriction beyond an application is possible if the application does not affect the predicate domain, for instance when a projection is applied to some set of tuples, and this projection is followed by a restriction involving one of the tuple fields preserved.

$$\mathtt{res}[\![p \circ q]\!] \circ \mathtt{map}[\![f]\!] \equiv \mathtt{map}[\![f]\!] \circ \mathtt{res}[\![p \circ q]\!] \textbf{ if } q \circ f \equiv q$$

Pushing application. Pushing a `map` down the operator tree (or graph) is especially profitable when selector functions are mapped (projections) or when `map` is pushed beyond a product.

$$\begin{array}{l} \mathtt{map}[\![f \circ h]\!] \circ \mathtt{prod} \circ (g_1, g_2) \equiv \mathtt{map}[\![f]\!] \circ g_1 \\ \qquad \textbf{if } h \circ (g_1, g_2) \equiv g_1 \text{ and } g_1, g_2 \text{ deliver sets and} \\ \qquad\quad g_2 \text{ delivers a non-empty set} \\ \mathtt{map}[\![f \circ h]\!] \circ \mathtt{prod} \circ (g_1, g_2) \equiv \mathtt{map}[\![f]\!] \circ g_2 \\ \qquad \textbf{if } h \circ (g_1, g_2) \equiv g_2 \text{ and } g_1, g_2 \text{ deliver sets and} \\ \qquad\quad g_1 \text{ delivers a non-empty set} \\ \mathtt{map}[\![(f_1 \times f_2)]\!] \circ \mathtt{prod} \equiv \mathtt{prod} \circ (\mathtt{map}[\![f_1]\!] \times \mathtt{map}[\![f_2]\!]) \end{array}$$

Generate as few intermediate results as possible

Grouping (or pipelining, [Beeri and Kornatzky 90]) avoids the generation of intermediate results by applying several operations during one access. The following rules express grouping:

$$\begin{aligned}
&\mathtt{map}[\![f]\!] \circ \mathtt{map}[\![g]\!] \equiv \mathtt{map}[\![f \circ g]\!] \\
&\mathtt{res}[\![p]\!] \circ \mathtt{res}[\![q]\!] \equiv \mathtt{res}[\![\mathtt{and}\circ(p, q)]\!] \\
&\mathtt{res}[\![p]\!] \circ \mathtt{map}[\![f]\!] \equiv \mathtt{gen}[\![p, f]\!] \\
&\mathtt{map}[\![f]\!] \circ \mathtt{res}[\![p]\!] \equiv \mathtt{accr}[\![+\circ(p \rightarrow [\,] \circ f; \mathtt{C}[\![[\,]]\!]), [\,]]\!]
\end{aligned}$$

In the last expression the parameter function of the functional `accr` appends the singleton list $[f:x]$ to the result if $p:x$ is true, otherwise the empty list is appended.

Avoid useless computations

Some rather simple examples of this rule are the following:

$$\begin{aligned}
&f \circ \mathtt{id} \equiv \mathtt{id} \circ f \equiv f \\
&\mathtt{el} \circ \{\} \equiv \mathtt{id} \\
&\mathtt{map}[\![f]\!] \circ \{\} \equiv \{\} \circ f \\
&\mathtt{map}[\![(s1, s2)]\!] \circ \mathtt{prod} \equiv \mathtt{prod} \\
&h \circ (s2, s1) \circ (f, g) \equiv h \circ (g, f) \textbf{ if } h \text{ is commutative} \\
&(f_1 \times \cdots \times f_n) \circ \underbrace{(g, \ldots, g)}_{n\ times} \equiv (f_1, \cdots, f_n) \circ g \\
&si \circ (f_1, \cdots, f_n) \equiv f_i
\end{aligned}$$

1.6 Future work

With respect to the translation of **TM** to ADL, one of the problems to be addressed in the future concerns the formalization of the notion of safety in **TM**. Another problem is the translation of predicative set expressions. The work done in [Partsch 90] concerning the development of (efficient) programs from formal problem specifications may be of help in solving this problem.

The actual optimization by rewriting in ADL is a major topic of research. The following subproblems may be distinguished.

- Definition of a cost model, possibly incorporating different access routines.

- Definition of a suitable set of rewrite rules. Because ADL has many operators and functionals, a large set of equivalence rules can be written down. The task is to define a minimal set of useful equivalence, or rewrite rules.

- Definition of a rewrite algorithm (algebraic optimizer). As is well-known, even for the relational model algebraic optimization represents a difficult search problem, and construction of an algebraic optimizer is a hard task [Freytag 87, Graefe and DeWitt 87]. Because ADL is a rich language, the problem even gets harder.

Bibliography

[Atkinson et al. 89] Atkinson, M., Bancilhon F., DeWitt D., Dittrich K., Maier D., and Zdonik S. "The Object-Oriented Database System Manifesto." In *Proc. 1st DOOD*, 1989.

[Backus 78] Backus, J. "Can Programming Be Liberated from the von Neuman Style? A Functional Style and its Algebra of Programs." *Communications of the ACM*, 21 (8), 1978, pp. 613-641.

[Balsters et al. 90A] Balsters, H., de By R.A., and de Vreeze C.C. "The **TM** Manual version 1.2." Manuscript, University of Twente, Enschede, 1990.

[Balsters et al. 90B] Balsters, H., de By R.A., and Zicari R. "Sets and Constraints in an Object-Oriented Data Model." Memoranda Informatica 90-75, University of Twente, Enschede, 1990.

[Balsters and Fokkinga 91] Balsters, H. and Fokkinga M.M. "Subtyping can have a Simple Semantics." To appear in TCS 87, 1991.

[Balsters and de Vreeze 91] Balsters, H. and de Vreeze C.C. "A Semantics of Object-Oriented Sets." In *Proc. 3rd International Workshop on Database Programming Languages"* , Nafplion, Greece, 1991.

[Beeri and Kornatzky 90] Beeri, C. and Kornatzky Y. "Algebraic Optimization of Object-Oriented Query Languages." In *Proc. 3rd ICDT* , LNCS 470, Springer-Verlag, 1990.

[de By 91] de By, R.A. "The Integration of Specification Aspects in Database Design." Ph.D. Thesis, University of Twente, Enschede, 1991.

[Cardelli 84] Cardelli, L. "A Semantics of Multiple Inheritance." In *Semantics of Data Types*, (Kahn G., MacQueen D.B., and Plotkin G. eds.), LNCS 173, Springer-Verlag, 1984, pp. 51-67.

[Elmasri and Navathe 89] Elmasri, R. and Navathe S.B. *Fundamentals of Database Systems.* Benjamin/Cummings Publishing Company Inc., 1989.

[Freytag 87] Freytag, J.C. "A Rule-Based View of Query Optimization." In *Proc. ACM SIGMOD*, 1987, pp. 173-180.

[Graefe and DeWitt 87] Graefe, G. and DeWitt D.J. "The EXODUS Optimizer Generator." In *Proc. ACM SIGMOD*, 1987, pp. 160-172

[Lécluse and Richard 89] Lécluse, C. and Richard P. "The O_2 Database Programming Language." In *Proc. 15th VLDB*, Amsterdam, 1989.

[ONTOS 90] "ONTOS Object Database Developer's Guide." Ontologic Inc., Burlington, 1990.

[Partsch 90] Partsch, H.A. *Specification and Transformation of Programs – A Formal Approach to Software Development.* Springer-Verlag, 1990.

[Turner 85] Turner, D.A. "Miranda: A Non-Strict Functional Language with Polymorphic Types." In *Proc. IFIP International Conference on Functional Programming Languages and Computer Architecture*, LNCS 201, Springer-Verlag 201, 1985.

[de Vreeze 91] de Vreeze, C.C. "Formalization of Inheritance of Methods in an Object-Oriented Data Model." Memoranda Informatica 90-76, University of Twente, Enschede, 1991.

[Zdonik and Maier 90] Zdonik, S.B. and Maier D., eds. *Readings in Object-Oriented Database Systems.* Morgan Kaufmann Publishers, 1990.