Fact-Oriented Modeling from a Programming Language Designer's Perspective

Betsy Pepels[1,3], Rinus Plasmeijer[1], and H.A. (Erik) Proper[2]

[1]Software Technology, [2]Information Retrieval and Information Systems
both of Radboud University Nijmegen, The Netherlands
[3]Informatics and Communication Academy,
HAN University of Applied Science, The Netherlands
{betsy, rinus, E.Proper}@cs.ru.nl

Abstract. We investigate how achievements of programming languages research can be used for designing and extending fact oriented modeling languages. Our core contribution is that we show how extending fact oriented modeling languages with the *single* concept of *algebraic data types* leads to a natural and straightforward modeling of complex information structures like unnamed collection types and higher order types.

1 Introduction

In this paper we consider modeling languages based on the fact-oriented paradigm [1]. The most well known are ORM [2] with its successor ORM2 [3], NIAM [4], FCO-IM [5], and PSM [6]. This group of closely related dialects we will call fact-oriented modeling (FOM) languages.

FOM has proven to be a powerful approach; yet for some information structures *easier or more intuitive* modeling facilities could be available. For instance types being types themselves (categorization types), unnamed collection types, and the crossing of levels/metalevels are difficult to model [7]. In [8] various modeling problems are addressed, like the identification of Dutch Cabinets, which we present in Section 4.

Throughout the above and related publications there is an on-going, but less structured and less explicit discussion about the *necessity* to introduce new modeling concepts. For instance, PSM extends the basic FOM expressive facilities with concepts like *Set* and *Sequence* to model unnamed sets.

In this paper, we take part in both discussions yet from a different angle: we treat FOM languages from the perspective of programming languages theory, and especially one of its sub-disciplines, type theory. Programming language theory makes a distinction between expressiveness being conceptually *essential* and expressiveness being *convenient*. Type theory provides the formal basis for the design, analysis and study of type systems. Type systems offer many powerful possibilities for modeling data structures.

Our aim is to demonstrate that achievements of programming languages theory can fruitfully be used for designing and extending FOM languages. As our

R. Meersman, Z. Tari, P. Herrero et al. (Eds.): OTM Workshops 2006, LNCS 4278, pp. 1170–1180, 2006.

core contribution we show in the Sections 3 and 4 that extending the *essential* expressiveness of FOM languages with the *single* concept of *algebraic data types* allows a natural and straightforward modeling of the aforementioned information structures and many others too.

We conclude with a brief sketch of some of the expected benefits and the research questions arising when FOM languages are extended following the programming languages approach.

2 Achievements of Programming Language Theory

In this subsection we give a short summary of the approach commonly accepted for the (formal) design of programming languages [9] and we introduce algebraic data types as well.

2.1 Formal Design of a Language

The expressive possibilities of languages are layered, as illustrated in Figure 1. The basis of formal languages is a *computational model.* Examples of computational models are the Turing machine, the lambda calculus, the relational algebra and Petri nets. The *aim* of a computational model is to establish a mathematical foundation for computations possible in a language. A computational model is a mathematical model and commonly has a simple and clear semantics.

With the computational model the essential computational *power* of the language is defined: every computation possible in the language has to be expressible in the computational model too.

On this mathematical foundation a core language is defined. The big difference between a computational model and a language is that the latter is meant to define programs in and that it can be executed on a suitable platform. Compared to the computational model, a core language does not have more *computational* power, only more convenience. It offers additional constructs to support programming, for instance module structures. Every construct in the core language has either a direct counterpart in the computational model, or can be translated to it.

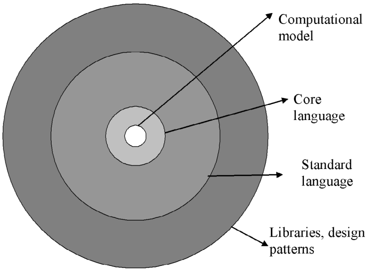


Fig. 1. Layering

The core language is often extended with all kinds of syntactic constructs for convenient programming. An example is the (Java and C construct) `i++` which is used for expressing a loop counter. This syntactically extended language is the *standard* language programmers work with. It is geared towards practicability and usability.

Most often a plethora of facilities enabling further ease of programming is available, like libraries, design patterns etc. All these facilities have in common that they are basically *programs* using the language defined by the lower layers.

Examples of languages defined this way are SQL (based on the relational algebra), workflow languages like YAWL [10] (based on Extended Workflow Nets which are again based on Petri nets), functional programming languages like Haskell [11] and Clean [12] (based on term rewriting systems which are again based on the lambda calculus) and the .NET framework [13].

It might be clear that no sharp distinction between the several layers can be made. Computational models are often layered themselves. Furthermore, some expressive feature of a language might be found either in the core language, in the standard language, or in libraries.

2.2 Algebraic Data Types

We briefly introduce algebraic data types. In the next section we will show how algebraic data types can be part of FOM models. An algebraic data type is a language construct known from the functional programming languages field. It can be regarded as a grammar with which new types can be defined. We explain this by a well-known example, the `List` data type:

```
:: List a = Cons a (List a) | Nil
```

For the notation we adopt the notation used in the functional programming language Clean. The `::` is a keyword indicating the beginning of a type definition. The definition should be read as the type `List` (holding elements of any type *a*) is *either* (indicated by the symbol `|`) the `Cons` of an element of type *a* and a `List` or `Nil` (the empty list). Square brackets `[]` are often used as a shorthand notation for lists. Examples of lists can be found in Table 1.

Table 1. Example lists

Example list	Shorthand notation	Which list?
`Nil`	`[]`	the empty list
`Cons 1 (Cons 2 Nil)`	`[1,2]`	the list consisting of the integers 1 and 2
`Cons 3.14 Nil`	`[3.14]`	the list consisting of the real 3.14

This `List` data type is *recursive*: the constructor `Cons` holds sub-values of the type `List`. It is also *polymorphic*: a `List` can hold values of *any* type.

In algebraic data types there is a difference between *types* and *terms*. `List Int` is a type and the list `Cons 1 (Cons 2 Nil)` is a term.

Type systems and hence algebraic data types are commonly specified by grammars. In Figure 2 we give a grammar for the definition of an algebraic type using EBNF notation. This grammar is taken from [12]; we will use it also in the next sections. `{q}+` means one or more appearances of `q`. Terminals are denoted in capitals, non-terminals in lower case. The underlined symbols are terminals as well, to make a distinction between similar symbols of the grammar itself. Note that the grammar also describes algebraic data types that are not well-formed.

One of the (many) well-formedness requirements is that a type variable used in the right hand side should be defined in the left hand side of the definition too.

3 Extending Fact-Oriented Modeling Languages

In this section we will demonstrate how, using the approach described in Section 2, FOM languages can be extended. Distinguishing between fundamental expressiveness and convenient expressiveness is applicable to FOM languages as well. There is however a difference with programming languages: FOM is not primarily meant to write programs in and is not expected to be *executed* (yet).

```
algebraicTypeDef = :: typeLhs = constructorDef {| constructorDef}
typeLhs = typeConstructorName {typeVariable}
constructorDef = constructorName {type}
type = typeVariable | TypeConstructorName | (type) | basicType
basicType = Int | Real | Char | Bool | String
typeConstructorName :== String
typeVariable :== String
```

Fig. 2. Grammar describing algebraic data types

3.1 A Grammar for Basic FOM Structures

As for the algebraic data types, we give in Figure 3 a grammar describing basic structures of fact oriented models. Our grammar describes what we consider to be a core FOM language. We limit ourselves considerably: we only describe a language for the *structure* of fact oriented models, not (yet) taking into account specialization and generalization, and leave out aspects like constraints as well. These restrictions however do not influence the generality of our discussion.

```
model = {namedtype}+
namedtype = facttype | objecttype
facttype = (NAME identifier, FT {role}+)
objecttype = (NAME identifier, OBJ identifier)
role = (PB roleplayer)
roleplayer = labeltype | identifier // identifier must be existing name
labeltype = LB identifier
identifier :== String
```

Fig. 3. Grammar describing FOM structures

We describe *the* basic characteristic of FOM structures: their basis in *facts*. Our grammar only provides for expressing fact types (in the grammar: FT), label types (in the grammar: LB) and objectification (in the grammar: OBJ). Roles are either played by (in the scheme: PB) label types or by items in the scheme that have a *name* (which is used to refer to them), the latter being either fact types or object types. In the next subsection we will explain how other constructs can be translated to the language defined by this grammar.

One of the well-formedness requirements for this grammar is that if a role is played by an identifier, this identifier must be the elsewhere defined name of a fact type or object type. Note furthermore that the grammar just produces fact oriented models in a mechanical manner: the *syntax* of the language. Its *semantics* have to be assigned separately. For FOM this is commonly done using set theory, for instance in [14]. Following the layered scheme of Section 2.1, we could designate set theory the *computational model*.

Example model. The same (well-formed) model defined by the grammar is expressed textually in Figure 4 and diagrammatically in Figure 5.

```
(NAME ft1, FT (PB (LB lb1)))
(NAME o1, OBJ ft1)
(NAME ft2, (PB o1) (PB o1))
(NAME o2, OBJ ft2)
(NAME ft3, FT (PB o2) (PB lb2))
```

Fig. 4. Example scheme as text

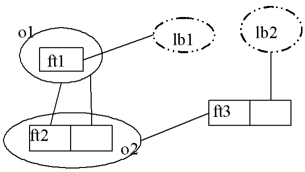


Fig. 5. Example scheme as diagram

3.2 Mapping Non-basic Structures to Basic Structures

Our grammar defines a core language. FOM languages like FCO-IM, ORM and PSM allow more constructs than our grammar defines; for instance ORM and PSM have *Objects* which are not defined by our grammar. Following the layered scheme we introduced in Section 2.1, we will now point out how some particular constructs of these FOM languages can be mapped to the language defined by our grammar.

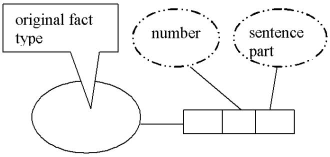


Fig. 6. Mapping of sentence

Mapping example 1. The structure of FCO-IM schemes is very much like those defined by our grammar. Each label and fact type in an FCO-IM scheme directly correspond to a similar label and fact type in our computational model. FCO-IM is fully communication oriented and hence every fact type (unary and higher) must be accompanied by a *sentence*. Such a sentence can be mapped to our language by objectifying the original fact type and adding a fact type expressing the sentence, as demonstrated in Figure 6. The fact type in the figure is to be read as "The fact type ⟨`Fact type`⟩ has as ⟨`number`⟩ th sentence part ⟨`sentence part`⟩.".

Mapping example 2. The ORM and ORM2 languages (and NIAM as well) have *Entity types*. These are not directly defined by our grammar, but can be translated to objectified unary fact types. ORM objects are required to have an identification, which is naturally accomplished this way.

Some constructs cannot be mapped to our core language:

Non-mapping example. PSM has the *Set* and *Sequence* type enabling the modeling of unnamed collections. With the *Set* type a fact type like ”Trainer ⟨`Name`⟩ trains team ⟨`Set of Player`⟩.” can be phrased. Such a structure fundamentally cannot be expressed in our core language.
The Set type is just one of the many examples of a *collection* type: a type encompassing a collection of members and having certain characteristics (like no duplicates for Set and an order for Sequence). Collection types are analyzed in [15].

FOM languages struggle with collection types, especially when they are unnamed. Many strategies to tackle this problem have been proposed, like introducing types for kinds of collections (in PSM), the extensional uniqueness constraint [16] thus avoiding complex semantics, higher order logic [17], and avoiding the problems at all by remodeling to a first-order scheme.

Our way to address this problem is by adding algebraic data types to the fundamental expressiveness of FOM languages.

3.3 Adding Algebraic Types to Fact Oriented Models

Our core idea is to allow that in FOM schemes *roles* can be played by algebraic data types as well. This we express by updating our grammar (see Figure 7).

```
roleplayer = labeltype | identifier | algebraictype
algebraictype = typeConstructorName roleplayer
```

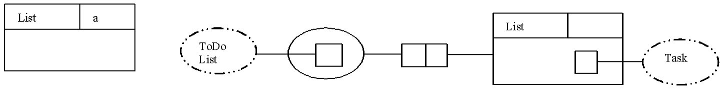
Fig. 7. Updates to grammar for FOM structures

The definition of the algebraic data type itself is not part of the scheme, but is to be given *separately* as a kind of program text.

A graphical notation for an algebraic data type has to be chosen. As yet, we pick *just one* of the many diagrammatic possibilities: a rectangle with the name of the type in it. Independent of which representation is chosen, somehow in the scheme it should be referred to the algebraic data type.

Illustrating example. A ToDo list, identified by its name, holds a list of tasks, identified by their names. Using the `List` type defined earlier we can phrase the fact type ”The ToDo list ⟨`Name`⟩ holds tasks ⟨`List Task`⟩”. This is illustrated in Figure 8. A example fact is ”The ToDo list *Urgent* holds tasks *[Finish research proposal, Mark recent exams].*”

The observant reader will notice that in fact types (algebraic) *types* are used, whereas facts use *terms*.



(a) Just the algebraic data type

(b) Use in the scheme

Fig. 8. Picturing algebraic data types

4 Modeling Examples

We now demonstrate with some examples how algebraic data types can be used for easy and straightforward modeling of complex information structures.

Unnamed list. For a medical survey, (only) the sex and birth dates of children of the involved families are recorded. Families are identified by a `FamilyId` (a number). Conceptually, per family an unnamed list of combinations of sex and birthdate is recorded. We define the following types:

```
:: ChildInfo = Info Sex Date
:: Sex = Male | Female
:: Date = D Year Month Day
```

Now the information recorded for a family is a `List` of `ChildInfo`, and for the information structure we phrase the fact type: ”Family ⟨`FamilyId`⟩ has children’s info ⟨`List ChildInfo`⟩.” Some example facts are: ”Family *23987* has children’s info *[]*.” and ”Family *56342* has children’s info *[Info Female (D 2001 12 5)]*.” and ”Family *34231* has children’s info *[Info Male (D 2002 3 25), Info Male (D 2002 3 25), Info Female (D 2003 11 16)]*.”, the last family having a male twin.

Identification of Dutch cabinets. The following example we borrow from [8]. The authors artfully solve this problem and others too by using generalization and by allowing null values in populations having a uniqueness constraint as well.

In the Netherlands a cabinet is named after its Prime Minister. For instance the Cabinet Den Uyl governed from 1973 to 1977. When a Prime Minister serves more than one term, the corresponding cabinets are named I, II, III, etc. For instance, the Cabinets Lubbers I, Lubbers II and Lubbers III were three successive cabinets governing from 1982 to 1994. To model this information structure, we define the following type:

```
:: CabinetName = OneTerm String | MoreTerms String RomanInt
:: RomanInt = ...
```

We now can phrase the fact type ”The term of cabinet ⟨`CabinetName`⟩ started in the year ⟨`Year`⟩.” Some example facts are: ”The term of cabinet *(OneTerm Den Uyl)* started in *1973*.” and ”The term of cabinet *(MoreTerms Lubbers I)* started in *1982*.”.

Set type. The well-known game of Quamsplash [5] is played by unnamed teams consisting of players identified by their name. Teams play matches against each other. To model this information structure, we define the algebraic data type:

`:: Set a = E | S a (Set a)` // E for Empty Set

Sets do not have duplicate members. Notice that such a property can not expressed with an algebraic data type, but should be implemented by access functions of the data type.

With the above `Set` type, and assuming that the type `PlayerName` is a string, a match is described by the fact type "⟨`Set PlayerName`⟩ plays against ⟨`Set PlayerName`⟩". An example fact is "Team *Set Guido E* plays against team *Set (Jan Pieter (Set Marko (Set Fazat E)))*.", or using the shorthand notation with curly brackets {} "Team {*Guido*} plays against team {*Jan Pieter, Marko, Fazat*}."

Higher order types. We present an information structure that is very much like the example in [7], which was again borrowed from [17].

ACME (A Company that Makes Everything) produces everything. Table 2 is a tiny part of the data available about ACME, presenting a list of its best selling products and some of their attributes. The third column specifies the colors in which a product is made by ACME. Some products can be enhanced with options, given in the fourth column. For example, for the *Portable hole* is made in two colors and optionally can be provided with an *Explosive* and/or a *Fence*. The last column indicates what attributes of the product are up to the costumer to be chosen. For instance, although the *Mojo* is manufactured in two colors, the customer cannot choose: the color is a surprise upon receipt.

Notice that Table 2 only gives information about ACME products, not about actual choices of customers.

Table 2. Best selling products of ACME

Prod.	**Product name**	**Possible colors**	**Options**	**Customer chosen**
101	Personal jetpack	{Green, Brown}	–	{Colors}
102	Inflatable submarine	{Yellow}	{Restroom}	{Options}
701	Portable hole	{Black, Grey}	{Explosive, Fence}	{Colors, Options}
1001	Mojo	{Pink, Purple}	–	{}

To elucidate the modeling problem, we quote from [7]: "First, it is in non-first normal form, allowing unnamed sets as entries (for instance Colors). ⟨snip⟩ Secondly, its final attribute (column) allows as entries unnamed sets whose instances appear to be attributes themselves, thus crossing levels/metalevels."

Using the standard FOM approach, this information structure either can be treated directly but then higher order logic is needed, or it is to be transformed to a first order scheme. Using algebraic data types, modeling can be done almost straightforwardly, because both metalevels and levels are just types. We define the following types (assuming that `ColorName` and `OptionName` are both strings):

```
:: Colors = Set ColorName
:: Options = Set OptionName
:: CustomerChoice = C Colors | O Options
:: CustomerChosen = Set CustumerChoice
```

Table 3 gives the fact types and example facts using the types defined above, where furthermore `no` is a number and `name` is a string.

Table 3. Fact types and example facts

Fact type (using types)	**Example fact** (using terms)
Product ⟨`no`⟩ has name ⟨`name`⟩.	Product *1001* has name *Mojo.*
Product ⟨`no`⟩ has possible colors ⟨`Colors`⟩.	Product *701* has possible colors {*Black, Grey*}.
Product ⟨`no`⟩ has options ⟨`Options`⟩.	Product *102* has options {*Restroom*}.
For product ⟨`no`⟩ may be chosen by customer ⟨`CustomerChosen`⟩.	For product *101* may be chosen by customer {*C Colors*}.

5 Reflection, Conclusion and Future Work

We showed how fact oriented models can be extended with algebraic data types, using the approach from the programming languages field. Thus we obtain natural and straightforward modeling of complex information structures like unnamed collection types and higher order types.

By introducing algebraic data types, we move from set theory and first order logic as formal foundation of FOM to type theory, the latter having its origins in the Principia Mathematica by Russell and Whitehead [18]. Set theory and type theory were both answers to paradoxes arising from naive set theory. Set theory together with classical logic is the standard foundation of modern mathematics, whereas type theory is one of the major pillars of computer science.

By regarding fact oriented models from the programming language perspective (by regarding them as types), a wealth of results from decades of programming language research might become applicable. We briefly mention the most interesting *research issues*:

Abstraction. Type theory provides for mechanisms for the definition of many more advanced types, like abstract data types and quantified types. Furthermore, type systems offer tremendously powerful abstraction mechanisms, which are one of the core features of functional programming languages. Abstraction is a heavily desired feature for fact oriented models, as many authors previously pointed out [19], [20], [21], [22].

Integration of functionality. With programming languages data structures are defined (using types) as well as algorithms manipulating these data structures. By having types as basis for fact oriented modeling languages we can integrate functionality very naturally into information models. An apparent example

are abstract data types, which define a specification of a set of data and the set of operations that are allowed on the data.
Implementation. With the definition of complex data types in fact oriented models, problems with mapping them to a target platform (for example relational tables) arise. This is a major research subject. A promising starting point is polytypic programming [23], using transformations working on types (models). Research for mapping types to relational tables can be based on this technology, together with the already existing orthogonal persistence of any type in files [24].

In our opinion it would be valuable to start a discussion in the fact oriented modeling community whether and how FOM languages could be designed and extended following the approach we sketched.

References

1. T.A. Halpin and M.E. Orlowska. Fact–oriented modelling for data analysis. *Journal of Information Systems*, 2(2):97–119, April 1992.
2. T. Halpin. Object-role modeling (ORM/NIAM). In P. Bernus, K. Mertins and G. Schmidt, editors, *Handbook on Architectures of Information Systems*. Springer Verlag, 1998.
3. Terry Halpin. ORM 2. In Robert Meersman, Zahir Tari, and Pilar Herrero, editors, *On the Move to Meaningful Internet Systems 2005: OTM 2005 Workshops*, volume 3762 of *LNCS*, pages 676–687, 2005.
4. G. M. Nijssen and Terry Halpin. *Conceptual Schema and Relational Database Design*. Prentice Hall, 1989.
5. G.P. Bakema, J.P.C. Zwart, and H. van der Lek. Fully communication oriented NIAM. In *NIAM-ISDM 1994 Conference, Working Papers*, pages L1–L35, 1994.
6. A.H.M. ter Hofstede and Th.P. van der Weide. Expressiveness in conceptual data modelling. *Data & Knowledge Engineering*, 10(1):65–100, February 1993.
7. Terry A. Halpin. Information modeling and higher-order types. In *CAiSE Workshops (1)*, pages 233–248, 2004.
8. Guido Bakema, Jan Pieter Zwart, and Harm van der Lek. *Volledig Communicatiegeorinteerde Informatiemodellering FCO-IM*. Academic Service, The Netherlands, 2005. Textbook in Dutch. The English version can be downloaded via http://www.casetalk.com/php/index.php?FCO-IM%20English%20Book.
9. Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.
10. W.M.P. van der Aalst and A.H.M. ter Hofstede. YAWL: yet another workflow language. *Information Systems*, 30(4):245–275, 2005.
11. Simon Peyton Jones et al. *Haskell 98 Language and Libraries: the Revised Report*. Cambridge University Press, 2003.
12. Rinus Plasmeijer and Marko van Eekelen. *Concurrent CLEAN Language Report (version 2.0)*, December 2001. http://www.cs.ru.nl/∼clean/.
13. http://www.microsoft.com/net/default.mspx. The `.NET` website.
14. Arthur H.M. ter Hofstede. *Information Modelling in Data Intensive Domains*. PhD thesis, University of Nijmegen, The Netherlands, 1993.
15. Terry Halpin. Modeling collections in UML and ORM.

16. A.H.M. ter Hofstede and Th.P. van der Weide. Deriving Identity from Extensionality. *International Journal of Software Engineering and Knowledge Engineering*, 8(2):189–221, June 1997.
17. Melvin Fitting. Databases and higher types. In *CL '00: Proceedings of the First International Conference on Computational Logic*, pages 41–52, London, UK, 2000. Springer-Verlag.
18. Bertrand Russell and Alfred North Whitehead. *Principia Mathematica*. Cambridge University Press, 1910–13.
19. L.J. Campbell, T.A. Halpin, and H.A. (Erik) Proper. Conceptual Schemas with Abstractions – Making flat conceptual schemas more comprehensible. *Data & Knowledge Engineering*, 20(1):39–85, 1996.
20. P.N. Creasy and H.A. (Erik) Proper. A Generic Model for 3–Dimensional Conceptual Modelling. *Data & Knowledge Engineering*, 20(2):119–162, 1996.
21. Mustafa Jarrar. Modularization and automatic composition of object-role modeling (ORM) schemes. In *OTM Workshops*, pages 613–625, 2005.
22. C. Maria Keet. Using abstractions to facilitate management of large ORM models and ontologies. In *OTM Workshops*, pages 603–612, 2005.
23. Ralf Hinze. Generics for the masses. In *ICFP '04: Proceedings of the ninth ACM SIGPLAN international conference on Functional programming*, pages 236–243, New York, NY, USA, 2004. ACM Press.
24. M.R.C. Pil. Dynamic types and type dependent functions. In Kevin Hammond, Tony Davie, and Chris Clack, editors, *Implementation of Functional Languages (IFL '98)*, volume 1595 of *LNCS*, pages 169–185. Springer Verlag, 1999.