Chapter 3.2.3
Using Extensible Grammars for Data Modelling

Florian Matthes, Joachim W. Schmidt, and Jens Wahlen

Technical University Hamburg-Harburg
Harburger Schloßstraße 20
D-21071 Hamburg, Germany

Summary We present a systematic approach to the rapid implementation of high-level data models and query languages in the polymorphic higher-order programming language TL. The static semantics of data model constructs are captured by a mapping to polymorphic types and associated data constructor and selector functions. The dynamic semantics of query languages are realised by a mapping to bulk data types and iteration abstractions. Contrary to conventional approaches, these two mappings are specified by user-defined grammar extensions of the target language TL based on user-defined library code and not by separate tools in the programming environment. We give examples of this syntax-directed approach to data modelling and discuss its advantages and limitations.

1. Introduction and Motivation

To address the modelling requirements of advanced data-intensive applications, a large number of high-level data models and query languages have been proposed in the literature to overcome the limitations of existing relational, deductive or object-oriented models.

Unfortunately, the implementation of a new data model or query language for evaluation purposes turns out to be a rather complex task since it requires non-trivial language processing, software engineering and database system construction know-how. Therefore, such data model proposals lead either to ambitious long-term database system development projects (see, e.g., [17] and [16], synopsis in Chapter 2.1.2) with a very long design-implementation-evaluation cycle, or, more frequently, to closed toy implementations that lack the operational support required for experiments in a realistic setting (persistence, interfaces to production systems, user interface support).

Due to their orthogonality and their full integration of persistent data, code and meta-data management, persistent higher-order languages constitute a promising platform for the rapid implementation of high-level data models and their query languages. The static semantics of data model constructs can be captured by a mapping to polymorphic types and associated data constructor and selector functions. The dynamic semantics of query languages can be realised by a mapping to bulk data types and iteration abstractions in the persistent programming language.

The mapping itself can be achieved either by standard source-code to source-code translators (B-Tool [8] for DBPL, Sidereus [5] for Galileo) or by *reflective* code generation techniques [15] (see also Chapter 3.2.2) that exploit the fact that the target language (run-time environment) and the code generator (compile-time environment) are tightly integrated in many persistent systems and that typed persistent program representations can be utilized instead of linear source-code.

In this paper, we report on our experience using a third mapping option that exploits *extensible grammars* as developed in [9, 10] and implemented in the Tycoon system (see Chapter 2.1.4) to translate systematically from declarations and statements of a high-level data model down to type and value expressions of the polymorphic Tycoon language TL (see Chapter 1.1.1).

This paper is organized as follows: We first introduce the concept and notation of extensible grammars by a simple programming language example. In section 3 we give an overview of our syntax-driven approach to data model and query language implementation. We then illustrate in section 4 this approach through a small example (SQL-style queries over multiple bulk types). Section 5 summarizes our experience gained in the Tycoon/Fibonacci add-on experiment, a larger exercise in data model implementation. The paper concludes with a comparison of our approach with generator-based techniques to data-model implementation.

2. Extensible Grammars

In this section we introduce the concept and notation of extensible grammars using a simple programming language example. We also explain how extensible grammars fit into the Tycoon system architecture (see Chapter 2.1.4).

2.1 Dynamic Syntax Extensions

The Tycoon language TL aims at minimality and orthogonality of semantic concepts which is also reflected by its syntax that provides only a small number of keywords and syntactic forms.

However, such a syntactic austerity may lead to stereotypical programming patterns in application code. As a simple example, consider a programming language that provides only a single **loop** ... **exit** ... **end** iteration construct. Other common loop structures (**while**, **repeat**, **for**) have to be expressed by systematic combinations of **loop** and **if** statements.

The idea behind extensible grammars is to provide programmers (or, more realistically speaking, application library designers) with a generic mechanism to introduce tailored syntax for the application domain at hand. The use of tailored syntactic forms often leads to more readable and understandable application code and helps to avoid accidental programming errors in repeating code patterns.

For example, the following dynamic TL syntax extension introduces a block-structured **for** ... **to** ... **do** ... **end** statement by a syntax-directed rewriting into a standard TL **loop** statement.

```
grammar
  value :Value |==
      "for" x=ide "=" l=value "to" h=value "do" b=bindings "end"
      hv=unique
      => value<<| begin
                    let hv = h
                    let var x = l
                    loop if x > hv then exit else b  x:=x+1 end
                  end |>>
end
```

Grammar definitions can be embedded freely into TL source code and therefore have to be enclosed in a **grammar end** block. In the example above, the syntax for TL *value*s is extended (indicated by the operator |==) by a new production that consists of a syntax definition preceding the arrow => and a pattern enclosed in <<| |>> that defines a *value* which is to be generated whenever the new syntax appears in the input program. The pattern utilizes pattern variables (*x, l, h, b, hv*) which are instantiated on expansion of a pattern with concrete identifiers (*x, hv*) or with complete subexpressions (*l, h, b*) as defined by the pattern variable bindings to the left of the arrow =>. For example, the loop

```
let g = 40
for i = 1 to g+7 do a[i]:=g end
```

is rewritten into the TL program

```
let g = 40
begin
  let hv = g+7
  let var i = 1
  loop if i > hv then exit else a[i]:=g  i:=i+1 end
end
```

TL grammar definitions are statically sort-checked. For example, the production *value* is checked to generate expressions of sort *Value* only. Similarly, the productions *ide* and *bindings* are guaranteed to return individual identifiers and sequences of bindings, respectively.

A syntax extension gives also a precise definition of the scoping, typing and evaluation rules for each newly defined language construct in terms of the scoping, typing and evaluation rules of the underlying base language. In the example above, the following rules apply:[1]

- The scope of the loop variable *x* is delimited by the loop body *b*. Inside *b*, a loop variable *x* hides a variable *x* in an outer scope. All other global variables are visible in the subexpressions *l*, *h* and *b*. The binding *hv*=**unique** ensures that on expansion, the identifier *hv* used as an auxiliary variable is chosen to be different from all other variables in the program and therefore remains invisible in all subexpressions.
- The loop variable *x* has to be of type *Int* since it appears as an argument to the integer infix function ">". This implies that the start and end expressions (*l* and *h*) also have to be of type *Int*. Assignments to the loop variable in *b* are allowed.
- The start and end expressions *l* and *h* are evaluated only once, *h* is evaluated before *l*.

Grammar extensions can be layered without restrictions, for example, a vector sum could be defined by a mapping onto a **for** loop.

In addition to the production extension operator |== used in the example above, TL provides two other operators to destructively override production definitions and to introduce new productions avoiding name clashes with existing productions.

As described in [9, 10], TL's extensible grammars are superior to other syntax definition formalisms since they respect static, block-structured scoping and avoid unwanted name clashes, unlike macro definitions and preprocessor-based systems.

[1] An alternative loop semantics can be realised easily by local changes to the syntax definition.

2.2 Initial Syntax Definition

The previous subsection explained how new syntactic forms can be defined based on an existing syntax. To fully decouple the concrete syntax from the TL abstract syntax, the initial TL syntax is not hard-wired into the compiler but provided as a grammar source file. In the current version of our system, this file is compiled once for each persistent store. For example, the syntax of the **loop** and **exit** statements is defined initially as follows using the production definition operator "===":

```
grammar
   value:Value === ...
     | "loop" b=bindings "end" => mkValueLoop(b)
     | "exit" => mkValueExit()
     | ...
end
```

The notation to specify the concrete syntax and pattern-variable bindings on the left-hand side of the productions is identical to the one used for dynamic syntax extensions. The right-hand side consists of constructor applications (*mkValueLoop, mkValueExit*). At compiler definition time, the names and signatures (argument and result sorts) of all TL constructors are made available to the extensible grammar front end. These constructors correspond directly to functions in the TL compiler that create typed program representations (abstract syntax trees).

Despite its rich language model, TL provides language objects of only four different semantic domains which simplifies the orthogonal combination of these objects in the mapping process for a higher-level data model. These semantic domains correspond to the following built-in sorts used in TL grammars:

- The sort *Type* subsumes closed and parameterized type expressions like base types, structured types and higher-order type operators. The canonical type **Ok** is the supertype of all closed types and contains the canonical value **ok**.
- The sort *Value* subsumes expressions and side-effecting statements. A statement returns the canonical value **ok**. The result of an expression used in a statement context is discarded.
- *Bindings* are ordered sequences of identifier/type or identifier/value pairs. If identifiers are omitted, so-called anonymous bindings are defined. This makes it possible, for example, to represent tuple attributes, array elements and statement sequences uniformly as bindings.
- *Signatures* are ordered sequences of identifier/type or identifier/supertype pairs. Again, identifiers can be omitted. Signatures appear, for example, in functions, tuple types and module interfaces.

2.3 On the Implementation of Extensible Grammars

Extensible grammars are realised by a fully self-contained polymorphic Tycoon library implementing an extensible parser and grammar checker which are bound statically as a front-end to the TL type checker and code generator (see figure 2.1). Without recompilation, the very same extensible grammar package can be used for other programming languages, as demonstrated by the TooL language [11].

The grammar checker and parser generator as well as the constructors for TL abstract syntax trees are defined in TL. Input phrases accepted by the extensible parser can be either new syntax definitions which are passed on to the grammar checker and parser generator to be available for future input parsing or they can

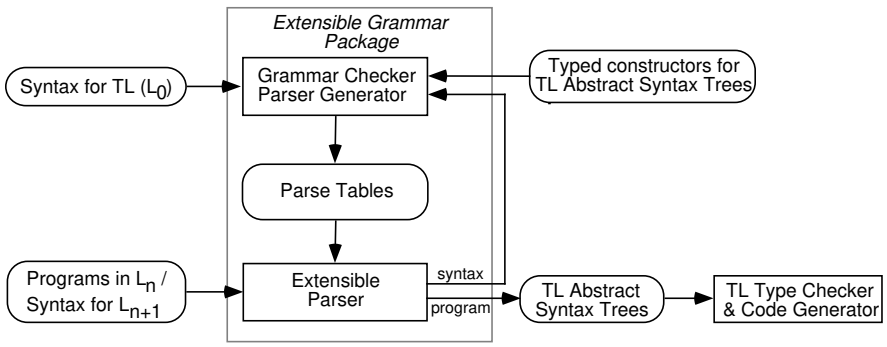

Fig. 2.1. Interfaces of the extensible grammar package

be TL abstract syntax trees that are processed further by the TL type checker and code generator.

The extensible parser is initialized with the syntax of the target language L_0 at hand (TL or TooL). Given an extensible parser for programs in a language L_n, a new extensible parser for a language L_{n+1} can be defined by specifying the context-free grammar and the rewrite rules (productions) that map L_{n+1} terms into L_n terms. This method is incremental since L_n is translated into L_{n-1} until the base target language is reached and the final (TL or TooL) abstract syntax tree is generated.

Static checks at grammar-definition time guarantee the sort-correctness of grammars (only legal syntax trees are generated during parsing), the termination of parsing for arbitrary input programs, and the absence of syntactic ambiguities. Formal proofs for these properties are given in [10]. This high degree of static correctness should be contrasted with other program transformation or reflective compilation techniques.

Extensible grammars can be implemented efficiently. In our persistent system environment, the first extensible TL parser is half as fast as the old, hand-coded standard LL(1) top-down parser. We expect the next re-implementaion of the extensible parser to outperform a handcoded parser using a simple optimization of the attribute evaluation strategy. Furthermore, the space required for the code and data of both parsers is of similar size.

3. Supporting Data Modelling with Extensible Grammars

Figure 3.1 sketches the generic architecture of a data model implementation in TL: Users interact with the system through data-model-specific tools like schema and data browsers. Data and schema declarations, and interactive and embedded queries are written in a data-model-specific syntax. This syntax is specified once per data model by a set of TL syntax modules (bulk syntax, query syntax, etc.). Each syntactic construct is mapped to functions and types exported from Tycoon interfaces. This mapping captures the static and dynamic semantics of the data model. The data-model-specific TL interfaces in turn are based either on the existing set of TL libraries or directly on the built-in TL constructs.

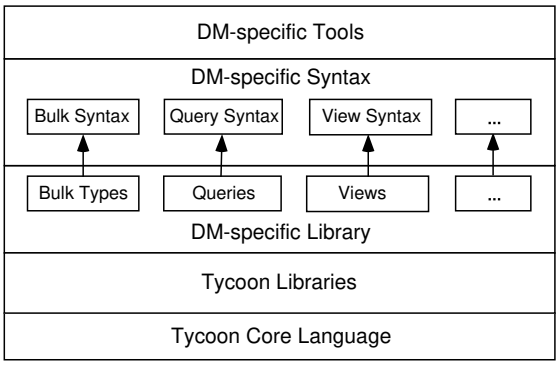

Fig. 3.1. A layered architecture for data model implementation in Tycoon

The top-down methodology to implement a given data model M in TL can be summarized as follows:

1. Identify the semantic concepts of M (e.g., table, row, attribute, domain, query, view).
2. Map each concept to a newly defined or pre-existing abstract TL type or type constructor. At this stage, the implementation of the abstract types is irrelevant.
3. Identify the abstract operations available on each of the semantic objects (e.g., create a table, insert a row, execute a query, define a view, union two tables, join two tables, ...).
4. Map these abstract operations to typed TL function signatures that refer to the types identified in step 2. In order to correctly capture the static semantic constraints on the abstract operations of M by TL type constraints, elaborate TL type concepts (parametric polymorphism, subtype polymorphism, ad-hoc reflective polymorphism based on dynamic type inspection) can be used. The abstract types and their operations are preferably clustered into TL interfaces named after their semantic concept.
5. In parallel, develop a TL grammar definition (incrementally) that maps each syntactic construct of M (e.g. attribute type definition, table type definition, query, subquery, projection list, ...) into nested TL expressions that utilize the types, type operators and functions defined in the previous step. An important task of the grammar is to correctly capture M's scoping rules.
   The name of a grammar production should be chosen based on the data model concept it represents. As a positive side-effect of this naming convention, later syntax error messages generated for non-well-formed data models will utilize this name and convey additional information to the data modeller (e.g., *error in tableDefinition*).
   The initial grammar should de-emphasize syntactic details of M (like precedence rules or alternative notations). Such detail can be added easily at a later stage. The sorts for the grammar productions (*Value*, *Type*, ...) follow immediately from the TL mapping chosen.
6. Verify the consistency of the model developed so far by compiling the grammar and by translating some examples using empty stub implementations for the TL functions defined in step 4. That is, the static semantics of M (scoping and typing rules) can be checked in isolation.

7. Choose appropriate implementations for the abstract data types and polymorphic functions introduced in steps 2 and 4 to implement the dynamic semantics of *M* by TL program code. The implementation typically makes heavy use of polymorphic data structures (lists, sets, bags, dictionaries, ...) and iteration abstractions (select, map, join, ...) provided by the TL libraries.
8. Provide tools for schema browsing, data visualization, code management, data import and export, etc. Again, these tools can frequently be derived schematically from the existing, strongly-typed TL tools.

This top-down approach can also be complemented by a bottom-up composition of existing modules and grammar extensions to increase code reuse and to speed up the data model mapping process.

4. Uniform Iteration Abstraction over Bulk Types

The Tycoon libraries implement multiple bulk data types (sets, lists, relations, etc.). Each bulk data type is represented by a module that exports an abstract type operator and a set of polymorphic functions that work on bulk values of that type. Additional bulk types can be added to the Tycoon library as needed (add-on vs. built-in approach [12]; synopsis in Chapter 1.4.2). Some of these bulk types (like SQL tables) are implemented by gateways to external servers. Each bulk type has to satisfy a *bulk type algebra* (similar to [7]) to create and inspect bulk type instances in a uniform manner.

Uniform declarative access to multiple bulk data types (selection, projection, mapping, join, aggregate functions, ...) is provided by the concept of abstract iterators implemented by a separate Tycoon library module *iter*. An iterator over a homogeneous collection of values of type *E* has type *iter.T(E)*. It can be inspected and manipulated by a large number of iteration abstractions such as *iter.select*, *iter.flatten* and *iter.map* exported from the module *iter*.

Set-oriented query access to bulk data values is achieved in TL as follows. First, the bulk data is (conceptually) transformed into the common iterator type *iter.T*. Then, iterator functions are applied to this iterator which can also combine an iterator with other iterators derived from other bulk data collections. The result can then be converted explicitly into a specific bulk data structure (set, list, bag, ...).

For example, the following iterator expression joins an iteration over *Person* values (tuples with a *name* and *cityZip* attribute) with an iteration over *City* values (tuples with a *zip* and *name* attribute) based on their *cityZip* attributes returning an iteration *residences* over person name and city name pairs:

```
let residences = iter.flatten(iter.map(persons
  fun(p :Person) iter.get(
      fun(c :City) tuple p.name c.name end
      cities
      fun(c :City) p.cityZip == c.zip)))
```

The function *iter.flatten* produces a single iteration by concatenating an iteration of iterations. The function *iter.map* applies its function argument to each element of the iteration supplied as its first argument and returns the results as a new iteration. The function *iter.get* provides combined selection and projection on an iteration. The first parameter is a projection function applied to each element (*select*) of the iteration supplied as the second parameter (*from*) restricted by the predicate of the

third parameter (*where*). The infix function "==" is a polymorphic test on object identity.

Clearly, it is desirable to support a more readable syntax for these declarative iterator queries. In the following, we describe how to define a SQL-style **select from where** syntax for the generalized Tycoon iteration abstractions. Other syntactic forms like *bulk comprehensions* [18] can be defined in a similar fashion. The above query can then be written more succinctly as:

```
let residences= select p.name c.name
                from persons p:Person, cities c:City
                where p.cityZip == c.zip
```

Based on the functions provided by the Tycoon *iter* module sketched above, the following small grammar extension suffices to add this SQL syntax to iterator expressions:

```
grammar
   value:Value |==
       "select" b=bindings "from"  => range(b)
   range(b:Bnds):Value ===
       v=value i=ide ":" t=type  => rangeOrPredicate(b v i t)
   rangeOrPredicate(b:Bnds v:Value i:Binder t:Type):Value ===
       "," r=range(b)   =>
              value<<| iter.flatten(iter.map(v fun(i :t) r)) |>>
   |   "where" p=value  =>
              value<<| iter.get(fun(i :t) tuple b end  v  fun(i :t) p) |>>
end
```

The expanded *value* production parses the target list (a list of bindings following the **select** keyword) and passes these bindings as parameters to the recursively defined productions *range* and *rangeOrPredicate*. A *range* specification defines an iterator *v*, a range variable *i* with local scope, and a type *t*. This information is again passed as a parameter to the production *rangeOrPredicate*. If there is just a single range iteration, the second alternative of the production *rangeOrPredicate* (starting with the keyword **where**) matches and a call to *iter.get* is generated. The first argument of this function is generated based on the structure of the target list *b*, the name of the range variable *i*, and the iteration element type *t*. The second argument is the range *v* and the third argument is the selection predicate *p*, again in the scope of the user-defined range variable *i*.

For each additional range iteration, the first alternative of the production *rangeOrPredicate* matches, and an enclosing *iter.flatten* call is generated which accumulates the results of nested iterations defined by a recursive invocation of *range*.

The reader is encouraged to verify that this syntax definition does in fact capture the scoping and typing constraints of the usual **select from where** syntax.

5. Tycoon/Fibonacci Add-On Experiment

The goal of the Tycoon/Fibonacci add-on experiment is the evaluation of the Tycoon system as an implementation platform for advanced data models. The experiment aims at a comparison of the effort required for the implementation of (a significant subset of) the Fibonacci language [2, 4] (see Chapter 1.1.2) using extensible TL grammars and Tycoon's persistent system infrastructure with the effort

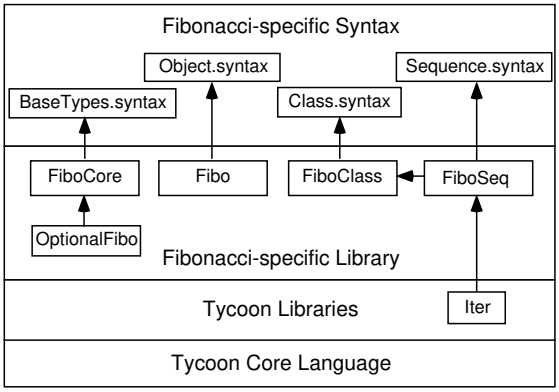

Fig. 5.1. Implementing the Fibonacci data model in Tycoon

required to implement Fibonacci from scratch using Modula-3 at Pisa University [3].

The Fibonacci TL implementation re-used Tycoon's persistent stores, code generator, interactive top level, module manager, incremental linker, and its gateways to commercial systems without change by a rather straightforward mapping of Fibonacci data and code objects onto corresponding TL persistent language objects. This fact nicely demonstrates that one of Tycoon's system design goals, the full separation of data storage, data manipulation and data presentation aspects from data modelling features has been achieved.

Another goal of the experiment is to fully capture Fibonacci's strong static type system (including objects with multiple roles [1]) by a *statically typed* mapping of data and code objects. This way, the full functionality of the Fibonacci type checker is emulated by the existing Tycoon TL type checker. In the remainder of this section we give some insight into this translation scheme at the value and type level. The development of this scheme revealed limitations of the existing TL language constructs available for the manipulation of signature lists and binding lists. The current TL primitives **Repeat** and **open** [13] (see Chapter 1.1.1) are not flexible enough to emulate the specific multiple inheritance conflict resolution strategy of Fibonacci. This deficiency could be solved by introducing a rather small (but ad-hoc) TL type checker extension matching the specific Fibonacci inheritance semantics. This extension would require approximately one week's work for a programmer who is familiar with the non-trivial TL type checker implementation. The Tycoon/Fibonacci experiment required a total of six person months by a scientist visiting our group who had no previous programming experience in persistent languages.

5.1 Module Structure and Overview

Figure 5.1 gives an overview of the implementation of the Fibonacci data model which was developed following the approach sketched in section 3 and involves five Tycoon interface modules and four associated grammar modules. The interfaces are implemented by modules written in statically-typed Tycoon code. A small set of record operations is implemented in type-unsafe C and is only available within type-safe programming patterns specified by the syntax-directed Fibonacci to TL

mapping. C code could be avoided completely by using dynamically-typed TL code. However, this would lead to "unnecessary" additional run-time type tests (which never fail).

Fibonacci Source Code	**TL Equivalent**
Let Address = [name:**String** country:**String**]	**Let** Address = fiboCore.T(**Tuple** name:fiboCore.StringT country:fiboCore.StringT **end**)
let address :Address = ["Bill" unknown]	**let** address :Address = fiboCore.new(**tuple** fiboCore.new("Bill") fiboCore.noneT.unknown **end**)
Let Object = **NewObject**	**let** Object = **tuple** **Let** T = **Record** _object:fibo.Object **end** **let** to(...):**Ok** = ... **let** copy(..):T = ... **let** lateImplementationSetting(...):**Ok** = ... **let** roleExistenceTest(...):**Ok** = ... **end**
Let Person = **IsA** Object **With** name :**String** **End**	**let** Person = **tuple** **Let** T = **Record** **Repeat** Object.T name:fibo.Dispatcher(fiboCore.StringT) **end** **let** propagateToSub(...):**Ok** = ... **end**
let john = **role** Person **methods** name = "John Major" **end**	**let** john = **begin** **let rec** _object:fibo.Object = **record** **let** _oid = fibo.newOid() **let** Person = me **end** **and** me :Person = **record** **let** _object = _object **let** name = fibo.newDispatcher("name" me **fun**()fiboCore.new("John Major")) **end** **end**
john.name	john.name.late()
john!name	john.name.strict()

Table 5.1. Translation scheme from Fibonacci to TL

Table 5.1 gives some examples of the systematic translation from Fibonacci to TL. Some details of the translation are described below. The implementation of Fibonacci class extents, sequences and relationships (modules *FiboClass* and *FiboSeq*) is based on the Tycoon bulk libraries and utilises syntax definitions (*Class.syntax* and *Sequence.syntax*) similar to the ones described in section 4 to map Fibonacci's query language to Tycoon iterator expressions. Value-based integrity constraints on class extents were not studied in the experiment but could be realised by a mapping to an existing Tycoon add-on library (*dbenv* [14]) that provides the required base services.

5.2 Mapping of Concrete Types

The interface *FiboCore* exports the Fibonacci base types and some predefined values like booleans and functions on the base types (comparisons, arithmetics, etc.). This interface relies on the services of the interface *OptionalFibo* to ensure that the value space of each Fibonacci base type and structured type also contains the special value **unknown** (a generic null value). This value is handled explicitly by the predefined functions, for example, by raising a run-time exception *fiboCore.error* that can be caught by the programmer. The type operator *fiboCore.T(A)* defines a representation for a Fibonacci type *A* by a mapping onto a TL union type with a variant *unknown* and a variant *known* with a value *val* of type *A*. The polymorphic functions *new* and *value* construct or inspect values of type *fiboCore.T(A)*, respectively.

```
interface FiboCore export
  Let T(A <:Ok) = Tuple case unknown case known with val:A end
  error :Exception
  new(A <:Ok a:A) :T(A)
  value(A <:Ok optional:T(A)) :A  (* may raise "error" *)
  Let NoneT = Tuple case unknown end
  Let AnyT = T(Ok)
  Let StringT = T(String)
  ...
end
```

By using the Tycoon type *NoneT* to represent the Fibonacci type **None** that only contains the value **unknown** and by mapping the type **Any** (the supertype of all Fibonacci types) to type *AnyT*, the subtype hierarchy of Fibonacci is fully mapped onto the TL subtype hierarchy. In particular, this mapping preserves the subsumption principle, i.e., a value of type *A* can be substituted freely in any context where a value of a supertype of *A* is expected.

Using the interface *FiboCore*, the syntactic mapping of Fibonacci base types and type expressions (functions, aggregates, unions) to Tycoon type expressions as defined in the module *BaseTypes.syntax* is rather straightforward:

```
grammar
  fiboType:Type ===
    x=simpleId => type<<|x|>>
  | "Null" => type<<|fiboCore.NullT|>>
  | "Any" => type<<|fiboCore.AnyT|>>
  | "String" => type<<|fiboCore.StringT|>>
  | ...
  | "Fun" "(" s=fiboSigs ")" ":" t=fiboType => type<<|fiboCore.T(Fun(s):t)|>>
  | "[" s=fiboSigs "]" => type<<|fiboCore.T(Tuple s end)|>>
  | "Choice" c=fiboCases "End" => type<<|fiboCore.T(Tuple c end)|>>
end
```

The first two rows of table 5.1 give an example of the translation of a Fibonacci type and a matching value expression into TL code.

5.3 Mapping of Object and Role Types

A key feature of Fibonacci is the support for statically-typed objects with multiple roles [2, 1]. Any interaction with a Fibonacci object takes place by sending messages

to a specific *role* of an object. The set of roles possessed by an object can change over time so that its behavior changes over time too. Here we do not want to go into the intricate details of the Fibonacci object model but only give a rough idea of its mapping to typed TL object representations.

A Fibonacci object is represented by a dynamically-constructed directed acyclic graph of roles that share a common object representation. Each role is a static Tycoon record that aggregates a set of named *dispatchers*, one for each message understood by a role. A Fibonacci role type is described by a Tycoon record type (a set of message signatures) and enables the static type checking of message send operations. A message dispatcher for a Fibonacci message (or attribute) of type *A* is a Tycoon tuple of type *fibo.Dispatcher(A)*:

```
Let Dispatcher(A<:Ok) = Tuple
   methodName :String
   self : Record end
   strict():A
   var late():A
end
```

The two functional components *strict* and *late* of a dispatcher realize the two forms of message dispatching supported by Fibonacci. A late binding as in *john.name* leads to the execution of the most specialized implementation of the method *name* (which depends on the run-time history of role acquisitions for the object *john*). A strict binding as in *john!name* executes the method specified statically for the current role. A dispatcher is created by a call to *fibo.newDispatcher* which provides an initial method implementation for *strict* and *late*. The function *late* is tagged as mutable with the **var** keyword since it is overridden dynamically whenever a matching dispatcher is created in a subrole. The dispatcher attributes *methodName* and *self* are utilised by the Tycoon library code that has to work uniformly on dispatchers of arbitrary roles.

The last two lines in table 5.1 show how the two forms of Fibonacci message send operations are translated into TL function calls. Once a systematic encoding of objects, roles and methods by a graph of linked TL records and function closures has been found, the translation of role type and role value expressions can be carried out in a type-safe way as indicated by examples in table 5.1.

An interesting detail is the fact that a Fibonacci role *type Person* is mapped to a Tycoon *value Person* that aggregates a type specification *Person.T* and a set of functions to work on instances of that type (*Person.copy*, *Person.roleExistenceTest*, ...). The aggregation of types and values into a first-class language object is a good example of the power available to data model implementors through Tycoon's higher-order type system.

The following fragments of the grammar definitions in the module *Object.syntax* (see figure 5.1) show that the mechanism of extensible grammars is sufficiently expressive to carry out the mappings indicated in table 5.1 and that these mappings can be written in a rather readable way.

```
grammar
   fiboValue:Value ===
        v=fiboSimpleValue  => optFiboValueApp(v)
   optFiboValueApp(v:Value):Value ===
        "!" method=ide  => value<<|v.method.strict()|>>
   |    "." method=ide  => value<<|v.method.late()|>>
   |    (* no suffix *) => v
end
```

```
grammar
  fiboObjectValue:Value ===
      "role" roleName=ide "methods" mbnds=fiboObjectMethodBnds "end"
      => value<<| begin
                    let rec _object :fibo.Object = record
                        let _oid = fibo.newOid()
                        let roleName = me
                      end
                    and me :roleName.T = record
                        let _object = _object
                        mbnds
                    end
                    me
                  end |>>
  | ...
end
```

Note that this rewriting captures the Fibonacci scoping rules, for example, the identifier *me* is bound to the "current" object in the scope of the method bindings *mbnds* by virtue of a recursive object binding in Tycoon. Similarly, the names of record attributes that should not be visible to Fibonacci programmers are made inaccessible by prefixing them with an underscore that is disallowed in the source syntax.

6. Related Work and Concluding Remarks

The work described in this paper can be seen as a contribution towards the "type alchemist's dream" [6], to express multiple data models uniformly in a sufficiently expressive type system.

The rapid implementation of a data model in the typed and persistent Tycoon framework has the following advantages compared with a system prototype implemented from scratch:

- The constructed system inherits most of Tycoon's system functionality (persistence management, garbage collection, code generation, modular compilation etc.);
- No compiler implementation know-how is needed to implement type checkers for polymorphic (higher-order) data models;
- The methodology described in section 3 supports early experimentation with newly developed data models.

Compared with source-to-source code generators and preprocessor-based systems, extensible grammars have the following advantages:

- Extensible grammars are fully integrated in the Tycoon compiler (intermediate source files are avoided, error messages refer to the original source code position, code management is simplified);
- Extensible grammars are easy to read and modify;
- Extensible grammars provide mechanisms to avoid name clashes and to invent fresh identifiers as needed.

Extensible grammars are best suited for *context-independent* local rewriting techniques. Some data model implementation techniques, e.g. the one described in [8], make heavy use of global schema information held in a central repository to drive

the generation of integrity-preserving transaction code. This is not possible with the mechanism of extensible grammars since the rewrite process can only depend on the input syntax and not on schema information available at compile-time. However, as exemplified by the Tycoon/Fibonacci add-on implementation, the handling of context dependencies can often be delayed until run-time where it is possible to work on persistent meta-data generated at compile-time.

Acknowledgement This research was supported by ESPRIT Basic Research, Project FIDE, #6309 and by a grant from the German Israeli Foundation for Research and Development (*bulk data classification*, I-183 060). Davide Berveglieri (Politecnico di Milano) contributed significantly to the Tycoon/Fibonacci experiment described in section 5. He was supported by the ESPRIT Basic Research Network of Excellence IDOMENEUS (No. #6606) and the ESPRIT Basic Research Project FIDE #6309.

References

1. A. Albano, G. Bergamini, G. Ghelli, and R. Orsini. An object data model with roles. In *Proceedings of the Nineteenth International Conference on Very Large Databases, Dublin, Ireland*, pages 39–51, 1993.
2. A. Albano, R. Bergamini, G. Ghelli, and R. Orsini. An introduction to the database programming language Fibonacci. FIDE Technical Report Series FIDE/92/64, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, 1993.
3. A. Albano, C. Brasini, M. Diotallevi, G. Ghelli, R. Orsini, and R. Rossi. A guided tour of the Fibonacci system. FIDE Technical Report Series FIDE/94/103, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, July 1994.
4. A. Albano, G. Ghelli, and R. Orsini. Fibonacci reference manual: A preliminary version. FIDE Technical Report Series FIDE/94/102, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, 1994.
5. L. Alfo, S. Coluccini, P. Corte, and D. Presenza. Manuale del sistema sidereus. Technical report, Dipartimento di Informatica, Università di Pisa, Italy, 1989.
6. M.P. Atkinson and P. Bunemann. Types and persistence in database programming languages. *ACM Computing Surveys*, 19(2), June 1987.
7. C. Beeri and P. Ta-Shma. Bulk data types, a theoretical approach. In C. Beeri, A. Ohori, and D.E. Shasha, editors, *Proceedings of the Fourth International Workshop on Database Programming Languages, Manhatten, New York*, Workshops in Computing. Springer-Verlag, February 1994.
8. A. Borgida, J. Mylopoulos, J.W. Schmidt, and I. Wetzel. Support for data-intensive applications: Conceptual design and software development. In *Proceedings of the Second International Workshop on Database Programming Languages, Portland, Oregon*, June 1989.
9. L. Cardelli, F. Matthes, and M. Abadi. Extensible grammars for language specialization. In C. Beeri, A. Ohori, and D.E. Shasha, editors, *Proceedings of the Fourth International Workshop on Database Programming Languages, Manhatten, New York*, Workshops in Computing, pages 11–31. Springer-Verlag, February 1994.

10. L. Cardelli, F. Matthes, and M. Abadi. Extensible syntax with lexical scoping. Technical Report 121, Digital Equipment Corporation, Systems Research Center, Palo Alto, California, February 1994.
11. A. Gawecki and F. Matthes. Tool: A persistent language integrating subtyping, matching and type quantification. FIDE Technical Report Series FIDE/95/135, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, 1995.
12. F. Matthes and J.W. Schmidt. Bulk types: Built-in or add-on? In *Database Programming Languages: Bulk Types and Persistent Data*. Morgan Kaufmann Publishers, September 1991.
13. F. Matthes and J.W. Schmidt. Definition of the Tycoon language TL – a preliminary report. Informatik Fachbericht FBI-HH-B-160/92, Fachbereich Informatik, Universität Hamburg, Germany, November 1992.
14. C. Niederée. Generic services for data-intensive applications: Iteration abstraction, integrity checking and recovery. Master's thesis, Fachbereich Informatik, Universität Hamburg, Germany, November 1992. (In German).
15. K.-D. Schewe, J.W. Schmidt, and I. Wetzel. Identification, genericity and consistency in object-oriented databases. In J. Biskup and R. Hull, editors, *Proceedings of the International Conference on Database Theory*, volume 646 of *Lecture Notes in Computer Science*, pages 341–356. Springer-Verlag, October 1992.
16. J.W. Schmidt and F. Matthes. The DBPL project: Advances in modular database programming. *Information Systems*, 19(2):121–140, 1994.
17. M. Stonebraker. Special issue on database prototype systems. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1990.
18. P. Trinder. Comprehensions, a query notation for DBPLs. In *Database Programming Languages: Bulk Types and Persistent Data*. Morgan Kaufmann Publishers, September 1991.