Appeared in: P.P.Spies, editor, *Proceedings of Euro-Arch'93 Congress*, pages 301-317,

Springer Verlag, October 1993

System Construction in the Tycoon[1] Environment: Architectures, Interfaces and Gateways

Florian Matthes Joachim W. Schmidt

Universität Hamburg
Vogt-Kölln Straße 30
D-22527 Hamburg, Germany
{matthes,J_Schmidt}@dbis1.informatik.uni-hamburg.de

Abstract

This paper outlines the basic concepts and the system architecture of the Tycoon[1] environment. Tycoon is designed for the construction of persistent object systems intended to be available on multiple software and hardware platforms.

Tycoon's contribution to system portability and scalability is achieved by strictly separating concerns of data modeling, data manipulation, and data storage into three distinct system layers each of which is based on state-of-the-art system technology (polymorphic programming languages, portable code representations, persistent object stores). Much emphasis is being placed on supporting interoperability between Tycoon applications and off-the-shelf tools and systems, such as database systems, user interface managers, and optimizing code generators.

We illustrate how higher-order functions, polymorphic typing and transparent persistence management reduce the amount of repetitive and type-unsafe programming in typical persistent object systems.

1 Introduction and Overview

In this paper we present the concept of *application frameworks in persistent higher-order languages* as a systematic approach to carry forward the conceptual and technological achievements of fourth-generation languages (4GLs) into open, heterogeneous environments.

In a fist step (section 2), we summarize the relative merits of third-generation languages and fourth-generation languages for the development of information systems. Based on this analysis we explain how persistent higher-order languages equipped with well-designed polymorphic libraries make it possible to blend the advantages of 3GLs and 4GLs in a conceptually simple and linguistically integrated system environment.

[1]Tycoon: Typed Communicating Objects in Open Environments.

This research is supported by ESPRIT Basic Research, Project FIDE, #6309.
This paper appeared in: P.P. Spies (Ed.): *Proceedings Euro-ARCH'93*, Informatik aktuell, pages 301–317, Springer-Verlag, 1993.

In the following sections we present Tycoon, a particular example of a persistent higher-order language that has been developed at the University of Hamburg [MS92, Mat93]. The Tycoon project draws heavily from practical experience gained in implementing successive generations of relational database programming languages [Sch77, KMP+83, SM92a, SMV93] and from recent research results of the Esprit Basic Research Project FIDE (Fully Integrated Data Environments).

Section 3 sketches Tycoon's characteristic language concepts that are required for the definition of type-safe, high-level application frameworks. Tycoon drastically reduces the amount of repetitive and type-unsafe programming in typical information systems by factoring-out much of the application's functionality into generic, extensible server libraries.

Section 4 outlines the overall Tycoon system architecture that achieves a high degree of portability and scalability by strictly separating concerns of data modeling, data manipulation, and data storage into three distinct system layers each of which is based on state-of-the-art system technology (polymorphic programming languages, portable code representations, persistent object stores).

Finally, section 5 gives an idea of the organization and functionality of the Tycoon libraries for bulk data management. We distinguish between internal library implementations (written entirely in Tycoon) and external library implementations (imported from external servers like SQL databases, graphical user interfaces, or operating system libraries). Due to the richness of Tycoon's type system it is possible to provide application programmers with a uniform view on both kinds of libraries.

2 Approaches to Persistent Object System Construction

We use the term *persistent object systems* (POS) to denote a class of software systems that give their users a flexible, problem-oriented and safe access to large sets of long-lived objects of various types [SM93, Mat93].

Due to technological and commercial developments there is an ever-increasing demand for persistent object systems that handle new object types like texts (electronic mail, text retrieval, "intelligent" text manipulation), two- and three-dimensional graphics (CAD and geographic information systems), digitized raster images (picture and font finder, fax manager, character recognition software), voice data, or even short, compressed image sequences (presentation software, scene analysis). Furthermore, persistent object systems have to be implemented in substantially different (PC-based, interactive, networked, distributed, semi-autonomous) system environments [Bla90, Cat91, BM91].

In addition to the classical problems of large-scale application development, developers of a POS are faced with the following characteristic tasks (see, e.g., [ZM89, ABW+90, SRL+90]):

Persistent Storage: The primary task in a persistent object system is the manipulation of the state of long-lived data objects shared by multiple, possibly concurrently executing applications. In many persistent object systems it is advantageous to be able to store, share and modify also program objects ("methods", "scripts") as first-class data objects.

Generic Programming: Persistent object systems are characterized by highly repetitive algorithmic and structural patterns (set-oriented queries, integrity constraints,

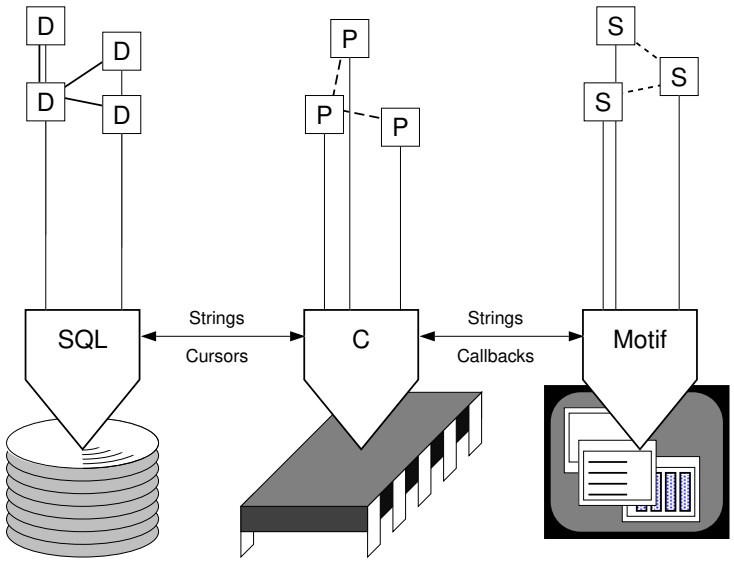

Figure 1: Narrow server interfaces in 3GLs

generic data structures, etc.). Therefore, many commercial tools utilize generator techniques to derive automatically stereotyped program fragments (form definitions, format conversions, sorting routines, iterators, etc.) from parameterized, high-level descriptions.

External Server Integration: The buzzword "open architectures" is of particular relevance for persistent object systems since it is often necessary to interact with external devices (screens, printers, keyboards, pointing devices, scanners, CD-ROMS, modems, fax machines, etc.) and external servers (window libraries, RPC libraries, directory services, etc.) for data acquisition, data presentation and data transfer.

In the following, we first sketch how these requirements are met partially by third- and fourth-generation languages, respectively, and we then develop the idea of application frameworks in persistent higher-order languages which can be understood as a synthesis of the 3GL and 4GL approach.

2.1 Low-Level Server Access from 3GLs

The majority of today's information systems is implemented in third-generation languages (COBOL, FORTRAN, C, Pascal). A severe limitation of these languages is their inability to capture correctly the generity inherent even in simple persistent object models. For example, the relational data model supports the definition of unary, binary, ternary, ... relations over various domains and it includes generic operations over relation variables. These operations cannot be captured adequately by any of the commercial programming languages mentioned above [BHR82].

As a consequence of this conceptual limitation, developers of database programming interfaces like Embedded SQL [Ing90] restrict themselves to the least common denominator of all 3GLs and handle any information interchange via strings, addresses and untyped byte arrays that are transferred at the smallest possible granularity (attribute-wise) between application programs and the database systems. A similar situation can be observed at the interface between programming languages and (graphical) user interfaces.

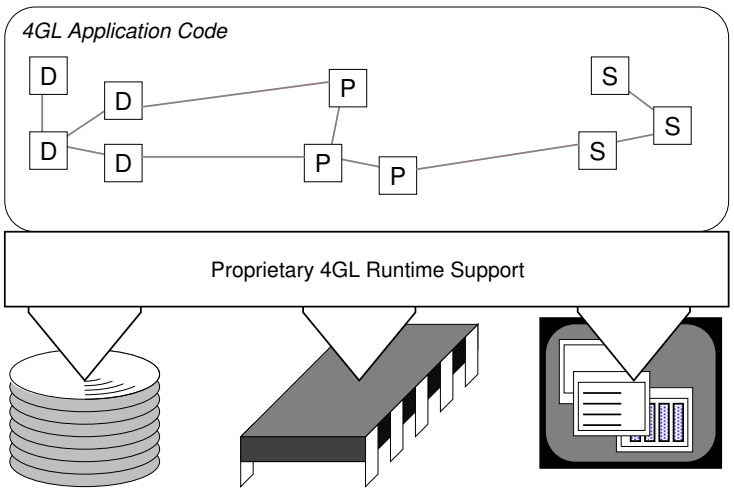

Figure 2: Server integration in proprietary 4GLs

The resulting POS scenario using 3GLs is depicted in Fig. 1: The overall functionality of a particular POS is provided by a coordinated manipulation of numerous heterogeneous objects on different media (D: database objects, P: program objects, S: screen objects, etc.). Objects on different servers are subject to server-specific naming, typing, lifetime, and binding rules (indicated by different line styles in Fig. 1) while inter-server communication takes place at a low level of abstraction using simple, machine-oriented protocols.

A crucial advantage of such a loose server coupling is the *openness* of the application development environment. If required, new servers can be integrated freely as libraries into existing persistent object systems. Furthermore, call interfaces provide a minimal common ground for data exchange between independently developed services and they also constitute a starting point for data abstraction and component exchangeability.

2.2 Server Integration in 4GLs

Fourth-generation languages recognize the need for a uniform, safe and application-oriented development environment across a variety of servers and above the minimal interoperation level of strings and cursors. As depicted in Fig. 2, a well-designed 4GL provides high-level naming, typing and binding mechanisms for objects on different media (on disk, in memory, and on screen) and thereby eliminates a substantial amount of the low-level, repetitive data conversion and data transfer code found in 3GL applications. 4GLs are tailored to the specific requirements of traditional, commercial database applications:

Functional requirements: record- and set-oriented data modeling, data manipulation, data storage and data display; generic operations (search, iterate, insert, delete, update, format, display, validate, ...) applicable uniformly to record and set objects of different component structure.

Operational requirements: persistent storage of data, application code, integrity constraints and user interface definitions; efficient manipulation of data on secondary storage; integration into transactional multi-user systems.

Modal requirements: interactive user-interfaces, access to precompiled queries and update transactions; support for incremental system evolution (e.g. schema changes); provision for *ad-hoc* queries.

Examples of relational 4GLs include Ingres-4GL [Ing89], PL-SQL [Ora91], and Informix-4GL [Inf86]. Furthermore, several object-oriented database systems are shipped with 4GL languages to simplify application development (e.g., O_2SQL and CO_2 for O_2 [BDK92]). Practical experience shows that 4GL constitute a significant advance towards achieving faster software development, improved long-term software maintainability, better software portability, and even enhanced execution efficiency for specialized applications (e.g., queries) when compared with hand-coded third-generation language solutions like C or COBOL [MJ89].

There are, however, some severe limitations to current 4GL languages that hamper their success in today's open, continually evolving system environments:

- 4GLs are deeply integrated into proprietary DBMSs and it is therefore impossible to move 4GL applications from one DBMS to another. This situation is in sharp contrast to 3GL applications using standardized programming and database languages (e.g., C and OpenSQL).
- It is typically very difficult to interface 4GLs with external servers (window systems, statistical packages, text-retrieval tools) that do not come bundled with the DBMS.
- Applications developed in 4GLs do not scale well due to a lack of appropriate typing and modularization support as found, for example, in Ada, C++, Modula-2 or Eiffel.
- On the system level, 4GL often incur an unnecessary system overhead since it is impossible to "downscale" the extensive 4GL runtime support (form management, event management, query support, report generation, transaction management, ...) to just the functionality needed by a particular application.
- The linguistic quality of 4GLs does not compare well with modern (object-oriented, modular, or functional) programming languages.

2.3 Scalable Application Frameworks in Persistent Higher-Order Languages

Most of the above 4GL limitations reflect the deficiencies of the implementation technology that is being used to construct 4GL interfaces. This technology simply does not support the abstractions required for a more flexible, systematic 4GL service integration. To overcome these limitations and to achieve 4GL openness and extensibility, we propose an architecture based on interacting POSs using well-organized safe libraries and an implementation which exploits higher-order language technology. According to our understanding, an ideal POS development environment involves the following components:

1. A persistent higher-order programming language that provides just the core functional, operational and modal primitives required by persistent object systems (see Sec. 3);

2. A lean system architecture that unbundles and repackages the services found in traditional programming languages, database systems, and operating systems to allow scalable system implementations ranging from simple, PC-based, single-user systems to networked, multi-user, client-server architectures (see Sec. 4).

3. A systematically designed application framework that consists of a rich and extensible set of libraries. The framework supplies application builders with predefined, composable, high-level services like bulk data structures (relations, sets, lists, graphs, etc.), iteration abstractions (queries, views, traversal patterns for recursive data structures, etc.), and declarative integrity checking mechanisms (see Sec. 5);

The success of such a development environment crucially depends (1) on the expressiveness and orthogonality of the base language, (2) on the degree of interoperability with pre-existing services and tools that can be achieved by the system architecture, and (3) on the quality of the initial set of libraries provided by the application framework.

Fig. 3 sketches the architecture of a POS implemented in an application framework provided by a persistent higher-order language. Similar to a 4GL, the relevant application objects (persistent database objects, volatile program objects, virtual screen objects, etc.) are subject to uniform naming, typing and binding rules, indicated by uniform line styles connecting the corresponding symbols in Fig. 3. In contrast to a 4GL, most of these object types are not built into the base language but are provided by an open set of libraries that are an integral part of the programming environment. Some of the libraries are simply interfaces to external services (database systems, language compilers, GUI toolkits), others libraries provide functionality that is implemented in the persistent higher-order language itself. Furthermore, Fig. 3 indicates that access to some of the libraries may be supported by syntax extensions to the persistent core language. The technology for such flexible, problem-oriented syntax extensions using extensible grammars is described in [CMA93].

In the rest of this paper we present the contributions of the Tycoon system, a particular example of a persistent higher-order application framework, to the three areas outlined above.

3 Language Concepts for High-Quality Information Systems

In this section we highlight some of Tycoon's higher-order language concepts. By treating functions and types as first-class language objects, it becomes possible to write generic libraries and generators without leaving Tycoon's language framework. This fact is exploited extensively in the Tycoon libraries and tools (see Sec. 5) to provide 4GL functionality without resorting to inflexible, monolithic built-in solutions.

3.1 Values, Types, Bindings and Signatures

The semantic model of Tycoon is based on higher-order type theories [ML75, CH85, Car88]. The core semantic entities of Tycoon are values, types, bindings and signatures [BL84, Car89]. Values and types can be named in bindings for identification purposes and to introduce shared or recursive structures at the value and the type level. Signatures act as (partial) specifications of static and dynamic bindings. Bindings are embedded into the syntax of values, i.e. they can be named, passed as parameters, etc. Accordingly,

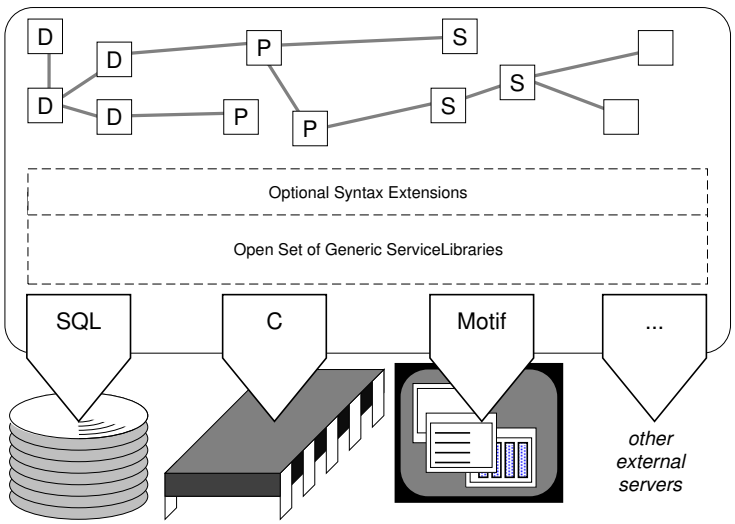

Figure 3: Scalable application frameworks in persistent higher-order languages

signatures appear in the syntax of types to describe these aggregated bindings. The mutual dependencies are illustrated best by the following excerpt from Tycoon's recursive language syntax. Further examples and explanations are given below.

Type::= *Ide* | ***Ok*** | ***Nok*** | ***Fun***(*Signatures*):*Type* | ***Tuple*** *Signatures* ***end***
| *Type*(*Bindings*) | ***Oper***(*Signatures*)*Type*
Signatures::= {*TypeSignature* | *ValueSignature* | *LocationSignature*}
TypeSignature::= *Ide* <:*Type*
ValueSignature::= *Ide* :*Type*
LocationSignature::= ***var*** *Name* :*Type*
Value::= *Literal* | *Ide* | ***ok*** | ***fun***(*Signatures*)*Value* | ***tuple*** *Bindings* ***end*** | *Value*(*Bindings*)
Bindings::= {*TypeBinding* | *ValueBinding* | *LocationBinding*}
TypeBinding::= ***Let*** *Ide*=*Type* | :*Type*
ValueBinding::= ***let*** *Ide*=*Value* | *Value*
LocationBinding::= ***let*** ***var*** *Ide*=*Value* | ***var*** *Value*

Types are partial specifications of values. Here are some examples of types (compare with the type syntax above):

Int, Real, String	library-defined type names
Age, IntPair	user-defined type names
Ok, **Nok**	top type and bottom type
Fun(*x :Int*) *:Int*	function type
Tuple *fst :Int snd :Int* ***end***	aggregate type

There is no syntactic distinction between library-defined base type names and user-defined type names. The predefined type **Ok** represents the weakest type specification. The predefined type **Nok** represents the strongest (non-satisfiable) type specification. In Tycoon there is a partial order on types based on their degree of precision: $A <: B$ ("A is subtype of B") iff type A is more precise than type B. The order $<:$ is defined inductively and Tycoon's type system contains the subsumption rule: if a value a has type A and $A <: B$ then a has also the weaker supertype B. For any type A, $\mathbf{Nok} <: A <: \mathbf{Ok}$ holds.

Function types are used to describe the signatures of parameterized objects like functions, procedures, methods, generators or relational queries. Aggregate types are used to describe the signatures of language entities like records, tuples, structures, modules, object definitions, or database definitions,.

For each type constructor there exist corresponding value constructors, for example:

3, 3.4, "XY"	literal values
ok, **raise** *abort*	top value and exception generator
true, false, pi	library-defined value identifiers
*+, –, *, sin, abs*	library-defined function identifiers
***fun**(x :Int) x+1*	user-defined function value
tuple let** fst=3 **let** snd=4 **end	aggregate value

The builtin value **ok** has type **Ok**. This value is returned by expressions that are not intended to produce a useful value and that only perform side-effects like assignments. There is no value that has type **Nok**. Type **Nok** is assigned by the compiler to expressions whose evaluation does not terminate, like statement sequences that raise an exception.

Bindings are ordered sequences of value, type and location bindings, for example:

***let** pi = 3.1415*	value binding
***let** succ = **fun**(x:Int) x+1*	function value binding
Let** IntPair=**Tuple** fst:Int snd:Int **end	type binding
***let var** age = 20*	location binding

A type binding defines a name for a type. A value binding defines a name for a (constant) value. A location binding defines a name for an anonymous location that in turn contains a value. Value, type and location bindings in Tycoon are immutable, i.e., it is not possible to destructively update bindings once they are established. Assignments can be performed only on the contents of locations (*age:=age+1*).

It is sometimes desirable to omit identifiers from bindings and to write *anonymous* bindings, like *succ(3)* or ***tuple** 3 4 **end*** instead of *succ(**let** x=3)* and ***tuple let** fst=3 **let** snd=4 **end***, respectively. There are type rules that govern the compatibility between anonymous bindings and named signatures in Tycoon.

Signatures classify bindings like types classify values. Signatures are ordered sequences of value, location or type signatures, for example:

pi :Real	value signature
*succ :**Fun**(x:Int) :Int*	function value signature
*IntPair <:**Tuple** fst:Int snd :Int **end***	type signature
***var** age :Int*	location signature

Signatures specify invariants on bindings and allow the verification of the correctness of (value or type) expressions depending on names without having access to the actual binding in which the name is defined. For example, based on the signature ***var** age :Int*, the type correctness of the expression *age:=age+1* can be verified without having access to the actual value bound to *age*. Signatures therefore play a central role in type-safe application frameworks that require separate compilation.

3.2 Higher-Order Functions

In Tycoon, functions are first-class typed values. The following examples illustrate three powerful proramming patterns that are based on the use of first-class functions. It should be noted that today's 3GLs and 4GLs impose restrictions on the use of functions that essentially rule out these programming patterns for all "interesting" cases, e.g., if functions contain references to non-local variables, if functions are to be stored persistently, or if functions are to be passed between programs running on different hardware architectures.

Function aggregation allows programmers to build complex data structures ("objects") that contain function values. A particular form of function aggregation underlies the modular and object-oriented programming style where aggregated functions are used to perform operations on encapsulated state variables:

let *object* = ***tuple let*** *print* = ***fun***() ***begin***...***end*** ***let*** *copy* = ***fun***() ***begin***...***end end***

The selection of function components in aggregates is achieved via the usual dot notation (*object.print(), object.copy()*).

Function parameterization allows programmers to pass functions dynamically as arguments to other functions. For example, the higher-order function *twice* takes a function argument and an integer argument and applies its fist argument twice to its second argument:

let *twice* = ***fun***(*f* :***Fun***(x:*Int*):*Int* a:*Int*) *f(f(a))*
twice(succ 3) ⇒ *5*

The capability to pass functions as arguments to other functions is exploited heavily in the Tycoon framework to factor-out application-dependent functionality from otherwise re-usable library code.

Finally, *function generation* allows programmers to return functions as the result of other functions. For example, the function *makeInc* returns a function that adds a fixed value to its argument. The increment step is determined by an argument to the function *makeInc*:

let *makeInc* = ***fun***(*step* :*Int*) ***fun***(x :*Int*) x + *step*
let *inc1* = *makeInc(1)* ***let*** *inc10* = *makeInc(10)*
inc1(3) ⇒ *4* *inc10(3)* ⇒ *13*

Technically speaking, each invocation of *makeInc* returns a fixed piece of code (*x+step*) with a different static environment that assigns values, types and locations to the global variables (*step* in this example).

3.3 Higher-Order Type Operators

Type operators denote parameterized type expressions that map types or type operators to types or type operators. For example, the type operator *Pair* takes any type *X* that is a subtype of the trivial type **Ok** and returns a tuple type with two fields of type *X*:

Let *Pair*=***Oper***(*X* <:***Ok***) ***Tuple*** *fst:X snd:X* ***end***	type operator binding
Pair(Int), Pair(String), Pair(Pair(Int))	type operator applications

Many programming languages have built-in type operators that map types to types (*Array*, *List*, *File*, *Pointer*, ...); some languages have support for user-defined type operators that map types to types (e.g., type definitions in ML [MTH90] and Haskell [HW86]); very few languages support higher-order type operators (Quest [Car89], Tycoon [MS92]).

The ability to introduce new type operators is required to supply generic data structures like relations, indices, stacks in the application framework. These generic types can later be instantiated with type parameters to construct application-specific types like "supplier relations", "student indices" or "integer stacks".

The syntax for type operator definitions and type operator applications in Tycoon emphasizes the analogy between the concept of (higher-order) functions, mapping (function) values to (function) values, and the concept of (higher-order) type operators. In particular, it is possible to perform type operator aggregation, type operator parameterization, and type operator generation in analogy to the functional concepts outlined in the previous section.

An example of a higher-order type operator is *Twice* which takes a type operator F and returns the result of applying F twice to a given type A:

Let *Twice=**Oper**(F <:**Oper**(X<:**Ok**)<:**Ok** A<:**Ok**) F(F(A))*
Twice(Pair Int)

For example, *Twice(Pair Int)* is equivalent to *Pair(Pair(Int))*. More useful examples of higher-order type operators require substantially more complex type expressions as they appear in interface specifications of large libraries.

3.4 Type Quantification in Signatures

The parameterization concepts on the value and type level outlined in the previous two secions are complemented in Tycoon by the concept of type quantification in signatures. Type identifiers introduced in a type signature can appear in subsequent type, value and location signatures. This makes it possible to express two kinds of dependencies on type variables[1]:

Existential Quantification: Type components embedded in the signature of an aggregate type allow to model type abstraction as found in modular programming languages:

Let *ADT =* ***Tuple*** *T <:**Ok** zero :T succ(:T):T* ***end***
let *nat =**tuple** **Let** T=Int* ***let*** *zero=0* ***let*** *succ=**fun**(x :Int) x+1* ***end***
let *one = nat.succ(nat.zero)*

Intuitively, the existential quantification inside of an aggregate type gives extra flexibility to the service provider, in this case the implementor of the value *nat* of type *ADT*, since he can choose the representation type for T freely, as long as the constraints of the other signatures are met:

let *nat =**tuple** **Let** T=String* ***let*** *zero="" **let** succ=**fun**(x :String) concat(x "1")* ***end***

Universal Quantification: Type components embedded in the signature of a function type lead to the well-known concept of (bounded) parametric polymorphism [CW85]:

[1] To a limited extent it is also possible to define type dependencies on value identifiers (*dependent types* [Mac86]).

```
Let Sort = Fun(X <:Ok  x :Array(X)  {<}:Fun(a:X b:X):Ok):Ok
let quickSort = fun(X <:Ok  x :Array(X)  {<}:Fun(a:X b:X):Ok) begin...end
quickSort(:Int  IntArray  {<})
quickSort(:Person  PersonArray  fun(a,b:Person) string.<(a.name b.name))
```

Sort describes the type of generic sort functions that work uniformly for all types X and that take an array x with elements of type X and a function $<$ to compare two elements of type X.

Intuitively, the universal quantification inherent in generic function types gives extra flexibility to the service consumer, in this case the caller of the function *quickSort*, since he can apply the function to a wider range of arguments.

Another example of a universally quantified type expression is the type of the assignment function ":=":

```
Let Assign = Fun(X <:Ok  var lhs :X  rhs :X):Ok
```

This signature expresses the fact that the assignment operation works uniformly over all subtypes X of **Ok** and takes a location of type X and a value of type X as its left-hand-side and right-hand-side argument. Since the type of ":=" is fully representable in the Tycoon type system, it is also possible to export alternative implementations of the assignment function in the Tycoon libraries, e.g. to perform concurency control, integrity checking or recovery operations in addition to the plain assignment operation.

4 The Scalable Tycoon System Architecture

It should be clear from the language overview in the previous section that Tycoon does not attempt to achieve expressiveness by an extensive list of featuers but by generalization and orthogonal combination of relatively few basic concepts. This minimalistic approach also simplifies the implementation of the supporting Tycoon system architecture that is sketched in this section.

The complexity of relational database systems (and recent OODBMS) results essentially from the fact that they are monolithic servers that bundle a large number of tightly coupled services like memory management, concurrency control, recovery, data structuring, access control, etc. Moreover, access to these services is only granted via a narrow database language interface that severely restricts access to individual services.

The Tycoon system attempts to unbundle the above DBMS and 4GL services by strictly separating data modeling, data manipulation and data storage issues. Horizontal bars in Fig. 4 indicate the three central Tycoon system abstractions: TL, TML and TSP.

TL (Tycoon Language) is the strongly-typed higher-order polymorphic programming language presented in Sec. 3 [MS92]. TL serves as a uniform application and system programming language and therefore has to support both, programming using high-level problem-specific data models (like entities and relationships) as well as the implementation of these data models in terms of system-oriented data structures (like hash tables, B-trees, or linear lists).

Large TL programs are typically divided into modules, interfaces and hierachically nested libraries. The Tycoon language processors support separate compilation and dynamic linking. Furthermore, it is possible to evaluate TL expressions interactively, e.g., to perform *ad-hoc* queries. Scanning, parsing, type checking, code generation, linking and

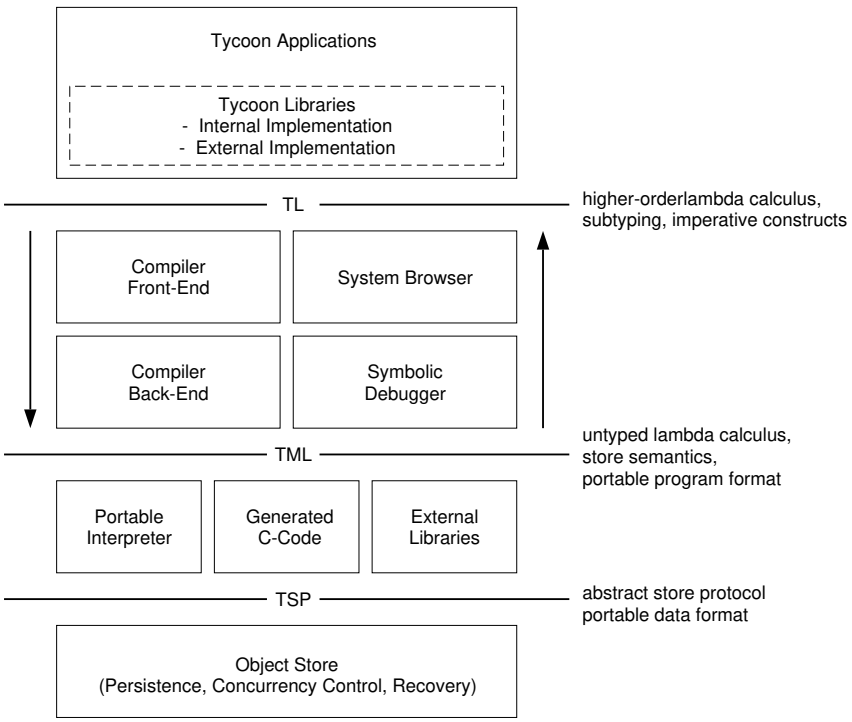

Figure 4: The Tycoon system architecture

execution is then performed immediately, giving the user the illusion of interpretative execution. Since expressions entered interactively may refer to precompiled TL modules, this interactive interface constitutes also a convenient programming development environment.

TML (Tycoon Machine Language) is a minimal intermediate language based on an untyped lambda calculus extended with imperative constructs that serves as a low-level, portable TL program representation in distributed heterogeneous environments. TML was designed to support efficient host-specific target code generation as well as dynamic optimizations analogous to query and transaction rewriting in database systems.

The ability to inspect abstract program representations at run-time, to generate new executable code from program representations, and to link newly created code to "live" systems is a key technology for generic system construction. This functionality has to be emulated in third- and fourth-generation languages either by resorting to interpretation techniques or to non-portable, ad-hoc binding mechanisms.

Currently, there exist two different TML evaluators: a compact, portable interpreter, immediately executable on a wide range of hardware-platforms and an optimizing code generator. The latter uses proprietary C compilers as portable target machine code generators. The current version of the Tycoon system runs on Sun Sparc, DEC Mips, and IBM Power architectures. We intend to make Tycoon also available on IBM PC and Apple Macintosh systems.

TSP (Tycoon Store Protocol) is a data-model-independent object store protocol based on the notion of a *persistent heap* that shields TML evaluators (and TL programmers) from operational aspects of the underlying persistent store like access optimization, storage reclamation, concurrency or recovery. By forcing all higher levels of the system to use the TSP (software) protocol, it provides an ideal starting point to add system functionality

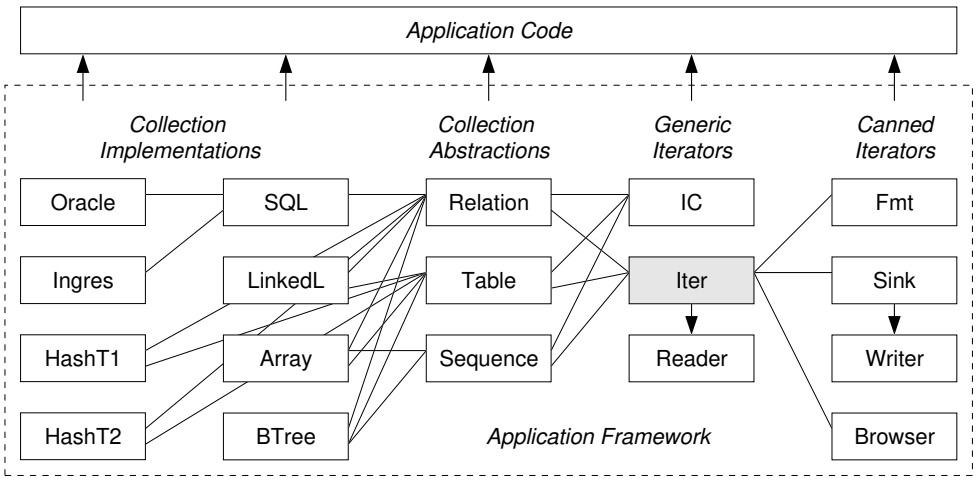

Figure 5: Components of the Tycoon Collection Type Library

at the store-level (e.g. distribution transparency or access-control).

A key contribution of the TSP to the overall Tycoon system functionality is support for *orthogonal persistence* [AB87]: data of any type (including functions) can exist as long or as short as required by the application. Programmers do not need to write explicit code to move data between persistent and volatile store.

Currently, there exist three different TSP implementations that can be combined freely with the existing TML evaluators: a compact, paged main memory store implementation with a simple file-based persistence mechanism [MMS92]; an interface to the Napier persistent object store [BMM+91] that provides an efficient single-user stable store with recovery mechanisms based on shadow copying exploiting low-level memory management hardware support; and a gateway to the object-oriented database system ObjectStore [LLOW92] supporting recoverable multi-user access to shared databases in client-server architectures. All store implementations feature automatic garbage collection and portable data import and export to files.

To summarize, this layered protocol-oriented architecture aims at system *scalability* and *portability* by de-emphasizing operational aspects of the protocol implementations and focussing on the core requirements of persistent object systems.

5 An Application Framework for Bulk Data Management

As explained in Sec. 2.3, application programmers do not use the bare Tycoon language TL but implement substantial parts of the overall application functionality by instantiating generic library code supplied by an application framework. In this section we give a brief overview of an application framework that is being developed at the University of Hamburg to meet the specific demands of data-intensive applications. One should note that there may be multiple application frameworks supported by the same core language and system architecture. For example, [SM92b] describes a Tycoon application framework for the construction of language processors, exporting scanner, parser and unparser generators as well as utility modules for source code, error log and cross-reference management.

Fig. 5 gives an idea of the organization of the main components of the Tycoon ap-

plication framework for bulk data management which provides services at several levels of abstraction. For simple applications that only need to print, display and possibly edit (persistent) bulk data, the modules *Browser, Fmt, Sink* and *Writer* provide pre-packaged program patterns that bundle the necessary data retrieval and data formatting code.

Other modules (*Iter, IC, Reader*) export query constructs (like select, project, join, aggregate, sort, or group by), loop statements and predicative integrity control mechanisms. These modules utilize higher-order functions, polymorphic functions and user-defined type operators to make their functionality applicable uniformly to a wide range of bulk data structures. These query constructs are fully integrated into the Tycoon language TL, i.e., they can be nested, named, parameterized, or stored persistently.

The declarative and set-oriented services of the modules *Iter, IC* and *rReader* are based on abstract collection type definitions (*Relation, Table, Sequence*). An abstract collection is represented as an object with a hidden state and a set of functions (size, member, insert, delete, get, set, ...) that modify or inspect the hidden state.

For each abstract collection type (e.g., *Relation*) there exist alternative collection type implementations. Some of these implementations are written in TL (e.g., *LinkedList, BTree, Array, HashTable*), others are imported from external servers (e.g. *SQL* Tables). All of these implementations are polymorphic, that is, they can be instantiated with different element types. For example, the following (simplified) interface *SQL* taken from the current Tycoon libraries describes the signatures of the polymorphic functions exported by the a SQL server:

```
interface SQL export
  Table <:Oper(E<:Tuple end)<:Ok
  openTable:Fun(Dyn E<:Ok name:String):Table(E)
  insert:Fun(E<:Ok table:Table(E) tuple:E):Ok
  insertAll:Fun(E<:Ok table:Table(E) tuples:Table(E)):Ok
  delete:Fun(E<:Ok table:Table(E) where:Fun(e:E):Bool):Ok
  ...
end
```

This interface exports an abstract type operator *Table* that maps a tuple type E (any subtype of the empty tuple type) to a hidden type (an arbitrary subtype of the type **Ok**). This parameterized hidden type describes the type of SQL tables with elements of type E. The type operator *Table* is used in subsequent function signatures of this interface, capturing the fact that the only way to manipulate SQL tables is via these generic functions. There may be several modules implementing this interface, i.e. defining type, value and location bindings that match the specified signatures. Currently, there exist a Tycoon/Ingres and a Tycoon/Oracle gateway that both have the interface *SQL*.

The following interface imports the type operator *Table* from the module *ingres* with interface *SQL* to define a application-specific database schema that declares two relation variables (*register, fees*).

```
interface PhoneDB import ingres export
  Let Entry=Tuple name:String num:String end
  Let Fee = Tuple prefix :String cost :Int end
  var register :ingres.Table(Entry)
  var fees :ingres.Table(Fee)
  end
```

The binding to the external Ingres relation variables is established in the corresponding implementation module:

```
module phoneDB import ingres export
  let var register=ingres.openTable("register")
  let var fees=ingres.openTable("fees")
end
```

A main program may use the bindings provided by the phone database as arguments to the SQL interface functions:

```
module main import ingres phoneDB export
  ingres.insert(phoneDB.register tuple "Peter" "249" end)
  ingres.delete(phoneDB.fees fun(f:Fee) f.prefix>="800")
end
```

This example illustrates how higher-order type systems contribute to lean languages and open system architectures. All the code necessary to implement the SQL gateway is encapsulated in a library that has to be imported only by those applications that actually need this service. Furthermore, the "type rules" which ensure that clients make proper use of this service are also encapsulated by the library interface signatures and do not need to be hard-wired into the language or its runtime system.

6 Concluding Remarks

Application frameworks in persistent higher-order languages provide the functionality of 4GLs for a much broader class of object types than the classical set- and record-oriented data structures. By a strict separation of data modeling, data manipulation and data storage issues into three distinct system layers, this generalization on the language level can be achieved in a relatively small, scalable and portable system environment with substantially improved interoperability.

Our current work in the Tycoon project aims at integrating further relevant generic servers (e.g., authentication, directory and RPC servers) into the Tycoon POS environment and to lift their services using state-of-the-art language technology.

References

[AB87] M.P. Atkinson and P. Bunemann. Types and Persistence in Database Programming Languages. *ACM Computing Surveys*, 19(2), June 1987.

[ABW+90] M. Atkinson, F. Bançilhon, D. De Witt, K. Dittrich, D. Maier, and S. Zdonik. The Object-Oriented Database System Manifesto. In *Deductive and Object-oriented Databases*. Elsevier Science Publishers, Amsterdam, Netherlands, 1990.

[BDK92] F. Bancilhon, C. Delobel, and P. Kanellakis. *Building an Object-Oriented Database System: The Story of O_2*. Morgan Kaufmann Publishers, 1992.

[BHR82] P. Bunemann, J. Hirschberg, and D. Root. A Codasyl Interface to Pascal and Ada. In *Proc. 2nd British National Conference on Databases (BNCOD 2)*. Cambridge University Press, 1982.

[BL84] R. Burstall and B. Lampson. A kernel language for abstract data types and modules. In *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*. Springer-Verlag, 1984.

[Bla90] A. Blaser, editor. *Database Systems of the 90s*, volume 466 of *Lecture Notes in Computer Science*, November 1990.

[BM91] R. Balzer and J. Mylopoulos. International Workshop on the Develpment of Intelligent Information Systems. Technical report, University of Southern California and University of Toronto, April 1991.

[BMM+91] A.L. Brown, G. Mainetto, F. Matthes, R. Müller, and D.J. McNally. An Open System Architecture for a Persistent Object Store. Persistent Programming Research Report CS/91/9, Univ. of St. Andrews, Dept. of Comp. Science, September 1991.

[Car88] L. Cardelli. Structural Subtyping and the Notion of Power Type. In *Proceedings of the Fifteenth ACM Symposium on Principles of Programming Languages*, 1988.

[Car89] L. Cardelli. Typeful Programming. Report 45, DEC Systems Research Center, 130 Lytton Avenue, Palo Alto, 94301 CA, May 1989.

[Cat91] R.G.G. Cattell. Next-Generation Database Systems. *Communications of the ACM*, 34(10), October 1991.

[CH85] T. Coquand and G. Huet. Constructions: a higher order proof system for mechanizing mathematics. Technical Report 401, INRIA, Domaine de Voluceau Rocquencourt 78153 Le Chesnay Cedex - France, May 1985.

[CMA93] L. Cardelli, F. Matthes, and M. Abadi. Extensible Grammars for Language Specialization. In *Proceedings of the Fourth Workshop on Database Programming Languages*, Manhatten, NY, 1993. Springer-Verlag. To appear.

[CW85] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[HW86] P. Hudak and P. Wadler. Report on the Programming Language Haskell Version 1.2. *SCM SIGPLAN Notices*, 21(7):219–233, July 1986.

[Inf86] Informix Software Corp. Informix-4GL Reference Manual. Technical report, Informix Software Corp., 1986.

[Ing89] Ingres Corporation. INGRES ABF/4GL Reference Manual for the UNIX and VMS Operating Systems. Technical Report INGRES Release 6.3, Ingres Corporation, 1080 Marina Village Parkway, Almeda, CA 94501, November 1989.

[Ing90] Ingres Corporation. INGRES Embedded SQL Companion Guide for C. Technical Report INGRES UNIX Release 6.3, Ingres Corporation, 1080 Marina Village Parkway, Almeda, CA 94501, December 1990.

[KMP+83] J. Koch, M. Mall, P. Putfarken, M. Reimer, J.W. Schmidt, and C.A. Zehnder. Modula/R Report, Lilith Version. Technical report, Department Informatik, ETH Zürich, Switzerland, February 1983.

[LLOW92] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore Database System. *Communications of the ACM*, 34(10):50–64, 1992.

[Mac86] D.B. MacQueen. Using dependent types to express modular structure. In *Conf. Record 13th Ann. Symp. Principles of Programming Languages*, pages 277–26. ACM, January 1986.

[Mat93] F. Matthes. *Persistente Objektsysteme: Integrierte Datenbankentwicklung und Programmerstellung*. Springer-Verlag, 1993. (In German).

[MJ89] V.M. Matos and P.J. Jalics. An Experimental Analysis of the Performance of 4GL Tools on PCs. *Communications of the ACM*, 32(11):1340–1352, 1989.

[ML75] P. Martin-Löf. An intuitionistic theory of types: predicative part. In H.E. Rose and J.C. Sheperdson, editors, *Logic Colloquium 1973*, pages 73–118, Amsterdam, 1975. North Holland Publishing Company.

[MMS92] F. Matthes, R. Müller, and J.W. Schmidt. Object Stores as Servers in Persistent Programming Environments – The P-Quest Experience. FIDE Technical Report Series TR/92/48, Fachbereich Informatik, Universität Hamburg, Germany, July 1992.

[MS92] 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.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.

[Ora91] Oracle Corporation. PL/SQL User's Guide and Reference, Version 1.0. Technical Report Part No. 800-V1.0, Oracle Corporation, June 1991.

[Sch77] J.W. Schmidt. Some High Level Language Constructs for Data of Type Relation. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Toronto, Canada*, August 1977.

[SM92a] J.W. Schmidt and F. Matthes. The Database Programming Language DBPL: Rationale and Report. FIDE Technical Report Series FIDE/92/46, Fachbereich Informatik, Universität Hamburg, Germany, July 1992.

[SM92b] G. Schröder and F. Matthes. Using the Tycoon Compiler Toolkit. DBIS Tycoon Report 061-92, Fachbereich Informatik, Universität Hamburg, Germany, May 1992.

[SM93] J.W. Schmidt and F. Matthes. Lean Languages and Models: Towards an Interoperable Kernel for Persistent Object Systems. In *Proceedings of the IEEE International Workshop on Research Issues in Data Engineering, Interoperability in Multidatabase Systems*, pages 2–16, Vienna, Austria, April 1993.

[SMV93] J.W. Schmidt, F. Matthes, and P. Valduriez. Building Persistent Application Systems in Fully Integrated Data Environments: Modularization, Abstraction and Interoperability. In P.P. Spies, editor, *Proceedings Euro-Arch'93*, 1993.

[SRL+90] M. Stonebraker, L.A. Rowe, B. Lindsay, J. Gray, M. Carey, M. Brodie, and P. Bernstein. Third-Generation Data Base System Manifesto. *ACM SIGMOD Record*, 19(3):31–44, September 1990.

[ZM89] S.B. Zdonik and D. Maier. *Readings in Object Oriented Database Management Systems*. Morgan Kaufmann Publishers, 1989.