To appear in "Proceedings International IEEE/RIDE Workshop on Interoperability, Vienna, April 1993

Lean Languages and Models: Towards an Interoperable Kernel for Persistent Object Systems [*]

Joachim W. Schmidt
Universität Hamburg, DBIS
Vogt-Kölln Straße 30
D-2000 Hamburg 54

Florian Matthes
Current Address: DEC SRC
130 Lytton Avenue
Palo Alto, CA 94301

{J_Schmidt, matthes}@dbis1.informatik.uni-hamburg.de

Abstract

Reliable interoperation between independently developed systems frequently requires *type-safe* access to *persistent* data objects and *generic* services while today's system architectures and interoperation tools still focus primarily on store-level access to volatile data and simple monomorphic or untyped services.

In this paper, we summarize our experience gained in a long-term project that provides persistence abstractions and generic database support in a strongly typed database environment which includes optimizing gateways to commercial relational database servers and module-based distributed programming tools. To keep the presentation focussed, we make use of a uniform language model based on higher-order polymorphic types to capture the essential interoperation semantics including classical cross-language calling mechanisms, remote procedure call models as well as relational and object-based database gateways.

This uniform language model is also the conceptual core of the *Tycoon*[1] database environment being developed at Hamburg University. Tycoon lifts Persistent Object System interoperability to a higher level of genericity and precision while further reducing overall system complexity by a *lean approach* to languages and models for data, execution and storage.

Since it is central to the concept of *lean production* to substantially reduce the *manufacturing penetration* by importing and reusing external services, interoperability is crucial to our approach.

[*]This research was supported by ESPRIT Basic Research (FIDE).
[1]Typed Communicating Objects in Open Environments

1 Introduction

Persistent Object Systems (POS) are integrated software artifacts that provide flexible and efficient access to and adequate operations on large and long-lived (sets of) data objects. Operations on simple data objects include searching, sorting, arithmetic, textual input and printing. More complex data objects (bit images, vector graphics, text documents, sounds, ...) require special-purpose operations for data retrieval, data presentation and data manipulation.

Persistent Object Systems have to address characteristic interoperation needs such as:

Persistent Object Management It is necessary to identify, store, retrieve and manipulate objects that outlive a single program execution and that may even exist independently of the application that created them. Appropriate naming and scoping mechanisms are required to establish temporary or persistent bindings between persistent objects created independently, possibly using different tools on different machines.

Data Integrity Type systems represent a particularly successful approach to enforce integrity constraints on data objects local to a single program. Similarly, language-independent mechanisms have to be provided for shared, persistent objects.

Generic Functions Many functions required in a Persistent Object System can be obtained by instantiating *generic* services. Sorting, searching, query processing, form management or report generation are typical examples of tasks that are accomplished best by instantiating generic algorithms with type-specific arguments. Interoperation protocols have to be capable of handling these generic functions without diminishing data integrity or duplicating code.

In this paper we seek to contribute to the answers of three interrelated questions: What are appropriate semantic models to describe POS *interaction*? How can these semantics be captured in a sound and concise linguistic framework? What is the impact of such lean languages and models on their supporting system architectures and their import and export interfaces?

The paper is organized as follows. In the first part we present our experience gained in using the database programming language DBPL for the construction of Persistent Object Systems. In Section 2 we argue that interoperability in mixed-language, cross-platform and multi-site environments can be substantially improved by making the linguistic framework for naming, scoping, typing, binding etc. uniformly applicable to all computational objects, be they small or large, short-lived or persistent, local or remote. DBPL achieves a high degree of type-safe system interoperability by providing built-in problem-specific language support for POS applications [34, 22, 24]. In Section 3 we report on our experience with importing into DBPL application functionality provided by servers external to DBPL. We concentrate on a gateway between the DBPL system and Ingres/SQL and discuss requirements for generic gateway construction.

In the second part of the paper (Sec. 4), we briefly sketch ongoing research in the Tycoon project [21, 26, 25]. This research aims at lean and language system architectures where Persistent Object Systems achieve essential parts of their functionality by importing external services into a conceptually sound linguistic framework.

Our running example is a miniature application involving simple data objects, namely telephone numbers, fax messages and telephone directories. Incremental program modifications illustrate how to add persistence, distribution, bulk data handling and external services.

2 Improving Inter-Object Interaction at the Instance Level

We present a canonical language-based model of Persistent Object Systems and demonstrate how this model correctly captures modularization, distribution, persistence and bulk data abstractions as provided by the DBPL database programming environment.

2.1 Naming, Typing and Binding for Safe and Flexible Object Interaction

In this section we introduce (informally) a model to describe the naming, typing and binding concepts involved in Persistent Object System interoperability. The model itself (subsequently referred to as the *Tycoon POS model*) is based on concepts of higher-order type systems [20, 10, 5] and is sufficiently expressive to serve as a language-independent framework for program translation, generation and binding. The presentation of the actual DBPL system and its gateways in the following sections makes use of a limited subset of the Tycoon POS model. The potential of the full model will be discussed in Section 4.

The model is not biased towards specific features of object-oriented languages. It is designed to capture equally well interoperability aspects that arise in the interaction with classical imperative languages (Cobol, Fortran, C, Pascal) and (relational, hierarchical, object-oriented) database systems.

It should be noted that the model as presented here is too expressive to serve as a "good" programming language notation. For example, it features dependent types in their full generality that complicate type checking and compilation. However, by introducing level distinctions and annotations to separate static and dynamic types, one can re-establish the static typing property and obtain efficient language implementations (see Section 4).

The Tycoon POS model is based on the notion of *types*, *signatures*, *values* and *bindings*. Types are understood as (partial) specifications of values. 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, signatures appear in the syntax of types to describe these aggregated bindings. The exact mutual dependencies are defined as follows.

The syntax for *types* includes a set of base types B_i (**Int**, **Real**, **String**, ...), a type constant **Any** (the trivial type), user-defined type variables, function types, aggregated signatures, parameterized type expressions (type operator definitions) and type operator applications:

Type::= B_i | **Any** | *TypeName* |
Fun*(Signatures) Type* | **Sig***(Signatures)* |
Oper*(Signatures) Type* | *Type(Bindings)*

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

Type operators denote parameterized type expressions that map types or type operators to types or type operators. 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 [28] and Haskell [15]); very few languages support higher-order type operators (Quest [6], Tycoon [26]).

Our notation for type operator definitions and type operator applications (*Array(E≡***Int***)*, *Stack(E≡Stack(E≡***Int***))*, *Dictionary(Key≡***String** *Value≡* **String***)*) emphasizes the analogy between the concept of (higher-order) functions, mapping (function) values to (function) values, and the concept of (higher-order) type operators.

Signatures are sequences of value, location or type signatures. A value signature associates a value name with a type (*peter :Student*). A location signature associates a location name with a type (**var** *age* :**Int**). A type signature associates a type name with a supertype specification (*Student <:Person*).

Signatures::= {*TypeSig* | *ValueSig* | *LocationSig*}
TypeSig::= TypeName <:Type
ValueSig::= ValueName :Type
LocationSig::= **var** *LocationName :Type*

Aggregated signatures are used to describe named *declarations* as they occur in function headings, module signatures, **external** declarations in C programs or in database schema definitions.

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 *age :Int*, the type-correctness of the expression *age + 1* can be verified without having access to the actual value bound to *age*. Signatures therefore play a central role in type-safe module systems.

In a well-formed signature sequence, types only depend on type variables introduced to their "left", avoiding non-well-founded cyclic supertype specifications:

Person <:**Any** *Student* <:*Person* *peter* :*Student*

The syntax for *values* includes base values $b_{ij} : B_i$, like 0, 3.4, "xyz", **true**, a canonical value of the type **Any**, function values including built-in functions like +, −, * of type **Fun**(*x* :**Int** *y*: **Int**) **Int** and user-defined functions and aggregated bindings:

Value::= b_{ij} | **any** | **fun**(*Signatures*)*Expr* | **bnd**(*Bindings*)

The syntax **bnd**(Bindings) defines that aggregated value and type bindings are first-class values, generalizing classical concepts like value, function and type aggregation in records, structures, abstract data types or modules. The syntax of *expressions* in functions is deliberately not specified here in order to be able to describe POS involving multiple languages. We assume that every language provides read and write access to locations in the store that are reachable through function arguments and declarations in the static scope of a function. Moreover, every language should have primitives for function definition, function application, binding formation and binding element selection in addition to the standard operations on the base types and basic control structures. For simplicity, we treat statements as expressions returning the value **any** :**Any**, possibly producing side-effects on the store.

Bindings are sequences of type, value and location bindings. A type binding *Age*≡**Int** defines a name for a type. A value binding *pi*=*3.1415* defines a name for a value. A location binding *petersAge*=**var**(α_i) *3* defines a name for an anonymous location identified by (α_i) that in turn contains a value (3).

Bindings::= {*TypeBnd* | *ValueBnd* | *LocationBnd*}
TypeBnd::= *TypeName*≡*Type*
ValueBnd::= *ValueName*=*Value*
LocationBnd::= *LocationName*=**var**(α_i) *Value*

Bindings model type, constant, variable and function declarations in programming languages and instances of database schemata in database systems.

Several names may be bound to the same location (*aliasing, sharing*):

bnd(*fred*=**var**(α_1)"*Fred*"
peter=**var**(α_2)**bnd**(*name*="*Peter*" *son*=**var**(α_1)"*Fred*")
mary=**var**(α_3)**bnd**(*name*="*Mary*" *son*=**var**(α_1)"*Fred*"))

To avoid redundancies, location bindings could be written simply as *son* = **var**(α_1), *fred* = **var**(α_1) and a separate *store* could provide the location to value mapping ($\alpha_1 \rightarrow$ "*Fred*"), $\alpha_2 \rightarrow$**bnd**(...), $\alpha_3 \rightarrow$**bnd**(...), However, mixing store and binding information simplifies the presentation of the following examples substantially.

By convention, the name of a location in an expression designates the contents (r-value) of the location. Only in certain contexts (e.g., if passed as a variable parameter in Algol-like languages) does it designate the location itself (l-value) [29]. All imperative languages also provide an assignment operation to update destructively the contents of a location by a new value, possibly affecting multiple location bindings at once (*side effects*).

The expression evaluation rules for a language L could be specified by an *evaluation function* that takes a function value, a set of matching argument bindings and a store returning a result and a modified store.

$Eval_L : Value \times Bindings \times Store \longrightarrow Value \times Store$

As we will see later in more detail, there are two basic mechanisms for interoperation between two languages L_1 and L_2 in Persistent Object Systems. Both mechanisms, dynamic cross-language function calls and static sharing of type, value and location across address spaces, are captured by the Tycoon POS model. These two mechanisms are tightly interrelated since functions take values as arguments and return values. Moreover, functions are treated as first-class values and can therefore be shared and passed on as function arguments and results.

As usual, types are intended to classify values, and signatures are to classify bindings. Moreover it is possible to define *generic* functions (functions that take type bindings as arguments), value-dependent type operators, abstract data types (bindings that contain partially-specified type components and operations on that type). The formal type rules for this model (defining well-formed types, the types of values, the subtype relationship between types and the signatures of bindings) become therefore quite subtle.

In traditional systems, types and signatures are handled exclusively at compile-time, while values and bindings only appear at run-time. In persistent systems, the distinction between compile-time and run-time blurs, and it becomes possible, for example, to inspect value and type bindings at compile-time, giving rise to powerful reflective algorithms [19, 35]. In the following, we present several examples where a generalized treatment of values, types, signatures and bindings can be exploited for improved POS interaction.

2.2 Modularity: The Basis for Interoperability

In this section we introduce the basic modularization concepts of DBPL (inherited from Modula-2 [37]).

DBPL programs are divided into modules with well-defined import relationships. Definition modules define signatures for type, value and location bindings defined in implementation modules. Program modules export a single, parameterless function value, the main program.

An interface and a skeleton DBPL module exporting basic data types and values to handle telephone numbers would look as follows:

```
definition module Phone;
type Number = array [0..20] of char;
var localPrefix, home :Number;
procedure operator(prefix :Number) :Number;
end Phone.

implementation module Phone;
procedure operator(prefix :Number) :Number;
```

begin ... **return** ... **end** *operator;*
begin *(* module initialization: *)*
localPrefix:="4940"; home:=concat(localPrefix,"85312");
end *Phone.*

During type checking of these compilation units, the DBPL compiler extracts type and value signatures that are represented as follows in the Tycoon POS model:

CompileEnv ≡ **Sig***(Phone <:***Sig***(Number <:***array**...
    var *localPrefix :Number* **var** *home :Number*
    *operator :***Fun***(prefix :Number) Number)*
  *linkPhone :***Fun***(imports :***Sig***()) Phone)*

The interface module is represented as a type signature which associates the interface name *Phone* (understood as a type variable) with a supertype that is an aggregate of all types and values signatures exported by the interface. The implementation module is represented as a single function, *linkPhone*, that takes an aggregated binding of all imported module values (in this case an empty binding) and returns a binding that conforms to the interface signature *Phone*.[2]

The signatures of *Phone* and *linkPhone* are both elements of a flat name space modelled by a signature bound to the type variable *CompileEnv*. This name space is managed implicitly via search paths to look up compiled interfaces, called "symbol files", in a typical DBPL system implementation.

The code generated for the implementation module *Phone* is stored as a named binding in a flat name space that collects all compiled modules. This name space is represented in the Tycoon POS model as a named value *linkEnv* whose type matches the signatures specified by *CompileEnv*:

linkEnv = **bnd***(linkPhone=***fun***(imports :***Sig***()) ...)*

This name space is also utilized by the DBPL linker to compose all module initialization functions that make up an executable application program. The main module

module *Main* **import** *Phone;* ... **end** *Main.*

is represented by the following link function:

*linkMain :***Fun***(imports :***Sig***(phone :Phone)) Main*

It leads to the following module initialization sequence:

*initEnv=***bnd***()*
phoneE=initEnv∪{*phone=linkEnv.linkPhone(initEnv)*}
mainE=phoneEnv∪{*main=linkEnv.linkMain(phoneE)*}

[2]As we will see later, we would lose some modelling precision if we were to treat the implementation module as a simple binding of type *Phone*.

The module value returned by *linkPhone* is an aggregate of two bindings to two distinct, newly created locations and binding to a function value.

> **bnd**(*localPrefix*=**var**(α_1)*"4940"*
>   *home*=**var**(α_2)*"4940 85312"*
>   *operator*=**fun**(*prefix :Number*) ...)

2.3 Cross-Platform Interoperability

The distributed version of DBPL [16, 17] exploits the basic module concepts described in the previous sections to add an additional layer of type-safety to standard remote procedure call mechanisms (RPC) in federated client-server programming models.

To give access to a local fax service at a site called "CentralOffice", this site would compile the following *remote definition module* and then *export* the compiled description (typically together with its source text for documentation purposes) only to those clients on the network who are to be authorized to use the service.

```
remote definition module Fax for "CentralOffice";
import Phone;
type Status = (error, busy, done);
procedure dial(number :Phone.Number) :Status;
procedure send(text :array of char) :Status;
procedure receive(var text :array of char) :Status
procedure hangup();
end Fax.
```

In this scenario, signatures of compiled definition modules accumulated in the compilation environment serve as protocol specifications:

```
CompileEnv ≡ Sig(
  Phone <:Sig(...)  (see Sec. 2.2)
  linkPhone :Fun(imports: Sig()) Phone
  Fax<:Sig(Status <:Int
    error :Status, busy :Status, done :Status
    dial :Fun(number :Phone.Number) Status
    send :Fun(text :array...) Status
    receive :Fun(var text :array...) Status
    hangup :Fun() Any )
    linkFax :Fun(imports :Sig(phone :Phone)) Fax )
```

Note that an enumeration type declaration in DBPL introduces a new type name (a subtype of the basic type **int**) and a number of values names of that type. Furthermore, the signature of the *receive* function illustrates that DBPL variable parameters correspond to location signatures in the Tycoon POS model.

Clients also need to have access to the definition module *Phone* that defines the argument type *Phone.Number* for the remote function *dial*. In the distributed DBPL environment, unique identifiers are assigned to compiled remote interface definitions. These identifiers are used during connection establishment to verify that all parts of a distributed program have been compiled using compatible versions of the interface specifications. This mechanism directly generalizes classical link-time consistency control mechanisms in non-distributed systems.

Frequently clients require distribution transparency. In this case it is of considerable advantage if a main program

> **module** *SendFax;* **import** *Phone, Fax;*
> **begin**
> *Fax.dial("853228");Fax.send("Sample fax.");Fax.hangup();*
> **end** *SendFaxFromModula;*

stays textually unchanged whether it uses a local definition module *Fax* or the above remote definition mofule *Fax* for "Central Office" offered by a server somewhere in the network. This distribution transparency is achieved as usual by a client stub and a server stub that marshal and unmarshal the arguments and results supplied to functions defined in the remote definition module [11].

In terms of the Tycoon POS model, RPC-based communication mechanisms are an implementation technology that enables the creation of function value bindings between names in a client program and function values in a server program. Using plain RPC mechanisms it is not possible to directly define location bindings spanning machine boundaries. In particular, we would have to revise the module interface *Phone* not to directly export the locations *localPrefix* and *home*. In section 2.4 we show how these traditional RPC restrictions are lifted by the provision of (distributed) persistent variables in DBPL.

The crucial feature of DBPL is to retain (static) type safety across machine boundaries by maintaining a distributed compilation environment that allows local and remote modules to *share* signatures for type checking purposes and to share module bindings for transparent connection establishment. For example, in the system for the main program *SendFax* the type variable *Fax* is shared between the fax client stub, fax servers stub, the actual fax implementation and the main program:

> *DistributedCompileEnv*≡**Sig***(*
>   *Phone <:***Sig***(...) (see Sec. 2.2)*
>   *Fax <:***Sig***(...)*
>   *linkSendFax :***Fun***(imports :***Sig***(fax :Fax)) Main*
>   *linkFaxClientStub :***Fun***(serverId :ServerId) Fax*
>   *linkFaxServer :***Fun***(imports :*
>   **Sig***(phone :Phone fax :Fax)) Main*
>   *linkFax :***Fun***(imports :***Sig***(phone :Phone)) Fax*
>   *linkPhone :***Fun***(imports :***Sig***()) Phone)*

The Tycoon POS model is also capable of giving precise signatures to the the generic client and server stub generators (built-in components of the distributed DBPL environment) working themselves on signatures and bindings:

```
clientStubGenerator :Fun(Scope<:Sig() Iface<:Sig())
     Fun(serverId :ServerId) Iface
```

The *clientStubGenerator* is a higher-order function that takes an interface signature *Iface* (an arbitrary subtype of the empty signature) and returns a link function. This link function has type **Fun***(serverId :ServerId) Iface*, i.e. based on a connection information supplied dynamically at run-time, it will return a module value of type *Iface*, an aggregation of bindings to stub procedures. Each stub procedure generates a linear, portable representation of its function arguments and transmits this representation to the server (using a value of the not further specified type *ServerId* supplied as a module argument). The *Scope* argument contains the signatures of transitively imported interfaces to access their type information (e.g., to transmit values of type *Phone.Number*).

The task of the *serverGenerator* is similar. It takes the link function for a user-supplied remote implementation (of type **Fun***(imports :Scope) Iface*) and returns a link function for a main program with the same imports. It creates a main program that listens for client requests and dispatches them to parameterless wrapper functions. For each function signature in *Iface* there is a wrapper function that takes arguments from a client connection channel and passes them as arguments to the corresponding function obtained from *linkRemoteImpl*. The return value is again linearized and returned to the client that issued the request.

```
serverGenerator :Fun(Scope<:Sig() Iface<:Sig()
   linkRemoteImpl :Fun(imports :Scope) Iface)
      Fun(imports :Scope) Main
```

The signatures of the generator functions illustrate a crucial feature of the Tycoon POS model: it allows the user to capture type dependencies between the arguments and the result of a generic function. For example, each application of the client stub generator to an argument of type *Iface* will return a function that returns values of type *Iface* (*parametric polymorphism* [27]). Since the supertype specified in the generator signature (*Iface<:***Sig***()*) is different from the type **Any**, this form of universal type quantification is also called *bounded parametric polymorphism* [8]. On the other hand, in contrast to classical models of polymorphism, we are assuming that the (function) value returned by the generators *depends* on its type argument.

Finally, it should be noted that existing proposals and standards for languages to describe data exchange protocols and remote procedure entry points [2] lack any support for polymorphic values and functions. This lack of adequate type descriptions is reflected by the fact that standards for remote database access (RDA [1]) use ad-hoc solutions to transfer query results and query expressions (e.g., as plain string values) and that the distributed DBPL system does not adhere to these standardization proposals.

2.4 The Potential of Persistent Objects

Interoperability aims at providing flexible and safe mechanisms to share data and programs across system boundaries. The previous two sections tackled the problem of type-safe sharing across compilation units and platform boundaries. This section demonstrates how the concept of *orthogonal persistence* [4] as found in DBPL extends the potential for interoperability along two new dimensions: sharing over time and sharing between multiple users.

The following two procedures (to be declared inside the implementation module *Phone*) illustrate a classical approach in managing data objects that outlive a single programming session via operating-system files:

```
procedure save(fileName :array of char);
   var f :File.T;
begin  f:= File.create(fileName);
   File.writeBytes(localPrefix); File.writeBytes(home);
   File.close(f);
end save.

procedure load(fileName :array of char);
   var f :File.T;
begin  f:= File.open(fileName);
   File.readBytes(localPrefix); File.readBytes(home);
   File.close(f);
end load.
```

In the terminology of the Tycoon POS model, the *save* and *load* procedures use operations on operating system files (exported by the module *File*) to implement *persistent value bindings*. In fact, one can view a file system as a large aggregated binding that binds file names (value identifiers) to locations (updatable files) of type **file of** *Byte*. In this view, hierarchical file systems provide nested bindings and (symbolic) links correspond to *aliases* (see Sec. 2.1).

Even in this small example, one can recognize the conceptual problems of this approach that hamper its usefulness for real-life data-intensive applications:

- Naming: The use of external file names is a common source of difficulties in system management since programs become sensitive to changes in their computing environment. It is interesting to note that early languages (notably Cobol) were rather careful to document explicitly dependencies on external resources in their program sources. Automatic (possibly parameterized) name-lookup functions (program-, language- or operating-system specific) tend to complicate the naming issues for persistent objects even more.

- Typing: Since the file system does not preserve the signatures of its bindings, it is necessary for the *save* operation to store a limited form of type information (e.g., a file type key) in each file that is to be checked by the *load* operation against the type information expected by the importer (see also [3])

- Genericity: For each type of value held in main memory, traversal routines have to be implemented that write linear value representations to files and read them back into main memory preserving sharing and cycles. This is a highly repetitive and error-prone task that can be automated by type-directed algorithms.

- Concurrency, consistency, recovery: Since the name space offered by the operating system is not only persistent but also *shared* between several programs, it becomes necessary for the programmer to insert special code to ensure that main memory and persistent data are properly synchronized even in the presence of concurrent access by several users. Furthermore, it is often necessary to ensure that destructive updates of persistent data structures are performed *atomically* to handle system and program failures graciously.

- Efficiency: In large-scale systems it is difficult to decide when to execute *save* and *load* operations and at which granularity to transfer data between volatile and persistent store, in particular, if the persistent data does not fit into main memory. It is therefore desirable to separate these operational aspects from the algorithmic task at hand.

DBPL introduces the notion of a *persistent module* (database module) to define persistent location bindings in a strongly-typed programming environment. For example, the following local changes marked by underscores are sufficient to turn *localPrefix* and *home* into persistent variables (compare Sec. 2.2).

```
database definition module Phone;
type Number = array [0..20] of char;
var localPrefix, home :Number;
procedure operator(prefix :Number) :Number;
end Phone.

implementation module Phone;
procedure operator(prefix :Number) :Number;
begin ... return ... end operator
database definition begin
localPrefix:="4940"; home:=concat(localPrefix,"85312");
end Phone.
```

The compilation of these modules defines a collection of signatures that is identical to the non-persistent environment *CompileEnv* defined in Sec. 2.2. The import semantics of persistent modules differ from volatile modules: the execution of the module initialization code *linkPhone* of a persistent module at link-time does not return a new set of bindings to newly created process-local locations (copy semantics), but returns bindings to locations that are *shared* between all programs importing the persistent module (reference semantics). Therefore, side effects created by one application on persistent variables are visible to other applications importing the same persistent module:

```
module Main1 import Phone;
begin Phone.localPrefix:= "004940" end Main1.
```

module *Main2;* **import** *Phone;*
begin *Phone.localPrefix:=concat(Phone.localPrefix,"-")*
end *Main2.*

The sequential execution of *Main1* and *Main2* would lead to the following location bindings:

bnd(*localPrefix*=**var**(α_1)*"004940-"*
home= **var**(α_2)*"494085312"*
operator=**fun**(*prefix :Number*) ...)

The statement sequence marked by the keywords **database definition begin** is executed only once during the lifetime of the database module *Phone*, namely before it is imported for the first time into a DBPL application program.

The main advantages of persistent modules lies in overcoming the naming, typing and genericity problems associated with file-based solutions without introducing additional linguistic complexity in the programming environment. Furthermore, DBPL supports user-defined (parameterized) *transactions* to handle the concurrency-control, recovery and integrity issues raised above based on standard database transaction models.

Bindings to persistent locations are implemented by maintaining (at compile- and at run-time) a mapping between module-level location names and *external* locations and by having the compiler insert *save* and *load* operations at appropriate code positions for values of appropriate granularity. The task of caching values from external locations in main-memory locations bears similarity to traditional register allocation techniques [30, 22]. The *save* and *load* operations and the transaction primitives (*begin, end, abort transaction*) also trigger concurrency-control and recovery mechanisms [33].

The DBPL approach to transparent persistence management blends well with the transparent RPC-based distribution mechanisms described in Sec. 2.3. For example, it is possible to import remote values and remote locations or to perform dynamic location bindings in remote function applications (i.e. to pass variable parameters or pointers to remote functions). Finally, it is possible to delay the marshalling and unmarshalling of bindings (e.g., in argument lists or records) until the value bound to a name is actually accessed (*call by need*)[3].

2.5 Bulk Types in DBPL

In the process of building a POS, it is often necessary to handle large, dynamic homogeneous collections of objects (e.g., class extents). Furthermore, it is necessary to represent relationships between object collections and to perform efficient, set-oriented update and retrieval operations.

Database systems have been designed to provide specific system and modelling support for these tasks. Expanding on our running example, let us assume that there is a need to store information on telephone numbers assigned to persons as well as fees per unit

[3]The latter mechanism has not been implemented in the DBPL system.

assigned to area codes. This kind of information is adequately described by the following relational database definition:

> **createdb** *SQLPhoneDB* ...
> **create table** *register (name* **char***(50), num* **char***(21))*
> **create table** *fees (prefix* **char***(50), cost* **int4***)*

The language SQL provides simple, efficient, declarative read and write access to the information held in the database (**insert into table**, **delete from table**, **select** ... **from** ... **where** ...). However, it turns out to be surprisingly difficult to access SQL databases from application programs, e.g., to use the *Fax* service to send a message to every person named "Smith". We do not want to go into the details of SQL host language embedding, but the problems encountered can be classified as follows:

Naming Problems The only way to name SQL objects (databases, tables, attributes) from a programming language is via strings. These strings adhere to quite non-standard scoping rules. There is no mechanism to directly identify table elements. Programmers therefore have to explicitly create *cursors* to navigate over tables. Essentially, the programmer is forced to explicitly introduce programming language variables (cursors, database handles, ...) for every DB object being accessed.

Typing Problems While it is easy to find an ad-hoc translation between values of the SQL base types (**int4**, **float8**, ...), there are no mechanisms to share type information (e.g., record or table types) between databases and application programs. Every application program therefore has to repeat the type declarations made in the database schema. Null-values in databases have to be handled via special "indicator variables" in application programs. Most importantly, there is no static type checking between type assumptions made in the application and those found in the database. Mismatches are only detected at run-time or result in "arbitrary" values being stored in communication buffers.

Binding Problems There is no mechanism to statically bind applications to databases. Binding and scoping is performed typically via side-effects, e.g., by issuing a *connect("SQLPhoneDB")* command, that affects the interpretation of all table names used in subsequent statements. Finally, query expressions are subject to "textual" substitutions that are incompatible with classical programming language semantics. For example, the correctness of the query **select** *x.num* **from** *:RangeExp* **where** *x.name* =*"%Smith%"* depends on the value of the string host variable *RangeExp* ("quoted" with a ":"). A legal substitution would be *RangeExp:= "register, x"*, while *RangeExp:= "fees, x"* yields a scoping error and *RangeExp:= "x>400"* yields a syntax error at run-time.

It should be noted that these problems are not specific to relational database languages; the very same difficulties arise in newly designed persistent object management systems that also provide special-purpose data definition and query languages [9, 18].

DBPL overcomes these difficulties by using the persistence and modularization concepts described in Sec. 2.2 and extending the language by a generic bulk types operator **relation** and predefined polymorphic operations on values of type relation. Due to the orthogonality of the DBPL type system, it is possible to define a richer set of data structures than it is possible in the classical relational model, but this flexibility is not required here [32]. The SQL database schema is represented by the following persistent DBPL module:

```
database definition module PhoneDB; import Phone;
type String = array [0..49] of char;
type Entry =
        record name :String num :Phone.Number end;
type Fee = record prefix :String  cost :integer end;
type Register = relation name of Entry;
type Fees = relation prefix of Fee;
var register :Register; var fees :Fees;
end PhoneDB.
```

Names have been assigned to all types to be re-used in application programs importing the database variables *register* and *fees*. DBPL provides a rich set of set-oriented update operators and an extended relational calculus including recursion based on fixed-point semantics to express bulk operations: [12]

```
if card(register) = 0 then ... end;
johnsEntry:= Entry{"John", "237"};
register:= Register{johnsEntry};
register:+ Register{{"Peter", "249"}};
register:–Register{each n in register:n.num>"240"};
```

Relations can be viewed as collections of *location* bindings (indexed by the key values defined in the relation type declaration). Consequently, it makes sense to provide element-oriented update operators and destructive iteration capabilities:

```
register["Peter"].num:= "250";
transaction addPrefix;
begin for each n in register :true do
        n.num:= concat(Phone.localPrefix, n.num) end
end addPrefix;
```

DBPL has special (generic) type rules for the built-in relation operators, e.g., to capture the fact that the set-oriented insertion operator ":+" can be applied to relations of arbitrary element type *E*, as long as the right-hand side expression is also a relation of the same element type *E*:

```
fees:+ Fees{{"853", 14}}
```

Relation types provide a good example for the use of type operators and type-parameterized functions in the Tycoon POS model introduced in Sec. 2.1. As a first step to define the signature of the interface *PhoneDB*, the built-in DBPL type environment could be represented as follows:

```
DBPLBuiltinEnv ≡ Sig(
    Relation <:Oper(ElementType<:Any) Any
    {} :Fun(E <:Any) Relation(Any)
    CARD :Fun(E <:Any  rel :Relation(E)) Int
    :+ :Fun(E <:Any  var lhs :Relation(E)
            rhs :Relation(E)) Any
    ... )
```

This environment declares a type operator *Relation* that maps arbitary types (subtypes of type **Any**) to a hidden representation type (a subtype of type **Any**). The subsequent function signatures make use of this type operator to express the type constraints on the built-in DBPL functions. For example, the standard function **CARD** takes an arbitrary type argument *E*, a relation *rel* of type *Relation(E)* and returns the cardinality of the relation, a value of type **Int**.

Using the abstract type operator *Relation* exported from *DBPLBuiltinEnv*, the signature of *PhoneDB* looks as follows:

```
CompileEnv ≡ DBPLBuiltinEnv  ∪
   Phone <:Sig(...)
   PhoneDB <:Sig(
      String <:array ...
      Entry<:Sig(name:String num:Phone.Number)
      Fee <:Sig(prefix :String cost :Int)
      Register <:Relation(Entry)
      Fees <:Relation(Fee)
      var register :Register  var fees :Fees)
```

3 Interoperability and Genericity

In the previous section we discussed dimensions of object interaction on the instance level. In the framework of the DBPL language and system we presented support for safe inter-object interactions based on naming, typing and binding mechanisms that apply uniformly to computational objects whether they are local or remote, short-lived or persistent, small or large.

The notion of interoperability applies, of course, to a more general setting in which independently developed, *generic* systems have to cooperate, i.e. to provide for inter-object interactions that handle *all* potential objects the systems may create and maintain. In this section we report on our experience with such generic cross-language interoperability via the DBPL/C and the DBPL/SQL gateway and outline general requirements for generic gateway implementation. We will continue focussing on the underlying naming, typing and binding concepts presented in terms of the Tycoon POS model introduced above.

3.1 Cross-Language Interoperability

The most primitive (but also most common) form of cross-language interoperability is achieved by having a standardized, language-independent link format (e.g., COFF of Unix System V) that allows static bindings in a language L_{imp} to bind to values or locations defined in another language L_{exp}. In this setting, L_{imp} is able to import from L_{exp}. The next step is to define standardized, language-independent parameters passing conventions that allow argument values or argument locations defined in L_{imp} to be bound dynamically to function parameters defined in L_{exp}. If the roles of L_{imp} and L_{exp} can be interchanged, full cross-language interoperability (including "call-back" mechanisms) is supported.

Since this interoperability takes place at the value, location and binding level only, all naming and typing consistency control enforced by the use of type names, types and signatures in compilers is effectively lost in this scenario[4].

The DBPL compilers for VAX, Sparc and Motorola architectures attack this problem by providing the DBPL programmer with a mechanism to recover type and signature information for external bindings via so-called *foreign definition modules*. For example, the interface of the *Fax* module defined in Section 2.3 could be revised as follows to define a FAX service implemented in the programming language C:

```
definition for C module Fax;
import Phone;
type Status = (error, busy, done);
...
end Fax.
```

The compiler will enforce the consistent use of the bindings exported by the external *Fax* package in all importing DBPL programs. For example, it would catch the following type error in the application of the function *Fax.dial* that attempts to pass an integer value as a string argument:

```
module SendFax; import Phone, Fax;
begin
Fax.dial(853228);Fax.send("Sample fax.");Fax.hangup();
end SendFax;
```

The skeleton of a C-program to provide value bindings matching the signatures *Fax* looks as follows.

```
typedef char* Phone_Number
typedef int Status
#define error ((Status) 0)
#define busy ((Status) 1)
#define done ((Status) 2)
Status Fax_dial(char* number){ ... }
```

[4]There are, however, attempts to define "architecture-neutral distribution formats" that preserve (low-level) type information until link-time [31].

```
Status Fax_send(char* text) { ... return done; ... }
Status Fax_receive(char** text) { ... return done; ... }
void Fax_hangup() { ... }
```

The location binding in the signature of the function *Fax_receive* is achieved by declaring the argument type for the parameter *text* with an extra level of (pointer) indirection. The compiled C object file (File "fax.o") can be viewed as a collection of function value bindings.

```
faxCObjectFile = bnd(error=0 busy=1 done=2
                     Fax_dial=fun(number :char*) ...
                     Fax_send=fun(text :char*) ...
                     Fax_receive=fun(var text :char*) ...
                     Fax_hangup=fun() ... )
```

In reality, the object file does not contain the constants defined by preprocessor macros and the type information associated with the function[5]. The binding *faxCObjectFile* therefore has the following signature:

```
faxCObjectFile :Sig(Fax_dial :Any  Fax_send :Any
                Fax_receive :Any  Fax_hangup :Any)
```

The DBPL language environment provides a *generic* (type-dependent) function (a generator) that is capable of extracting relevant external bindings specified by a DBPL interface signature *Iface* from a given external binding. In the Tycoon POS model this generator has the following signature:

```
linkC: Fun(Iface <:Sig()  prefix :string
           externalObject :Sig()) Iface
```

The type signature *Iface* <:**Sig**() matches any actual type parameter that is an aggregated signature without further constraints on the signature elements. For example, the interface heading **definition for** *C* **module** *Fax* causes the DBPL compiler to automatically generate the following module initialization code *linkFax* that calls *linkC* with module-specific type and value arguments:

```
linkFax = fun(imports :Sig(phone :Phone))
          linkC(Iface≡Fax  prefix="Fax"
                externalObject=faxCObjectFile)
```

The prefix "Fax" is used to locate the value bindings *dial, send, receive, hangup* in *faxCObjectFile* based on the (C) naming convention that the names of components in an aggregated binding are prefixed by the name of their enclosing scope. If the generic *linkC* function encounters value and location specifications in a DBPL signature for which no matching bindings can be found, a "link error" is reported.

[5]See [13] for an approach to partially encode the signatures of values in their names to support a limited form of cross-module type checking in C++.

An additional level of consistency could be achieved by implementing a tool that generates C type, constant and function prototype declarations based on the signature information extracted from a DBPL foreign definition module (and all its transitively imported interfaces)[6]. Since modern C compilers are capable of verifying function declarations against given prototypes, this tool would guarantee full type consistency for cross-language bindings (static and dynamic) from DBPL to C. However, in view of the fact that the vast majority of external library code is not shipped as source text but as untyped object code, such a tool is of limited use in current programming environments.

Since relation types and relation operations are fully integrated into the DBPL language, they can be freely combined with the cross-language binding mechanisms:

```
for each b in register :contains(n.name, "Smith") do
    Fax.dial(n.num); Fax.send(...); end
```

It is also possible to call DBPL transactions on bulk objects from C, e.g., *dbpl_addPrefix()*.

3.2 The DBPL/SQL Gateway

Cross-language linking as described in Sec. 3.1 is an instance-base interoperability mechanism limited to individual function and data values. DBPL generalizes these concepts to provide transparent translation of families of data structures and expressions that are defined in terms of generic types and polymorphic functions.

Analagous to external function bindings, DBPL supports bindings to external persistent objects in addition to internal persistent DBPL objects [23]. For example, the following modification of the header of module *PhoneDB* binds the location variables *register* and *fees* to *external* SQL relations *register* and *fees* defined in an Ingres database named *SQL-PhoneDB*:

```
database definition for Ingres module PhoneDB;
import Phone;
...
var register :Register; fees :Fees;
end PhoneDB.
```

All DBPL statements and expressions referring to these variables are translated fully transparently into SQL update and selection expressions submitted to the Ingres SQL database management system. These SQL expressions typically take DBPL program variables (value and location bindings) as arguments and return (set) values that are converted appropriately for further processing within DBPL. For example, the query

```
if all n in register n.phone > x then ... end
```

[6]Similar tools play a central role in window systems involving the PostScript language [36] and proposals for distributed object management[14].

is translated into a **select from where** SQL expression that uses the actual value stored in the DBPL location x of type **String**. Depending on the cardinality of the set-valued result, a boolean value is then returned to the compiled DBPL code.

It should be noted that DBPL can handle arbitrarily nested (possibly recursive) query expressions that mix volatile relations, persistent DBPL relations residing in local or remote databases and SQL database relations. Therefore, much care has been devoted to develop evaluation heuristics that minimize data transfer and make best use of index information available for individual relations. Evaluation strategies are not determined at compile-time but depend on cardinality and index information available at run-time.

Again, a conceptually simple generalization of an existing programming language concept suffices to overcome the interoperability deficiencies of todays database programming interfaces that have developed in a system-driven, bottom-up fashion.

3.3 Architectural Requirements of Generic Gateway Implementations

Although the system details of the DBPL/SQL gateway are quite delicate and often require ad-hoc case analysis to achieve good system performance, this specific gateway implementation follows a more general pattern that directly reflects the model of typed programming languages in terms of types, signatures, values and bindings presented in Sec. 2.1. In order to extend a language L_{int} by a generic gateway to an external language L_{ext} successfully (i.e., to embed L_{ext} as a sublanguage of L_{int}), the following conditions have to be met.

The *type* syntax of L_{int} must be sufficiently expressive to capture the structure of values in L_{ext}. This may require extensions to the set of base types (e.g., to handle SQL date, time and table key values) as well as extensions to the set of type constructors (e.g., to handle SQL relations, views or indices). In DBPL, the base type extensions are covered by user-defined abstract data types, while the type constructors are mapped to built-in DBPL type constructors.

There have to be tools to aid in the mapping between *signatures* in L_{int} and L_{ext}. Instead of writing a generator that maps from Ingres DB schema information to DBPL database definitions or vice versa, we developed an interactive binding tool that automatically extracts signature information from DBPL module descriptions and the Ingres data dictionary and verifies the compatibility of these separately developed descriptions.

Tool support is also required at run-time to establish value and location *bindings* from names in L_{int} to entities in L_{ext} and vice versa. In the DBPL/SQL scenario this is achieved by using *DynamicSQL*, a set of library routines shipped with the Ingres DBMS to create cursors and communication buffers to convert Ingres values (element-by-element, attribute-by-attribute according to their type structure) to DBPL values and to pass arguments (of scalar types) to SQL query strings.

If *expressions* of L_{ext} are to be generated from (a subset of) expressions in L_{int} (e.g., SQL queries based on DBPL queries), the expression syntax of L_{int} has to be sufficiently general to emulate high-level external abstractions present in L_{ext}. For example, DBPL's

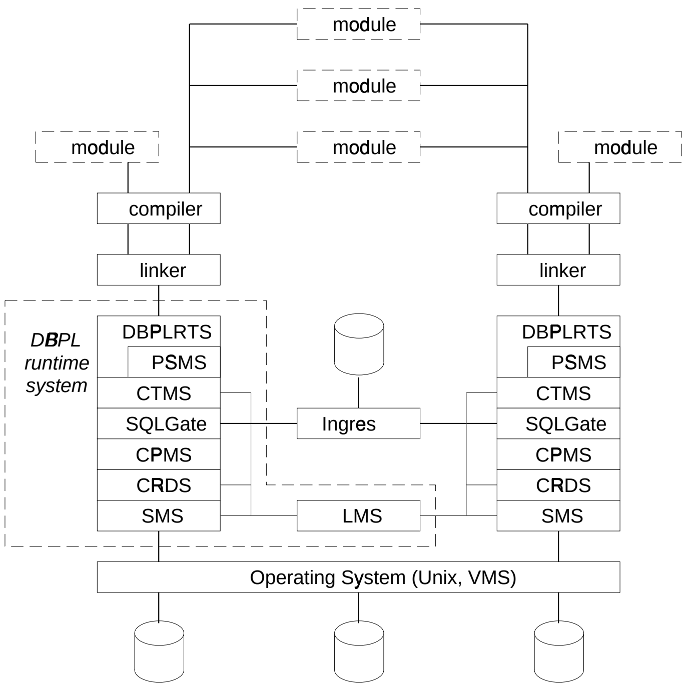


Figure 1: Integration of the SQL gateway into the DBPL system architecture

query expressions directly correspond to SQL **select from where** expressions, whereas a mapping from arbitrary imperative loop statements to SQL expressions does not seem to be feasible.

Finally, there have to be means to ensure that typing assumptions in the static compilation context of L_{int} (expressed, for example, as signatures of database variables) are met by the corresponding bindings provided in L_{ext} at run-time. In the DBPL/SQL context, this limited form of dynamic type checking is achieved by having the compiler generate run-time type representations for external database variables that are checked during program startup (accessing the SQL data dictionary).

In addition to these linguistic and technical prerequisites, a smooth integration of internal and external services also requires adequate architectural support, like access to the scoping and typing phase of the compiler, support for separate compilation and persistent storage of type and signature information, or abstract program representations that support static (and possibly dynamic) program analysis and translation.

The majority of the additional interoperation services (cross-language binding, cross-platform binding, transparent language translation) described in this paper are handled by localized extensions of the layered DBPL architecture displayed in Fig. 1. For example, the layer *SQLGate* inserted between the (predicate-based) transaction management system *CTMS* and the predicate evaluation system *CPMS* (the non-recursive query evaluator) localizes virtually all extensions to the DBPL system required to handle external databases (6K lines Modula-2 code of a total of 52K lines DBPL run-time system code). *SQLGate* transparently dispatches subqueries accessing local database objects to *CPMS* amd subqueries accessing external database objects to Ingres/Oracle.

4 Towards Open Communicating Environments

After seven years of development, the DBPL system has now reached a level of maturity and interoperability that makes its linguistic abstractions readily available for implementors of non-trivial Persistent Object Systems on several hardware-platforms.[7]

From a research point of view, an interesting side-effect of this DBPL implementation effort is an insight into repeating patterns of language and system extension requirements, some of which are outlined in Sec. 3.3. Consequently, our current work in the Tycoon project investigates languages and architectures that facilitate such incremental, problem-specific extensions in a type-safe environment.

In contrast to DBPL, Tycoon takes a rather radical approach by not maintaining upward compatibility with existing programming languages (Modula-2) and data models (complex-object models). Also its internal protocols for store access, program representation and linkage do not adhere to pre-existing standards. The rationale behind the design of Tycoon is to provide a lean language and system environment that provides just the kernel services and abstractions needed to define higher-level, problem-oriented "languages" and "data models".

The following two sections sketch how Tycoon overcomes limitations of classical languages and architectures and contributes to the development of future open communicating environments.

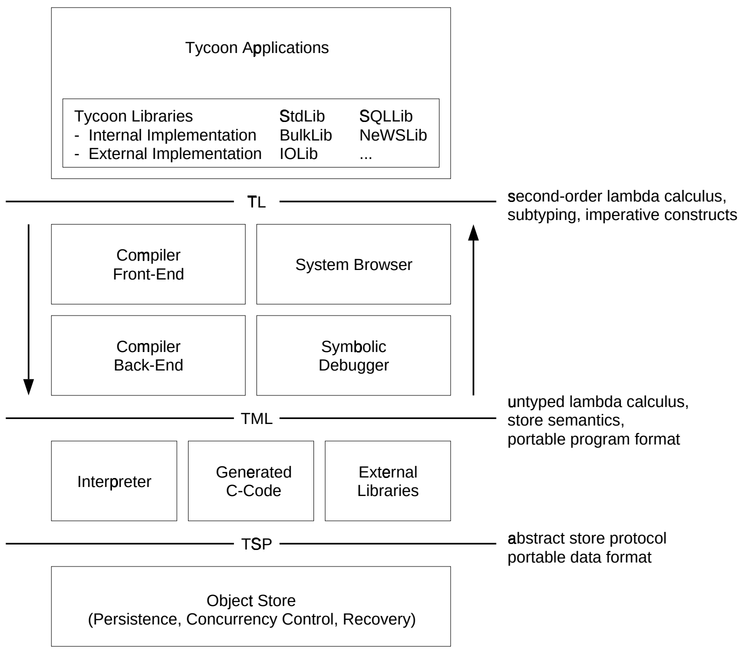
4.1 Beyond Classical Database System Architectures

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, bulk data indexing, iteration abstraction, meta-data management, access control, data distribution 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 services by strictly separating data modelling, data manipulation and data storage issues. Horizontal bars in Fig. 2 indicate the three central Tycoon system abstractions:

- TL is a strongly-typed higher-order polymorphic programming language based on the notions of types, signatures, values and bindings as introduced in Sec. 2.1. TL serves as a uniform application and system programming language and therefore has to support both, programming using high-level problem-specific data models as well

[7]The DBPL system is distributed by Hamburg University.



Tycoon Applications
Tycoon Libraries
- Internal Implementation
- External Implementation
StdLib
BulkLib
IOLib
SQLLib
NeWSLib
...
TL
second-order lambda calculus, subtyping, imperative constructs
Compiler Front-End
System Browser
Compiler Back-End
Symbolic Debugger
TML
untyped lambda calculus, store semantics, portable program format
Interpreter
Generated C-Code
External Libraries
TSP
abstract store protocol portable data format
Object Store (Persistence, Concurrency Control, Recovery)

Figure 2: The Tycoon system architecture

as the implementation of these data models in terms of low-level implementation-oriented data structures. Further aspects of TL are discussed in the Sec. 4.2

- TML is a minimal intermediate language (22 instructions) 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.

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

This layered protocol-oriented architecture aims at system *scalability* and *portability* by de-emphasizing operational aspects of the protocol implementations. For example, two implementations of TML currently exist, a compact, portable interpreter, immediately executable on a wide range of hardware-platforms and an optimizing code generator. Both can be combined freely with several object store implementations ranging from garbage collected, paged main memory implementations suitable for personal computers to sophisticated persistent store implementations exploiting virtual memory hardware available on file servers.

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 interoperability. This functionality had to be emulated in the DBPL project either by resorting to interpretation techniques (e.g., to execute access plans generated by the query optimizer) or ad-hoc binding mechanisms in the distributed DBPL system.

It will be interesting to compare the efficiency of the layered Tycoon architecture that does not provide set-abstractions at the store level with the efficiency of traditional, tailored DBMS query interfaces.

4.2 Beyond First-Order Type Systems

Given the architecture in Fig. 2, several tasks handled by hard-wired algorithms inside the DBPL runtime system (Fig. 1) or inside an SQL database server can be defined in the TL language. For example, it is possible to define new collection types (bulk data structures) and iteration abstractions (query formalisms) as well as higher-level language concepts for integrity maintenance and transaction management. In this section, we demonstrate merely how these tasks can be delegated to external servers while giving the user the illusion of a fully integrated persistent system.

In order to factor out repeating generic programming tasks from individual applications into reusable library code, TL requires type abstraction capabilities that are beyond classical first-order type systems. The type-theoretic starting point for TL is $F_{\leq}$, a second-order lambda calculus with subtyping [7]. This is extended by a rather small set of standard programming concepts (base types, tuples, arrays, control structures and exceptions) as found in other functional or imperative programming languages. Furthermore, the type concepts of $F_{\leq}$ are generalized to higher-order (in the spirit of Quest [6]), providing the formal basis for *generic* interoperability as sketched in the Sec. 3.

In the following, we express an idea how the type-safe integration of external generic servers is accomplished in the Tycoon system. Due to space limitations we cannot go into the details of the language TLbut based on the informal Tycoon model presented in Sec. 2.1 the reader should be able to follow the examples. A formal definition of TL can be found in [26]. Here is a (simplified) interface taken from the current Tycoon libraries that describes the signatures of the polymorphic functions exported by an SQL server:

```
interface SQL export
  error :Exception with sqlError :String end
  Table(E<:Tuple end)<:Ok
  openTable(Dyn E<:Ok name:String):Table(E)
  insert(E<:Ok var table:Table(E) tuple:E):Ok
  :+, :-(E<:Ok var table:Table(E) tuples:Table(E)):Ok
  delete(E<:Ok var table:Table(E) where(: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** which corresponds to type **Any** in Sec. 2.1). 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:

```
module ingres import DynamicSQL export
  let error=exception "Ingres Error"
             with sqlError :String end
  Let Table(E<:Ok) =
     Tuple hostId, tableId :String ... end
  let openTable(Dyn E<:Ok name:String) =
     begin ... end
  ...
end
```

The scope for modules and interface names is represented explicitly via libraries:

```
library SQLLib with
  interface DynamicSQL ... SQL
  module ... ingres :SQL  oracle :SQL
  hide DynamicSQL ...
end
```

The library *SQLLib* exports an interface *SQL* and two modules implementing this interface, and it hides several local modules required for gateway implementation. Libraries typically appear as components of larger libraries:

```
library PhoneApplication with
  library SQLLib
  interface PhoneDB
  module phoneDB :PhoneDB  test :Main
end
```

The *PhoneApplication* subsumes an interface to an external Ingres *PhoneDB* (see DB definition in Sec. 2.5):

```
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 relation variables is established in the module implementation:

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

This example illustrates how higher-order type systems contribute to lean languages and systems. 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 compilers. (Further examples of TL signatures for object-oriented and deductive data models are given in [21]).

As a specific example, the SQL statement *delete* ... *from* ... *where* ... removes all elements of a table that fulfil a given predicate which has to be compatible with the table structure. These semantic restrictions are captured correctly by the signature of the function *delete* in the interface *SQL*. It specifies that the function works uniformly over all types E given a location binding to a *table* with elements of type E and a binding to a function *where* mapping elements of type E to a boolean value (i.e. a predicate on domain E).

Given the flexibility of higher-order type systems, it is not necessary to limit the language to a fixed set of base types. In fact, Tycoon does not provide any built-in operations on base types (except the conditional for booleans). The usual arithmetic and string operations are all implemented in external libraries that are (dynamically) bound to the Tycoon system. This makes it particularly easy to modify the internal representation of the base types or to add new base types with their associated operations as abstract data types (e.g. sound or pixel data types).

5 Conclusion

Within a uniform framework based on the notions of types, signatures, values and bindings, this paper presents two approaches to reduce the complexity of Persistent Object Systems and to increase significantly programmer productivity.

The relational database language DBPL provides built-in system and language support for a uniform handling of computational objects whether they are local or remote, short-lived or persistent, small or large. Generic gateways provide transparent and optimizing access from DBPL to external programming languages and commercial database systems.

The Tycoon environment minimizes the amount of built-in system and language support by exploiting higher-order type concepts and by strictly separating the issues of data storage, data manipulation and data modelling into three distinct formalisms and system layers.

We expect such lean languages and systems, where most of the functionality of a Persistent Object System is achieved by importing external services into a conceptually sound linguistic framework, to possess a higher potential for scalability, portability and interoperability than classical "all-in-one" database systems.

References

[1] ISO / IEC JTC 1 / SC21/ WG 3. Information processing systems – open system connection (OSI) – remote database access (RDA), generic model. Technical report, ISO 9579, 1991.

[2] ISO / IEC JTC 1 / DIS 8824. Abstract Syntax Notation One (ASN.1), draft international standard. Technical report, ISO, 1992.

[3] M. Abadi, L. Cardelli, B. C. Pierce, and G.D. Plotkin. Dynamic typing in a statically typed language. Report 47, DEC Systems Research Center, 130 Lytton Avenue, Palo Alto, 94301 CA, June 1989.

[4] M.P. Atkinson and P. Bunemann. Types and persistence in database programming languages. *ACM Computing Surveys*, 19(2), June 1987.

[5] L. Cardelli. Structural subtyping and the notion of power type. In *Proceedings of the Fifteenth ACM Symposium on Principles of Programming Languages*, 1988.

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

[7] L. Cardelli, S. Martini, J.C. Mitchell, and A. Scedrov. An extension of system F with subtyping. In T. Ito and A.R. Meyer, editors, *Theoretical Aspects of Computer Software, TACS'91*, Lecture Notes in Computer Science, pages 750–770. Springer-Verlag, 1991.

[8] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[9] R.G.G. Cattell. Next-generation database systems. *Communications of the ACM*, 34(10), October 1991.

[10] 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.

[11] J.R. Corbin. *The Art of Distributed Applications.* Sun Technical Reference Library. Springer-Verlag, 1991.

[12] J. Eder, A. Rudloff, F. Matthes, and J.W. Schmidt. Data construction with recursive set expressions in DBPL. In *Proceedings of the Kiev East/West Workshop on Next Generation Database Technology*, volume 504 of *Lecture Notes in Computer Science*, April 1991.

[13] M.A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[14] Object Management Group. The common object request broker: Architecture and specification. Document 91.12.1, Rev. 1.1, OMG, 1991.

[15] P. Hudak and P. Wadler. Report on the programming language Haskell version 1.2. *SCM SIGPLAN Notices*, 21(7):219–233, July 1986.

[16] W. Johannsen, L. Ge, W. Lamersdorf, K. Reinhard, and J.W. Schmidt. Database application support in open systems: Language support and implementation. In *Proc. IEEE 4th Int. Conf. on Data Engineering*, Los Angeles, USA, February 1988.

[17] W. Johannsen, W. Lamersdorf, K. Reinhard, and J.W. Schmidt. The DURESS project: Extending databases into an open systems architecture. In *Advances in Database Technology, EDBT '88*, volume 303 of *Lecture Notes in Computer Science*, pages 616–620. Springer-Verlag, 1988.

[18] W. Kim and F.H. Lochowsky. *Object-Oriented Concepts, Databases and Applications.* ACM Press Books, 1989.

[19] G.N.C. Kirby. Persistent programming with strongly typed linguistic reflection. FIDE Technical Report Series FIDE/92/40, Fachbereich Informatik, Universitaet Hamburg, Germany, 1992.

[20] 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.

[21] F. Matthes. *Persistent Object Systems: Linguistic and Architectural Foundations.* Springer-Verlag, 1993. (in German, to appear).

[22] F. Matthes, A. Rudloff, J.W. Schmidt, and K. Subieta. The database programming language DBPL: User and system manual. FIDE Technical Report FIDE/92/47, Fachbereich Informatik, Universitaet Hamburg, Germany, July 1992.

[23] F. Matthes, A. Rudloff, J.W. Schmidt, and K. Subieta. A gateway from DBPL to Ingres. FIDE Technical Report Series FIDE/92/54, Fachbereich Informatik, Universitaet Hamburg, Germany, November 1992.

[24] F. Matthes and J.W. Schmidt. The type system of DBPL. In *Proceedings of the Second International Workshop on Database Programming Languages, Salishan, Oregon*, pages 255–260, June 1989.

[25] F. Matthes and J.W. Schmidt. Towards database application systems: Types, kinds and other open invitations. In *Proceedings of the Kiev East/West Workshop on Next Generation Database Technology*, volume 504 of *Lecture Notes in Computer Science*, April 1991. (also appeared as TR FIDE/91/14).

[26] F. Matthes and J.W. Schmidt. Definition of the Tycoon language TL – a preliminary report. DBIS Tycoon Report 062-92, Fachbereich Informatik, Universitaet Hamburg, Germany, October 1992.

[27] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

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

[29] R. Morrison, M.P. Atkinson, and A. Dearle. Flexible incremental bindings in a persistent object store. Persistent Programming Research Report 38, Univ. of St. Andrews, Dept. of Comp. Science, June 1987.

[30] J.E. Richardson. E: A persistent systems implementation language. Technical Report 868, Computer Sciences Department, University of Wisconsin-Madison, August 1989.

[31] RSRE. TDF specification. Technical report, Defense Research Agency, RSRE, St. Andrews Road, Malvern, Worcestershire WR 14 3PS, UK, October 1991. (2 parts).

[32] J.W. Schmidt and F. Matthes. Modular and rule-based database programming in DBPL. FIDE Technical Report Series FIDE/91/15, Fachbereich Informatik, Universitaet Hamburg, Germany, February 1991.

[33] J.W. Schmidt and F. Matthes. Naming schemes and name space management in the DBPL persistent storage system. In *Proceedings of the Fourth International Workshop on Persistent Object Systems, Martha's Vineyard, Massachusetts.* Morgan Kaufmann Publishers, January 1991.

[34] J.W. Schmidt and F. Matthes. The database programming language DBPL: Rationale and report. FIDE Technical Report Series FIDE/92/46, Fachbereich Informatik, Universitaet Hamburg, Germany, July 1992.

[35] D. Stemple, T. Sheard, and L. Fegaras. Linguistic reflection: A bridge from programming to database languages. In *Proc. HICSS, Hawaii*, pages 46–55, 1992.

[36] Sun Microsystems. NeWS 2.1 programmer's guide. Manual 800-4888-10, Sun Microsystems, 1992.

[37] N. Wirth. Report on the programming language Modula-2. Springer-Verlag, 3rd edition, 1985.