Chapter 3.3.3
Lean Languages and Models: Towards an Interoperable Kernel for Persistent Object Systems

Joachim W. Schmidt and Florian Matthes

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

Summary This text is a synopsis of [4].

1. Introduction

Reliable interoperation between independently developed Persistent Object Systems (POS) requires type-safe access to persistent data objects and generic services:

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 POS can be obtained by instantiating generic services. 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?

2. Persistent Object System Model

We introduce a POS model to describe the naming, typing and binding concepts involved in Persistent Object System interoperability. The model itself is based on concepts of higher-order type systems and is sufficiently expressive to serve as a language-independent framework for program translation, generation and binding.

The POS model is based on the notion of *types*, *signatures*, *values* and *bindings*. Types are understood as (partial) specifications of values. Types include a set of

base types, type **Any**, user-defined type variables, function types, aggregated signatures and type operator applications. Type operators denote parameterized type expressions that map types or type operators to types or type operators. 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 in the full paper.

Interoperability aims at providing flexible and safe mechanisms to share data and programs across system boundaries. We demonstrate how the POS model captures modularization as basis for interoperability, distribution (cross-platform interoperability), persistence (interoperability over time) and bulk data abstractions as provided by the DBPL database programming environment (see Chapter 2.1.2).

Modularization 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.

Distribution The distributed version of DBPL [1, 2] adds an additional layer of type-safety to standard remote procedure call mechanisms (RPC) in federated client-server programming models. In terms of the 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. The POS model is also capable of giving precise signatures to the the generic client and server stub generators working themselves on signatures and bindings. The signatures of the generator functions illustrate a crucial feature of the POS model: it allows the user to capture type dependencies between the arguments and the result of a generic function.

clientStubGenerator :**Fun**(*Scope*<:**Sig**() *Iface*<:**Sig**()) **Fun**(*s* :*ServerId*) *Iface*

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*). 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*. 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.

Persistence The concept of orthogonal persistence extends the potential for interoperability along two new dimensions: sharing over time and sharing between multiple users. 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.

Bulk Data 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. DBPL provides a generic bulk types operator *Relation* and predefined polymorphic operations on values of type relation. 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.

Relation types provide a good example for the use of type operators and type-parameterized functions in the POS model. 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 arbitrary 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**.

3. Interoperability and Genericity

The notion of interoperability applies 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 the full paper 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 continue focusing on the underlying naming, typing and binding concepts presented in terms of the POS model.

The most primitive (but also most common) form of cross-language interoperability is achieved by having a standardized, language-independent link format 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.

Cross-language linking 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. Analogous to external function bindings, DBPL supports bindings to external persistent objects in addition to internal persistent DBPL objects [3].

All DBPL statements and expressions referring to these variables are translated fully transparently into SQL update and selection expressions submitted to the 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.

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. 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 SQL 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 SQL 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. 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}.

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.

4. Towards Open Communicating Environments

In contrast to DBPL, Tycoon (see Chapters 1.1.1 and 2.1.4) takes a rather radical approach by not maintaining upward compatibility with existing programming languages and data models. Also its internal protocols for store access, program representation and linkage do not adhere to pre-existing standards. 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 modeling into three distinct formalisms and system layers.

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". 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.

Acknowledgement This research was supported by ESPRIT Basic Research, Project FIDE, #6309.

References

1. W. Johannsen, L. Ge, W. Lamersdorf, K. Reinhard, and J.W. Schmidt. Database application support in open systems: Language support and implementation. In *Proceedings of the IEEE Fourth International Conference on Data Engineering, Los Angeles, California*, February 1988.
2. W. Johannsen, W. Lamersdorf, K. Reinhard, and J.W. Schmidt. The DURESS project: Extending databases into an open systems architecture. In *Proceedings of the First Conference on Extending Database Technology, EDBT'88*, volume 303 of *Lecture Notes in Computer Science*, pages 616–620. Springer-Verlag, 1988.
3. F. Matthes, A. Rudloff, J.W. Schmidt, and K. Subieta. A gateway from DBPL to Ingres: Modula-2, DBPL, SQL+C, Ingres. FIDE Technical Report Series FIDE/92/54, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, August 1992.
4. 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*, pages 2–16, April 1993.