Chapter 1.1.1

Higher-Order Persistent Polymorphic Programming in Tycoon

Florian Matthes

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

Abstract. This text provides an introduction to Tycoon\textsuperscript{1}, an open persistent polymorphic programming environment. The Tycoon language TL is based on expressive and orthogonal naming, typing and binding concepts as they are required, for example, in advanced data-intensive applications. The characteristic language mechanisms of TL are first-class functions and modules, parametric and subtype polymorphism extended to a fully higher-order type system. Tycoon programs are statically typed but may include explicit dynamic type variables which can be inspected at run-time.

1 Introduction and Motivation

The Tycoon system is an open persistent polymorphic programming environment based on higher-order language concepts. It is designed as a robust linguistic and architectural framework for the definition, integration and interoperation of generic services represented as polymorphically-typed libraries. The architecture of the Tycoon system is described in Chapter 2.1.4.

The Tycoon language \textsuperscript{2}TL described in this paper is used for the following two activities in database application programming (see also [18]):

Strongly typed, high-level application programming: TL is used by application programmers to implement the full functionality of data-intensive applications which require a tight and controlled interaction between objects on the screen, objects in main memory, objects on disk, and objects on the wire. For example, a value from a screen form may be passed as a parameter to a transaction, be stored in a database and finally be transmitted to a remote log server. TL supports such programming tasks by providing uniform and generalized naming, typing and binding concepts that abstract from the specifics of the underlying object servers like GUI toolkits, programming languages, database systems and RPC services. In particular, Tycoon’s type system statically detects any attempt to apply an inappropriate operation from one server to an object from another server. This should be seen in contrast to the current practice in data-intensive applications where there is virtually

\textsuperscript{1} Tycoon: Typed Communicating Objects in Open Environments

\textsuperscript{2} TL: Tycoon Language
no inter-server consistency checking due to the lack of an integrated type system model.

**Generic server integration:** Different from fourth-generation languages, high-level application programming in the Tycoon system is not restricted to built-in object types like tables, forms and reports. By virtue of Tycoon’s polymorphic (higher-order) type system it is possible to also integrate pre-existing, independently developed generic servers (like object-oriented databases, C++ GUI libraries or RPC communication services) as strongly typed parametric libraries into the Tycoon programming environment. Therefore, systems developed in TL fit smoothly into open system architectures.

The idea of an open, library-based approach to system construction is currently being pursued in several system frameworks that are based on C++ or distributed object models of similar expressiveness. Tycoon aims at a higher system development productivity in a language framework with the following characteristics:

**Improved language orthogonality:** All language entities in TL (like values, functions, modules and types) have first class status. For example, it is possible to write a TL function that receives a type as its argument and returns a module which aggregates a set of dynamically constructed functions for a fresh abstract data type. Such higher-order language concepts are particularly helpful to factor-out repetitive programming tasks from individual applications into shared, reusable library code.

**Increased type system expressiveness:** TL combines subtype and parametric polymorphism. Furthermore, both forms of polymorphism are generalized to (higher-order) type operators supporting the type-safe definition of highly polymorphic system libraries.

**Orthogonal persistence abstraction:** TL programmers don’t have to distinguish between local volatile data and shared global and persistent data. As a consequence, programmers can fully abstract from store properties (size of main memory, garbage collection, transfer between primary and secondary store, data format conversion between nodes in heterogeneous networks, etc.).

**Reflective programming support:** Some system tasks in data-intensive applications (e.g. query optimization, transaction scheduling, GUI generation) are based on run-time reflective programming techniques. Run-time linguistic reflection denotes the ability of a system to inspect parts of the system (e.g. query expressions, transaction instruction sequences, type structures) at run-time and to dynamically extend or modify the system based on the outcome of this inspection [25]. For example, the TL programming environment exports a (strongly-typed) function reflect.optimize that takes a TL function value, re-invokes the TL compiler back-end on this function, and returns an optimized version of the function. Contrary to static code optimizations which are based on a limited static context (a single function or a single module), such dynamic code optimizations can exploit run-time information available for the dynamic context of a function (e.g. external function implementations or values of abstract data types).