Arbeitsbereich DBIS
Prof. Dr. Joachim W. Schmidt
Fachbereich Informatik
Universität Hamburg
Vogt-Kölln-Straße 30
D-22527 Hamburg (FRG)

Tycoon

Title: # The Tycoon System and Library Manual

Author: Bernd Mathiske
Florian Matthes
Sven Müßig

Identification: DBIS Tycoon Report 101-96

Status: Revised for system version TL (tl0-dyn 2.0)

Date: January 1996

Description: This document provides a practical introduction to the interactive Tycoon system environment and an overview of its polymorphic libraries. It explains how to bind external C libraries to Tycoon programs and how to work with persistent stores. Moreover, it proposes formatting und naming guidelines for Tycoon programs.

Related Documents: Definition of the Tycoon Language TL:
A Preliminary Report [MS92]
Persistent Object Systems [Mat93]
Online documentation shipped with the Tycoon system software (Unix, MacOS, Windows)
Tycoon WWW home page [Tyc92]

Contents

1 Introduction

Tycoon [1] is a polymorphic persistent programming environment for the development of data-intensive applications in open environments. The Tycoon system emphasizes system scalability and interoperability with commercial servers like Ingres, Oracle, ObjectStore, O2, Inquery, SAP R/3, NeWS, StarView, C and C++ libraries, Sun-RPC, DCE-RPC and Kerberos. Flexible and safe interoperation between these servers in heterogeneous distributed environments is supported by

- an elaborate higher-order type system (precise and generic service definitions),
- orthogonal persistence (longevity for objects of arbitrary complexity and size), and
- orthogonal mobility (unrestricted migration of data, code and threads between multiple system platforms).

Tycoon is a long-term research project at DBIS, Hamburg University that started in 1992. Tycoon is the most recent member of the family of database programming languages and multi-user database systems developed by our group since 1978 (Pascal/R, Modula-R, DBPL).

The Tycoon programming language TL is described in several publications [MMS94, MS94, Mat95, MSW95, MMS95a, MMS95b] and in a formal language report [MS92]. This text complements these publications and provides guidance for programmers during their first steps with the Tycoon system and also some practical programming hints for advanced Tycoon programmers.

2 The Interactive Tycoon Development Environment

TL is implemented as an interactive compiler. This means that the system behaves very much like an interpreter. The user inputs a phrase in order to evaluate a term and the system produces a reply. In between, the term is parsed, type-checked, translated into portable Tycoon machine code and linked with other TL values or external machine code.

Larger TL programs are typically divided into modules, interfaces and libraries that are compiled separately, then linked and executed. Larger fragments of TL source code are usually stored in text files. However, there is no distinction between input read from files or interactively from a terminal. In particular, it is possible to enter modules, interfaces and libraries interactively and to read expressions and commands from files.

The level of interaction at which the user enters a phrase and receives an answer is called the *top level interface* of the system. The details of the top level user interface depend on the host operating system (Unix, Windows, MacOS, Linux, ...) and are described in the online documentation that is shipped with the Tycoon system software.

2.1 Components of the Tycoon System

The Tycoon system consists of two parts:

- The Tycoon store contains program code, persistent data and persistent threads (descriptions of programs in the process of being executed) in a platform-neutral format;

[1] Tycoon: *Typed communicating objects in open environments.*

- The platform-specific Tycoon application is responsible for multi-threaded program execution, persistent store management and for host operating system access (MacOS, Windows, Unix, Linux,...). The online documentation describes how to invoke the Tycoon application with a given Tycoon store.

The evaluation of expressions and the linking of programs takes place in a persistent store. Bindings established in one Tycoon session can be made accessible for later Tycoon sessions. Furthermore, the persistent store is shared. Multiple TL application programs can have access to a linked module. Assignments to shared variables defined in such a module are visible to all importers of that module.

A Tycoon store contains a set of linked modules and interfaces as well as value and type bindings entered at the top level. Within a persistent store there is a single name space for modules, interfaces and libraries. However, every user can set up a private name space for top level bindings. This name space is called a *top level*.

During (interactive) program development, a Tycoon store also contains the Tycoon compiler, linker and library manager which are loaded from a precompiled boot file whenever a fresh store is created.

2.2 Entering Top Level Phrases

The Tycoon top level interface prompt looks as follows:

```
|>
```

Once at the top level, *top level phrases* can be entered, for example to bind a value to a variable:

> ***let*** *peter =* ***tuple*** *"Peter" 5000* ***end****;*
> *⇒ tuple "Peter" 5000 end :Tuple ? :String ? :Int end*

The system displays the value computed for the last binding and its (inferred) type. Note that all input at the top level interface has to be terminated by a semicolon. If the semicolon is omitted, the Tycoon system prompts for further input by:

```
..
```

It is also possible to bind types to variables:

> ***Let*** *Person =* ***Tuple*** *name :String salary :Int* ***end****;*
> *⇒ :Tuple name :String salary :Int end*

A function binding is entered as follows:

> ***let*** *rich(p :Person) :Bool = p.salary > 4000;*
> *⇒ fun(p :Person) :Bool <hidden> :Fun(p :Person) :Bool*

Variables introduced in bindings can be used in subsequent bindings:

> *rich(peter);*
> *⇒ true :Bool*

A top level phrase that starts with the keyword **do** is not parsed and executed as a TL compilation unit but as a top level command. The top level commands control various aspects of the Tycoon system components.

> ***do*** *help;*

The latter command prints a list with the syntax and the description of all available top level commands.

Sequences of Tycoon top level phrases can be stored in files that by convention have the extension *.tyc*. Assuming that the file *Test.tyc* contains the phrases above, the command

> ***do*** *"Test.tyc";*
> *⇒ (Including input from 'Test.tyc')*

reads and immediately executes the phrases. Nested include files are admissible. Files or pipes of top-level commands are a primitive mechanism to access the Tycoon system from other programs. However, their use in normal program development is discouraged.

2.3 Understanding Compile-Time Error Messages

All compile-time error messages start with a source position. For example:

> *stdenv/string.tm:66.13 Syntax error: expecting 'then', found "end"*

indicates that a syntax error was found on the symbol **end** that starts in column 13 of line 66 in the source file *stdenv/string.tm* where the symbol **then** was expected.

Each individual top level input terminated with a semicolon is viewed as an anonymous, numbered source file. For example:

> a ***end***
> *⇒ #15:1.3 Syntax error: expecting <end-of-input>, found "end"*

indicates a syntax error in colum 3 of line 1 of the top level input number 15.

Some effort has been devoted to the generation of informative type error messages:

> ***Let*** *Address* = ***Tuple*** *street :String* ***end***
> ***Let*** *Person* = ***Tuple*** *name :String age :Int address :Address* ***end***
> ***let*** *f(x :Person)* = ***ok***
> ***let*** *address* = ***tuple let*** *street* = *3* ***end***
> *f(****tuple*** *"Peter" 40 address* ***end****);*
> *#25:6.2 Argument type mismatch: '_builtin.String' expected, '_builtin.Int' found*
> *#25:5.25 [while checking tuple field 'street']*
> *#25:6.20 [while checking tuple field 'address']*
> *#25:6.3 [while checking function argument 'x']*

In the example above, a type error occurs because the type *Int* of the *address.street* attribute of the argument passed to the function *f* does not match the type *String* specified for the *address.street* field of the variable *x* in the signature of the function.

The first line of a type error message prints the error context (*Argument type mismatch*) followed by a description of the type expected by the type checker (*_builtin.-String* is the type defined for the *street* field in type *Address*). The next line prints the offending type found in the program (*_builtin.Int* is the type inferred by the compiler for the number 3).

If a type mismatch occurs inside a composite type expression, further error messages in square brackets give a *traceback* that help to localize the error. Source position specifications preceding the square brackets refer to the declaration point of the respective subexpression. In the example above, the error occurs while checking the field *x.address.street* and the faulty binding ***let** street = 3* is found in line 5, column 25.

In a long error listing it is a good idea to look at the non-bracketed error messages first and to consult the type error traceback only if needed.

In addition to syntax and type errors, the Tycoon compiler also flags lexical errors. Beginners sometimes find it difficult to localize these errors. For example, if a piece of input contains a string constant that is not properly delimited by a closing ", the compiler flags a large number of lexical errors since it assumes that the string extends upto the end of the input and since it encounters several line breaks which are not allowed inside string constants:

```
Lexical error: illegal character in source file
```

A similar problem occurs if a closing comment bracket **)* is omitted since the compiler will continue to prompt the user at the top level for additional input until a closing comment bracket is entered. A standard solution to recognize such situations is to enter "3;" several times at the top level and to check which output is produced by the compiler.

2.4 Understanding Run-Time Error Messages

By virtue of Tycoon's static type system, many run-time errors of other languages like *message not understood*, *nil dereference error* or *dynamic type test error* can be avoided. Other run-time errors like *division by zero error* or *array index out of bounds error* are mapped to Tycoon exceptions with typed arguments that can be handled selectively with **try** expressions.

Unhandled exceptions that propagate to the Tycoon top level are displayed by the Tycoon compiler and their arguments are printed in a raw data format, for example:

```
  let a = array "1" "2" "3" end
  a[400];
⇒ *** Exception: {"indexOutOfBoundsError" 2 2 "#9" {"1" "2" "3"} 400}
```

The signatures of all language-defined exceptions are defined in the file *boot.tyc*. Each of these exceptions provides also the detailed source code position of the faulty expression as an explicit argument which can be inspected with a **try** expression, for example:[2]

```
try
  let a = array "1" "2" "3" end
  a[400]
when indexOutOfBoundsError with exc then
  "cannot access index " <> fmt.int(exc.index) <>
  " in line " <> fmt.int(exc.line) <>
  " of source file " <> exc.where
end
⇒ "cannot access index 400 in line 5..." :String
```

2.5 Using Modules and Libraries

Much of Tycoon's functionality (data types like *String*, *Date* or *Time*, file input and output, bulk data management, etc.) are factored-out into library modules that can

[2]The identifier *fmt* denotes a standard formatting module that can be imported with the command *import fmt*.

be imported into applications as needed. This section describes how to work with modules and interfaces.

2.5.1 Syntax for Interfaces and Modules

An interface defines signatures for values and types exported by a module. Modules that match the following interface (*Counter*) export an abstract type *T* and operations on values of this type:

```
interface Counter
export
  T <:Ok
  new(init :Int) :T
  dec(counter :T) :Ok
  isZero(counter :T) :Bool
end;
```

A possible implementation of this interface is the following module *counter*:

```
module counter
import int
export
  Let T = Tuple var value :Int end
  let new(init :Int) = tuple let var value = init end
  let dec(counter :T) = counter.value := counter.value - 1
  let isZero(counter :T) = counter.value == 0
end;
```

In Tycoon, there can be multiple modules implementing the same interface.

2.5.2 Libraries and Import Relationships

Modules and interfaces can import other modules and interfaces by listing their names in an import clause. The name space for modules and interfaces is defined by a root library. A root library lists the names of all components that make up a persistent object system. There are three kinds of library component declarations:

Component	**Description**
library *stdenv*	a (nested) library named *stdenv*
interface *Counter*	an interface named *Counter*
module *counter :Counter*	a module named *counter* implementing interface *Counter*

Since the module *int* imported by *counter* is defined in the Tycoon standard library *stdenv*, a root library declaration for the example above has to include the name of the standard library:

```
library Root
with
  library stdenv
  interface Counter
  module counter :Counter
end;
```

The declaration order of library components is significant since types and values exported from a library component can be imported only into components that follow its declaration.

For example, the library definition for *stdenv* has the following form:

```
library stdenv
with
   interface RuntimeCore
   module runtimeCore :RuntimeCore
   ...

   hide RuntimeCore runtimeCore
end;
```

A library exports all its locally declared components excluding those interfaces and modules listed explicitly in the **hide** clause.

2.5.3 Compiling a Library

The first step in developing a modular application is to write and compile its root library definition. By convention, library definitions are stored in files with the suffix *.tl*. Compiled library definitions are consulted by the compiler during the compilation of modules, interfaces and other library definitions to validate inter-module scoping constraints and to locate compiled components. Library definitions have to be compiled inside out. For example, the library definition of *stdenv* has to be compiled before compilation of the library definition *Root*.

By default, compiled library definitions, interfaces and modules are stored as binary files in the file system (and cached in the object store). The components of separate libraries are stored in separate directories. The name of the directory is derived from the name of its enclosing library.

The command ***do*** *makeLibraries* automatically reads and (re-)compiles library definitions reachable from the current root library. It inspects the file modification date of library source files to determine which library definitions require recompilation. The name of the current root library can be changed with the command ***do*** *setRootLibrary L* or by the command ***do*** *makeLibraries L*.

2.5.4 Compiling Interfaces and Modules

Interface and module definitions are stored in files with the suffix *.ti* and *.tm*, respectively. Imported interfaces and the interfaces of imported modules have to be compiled before the importing module or interface can be compiled. In the example above, the interface *IntOp* for the module *int* has to be compiled before the module *counter* can be compiled.

The command ***do*** *makeComponent x* automatically reads and (re-)compiles the component *x* and all its transitively imported interfaces that are out of date. The command ***do*** *make x* automatically reads and (re-)compiles the component *x* and all its transitively imported interfaces and modules that are out of date. There are other **do** commands to control separate compilation and module linkage, Use the command ***do*** *help* to find out more about thiese commands.

The compiler assigns time stamps to compiled interfaces to validate that all components of a system have been compiled with a consistent set of interfaces. Version conflicts are reported by printing the conflicting time stamps valid at compilation and import-time. For example, if the interface *Counter* has been recompiled without recompiling the dependent module *counter*, the following error message is printed:

```
17.18 'counter' was compiled with version 8535d 22h 46m 23.361s
```

```
    of interface 'Counter'
17.18 [Current version of 'Counter' is 8536d 2h 50m 9.15s]
```

The command ***do** link* x reads the compiled modules that are imported transitively by x into the object store (if they are not cached) and then executes their module bodies. If a module *y* is imported by another module x then the module body of *y* is executed before the module body of x. Linking aborts as soon as the execution of a module body raises an unhandled exception. The module bodies of modules that are already linked are not re-executed. This makes it possible to share (persistent) variables between multiple applications.

The command ***do** unlink* x marks the module x as unlinked. A subsequent import operations on x (or on another module *y* importing x) returns a fresh instance of x by re-executing its module code. Unlinking is performed automatically as soon as a module is recompiled.

2.5.5 Using Modules and Interfaces at the Top Level

The ***do** link* command is sufficient for main programs that do not export any bindings. To get access to exported bindings and signatures it is also possible to import module and interface identifiers into the top level:

```
import counter;
⇒ (':Counter' defined)
   ('counter' defined)
```

This command links the module *counter* and imports its identifier *counter* into the top level. Notice that also the name of its interface *Counter* is made accessible. More precisely, the import of a module forces the import of all modules and interfaces transitively imported by its interface. This import is necessary to type-check further access to the module value via top-level commands.

The import of an interface X with export signatures S is equivalent to the declaration ***Let** X = **Tuple** S **end***. The import of a module *x* with interface *X* defines a value binding with signature *x :X*. Therefore, access to exported module components requires qualification by the module identifier:

```
let c = counter.new(2);
counter.dec(c);
counter.dec(c);
counter.isZero(c);
⇒ true :Bool
```

A *open* expression can be used to access module components without qualification.

```
import counter;
open counter;
let c = new(2)  dec(c)  dec(c)  isZero(c);
⇒ true :Bool
```

The disadvantage of *open* is that the top level will be *polluted* with names that may hide other bindings.

2.5.6 Organizing Source and Object Files

In order to support the exchange of compiled library definitions, interfaces and (unlinked) module code between autonomous object stores, all information derived during compilation of a library component is also stored outside the persistent object store in operating system files. The following naming conventions are imposed by the compiler. The base directories *librarySourceDir*, and *libraryObjectDir* can be defined via ***do*** *set* commands.

Directory	**Files**	**Contents**
librarySourceDir/L/	*L*.tl	source code for library definition *L*
	X.ti	source code for interface *X* in library *L*
	x.tm	source code for module *x* in library *L*
libraryObjectDir/L/	*L*.tl.x	compiled library definition for library *L*
	X.ti.x	compiled interface *X* in library *L*
	x.tm.x	compiled module *x* in library *L* (portable)

3 Layout and Naming Conventions

Common formatting conventions are worth a lot, especially when several people work together on large projects. In addition to the communication inefficiencies caused by differing conventions, newcomers to a programming language often spend a significant amount of time incrementally developing and retrofitting their own style, usually relearning what turn out to be simple lessons that others have already learned. While this is not always wasteful, it is clearly worthwhile to have a good set of guidelines at hand, if only for reference. Also adherence to common conventions makes automatic formatting tools easier to provide and more useful [RLW85].

This section offers a complete set of conventions for formatting TL modules and interfaces. Such conventions address indentation, capitalization, punctuation, comments, etc. The following points of style produce a visually pleasing program. Consistently applied, they also provide syntactic cues to semantics that make a program easier to read.

Furthermore, a subsection presents naming conventions for TL programs. To simplify the usage of the TL libraries it is necessary to standardize the names of value, type, function and exception identifiers.

3.1 Spelling

Identifiers that name *values* (e.g. variables, functions, modules and exceptions) start with a lower-case character and identifiers that name *types* (e.g. type operators and interfaces) start with an upper-case character. All following characters are entirely lower case, except for composed identifiers. Each first character of a subcomponent is capitalized (e.g. *longNameForAValue*).

Reserved keywords that are used in *value contexts* are entirely lower-case and keywords that are used in *type contexts* start with an upper-case character.

3.2 Punctuation

A space (␣) appears before and after a binary operator in infix notation and in a **let**-binding or destructive assignment.

```
3␣+␣4  "con"␣<>␣"cat"
let a␣=␣3
a␣:=␣5
```

A space appears before but not after a colon or a subtype sign.

```
let p␣:Person = ...
let n␣<:Ok = ...
```

A space appears after but not before a comma or a semicolon.

```
add(x,␣y :Int) = . . .
module . . . end;
```

A space appears neither before nor after a point, a question-mark or an exclamation-mark.

```
person.name
address?national
address!national
```

Except as required by adjacent tokens, no spaces appear before or after left and right parenthesis, left and right sqare brackets and left and right curly brackets.

```
fac(3)  get(peter).name
a[3]  p[3].name
{3 + 4}
```

Two spaces appears between two signatures, e.g. in function or tuple signatues. Two spaces also appears between **let**-bindings function applications or tuple components.

```
get(E <:Ok␣␣coll :T(E)␣␣index :Int) :E
Tuple name :String␣␣age :Int end
f(let x = 2␣␣let y = 7)
tuple let a = 6␣␣let b = "hallo" end
```

A single space appears between actual parameters in function applications and between tuple components (anonymous bindings).

```
get(:Person␣persons␣7)
tuple "Peter"␣29 end
array 1␣2␣3 end
```

3.3 Indentation

Indenting is used to emphasize program structure. Each nesting level is two spaces wide. A statement sequence is indented under the construct that introduces it. Section 3.6 gives indentation examples for all TL syntatic forms.

These forms only apply to contructs that do not fit on a single line. For example, if the statement sequence following a **then** or **else** fits on a single line, it can appear on the same line with the tokens that introduce and terminate it. Similary, if the statement sequence of a loop body is short, it can be moved up to the line that introduces the loop, along with the trailing **end**.

```
if z > 10 then x := z  y := z end
for i = 1 upto 10 do a[i] := 0 end
```

3.4 Comments

The text of a multiline comment begins on the same line as the opening left-comment. Subsequent lines are indented the same as the first word of the comment. The terminating right-comment appears on the last line of the comment.

```
(* A comment that fits entirely on one line by itself. *)

(* A long comment that does not fit on one line. A long
    comment that does not fit on one line. A long comment
    that does not fit on one line. *)
```

By convention, comments which refer to a group of items appear before the group. Comments associated with a single definition or declaration appear immediately after it. This comments starts at the same indentation level as the definition or declaration begins.

```
...
(* Exceptions:
   This is a comment for a group of items. *)

error, overflow :Exception with data :Int end
(* Standard exceptions. *)

test :Exception    (* Only for debugging. *)
...
```

Interfaces and modules have a multiline comment with predefined fields. This comment is placed immediately after the interface respectively the module name. It is used to give some initial information about the contents. The terminating right-comment stands on a separate line.

```
interface Editor
(* System:  editenv
   File:     Editor.ti
   Author:   Florian Matthes, Sven Muessig
   Date:     02-dec-91
   Purpose:  Generic data editor and browser.
*)
import
   ...
export
   ...
end
```

3.5 Naming

This section gives guidelines how to name variables, types, functions and exceptions.

variables: Locally used variables are named like the first character of its type, like *i* for an integer or *s* for a string variable. The iteration-variable in **for**-loops is named *i*.

types: The exported type of a library module is named *T*, for example *int.T*.

functions: A large number of functions has a *historically* grown meaning (See also the files of *StdLib*). These functions are presented in the following table.

Function	**Description**
copy	create a shallow copy
create	create a collection by an enumeration of its components
default	default value
elements	iteration over elements of a collection
empty	test for emptiness
equal, ==	test the equality of two values
fmt	conversion to string
free	deallocation of volatile resources
get	access by index
hash	hash function
new	create a new object possibly with local state
nil	empty element
notEqual, !=	negation of *equal*, ==
order	defines a linear order
set	destructive update by index
setSub	subcollection update by index range
size	size of a collection
sub	subcollection selection by index range

exceptions: If a module provides exception handling, the major exception is named *error*, for example *int.error*.

3.6 Indentation Examples

The following sections illustrate the recommended form of the TL constructs. A statement sequence (*ss* in the following examples) is indented under the construct that introduces it, which lines up vertically with its corresponding **end**.

```
let a = 4                                      Let T = Int
let a = 123 / 3  and var b = a + 2             Let T = String and U = T

let t = tuple "Peter" 3 end                    Let Person = Tuple :String :Int end

let t =                                        Let Person =
  tuple                                          Tuple
    let name = "Peter"                             name :String
    let age = 3                                    age :Int
  end                                            end

let address =                                  Let Address =
  tuple case national of Address                 Tuple
    let street = "Johnsallee 21"                   street, city :String
    let city = "Hamburg"                           case national with
    let zip = 21234                                  zip :Int
  end                                              case international with
                                                     state :String
                                                     zip :String
                                                 end

let paul :Student =                            Let Student =
  tuple                                          Tuple
    open peter                                     Repeat Person
    let semester = 6                               semester :Int
  end                                            end
```

```
let rec fac(n :Int) :Int =                     Let Fac = Fun(n :Int) :Int
  if n == 0 then 1 else n * fac(n - 1) end
                                               newSubWindow(client :Canvas.T
let swap(A <:Ok var x, y :A) :Ok =                          label :DisplayItem
  begin                                                     dismiss, unpin :Action.T
    let temp = x                                            super :Window.T) :Popup.T
    x := y
    y := temp
  end
```

```
setisum(x + y            iter.enum of
        z                          1
        a)                         2
                                   3
                                 end
```

```
begin               if bool then          case address
  ss                  ss                  when national with n then
end                 elsif bool then         fmt.int(n.zip)
                      ss                  else
loop                else                    "not national"
  ss                  ss                  end
end                 end


while bool do       if bool               typecase type
  ss                andif bool            when Int then fmt.int(7)
end                 orif bool then        when String then "abc"
                      ss                  else "???"
for i = x upto y do else                  end
  ss                  ss
end                 end


let noCredit =                            try
  exception "No Credit" with od :Int end    withdraw(petersAccount  300)
                                            print.string("Transfer succeded")
                                          when noCredit with exc then
                                            print.string("Overdrawn")
                                          else
                                            print.string("Unexpected exception")
                                          end
```

```
interface name        module name         library name
import                import              import
  importList            importList          importList
export                export              with
  ss                    ss                  library
end                   end                     list
                                            interface
                                              list
                                            module
                                              list
                                            hide
                                              list
                                          end
```

4 Overview of the Tycoon Libraries

A significant part of the Tycoon system functionality is defined in an open set of libraries. Currently, the following libraries are available:

[*** xx Gerald to update based on library definitions and DBIS.bib ***]

Library	Description	Documentation
stdenv	basic data types and operations	
machineenv	low level functions	
bulkenv	bulk data types	
compenv	compiler toolkit	
dbenv	transaction and integrity support	
sqlenv	interface to SQL databases	
commenv	communication support (RPC)	
newsenv	interface to The NeWS Toolkit (TNT)	
editenv	generic data visualization	

As indicated by the references in the last column, for some of these libraries additional documentation is available as German master's theses.

5 Resolving Cyclic Import Relationships

The current Tycoon library concept does not allow libraries with cyclic import relationshps. The main problem with cyclic imports is that the compiler should ensure that the module initialization code of a module M only accesses modules M_i that are already properly initialized. This check is analogous to the checks performed by the TL compiler on recursive value bindings.

In the following, a systematic approach to circumvent this limitation is described.

Suppose the following recursive module system is to be implemented:

```
library Test
  interface A, B
  rec module a:A b:B
end

module a import b export let f(): Int = b.g() end
module b import a export let g(): Int = a.f() end
```

First, it is necessary to determine a correct nitialization sequence. For example, if a has to be initialized prior to *b*, one has to write:

```
library Test
  interface A', B
  module a:A'
  module b:B
end
```

where *A'* is defined as follows:

```
Interface A'
export
  (* All signatures of interface B needed in module a prefixed with "var": *)
  pre : Tuple
    var g() :Int
  end

  (* All remaining declarations of A without changes: *)
  f() :Int
end

module a (* import of b no longer required here! *)
```

```
export
  let pre = tuple
    let var g() :Int = raise "module b not yet initialized"
  end

  let f() = pre.g()
end

module b import a
export
  (* All bindings of module b without changes: *)
  let g(): Int = a.f()

  (* Fixup of forward-references in all other modules: *)
  a.pre.g:= g
end
```

6 Using Dynamic Types

This section describes how to use dynamic types in the current TL language implementation. It also gives some background information on the rationale behind these TL language mechanisms.

6.1 Motivation

Programmers of data-intensive and long-lived applications are faced with situations that cannot be handled with static type checking alone:

1. The inspection of type information at run time is needed, for example, to write a schema browser or to dynamically generate a type-specific user-interface for a complex run-time data value.
2. The manipulation of values with a dynamic type that cannot be determined statically at compile time is necessary whenever values generated outside the static scope of the type checker have to be manipulated by strongly typed code. For example, the type of a data value retrieved from disk or recieved via a communication channel from a different network node has to be checked at run-time against the type information utilized at compile-time.

It is important to note that most language proposals for dynamic typing only address issue (2) but not issue (1). In such languages it is not possible to inspect a type without having a run-time instance of that type available.

6.2 The Initial Proposal for Dynamic Typing

The original TL language report [MaSc92] defines the following language constructs to address the issues raised in the previous section:

- The signature of a type variable T can be prefixed by the keyword **Dyn**. Such a type variable can be inspected (like a value variable) at run time to obtain information about the actual type bound to T.
- Type variables can be inspected using a **typecase** expression. In the branches of a **typecase** expression, the type variable is narrowed to a statically-known type. Therefore, it becomes possible to statically type-check the access to values of that dynamic type.

For example, the following polymorphic function *f* receives a dynamic type representation *T* and a value x of that type as its argument. The result type of the function is statically known to be identical to the type of the argument. Furthermore, in each of the typecase branches, the type *T* is known to be a subtype of the type in its when clause.

```
let f(Dyn T<:Ok x:T) :T =
  typecase
  when Int then x+1
  when String then x <> "Test"
  else a
  end
```

The first TL compiler already implemented the type rules necessary to type check **typecase** expressions and to verify the matching of signatures and bindings including **Dyn** keywords. However, the run-time support for these language constructs required a bootstrap of the TL compiler to re-use the subtype test algorithm of the compiler also for the run-time subtype tests.

During the implementation of this run-time support code the following limitations of the original approach became clear:

- The facilities of the typecase expression are not sufficient for highly generic type-directed (reflective) programming techniques where it is necessary to dynamically explore the structure of a type representation using algorithmically-complete code. The typecase expression is essentially limited to programming situations where a dynamic type is verified to be a subtype of a fixed static type.
- Due to the higher-order nature of Tycoon, it is desirable to be able to inspect types (Int, Person, PersonList), type operators (Array, List, Bag) and higher-order type operators (e.g., mapping type operators to other type operators) in a uniform way. However, TL does not provide a kind specification to quantify over the universe of types and higher-order type operators. For example, in TL one has to write different functions to inspect types (subtypes of the top type **Ok**), unary type operators (subtypes of the type ***Oper****(X<:****Nok****)* ***Ok***), and binary type operators (subtypes of the type ***Oper****(X<:****Nok*** *Y<:****Nok****)* ***Ok***):

  ```
  let inspectType(Dyn T<:Ok) = ...
  let inspectTypeOper1(Dyn T<:Oper(X<:Nok) Ok) = ...
  let inspectTypeOper2(Dyn T<:Oper(X<:Nok Y<:Nok) Ok) = ...
  ```

  Instead of this one would like to write a single generic piece of code with an appropriate signature, e.g.

  ```
  let inspectAnyType(Dyn T<: ???) = ...
  ```

- The code generation for dynamic type variables (passed as parameters, bound to local variables or embedded as components of aggregates) introduces quite some hidden complexity into the TL language processors.

Following the minimalistic add-on approach of Tycoon, we therefore decided to make the management of type representations at run-time much more explicit and factor it out into ordinary TL libraries.

6.3 The New Approach to Dynamic Typing

Support for the inspection of type information at run time (task 1 described above) is provided by the module typeRep defined in the Tycoon standard library *tl_reflective*. It exports the following (semi-abstract) types and functions

- The type *T* is the type of a run-time type representation for an arbitrary Tycoon type or (higher-order) type operator. A type representation is a first-class-value representing a static type. Contrary to static types, type representations do not need an environment in which they are valid.
- The type *Expansion* gives a more detailed description of a type representation.
- The function *inspect(type :T) :Expansion* returns more detailed information on a type that is represented by a type representation. It returns a one-level unfolding of a type representation.
- A number of type representations is provided that match the standard TL types defined in the initial environment (*Int, String, ...*).
- A number of constructor functions support the dynamic creation of type representation for structured types (arrays, ...).

The function to construct a type representation for a given static type cannot be exported by the add-on module *typeRep* since its implementation requires special compiler support to access static type descriptions. Therefore, this function is not named *typeRep.new*, but *typeRep_new* and it is available without explicit import at the top level and in every module.

As explained in the previous section, the signature of this function cannot be expressed completely in the TL type system:

typeRep_new(T <: ???) :typeRep.T

Here are some examples how to create values of type typeRep.T.

typeRep_new(:Int) *(* builtin type *)*
*typeRep_new(:**Tuple** :String :Real **end**)* *(* structured type *)*
*typeRep_new(:**Fun**(:String) :Int)* *(* function type *)*
typeRep_new(:Array(Int)) *(* type operator application *)*
typeRep_new(:list.T) *(* type operator *)*
*typeRep_new**Oper**(**Oper**():**Ok**):**Ok**)* *(* higher-order type operator *)*

Support for the manipulation of values with a dynamic type (task 2 described above) is provided by the module *dynamic*, also located in the library *tl_reflective*. This module exports types and functions on dynamic values, such as:

- The type *T* is the type of dynamic values in TL. A dynamic value is a pair consisting of a value *x* of type *T* and a type representation *t_RT* for *T*. The module *dynamic* ensures that each access to *x* respects the type *T*.
- The type *Expansion* gives a more detailed description of a dynamic value.
- The function *inspect(value :T) :Expansion* performs a one-level unfolding of a structured value. This function is particularly useful in cases where the programmer does not have any knowledge of the static type of a dynamic value.
- The function *typeOf(value :T) :typeRep.T* returns the type representation component of a dynamic value.
- A number of constructor functions is provided that return structured dynamic values.

In addition to these library functions there are two predefined functions that require compile-time support by the TL language processors. These functions are available without explicit import at the top level and in every module:

- The constructor *dynamic_new(T<:**Ok** d :T) :DynValue* takes a value *d* of a closed type *T* and returns a new dynamic value consisting of a type representation for *T* and the value *d*.

  Here are some examples how to create dynamic values:

  ```
  dynamic_new(3)                   (* integer value *)
  dynamic_new(tuple 3 4 end)          (* structured value *)
  dynamic_new(fun(x :Int) :Int x + 1) (* function value *)
  dynamic_new(array 1 2 3 end)        (* built-in parameterized type *)
  dynamic_new(list.new(:Int))        (* user-defined parameterized type *)
  ```

- The function *dynamic_be(d :dynamic.T T<:**Ok**) :T* takes a dynamic value *d* and a type variable *T* which has to be a closed type and returns *d*'s value component. A run-time exception (*dynamic.error*) is raised if the value component's static type is not a subtype of *T*. This function provides the core functionality of the **typecase** construct of the original TL language design.

Here are some examples how to extract the value component of a dynamic value:

```
let i = dynamic_new(3)
let a = dynamic_new(array 1 2 3 end)
let t = dynamic_new(tuple 1 2 3 end)
dynamic_be(i :Int)            (* ⇒ 3 *)
dynamic_be(a :Array(Int))        (* ⇒ array 1 2 3 end *)
dynamic_be(t :Tuple :Int end)     (* ⇒ tuple 1 end *)
try
  dynamic_be(i :String)          (* ⇒ raises dynamic.error *)
when dynamic.error then
  "error occured"
end
```

6.4 Current Restrictions

Naming issues: Due to historic reasons, the identifiers and keywords supported by the current TL compiler still differ from the names used in the previous section:

- The module *dynamic* is currently called *dynamics*.
- The type identifier *typeRep_T* is currently called *DynType*.
- The value identifier *dynamic_T* is currently called *DynValue*.
- The value identifier *dynamic.error* is also available through the name *typecaseError* in the initial environment.

However, the next public release of the system will use the names described in this text.

Type unification variables: The type of a dynamic value may not contain type unification variables introduced by the TL type checker during type argument synthesis. For example, it is not possible to package the polymorphic empty list directly into a dynamic value:

```
dynamic_new(list.new())          (* ⇒ compile-time error *)
```

Instead of this, one has to explicitly instantiate the type parameter of the list.new function:

dynamic_new(list.new(:Int)) *or*
dynamic_new(list.new(:String))

However, explicit polymorphic functions (like the polymorphic identity function) can be turned freely into dynamic values.

*dynamic_new(**fun**(A<:**Ok** a:A) a)*

Universally quantified type variables: The type of a dynamic value may not contain (free) universally quantified type variable, for example,

***let** f(T <:**Ok** x :T) :dynamic.T = dynamic_new(x)*

is rejected at compile-time since the type of *x* depends on the free, universally quantified type variable *T*. Alternatively, the compiler could construct a dynamic value with a type representation describing the statically-known bound type **Ok** of *T*, but we prefer to warn the programmer that this function does not exhibit the desired run-time behavior.

To obtain the desired behaviour, the keyword **Dyn** has to be added in front of the type variable *T* to ensure that at run-time a representation of the actual type of the value bound to *x* is available:

***let** f(**Dyn** T <:**Ok** x :T) :dynamic.T = dynamic_new(x)*

7 Programming with External C Libraries

Tycoon provides a bidirectional programming interface between TL and C that features a seamless integration of both languages' function paradigms. External C functions can be integrated into TL as ordinary function values. TL functions can be wrapped in a way that makes it possible to use them as C function pointers.

7.1 Function Calls from Tycoon to External C Libraries

Tycoon provides a generic mechanism to use system functionality implemented in external languages. The binding of TL identifiers to external function values is achieved by the predefined function *bind*. This function has the following signature:

*bind(Function <:**Ok** library, label, format :String) :Function*

The parameters of the *bind* function have the following meaning: *Function* describes the type of the resulting TL function. It has to be of the form ***Fun**(...) :A*. The *library* parameter is a string that identifies the library file that contains the required external C function. This can either be the full path name[3] of a dynamic library or the string result of one of the functions exported by the module *runtimeCore* (see the following table) belonging to the Tycoon library *stdenv*.

Function	**Description**
library	identifies the core of the Tycoon runtime system
cLibrary	identifies the standard C library
dynamicLibrary	identifies dynamically bound libraries
staticLibrary	identifies statically bound libraries

[3]If the same shared library is referenced several times the path should always be exactly the same. Otherwise the dynamic linker loads several instances of the shared object. This means not only consuming more process memory than necessary, but leads to subtle bugs when global C variables are defined multiple times in the same process.

The *label* parameter is a string that contains the original C source text name of the C function. The *format* parameter is a string that specifies the assumed parameter format of the C function. Every single character of this string corresponds to one parameter. It specifies the conversions between tagged and untagged data representations to happen before and after a call. The parameter order is *from left to right* like in C, except for the function result type. It is given by the last character that is mandatory. The following table contains the set of characters that denote parameter formats.

Format	**Tl type**	**C type**	**Description**
i	*Int*	**long**	integer number
r	*Real*	**double**	floating point number
c	*Char*	**char**	ASCII character
b	*Bool*	**long**	boolean value (see text)
v	**Ok**	**void**	return value only
=	<:**Ok**	**void** *	Tycoon value, no conversion
&	*address.T*	**void** *	memory address (see text)
s	*String*	**char** *	zero-terminated string
w	*word.T*	**void** *	32-bit word

There is no predefined type for boolean values in C. Boolean TL values are converted to C long values as follows.

true $\rightarrow$ *1*
false $\rightarrow$ *0*

A C value x produces the following TL boolean values:

x != 0 $\rightarrow$ *true*
x == 0 $\rightarrow$ *false*

When *s* is used as a format character for a parameter, it acts exactly like &, because all Tycoon strings are represented with zero-termination. Specifying *s* instead of & means just to provide a little documentation. In case of return values this is different. Stings returned from C are copied into newly created store objects (*copy-out*).

For Tycoon store objects (strings, tuples, arrays, etc.) that are passed to C by using the format character &, every call is enclosed by automatic fix and unfix operations for the argument. The result of the fix operation is passed to C.

Suppose a library */usr/lib/libexample.so* contains a function *example* that takes a string argument and returns a 32 bit integer number. Thus, *example* can be assumed to match the following declaration:

extern long *example(***char** **s);*

An appropriate binding for *example* in TL is:

let *cCallExample =*
  *bind(:***Fun***(:String) :Int "/usr/lib/libexample.so" "example" "si")*

The value *cCallExample* has the type **Fun***(:String) :Int*. Therefore, the C function can be called as follows.

let *result :Int = cCallExample("My favorite String")*

Note that external bindings are persistent and portable across host architectures. For example, if the value *cCallExample* is transferred with *dynamic.extern/intern*, the C binding would be reestablished automatically.

7.2 Function Calls from External C Libraries to Tycoon

The programming interface between TL and C is bidirectional. It is not only possible to call C functions from TL, but also to *call back* from C to TL.

In order to minimize the programming effort for callbacks on the C side, it is desirable to make TL functions appear like ordinary C function pointers. Moreover, this is indispensable in situations where an external software component requires C callbacks but cannot be changed.

The module *cCallback* exports an abstract type *cCallback.T* which represents C function pointers that refer to callbacks. The creation function for values of type *cCallback.T* has the following signature:

```
new(Function <:Ok  function :Function  format :String) :T(Function)
```

The first value argument (*function*) of *new* must always be a function, although this restriction is not checked. Nevertheless, the function's signature has to be mirrored in the *format* string that specifies how C arguments of the resulting callback are converted into TL values. Every single character of the string corresponds to one parameter. The parameter order is *from left to right* like in C, except for the function result type which is given by the last character. The latter is mandatory. The format characters are shown in the table in section 7.1.

If the format character *s* is used for a parameter, a C string argument will be copied into a newly created store object (*copy-in*). In case of a return value, a C string is copied into a chunk of memory allocated by *malloc*. Hence the parameter passing semantics are *copy-out* instead of *call by reference* which apply when the format character *&* is chosen.

The format character *&* does not apply to function return parameters.

The format character *v* specifies a function result value of **ok** irrespective of the actual C value returned by C. For

A simple example follows:

```
import cCallback fmt

let myMessage(n :Int  r :Real) :String =
  fmt.int(n) <> " = " <> fmt.real(r)

let myMessageCallback =
  cCallback.new(myMessage  "irs")

let test =
  bind(:Fun(n :Int  messageCallback :cCallback.T) :Ok
      ".../example.so.1.0"  "test"  "i&v")

test(2  myMessageCallback)
```

The resulting console output would be

```
"pi * 2 = 6.28"
ok
```

assuming that the corresponding C program *.../example.c* looks like this:

```
#include <stdio.h>
```

```
void test(long n, char *message(long n, double r))
{
  printf("pi * %s\n", message(n, n * 3.14));
}
```

As callbacks can be transferred to address spaces which are not under control of the Tycoon system, there is in general no way to determine their temporal extent automatically. Callbacks are never persistent. Callbacks occupy some memory resources that can only be released explicitly:

```
cCallback.free(myMessageCallback)
```

After freeing a callback it is invalid. Any subsequent usage is most likely to cause strange system behaviour (e.g. crashes). Attentive readers may have noticed some more problems in the example: In what manner does the result string of the function *message* get allocated on the C side before it is passed to *printf*, who is in charge of releasing its memory and how can this be done?

The current solution is that the format character *s* in a return value position causes the allocation of an appropriate memory block by calling *malloc*. This block has to be released by the C programmer by a call to *free*. Thus, the C program in the example should be written as follows:

```
#include <stdio.h>
#include <malloc.h>

void test(long n, char *message(long n, double r))
{
  char *p;

  p = message(n, n * 3.14);
  printf("pi * %s\n", p);
  free(p);
}
```

References

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

[Mat95] F. Matthes. Higher-order persistent polymorphic programming in Tycoon. In M.P. Atkinson, editor, *Fully Integrated Data Environments.* Springer-Verlag (to appear), 1995.

[MMS94] F. Matthes, S. Müßig, and J.W Schmidt. Persistent polymorphic programming in Tycoon: An introduction. FIDE Technical Report FIDE/94/106, Fachbereich Informatik, Universität Hamburg, Germany, August 1994.

[MMS95a] B. Mathiske, F. Matthes, and J.W. Schmidt. On migrating threads. In *Proceedings of the Second International Workshop on Next Generation Information Technologies and Systems, Naharia, Israel,* June 1995. (Also appeared as TR FIDE/95/136).

[MMS95b] B. Mathiske, F. Matthes, and J.W. Schmidt. Scaling database languages to higher-order distributed programming. In *Proceedings of the Fifth International Workshop on Database Programming Languages, Gubbio, Italy.* Springer-Verlag, September 1995. (Also appeared as TR FIDE/95/137).

[MS92] F. Matthes and J.W. Schmidt. Definition of the Tycoon language TL – a preliminary report. Informatik Fachbericht FBI-HH-B-160/92, Fachbereich Informatik, Universität Hamburg, Germany, November 1992.

[MS94] F. Matthes and J.W. Schmidt. Persistent threads. In *Proceedings of the Twentieth International Conference on Very Large Data Bases, VLDB,* pages 403–414, Santiago, Chile, September 1994.

[MSW95] F. Matthes, J.W. Schmidt, and J. Wahlen. Using extensible grammars to support data modeling. In M.P. Atkinson, editor, *Fully Integrated Data Environments.* Springer-Verlag (to appear), 1995.

[RLW85] P. Rovner, R. Levin, and J. Wick. On extending modula-2 for building large, integrated systems. Technical Report 3, Digital Equipment Corporation, Systems Research Center, Palo Alto, California, January 1985.

[Tyc92] WWW home page for the Tycoon project. http://idom-www.informatik.-uni-hamburg.de/Projects/Tycoon/entry.html, 1992.