Chapter 1.1.1
Higher-Order Persistent Polymorphic Programming in Tycoon

Florian Matthes

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

Summary This text provides an introduction to Tycoon[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 TL[2] 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 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

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

(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).

A more detailed discussion of the rationale behind Tycoon is given in [14] and [18]. Readers interested in the formal definition of the TL semantics are referred to [17].

This text is organized as follows: Section 2 gives a quick overview of the Tycoon language in comparison with other modern (persistent) programming languages. The subsequent sections (Section 3–8) provide a step-by-step introduction to Tycoon's language concepts in a functional setting (values, types, bindings, signatures, predefined type and value constructors, user-defined types and type operators, subtype and parametric polymorphism). Section 9 explains how these concepts interact with the imperative concepts of Tycoon, namely mutable variables, destructive assignment, sequential execution and exception handling. Section 10 and 11 discuss alternative approaches to the structuring of large Tycoon software systems into interfaces, modules and libraries. Section 12 and 13 present some important system-oriented aspects of Tycoon like its transparent persistence management and bindings from and to external C libraries.

2. Language Classification

This section is intended primarily for readers who are familiar with the state of the art in programming language research and who are interested in a rough TL language classification.

The programming language TL evolved from the experimental languages Quest[3] [5, 6] and P-Quest[4] [21, 15]. All semantic concepts of these languages are supported in TL(in a slightly varied syntactic from). TL eliminates some ad-hoc restrictions of Quest's language orthogonality. Furthermore, it introduces new language concepts such as subtyping between type operators, recursive type operators, extensible record values, and libraries as scopes for modules and interfaces.

The syntactic structure and the module concept of TL are similar to those of the languages of the Modula family (Modula-2 [12], Oberon [26], Modula-2+ [23], Modula-3 [22], and Ada [11]). Regarding its semantics, TL is more closely related to the polymorphic functional languages of the ML language family [5, 6, 19, 8, 10]. The semantic concepts of TL are derived from the language $F_{\leq}$ [7], a widely accepted formal basis for the study of modern type systems.

Like C [13] TL is intended for application programming and for system programming tasks. By virtue of its polymorphic type system TL can also be utilized as a data modeling language. In this respect, TL resembles Lisp development systems [3] and commercial object-oriented languages like Smalltalk [9]. From integrated database programming languages like PS-Algol [2], Napier88 (see Chapter 1.1.3), Amber [4], and P-Quest, mentioned before, *Tycoon* inherits the orthogonality of elementary kernel concepts for persistence abstraction, type-complete data structuring, and iteration abstraction [1, 24].

Motivated by an analysis of the conceptual and technological foundations of existing database languages (see Chapter 1.4.2), the *Tycoon* system pursues the idea of a strictly reduced kernel language supporting naming, binding and typing of predefined semantic objects (variables, functions, type variables, type operators). On the other hand, it is possible to extend the language kernel with external semantic objects (integers, floating-point numbers, strings, arrays, relations, views, files, windows, etc.) and generic functions associated with these objects in a completely type-safe way (*add-on* vs. *built-in*) [16].

TL enables the programmer to use different modelling styles. *Functional* and *imperative* programming are supported directly. Due to the linguistic neutrality several variants of the *object-oriented* programming style are supported by TL. *Relational* and *logic-based* programming [20] are not supported directly, since unification-based evaluation models and declarative approaches deviate strongly from functional and imperative structures.

The *Tycoon* system offers an interactive programming environment. Such environments are known from functional systems (ML, Lisp). This distinguishes the *Tycoon* system from conventional translation systems like, for example, C, Modula-2, or Ada compilers. Due to the interactive environment, ad-hoc TL database queries are possible in addition to the use of TL as a database programming language. The persistence concept enables the user to perform incremental system development spanning several sessions. At the same time, the library concept of TL supports the controlled use of shared data and programs by several users.

[3] Quest: *Quantifiers and Subtypes*.
[4] P-Quest is a Quest System extended by an orthogonal persistence concept

3. Lexical and Syntactical Rules

This section introduces the most important lexical and syntactical rules of TL for the construction of symbols, reserved identifiers, and productions.

3.1 Symbols

The character set predefined on a given system is partitioned into the disjoint classes of letters, digits, delimiters, printable special symbols, and non-printable formatting characters. On the basis of this classification a sequence of characters is divided into atomic symbols (e.g., numbers and identifiers).

TL distinguishes *alphanumeric identifiers* and *infix symbols*. Alphanumeric identifiers consist of a character followed by a sequence of characters and digits whereas infix symbols are composed solely of special symbols. A space is only required between two alphanumeric identifiers or two infix symbols that appear in direct succession.

3.2 Reserved Keywords

Reserved keywords and infix symbols of must not be used as user-defined identifiers or infix symbols.

The reserved keywords are written in **bold face** in the programming examples. This facilitates the distinction from the rest of the symbols in the examples which are presented in *italics*.

Keywords associated with types start with a capital letter whereas all other keywords begin with lower case characters. It is advisable to adopt this rule as a convention for all other identifiers to improve the readability of programs.

3.3 Comments

In TL, comments are enclosed between *(** and **)*. Arbitrary nesting of comments is possible. Comments may include arbitrary (printable and non-printable) characters and can span several lines.

(This is a comment. *)*

3.4 Factoring of Expressions

Operators represented by infix symbols are left-associative and of equal precedence. The parsing of type and value expressions containing infix symbols can be controlled by the use of curly brackets. Consequently, the following expressions are equivalent.

```
3 – 7 * 4
{3 – 7} * 4
{*(–(3 7) 4)}
int.mul(int.sub(3 7) 4)
⇒ –16 :Int
```

Infix symbols starting with a colon (e.g., *:=* and *:+*) have a weaker precedence than the other infix symbols. These operators are also left-associative. The bracketed expressions in the following examples show the factoring of the corresponding expressions without brackets.

x := a + b
x := {a + b}

x := y := z
{x := y} := z

3.5 Coercion and Overloading

Automatic coercions, e.g., of integers to real numbers, are not performed in TL. For example, a type error is caused by the following expression.

3.0 + 4
⇒ Argument type mismatch: '_builtin.Int' expected, 'Real' found
[while checking function argument '<anonymous>']

Neither symbolic nor alphanumeric identifiers may be overloaded. For this reason it is, for example, necessary to have different operators (infix symbols) for the addition of integers and real numbers, respectively.

2 + 7
⇒ 9 :Int
2.4 ++ 3.8
⇒ 6.2 :Real

4. Predefined Values and Functions

This section presents the basic semantic rules of TL.

4.1 Naming, Binding, and Typing

Contrary to many traditional programming languages, neither the base types, nor their constants, nor the functions defined on them are predefined in TL. The identifiers of the base types (e.g., *bool.T*), the constants of the base types (e.g., *bool.true* and *bool.false*), and the functions defined on the base types can be imported explicitly from modules of the standard *Tycoon* library. They obey the same syntax, typing, and evaluation rules as user-defined types, values, and functions. The rationale behind this approach is to give predefined and user-defined data types equal status in the language.

In order to avoid the notational disadvantages resulting from this approach, the base types and many functions defined on the base types are bound to symbolic identifiers and infix symbols, respectively, in an initial context that is defined when the system is started. Thereby, the identifiers appear to be built into the *Tycoon* system environment without including them into the language TL. In the following sections we, therefore, use the phrases 'predefined base types' and 'predefined functions'.

4.2 Literals

The following enumeration lists examples of literal values of the base types *Int*, *Real*, *Char*, *String*, and *Bool*, respectively, from left to right.

3 ∼3 3.0 'c' "string" true

Note that TL avoids overloading. For this purpose negative integer numbers are marked by an prefix "∼". The symbol "−" is reserved for integer subtraction.

5. User-defined Values and Functions

The *binding* of a user-defined *identifier* to a semantic object and the repeated use of this identifier in *expressions* denoting the bound object is a basic concept in TL. Furthermore, a *signature* assigns static type information in the form of a *type expression* to an identifier. A signature restricts the set of possible semantic objects that can be bound to an identifier. This makes it possible to control the correct use of identifiers in expressions [14].

In this section, the discussion of naming and scoping concepts is restricted to value bindings. The orthogonal extension of these concepts to type bindings, presented in section 6, gives rise to much of the expressive power of TL.

5.1 Static Bindings

Static value bindings in TL are defined as follows.

let *n* = *10*

After evaluating the term, the variable *n* is statically bound to the value *10*. Every subsequent use of the identifier *n* in an expression evaluates to the bound value.

let x = *1* + {*2* * *n*}
⇒ *21 :Int*

Sequences of bindings are interpreted as *sequential* bindings in TL.

let *n* = *10*
let x = *1* + {*2* * *n*}

The identifier *n* used in the second binding, therefore, refers to the binding *n=10* established in the first line of the example above. In order to achieve a *simultaneous* binding, the single bindings have to be connected by the keyword **and**.

let a = *4*
let a = *123* / *3* **and** *b* = a + *2* **and** *c* = *true*

The variable *b* is bound to the value *6* in this expression. The associated binding for an expression is determined by static *scoping rules* in TL.

let a = *1.0*
begin let a = 'x' **let** *b* = a **end**
let *c* = a

The scope of the local identifiers a and *b* is restricted to the block delimited by the keywords **begin** and **end** (see section 9.3.1). For this reason, the identifier *c* is bound to the value *1.0* denoted by the global identifier a, whereas the local identifier *b* is bound to the value *'x'* denoted by the local identifier a.

The bindings described above are determined by two basic scoping rules: local declarations have precedence over global declarations and an identifier in an expression always refers to the last binding established for this identifier.

A block in TL evaluates to the value of its last binding. This is illustrated by the following example.

begin let a = *3* **let** *b* = *true* **end**
begin let a = *3* *true* **end**
begin *3* *true* **end**
begin end

Evaluation of the first three blocks yields the value *true* whereas the result of evaluating the last block is the canonical value **ok** of type **Ok**. The second and third example contain so-called *anonymous* bindings, i.e., bindings without an identifier.

Signatures assign static type information to bindings; they are ordered sequences of pairs each consisting of an identifier and a type. The signatures of the bindings established by the previous example are considered as illustrative examples.

a *:Int* *b* *:Bool*
a *:Int* *:Bool*
:Int *:Bool*
(empty signature *)*

It is possible to declare the type of the bound value in a binding explicitly. This declaration is optional. If the type specification is omitted, it is inferred by the compiler from the expression given in the binding.

let a *:Int* = *3*
let *b* *:Bool* = *true*

Recursive bindings are used for the construction of *recursive* and *cyclic* data structures. In TL, pointer types are not necessary for this purpose. Recursive bindings are introduced by the keyword **rec**. Examples of recursive value bindings are given in section 5.2.2 and section 6.4 since they have to be used in combination with functions and recursive data types.

The problem of uninitialized identifiers is avoided completely in TL, since identifiers can only be introduced in bindings and, furthermore, recursive bindings are subject to static constraints that avoid access to uninitialized variables [14].

5.2 Dynamic Bindings

Dynamic bindings are established by passing parameters to functions. In addition to *simple* and *recursive* functions known from other programming languages, TL supports *higher-order* functions and *polymorphic* functions. Simple and recursive functions as well as higher-order functions are presented in the following sections. The description of the polymorphic functions is postponed until section 8.1 for didactical reasons.

5.2.1 Simple Functions. Functions are introduced by the keyword **fun**. In TL, functions can be defined without binding them to an identifier. Such a *function abstraction* consists of an ordered, possibly empty list of formal parameters (signatures) and an expression defining the body of the function.

```
fun(x :Int) x + 1
```

The body of the function (here *x + 1*) can refer to identifiers of different scopes. The formal parameters introduced by the signature of the function, the global identifiers present in the static scope of the function, and the identifiers defined locally inside the function are all visible in the function body.

```
let global = 1
fun(x :Int) begin let local = 3  x + global - local end
```

A function defined by a function abstraction can be bound to an identifier.

```
let succ = fun(x :Int) x + 1
let add = fun(x :Real  y :Real) x ++ y
let succ2 = succ
```

The first function (*succ*) expects a parameter of type *Int* and returns a value of type *Int* as its result. It computes the successor of an integer value passed as a parameter. The second function (*add*) adds two real numbers. It takes two parameters of type *Real* and returns a value of type *Real*. The third identifier (*succ2*) is bound to the function denoted by *succ*. The syntax of TL also supports the following abbreviated notation.

```
let succ(x :Int) = x + 1
let add(x, y :Real) = x ++ y
```

The type of the result can be made explicit, improving the readability of the program; if omitted it is inferred by the compiler.

```
let succ(x :Int) :Int = x + 1
let add(x, y :Real) :Real = x ++ y
```

Infix symbols can be chosen as names for functions that can be used as binary infix operators. In the following example, the function concatenating two strings (*string.concat*) is bound to the infix symbol <>.

```
let <> = string.concat
```

A function bound to an infix symbol can be applied in two different ways, either using the standard prefix notation or the infix notation:

```
<>("concat" "enation")
"concat" <> "enation"
```

As shown in the next example, the use of the infix symbol in the prefix notation can lead to unexpected results because of the factoring rules for expressions.

```
begin
  let a = 3
  <>("concat" "enation")
end
```

The above expression causes a syntax error since the compiler recognizes an expression of the form *3 <> ("concat" "enation")*. As usual, such problems can be avoided by the use of brackets to control the parsing of expressions.

```
begin
  let a = 3
  {<>}("concat" "enation")
end
```

5.2.2 Recursive Functions. TL supports the definition of recursive functions. Recursive bindings introduced by the keyword **rec** are used for this purpose. In contrast to normal bindings where the types of the bound values can be inferred by the compiler they have to be specified explicitly for recursive bindings. The well-known computation of the factorial function is an example of a recursive function binding.

```
let rec fac(n :Int) :Int = if n == 0 then 1 else n * fac(n - 1) end
```

As mentioned above, TL does not support the overloading of operators. The operator for an equality test, therefore, is the doubled equality sign (==) and not the simple equality sign (=) used in **let**-bindings. The polymorphic operator == tests simple values like numbers and booleans for equality whereas it checks structured values like tuples and arrays for identity, i.e., the equality of the values of the tuple and array components is not tested.

Mutually recursive functions have to be defined in *parallel*. In TL the bindings are connected by the keyword **and** for this purpose. A parity test is given as an illustrative example.

```
let rec even(x :Int) :Bool =
  if x == 0 then true else odd(x - 1) end
and odd(x :Int) :Bool =
  if x == 0 then false else even(x - 1) end
```

5.2.3 Function Types. Since function types are a prerequisite for the definition of higher-order functions, they are introduced here in anticipation of the discussion in section 6. A function type defines the signature of a function value, i.e. the names and types of its formal parameters and the function result type. Function types are introduced by the keyword **Fun**. The types of the previously defined functions *succ*, *add*, and *succ2* are given as examples.

```
succ :Fun(x :Int) :Int
add :Fun(x :Real  y :Real) :Real
succ2 :Fun(x :Int) :Int
```

The following abbreviating notation is also supported in TL.

```
succ(x :Int) :Int
add(x :Real  y :Real) :Real
```

5.2.4 Higher-Order Functions. Higher-order functions are functions accepting functions as parameters or returning functions as a result.

The functions *twice* and *newInc* are examples of higher-order functions.

```
let twice = fun(f :Fun(:Int) :Int  a :Int) :Int  f(f(a))
let newInc = fun(x :Int) :Fun(:Int) :Int  fun(y :Int) :Int  x + y
```

Again, the functions can be written down more concisely.

let *twice(f(:Int) :Int a :Int) = f(f(a))*
let *newInc(x :Int)(y :Int) = x + y*

The function *twice* receives two parameters. The first parameter is a function mapping an integer value to an integer value, and the second parameter is an integer. In the function body (*f(f(a))*) of *twice*, the function passed as a parameter is applied twice to the second parameter.

twice(succ 3)
⇒ *5 :Int*
*twice(***fun***(x :Int) x * x 3)*
⇒ *81 :Int*

The function *newInc* is an example of a function with a function result. An application of *newInc* returns an anonymous function whose application finally computes the addition.

let *add2 = newInc(2)*
add2(5)
⇒ *7 :Int*
newInc(3)(5)
⇒ *8 :Int*

As illustrated by the example, the application of the function can be performed in a single step or in two steps (currying).

6. Predefined Value and Type Constructors

The predefined type constructors of TL, tuple, tuple with variants, and record are presented in this section. Function types have already been introduced in section 5.2.3.

6.1 Tuple Types

The tuple types of TL resemble records in Pascal and in Modula-2 as well as structures in C. A tuple type is a labelled Cartesian product type. The fields of a tuple are described by an ordered, possibly empty, sequence of signatures. The signatures may contain anonymous identifiers.

Let *Person* = **Tuple** *name :String age :Int* **end**
Let *IntPair* = **Tuple** *:Int :Int* **end**

Tuple values are ordered lists of bindings.

let *peter* = **tuple let** *name = "Peter"* **let** *age = 3* **end**
let *paul* = **tuple** *"Paul" 5* **end**
let *pair* = **tuple** *12 21* **end**

The scope of the field names *name* and *age* is restricted to the block limited by the keywords **tuple** and **end**. Components of tuples are referenced using the dot notation.

peter.age
⇒ *3 :Int*

The rules for type compatibility of TL make an α-conversion between anonymous and non-anonymous field names possible. This conversion takes the order of the fields defined by the binding into account.

```
let p :Person = paul
p.name
⇒ "Paul" :String

let namedPair :Tuple x, y :Int end = pair
namedPair.x
⇒ 12 :Int
```

In TL it is possible to include functions as fields in tuples. Combining this concept with recursive bindings makes it possible to capture the concept of methods known from object-oriented programming. Illustrative examples are presented in the sections 8.4 and 9.1 and in [14].

6.2 Variant Types

Tuples with variants resemble variant records in Pascal and in Modula-2. Like tuples, tuples with variants represent ordered sequences of signatures.

```
Let Address =
  Tuple
    case national with street, city :String  zip :Int
    case international with street, city, state :String  zip :String
  end
```

The two variants *national* and *international* in the example have a common prefix. This prefix can be extracted from the variants and placed in front of them.

```
Let Address1 =
  Tuple
    street, city :String
    case national with zip :Int
    case international with state :String  zip :String
  end
```

If all signatures of the variants are empty, the tuple type with variants degenerates to an enumeration type.

```
Let Day = Tuple case mon, tue, wed, thu, fri, sat, sun end
```

The definition of a value of a tuple type with variants consists of the choice of a variant and the definition of the corresponding bindings.

```
let address1 =
  tuple case national of Address1 with
    let street = "Johnsallee 21"
    let city = "Hamburg"
    let zip = 21234
  end
```

The keyword **with** in the definition of *address1* is optional. It is also possible to use anonymous bindings in tuples with variants.

```
let address2 =
  tuple case national of Address1
    "Johnsallee 21"  "Hamburg"  21234
  end
```

A value of type *Day* can be defined as follows.

```
let today = tuple case mon of Day end
```

The projection on fields in the prefix and on fields of the variants requires two distinct notations. Fields of the prefix can be accessed using the dot notation as in the case of simple tuple fields.

```
address1.street
⇒ "Johnsallee 21" :String
```

For the fields of the variants a complete (**case of**) or an incomplete case analysis (**case**) is necessary.

```
case of address1
  when national with n then fmt.int(n.zip)
  when international with i then i.zip
end
```

The use of the complete case analysis avoids unexpected runtime errors by ensuring that later extensions of a tuple type with new variants are accompanied by corresponding extensions of the case analysis. The incomplete case analysis has the following form.

```
case address1
  when national with n then fmt.int(n.zip)
end
```

Since an incomplete case analysis can lead to runtime errors, an **else**-branch can (and should) be specified in this situation.

```
case address1
  when national with n then fmt.int(n.zip)
  else "not national"
end
```

Finally, two abbreviating notations for the simple test of variants and for the projection of variants are presented.

```
address1?national
address1!national
```

These two examples are equivalent to the following expressions.

```
case address1
  when national then true else false
end

case address1
  when national with n
  then n
  else raise tupleProjectError with line column "national" 1 end
end
```

The variant projection opens the scope of the selected variant.

address1!national.zip
⇒ *21234 :Int*

6.3 Record Types

In contrast to tuple types, record types represent *unordered,* possibly empty sets of *non-anonymous* signatures in TL. The names of all fields have to be different.

Let *Person* = **Record** *name :String age :Int* **end**

Record values are unordered sets of non-anonymous bindings.

let *peter* = **record let** *age* = *3* **let** *name* = *"Peter"* **end**

As for tuple values, the scope of the field names *name* and *age* is restricted to the block enclosed by **record** and **end**. The fields of a record are accessed using the dot notation.

peter.age
⇒ *3 :Int*

In contrast with tuple values, record values can be extended dynamically by non-anonymous bindings without losing their identity. The keyword **extend** is provided for this purpose. In the process of extending a record, the uniqueness of the field names has to be ensured.

let *peterAsStudent* = **extend** *peter* **with let** *semester* = *1* **end**

The infix operator == checks the identity of two values.

peter == *peterAsStudent*
⇒ *true :Bool*

The record value *peterAsStudent* fulfills the following type specifications, amongst others (see also sec. 7.3)

Record *name :String age :Int semester :Int* **end**
Record *age :Int name :String semester :Int* **end**
Record *semester :Int name :String age :Int* **end**
Record *name :String age :Int* **end**

6.4 Recursive Data Types

Recursively defined data structures like lists, sets, and trees play a central role in computing science. TL provides means for the definition of recursive data types enabling a straightforward realisation of recursively defined data structures.

A recursive type definition is introduced by the keyword **Rec** in TL. A supertype (e.g., *IntegerList* <:**Ok**) has to be specified when defining a recursive type. The definition of a list of integer values is presented as an example.

Let Rec *IntegerList* <:**Ok** =
  Tuple
    case *nil*
    case *cons* **with** *car :Int cdr :IntegerList*
  end

The following expressions show the construction of an empty list and the construction of a new list from an existing (possibly empty) list by appending a new element.

```
let emptyList =
  tuple case nil of IntegerList end

let singleList =
  tuple case cons of IntegerList with
    let car = 7
    let cdr = emptyList
  end
```

The next example shows the definition of a recursive value. As in the case of recursive functions, the type of the value has to be given explicitly.

```
let rec circularList :IntegerList =
  tuple case cons of IntegerList
    7  circularList
  end
```

6.5 Dynamic Data Types

In data-intensive applications there are programming situations where a context has to use a value generated by another context although the two context do not share common type information supporting static checking. In such situations it is desirable to defer the type checking to well-defined points during program evaluation. In TL, the keywords **Dyn** and **typecase** are provided for this purpose. Their application is illustrated by the following example.

```
Let Auto = Tuple Dyn T <:Ok  x :T end
let a1 = tuple Let Dyn T = Int  let x = 3 end
let a2 = tuple Let Dyn T = String  let x = "Hello" end

let asString(a :Auto) :String =
  typecase a.T
    when Int then fmt.int(a.x)
    when String then a.x
    when Tuple name :String end then a.x.name
    when Tuple end then "Tuple"
    else "???"
  end
```

As discussed in [14] and in [22] structural equivalence is a prerequisite for dynamic type checking in persistent, distributed systems.

7. Subtype Relationships and Subtype Polymorphism

In TL, a signature of the form *x :A* is considered a partial specification. A value bound to the variable *x* has to fulfil at least the specification defined by the type *A*. The underlying partial order on types (*B* is more precise than *A*) is described explicitly by an inductively defined subtype relationship (B <:A, B is subtype of A).

The supertype of all non-parameterized types is **Ok** [14]. It represents the trivial specification that is fulfilled by all values. The following example illustrates the use of the type **Ok**. The functions *fst* and *snd* both discard one of their parameters. The type of this parameter needs just a trivial specification.

let *fst(a :Int b :***Ok***) :Int = a*
let *snd(a :***Ok** *b :Int) :Int = b*
fst(3 4) fst(3 true) snd(3 4) snd(true 4)
⇒ *3 :Int 3 :Int 4 :Int 4 :Int*

The kind of polymorphism represented by the functions *fst* and *snd* is called *subtype polymorphism*. According to the subsumption principle the dynamic binding of formal parameters of a static type A to values of an arbitrary subtype B <:A is possible.

7.1 Subtyping on Predefined Types

All predefined base types fulfil the trivial specification (<:**Ok**), e.g.:

Int <:**Ok**
Real <:**Ok**
Fun*(:Int) :Int* <:**Ok**
Tuple *:Int* **end** <:**Ok**

Non-trivial subtype relationships, e.g., of the form *Int* <:*Real* do not exist between the base types of *Tycoon*. It is, however, possible to define subtypes of these types in the *Tycoon* libraries as, for example, *directory.T* <:*String*. Values of the type *directory.T* represent syntactically correct path names within the file system and thereby are also strings. The reverse is not true.

7.2 Subtyping on Tuple Types

In TL, subtyping on tuple types is based on structural compatibility. Subtype relationships are not only defined between two tuple types without variants and two tuple types with variants, but also between a tuple type without and a tuple type with variants.

A tuple type B without variants is a subtype of a tuple type A without variants if the signatures of B are a prefix of the signatures of A, e.g., for

Let *Student* = **Tuple** *name :String age :Int semester :Int* **end**
Let *Person* = **Tuple** *name :String age :Int* **end**
Let *Car* = **Tuple** *name :String age :Int* **end**
Let *Machine* = **Tuple** *name :String fuel :String* **end**
Let *NamedThing* = **Tuple** *name :String* **end**

the following subtype relationships hold

Student <:*Person* *Person* <:*NamedThing*
Car <:*NamedThing* *Machine* <:*NamedThing*

but also

Car <:*Person* *Person* <:*Car*

On the other hand, it is described in [14] how the *Tycoon* subtype relationship can be restricted systematically to explicitly defined *subclasses* ensuring, for example, that $car.T \not<: person.T$ holds.

A subtype of a tuple type can also be defined by specializing the types of tuple fields. For example, the definition

Let *Student2* = **Tuple** *name* :**Ok** *age* :*Int* *semester* :*Int* **end**

implies the following subtype relationship.

Student <:*Student2*

In TL, a subtype can be defined without repeating explicitly all the components of the supertype. The keyword **Repeat** is used for this purpose. A definition of the type *Student* on the basis of the type *Person* may look as follows.

Let *Student* = **Tuple** **Repeat** *Person* *semester* :*Int* **end**

This notation is applicable wherever signatures are expected, e.g., in function signatures. A corresponding construct exists on the value level. The keyword **open** supports the repetition of existing bindings. The value *peter* defined in section 6.1, for example, can be extended by specifying its semester.

let *peterAsStudent* :*Student* = **tuple** **open** *peter* **let** *semester* = *6* **end**

The major advantage of subtyping is the fact that functions working on a type also accept values of arbitrary subtypes of this type. Thereby, subtyping facilitates a later extension of programs, in particular, the extension of data structures with new components. Functions written for the original type are also applicable to values of the new type. These values are recognized as instances of the old type. A function expecting two parameters of type *NamedThing* also works on values of the types *Student*, *Car*, or *Machine*.

let *sameName(x, y :NamedThing) :Bool* =
string.equal(x.name y.name)

let *fiat* :*Machine* = **tuple** **let** *name* = *"Uno"* **let** *fuel* = *"unleaded"* **end**

sameName(peter fiat)
⇒ *false :Bool*

A tuple type B with variants is a subtype of a tuple type A with variants, if the ordered sequence of variant names of B is a prefix of the sequence of variant names of A and the signatures S_i of each variant of B are tuple subsignatures of the corresponding variant signatures S_i' of A. For example, the relationship *RGBColor* <:*Color* holds for

Let *RGBColor* = **Tuple** **case** *red, green, blue* **end**
Let *Color* = **Tuple** **case** *red, green, blue, cyan, yellow* **end**

and the relationship *Address* <:*Address2* holds for the type *Address* declared in section 6.2 if *Address2* is defined as follows:

```
Let Address2 =
  Tuple
    case national with street, city :String  zip :Int
    case international with street, city, state :String  zip :String
    case unknown
  end
```

Finally, a tuple type without variants having signatures S is a subtype of a tuple type A with variants if the signatures S are tuple subsignatures of the signatures S' of the first variant of A. The relationship *AddressTuple* <:*Address2*, therefore, holds for

```
Let AddressTuple = Tuple street, city :String  zip :Int end
```

7.3 Subtyping on Record Types

As in the case of tuple types, subtypes of record types can be constructed by specialization of types of existing components as well as by extension with new components. Additionally, the fact that the signatures of record types are not ordered is taken into account by the subtyping rules. A record type B with signatures S is a subtype of a record type A with signatures S' if the signatures S contain a subset of signatures S'' which are subsignatures of S'.

```
Let Person = Record name :String  age :Int end
Let Student = Record name :String  semester :Int end
Let Employee = Record ssno :String  salary :Real end
Let Tutor = Record name, ssno :String age, semester :Int salary :Real end
```

For these types the following subtype relationships hold in TL,

Tutor <:*Person* *Tutor* <:*Student* *Tutor* <:*Employee*

enabling the application of functions which are defined for arguments of the types *Person*, *Student*, or *Employee* to values of the type *Tutor*. Furthermore, it becomes possible to construct heterogeneous data structures consisting, for example, of values of the types *Person*, and *Tutor*. Subtyping hierarchies over record types are not restricted to tree structures as in the case of tuple types. Other directed acyclic graphs are also possible, thus making the representation of multiple inheritance hierarchies possible.

7.4 Subtyping on Function Types

The interpretation of types and signatures as partial specifications and of subtypes and subsignatures as specification refinements implies the well-known *contravariance rule* for the subtyping on function types. According to this rule a function type with signatures S for its formal parameters and result type B is a subtype of a function type with signatures S' and result type A iff $B <: A$ and S' are subsignatures of S. In other words, a function F_1 is a specialization of a function F_2, if under the assumption, that the preconditions of F_2 hold, the postconditions of F_1 fulfil at least the postconditions of F_2 and the preconditions of F_1 are not more restrictive than the preconditions of F_2 [14].

For example, assuming the relationship *Student* <:*Person* the relationship *GetRichestStudent* <:*GetRichestPerson* holds for the following function types because the result types of the function types are in covariance relationship.

Let *GetRichestPerson* = **Fun**() :*Person*
Let *GetRichestStudent* = **Fun**() :*Student*

However, for the function types

Let *HirePerson* = **Fun**(:*Person*) :**Ok**
Let *HireStudent* = **Fun**(:*Student*) :**Ok**

the relationship *HirePerson* <:*HireStudent* holds because the types of the function parameters are contravariant.

8. Parametric Polymorphism

The type system of TL supports two kinds of polymorphism: subtype polymorphism and parametric polymorphism. Subtype polymorphism is presented in the previous section. Parametric polymorphism is the topic of this section. It makes the introduction of explicit type parameters into function and type definitions possible.

Function definitions with parameters describe *polymorphic* functions, whereas the introduction of type parameters into type definitions results in *type operators*. Type operators are functions mapping types to types. They introduce parametrization into type declarations.

If a type is restricted explicitly to a subtype of a given type, this kind of polymorphism is called *bounded parametric polymorphism*.

8.1 Polymorphic Functions

A function is made polymorphic (generic) by extending its signature by one or more type parameters. The type parameters are instantiated with type expressions when the function is applied.

The polymorphic identity function is a simple example of a polymorphic function.

let *id*(*A* <:**Ok** a :*A*) :*A* = a

Such a function is called with a type (e.g.,:*Int*) and a value (e.g., 7) of this type as actual parameters.

id(:Int 7)
⇒ *7 :Int*
id(:String "Peter")
⇒ *"Peter" :String*

The specification of the type argument can be omitted in most cases because it can be inferred by the system from the value passed as parameter.

id(7)
⇒ *7 :Int*
id("Peter")
⇒ *"Peter" :String*

The instantiation of type parameters is not restricted to base types. Arbitrary user-defined types (e.g., the type *Person* with the value *peter*) can be chosen as parameter for a polymorphic function

```
id(peter)
⇒ tuple name = "Peter" age = 3 end :Person
```

Parametric polymorphism makes it possible to write functions working uniformly for arbitrary types. Polymorphic functions, therefore, can be used to describe type-independent behaviour. Separate functions for each considered parameter type would be necessary for this purpose in languages like C or Modula-2.

Much of the power of TL results from the possibility of combining the concepts of polymorphic and higher-order functions with each other. This is illustrated by a polymorphic sorting function. The pure sorting process, i.e. the permutation of the elements, is type-independent. Therefore, it can be described by a polymorphic function. On the other hand, the comparison of the elements during the sorting process is type-dependent. This task can be solved by passing a function whose application compares two elements as a parameter to the polymorphic sorting function. The signature of a polymorphic function sorting arrays of an arbitrary element type follows.

```
let sort(A <:Ok  a :Array(A)  order(a, b :A) :Bool) :Array(A) = ...
```

In order to sort an array of a specific type, it is sufficient to write a function for the element comparison of this array. The sorting of persons by ascending age is considered as an example.

```
let older(a, b :Person) :Bool = a.age >= b.age
```

The following call of the function *sort* sorts an array of persons according to their age.

```
sort(:Person personArray older)
```

Again, the type parameter can be omitted.

```
sort(personArray older)
```

Further examples of polymorphic functions can be found in the sections 8.3 and 8.4.

8.2 Bounded Parametric Polymorphism

Bounded parametric polymorphism, a restricted form of parametric polymorphism, is introduced into polymorphic functions by specifying a type as a bound for the formal type parameter in the signature. Only subtypes of the given type can be passed as parameters to such functions. Employing subtype polymorphism, a function comparing the component *age* of two values of an arbitrary subtype of the type *Person* can be defined in the following way.

```
let chooseOlder(p1, p2 :Person) :Person =
  if p1.age > p2.age then p1 else p2 end

let peter :Student =
  tuple let name = "Peter"  let age = 24  let semester = 3 end

let paul :Student =
  tuple let name = "Paul"  let age = 29  let semester = 6 end

chooseOlder(peter paul)
⇒ tuple let name = "Paul" let age = 29 end :Person
```

Considering the function result displayed by the system, it can be seen that the attribute *semester* of *paul* is missing. This is a consequence of the fact that type information is lost at compile time. In order to avoid the loss of attributes of subtypes, *chooseOlder* has to be defined as a polymorphic function.

```
let chooseOlder(P <:Person  p1, p2 :P) :P =
  if p1.age > p2.age then p1 else p2 end

chooseOlder(peter paul)
⇒ tuple name = "Paul" age = 29 semester = 7 end :Student
```

As a result of integrating enough type information into the function definition, all attributes are taken into account. By introducing the specification *P <:Person*, the function *chooseOlder* is made polymorphic, but in contrast to the unrestricted parametric polymorphism, the polymorphism is restricted to subtypes of the type *Person*. The type variable *P* makes the intended relationship between the type of the formal parameters and the type of the result of the function explicit.

8.3 Type Operators

Polymorphic functions support the description of type-independent behaviour. Similarly, type-independent patterns on the type level lead to generic type expressions in the form of type operators, which can be instantiated with concrete types.

8.3.1 Simple Type Operators. A simple example of a type operator is an identity function on the type level corresponding to the polymorphic identity function presented above.

```
Let Id = Oper(A <:Ok) A
```

Similar to the function definition, the following abbreviated notation is also possible.

```
Let Id(A <:Ok) = A
```

The operator *Id* maps the type passed as parameter to itself.

```
:Id(Int)
⇒ :Int
:Id(Id(Bool))
⇒ :Bool
```

A more practical application of the type operators is the description of optional values.

```
Let Opt(A <:Ok) =
  Tuple
    case nil
    case notNil with val :A
  end
```

The syntax for the application of a type operator is equivalent to the syntax for the function application.

```
Opt(Person)
Id(Opt(Id(Person)))
```

Symbolic identifiers bound to type operators can be used in infix notation. These infix operators on the type level are all left-associative and of the same precedence. The classical binary type operators of functional programming languages can be introduced into TL in the following way.

```
Let ->(X, Y <:Ok) = Fun(:X) :Y
Let *(X, Y <:Ok) = Tuple fst :X  snd :Y end
Let +(X, Y <:Ok) = Tuple case fst with x :X  case snd with y :Y end
```

The preceding examples are restricted to first-order type operators. TL also supports the definition of higher-order type operators. Higher-order type operators are, for example, type operators accepting a type operator as parameter and applying it to different types in the body. Other examples are type operators generating type operators as result based on non-parametrised types passed as parameters. The coding of case selections on the type level is considered as an illustrative example.

```
Let Boolean(Then, Else <:Ok) = Ok
Let True(Then, Else <:Ok) = Then
Let False(Then, Else <:Ok) = Else
Let Cond(If <:Boolean  Then, Else <:Ok) = If(Then Else)

let i :Cond(True Int String) = 3
let s :Cond(False Int String) = "Peter"
```

8.3.2 Recursive Type Operators. In addition to recursive types TL supports recursive type operators. The type operator *List* maps a type E to the type of a list with elements of type E. For this purpose a type parameter is introduced into the definition of lists presented in section 6.4.

```
Let Rec List(A <:Ok) <:Ok =
  Tuple
    case nil
    case cons with car :A  cdr :List(A)
  end
```

The corresponding list operations can be implemented by polymorphic functions.

```
let new(A <:Ok) :List(A) =
  tuple case nil of List(A) end

let cons(A <:Ok  head :A  tail :List(A)) :List(A) =
  tuple case cons of List(A) with head tail end
```

The polymorphic functions *new* generates empty lists with elements of an arbitrary but specific type. Elements can be added at the beginning of the list by the function *cons*.

In contrast to commercial programming languages, generic list, set, and tree types and polymorphic operations on these types can be defined in TL reducing the number of functions that have to be implemented.

The concepts of polymorphic functions and type operators complement each other. Generic code can be used for the description of structures as well as of behaviour.

8.4 Abstract Data Types

An abstract data type (ADT) consists of a data type and a set of operations defined over this data type. Only the name of the type and the names and signatures of the operations are visible to programs that use the ADT. The implementation of the type and of the operations are hidden by the ADT. The operations provided by the ADT are the only possible and legal ones on the ADT. This protects the values of the ADT against undesired manipulations.

Since the implementation is hidden, it can be changed locally without invalidating programs that use the ADT. If different implementations exist for an ADT, these implementations can be exchanged dynamically.

A general functional stack is presented as an example. The stack is declared employing a polymorphic abstract data type.

```
Let Stack =
  Tuple
    T(E <:Ok) <:Ok
    new(E <:Ok) :T(E)
    empty(E <:Ok stack :T(E)) :Bool
    push(E <:Ok element :E stack :T(E)) :T(E)
    pop(E <:Ok stack :T(E)) :T(E)
    top(E <:Ok stack :T(E)) :E
  end
```

In the following an implementation of this interface based on the module *list*[5] is presented.

```
let listStack :Stack =
  tuple
    Let T(E <:Ok) <:Ok = list.T(E)
    let new(E <:Ok) :T(E) = list.new(:E)
    let empty(E <:Ok stack :T(E)) :Bool = list.empty(stack)
    let push(E <:Ok element :E stack :T(E)) :T(E) =
      list.cons(element stack)
    let pop(E <:Ok stack :T(E)) :T(E) = list.tail(stack)
    let top(E <:Ok stack :T(E)) :E = list.head(stack)
  end
```

In this example, *T* is defined as a type operator and the operations *new*, *empty*, *push*, *pop*, and *top* are implemented as polymorphic functions. Therefore, stacks for arbitrary data types can be generated.

ADTs are based on the concept of type signatures in tuple types. This leads to the concept of *semi-abstract* data types in TL. As an example, the signature of the type operator in the ADT *Stack* above can be modified in order to exhibit more information about the implementation of the ADT.

```
Let Stack2 =
  Tuple
    T <:list.T
    ...
  end
```

[5] *list* is a module of the standard library implementing polymorphic lists and the associated operations.

```
let listStack2 :Stack2 =
  tuple
    ...
  end
```

Users of the ADT *Stack2* can apply operations that expect values of the type *list.T* on the values of type *listStack2.T* in addition to the functions defined for the stack. On the other hand, a value of type *list.T* is incompatible with functions defined by *listStack*.

In TL, a subsignature relationship between signatures of ADTs is defined:

```
Stack2 <:Stack
```

Finally, a tuple signature can make a local type binding visible globally.

```
Let Stack3 =
  Tuple
    Let T = list.T
    ...
  end
```

Such type definitions are particularly useful in interfaces of modules when programming libraries (see section 11.1).

9. Imperative Programming

The discussions in the previous sections are restricted to the functional concepts underlying the language TL. The imperative programming features are described in this section.

Imperative programming is based on *mutable* variables in a global (possibly *persistent*) store. The flow of control between operations as allocation, inspection, and destructive update of objects in the store is determined by constructs for sequences and loops.

9.1 Mutable Variables

Binding a value to an identifier by the **let**-construct is equivalent to the definition of a *constant* because no update of the value bound to the identifier is possible. A further binding of an existing identifier to a new value employing the **let**-construct establishes a new constant.

In TL, bindings of identifiers to mutable variables are marked by the keyword **var**. Subsequently, an existing mutable variable can be updated with new values employing the *destructive* assignment :=.

```
let var x = 3
let var y = 4
x y
⇒ 3 :Int  4 :Int

x := y
⇒ ok :Ok
x y
```

```
⇒ 4 :Int  4 :Int

y := 5
⇒ ok :Ok
x y
⇒ 4 :Int  5 :Int
```

For anonymous variables, e.g., inside of tuple values, the keyword **var** precedes the value used to initialize the variable.

```
tuple var "text"  var 3 end
```

The destructive assignment is a globally defined function with the following signature.

```
:=(A <:Ok  var lValue :A  rValue :A) :Ok
```

The signature of the function defines that the assignment evaluates to the trivial value **ok** of type **Ok** as it is the case for the empty block. Note that the infix symbol := is not a keyword. Therefore, it can be bound locally to a user-defined polymorphic function.

TL realizes two parameter passing mechanisms for functions, namely the concept of value parameters (*call by value*) presented in section 5.2.1 and the concept of variable parameters (*call by reference*). The latter concept is illustrated by the following example.

```
let swap(A <:Ok  var x, y :A) =
  begin  let tmp = x  x := y  y := tmp  end

let var a = 3 and var b = 5
swap(:Int a b)
a b
⇒ 5 :Int  3 :Int
```

When applied, the function *swap* references the L-values of the mutable variables *a* and *b* through the formal parameters in the signature. On return from the function, the values of the mutable variables are swapped.

The concept of higher-order functions presented in section 5.2.4 supports the dynamic generation of encapsulated state variables, which can be shared by several functions (*shared variables*). In the following example, the variable *state* can be updated and inspected, respectively, by the three defined functions only.

```
let newCounter() =
  begin
    let var state = 0
    tuple
      let reset() :Ok =  state := 0
      let inc() :Ok =  state := state + 1
      let value() :Int =  state
    end
  end

let cntr1 = newCounter() and cntr2 = newCounter()
cntr1.inc()  cntr1.value()  cntr2.value()
⇒ ok :Ok  1 :Int  0 :Int
```

Mutable function bindings enable the programmer to *override* functions.

```
let var f(x :Int) = x + 1
f(3)
⇒ 4 :Int
f := fun(x :Int) x - 1
f(3)
⇒ 2 :Int
```

9.2 Subtyping Rules for Mutable Bindings

The details of the interaction between the subtyping rules and the destructive assignment are very important for the type-safety of polymorphic programming languages. TL follows the example of the language Quest (and loosely related concepts in C++) and disallows the application of the subsumption rule to mutable variables. For this reason

```
Fun(x :Person  y :Person) :Ok <:Fun(x :Student  y :Student) :Ok
Tuple x :Int end <:Tuple x :Ok end
```

holds, but the following subtyping relationships do not hold in TL [14]:

```
Fun(var x :Person  y :Person) :Ok <:Fun(var x :Student  y :Student) :Ok
Tuple var x :Int end <:Tuple var x :Ok end
```

However, the concept of the bounded parametric polymorphism supported in TL makes the definition of a type-safe polymorphic function *update* working uniformly for arbitrary subtypes of type *Person* possible.

```
let update(P <:Person  var a, b :P) = a:= b
```

Finally, it is possible in TL to type a mutable variable in an aggregate as a non-mutable value. As a consequence, the more liberal subtyping rules can be applied for the read-only access to value variables, e.g.,

```
Tuple var x :Int end <:Tuple x :Int end <:Tuple x :Ok end
```

9.3 Control Structures

In addition to *sequences*, TL offers control structures for *conditional expressions*, three kinds of *loops* and structured *exception handling* supporting a flexible imperative programming style. For the sake of completeness, concepts already introduced in previous sections are mentioned again.

9.3.1 Sequences. A sequence describes the sequential execution of expressions. As mentioned in section 5.1, expressions are enclosed by the keywords **begin** and **end** in order to form a block. The type of a sequence is determined by the type of the last expression or binding in the block.

```
begin
  let s = "text"
  let x = 1
end
⇒ 1 :Int
```

9.3.2 Conditional Expressions. The simplest form of a conditional expression is described by an **if**-expression in TL. The result types of all **then**-branches and the **else**-branch have to be compatible.

```
if x == 0 then
  0
elsif x < 0 then
  -1
else
  1
end
```

Several conditions can be conjoined by **andif** and **orif** operators in TL.

```
if x == 0 andif y == 0 orif x != 0 andif y != 0 then
  0
else
  x
end
```

Since **andif** and **orif** have the same precedence and are evaluated from left to right the above expression is equivalent to the following more complex expression:

```
if x == 0 then
  if y == 0 then
    0
  else
    if x != 0 then
      if y != 0 then
        0
      else
        x
      end
    else
      x
    end
  else
    x
  end
else
  x
end
```

Two further constructs for conditional expressions are provided in TL (see sections 6.2 and 6.5).

```
let weekDay(d :Day) :Bool =
  case d
    when mon, tue, wed, thu, fri then true
    else false
  end

let asString(Dyn A <:Ok  a :A) :String =
  typecase A
    when Int then fmt.int(a)
```

```
    when String then a
    else "???"
  end
```

9.3.3 Loops. Loops enclose sequences of expressions for the purpose of iteration. The most general kind of loop is introduced by the keyword **loop**. The result type of loops of this kind is **Ok**.

```
loop
  x := x + 2
  if x > 100 then exit end
  x := x - 1
end
```

Furthermore, there are two more special forms of loops, the prechecking **while**-loops and the enumerating **for**-loops. The function computing the greatest common divisor of two integers is considered as an example of a prechecking loop.

```
let gcd(n, m :Int) :Int =
  begin
    let var vn = n  and var vm = m
    while vn != vm do
      if vn > vm then vn := vn % vm end
      if vn < vm then vm := vm % vn end
    end
    vn
  end
```

Two versions of enumerating loops are distinguished in TL, one counting upwards (**upto**) and the other counting downwards (**downto**).

```
let var x = 0
for i = 50 downto 1 do
  x := x + i
end
x
⇒ 1275 :Int
```

Note, that it is not necessary to declare the loop variable *i* which has local scope. In section 9.4, an example of a loop counting upwards is presented.

9.3.4 Exception Handling. Exception handling is a further important structuring facility. In TL, it is also integrated smoothly into the type system. Exceptional situations that have to be handled can be caused by partially defined functions as, for example, *int.div* (division by zero), and also by an overflow of a partially represented domain. Furthermore, the projection on variants described in section 6.2 can raise exceptions in TL.

Each exception returns an exception package containing a string which supports the identification of the exception at the outermost level.

```
3 / 0
⇒ Exception: "Int error"
```

If an exception occurs inside a composite expression, the further evaluation is aborted and the exception package is propagated.

```
let safeDiv(x, y :Int) :Int =
  try x / y else int.maxValue end
```

In this example, the further propagation of an exception is stopped by the **try**-construct. As in the case of the **if**-construct, the result types of both blocks have to match.

In addition to the standard exceptions, TL also supports user-defined exceptions. The definition of an exception includes the identifier of the exception and, optionally, a signature for exception arguments. The definition is introduced by the keyword **exception**.

```
let noCredit = exception "No Credit" with overdrawn :Int end
```

The type of an exception defines only its signature not its identity.

```
noCredit :Exception with overdrawn :Int end
```

A **raise**-expression returns an exception package as its result, which encapsulates the identity of the exception. Depending on the exception signature it can contain further bindings.

```
let withdraw(var account :Int  amount :Int) =
  if amount <= account then
    account := account - amount
  else
    raise noCredit with let overdrawn = amount - account end
  end
```

Exception packages propagate through nested expressions along the dynamic call hierarchy until an exception block enclosed by **try** and **end** or the main program is reached.

```
try
  withdraw(petersAccount  300)
  print.string("Transfer succeeded")
when noCredit with exc then
  print.string("Overdrawn by " <> fmt.int(exc.overdrawn))
else
  print.string("Unexpected exception occurred")
end
```

Similar to the **case**-expressions a local value variable (here *exc*) in a **when**-branch of the **try**-construct can be used to access the bindings of the exception package in a type-safe way.

A handled exception can be propagated explicitly by reraising it (**reraise**). The following example also shows that exceptions raised by functions exported from *Tycoon*-libraries are bound to exported identifiers enabling the user to define handlers for these exceptions.

```
try x / y
when int.overflow then int.maxValue
when int.error then print.string("Division by zero")  reraise
end
```

Types in TL only specify the values of terminating computations. For this reason an *arbitrary* type can be assigned to terms containing **raise**, **reraise**, or **exit** (see section 9.3.3), e.g.,

```
if a then 3 else raise int.overflow end  :Int
if a then "String" else raise int.overflow end  :String
```

This fact is reflected by the type rules **raise** ... **end** :**Nok**, **reraise** :**Nok**, and **exit** :**Nok**, where **Nok** denotes the subtype of all non-parametrised types in TL, i.e.,

A <:**Ok** ⇒ **Nok** <:*A*

This property of the type **Nok** is frequently used for the definition of polymorphic *null elements*.

```
Let Rec AnyStream <:Ok =
  Tuple empty() :Bool  get() :Nok  rest() :AnyStream end
let emptyExc = exception "Empty Stream"

let emptyStream :AnyStream =
  tuple
    let empty() :Bool = true
    let get() :Nok = raise emptyExc
    let rest() :AnyStream = raise emptyExc
  end
```

9.4 Arrays and Array Indexing

An array is an ordered, possibly empty sequence of anonymous bindings to mutable variables of a common supertype indexed by non-negative integers (*i* >= *0*). The size of an array is fixed statically when it is generated. The size cannot be modified dynamically, whereas the elements of an array can be updated dynamically by destructive assignments. A check of the index bounds is performed at runtime only.

In TL, array types are defined using the constructor *Array(A)* and array values are initialized enumerating their elements inside of an **array-end**-block. The elements of an array are accessed by indexing. The indices are expressions of type *Int* enclosed in square brackets.

```
let a :Array(Int) = array 0 1 2 3 4 5 end
a[0] := a[1] + 7
```

Using the **for**-loop it is possible to define, for example, a summation function accepting arrays of an arbitrary size.

```
let sum(arr :Array(Int)) :Int =
  begin
    let var result = 0
    for i = 0 upto extent(arr) do
      result := result + arr[i]
    end
    result
  end
```

Finally, examples for the application of the function *sum* are presented illustrating the *listfix* notation that can be used for functions on arrays in TL.

```
sum(array 1 2 3 4 end)
sum of 1 2 3 4 end
```

10. Multi-Paradigm Programming in Tycoon

It has been one of the design goals of the Tycoon system to support generic, model-independent naming, binding, and typing schemata providing an environment that is open for external services. TL can be used to support programming styles, which differ substantially from each other (see [14]). This is illustrated here by the realisation of two concepts: abstract data types and object-oriented encapsulation.

The different programming styles and concepts are not supported by special language constructs but are realised using the TL primitives for naming, binding, and typing.

In order to implement the concept of abstract data types the primitives of TL can be combined in three different ways. It is possible to aggregate an opaque type together with the functions working on this type[6]. This approach can be implemented purely functionally or state-based resulting in the first two realisation variants. The third variant aggregates methods that work on a hidden, internal state. The different realisation alternatives are called

- *functional* encapsulation,
- *imperative* encapsulation, and
- *method-based* encapsulation.

A generic stack implementation is used to compare the three different realisation alternatives. Each of them provides a type operator that maps the element type of the stack to a tuple type. In the first two cases this tuple type aggregates the opaque stack type T, the common stack operations *empty, push, pop, top*, and a parameterless function *new* for the creation of new empty stacks. For the third alternative, the tuple type aggregates only the stack operations. It represents the type of a stack. The function *new* has to be defined outside this type signature.

Functional Encapsulation. In the functional realisation the modified stack is returned by the the update operations. Therefore, the result type of these functions is the opaque type T:

```
Let FunStack (E <:Ok) =
  Tuple
    T <:Ok
    new() :T
    empty(stack :T) :Bool
    push(element :E stack :T) :T
    pop(stack :T) :T
    top(stack :T) :E
  end
```

The data type is implemented by a parametrised variable. It provides a tuple value consisting of a definition of the representation type and of the functions of this type. The generic service (*list*) providing the list type and the operations on this type is also based on a functional implementation.

```
let listStack (E <:Ok) : FunStack(E) =
  tuple
    Let T <:Ok = list.T(E)
    let new() :T = list.new(:E)
    let empty(stack :T) :Bool = list.empty(stack)
```

[6] This is comparable to an implementation of abstract types in Modula-2 using the module concept of that language.

```
    let push(element :E stack :T) :T = list.cons(element stack)
    let pop(stack :T) :T = list.tail(stack)
    let top(stack :T) :E = list.head(stack)
  end
```

Using this realisation a new stack with elements of type *Int* containing the element 4 can be created by the following function calls:

```
 let intStack = listStack(:Int)
 let myStack = intStack.push(4 intStack.new())
```

Imperative Encapsulation. In the imperative realisation the update operations change the state of the stack passed as parameter via side-effects. In contrast to the functional solution, the modified stacks are not returned as function results. For this reason the result type of these operations is **Ok**.

```
Let ImpStack (E <: Ok) =
  Tuple
    T <: Ok
    new() : T
    empty(stack :T) :Bool
    push(element :E stack :T) :Ok
    pop(stack :T) :Ok
    top(stack :T) :E
  end
```

The representation type of the implementation is defined as a tuple type with a mutable component. The update operations are implemented using assignments.

```
let listStack (E <:Ok) : ImpStack(E) =
  tuple
    Let T <:Ok = Tuple var l :list.T(E) end
    let new() :T = tuple let var l = list.new(:E) end
    let empty(stack :T) :Bool = list.empty(stack.l)
    let push(element :E stack :T) :Ok= stack.l := list.cons(element stack.l)
    let pop(stack :T) :T = stack.l := list.tail(stack.l)
    let top(stack :T) :E = list.head(stack.l)
end
```

In this case a new stack containing the integer 4 can be created by the following function calls:

```
 let intStack = listStack(:Int)
 let myStack = intStack.new()
 intStack.push(4 myStack)
```

Method-Based Encapsulation. The third encapsulation technique binds functions to an internal, shared, mutable variable. The function signatures are defined by a tuple type. This type represents the type of a stack object.

```
Let StackObject (E <:Ok) =
  Tuple
    empty() :Bool
    push(element :E) :Ok
    pop() :Ok
    top() :E
  end
```

An implementation of the stack functions is provided by a *new*-function that can be used to create "objects" of this type. The implementation of their methods are value components of these objects. In this implementation framework, method overwriting for subtype objects can also be realised (see [14]). For this reason this encapsulation method is called object-oriented in [14]. It avoids an opaque type and input parameters of this type[7].

```
let newStackObject(E <:Ok) :StackObject(E) =
  begin
    let var state = list.new(:E)
    tuple
      let empty() :Bool = list.empty(state)
      let push(element :E) :Ok = state := list.cons(element state)
      let pop() :Ok = state := list.tail(state)
      let top() :E = list.head(state)
    end
end
```

For this realisation variant the creation of an integer stack with the element 4 looks as follows:

```
let myStack = newStackObject(:Int)
myStack.push(4)
```

11. Programming in the Large

Besides function abstraction, modularization is the most important structuring facility in modern programming languages. Large programs can be split into *interfaces* and *modules* in TL. Moreover, it is possible to group interfaces and modules into *libraries*.

These structuring mechanisms do not introduce new concepts for naming, binding, or typing. They just deliberately restrict existing concepts of TL [14].

11.1 Modules and Interfaces

Interfaces define the signatures of exported values, functions, types, and type operators. Interfaces, therefore, can be viewed as named tuple types containing references to explicitly imported modules (e.g., *bool*) and interfaces visible in the global scope.

```
interface List
import bool
export
  T(E <:Ok) <:Ok
  Let AnyT = T(Nok)
  error :Exception
  nil :AnyT
  cons, ::(E <:Ok  hd :E  tl :T(E)) :T(E)
  empty(E <:Ok  l :T(E)) :bool.T
  car(E <:Ok  l :T(E)) :E
  cdr(E <:Ok  l :T(E)) :T(E)
end
```

[7] Operations expecting more than one input parameter of the opaque type lead to recursive type definitions when this implementation style is used[14].

Interfaces can include type bindings (*AnyT*). The definition of these types is visible to all users of the interface. If such types are imported by other modules, the name of the interface as well as the name of the module can be employed as qualifying identifiers.

A module defines a tuple value aggregating bindings according to its interface. In TL, an arbitrary number of modules can exist for a single interface.

```
module list
import bool
export
  Let Rec T(E <:Ok) <:Ok =
    Tuple case nil  case cons with hd :E  tl :T(E) end
  Let AnyT = T(Nok)
  let error = exception "Empty list"
  let nil = tuple case nil of AnyT end
  let cons(E <:Ok  hd :E  tl :T(E)) :T(E) =
    tuple case cons of T(E) hd tl end
  let :: = cons
  let empty(E <:Ok  l :T(E)) :bool.T = l?nil
  let car(E <:Ok  l :T(E)) :E =
    try l!cons.hd else raise error end
  let cdr(E <:Ok  l :T(E)) :T(E) =
    try l!cons.tl else raise error end
end
```

Type bindings established in the interface (here *AnyT*) have to be repeated in the module. Modules and interfaces are *first-class* objects of the language. They can be bound to identifiers and passed as parameters to functions. Modules and interfaces can be imported by other modules and interfaces employing the **import**-clause. In the library a unique interface is assigned to every module name.

Interfaces and modules are definable at the *top level* of the interactive programming environment. Their definition implicitly generates persistent data structures describing the types of interfaces and the values of modules, respectively.

After importing a module its components are referenced by the dot notation.

```
module main
import list  print
export
  let l = list.cons(3  list.nil)
  let l2 :list.AnyT = list.nil
  if not(list.empty(l)) then
    print.int(list.car(l))
  end
end
```

11.2 Libraries

The rapidly growing number of modules in real systems and the necessity for tools supporting consistent system restructuring in persistent systems makes it necessary to organize modules and interfaces into libraries and suggests the introduction of a library concept into the language TL.

A library defines the scope of the names of its local modules and interfaces and supports the definition of subsystems encapsulating hidden modules and interfaces.

The definition of the standard library(*StdLib*) is presented as a simple example.

```
library StdLib
with
  interface
    Bool Int Char Real ArrayOp
  module
    bool :Bool  int :Int  char :Char  real :Real  arrayOp :ArrayOp
  interface
    List
  module
    list :List
end
```

The order in which the names of the modules and interfaces are listed matters. A module or interface can only import interfaces or modules that are declared before this module or interface in the library, e.g. *bool* cannot import *list*.As a consequence, cyclic dependencies are ruled out. The modules of the standard library introduced in the previous example can be imported into a further library (*BulkLib*).

```
library BulkLib
import arrayOp :ArrayOp  list :List  iter :Iter
with
  interface
    Set Bag Assoc Dictionary VarList
  module
    linkedSet :Set  bitSet :Set  hashedSet :Set
  module
    bag :Bag  assoc :Assoc  dictionary :Dictionary
  hide
    varList :VarList
end
```

For this purpose, the libraries *StdLib* and *BulkLib* have to be defined in the following order as parts of an enclosing library.

```
library Root
with
  library
    StdLib BulkLib
  interface
    Test
  module
    test :Test
end
```

The example of the library *BulkLib* illustrates two further facilities of the library concept of TL. Modules and interfaces can be hidden in the library employing the **hide**-clause. Furthermore, it is possible to specify different modules (*linkedSet*, *bitSet*, and *hashedSet*) for a single interface (*Set*).

TL supports hierarchic library structures, but the names of all modules, interfaces and libraries inside of a library have to be unique. This is also true for components defined as hidden.

12. Persistence and Garbage Collection

In *Tycoon* no linguistic difference between *persistent* and *temporary* data is made. Every object can be made persistent. Persistence is defined by *reachability* either from a linked library module or from a local name space of a user (*top level*). This persistence concept works for values, functions, and (dynamic) type bindings.

Consistent states of the object store are marked by explicitly stabilising the object store. The module *store* provides the function *stabilise* for this purpose. A call of this function stabilises the actual state of the object store.

```
import store;
store.stabilise();
```

The operation *store.stabilise* generates a *checkpoint*. If a user quits the session with the command **do exit** or if a system crash occurs, all changes of objects in the persistent store performed after the checkpoint are undone (*rollback*). At the beginning of the next session the object store is in the state of the last checkpoint. Furthermore, it is possible to rollback explicitly to the state of the last checkpoint without leaving the system. This is accomplished by the function *restart* which is also exported by the module *store*. The effect of the functions *stabilise* and *restart* is illustrated by the following example.

```
import store list;
let var l = list.nil
l := list.cons(1 l)
l := list.cons(2 l)
store.stabilise();
(* object store with list l containing elements 1 and 2 is stabilised *)

l := list.cons(3 l)
store.restart();
(* rollback to last checkpoint; insertion of 3 into l is undone *)

list.car(l);
⇒ 2
```

Objects that are no longer reachable are automatically deleted from the object store by a *garbage collector*.

13. 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 directly as C function pointers.

13.1 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[8] 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

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* as in C, except for the function result type. It is given by the last character which 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)
s	*String*	**char** *	zero-terminated string
v	**Ok**	**void**	return value only
=	<:**Ok**	**void** *	*Tycoon* value, no conversion
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 → *1*
false → *0*

A C value x produces the following TL boolean values:

x != 0 → *true*
x == 0 → *false*

When *s* is used as a format character for a string parameter, every call is enclosed by automatic fix and unfix operations for the argument. The result of the fix operation, a main memory pointer is passed to C. The latter refers to a valid C string, because all *Tycoon* strings are represented with zero-termination.

[8] 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.

Used in return value position the format character *s* causes strings returned from C to be copied into newly created store objects (*copy-out*).

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* is 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 re-established automatically.

13.2 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*) :*callback.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* as 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 13.1.

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 *by reference* which apply for strings within C only.

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

For parameters with format code *w* a TL type that is equivalent to *word.T* or to an instance of *word.Handle* must be used. In particular callbacks conform to *word.Handle*, because *cCallback.T* <: *word.Handle*.

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"  "iwv")

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);
}
```

Acknowledgement This research was supported by ESPRIT Basic Research, Project FIDE, #6309. The Tycoon system described in this paper was developed by Andreas Gawecki, Bernd Mathiske, Florian Matthes and Rainer Müller. Section 12 of this text was written by Bernd Mathiske who also developed Tycoon's external language bindings. The author would also like to thank Claudia Niederée, Gerald Schröder, Petra Münnix, Andreas Rudloff and Dominic Juhász for their careful reviewing of this paper and numerous hints which helped to improve the presentation of the material.

References

1. M.P. Atkinson and P. Bunemann. Types and persistence in database programming languages. *ACM Computing Surveys*, 19(2), June 1987.
2. M.P. Atkinson, K.J. Chisholm, and W.P. Cockshott. PS-algol: An algol with a persistent heap. *ACM SIGPLAN Notices*, 17(7), July 1981.
3. D.G. Bobrow, L.G. De Michiel, R.P. Gabriel, S.E. Keene, G. Kiczales, and D.A. Moon. Common lisp object system specification. *ACM SIGPLAN Notices*, 23, September 1988.
4. L. Cardelli. Amber. In *Combinators and Functional Programming Languages*, volume 242 of *Lecture Notes in Computer Science*. Springer-Verlag, 1986.
5. L. Cardelli. Typeful programming. Technical Report 45, Digital Equipment Corporation, Systems Research Center, Palo Alto, California, May 1989.
6. L. Cardelli. The Quest language and system (tracking draft). Technical report, Digital Equipment Corporation, Systems Research Center, Palo Alto, California, 1990. (shipped as part of the Quest V.12 system distribution).
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. A.J. Field and P.G. Harrison. *Functional Programming*. Addison-Wesley Publishing Company, 1988.
9. A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley Publishing Company, 1983.
10. P. Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411, September 1989.
11. Ichbiah et al. The programming language Ada: Reference manual. Technical Report MIL-STD-1815A-1983, ANSI, 1983.
12. ISO/IEC JTC1/SC22/WG13. *Interim Version of the 4th Working Draft Modula-2 Standard*, 1991.
13. B.W. Kernighan and D.M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, New Jersey, 1977.
14. F. Matthes. *Persistente Objektsysteme: Integrierte Datenbankentwicklung und Programmerstellung*. Springer-Verlag, 1993. (In German).
15. F. Matthes, R. Müller, and J.W. Schmidt. Object stores as servers in persistent programming environments – the P-Quest experience. FIDE Technical Report Series FIDE/92/48, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, July 1992.
16. 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).

17. 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.
18. F. Matthes and J.W. Schmidt. System construction in the Tycoon environment: Architectures, interfaces and gateways. In P.P. Spies, editor, *Proceedings of Euro-Arch'93 Congress*, pages 301–317. Springer-Verlag, October 1993.
19. M. Mauny. Functional programming using CAML. Technical report, INRIA, Domaine de Voluceau, Rocquencourt 78153 Le Chesnay Cedex, France, September 1991.
20. J. Minker. *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann Publishers, 1988.
21. R. Müller. Language processors and object stores: Interface design and implementation. Master's thesis, Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt, Germany, November 1991. (In German).
22. G. Nelson, editor. *Systems programming with Modula-3*. Series in innovative technology. Prentice Hall, Englewood Cliffs, New Jersey, 1991.
23. 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.
24. J.W. Schmidt and F. Matthes. Language technology for post-relational data systems. In A. Blaser, editor, *Database Systems of the 90s*, volume 466 of *Lecture Notes in Computer Science*, pages 81–114, November 1990.
25. D. Stemple, R.B. Stanton, T. Sheard, P. Philbrow, R. Morrison, G.N.C. Kirby, L. Fegaras, R.L. Cooper, R.C.H. Connor, M.P. Atkinson, and S. Alagic. Type-safe linguistic reflection: A generator technology. Research Report CS/92/6, University of St. Andrews, Department of Computing Science, July 1992.
26. N. Wirth. The programming language Oberon. Technical report, Department Informatik, ETH Zürich, Switzerland, 1987.