Persistent Polymorphic Programming in Tycoon:
An Introduction*

Florian Matthes       Sven Müßig       J.W. Schmidt

Universität Hamburg
Vogt-Kölln Straße 30
D-22527 Hamburg, Germany
{matthes,muessig,J_Schmidt}@informatik.uni-hamburg.de

Abstract

This text provides an introduction to Tycoon\(^1\), an open persistent polymorphic pro-
gramming 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.

ACKNOWLEDGEMENTS

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 authors 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.

*This research has been supported in part by the Esprit-III Basic Research Project FIDE-2 and by a grant
from the German-Israel Foundation for Scientific Research and Development.

\(^1\)Tycoon: Typed Communicating Objects in Open Environments.
# Contents

1 Introduction and Motivation  
   1

2 Language Classification  
   2

3 Lexical and Syntactical Rules  
   3
   3.1 Symbols  
   3.2 Reserved Keywords  
   3.3 Comments  
   3.4 Factoring of Expressions  
   3.5 Coercion and Overloading  
   4

4 Predefined Values and Functions  
   5
   4.1 Naming, Binding, and Typing  
   4.2 Literals  
   5

5 User-defined Values and Functions  
   6
   5.1 Static Bindings  
   5.2 Dynamic Bindings  
   7

6 Predefined Value and Type Constructors  
   10
   6.1 Tuple Types  
   6.2 Variant Types  
   6.3 Record Types  
   6.4 Recursive Data Types  
   6.5 Dynamic Data Types  
   15

7 Subtype Relationships and Subtype Polymorphism  
   15
   7.1 Subtyping on Predefined Types  
   7.2 Subtyping on Tuple Types  
   7.3 Subtyping on Record Types  
   7.4 Subtyping on Function Types  
   18

8 Parametric Polymorphism  
   19
   8.1 Polymorphic Functions  
   8.2 Bounded Parametric Polymorphism  
   8.3 Type Operators  
   8.4 Abstract Data Types  
   23

9 Imperative Programming  
   24
   9.1 Mutable Variables  
   9.2 Subtyping Rules for Mutable Bindings  
   9.3 Control Structures  
   9.4 Arrays and Array Indexing  
   31

10 Multi-Paradigm Programming in Tycoon  
   31
1 Introduction and Motivation

The Tycoon\(^2\) 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 Tycoon language TI\(^3\) described in this paper is used for the following two activities in database application programming (see also [MS93]):

**Strongly typed, high-level application programming:** TI 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. TI 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 typed system model.

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

\(^2\)Tycoon: Typed Communicating Objects in Open Environments.
\(^3\)TI: Tycoon Language.
Orthogonal persistence abstraction: Tt 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 [SSS+92]. For example, the Tt programming environment exports a (strongly-typed) function reflect.optimize that takes a Tt function value, re-invokes the Tt 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 [Mat93] and [MS93].

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.

Appendix A and B are intended mainly for reference purposes and summarize the syntax and the predefined identifiers of the language. Readers interested in the formal definition of the Tt semantics are referred to [MS92].

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 Tt language classification.

The programming language Tt evolved from the experimental languages Quest4 [Car89, Car90] and P-Quest5 [Mat91, Mül91, NMM92]. All semantic concepts of these languages are supported in Tt (in a slightly variated syntactic form). Tt 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.

4 Quest: Quantifiers and Subtypes.

5 P-Quest is a Quest System extended by an orthogonal persistence concept.
The syntactic structure and the module concept of Tl are similar to those of the languages of the Modula family (Modula-2 [MOD91], Oberon [Wir87], Modula-2+ [RLW85], Modula-3 [Ne91], and Ada [1±83]). Regarding its semantics, Tl is more closely related to the polymorphic functional languages of the ML language family [Car89, Car90, Mau91, FH88, Hud89]. The semantic concepts of Tl are derived from the language F ≤ [CMMS91], a widely accepted formal basis for the study of modern type systems.

Like C [KR77] 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 [BDMG88] and commercial object-oriented languages like Smalltalk [GR83]. From integrated database programming languages like PS-Algo [ACC81], Napier88 [DCBM89], Amber [Car86], and P-Quest, mentioned before, Tycoon inherits the orthogonality of elementary kernel concepts for persistence abstraction, type-complete data structuring, and iteration abstraction [AB87, SM90].

Motivated by an analysis of the conceptual and technological foundations of existing database languages [MS91a, MS91b], 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) [MS91b].

Tl enables the programmer to use different modeling styles. Functional and imperative programming are supported directly. Due to the considerable linguistic neutrality several variants of the object-oriented programming style are supported by Tl. Relational and logic-based programming [Min88] 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 developments 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 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. A precise definition of the syntax of Tl is given in appendix A.

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

Appendix A.3 lists all reserved keywords and infix symbols of Tl. These identifiers 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. A proposal for the layout and for further naming conventions in Tl programs is described in section 14.

### 3.3 Comments

In Tl, comments are enclosed in (* 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.

\[
3 - 7 * 4 \\
\{3 - 7\} * 4 \\
\{(3 - 7) * 4\} \\
\text{int.mul(int.sub(3 7) 4)} \\
\Rightarrow -16 :\text{Int}
\]

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 (see appendix B for details).

\[
2 + 7
\]

⇒ 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 (see appendix B). 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 (compare appendix A.2).

\[
3 \sim 3.0 \ 'c' \ "string" \ true
\]

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 [Mat93].

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.

\[ \text{let } n = 10 \]

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.

\[ \text{let } x = 1 + \{2 \times n\} \]
\[ \Rightarrow 21 : \text{Int} \]

Sequences of bindings are interpreted as sequential bindings in TL.

\[ \text{let } n = 10 \]
\[ \text{let } x = 1 + \{2 \times n\} \]

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 \textbf{and}.

\[ \text{let } a = 4 \]
\[ \text{let } a = 123 / 3 \textbf{ and } b = a + 2 \textbf{ and } c = true \]
\[ \text{ begin let } a = 'x' \textbf{ let } b = a \textbf{ end } \]
\[ \text{ let } c = a \]

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.

\[ \text{let } a = 1.0 \]
\[ \text{begin let } a = 'x' \textbf{ let } b = a \textbf{ end } \]
\[ \text{let } c = a \]

The scope of the local identifiers \( a \) and \( b \) is restricted to the block delimited by the keywords \textbf{begin} and \textbf{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 an illustrative examples.

\[ a : \text{Int} \quad b : \text{Bool} \]
\[ a : \text{Int} \quad : \text{Bool} \]
\[ : \text{Int} \quad : \text{Bool} \]
\[ (* \text{empty signature} *) \]

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.

\[ \text{let} \quad a : \text{Int} = 3 \]
\[ \text{let} \quad b : \text{Bool} = \text{true} \]

Recursive bindings are used for the construction of recursive and cyclic data structures. In Ti, 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 Ti, since identifiers can only be introduced in bindings and, furthermore, recursive bindings are subject to static constraints that avoid access to uninitialized variables ([Mat93]).

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, Ti 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 Ti, 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.

\[ \text{fun} (x : \text{Int}) \quad x + 1 \]
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.

```haskell
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.

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

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

```haskell
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.

```haskell
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 (\( \text{string.concat} \)) is bound to the infix symbol `<>`.

```haskell
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:

```haskell
<>("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.

```haskell
begin
let a = 3
<>("concat" "enation")
end
```
The above expression causes a syntax error since the compiler recognizes an expression of the form \(3 \langle\rangle\) ("concat" "enation"). As usual, such problems can be avoided by the use of brackets to control the parsing of expressions.

\[
\begin{align*}
\text{begin} \\
\text{let } a = 3 \\
\{\langle\rangle\} ("\text{concat" } \text{"enation")} \\
\text{end}
\end{align*}
\]

5.2.2 Recursive Functions

TI supports the definition of recursive functions. Recursive bindings introduced by the keyword \texttt{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.

\[
\text{let rec } \text{fac}(n : \text{Int}) : \text{Int} = \text{if } n == 0 \text{ then } 1 \text{ else } n \ast \text{fac}(n - 1) \text{ end}
\]

As mentioned above, TI 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 \texttt{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 \texttt{parallel}. In TI the bindings are connected by the keyword \texttt{and} for this purpose. A parity test is given as an illustrative example.

\[
\begin{align*}
\text{let rec } \text{even}(x : \text{Int}) : \text{Bool} &= \\
\text{if } x == 0 \text{ then } \text{true else } \text{odd}(x - 1) \text{ end} \\
\text{and } \text{odd}(x : \text{Int}) : \text{Bool} &= \\
\text{if } x == 0 \text{ then } \text{false else } \text{even}(x - 1) \text{ end}
\end{align*}
\]

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 function values, i.e., the names and types of their formal parameters and the function result type. Function types are introduced by the keyword \texttt{Fun}. The types of the previously defined functions \texttt{succ, add}, and \texttt{succ2} are given as examples.

\[
\begin{align*}
\text{succ :Fun}(x : \text{Int}) : \text{Int} \\
\text{add :Fun}(x : \text{Real } y : \text{Real}) : \text{Real} \\
\text{succ2 :Fun}(x : \text{Int}) : \text{Int}
\end{align*}
\]

The following abbreviating notation is also supported in TI.

\[
\begin{align*}
\text{suc}(x : \text{Int}) : \text{Int} \\
\text{add}(x : \text{Real } y : \text{Real}) : \text{Real}
\end{align*}
\]
5.2.4 Higher-Order Functions

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

The functions \texttt{twice} and \texttt{newInc} are examples of higher-order functions.

\begin{verbatim}
let twice = fun(f : Fun(:Int) :Int a :Int) :Int :Int f(f(a))
let newInc = fun(x :Int) :Fun(:Int) :Int fun(y :Int) :Int x + y
\end{verbatim}

Again, the functions can be written down more concisely.

\begin{verbatim}
let twice(f(:Int) :Int a :Int) = f(f(a))
let newInc(x :Int)(y :Int) = x + y
\end{verbatim}

The function \texttt{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 \texttt{twice}, the function passed as a parameter is applied twice to the second parameter.

\begin{verbatim}
twice(succ 3)
\Rightarrow 5 :Int
twice(fun(x :Int) x * x 3)
\Rightarrow 81 :Int
\end{verbatim}

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

\begin{verbatim}
let add2 = newInc(2)
add2(5)
\Rightarrow 7 :Int
newInc(3)(5)
\Rightarrow 8 :Int
\end{verbatim}

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

\begin{verbatim}
Let Person = Tuple name :String age :Int end
Let IntPair = Tuple :Int :Int end
\end{verbatim}
Tuple values are ordered lists of bindings.

```plaintext
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 by the dot notation.

```plaintext
peter.age
⇒ 3 :Int
```

The rules for type compatibility of Ti. 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.

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

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

In Ti. it is possible to include functions as fields into 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 [Mat93].

### 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.

```plaintext
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.

```plaintext
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.

\[ \text{Let } \text{Day} = \text{tuple case mon, tue, wed, thu, fri, sat, sun end} \]

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.

\[
\text{let address1 =}
\text{tuple case national of Address1 with}
\text{let street = }"\text{Johnsallee 21}" \\
\text{let city = }"\text{Hamburg}" \\
\text{let zip = 21234}
\text{end}
\]

The keyword \textbf{with} in the definition of \textit{address1} is optional. It is also possible to use
anonymous bindings in tuples with variants.

\[
\text{let address2 =}
\text{tuple case national of Address1}
\text{"Johnsallee 21" "Hamburg" 21234}
\text{end}
\]

A value of type \textit{Day} can be defined as follows.

\[
\text{let today = tuple case mon of Day end}
\]

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.

\[
\text{address1.street} \\
\Rightarrow "\text{Johnsallee 21" :String}
\]

For the fields of the variants a complete (\textbf{case of}) or an incomplete case analysis (\textbf{case})
is necessary.

\[
\text{case of address1}
\text{when national with n then fmt.int(n.zip)}
\text{when international with i then i.zip}
\text{end}
\]

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 exten-
sions of the case analysis. The incomplete case analysis has the following form.

\[
\text{case address1}
\text{when national with n then fmt.int(n.zip)}
\text{end}
\]
Since an incomplete case analysis can lead to runtime errors, an `else`-branch can (and should) be specified in this situation.

```lisp
  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.

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

These two examples are equivalent to the following expressions.

```lisp
  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.

```lisp
  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.

```lisp
  let peter = record let age = 3 let name = "Peter" end
```

Like in 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.

```lisp
  peter.age
  ⇒ 3 :Int
```

Different from 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.
\[ \text{let peterAsStudent } = \text{extend peter with let semester } = 1 \text{ end} \]

The infix operator \( =\) checks the identity of two values.

\[ \text{peter } =\text{ peterAsStudent} \]
\[ \Rightarrow \text{true :Bool} \]

The record value \(\text{peterAsStudent}\) fulfills the following type specifications (see also sec. 7.3)

\[ \text{Record name :String age :Int semester :Int end} \]
\[ \text{Record age :Int name :String semester :Int end} \]
\[ \text{Record semester :Int name :String age :Int end} \]
\[ \text{Record name :String age :Int end} \]

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 realization of recursively defined data structures.

A recursive type definition is introduced by the keyword \texttt{Rec} in Tl. A supertype (e.g., \texttt{IntegerList <: Ok}) has to be specified when defining a recursive type. The definition of a list for integer values is considered as an example.

\[ \text{Let Rec IntegerList <: Ok} = \]
\[ \text{Tuple} \]
\[ \text{case nil} \]
\[ \text{case cons with car :Int cdr :IntegerList end} \]

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.

\[ \text{let emptyList } = \]
\[ \text{tuple case nil of IntegerList end} \]

\[ \text{let singleList } = \]
\[ \text{tuple case cons of IntegerList with} \]
\[ \text{let car } = 7 \]
\[ \text{let cdr } = \text{emptyList} \]
\[ \text{end} \]

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.

\[ \text{let rec circularList :IntegerList } = \]
\[ \text{tuple case cons of IntegerList with} \]
\[ 7 \text{ circularList} \]
\[ \text{end} \]
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 T1, the keywords **Dyn** and **typeofcase** 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 =
  typeofcase 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 [Mat93] and in [Nel91] structural equivalence is a prerequisite for dynamic type checking in persistent, distributed systems.

7 Subtype Relationships and Subtype Polymorphism

In T1, 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** [Mat93]. 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)
g => 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.
\[ \text{Int} <: \text{Ok} \\
\text{Real} <: \text{Ok} \\
\text{Fun}(\text{Int}) : \text{Int} <: \text{Ok} \\
\text{Tuple} : \text{Int end} <: \text{Ok} \]

Non-trivial subtype relationships, e.g., of the form \( \text{Int} <: \text{Real} \) 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, \( \text{directory.T} <: \text{String} \). Values of the type \( \text{directory.T} \) represent syntactically correct path names of the operating systems and thereby also strings. The reverse is not true.

### 7.2 Subtyping on Tuple Types

In \( T_1 \), 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

\[
\begin{align*}
\text{Let Student} &= \text{Tuple} \ \text{name} : \text{String} \ \text{age} : \text{Int} \ \text{semester} : \text{Int end} \\
\text{Let Person} &= \text{Tuple} \ \text{name} : \text{String} \ \text{age} : \text{Int end} \\
\text{Let Car} &= \text{Tuple} \ \text{name} : \text{String} \ \text{age} : \text{Int end} \\
\text{Let Machine} &= \text{Tuple} \ \text{name} : \text{String} \ \text{fuel} : \text{String end} \\
\text{Let NamedThing} &= \text{Tuple} \ \text{name} : \text{String end}
\end{align*}
\]

the following subtype relationships hold

\[
\begin{align*}
\text{Student} &<: \text{Person} \quad \text{Person} <:\text{NamedThing} \\
\text{Car} &<: \text{NamedThing} \quad \text{Machine} <:\text{NamedThing}
\end{align*}
\]

but also

\[
\begin{align*}
\text{Car} &<: \text{Person} \quad \text{Person} <:\text{Car}
\end{align*}
\]

On the other hand, it is described in [Mat93] how the Tycoon subtype relationship can be restricted systematically to explicitly defined sub\( \text{classes} \) ensuring, for example, that \( \text{car.T} \not<: \text{person.T} \) holds.

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

\[
\begin{align*}
\text{Let Student2} &= \text{Tuple} \ \text{name} : \text{Ok} \ \text{age} : \text{Int} \ \text{semester} : \text{Int end}
\end{align*}
\]

implies the following subtype relationship.

\[
\text{Student} <: \text{Student2}
\]

In \( T_1 \), 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 \texttt{Student = Tuple Repeat Person} semester :\texttt{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 \texttt{open} supports the repetition of existing bindings. The value \texttt{peter} defined in section 6.1, for example, can be extended by specifying its semester.

\texttt{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 \texttt{NamedThing} also works on values of the types \texttt{Student, Car, or Machine}.

\texttt{let sameName(x, y :NamedThing) :Bool =}
\texttt{string.equal(x.name y.name)}

\texttt{let flat :Machine = tuple let name = "Uno" let fuel = "unleaded" end}

\texttt{sameName(peter flat)}
\texttt{⇒ 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 \texttt{RGBColor <:Color} holds for

\texttt{Let RGBColor = Tuple case red, green, blue end}
\texttt{Let Color = Tuple case red, green, blue, cyan, yellow end}

and the relationship \texttt{Address <:Address2} holds for the type \texttt{Address} declared in section 6.2 if \texttt{Address2} is defined, for example, as follows:

\texttt{Let Address2 =}
\texttt{ Tuple}
\texttt{ case national with street, city :String zip :Int}
\texttt{ case international with street, city, state :String zip :String}
\texttt{ case unknown}
\texttt{ 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 \texttt{AddressTuple <:Address2}, therefore, holds for

\texttt{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 $\text{Person} = \text{Record}$ name : $\text{String}$ age : $\text{Int}$ end
Let $\text{Student} = \text{Record}$ name : $\text{String}$ semester : $\text{Int}$ end
Let $\text{Employee} = \text{Record}$ ssno : $\text{String}$ salary : $\text{Real}$ end
Let $\text{Tutor} = \text{Record}$ name, ssno : $\text{String}$ age, semester : $\text{Int}$ salary : $\text{Real}$ end

For these types the following subtype relationships hold in $\mathcal{T}_1$,

$\text{Tutor} <: \text{Person}$  $\text{Tutor} <: \text{Student}$  $\text{Tutor} <: \text{Employee}$

enabling the application of functions which are defined for arguments of the types $\text{Person}$, $\text{Student}$, or $\text{Employee}$ to values of the type $\text{Tutor}$. Furthermore, it becomes possible to construct heterogeneous data structures consisting, for example, of values of the types $\text{Person}$, and $\text{Tutor}$. 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 \textit{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 $F_2$ [Mat93].

For example, assuming the relationship $\text{Student} <: \text{Person}$ the relationship $\text{GetrichestStudent} <: \text{GetrichestPerson}$ holds for the following function types because the result types of the function types are in covariance relationship.

Let $\text{GetrichestPerson} = \text{Fun()} : \text{Person}$  Let $\text{GetrichestStudent} = \text{Fun()} : \text{Student}$

However, for the function types

Let $\text{HirePerson} = \text{Fun(} : \text{Person}\text{)} : \text{Ok}$  Let $\text{HireStudent} = \text{Fun(} : \text{Student}\text{)} : \text{Ok}$

the relationship $\text{HirePerson} <: \text{HireStudent}$ 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.

```ml
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.

```ml
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 (type inference).

```ml
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

```ml
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 behavior. Separate functions for each considered parameter type would be necessary for this purpose in languages like C or Modula-2.

19
Much of the power of $\text{T}_1$ results from the possibility to combine 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 could look as follows.

$\textbf{let}$ $\text{sort}(A <: \text{Ok} \ a: \text{Array}(A) \ \text{order}(a, b : A) : \text{Bool}) : \text{Array}(A) = \ldots$

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.

$\textbf{let}$ $\text{older}(a, b : \text{Person}) : \text{Bool} = a.\text{age} \geq b.\text{age}$

The following call of the function $\text{sort}$ sorts an array of persons according to their age.

$\text{sort}(\text{Person personArray older})$

Again, the type parameter can be omitted.

$\text{sort(personArray older)}$

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 parameter to such functions. Employing subtype polymorphism, a function comparing the component age of two values of an arbitrary subtype of the type $\text{Person}$ can be defined in the following way.

$\textbf{let}$ $\text{chooseOlder}(p1, p2 : \text{Person}) : \text{Person} =$
$\textbf{if}$ $p1.\text{age} > p2.\text{age}$ $\textbf{then}$ $p1$ $\textbf{else}$ $p2$ $\textbf{end}$

$\textbf{let}$ $\text{peter : Student} =$
$\textbf{tuple let}$ $\text{name} = "Peter" \ \text{let age} = 24 \ \text{let semester} = 3 \textbf{end}$

$\textbf{let}$ $\text{paul : Student} =$
$\textbf{tuple let}$ $\text{name} = "Paul" \ \text{let age} = 29 \ \text{let semester} = 6 \textbf{end}$

$\text{chooseOlder(peter paul)}$
$\Rightarrow \textbf{tuple let}$ $\text{name} = "Paul" \ \text{let age} = 29 \textbf{end} : \text{Person}$

Considering the function result displayed by the system, it can be seen that the attribute $\text{semester of paul}$ 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, $\text{chooseOlder}$ has to be defined as a polymorphic function.
let chooseOlder(\(P <\text{: Person} \ p1, p2 :P\)) \ = 
\[
\begin{cases} 
\text{if } p1.\text{age} > p2.\text{age} \ \text{then} \ p1 \ \text{else} \ p2 
\end{cases}
\]

\(\text{chooseOlder(peter paul)}\)
\(\Rightarrow \text{tuple name = "Paul" age = 29 semester = 7 end :Student}\)

As a result of integrating enough type information into the function definition, all attributes are taken into account. By introducing the specification \(P < :\text{Person}\), the function \(\text{chooseOlder}\) is made polymorphic, but in contrast to the unrestricted parametric polymorphism, the polymorphism is restricted to subtypes of the type \(\text{Person}\). 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 behavior. 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.

\[
\text{Let } \text{Id} = \text{Oper}(A < :\text{Ok}) \ A
\]

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

\[
\text{Let } \text{Id}(A < :\text{Ok}) = A
\]

The operator \(\text{Id}\) maps the type passed as parameter to itself.

\(\text{:Id(Int)}\)
\(\Rightarrow :\text{Int}\)
\(\text{:Id(Id(Boolean))}\)
\(\Rightarrow :\text{Boolean}\)

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

\[
\text{Let } \text{Opt}(A < :\text{Ok}) =
\text{Tuple(A < :Ok)} =
\text{case nil}
\text{case notNil with val :A}
\text{end}
\]

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

\(\text{Opt(Person)}\)
\(\text{Id(Opt(Id(Person)))}\)
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.

\[ \text{Let } \rightarrow (X, Y <: \text{Ok}) = \text{Fun}(X) : Y \]
\[ \text{Let } * (X, Y <: \text{Ok}) = \text{Tuple } \text{lst} : X \text{ snd } : Y \text{ end} \]
\[ \text{Let } +(X, Y <: \text{Ok}) = \text{Tuple case } \text{lst with } x : X \text{ case snd with } y : Y \text{ end} \]

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-parametrized types passed as parameters. The coding of case selections on the type level is considered as an illustrative example.

\[ \text{Let } \text{Boolean(Then, Else } <: \text{Ok}) = \text{Ok} \]
\[ \text{Let } \text{True(Then, Else } <: \text{Ok}) = \text{Then} \]
\[ \text{Let } \text{False(Then, Else } <: \text{Ok}) = \text{Else} \]
\[ \text{Let } \text{Cond(If } <: \text{Boolean Then, Else } <: \text{Ok}) = \text{If(Then Else)} \]

\[ \text{let } i : \text{Cond(True Int String)} = 3 \]
\[ \text{let } s : \text{Cond(False Int String)} = "Peter" \]

### 8.3.2 Recursive Type Operators

In addition to recursive types TL supports recursive type operators. The type operator \text{List} maps a type \text{E} to the type of a list with elements of type \text{E}. For this purpose a type parameter is introduced into the definition of lists presented in section 6.4.

\[ \text{Let Rec } \text{List}(A <: \text{Ok}) <: \text{Ok} = \]
\[ \text{Tuple} \]
\[ \text{case } \text{nil} \]
\[ \text{case } \text{cons with car :A cdr :List(A)} \]
\[ \text{end} \]

The corresponding list operations can be implemented by polymorphic functions.

\[ \text{let } \text{new(A } <: \text{Ok) :List(A)} = \]
\[ \text{tuple case nil of List(A)} \text{ end} \]

\[ \text{let } \text{cons(A } <: \text{Ok head :A tail :List(A)) :List(A)} = \]
\[ \text{tuple case cons of List(A) with head tail end} \]

The polymorphic functions \text{new} generates empty lists of arbitrary types. Elements can be added at the beginning of the list by the function \text{cons}.

In contrast to traditional 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 behavior.
8.4 Abstract Data Types

An abstract data type (ADT) consists of a data type and of a set of operations defined on 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.

\[ \begin{align*}
\text{Let Stack =} \\
\text{Tuple} \\
T(E <: \text{Ok}) <: \text{Ok} \\
\text{new}(E <: \text{Ok}) : T(E) \\
\text{empty}(E <: \text{Ok} \text{ stack : } T(E)) : \text{Bool} \\
\text{push}(E <: \text{Ok} \text{ element : } E \text{ stack : } T(E)) : T(E) \\
\text{pop}(E <: \text{Ok} \text{ stack : } T(E)) : T(E) \\
\text{top}(E <: \text{Ok} \text{ stack : } T(E)) : E \\
\end{align*} \]

In the following an implementation of this interface based on the module list\(^6\) is presented.

\[ \begin{align*}
\text{let listStack : Stack =} \\
\text{tuple} \\
\text{Let } T(E <: \text{Ok}) <: \text{Ok} = \text{list} \cdot T(E) \\
\text{let } \text{new}(E <: \text{Ok}) : T(E) = \text{list} \cdot \text{new}(E) \\
\text{let } \text{empty}(E <: \text{Ok} \text{ stack : } T(E)) : \text{Bool} = \text{list} \cdot \text{empty}(\text{stack}) \\
\text{let } \text{push}(E <: \text{Ok} \text{ element : } E \text{ stack : } T(E)) : T(E) = \\
\text{ } \text{ } \text{list} \cdot \text{cons}(\text{element stack}) \\
\text{let } \text{pop}(E <: \text{Ok} \text{ stack : } T(E)) : T(E) = \text{list} \cdot \text{tail}(\text{stack}) \\
\text{let } \text{top}(E <: \text{Ok} \text{ stack : } T(E)) : E = \text{list} \cdot \text{head}(\text{stack}) \\
\end{align*} \]

In this example, \(T\) is defined as a type operator and the operations \text{new}, \text{empty}, \text{push}, \text{pop}, and \text{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 \(T1\). As an example, the signature of the type operator in the ADT \text{Stack} above can be modified in order to exhibit more information about the implementation of the ADT.

\[ \begin{align*}
\text{Let Stack2 =} \\
\text{Tuple} \\
T <: \text{list} \cdot T \\
\end{align*} \]

\(^6\)\text{list} is a module of the standard library implementing polymorphic lists and the according operations.
... 
end

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 TI, 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 (compare section 11.1).

9 Imperative Programming

The discussions in the previous sections are restricted to the functional concepts underlying the language TI. 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 TI, 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

24
\[ x := y \]
\[ \Rightarrow \text{ok : Ok} \]
\[ x \ y \]
\[ \Rightarrow 4 : \text{Int} \ 4 : \text{Int} \]

\[ y := 5 \]
\[ \Rightarrow \text{ok : Ok} \]
\[ x \ y \]
\[ \Rightarrow 4 : \text{Int} \ 5 : \text{Int} \]

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

\textbf{tuple var "text" var 3 end}

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

\[ :=(A <:\text{Ok} \ \text{var } \text{Value :A} \ \text{rValue :A}) : \text{Ok} \]

The signature of the function defines that the assignment evaluates to the trivial value \texttt{ok} of type \texttt{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.

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

\texttt{let swap(A <:\text{Ok} \ \text{var } x, y :A) =}
\[ \begin{align*}
\text{begin} \ & \text{let } \text{tmp} = x \ x := y \ y := \text{tmp} \ \text{end}
\end{align*} \]

\texttt{let var a = 3 and var b = 5}

\texttt{swap(:Int a b)}

\[ \begin{align*}
a \ b \quad \Rightarrow 5 : \text{Int} \ 3 : \text{Int}
\end{align*} \]

When applied, the function \texttt{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 (\textit{shared variables}). In the following example, the variable \texttt{state} can be updated and inspected, respectively, by the three defined functions only.

\texttt{let newCounter() =}
\[ \begin{align*}
\text{begin} \ & \text{let } \text{var } \text{state} = 0 \\
\text{tuple} \ & \text{let } \text{reset()} : \text{Ok} = \text{state} := 0 \\
& \text{let } \text{inc()} : \text{Ok} = \text{state} := \text{state} + 1
\end{align*} \]
let value() : Int = state
end
end

let cnt1 = newCounter() and cnt2 = newCounter()
cnt1.inc() cnt1.value() cnt2.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. T₁ 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 T₁ [Mat93]:

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 T₁ 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 T₁ 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, T₁ offers control structures for conditional expressions, three kinds of loops and a 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 T1. 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 conjuncted by andif and orif operators in T1.

```
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 as, 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 (compare section 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

28
Two versions of enumerating loops are distinguished in T1, one counting upwards (**upto**) and the other counting downwards (**downto**).

```plaintext
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 T1, it is also integrated smoothly into the type system. Exceptional situations that have to be handled cannot only be caused by partially defined functions as, for example, `int.div` (division by zero), but also by an overflow of a partially represented domain. Furthermore, the projection on variants described in section 6.2 can raise exceptions in T1.

Each exception returns an exception package containing a string which supports the identification of the exception on the top level.

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

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

```plaintext
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, T1 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**.

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

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

```plaintext
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 reraise 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-parametrized 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"

30
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 T1, 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 T1.

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. T1 can be used to support programming styles, which differ substantially from each other (see [Mat93]). This is illustrated here by the realization 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 realized using the T1 primitives for naming, binding, and typing.
In order to implement the concept of abstract data types the primitives of $T_1$ can be combined in three different ways. It is possible to aggregate an opaque type together with the functions working on this type\(^7\). This approach can be implemented purely functionally or state-based resulting in the first two realization variants. The third variant aggregates methods that work on a hidden, internal state. The different realization alternatives are called

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

A generic stack implementation is used to compare the three different realization 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 realization the modified stack is returned by the the update operations. Therefore, the result type of these functions is the opaque type $T$:

\[
\text{Let FunStack (E <: Ok) =}
\]

\[
\text{Tuple}
\]

\[
T <: Ok
\]

\[
\text{new()} : T
\]

\[
\text{empty(stack : T) : Bool}
\]

\[
\text{push(element : E stack : T) : T}
\]

\[
\text{pop(stack : T) : T}
\]

\[
\text{top(stack : T) : E}
\]

\[
\text{end}
\]

The data type is implemented by a parametrized 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.

\[
\text{let listStack (E <: Ok) : FunStack(E) =}
\]

\[
\text{tuple}
\]

\[
\text{Let } T <: Ok = \text{list.T(E)}
\]

\[
\text{let new() : T = list.new(:E)}
\]

\[
\text{let empty(stack : T) : Bool = list.empty(stack)}
\]

\[
\text{let push(element : E stack : T) : T = list.cons(element stack)}
\]

\[
\text{let pop(stack : T) : T = list.tail(stack)}
\]

\(^7\)This is comparable to an implementation of abstract types in Modula-2 using the module concept of this language.
```plaintext
let top(stack :T) :E = list.head(stack)
end
```

Using this realization a new stack with elements of type `Int` containing the element 4 can be created by the following function call:

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

**Imperative Encapsulation** In the imperative realization 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 result. 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.

```plaintext
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:

```plaintext
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 realized (see [Mat93]). For this reason this encapsulation method is called object-oriented in [Mat93]. It avoids an opaque type and input parameters of this type\(^8\).

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 realization 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 restrict deliberately existing concepts of Tl [Mat93].

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.

\(^8\)Operations expecting more than one input parameter of the opaque type lead to recursive type definitions when this implementation style is used [Mat93].
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

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 identifier.

A module defines a tuple value aggregating bindings according to its interface. In T1, 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 on 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.Any T = 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 Tt.

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. Only modules and interfaces preceding the importing modules and interfaces in the library can be imported. 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

36
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 Ti. 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).

Ti 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 stabilizing the object store. The module store provides the function stabilise for this purpose. A call of this function stabilizes 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();
```
l := list.cons(3, 1)
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\(^9\) 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.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>library</td>
<td>identifies the core of the Tycoon runtime system</td>
</tr>
<tr>
<td>cLibrary</td>
<td>identifies the standard C library</td>
</tr>
<tr>
<td>dynamicLibrary</td>
<td>identifies dynamically bound libraries</td>
</tr>
<tr>
<td>staticLibrary</td>
<td>identifies statically bound libraries</td>
</tr>
</tbody>
</table>

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

---

\(^9\)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.
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 which is mandatory. The following table contains the set of characters that denote parameter formats.

<table>
<thead>
<tr>
<th>Format</th>
<th>TL type</th>
<th>C type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>Int</td>
<td>long</td>
<td>integer number</td>
</tr>
<tr>
<td>r</td>
<td>Real</td>
<td>double</td>
<td>floating point number</td>
</tr>
<tr>
<td>c</td>
<td>Char</td>
<td>char</td>
<td>ASCII character</td>
</tr>
<tr>
<td>b</td>
<td>Bool</td>
<td>long</td>
<td>boolean value (see text)</td>
</tr>
<tr>
<td>s</td>
<td>String</td>
<td>char *</td>
<td>zero-terminated string</td>
</tr>
<tr>
<td>v</td>
<td>Ok</td>
<td>void</td>
<td>return value only</td>
</tr>
<tr>
<td>=</td>
<td>&lt;:Ok</td>
<td>void *</td>
<td>Tycoon value, no conversion</td>
</tr>
<tr>
<td>w</td>
<td>word.T</td>
<td>void *</td>
<td>32-bit word</td>
</tr>
</tbody>
</table>

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.

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 reestablished automatically.
13.2 Function Calls from External C Libraries to Tycoon

The programming interface between T1 and C is bidirectional. It is not only possible to call
C functions from T1, but also to call back from C to T1.

In order to minimize the programming effort for callbacks on the C side, it is desirable to
make T1 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 T1
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 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 T1 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 fnt

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

```c
#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 behavior (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:

```c
#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);
}
```

14 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 re-learning 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].

41
This section offers a complete set of conventions for formatting TI 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.

14.1 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. `longName-ForAValue`).

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.

14.2 Punctuation

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

```latex
3u+4A "con"u<>u"cat"
let a_w:=_w3
a_w:=_w5
```

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

```latex
let p_w:Person = ... 
let n_w:<Ok = ...
```

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

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

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

```latex
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 square brackets and left and right curly brackets.

```latex
fac(3) get(peter).name
{3 + 4}
```

Two spaces appears between two signatures, e.g. in function or tuple signatures. Two spaces also appears between `let`-bindings function applications or tuple fields.
get(E <:Ok, col : 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 fields (anonymous bindings).

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

14.3 Indentation

Indenting is used to emphasize program structure. Each nesting level is two spaces wide. A binding or signature sequence is indented under the construct that introduces it. For example:

let t =
tuple
  let name = "Peter"
  let age = 3
end
Let Person =
  Tuple
    name : String
    age : Int
end
if bool then
  ss
elsif bool then
  ss
else
  ss
end

Function signatures and function parameter lists that do not fit on a single line are split element-wise across separate lines. Subsequent lines are indented one nesting level, for example:

let newSubWindowWithTitle(window : window.T
  title : String
  windowOptions : window.Options)
newSubWindowWithTitle(longExpressionWithManyArguments
  longExpressionWithManyArguments
  longExpressionWithManyArguments)

These forms only apply to constructs 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. Similarly, 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 \texttt{end}.

\begin{verbatim}
if z > 10 then x := z \ y := z end
for i = 1 \upto\ 10 do a[i] := 0 end
\end{verbatim}

14.4 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.

\begin{verbatim}
(* 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. *)
\end{verbatim}

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.

\begin{verbatim}
... (* Exceptions:
       This is a comment for a group of items. *)

error, overflow :\texttt{Exception} with data :\texttt{Int end}
(* Standard exceptions. *)

test :\texttt{Exception}   (* Only for debugging. *)
...
\end{verbatim}

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.

\begin{verbatim}
\texttt{interface Editor}
(* System: editenv
 File: Editor.ti
 Author: Florian Matthes, Sven Muessig
 Date: 02-dec-91
 Purpose: Generic data editor and browser. *)

\texttt{import}
...

\texttt{export}
...

\texttt{end}
\end{verbatim
A The TL Grammar

A.1 Syntax Notations

The following notation is used for the definition of syntactical and lexical elements. Id denotes a non-terminal symbol (a meta variable) and A and B denote syntactic expressions.

\[ \text{Id}_1, \ldots, \text{Id}_n := A; \text{ the non-terminal symbols } \text{Id}_i \]

- are defined as A (n ≥ 1)
- \text{Id} a non-terminal symbol
- \text{if} a terminal symbol
- "x" the character x ("" is the empty string
- "" is a double quote)
- (A) means A
- A B means A followed by B (binds strongest)
- A | B means A or B
- [A] means ("" | A)
- {A} means ("" | A {A})

A.2 Symbols

The source text of a TL program consists of a sequence of characters that is converted into a sequence of symbols of the categories \textit{int}, \textit{real}, \textit{longreal}, \textit{char}, \textit{string}, \textit{identifier}, \textit{infix}, \textit{colonInfix} and \textit{delimiter}.

The set of formatting characters is an implementation-dependent subset of the non-printable characters and includes at least the characters space, tab, carriage return, line feed and vertical tab.

Comments are sequences of arbitrary printable or formatting characters that are enclosed by (* *). Comments can be nested.

To read a symbol, all formatting characters are skipped and then the longest sequence of characters that forms a symbol is read. Therefore, a space in the source text is only required between two identifiers or two (colon-) infix symbols that appear in direct succession.

\[ \text{int} := \]

\[ ["-"] \text{ digit } \{ \text{ digit } \}; \]

\[ \text{real} := \]

\[ \text{int } \".\" \text{ digit } \{ \text{ digit } \} | \]

\[ \text{int } [\".\" \text{ digit } \{ \text{ digit } \}] \"E\" \text{ int}; \]

\[ \text{longreal} := \]

\[ \text{int } [\".\" \text{ digit } \{ \text{ digit } \}] \"D\" \text{ int}; \]

\[ \text{char} := \]

\[ \"\" \{ \text{ digit } | \text{ alpha } | \text{ special } | \text{ escape } | \text{ delimiter } | \text{ reserved } \} \"\"; \]

\[ \text{string} := \]

\[ \"\" \{ \text{ digit } | \text{ alpha } | \text{ special } | \text{ escape } | \text{ delimiter } | \text{ reserved } \} \"\"; \]

\[ \text{infix} := \]

\[ \text{special } \{ \text{ special } \}; \]

\[ \text{colonInfix} := \]

\[ \":\" \{ \text{ special } \}; \]

45
identifier ::= 
    alpha { digit | alpha }; 

delimiter ::= 
    \n | \t | \x | \f | \' | " | \ \n
alpha ::= 
    A | B | ... | Z | a | b | ... | z;

reserved ::= 
    \n | \t | \x |

special ::= 
    \n | \t | \x | \f | \' | " | \ \n
escape ::= 
    \n ( "n" | "t" | "r" | "f" | "\" | "n" | "n" | "n" | digit digit digit );

Escape characters in character and string literals are interpreted as follows:

<table>
<thead>
<tr>
<th>Character</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\n</td>
<td>new line</td>
</tr>
<tr>
<td>\t</td>
<td>tab</td>
</tr>
<tr>
<td>\x</td>
<td>carriage return</td>
</tr>
<tr>
<td>\f</td>
<td>form feed</td>
</tr>
<tr>
<td>'</td>
<td>'</td>
</tr>
<tr>
<td>&quot;</td>
<td>&quot;</td>
</tr>
<tr>
<td>\</td>
<td>\</td>
</tr>
</tbody>
</table>
| \n \n \n | A single character with the code \n \n \n (three decimal digits that denote an integer in the interval [0,255])
| \f \f \f | The sequence of formatting characters \f is ignored |

The last rule allows the programmer to define string literals that exceed the length of a single source line.

The above definitions make a scanner implementation possible that requires just a single character lookahead.

### A.3 Reserved Keywords

The following identifiers and (colon-) infix symbols are reserved keywords and cannot be used as user-defined identifiers in T# programs.

and and if assert begin case do down to else elsif end exception
exit export extend for fun hide if import in interface let library
loop module of ok open or if raise rec record reraise then try
tuple type case when while with

Dyn Exception Fun Let Nok Ok Oper Rec Record Repeat Tuple

= <= : ?!
A.4 Productions

A.4.1 Compilation Units

Based on the symbols and keywords defined in the previous sections, the grammar of TL is described by the following productions that define a non-ambiguous LL(1) grammar. Unit is the root production for the language.

Unit::=
   ( Library | Interface | Module | Import | Bindings ) ";;";
Library::=
   library identifier Import with { ComponentSignatures }
   [ hide { identifier } ] end;
ComponentSignatures::=
   library { identifier } |
   interface { identifier } |
   module { identifier "::" identifier };
Interface::=
   interface identifier Import export Signatures end;
Module::=
   module identifier Import export Bindings end;
Import::=
   [ import { [ "::" ] identifier } ];

A.4.2 Bindings

Bindings::=
   { TypeBindings | ValueBindings | open ValueId { "::" Type } "::" [ Dyn ] Type | [ var ] Value };
TypeBindings::=
   { Let | Rec | TypeBinding { and TypeBinding } };
TypeBinding::=
   [ Dyn ] TypeId Parameters [ "::" Type ] ":=" Type;
ValueBindings::=
   { let | rec | ValueBinding { and ValueBinding } };
ValueBinding::=
   [ var ] ValueId Parameters [ "::" Type ] ":=" Value;

A.4.3 Values

Value::=
   Value1 { ( orif | andif | colonInfix ) Value1 };
Value1 ::= Value2 { infix Value2 };
Value2 ::= Value3 { "(" Bindings ")" | "?:" CaseId | ":" CaseId |
   ":" FieldId | "[" Value "]" |
   of Bindings Location end };

Values ::= 
   "?" Value "?" | 
   ValueId | 
   ok | 
   start | char | string | real | longreal | 
   fun "(" Signatures ")" | 
   Type Location Value | 
   tuple Location [ case CaseId of Type [ with ] ] Bindings end | 
   record Location Bindings end | 
   extend Value with Bindings end | 
   array Location Bindings end | 
   exception Value [ with Signatures end ] | 
   begin Location Bindings end | 
   if Value then Bindings { elseif Value then Bindings } | 
   else Bindings | 
   case [ of ] Value { when CaseIdList [ with ValueId ] then Bindings } | 
   else Bindings | 
   typcase { ValueId "," } TypeId { when Type then Bindings } | 
   else Bindings | 
   loop Bindings end | 
   exit | 
   while Value do Bindings end | 
   for ValueId "=" Value [ upto | downto ] Value do Bindings end | 
   try Bindings { when Value [ with ValueId ] then Bindings } | 
   else Bindings | 
   raise Value [ with Bindings end ] | 
   reraise | 
   assert Value;

Location ::= 
   [ in Value ];

A.4.4 Signatures

Signatures ::= 
   { TypeSignatures | ValueSignatures | TypeBindings | Repeat Type };

TypeSignatures ::= 
   [ Dyn ] [ TypeIdList Parameters ] "<:" Type;

ValueSignatures ::= 
   [ var ] [ ValueIdList Parameters ] ";" Type;

Parameters ::= 
   { "(" Signatures ")" };

A.4.5 Types

Type ::= 
   Type1 { colonInfix Type1 };

Type1 ::= 
   Type2 { infix Type2 };

48
Type₂ ::= 
   Type₁ { "(" { Type } ")" };
Type₃ ::= 
   "{" Type "}" |
   { ValueId "." } TypeId |
   Ok | Nok |
Fun "(" Signatures ")" ":" Type |
Tuple Signatures { case CaseIdList [ with Signatures ] } end |
Record Signatures end |
Exception [ with Signatures end ] |
Opers "(" Signatures ")" [ ":" Type ] Type;

A.4.6 Identifier

ValueIdList, TypeIdList, CaseIdList ::= 
   Ide { "," Ide };
Ides, ValueId, TypeId, FieldId, CaseId ::= 
   identifier | infix | colonInfix | ":{" Ide "}";

B Predefined Identifiers

The following tables list all identifiers of T₁ that are accessible without explicit import. These identifiers are not reserved keywords and can be rebound (locally) in T₁ programs. They are introduced and explained in more detail in the T₁ bootfiles boot.tyc and tycoon.tyc.

B.1 Type Identifiers

<table>
<thead>
<tr>
<th>Type</th>
<th>Supertype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bool</td>
<td>Ok</td>
<td>type of boolean literals</td>
</tr>
<tr>
<td>Char</td>
<td>Ok</td>
<td>type of character literals</td>
</tr>
<tr>
<td>Int</td>
<td>Ok</td>
<td>type of integer number literals</td>
</tr>
<tr>
<td>Real</td>
<td>Ok</td>
<td>type of floating point number literals</td>
</tr>
<tr>
<td>String</td>
<td>Ok</td>
<td>type string literals</td>
</tr>
<tr>
<td>Locality</td>
<td>Ok</td>
<td>type of locality values</td>
</tr>
<tr>
<td>Array</td>
<td>Oper(E &lt;:Ok) Ok</td>
<td>arrays with mutable elements of type E</td>
</tr>
</tbody>
</table>

B.2 Value Identifiers

<table>
<thead>
<tr>
<th>Value</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>false, true</td>
<td>Bool</td>
<td>boolean values</td>
</tr>
<tr>
<td>somewhere</td>
<td>Locality</td>
<td>locality value</td>
</tr>
</tbody>
</table>
### B.3 Infix Functions

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>:=</td>
<td><code>Fun(A &lt;: Ok var : A : A) : Ok</code></td>
<td>destructive assignment</td>
</tr>
<tr>
<td>==</td>
<td><code>Fun(A &lt;: Ok : A : A) : Bool</code></td>
<td>identity test</td>
</tr>
<tr>
<td>!=</td>
<td><code>Fun(A &lt;: Ok : A : A) : Bool</code></td>
<td>negation of ==</td>
</tr>
<tr>
<td>&lt;</td>
<td><code>Fun(Int : Int) : Bool</code></td>
<td>less than</td>
</tr>
<tr>
<td>&gt;</td>
<td><code>Fun(Int : Int) : Bool</code></td>
<td>greater than</td>
</tr>
<tr>
<td>&lt;=</td>
<td><code>Fun(Int : Int) : Bool</code></td>
<td>less equal than</td>
</tr>
<tr>
<td>&gt;=</td>
<td><code>Fun(Int : Int) : Bool</code></td>
<td>greater equal than</td>
</tr>
<tr>
<td>+</td>
<td><code>Fun(Int : Int) : Int</code></td>
<td>addition</td>
</tr>
<tr>
<td>-</td>
<td><code>Fun(Int : Int) : Int</code></td>
<td>subtraction</td>
</tr>
<tr>
<td>*</td>
<td><code>Fun(Int : Int) : Int</code></td>
<td>multiplication</td>
</tr>
<tr>
<td>/</td>
<td><code>Fun(Int : Int) : Int</code></td>
<td>division</td>
</tr>
<tr>
<td>%</td>
<td><code>Fun(Int : Int) : Int</code></td>
<td>modulo</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td><code>Fun(Real : Real) : Bool</code></td>
<td>less than</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td><code>Fun(Real : Real) : Bool</code></td>
<td>greater than</td>
</tr>
<tr>
<td>&lt;&lt;=</td>
<td><code>Fun(Real : Real) : Bool</code></td>
<td>less equal than</td>
</tr>
<tr>
<td>&gt;&gt;=</td>
<td><code>Fun(Real : Real) : Bool</code></td>
<td>greater equal than</td>
</tr>
<tr>
<td>++</td>
<td><code>Fun(Real : Real) : Real</code></td>
<td>addition</td>
</tr>
<tr>
<td>--</td>
<td><code>Fun(Real : Real) : Real</code></td>
<td>subtraction</td>
</tr>
<tr>
<td>**</td>
<td><code>Fun(Real : Real) : Real</code></td>
<td>multiplication</td>
</tr>
<tr>
<td>//</td>
<td><code>Fun(Real : Real) : Real</code></td>
<td>division</td>
</tr>
<tr>
<td>∨</td>
<td><code>Fun(Bool : Bool) : Bool</code></td>
<td>or</td>
</tr>
<tr>
<td>/</td>
<td><code>Fun(Bool : Bool) : Bool</code></td>
<td>and</td>
</tr>
<tr>
<td>not</td>
<td><code>Fun(Bool) : Bool</code></td>
<td>not</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td><code>Fun(String : String) : String</code></td>
<td>concatenation</td>
</tr>
<tr>
<td>extent</td>
<td><code>Fun(E &lt;: Ok : Array(E)) : Int</code></td>
<td>size</td>
</tr>
</tbody>
</table>

### B.4 Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>builtin</td>
<td>:<code>Fun(Dyn FctType &lt;: Ok</code> name : String ifFail : FctType) : FctType</td>
<td>builtin function of the code generator</td>
</tr>
<tr>
<td>bind</td>
<td>:<code>Fun(Dyn FctType &lt;: Ok</code> `lib, label, type : String) : FctType</td>
<td>external binding to a C language function</td>
</tr>
</tbody>
</table>
### Exceptions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>intOverflow</code></td>
<td>Exception end</td>
<td>integer overflow</td>
</tr>
<tr>
<td><code>intError</code></td>
<td>Exception end</td>
<td>integer error</td>
</tr>
<tr>
<td><code>realError</code></td>
<td>Exception end</td>
<td>real error</td>
</tr>
<tr>
<td><code>ccallError</code></td>
<td>Exception with <code>lib</code>, <code>entry</code>, <code>type</code> :<code>String</code> end</td>
<td>cannot find library or entry point, or bad arguments</td>
</tr>
<tr>
<td><code>assertError</code></td>
<td>Exception with <code>line</code>, <code>column</code> :<code>Int</code> where :<code>String</code> end</td>
<td>condition in assert statement violated</td>
</tr>
<tr>
<td><code>typecaseError</code></td>
<td>Exception with <code>line</code>, <code>column</code> :<code>Int</code> where :<code>String</code> end</td>
<td>non-handled type in typecase statement</td>
</tr>
<tr>
<td><code>caseError</code></td>
<td>Exception with <code>line</code>, <code>column</code> :<code>Int</code> where :<code>String</code> variant :<code>Int</code> end</td>
<td>non-handled case label in case statement</td>
</tr>
<tr>
<td><code>tupleProjectError</code></td>
<td>Exception with <code>line</code>, <code>column</code> :<code>Int</code> where :<code>String</code> variant :<code>Int</code> end</td>
<td>variant projection error in <code>!</code> expression</td>
</tr>
<tr>
<td><code>extendError</code></td>
<td>Exception with <code>line</code>, <code>column</code> :<code>Int</code> where :<code>String</code> <code>rcd</code> :Record end</td>
<td>duplicate label in record <code>extent</code> expression</td>
</tr>
<tr>
<td><code>indexOutOfBoundsError</code></td>
<td>Exception with <code>line</code>, <code>column</code> :<code>Int</code> where :<code>String</code> <code>arr</code> :Array(<code>Ok</code>) <code>index</code> :<code>Int</code> end</td>
<td>array index out of bounds</td>
</tr>
</tbody>
</table>
References


52


Index

A
abstract data type 23, 32
  polymorphic ~ 23
  semi ~ 23
ADT 23
alphanumeric identifier 4
and 9
andif 27
anonymous binding 12
anonymous identifier 10
anonymous variable 25
application programming 3
array 31
  ~ index bound 31
  ~ indexing 31
array 31
assignment 25

B
base type 5
begin 6, 27
binding 6, 31, 48
  anonymous ~ 7
  dynamic ~ 7
  function ~ 8
  recursive ~ 7
  repetition of ~ 17
  sequential ~ 6
  simultaneous ~ 6
  static ~ 6
  value ~ 6
block 6, 27
bound object 6
bounded parametric polymorphism 20
bracket 4, 9

C
C 38
C function 38
C library 38
call by reference 25, 40
call by value 25
callback 40
capitalization 42
case 11, 12
case analysis 12
  complete ~ 12
  incomplete ~ 12
case of 12
character set 3
checkpoint 37
coercion 5
comment 4, 44
  interface ~ 44
  module ~ 44
  multiline ~ 44
compilation unit 48
conditional expression 27
constant definition 24
contravariance rule 18
control structure 26
convention 4, 42
copy-in 40
copy-out 39, 40
covariance relationship 18
currying 10
cyclic dependency 36

D
data modeling 3
data-intensive application 15
declaration
  global ~ 6
  local ~ 6
do 28
dot notation 11, 12, 13, 36
downto 29
Dyn 15
dynamic binding 7
dynamic type 15

E
else 13, 27
elsif 27
encapsulation 32
  functional ~ 32
  imperative ~ 33
  method-based ~ 34
object-oriented ~ 32
equation type 12
equality sign 9
equality test 9
evaluation 5
exception 29, 52
  ~ argument 29
  ~ handling 29
  ~ package 29
  ~ propagation 30
  ~ raising 29
  ~ rerasing 30
  ~ signature 29
  ~ type 29
  standard ~ 29
user-defined ~ 29
Exception 29
exception 29
exit 28, 30
export 35
export 35
expression 6
extend 13
external function 38
external language 38
external library 38

F
factoring rule 8
field name 11
  anonymous ~ 11
  non-anonymous ~ 11
first class object 35
for 28
formal parameter 7
formatting convention 42
Fun 9
fun 7
function 7
  anonymous ~ 10
  external ~ 38
  ~ abstraction 7
  ~ application 8, 10, 19
  ~ as parameter 10
  ~ as result 10
  ~ as tuple field 11
  ~ binding 8
  ~ body 7
  ~ on type level 19
  ~ overriding 26
  ~ result 8
  ~ result type 9
  ~ signature 9
generic ~ 19
  higher-order ~ 10
  polymorphic ~ 19
  recursive ~ 9
  simple ~ 7
function type 9
  subtyping on ~ 18
functional programming 3
G
garbage collector 38
generic code 22
generic function 19
H
heterogeneous data structure 18
hide 37
higher-order function 10
I
identifier 4, 50
  global ~ 6
  local ~ 6
  use of ~ 6
  user-defined ~ 6
identity test 9
if 27
imperative programming 3, 24
import 35
  ~ed interface 35
  ~ed module 35
import 35
indentation 43
infix notation 8, 22
infix operator 8, 22, 51
infix symbol 4
interactive programming environment 3
interface 35, 48
  hidden ~ 37
  ~ comment 44
  ~ definition 36
interface 35
iteration 28

$L$
labeled Cartesian product 10
layout convention 42

$\text{Let}$ 10, 21
$\text{let}$ 6, 24
lexical rule 3

library 36, 48
  $C \sim 38$
  enclosing $\sim 37$
  external $\sim 38$
  hierarchic $\sim$ structuring 37

library 36
list 14
list fix notation 31

literal 5
loop 28
  enumerating $\sim 28$
  $\sim$ variable 29
  prechecking $\sim 28$

loop 28
loss of type information 20

$M$
method 11, 34
method overwriting 34
modularization 34
module 35, 48
  hidden $\sim 37$
  $\sim$ comment 44
  $\sim$ definition 36

module 35
multiline comment 44
multiple inheritance 18
mutable variable 24
mutually recursive functions 9

$N$
naming 6, 31
naming convention 42
nesting level 43

$\text{Nok}$ 30
non-anonymous binding 13
non-anonymous signature 13
non-parametrized type 15

$O$
object 34
object store 37
  $\sim$ stabilizing 37
object-oriented programming 11, 34
of 12
$\text{Ok}$ 7, 15, 25, 28
$\text{ok}$ 7, 25
opaque type 32
open 17

$\text{Oper}$ 21
operator precedence 4
optional value 21

$\text{orif}$ 27
overloading 5

$P$
P-Quest 2
parallel definition 9
parameter format 39
parameter passing 7
  $\sim$ mechanism 25
parametric polymorphism 19
parametrization 19
partial specification 15

persistence 37
persistent data 37
pointer type 7
polymorphic function 19, 22, 23
polymorphic null element 30
polymorphic type system 3
polymorphism 19
  bounded parametric $\sim 20$
  parametric $\sim 19$
  subtype $\sim 15$

postcondition 18
precondition 18
predefined
  $\sim$ constant 5, 50
  $\sim$ function 5, 51
  $\sim$ identifier 50
  $\sim$ type 5, 50
subtyping on $\sim$ types 15

program extension 17
programming
  multi-paradigm $\sim 31$
object-oriented $\sim 11, 34$
  $\sim$ style 31
punctuation 42

$Q$
qualifying identifier 35
Quest 2

$R$
raise 29
reachability 37
read-only access 26
readability 4
Rec 14
rec 7, 9
Record 13
record 13
record type 13
  subtyping on $\sim$ 18
record value 13
  extension of $\sim$ 13
recursive
  $\sim$ binding 9
  $\sim$ data structure 14
  $\sim$ data type 14, 34
  $\sim$ function 9
  $\sim$ type operator 22
  $\sim$ value 14
Repeat 16
repeating component 16
repetition of binding 17
reenable 30
reserved keyword 4, 47
rollback 37
runtime error 13

$S$
scoping 11, 13
  $\sim$ rule 6
semantic concept 3
semi-abstract data type 23
sequence 27
sequential execution 27
shared variable 25, 34
signature 6, 7, 15, 49
space 4, 42
spelling 42
stack 23, 32
standard library 5, 36
state 32
state-based 32
static binding 6
static type checking 15
static type information 6
structural compatibility 16
structural equivalence 15
subclass 16
subsignature 18
  $\sim$ relationship 24
subsumption principle 15
subsumption rule 26
subtype 15
  $\sim$ definition 16
  $\sim$ polymorphism 15, 20
  $\sim$ relationship 15
subtyping 15
  $\sim$ for mutable bindings 26
  $\sim$ hierarchies 18
  $\sim$ on function types 18
  $\sim$ on predefined types 15
  $\sim$ on record types 18
  $\sim$ on tuple types 16
  $\sim$ on variant types 17
supertype 14, 15
syntactic structure 3
syntactical rule 3
syntax error 9
system programming 3

$T$
then 27, 30
top level 36
trivial specification 15
try 29
tuple 10
  $\sim$ field 10
  $\sim$ field selection 11
  $\sim$ signature 24
  $\sim$ value 11
  $\sim$ with variant 11
Tuple 10, 11
tuple 11, 12
tuple type 10
  subtyping on $\sim$ 16
  $\sim$ with variants 11
type 49
defered $\sim$ checking 15
dynamic \sim 15
exception \sim 29
function \sim 9
non-parametrized \sim 15
opaque \sim 32
record \sim 13
tuple \sim 10
\sim\text{ argument} 19
\sim\text{ compatibility} 11
\sim\text{ constructor} 10
\sim\text{ expression} 6
\sim\text{ inference} 7, 8, 19
\sim\text{ parameter} 19
variant \sim 11
type operator 19, 21, 23
  \text{ first-order} \sim 22
  \text{ higher-order} \sim 22
  \text{ recursive} \sim 22
  \text{ simple} \sim 21
  \sim\text{ application} 21
type-independent behavior 19
typecase 15
typing 5, 7, 31

\textit{U}
uninitialized identifier 7
\textbf{upto} 29

\textit{V}
value 48
  \sim\text{ parameter} 25
\textbf{var} 24
variable 24
  \text{ mutable} \sim 24
  \sim\text{ declaration} 24
  \sim\text{ parameter} 25
variant 11
  \sim\text{ field projection} 12
variant type 11
  \text{ subtyping on} \sim 17
visibility 8, 24

\textit{W}
\textbf{when} 12, 15, 30
\textbf{while} 28
\textbf{with} 11, 13, 29, 30, 36

\textit{Others}