Definition of the Tycoon Language T1
A Preliminary Report

Florian Matthes  Joachim W. Schmidt

FBI-HH-B-160/92

Dezember 1992 (revidiert August 1995)

Fachbereich Informatik
Universität Hamburg
Vogt-Kölln-Str. 30
D-W-200 Hamburg 54
Federal Republic of Germany
Abstract
This document defines the language Tl that is used as the application and system programming language in the Tycoon database environment. The Tycoon project follows an add-on approach to generic database programming that emphasizes type-safe system scalability and extensibility.

Tl is a polymorphic second-order functional language with imperative features and inductively defined subtyping rules over types and type operator, extended by language constructs motivated by the needs of database programming.

Zusammenfassung
Dieses Dokument definiert die Sprache Tl, die sowohl als Anwendungs- als auch Systemprogrammiersprache in der Tycoon Datenbankumgebung verwendet wird. Das Tycoon Projekt folgt dem add-on-Ansatz zur generischen Datenbankprogrammierung, der besonders die typsichere Systemskalierbarkeit und -erweiterbarkeit unterstützt.

Tl ist eine polymorphe funktionale Programmiersprache zweiter Ordnung mit imperativen Eigenschaften und induktiv definierten Subtypisierungsregeln über Typen und Typoperatoren, erweitert um Sprachkonstrukte, die durch die Anforderungen der Datenbankprogrammierung begründet sind.
## Contents

1 The Rationale behind Tl ................................................. 1

2 Tl: A Language Overview ........................................... 3

3 The Tl Grammar .......................................................... 9
   3.1 Symbols .......................................................... 9
   3.2 Reserved Keywords .......................................... 10
   3.3 Compilation Units ........................................... 11
   3.4 Bindings ....................................................... 11
   3.5 Values ......................................................... 12
   3.6 Signatures .................................................... 13
   3.7 Types ........................................................ 13
   3.8 Identifier ..................................................... 14

4 The Tl Abstract Syntax .............................................. 15
   4.1 Syntactic Objects of Tl ..................................... 15
   4.2 Normalization of Tl Programs ............................. 15
   4.3 Signatures, Bindings and Types ........................... 18
   4.4 Values ......................................................... 18

5 The Static Semantics of Tl ......................................... 21
   5.1 Overview over the Type Notations ........................ 21
   5.2 Substitutions ................................................ 22
   5.3 Qualified Type Variables ................................... 22
   5.4 Contractive Types .......................................... 23
   5.5 Well-formed Signatures .................................... 23
   5.6 Well-formed Types .......................................... 24
   5.7 Signatures of Types and Values ........................... 25
   5.8 Subsignatures ............................................... 25
   5.9 Subtypes ...................................................... 25
   5.10 Signatures of Bindings .................................... 27
   5.11 Types of Values ............................................ 28
   5.12 Restrictions ................................................ 29
1 The Rationale behind T1

The functionality of data-intensive applications is provided by a heterogeneous mix of tools and servers that have been developed independently and that typically communicate only via narrow, ad-hoc interfaces. Examples of commercially available *generic* servers are relational database systems, object stores, transaction monitors, deduction engines, graphical user interface generators or communication subsystems.

The *Tycoon*\(^1\) project aims at a substantially improved server exploitation and database programmer productivity by providing an integrated linguistic and architectural framework for the *flexible integration of generic services* into an open database programming environment. The Tycoon project activities towards this goal include [Matthes 1992]

- the development and formal definition of a highly generic language kernel supporting naming, binding and typing of objects and services required for data-intensive applications in a way that is not biased towards a specific data model;

- the extension of this language kernel to a fully-fledged strongly-typed polymorphic programming language (T1, *Tycoon Language*) that also includes specific language concepts for integrating, extending, tailoring and using the generic database servers mentioned above;

- the definition of the data and program representation as well as the evaluation semantics of an untyped intermediate language (TML, *Tycoon Machine Language*) that not only supports an efficient host-specific target code generation but also portability, interoperability in heterogeneous environments and dynamic optimizations analogous to query optimization in database systems;

- the definition of a data-model-independent object store protocol (Tsp, *Tycoon Store Protocol*) that decouples TML evaluators from object stores and their components for access optimization, storage reclamation, persistence, concurrency, recovery or distribution.

The overall Tycoon system architecture is sketched in Fig. 1 on page 2. This report presents the design of T1, an integrated application and system programming language that

- provides uniform and expressive naming, binding and scoping rules for all language entities (values, functions, types, type operators),

- is not restricted to a particular modeling or programming style (e.g. purely functional, relational or strictly object-oriented),

- minimizes built-in language functionality in favor of flexible system add-ons [Matthes and Schmidt 1991a],

- is equipped with powerful type abstraction mechanisms like (bounded) type parameterization, (partial) type hiding and run-time type discrimination [Matthes and Schmidt 1991b],

---

\(^1\)Tycoon: Typed COmmunicating Objects in Open eNvironments
supports the operational requirements of today's data-intensive environments (longevity, evolution, distribution, merging of independently developed system components) [Schmidt and Matthes 1991].

A first prototypical set of the TL language processors (scanner, parser, unparsr, type checker, variable allocator, TL to TML translator, TML to C translator, TML interpreter, dynamic object code linker) is currently being implemented at Hamburg University using the P-Quest system, a persistent version of the Quest language developed by Luca Cardelli [Cardelli 1989; Cardelli 1990] on top of the Napier Persistent Object Store developed at the University of St. Andrews [Brown and Rosenberg 1991]. Much emphasis is put on the exchangeability and reusability of the individual compiler components by using (wherever possible) compiler generator technology [Schröder and Matthes 1992].
2 Tl: A Language Overview

The type-theoretic core of Tl is $F_\leq$, a second-order lambda calculus with subtyping [Cardelli et al. 1991]. This core is extended by a rather small set of standard programming concepts (base types, tuples, arrays, control structures and exceptions) as found in other functional or imperative programming languages. Some of the characteristic features of Tl are sketched below.

Syntax The regular LL(1) grammar for the concrete syntax of Tl is heavily influenced by languages like Modula-2, Modula-3 and Quest that all give preference to notations that are easy to read and understand over concise or cryptic notations that are easy to write. Tl provides adequate notations for the non-trivial bindings and signatures that occur in highly polymorphic library code. It emphasizes the symmetry between constructs at the value and type level and supports the easy discrimination between value and type expressions to simplify type inference in bindings.

Base Types Tl carefully avoids any non-standard treatment (like static binding, pervasive scoping, non-standard associativity, precedence or overloading) of the usually built-in functions on base types (integers, reals, strings, etc.). In order to achieve maximum system flexibility, these functions are provided by modules in the standard programming environment. The module implementations are written in C (resp. Modula-2) and are dynamically or statically linked to the generated TML or C code. Standard optimization techniques in the C backend (i.e. inlining) eliminate most of the possible overhead implied by this approach. Flexible handling of the base types is also supported by the treatment of integer, real, character and string literals within the relevant Tl language processor components (scanner, type checker, code generator).

Mutability Value variables in a Tl binding or signature can be marked with a var keyword. The phrase let var $i = 1$ introduces a mutable binding between the variable $i$ and the value 1. The variable $i$ can be re-bound using a polymorphic assignment function in the Tycoon programming environment. This function has the signature :=($A <\Rightarrow Ok$ var $lhs :A rhs :A$) and can redeclared locally (e.g. to attach recovery code to the destructive assignment). Mutable bindings cover the the concepts of local and global imperative variables, mutable components in aggregate values and also variable parameters as found in imperative programming languages. Variable parameters introduce l-value bindings (aliasing). Modifiability in Tl is therefore not captured by first-class type and value constructors (like ref in ML) but is a property of bindings and signatures. Imperative and functional realms of Tl cannot be strictly separated since it is possible to view a mutable binding through a signature that defines the binding as immutable. The latter language design decision is motivated by the desire to provide both, flexible subtyping and assignment in a statically typed programming language (for an in-depth discussion of alternative approaches see [Connor et al. 1991]).

Bindings and Signatures Tl recognizes the importance of naming, scoping and binding concepts in large-scale programming. Therefore, the main semantic building blocks of Tl are not values and types in isolation but bindings (ordered associations of types to type variables and of values to value variables) and signatures (ordered associations of types to type bounds and values to types) (see also [Burstall and Lampson 1984; Cardelli 1989]).
The scoping rules for type and variable names, the matching rules between bindings and their signatures and the refinement relation on signatures can all be specified without reference to the particular context in which bindings or signatures occur. Possible contexts for signatures include formal function parameters, module interfaces as well as function types, tuple, record and exception types. Possible contexts for bindings include actual function parameters, module implementations, tuple, record and exception values and top level phrases entered interactively.

TL allows to **Repeat** another named signature in the definition of a new signature and to **open** (a projection of) existing bindings for the definition of new bindings. These language mechanisms can substantially improve the understandability and consistency of large-scale systems by factoring-out common subsignatures and subbindings. In the context of tuples and records they also cover aspects of **inheritance** as found in object-oriented languages.

**Higher-Order Subtyping** Type judgements in TL are understood as *partial* specifications. For example, the signature \( x : A \) asserts that a value bound to the variable \( x \) satisfies *at least* the specification given by the type expression \( A \). An actual value binding \( \text{let } x = a \) may satisfy a more precise specification \( x : B \). The idea that a value satisfying a stronger specification \( B \) also satisfies a weaker specification \( A \) is captured by an inductively defined subtyping relationship on types (TL notation: \( B <: A \)).

For aggregate values, this view coincides with the notion of subtyping based on information content [Ohori et al., 1989]. For example, the type \( \text{Let } A = \text{Tuple name :String end} \) provides less information than the type \( \text{Let } B = \text{Tuple name :String age :Int end} \). The view of types as partial specifications generalizes to first-class function values and leads to the well-known contravariance rule for the function type constructor.

TL provides two type constants to denote the trivial specification (\( \text{Ok}, \) top type) and the unsatisfiable specification (\( \text{Nok}, \) bottom type). The latter appears naturally in the typing rules for exceptions and empty arrays (see §5.11).

In TL it is necessary to specify a *bound* for a type variable that appears in a signature (e.g., \( X <: A \)). Again, this bound is to be understood as a partial specification of the actual type that appears in the matching binding \( \text{Let } X = B \) and may involve the constants \( \text{Ok} \) and \( \text{Nok} \).

Finally, TL extends the subtyping relation to type operators, i.e. parameterized specifications (see §5.9)

\[
\begin{align*}
\text{Subtype Oper} \\
& \frac{S \vdash S'' <: S' \quad S, S'' \vdash A <: B}{S \vdash \text{Oper}(S')A <: \text{Oper}(S'')B}
\end{align*}
\]

For example, given

\[
\text{Let } F = \text{Oper}(X <: \text{Tuple name :String end}) \text{ Tuple } x : X \ y : X \text{ end} \\
\text{Let } G = \text{Oper}(X <: \text{Tuple name :String age :Int end}) \text{ Tuple } x : X \text{ end}
\]

\( F <: G \) is derivable, since any specification \( F(X) \) obtained by instantiating \( F \) with an actual type parameter \( X \) that is admissable for \( G \) yields a specification that is at least as strong as \( G(X) \).
The introduction of a top type Ok and the generalization of the subtyping relation to type operators also covers the notion of kinds (types of types), as introduced in [Cardelli 1989]. For example, all unbounded unary type operators (e.g. List, Array) are subtypes of \( \text{Oper}(X <\text{:Ok}) \text{Ok} \).

**Abstract Types** Tl. follows the tradition of modular languages like Modula-2, Modula-3, Oberon and Quest and uses the dot notation to denote the witness type in an abstract data type, for example:

\[
\begin{align*}
\text{Let} & \quad \text{Nat} = \text{Tuple } T<\text{:Ok} \text{ zero } :T \text{ succ}(\cdot T) :T \text{ eq}(\cdot T :T) :T \text{ end} \\
\text{let} & \quad \text{nat1} = \text{Tuple Let } T = \text{Int} \cdots \text{ end} \\
\text{let} & \quad \text{nat2} = \text{Tuple Let } T = \text{String} \cdots \text{ end} \\
\text{let} & \quad \text{null}(x :\text{nat1}.T) = \text{nat1.eq}(x \text{ nat1.zero}) \\
\text{let} & \quad \text{null2}(\text{rep } :\text{Nat } x :\text{rep}.T) = \text{rep.eq}(x \text{ rep.zero})
\end{align*}
\]

Bound specifications in the signature of an abstract data type allow to reveal partial type information about the witness type \( T \) (e.g. compatibility with other witness types). Special (path name equivalence) rules are introduced for a flexible handling of ADT values in complex data structures.

**Recursive Type Operators** The subtyping rules for recursive types and type operators in Tl. violate the concept of purely structural subtyping. For example, given

\[
\begin{align*}
\text{Let Rec} & \quad \text{List}(A <\text{:Ok}) = \text{Tuple } \text{hd } :A \text{ tl } :\text{List}(A) \text{ end} \\
\text{Let Rec} & \quad \text{List2}(A <\text{:Ok}) = \text{Tuple } \text{hd } :A \text{ tl } :\text{List}(A) \text{ end}
\end{align*}
\]

none of the following subtype relationships is derivable:

\[
\begin{align*}
\text{List(Ok)} <\text{:List2(Ok)} \\
\text{List2(Ok)} <\text{:List(Ok)} \\
\text{List } <\text{:List2} \\
\text{List2}<\text{:List}
\end{align*}
\]

On the other hand, a single step “unfolding” of \( \text{List}(A) \) in the definition of \( \text{List} \) yields a type operator that is structurally equivalent to \( \text{List2} \):

Since the equality between two recursive types in Tl. is therefore not based on the equality of their infinite expansions (two regular tree) but on a more restrictive compatibility notion (see also [Cardelli 1992b]), the problem of checking the equivalence between parameterized recursive type expressions becomes decidable (compare [Solomon 1978]) and a class of subtype relationships that play an important role in object-oriented programming can be handled within Tl. For example, given

\[
\begin{align*}
\text{Let Rec} & \quad \text{List}(E <\text{:Ok}) = \text{Tuple} \\
& \quad \text{cons}(\cdot E) :\text{List}(E) \\
& \quad \text{hd}(\cdot) :E
\end{align*}
\]
\[ tl() : \text{List}(E) \]
\[ \text{map}(E2 <: \text{Ok} \ f(:E) : E2) : \text{List}(E2) \]
\textbf{end}

\textbf{Let Rec} \ List2(E <: \text{Ok}) = \text{Tuple}
\[ \text{cons}(E) : \text{List2}(E) \]
\[ \text{hd}(E) : E \]
\[ tl(E) : \text{List2}(E) \]
\[ \text{map}(E2 <: \text{Ok} \ f(:E) : E2) : \text{List2}(E2) \]
\[ \text{powerset}(p(E) : \text{Bool}) : \text{List2}(	ext{List}(E)) \]
\textbf{end}

\text{List2}(	ext{Int}) <: \text{List}(	ext{Int}) \text{ and } \text{List2} <: \text{List} \text{ are derivable type judgements in Tl.}

\textbf{Tuples and Variants} \text{ The concepts of labeled cartesian product types and discriminated union types are captured by tuples and variants in Tl that exhibit an interesting subtype relationship. A pure discriminated union type is represented by a variant type with variants that do not share common fields while a pure cartesian product type can be viewed a variant type with a single variant and an anonymous variant label. A prime motivation behind this view is to simplify the handling of discriminated unions that share common attributes between their variants.}

Another motivation is to provide a type-safe solution to a common problem in persistent systems, namely the need for unforeseen extensions and variations of data structures without invalidating existing persistent bindings (in databases, files or possibly compiled programs). For example, an early system version may introduce the following binding

\textbf{Let} \text{ Address = Tuple city : String zip : Int end}
\textbf{let} \text{ a : Address = tuple let} \text{ city = "Hamburg" let} \text{ zip = 2000 end}

that is still usable in an extended system version with the following type and function definitions:

\textbf{Let} \text{ NewAddress = Variant city : String}
\textbf{case} \text{ national with} \text{ zip : Int}
\textbf{case} \text{ international with} \text{ country : String zip : String}
\textbf{end}
\textbf{let} \text{ print(adr : NewAddress) : Ok = ...}
\text{print(a)}

\textbf{Extensible Records} \text{ As argued in [Atkinson and Morrison 1988; Morrison \textit{et al.} 1987; Schmidt and Matthes 1991], long-lived data-intensive applications require naming schemes that are capable of handling \textit{dynamic} collections of bindings as they are found, for example, in directory structures of operating systems or in data dictionaries of databases. Extensible records address this need and provide a particular combination of static type safety and dynamicity that blends well with the subtyping rules of Tl. Record extension operations are also useful for modelling object-oriented programming concepts like objects with an immutable identity or method dictionaries.}
In contrast with other bindings and signatures in T1, record bindings are unordered. Neither duplicate nor anonymous variable names are permitted in record bindings and record signatures.

\[\text{let } db = \text{record let} \text{ parts } = \ldots \text{ let} \text{ suppliers } = \ldots \text{ end} \]

The selection of a field in a record value \( db . parts \) is checked statically against the signature of the record type and never fails at run-time. The extension of a record value with additional bindings \( \text{let new DB = extend } db \text{ with let assembly = } \ldots \text{ end} \) yields a record value with the same identity as \( db \) and additional static type information. However, the extension operation may fail at run-time by raising an exception due to name clashes (e.g., in cases where \( db \) is already extended by a field \( \text{assembly} \)). In contrast to Napier88 [Dearle et al. 1989], there is no mechanism to remove an existing binding from a record.

There are several proposals to also check the record extension operation statically, essentially by encoding negative information in record type expressions (e.g., values of record type \( X \) are defined not have an \( \text{assembly} \) field) [Cardelli 1992a; Wand 1987; Stansifer 1988; Cardelli and Mitchell 1989; Rémy 1991]. The usefulness of the these type rules in type system that make heavy use of the subsumption rule is not yet clear. Therefore, the design of T1 gives up some static type safety in favor of language simplicity.

Since record types are subject to more flexible type rule than plain tuple types, they are also of interest even if the dynamicity provided by the \( \text{extend} \) operator is not required: The record type \( \text{Record } S \text{ end} \) is a subtype of \( \text{Record } S' \text{ end} \) if the signatures \( S \) can be obtained by dropping subsignatures at arbitrary positions from \( S' \) (see [Subsig Rcd] in §5.8). The subtype graph for record types can therefore form a DAG (multiple inheritance) and not just a tree (single inheritance) like for tuple types.

**Modules and Libraries** The modularization facilities in T1 are \text{interface}, \text{module} and \text{library}. An interface is a named tuple type. A module is a named parameterless function. The evaluation of that function during module linkage returns a tuple value of its interface type. There may be more than one module implementing the same interface. The scope for interface and module names is defined by a library. The scope for interface, module and library names is flat. A library may contain other libraries as its component. All components of a nested library except those explicitly listed in a \text{hide} phrase are visible in the surrounding library. A library, an interface and a module may refer to other libraries, interfaces or evaluated modules only through \text{import} statements. Cyclic import dependencies are not allowed. If a module \( m1 \) is imported into a module \( m2 \), the evaluation of \( m1 \) precedes the evaluation of \( m2 \).

The modularization facilities in T1 do not introduce \text{new} naming, typing and binding concepts into the language, they simply \text{restrict} the orthogonality of the existing concepts (functions, tuple types, nesting of scopes, sequential evaluation). These restrictions are designed to simplify the tool-supported maintenance of large (possibly persistent and partially bound) software systems (component replacement, incremental interface extension, type specialization).
**Dynamic Types** In analogy to the mutability attribute for value variables, a dynamic (Dyn) attribute can be assigned to type variables in bindings and signatures. The only difference between dynamic and non-dynamic type variables is the possibility to inspect the type associated with a type variable at run-time. Compared with other proposals [Abadi *et al.* 1989; Abadi *et al.* 1992] this approach to dynamic types does not require to explicitly inject and project individual values into infinite union types and blends well with automatic inference mechanisms for type components in bindings. Furthermore, it allows to perform reflective type analysis even in cases where no value of the dynamic type is available.

With respect to decidability, TL is in the same position as $\Gamma_\subseteq$ [Pierce 1992]. Decidability problems arise through the subtyping rule [Subtype Fun] in combination with the subsignature rule [Subsig Ide]. More precisely, there exist syntactically well-formed, but ill-typed TL programs for which the TL type checker does not terminate. By disallowing the specialization of bound types assigned to type variables in polymorphic procedures (rule [Subsig Ide]) in the context of rule [Subtype Fun], termination of the subtype algorithm can be guaranteed [Pierce 1992].
3 The T1 Grammar

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.

$Id_1, \ldots, Id_n := A$; the non-terminal symbols $Id_i$ are defined as $A$ ($n \geq 1$)

$Id$ a non-terminal symbol

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 ("" | $A$ { $A$ } )

3.1 Symbols

The source text of a T1 program consists of a sequence of characters that is converted into a sequence of symbols of the categories int, real, longreal, char, string, identifier, infix, colonInfix and 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.

int ::= 
    ["-" ] digit { digit } 
; 
real ::= 
    int "." digit { digit } | 
    int [ "." digit { digit } ] "E" int 
; 
longreal ::= 
    int [ "." digit { digit } ] "D" int 
; 
char ::= 
    "" ( digit | alpha | special | escape | delimiter | reserved ) ""
; 
string ::= 
    """" ( digit | alpha | special | escape | delimiter | reserved ) """"
; 
infix ::= 
    special { special }
colonInfix ::= 
   ":" { special }
;
identifier ::= 
   alpha { digit | alpha }
;
delimiter ::= 
   \n | \t | \r | \f | \' | " | \" | \\ | \nnnn

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

| \n    | new line              |
| \t    | tab                   |
| \r    | carriage return       |
| \f    | form feed             |
| \'    | '                      |
| "     | "                      |
|  \   | \                      |
| \n    | \nnnn A single character with the code nnn      |
| \f...f | (three decimal digits that denote an integer in the interval [0,255]  |
| \f...f | The sequence of formatting characters f is ignored |

The last rule enables the definition of string literals that exceed the length of a single source line.

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

### 3.2 Reserved Keywords

The following identifiers and (colon-) infix symbols are reserved keywords and cannot be used as user-defined identifiers in TL programs.
and andif as begin case do downto dynamic_new dynamic_be else elseif end exception exit export extend fun hide infix if import in interface let library loop module of ok open orif raise reraise rec then try tuple typeRep_new upto var variant when while with
Dyn Exception Fun Let Nok Ok Oper Rec Repeat Tuple Variant
= <: : ?!

3.3 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 } ]

3.4 Bindings

Bindings::=
{ TypeBindings | ValueBindings | open ValueIde [ as Type ]
  "::" [ Dyn | Type | [ var ] Value }

TypeBindings::=
{ Let [ Rec ] TypeBinding { and TypeBinding } }

TypeBinding::=
[ Dyn ] TypeIde Parameters [ "::" Type ] ":=" Type

;
Value Bindings ::= { let [ rec ] ValueBinding { and ValueBinding } }

Value Binding ::= [ var ] Value Ide Parameters [ "=" Type ] "=" Value

3.5 Values

Value ::= fun "(" Signatures ")" [ "=" Type ] Value |
assert Value |
Value₁ := Value₀ { ( orif | andif | colonInfix ) Value₀ }
Value₂ ::= Value₂ { infix Value₂ }
Value₃ ::= Value₃ { infix Value₃ }
Value₄ ::= "{" Value "}" |
Value Ide |
ok |
int | char | string | real | longreal |
tuple Bindings end |
variant Case Ide of Type [ with Bindings ] end |
record Bindings end |
extend Value [ with Bindings ] end |
array Bindings end |
exception Value [ with Signatures ] end |
begin Bindings end |
if Value then Bindings { elsif Value then Bindings }
[ else Bindings ] end |
case [ of ] Value { when Case Ide list [ with Value Ide ] then Bindings }
[ else Bindings ] end |
loop Bindings end |
exit |
while Value do Bindings end |
for Value Ide "=" Value ( upto | downto ) Value do Bindings end |
try Bindings { when Value [ with Value Ide ] then Bindings }
3.6 Signatures

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

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

ValueSignatures ::= 
   [ var ] [ ValuedList Parameters ] ":::" Type

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

3.7 Types

Type ::= 
   Oper "(" Signatures ")" [ "<;:" Type ] Type |
   Fun "(" Signatures ")" :::" Type |
   Type1

Type1 ::= 
   Type2 { colonInfix Type2 }

Type2 ::= 
   Type3 { infix Type3 }

Type3 ::= 
   Type4 { "(" { Type } ")" }
Type$\_4$::=  
"{" Type ")"" | 
{ ValueIde ",." } TypeIde | 
Ok | Nok | 
Tuple Signatures end | 
Variant Signatures { case CaseIdeList [ with Signatures ] } end | 
Record Signatures end | 
Exception [ Signatures ] end

;  

3.8 Identifier

ValueIdeList, TypeIdeList, CaseIdeList::=  
Ide { "," Ide }

;  

Ide, ValueIde, TypeIde, FieldIde, CaseIde::=  
identifier | \textbf{infix} ( infix | colonInfix )

;
4 The T1 Abstract Syntax

The concrete syntax of T1 provides several abbreviating notations for values, types, bindings and signatures and makes heavy use of initial and final keywords. A compact abstract syntax provides a more adequate starting point for the definition of the static semantics of T1. It also constitutes the canonical internal representation used by the T1 language processors.

4.1 Syntactic Objects of T1

The definition of the abstract T1 syntax involves syntactic objects that are denoted by meta variables according to the following conventions:

\[ S \in \text{Sig} \quad \text{sequences of signatures} \]
\[ D \in \text{Bind} \quad \text{sequences of bindings} \]
\[ E \in \text{BindElem} \quad \text{single bindings} \]
\[ A, B \in \text{Type} \quad \text{types} \]
\[ C \in \text{Case} \quad \text{variants in tuple types} \]
\[ z \in \text{CaseIde} \quad \text{variant labels} \]
\[ X, Y \in \text{Ide} \quad \text{type identifiers} \]
\[ x, y \in \text{ide} \quad \text{value identifiers} \]
\[ a, b, c \in \text{Value} \quad \text{values} \]
\[ p, q \in \text{Qualifier} \quad \text{selectors of signature components} \]

The infinite sets Ide, ide and CaseIde are built from the union of the symbols of the categories identifier, infix and colonInfix as defined in §3.1 extended by the symbol ?, called the anonymous identifier. These sets and the set of literals (symbols of the categories int, real, longreal, char, string) constitute the basis for the inductive definition of the sets of the other syntactic objects as summarized in table 1 and table 2.

4.2 Normalization of T1 Programs

The one-to-one translation from a parse tree of a phrase matching the concrete T1 syntax into an equivalent term in the abstract T1 syntax is rather straightforward. It is (conceptually) preceded by a normalization process that

- introduces the anonymous identifier ? in bindings

\[ :A \ a \Rightarrow \text{Let } ? = A \ \text{let } ? = a \]

and in signatures

\[ <:A :B \Rightarrow ? <:A ? :B \]


- expands abbreviating notations in bindings
let \( x(S_1) \ldots (S_n) : A = a \)  
\[
\Rightarrow \text{ let } x = \text{fun}(S_1) \ldots \text{fun}(S_n) : A = a
\]

let \( x(S_1) \ldots (S_n) = a \)  
\[
\Rightarrow \text{ let } x = \text{fun}(S_1) \ldots \text{fun}(S_n) = a
\]

Let \( X(S_1) \ldots (S_n) <: A = B \)  
\[
\Rightarrow \text{ Let } X = \text{Oper}(S_1) \ldots \text{Oper}(S_n) <: A = B
\]

Let \( X(S_1) \ldots (S_n) = A \)  
\[
\Rightarrow \text{ Let } X = \text{Oper}(S_1) \ldots \text{Oper}(S_n) = A
\]

and in signatures

\[
\begin{align*}
& x_1, \ldots, x_n : A \Rightarrow x_1 : A \ldots x_n : A \\
& \text{var } x_1, \ldots, x_n : A \Rightarrow \text{var } x_1 : A \ldots \text{var } x_n : A \\
& \text{Dyn } x_1, \ldots, x_n <: A \Rightarrow \text{Dyn } x_1 <: A \ldots \text{Dyn } x_n <: A \\
& x(S_1) \ldots (S_n) : A \Rightarrow x : \text{Fun}(S_1) \ldots : \text{Fun}(S_n) : A \\
& X(S_1) \ldots (S_n) <: A \Rightarrow X : \text{Oper}(S_1) \ldots : \text{Oper}(S_n) : A
\end{align*}
\]

– transforms infix and listfix function applications into a standard prefix notation

\[
\begin{align*}
& x_1 \text{ infix } x_2 \Rightarrow \{\text{infix}(x_1 \ x_2)\} \\
& x_1 \text{ colonInfix } x_2 \Rightarrow \{\text{colonInfix}(x_1 \ x_2)\} \\
& \text{array } x_1 \ldots x_n \text{ end} \Rightarrow \text{array}(x_1 \ldots x_n \text{ end})
\end{align*}
\]

– transforms infix type operator applications into prefix notation

\[
\begin{align*}
& x_1 \text{ infix } x_2 \Rightarrow \{\text{infix}(x_1 \ x_2)\} \\
& x_1 \text{ colonInfix } x_2 \Rightarrow \{\text{colonInfix}(x_1 \ x_2)\}
\end{align*}
\]

– replaces missing else branches in if and try expressions (but not in case expressions) by an empty expression sequence (that evaluates to \(\text{ok}\), see \[\text{Value seq empty} in \S 5.11\), for example,

\[
\text{if } a \text{ then } b \text{ end} \Rightarrow \text{if } a \text{ then } b \text{ else end}
\]

– normalizes elsif branches into nested if expressions

\[
\begin{align*}
& \text{if } a_1 \text{ then } b_1 \\
& \text{elsif } a_2 \text{ then } b_2 \\
& \ldots \\
& \text{end}
\end{align*}
\]

\[
\begin{align*}
& \Rightarrow \text{if } a_1 \text{ then } b_1 \\
& \text{else} \\
& \text{if } a_2 \text{ then } b_2 \\
& \ldots \\
& \text{end}
\end{align*}
\]

– replaces assertions by conditionals

\[
\begin{align*}
& \text{if } a \text{ then ok} \\
& \text{else} \\
& \text{raise exception} \\
& \text{“Assertion in file \ldots at line \ldots failed”}
\end{align*}
\]

\[
\text{end}
\]

\end{align*}
\]

\end{align*}
\]
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S ::= \varnothing$</td>
<td>empty signature</td>
</tr>
<tr>
<td>$S, x : A$</td>
<td>value signature</td>
</tr>
<tr>
<td>$S, X &lt;: A$</td>
<td>type signature</td>
</tr>
<tr>
<td>$S, X = A$</td>
<td>type definition</td>
</tr>
<tr>
<td>$S, \text{Repeat}(A)$</td>
<td>signature inheritance</td>
</tr>
<tr>
<td>$D ::= \varnothing$</td>
<td>empty binding</td>
</tr>
<tr>
<td>$D, E$</td>
<td>sequential binding</td>
</tr>
<tr>
<td>$D, E_1 \ldots E_n$</td>
<td>parallel binding</td>
</tr>
<tr>
<td>$D, \text{open}(x)$</td>
<td>binding inheritance</td>
</tr>
<tr>
<td>$D, \text{open}(x, A)$</td>
<td>binding projection</td>
</tr>
<tr>
<td>$E ::= x = a$</td>
<td>value binding</td>
</tr>
<tr>
<td>$x : A = a$</td>
<td>constrained value binding</td>
</tr>
<tr>
<td>$X = A$</td>
<td>type binding</td>
</tr>
<tr>
<td>$X &lt;: A = B$</td>
<td>constrained type binding</td>
</tr>
<tr>
<td>$A, B ::= \text{Ok}$</td>
<td>supertype of all types (top)</td>
</tr>
<tr>
<td>$\text{Nok}$</td>
<td>subtype of all types (bottom)</td>
</tr>
<tr>
<td>$X$</td>
<td>type identifier</td>
</tr>
<tr>
<td>$p, X$</td>
<td>abstract type identifier</td>
</tr>
<tr>
<td>$\text{Bool}</td>
<td>\text{Int}</td>
</tr>
<tr>
<td>$\text{Arr}(A)$</td>
<td>array type</td>
</tr>
<tr>
<td>$\text{Fun}(S : C)$</td>
<td>(polymorphic) function type</td>
</tr>
<tr>
<td>$\text{Tup}(S; C)$</td>
<td>tuple type (with variants)</td>
</tr>
<tr>
<td>$\text{Rec}(S)$</td>
<td>record type</td>
</tr>
<tr>
<td>$\text{Exp}(S)$</td>
<td>exception type</td>
</tr>
<tr>
<td>$\text{Rec}(X, X_1 &lt;: A_1 = B_1 \ldots X_n &lt;: A_n = B_n)$</td>
<td>recursive type ($1 \leq n$)</td>
</tr>
<tr>
<td>$\text{Var}(A)$</td>
<td>type of mutable values</td>
</tr>
<tr>
<td>$\text{Dyn}(A)$</td>
<td>dynamic type</td>
</tr>
<tr>
<td>$\text{Oper}(X_1 &lt;: A_1, \ldots, X_n &lt;: A_n) B$</td>
<td>type operator</td>
</tr>
<tr>
<td>$A(D)$</td>
<td>type operator application</td>
</tr>
<tr>
<td>$C ::= \text{Case}(z_1, S_1) \ldots \text{Case}(z_n, S_n)$</td>
<td>variants in tuple types ($0 \leq n$)</td>
</tr>
<tr>
<td>$p ::= x$</td>
<td>value signature selector</td>
</tr>
<tr>
<td>$p, x$</td>
<td>field signature selector</td>
</tr>
</tbody>
</table>

Table 1: Abstract syntax for signatures, bindings and types
4.3 Signatures, Bindings and Types

Table 1 summarizes the inductive definition of signatures, bindings and types in Tl.

The distinction between type signatures and type definitions in signatures reflects the fact that most type judgements provide a partial information ($X <: A$) and that only in few cases (e.g. Let type bindings in the local scope) the exact type associated with a type variable $X$ is available ($X = A$).

The base type constants (Int, Bool ...) are required to denote the types of Tl literals and the argument types expected by built-in language constructs (e.g. booleans for the conditional). They do not appear in the concrete Tl syntax since they are not hard-wired as keywords but bound to identifiers in the programming environment.

The rather heavy notation for recursive types is required to directly represent mutually recursive type bindings, like

$$\textbf{Let Rec } X <: \text{Ok} = F(Y) \text{ and } Y <: \text{Ok} = G(X);$$

This binding is represented in the abstract syntax as a parallel binding of two type variables ($X$ and $Y$) to two independent recursive types that both describe the full recursive equation system:

$$X = \text{Rec}(X_1, X_1 <: \text{Ok} = F(X_2), X_2 <: \text{Ok} = G(X_1))$$

and

$$Y = \text{Rec}(X_2, X_1 <: \text{Ok} = F(X_2), X_2 <: \text{Ok} = G(X_1))$$

This representation proved to be advantageous for the definition of the scoping and typing rules and the implementation of the relevant checking algorithms. The bound specifications in recursive type bindings ($<: \text{Ok}$ in the example above) are required to simplify the checking of type operator applications within the recursive binding, for example

$$\textbf{Let Rec } A1 = A2(A2) \text{ and } A2 = A3(A3) \text{ and ...}$$
$$\text{and } A_n = \text{Oper}(X(Y <: \text{Ok}) <: \text{Ok}) X(\text{Int})$$

has to be written as

$$\textbf{Let } \text{Unary} = \text{Oper}(X(Y <: \text{Ok}) <: \text{Ok}) \text{ Ok}$$
$$\textbf{Let } A1 <: \text{Unary} = A2(A2) \text{ and } A2 <: \text{Unary} = A3(A3) \text{ and ...}$$
$$\text{and } A_n <: \text{Unary} = \text{Oper}(X(Y <: \text{Ok}) <: \text{Ok}) X(\text{Int})$$

The var attribute of value bindings and signatures and the Dyn attribute of type bindings and signatures are both mapped to explicit type constructors. This implies that types like Var(Var(A)) do not appear in Tl programs.

Finally, it should be noted that entities of the value level enter the type level via qualifiers for abstract data types. The syntax for qualifiers is restricted and just allows simple path expressions involving value variables.

4.4 Values

Table 2 summarizes the inductive definition of values in Tl. The syntax is quite standard and therefore does not require further explanation.
\[a, b ::= \text{ok} \quad \text{the canonical value of type Ok}
| \quad \text{int} \quad \text{integer literals (s. \S 3.1)}
| \quad \text{real} \quad \text{floating point literals (s. \S 3.1)}
| \quad \text{longreal} \quad \text{double precision floating point literals (s. \S 3.1)}
| \quad \text{char} \quad \text{character literals (s. \S 3.1)}
| \quad \text{string} \quad \text{string literals (s. \S 3.1)}
| \quad x \quad \text{value identifier}
| \quad \text{fun}(S)a \quad \text{polymorphic function constructor}
| \quad a(D) \quad \text{function application}
| \quad \text{arr}(D) \quad \text{array constructor}
| \quad a[b] \quad \text{array element selection}
| \quad \text{tup}(D, z, D') \quad \text{tuple constructor (with variants)}
| \quad \text{rcd}(D) \quad \text{record constructor}
| \quad a, x \quad \text{tuple, record and execution field selection}
| \quad a!z \quad \text{variant projection}
| \quad a?z \quad \text{variant test}
| \quad \text{extend}(a, D) \quad \text{record extension}
| \quad \text{exc}(a, D) \quad \text{exception value generation}
| \quad \text{seq}(D) \quad \text{sequential evaluation}
| \quad \text{if}(a, b_1, b_2) \quad \text{conditional}
| \quad \text{case}(a,
\begin{align*}
& (z_1, \ldots, z_{i_k}, y_1, b_1) \ldots \\
& (z_n, \ldots, z_{i_k}, y_n, b_n), b)
\end{align*}
| \quad \text{case}(a,
\begin{align*}
& (z_1, \ldots, z_{i_k}, y_1, b_1) \ldots \\
& (z_n, \ldots, z_{i_k}, y_n, b_n))
\end{align*}
| \quad \text{loop}(a_1) \quad \text{loop}
| \quad \text{exit} \quad \text{loop exit}
| \quad \text{while}(a, b) \quad \text{pre-check loop}
| \quad \text{for}(x, a_1, a_2, a_3) \quad \text{integer iteration}
| \quad \text{try}(a,
\begin{align*}
& (a_1, y_1, b_1) \ldots (a_n, y_n, b_n), b)
\end{align*}
| \quad \text{raise}(a, D) \quad \text{exception generation}
| \quad \text{reraise} \quad \text{exception propagation}
| \quad \text{and}(a, b) \quad \text{conditional conjunction}
| \quad \text{or}(a, b) \quad \text{conditional disjunction}
| \quad ;
\]

Table 2: Abstract syntax for values
The syntax does not contain an assignment operator since assignments are provided by a polymorphic function in the programming environment with the following signature:

\[ (A <: Ok \ var \ lhs :A \ rhs :A) \]

There are two multi-branch case expressions. The type rules for one version requires an exhaustive enumeration of all possible variant values, while the other is equipped with a non-empty \texttt{else} branch.

\footnote{Different implementations of the assignment function may attach recovery or concurrency control functionality to the plain destructive update operator.}
5 The Static Semantics of T1

A medium-term goal in the Tycoon project is to provide a formal definition of all static and dynamic aspects of the language. For the purpose of this preliminary language definition, only the static semantics of T1 have been captured formally (omitting modules, libraries, object locality, the enforcement of the proper nesting of exit statements in loops, and the inference rules for type arguments in function applications).

The dynamic semantics of T1 are implicitly defined by a mapping of T1 terms to untyped terms of the underlying untyped Tycoon machine language TML. The evaluation semantics of TML in turn are defined w.r.t. the abstract operational semantics of the Tycoon store protocol Tsp.

5.1 Overview over the Type Notations

The following notations are used for the definition of the static semantics of T1:

<table>
<thead>
<tr>
<th>notation</th>
<th>description</th>
<th>page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( A \mid X )</td>
<td>The type ( A ) is contractive in variable ( X )</td>
</tr>
<tr>
<td>( S \text{ sig} )</td>
<td>( S ) are well-formed</td>
<td>§ 5.5</td>
</tr>
<tr>
<td>( S \text{ recsig} )</td>
<td>( S ) are well-formed record signatures</td>
<td>§ 5.5</td>
</tr>
<tr>
<td>( A \text{ type} )</td>
<td>( A ) is well-formed</td>
<td>§ 5.6</td>
</tr>
<tr>
<td>( A \text{ sig} S )</td>
<td>( A ) has signatures ( S )</td>
<td>§ 5.7</td>
</tr>
<tr>
<td>( p \text{ sig} S )</td>
<td>( p ) identifies a value with signatures ( S )</td>
<td>§ 5.7</td>
</tr>
<tr>
<td>( S &lt;:: S' )</td>
<td>( S ) are subsignatures of ( S' )</td>
<td>§ 5.8</td>
</tr>
<tr>
<td>( S \text{ tup} S' )</td>
<td>( S ) are tuple subsignatures of ( S' )</td>
<td>§ 5.8</td>
</tr>
<tr>
<td>( S \text{ rec} S' )</td>
<td>( S ) are record subsignatures of ( S' )</td>
<td>§ 5.8</td>
</tr>
<tr>
<td>( A &lt;: B )</td>
<td>( A ) is a subtype of ( B )</td>
<td>§ 5.9</td>
</tr>
<tr>
<td>( D :: S )</td>
<td>( D ) have signatures ( S )</td>
<td>§ 5.10</td>
</tr>
<tr>
<td>( a : A )</td>
<td>( a ) has type ( A )</td>
<td>§ 5.11</td>
</tr>
</tbody>
</table>

The last column indicates the section that contains the definition of each judgement. Note that virtually all definitions are mutually recursive and closely follow the structure of the inductive definition of the underlying syntactic objects given in the previous section.

The order-preserving concatenation of signatures and bindings is denoted as follows:

\[ S, S' \quad \text{the signatures in } S \text{ followed by signatures in } S' \]
\[ D, D' \quad \text{the bindings in } D \text{ followed by bindings in } D' \]

Despite the fact that signatures are ordered sequences, the following notation treats them like finite mappings from value names to types resp. type names to type bounds:

\[ x \in \text{Dom}(S) \quad \text{the value variable } x \text{ is defined in } S \]
\[ X \in \text{Dom}(S) \quad \text{the type variable } X \text{ is defined in } S \]

The type rules of T1 are presented as a system of axioms and deduction rules where the judgements in the premises and consequences in general refer to a static environment. The environment itself is represented as an ordered list \( S \) of signatures:

\[ S \vdash J \quad \text{in the static environment } S \text{ the judgement } J \text{ is true} \]
5.2 Substitutions

The type rules for the instantiation of polymorphic function [Value apply] in § 5.11 and of type operators [Type apply] in § 5.6 use the following notation to denote the variable capture avoiding substitution of free type variables in a type expression by an actual type parameter:

\[ A\{X \leftarrow B\} \quad \text{A where each free occurrence of } X \text{ is substituted by } B \]

This substitution operation could be made more explicit by replacing the applied occurrence of a variable name by the (relative) index of its defining occurrence within the static environment and thereby reducing the substitution operation to a mere index manipulation [de Bruijn 1972; Abadi et al. 1990].

A sequence of bindings \( D \) passed as actual parameters to a polymorphic function or a type operator defines an iterated substitution of the type variables \( X \) defined in \( D \) by their associated type expressions:

\[ A\{\leftrightarrow D\} \quad A \text{ where all } X, x \in \text{Dom}(D) \text{ are substituted as follows} \]

\[
\begin{align*}
\text{[Subst empty]} & \quad S \vdash A\{\leftrightarrow \emptyset\} = A \\
\text{[Subst Ide]} & \quad S \vdash D :: S', S, S' \vdash D' :: X <:: B \\
& \quad S \vdash A\{\leftrightarrow D, D'\} = A\{\leftrightarrow D\}\{X \leftarrow B\} \\
\text{[Subst Let]} & \quad S \vdash D :: S', S, S' \vdash D' :: X = B \\
& \quad S \vdash A\{\leftrightarrow D, D'\} = A\{\leftrightarrow D\}\{X \leftarrow B\} \\
\text{[Subst ide]} & \quad S \vdash D :: S', S, S' \vdash D' :: x : B \\
& \quad S \vdash A\{\leftrightarrow D, D'\} = A\{\leftrightarrow D\} 
\end{align*}
\]

5.3 Qualified Type Variables

The access to the type information associated with individual type or value components in a tuple or a record [Subtype Dot] [Value tuple dot] [Value record dot] requires the qualification of type variables by a path that identifies the enclosing tuple or record value to avoid name conflicts with type variables in the global scope and to achieve the type abstraction implied by packaged type components.

\text{qualify}(T; p; S) \quad \text{substitute in } T \text{ each occurrence of } X \in \text{Dom}(S) \text{ by } p.X

where \( T \in \text{Sig} \cup \text{Type} \). This qualification is defined inductively as follows:

\[
\begin{align*}
\text{qualify}(T; p; \emptyset) &= T \\
\text{qualify}(T; p; S, x : B) &= \text{qualify}(T\{x \leftarrow p.x\}; p; S) \\
\text{qualify}(T; p; S, X <: B) &= \text{qualify}(T\{X \leftarrow p.X\}; p; S) \\
\text{qualify}(T; p; S, X = B) &= \text{qualify}(T\{X \leftarrow p.X\}; p; S)
\end{align*}
\]

\[ 3 \text{This is also the mechanism used in the TL type checker.} \]
5.4 Contractive Types

A final auxiliary notation is required in rule \([\text{Type Rec}]\) in § 5.6 to restrict the structure of recursive types to contractive type expressions.

\[ S \vdash A \mid X \quad \text{In environment } S \text{ type } A \text{ is contractive in variable } X \]

Systems of recursive type equations \( \text{Rec}(X_i, X_1 <: A_1 = B_1 \ldots X_n <: A_n = B_n) \) that are contractive in their recursion variable \( X_i \) define well-formed regular trees, the basis for semantic models of type structures [MacQueen et al. 1986] as well as termination proofs for type checking algorithms [Ohori 1989].

The following judgements are independent of the identifiers occurring in the the static environment:

\[ Y \mid X \Rightarrow Y \neq X \]

\[ \text{Ok} \mid X \quad \text{Nok} \mid X \quad \text{Bool} \mid X \quad \text{Int} \mid X \quad \text{String} \mid X \]

\[ X, Y \mid X \quad \text{Fun}(S') \mid B \mid X \quad \text{Tup}(S', C) \mid X \quad \text{Rcd}(S') \mid X \quad \text{Exc}(S') \mid X \quad \text{Arr}(A) \mid X \]

\[ \text{Var}(A) \mid X \Rightarrow A \mid X \]

\[ \text{Dyn}(A) \mid X \Rightarrow A \mid X \]

The last three judgements refer to the static environment \( S \):

\[ [\text{Contractive Oper}] \]

\[ S, S' \vdash B \mid X \]

\[ S \vdash \text{Oper}(S')B \mid X \]

\[ [\text{Contractive Rec}] \]

\[ S \vdash \text{Rec}(X_i, X_1 <: A_1 = B_1 \ldots X_n <: A_n = B_n) \]

\[ S \vdash \text{Rec}(X, X_1 <: A_1 = B_1 \ldots X_n <: A_n = B_n) \mid Y \]

\[ [\text{Contractive Apply}] \]

\[ S \vdash A <: \text{Oper}(S')B \quad S \vdash D :: S' \quad S \vdash B \{\leftarrow D\} \mid X \]

\[ S \vdash A(D) \mid X \]

The above definition is well-founded although rule \([\text{Contractive Rec}]\) itself requires the definition of well-founded types, since the syntax of T1 does not allow statically nested recursive type declarations.

5.5 Well-formed Signatures

\[ [\text{Sig empty}] \]

\[ S \vdash A \quad \text{type} \]

\[ \vdash \emptyset \quad \text{sig} \]

\[ [\text{Sig ide}] \]

\[ S \vdash A \quad \text{type} \]

\[ \vdash S, x : A \quad \text{sig} \]

\[ [\text{Sig Ide}] \]

\[ S \vdash A \quad \text{type} \]

\[ \vdash S, X <: A \quad \text{sig} \]

\[ [\text{Sig Ide Let}] \]

\[ S \vdash A \quad \text{type} \]

\[ \vdash S, X = A \quad \text{sig} \]

\[ [\text{Sig Repeat}] \]

\[ S \vdash A \quad \text{sig} \]

\[ S' \quad \vdash S' \quad \text{sig} \]

\[ \vdash S, \text{Repeat}(A) \quad \text{sig} \]
5.6 Well-formed Types

[Type Builtin]
\[\vdash S \text{ sig } \quad A \in \{\text{Ok, Nok, Bool, Int, String}\} \]
\[S \vdash A \text{ type} \]

[Type Ide Let]
\[\vdash S, X <: A, S' \text{ sig } \quad X \not\in \text{Dom}(S') \]
\[S, X = A, S' \vdash X \text{ type} \]

[Type Dot]
\[S \vdash p \text{ :: } S' \quad S, S' \vdash X \text{ type } \quad X \in \text{Dom}(S') \]
\[S \vdash p.X \text{ type} \]

[Type Tup]
\[1 \leq n \quad z_i \neq z_j \quad i \neq j \quad \vdash S, S' \text{ sig } \quad \vdash S, S', S_i \text{ sig } \quad i = 1 \ldots n \]
\[S \vdash \text{Tup}(S'; \text{Case}(z_1, S_1) \ldots \text{Case}(z_n, S_n)) \text{ type} \]

[Type Rcd]
\[S \vdash S' \text{ rcdsig} \]
\[S \vdash \text{Rcd}(S') \text{ type} \]

[Type Exc]
\[S \vdash S' \text{ sig} \quad S \vdash \text{Exc}(S') \text{ type} \]

[Type Rec]
\[S, X_1 <: A_1, \ldots, X_n <: A_n \vdash B_i <: A_i \quad i = 1 \ldots n \quad 1 \leq j \leq n \quad S, X_1 = B_1, \ldots, X_n = B_n \vdash B_i \mid X_i \quad i = 1 \ldots n \]
\[S \vdash \text{Rec}(X_i, X_1 <: A_1 = B_1 \ldots X_n <: A_n = B_n) \text{ type} \]

[Type Dyn]
\[S \vdash A \text{ type} \]
\[S \vdash \text{Dyn}(A) \text{ type} \]

[Type Arr]
\[S \vdash A <: \text{Ok} \]
\[S \vdash \text{Arr}(A) \text{ type} \]

[Type Oper]
\[S \vdash S' \text{ sig} \quad S, S' \vdash B \text{ type} \]
\[S \vdash \text{Oper}(S') B \text{ type} \]

[Type Apply]
\[S \vdash A <: \text{Oper}(S') B \quad S \vdash D :: S' \]
\[S \vdash A(D) \text{ type} \]
5.7 Signatures of Types and Values

[Select ide]
\[
S \vdash x : A \quad S \vdash A \overset{\text{sig}}{\Rightarrow} S'
\]
\[
S \vdash x : S'' \overset{\text{ide}}{\Rightarrow} S''
\]

[Select dot]
\[
S \vdash p : S' \overset{\text{dot}}{\Rightarrow} S'
\]
\[
S \vdash p.x : S'' \overset{\text{dot}}{\Rightarrow} S''
\]

[Select Tup]
\[
S \vdash A : \text{Tup}(S'; C)
\]
\[
S \vdash A \overset{\text{Tup}}{\Rightarrow} S'
\]

[Select Rec]
\[
S \vdash A : \text{Rec}(S')
\]
\[
S \vdash A \overset{\text{Rec}}{\Rightarrow} S'
\]

[Select Exc]
\[
S \vdash A : \text{Exc}(S')
\]
\[
S \vdash A \overset{\text{Exc}}{\Rightarrow} S'
\]

[Select Fun]
\[
S \vdash A : \text{Fun}(S') : B
\]
\[
S \vdash A \overset{\text{Fun}}{\Rightarrow} S'
\]

5.8 Subsignatures

[Subsig reflexive]
\[
\vdash S, S' \overset{\text{sig}}{\Rightarrow} S'
\]
\[
S \vdash S', x : A \overset{\text{sig}}{\Rightarrow} S'', x : B
\]

[Subsig ide]
\[
S \vdash S' \overset{\text{ide}}{\Rightarrow} S''
\]
\[
S \vdash S', x : A \overset{\text{ide}}{\Rightarrow} S'', x : B
\]

[Subsig Ide]
\[
S \vdash S', x <\!: A \overset{\text{ide}}{\Rightarrow} S'', x <\!: B
\]
\[
S \vdash S', x <\!: A \overset{\text{ide}}{\Rightarrow} S'', x <\!: B
\]

[Subsig Ide Let]
\[
S \vdash S' \overset{\text{ide let}}{\Rightarrow} S'' \quad S \vdash S', x <\!: A \overset{\text{ide let}}{\Rightarrow} S'', x <\!: B
\]
\[
S \vdash S', x = A \overset{\text{ide let}}{\Rightarrow} S'', x <\!: B
\]

[Subsig Repeat]
\[
S \vdash S' \overset{\text{repeat}}{\Rightarrow} S'' \quad S \vdash S', \text{Repeat}(A) \overset{\text{repeat}}{\Rightarrow} S'', S'''
\]
\[
S \vdash S', \text{Repeat}(A) \overset{\text{repeat}}{\Rightarrow} S'', S'''
\]

[Subsig Tup]
\[
S \vdash S'' \overset{\text{typ}}{\Rightarrow} S_1\overset{\text{Tup}}{\Rightarrow} S_2\overset{\text{Tup}}{\Rightarrow} S_3\overset{\text{Tup}}{\Rightarrow} S_4
\]
\[
S \vdash S', \text{Repeat}(A) \overset{\text{repeat}}{\Rightarrow} S'', S'''
\]

[Subsig Rcd]
\[
S \vdash S_1, S_2, S_3 \overset{\text{rcd sig}}{\Rightarrow} S \vdash S_1, S_2 \overset{\text{rcd}}{\Rightarrow} S_3 \overset{\text{Tup}}{\Rightarrow} S_4
\]
\[
S \vdash S_1, S_2, S_3 \overset{\text{rcd}}{\Rightarrow} S_1, S_2 \overset{\text{Tup}}{\Rightarrow} S_3
\]

5.9 Subtypes

[Subtype reflexive]
\[
S \vdash A \overset{\text{type}}{\Rightarrow} A
\]
\[
S \vdash A <\!: A
\]

[Subtype transitive]
\[
S \vdash A <\!: A' \quad S \vdash A' <\!: A''
\]
\[
S \vdash A <\!: A''
\]

[Subtype Nok]
\[
S \vdash A <\!: \text{Ok}
\]
\[
S \vdash A <\!: \text{Ok}
\]

[Subtype Ok Built in]
\[
\vdash S \overset{\text{sig}}{\Rightarrow} A \in \{\text{Bool}, \text{Int}, \text{String}\}
\]
\[
S \vdash A <\!: \text{Ok}
\]

[Subtype Ok Fun]
\[
S \vdash \text{Fun}(S') : B \overset{\text{type}}{\Rightarrow} \text{Fun}(S') : B <\!: \text{Ok}
\]

\[
S \vdash \text{Fun}(S') : B <\!: \text{Ok}
\]
### Subtype Rules

<table>
<thead>
<tr>
<th>Subtype Rule</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Subtype Ok Tup</strong></td>
<td>$S \vdash \text{Tup}(S'; C)$ type</td>
<td>$S \vdash \text{Tup}(S'; C) \prec:: \text{Ok}$</td>
</tr>
<tr>
<td><strong>Subtype Ok REd</strong></td>
<td>$S \vdash \text{REd}(S')$ type</td>
<td>$S \vdash \text{REd}(S') \prec:: \text{Ok}$</td>
</tr>
<tr>
<td><strong>Subtype Ok Exc</strong></td>
<td>$S \vdash \text{Exc}(S')$ type</td>
<td>$S \vdash \text{Exc}(S') \prec:: \text{Ok}$</td>
</tr>
<tr>
<td><strong>Subtype Ok Arr</strong></td>
<td>$S \vdash \text{Arr}(A)$ type</td>
<td>$S \vdash \text{Arr}(A) \prec:: \text{Ok}$</td>
</tr>
<tr>
<td><strong>Subtype Ok Var</strong></td>
<td>$S \vdash \text{Var}(A)$ type</td>
<td>$S \vdash \text{Var}(A) \prec:: \text{Ok}$</td>
</tr>
<tr>
<td><strong>Subtype Dyn</strong></td>
<td>$S \vdash A \prec:: \text{Dyn}(A)$</td>
<td>$S \vdash A \prec:: \text{Dyn}(B)$</td>
</tr>
</tbody>
</table>

### Subtype Ide

- $S, X <\!: A, S' \vdash X <\!: A$ if $X \not\in \text{Dom}(S')$
- $S, X = A, S' \vdash A : X$
- $S \vdash \text{Fun}(S') : A <\!: \text{Fun}(S'') : B$
- $S \vdash \text{Arr}(A) <\!: \text{Arr}(B)$
- $S \vdash_{\text{Tup}} S_0, S_i <\!: S'_i $ $i = 1 \ldots n$ $1 \leq i$
- $S \vdash_{\text{REd}} S' \prec:\: S''$
- $S \vdash_{\text{Exc}} S' \prec:\: S''$
- $S \vdash_{\text{Var}} A \prec:\: A$
- $S \vdash \text{Dyn}(A) \prec:\: \text{Dyn}(B)$

### Subtype Non-Rec Rec

- $A \neq \text{Rec}(Y, D')$
- $S \vdash A <\!: B_j \{\leftarrow X_1 = \text{Rec}(X_1, D), \ldots, X_n = \text{Rec}(X_n, D)\}$
- $S \vdash_{\text{Rec}} X_j, X_1 <\!: A_1 = B_1 \ldots X_n <\!: A_n = B_n$

### Subtype Rec Non-Rec

- $A \neq \text{Rec}(Y, D')$
- $S \vdash B_j \{\leftarrow X_1 = \text{Rec}(X_1, D), \ldots, X_n = \text{Rec}(X_n, D)\} <\!: A$
- $S \vdash_{\text{Rec}} X_j, X_1 <\!: A_1 = B_1 \ldots X_n <\!: A_n = B_n <\!: A$
5.10 Signatures of Bindings

**[Bind empty]**

\[ S \vdash \emptyset : \emptyset \]

**[Bind ide]**

\[ S \vdash D :: S', S, S' \vdash a : A \]

\[ S \vdash D, x = a :: S', x : A \]

**[Bind ide restrict]**

\[ S \vdash D :: S', S, S' \vdash a : A \]

\[ S \vdash D, x : A = a :: S', x : A \]

**[Bind Ide]**

\[ S \vdash D :: S', S, S' \vdash A :: type \]

\[ S \vdash D, X = A :: S', X = A \]

**[Bind Ide restrict]**

\[ S \vdash D :: S', S, S' \vdash B :: A \]

\[ S \vdash D, X :: A = B :: S', X = B \]

**[Bind and]**

\[ S \vdash D :: S', S, S' \vdash E_i :: S_i \quad i = 1 \ldots n \quad 1 \leq n \]

\[ S \vdash D, E_1 \ldots E_n :: S', S_1, \ldots, S_n \]

**[Bind rec]**

\[ S \vdash D :: S', S, S', x_1 : A_1, \ldots, x_n : A_n \vdash a_i : A_i \quad i = 1 \ldots n \quad 1 \leq n \]

\[ S \vdash D, \text{rec}(x_1 : A_1 = a_1 \ldots, x_n : A_n = a_n) :: S', x_1 : A_1, \ldots, x_n : A_n \]

**[Bind open]**

\[ S \vdash D :: S' \quad S, S' \vdash p \quad S' :: S'' \]

\[ S \vdash D, \text{open}(p) :: S', \text{qualify}(S'' ; p ; S'') \]

**[Bind open restrict]**

\[ S \vdash D :: S' \quad S, S' \vdash p \quad S'' \quad S \vdash A :: S'' \quad S \vdash S'' :: S''' \]

\[ S \vdash D, \text{open}(p, A) :: S', \text{qualify}(S''' ; p ; S'') \]
5.11 Types of Values

\[
\begin{array}{ll}
\text{[Value subsumption]} & S \vdash a : A \quad S \vdash A <; B \\
& S \vdash a : B \\

\text{[Value function]} & S, S' \vdash a : A \\
& S \vdash \text{fun}(S')a \vdash \text{Fun}(S') : A \\

\text{[Value array]} & S \vdash D :: x_1 : A \ldots x_n : A \quad 1 \leq n \\
& S \vdash \text{arr}(D) : \text{Arr}(A) \\

\text{[Value index]} & S \vdash a : \text{Arr}(A) \quad S \vdash b : \text{Int} \\
& S \vdash a[b] : A \\

\text{[Value tuple]} & S \vdash D :: S' \\
& S \vdash \text{tup}(D) : \text{Tup}(S' ; \text{Case}(?, \varnothing)) \\

\text{[Value tuple variant]} & S \vdash A : \text{Tup}(S' ; \text{Case}(z_1, S_1) \ldots \text{Case}(z_n, S_n)) \\
& S \vdash D :: S', S_i \quad 1 \leq z_i \leq n \\
& S \vdash \text{tup}(D ; z_i ; A) : A \\

\text{[Value variant project]} & S \vdash x : \text{Tup}(S' ; \text{Case}(z_1, S_1) \ldots \text{Case}(z_n, S_n)) \\
& S \vdash x!z_i : \text{Tup}(S', S_i ; \text{Case}(?, \varnothing)) \\

\text{[Value variant test]} & S \vdash x : \text{Tup}(S' ; \text{Case}(z_1, S_1) \ldots \text{Case}(z_n, S_n)) \\
& S \vdash x?z_i : \text{Bool} \\

\text{[Value record]} & S \vdash D :: S' \quad S \vdash S' \text{ record} sig \\
& S \vdash \text{rcd}(D) : \text{Rcd}(S') \\

\text{[Value record dot]} & S \vdash a : \text{Rcd}(S' ; x : A, S'' ; C) \quad x \notin \text{Dom}(S'') \\
& S \vdash a.x : \text{qualify}(A ; a ; S') \\

\text{[Value extend]} & S \vdash a : \text{Rcd}(S') \\
& S, S' \vdash D :: S'' \quad S \vdash S', S'' \text{ record} sig \\
& S \vdash \text{extend}(a, D) : \text{Rcd}(S', S'') \\

\text{[Value exception]} & S \vdash a : \text{String} \\
& S \vdash \text{exc}(a, D) : \text{Exec}(S') \\

\text{[Value seq empty]} & S \vdash D :: S' \\
& S \vdash \text{seq}(\varnothing) : \text{Ok} \\

\text{[Value seq ide]} & S \vdash S \vdash D :: S' \quad S, S' \vdash a : A \\
& S \vdash \text{seq}(\varnothing) : \text{Ok} \\

\text{[Value seq ide restrict]} & S \vdash D :: S' \quad S, S' \vdash a : A \\
& S \vdash \text{seq}(D, x : A = a) : A \\

\text{[Value seq Ide]} & S \vdash D :: S' \quad S, S' \vdash A \text{ type} \\
& S \vdash \text{seq}(D, X = A) : \text{Ok}
\end{array}
\]
[Value seq Ide restrict]

\[ S \vdash D :: S' \quad S, S' \vdash B <; A \]

\[ S \vdash \text{seq}(D, X <; A = B) : \text{Ok} \]

[Value if]

\[ S \vdash a : \text{Bool} \quad S \vdash b : B \quad S \vdash b' : B \]

\[ S \vdash \text{if}(a, b, b') : B \]

[Value case]

\[ S \vdash a : \text{Tup}(S'; \text{Case}(z_1', S_1) \ldots \text{Case}(z_m', S_m)) \]

\[ S \vdash b : B \]

\[ S, y_i : \text{Tup}(S', S''; \text{Case}(?, \odot)) \vdash b_i : B \quad 0 \leq n \quad i = 1 \ldots n \]

\[ S, S' \vdash S'' <; S_{z_{ij}} \quad j = 1 \ldots k_i \]

\[ \bigcup_{i=1}^{n} \bigcup_{j=1}^{k_i} z_{ij} \subseteq \bigcup_{i=1}^{m} z_i' \quad z_{ij} \neq z_{ij}', i \neq i', j \neq j' \]

\[ S \vdash \text{case}(a, (z_1 \ldots z_{1k_1}, y_1, b_1) \ldots (z_{n1} \ldots z_{nk_n}, y_n, b_n), b) : B \]

[Value case exhaustive]

\[ S \vdash a : \text{Tup}(S'; \text{Case}(z_1', S_1) \ldots \text{Case}(z_m', S_m)) \]

\[ S, y_i : \text{Tup}(S', S''; \text{Case}(?, \odot)) \vdash b_i : B \quad 1 \leq n \quad i = 1 \ldots n \]

\[ S, S' \vdash S'' <; S_{z_{ij}} \quad j = 1 \ldots k_i \]

\[ \bigcup_{i=1}^{n} \bigcup_{j=1}^{k_i} z_{ij} = \bigcup_{i=1}^{m} z_i' \quad z_{ij} \neq z_{ij}', i \neq i', j \neq j' \]

\[ S \vdash \text{case}(a, (z_1 \ldots z_{1k_1}, y_1, b_1) \ldots (z_{n1} \ldots z_{nk_n}, y_n, b_n)) : B \]

[Value loop]

\[ S \vdash a : \text{Ok} \]

\[ S \vdash \text{loop}(a) : \text{Ok} \]

[Value exit]

\[ \vdash S \text{ sig} \]

\[ S \vdash \text{exit} : \text{Nok} \]

[Value while]

\[ S \vdash a : \text{Bool} \quad S \vdash b : \text{Ok} \]

\[ S \vdash \text{while}(a, b) : \text{Ok} \]

[Value for]

\[ S \vdash a : \text{Int} \quad S \vdash b : \text{Int} \quad S, x : \text{Int} \vdash c : \text{Ok} \]

\[ S \vdash \text{for}(x, a, b, c) : \text{Ok} \]

[Value try]

\[ S \vdash a : B \]

\[ S \vdash b : B \]

\[ S \vdash a_i : \text{Exc}(S_i) \quad i = 1 \ldots n \quad 0 \leq n \]

\[ S, x_i : \text{Tup}(S_i; \text{Case}(?, \odot)) \vdash b_i : B \quad i = 1 \ldots n \]

\[ S \vdash \text{try}(a_i, (a_1, x_1, b_1) \ldots (a_n, x_n, b_n), b) : B \]

[Value raise]

\[ S \vdash a : \text{Exc}(S') \quad S \vdash D :: S' \]

\[ S \vdash \text{raise}(a, D) : \text{Nok} \]

[Value reraise]

\[ S \vdash \text{sig} \]

\[ S \vdash \text{reraise} : \text{Nok} \]

5.12 Restrictions

The rule [Subtype Var] is not applicable in function signatures since the implementation of variable parameters (l-value bindings) is incompatible with the implementation of value parameters (r-value bindings). This restriction is checked statically.
References


Cardelli 1990: Cardelli, L. The Quest language and system (tracking draft). Digital Systems Research Center, DEC SRC Palo Alto, 1990. (shipped as part of the Quest V.12 system distribution).


Cardelli 1992b: Cardelli, L. F-sub, the system. Digital systems research center, DEC SRC Palo Alto, Februar 1992. (shipped as part of the Fsub 1.4 system distribution).


