Bericht Nr. 160

Definition of the Tycoon Language Tl
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

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 (Tl, *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 Tl, 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, (parital) type hiding and run-time type discrimination [Matthes and Schmidt 1991b],

[1] Tycoon: Typed COmmunicating Objects in Open eNvironmens

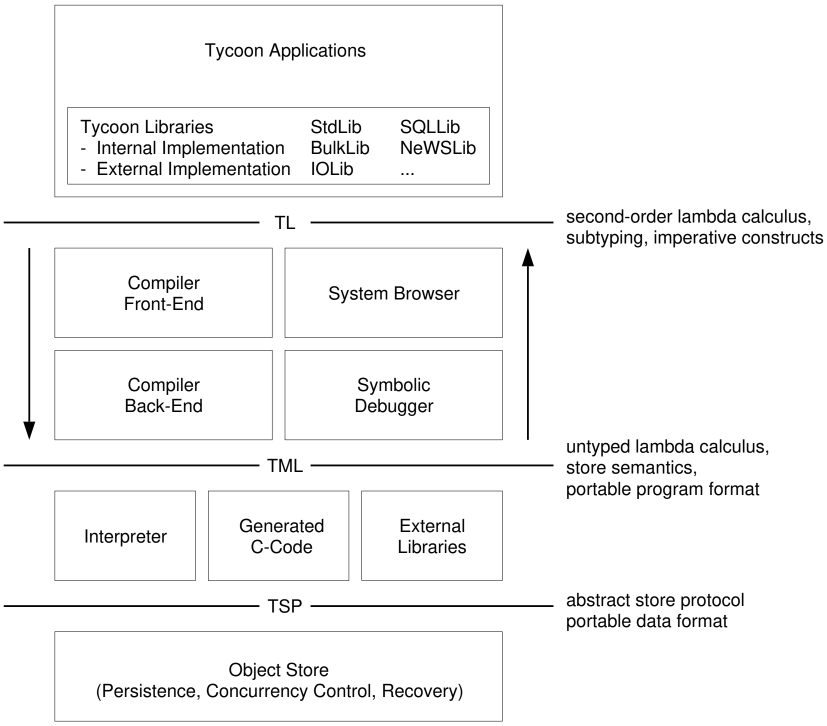

Figure 1: Layers and interfaces of the Tycoon system

▷ supports the operational requirements of todays 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, unparser, 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 develped at the University of St. Andrews [Brown and Rosenberg 1991]. Much emphasis is put on the exchangability 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 <:**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 paticular 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 ***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 ***Let*** *A* = ***Tuple*** *name :String* ***end*** provides less information than the type ***Let*** *B* = ***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 (**Ok**, top type) and the unsatisfiable specification (**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 ***Let*** $X = B$ and may involve the constants **Ok** and **Nok**.

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

[Subtype Oper]

$$\frac{S \vdash S'' <:: S' \qquad S, S'' \vdash A <: B}{S \vdash \mathbf{Oper}(S')A <: \mathbf{Oper}(S'')B}$$

For example, given

Let *F* = ***Oper***(*X* <:***Tuple*** *name :String* ***end***) ***Tuple*** *x :X y :X* ***end***
Let *G* = ***Oper***(*X* <:***Tuple*** *name :String age :Int* ***end***) ***Tuple*** *x :X* ***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 ***Oper**(X <:**Ok**) **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:

Let *Nat* = ***Tuple*** *T<:**Ok** zero :T succ(:T) :T eq(:T :T) :T* ***end***
let *nat1* = ***Tuple Let*** *T* = *Int* ... ***end***
let *nat2* = ***Tuple Let*** *T* = *String* ... ***end***
let *null(x :nat1.T) = nat1.eq(x nat1.zero)*
let *null2(rep :Nat x :rep.T) = rep.eq(x rep.zero)*

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

Let Rec *List(A <: **Ok**)* = ***Tuple*** *hd :A tl :List(A)* ***end***
Let Rec *List2(A <: **Ok**)* = ***Tuple*** *hd :A*
  *tl :**Tuple** hd :A tl :List(A)* ***end***
end

none of the following subtype relationships is derivable:

*List(**Ok**) <:List2(**Ok**)*
*List2(**Ok**) <:List(**Ok**)*
List <:List2
List2<:List

On the other hand, a single step "unfolding" of *List(A)* in the definition of *List* yields a type operator that is structurally equivalent to *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

Let Rec *List(E <:**Ok**)* = ***Tuple***
  cons(:E) :List(E)
  hd() :E

```
  tl() :List(E)
  map(E2 <:Ok f(:E) :E2) :List(E2)
end
Let Rec List2(E <:Ok) = Tuple
  cons(:E) :List2(E)
  hd() :E
  tl() :List2(E)
  map(E2 <:Ok f(:E) :E2) :List2(E2)
  powerset(p(:E) :Bool) :List2(List(E))
end
```

List2(Int) <:*List(Int)* and *List2* <:*List* are derivable type judgements in TL.

Tuples and Variants The concepts of labeled cartesian product types and discriminated union types are captured by tuples and variants in TLthat 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

```
Let Address = Tuple city :String zip :Int end
let a :Address = tuple let city = "Hamburg"  let zip = 2000 end
```

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

```
Let NewAddress = Variant city :String
 case national with zip :Int
 case international with country :String  zip :String
end
let print(adr :NewAddress) :Ok = ...
print(a)
```

Extensible Records As argued in [Atkinson and Morrison 1988; Morrison *et al.* 1987; Schmidt and Matthes 1991], long-lived data-intensive applications require naming schemes that are capable of handling *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 TL, record bindings are unordered. Neither duplicate nor anonymous variable names are permitted in record bindings and record signatures.

let *db* = ***record let*** *parts* = ... ***let*** *suppliers* = ... ***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 ***let*** *newDB* = ***extend*** *db* ***with let*** *assembly* = ...***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 *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 *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 TLgives 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 **extend** operator is not required: The record type ***Record*** *S* ***end*** is a subtype of ***Record*** *S'* ***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 TL are **interface**, **module** and **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 **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 **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 TL do not introduce *new* naming, typing and binding concepts into the language, they simply *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 $F_{\leq}$[Pierce 1992]. Decidability problems arise through the subtyping rule [Subtype Fun] in combiation 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 Tl 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)
$A \mid B$	means A or B
[A]	means ("" | A)
{ A }	means ("" | A { A })

3.1 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 *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::=
    "(" | ")" | "{" | "}" | "[" | "]" | "." | "," | ";"
;
digit::=
    "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
;
alpha::=
    "A" | "B" | ... | "Z" | "a" | "b" | ... | "z"
;
reserved::=
    "~" | " "
;
special::=
    "@" | "#" | "$" | "%" | "&" | "*" | "_" | "+" | "=" | "-" |
    "|" | "\" | "‘" | ":" | "<" | ">" | "/" | "^" | "?" | "!"
;
escape::=
    "\" ("n" | "t" | "r" | "f" | "\" | "’" | """ | *digit digit digit*)
;

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

`\n`	new line
`\t`	tab
`\r`	carriage return
`\f`	form feed
`\'`	'
`\"`	"
`\\`	\
`\nnn`	A single character with the code *nnn*
	(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 elsif 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*
;

ValueBindings::=
  { ***let*** [***rec***] *ValueBinding* { ***and*** *ValueBinding* } }
;
ValueBinding::=
  [***var***] *ValueIde Parameters* [":" *Type*] "=" *Value*
;

3.5 Values

Value::=
  fun "(" *Signatures* ")" [":" *Type*] *Value* |
  assert *Value* |
  $Value_1$
;
$Value_1$::=
  $Value_2$ { (***orif*** | ***andif*** | *colonInfix*) $Value_2$ }
;
$Value_2$::=
  $Value_3$ { *infix* $Value_3$ }
;
$Value_3$::=
  $Value_4$ { "(" *Bindings* ")" | "?" *CaseIde* | "!" *CaseIde* |
    "." *FieldIde* | "[" *Value* "]" |
    of *Bindings* ***end*** }
;
$Value_4$::=
  "{" *Value* "}" |
  ValueIde |
  ok |
  int | *char* | *string* | *real* | *longreal* |
  tuple *Bindings* ***end*** |
  variant *CaseIde* ***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*** *CaseIdeList* [***with*** *ValueIde*] ***then*** *Bindings* }
    [***else*** *Bindings*] ***end*** |
  loop *Bindings* ***end*** |
  exit |
  while *Value* ***do*** *Bindings* ***end*** |
  for *ValueIde* "=" *Value* (***upto*** | ***downto***) *Value* ***do*** *Bindings* ***end*** |
  try *Bindings* { ***when*** *Value* [***with*** *ValueIde*] ***then*** *Bindings* }

[***else*** *Bindings*] ***end*** |
raise *Value* [***with*** *Bindings*] ***end*** |
reraise |
typeRep_new *"(" ":" Type ")"* |
dynamic_new *"(" Value ")"* |
dynamic_be *"(" Value ":" Type ")"*
;

3.6 Signatures

Signatures::=
{ *TypeSignatures* | *ValueSignatures* | *TypeBindings* | ***Repeat*** *Type* }
;
TypeSignatures::=
[***Dyn***] [*TypeIdeList Parameters*] *"<:" Type*
;
ValueSignatures::=
[**var**] [*ValueIdeList Parameters*] *":" Type*
;
Parameters::=
{ *"(" Signatures ")"* }
;

3.7 Types

Type::=
Oper *"(" Signatures ")"* [*"<:" Type*] *Type* |
Fun *"(" Signatures ")" ":" Type* |
$Type_1$
;
$Type_1$*::=*
$Type_2$ { *colonInfix* $Type_2$ }
;
$Type_2$*::=*
$Type_3$ { *infix* $Type_3$ }
;
$Type_3$*::=*
$Type_4$ { *"("* { *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 | ***infix*** (*infix* | *colonInfix*)
;

4 The Tl Abstract Syntax

The concrete syntax of TL 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 TL. It also constitutes the canonical internal representation used by the TL language processors.

4.1 Syntactic Objects of Tl

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

$S \in$ *Sig*	sequences of signatures
$D \in$ *Bind*	sequences of bindings
$E \in$ *BindElem*	single bindings
$A, B \in$ *Type*	types
$C \in$ *Case*	variants in tuple types
$z \in$ *CaseIde*	variant labels
$X, Y \in$ *Ide*	type identifiers
$x, y \in$ *ide*	value identifiers
$a, b, c \in$ *Value*	values
$p, q \in$ *Qualifier*	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 Tl Programs

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

▹ introduces the anonymous identifier **?** in bindings

$$:A \text{ a} \quad \Rightarrow \quad \textbf{\textit{Let ?}} = A \ \textbf{\textit{let ?}} = \text{a}$$

and in signatures

$$<:A\ :B \quad \Rightarrow \quad \textbf{\textit{?}} <:A\ \textbf{\textit{?}}\ :B$$

An anonymous identifier **?** matches any other identifier x, X in the subtyping rules for signature matching [Subsig ide, Subsig Ide, Subsig Ide Let] in § 5.8.

▹ expands abbreviating notations in bindings

$$
\begin{array}{lcl}
\textbf{\textit{let}}\ x(S_1)\ldots(S_n)\ {:}A = a & \Rightarrow & \textbf{\textit{let}}\ x = \textbf{\textit{fun}}(S_1)\ldots\textbf{\textit{fun}}(S_n)\ {:}A = a \\
\textbf{\textit{let}}\ x(S_1)\ldots(S_n) = a & \Rightarrow & \textbf{\textit{let}}\ x = \textbf{\textit{fun}}(S_1)\ldots\textbf{\textit{fun}}(S_n) = a \\
\textbf{\textit{Let}}\ X(S_1)\ldots(S_n) <{:}A = B & \Rightarrow & \textbf{\textit{Let}}\ X = \textbf{\textit{Oper}}(S_1)\ldots\textbf{\textit{Oper}}(S_n) <{:}A = B \\
\textbf{\textit{Let}}\ X(S_1)\ldots(S_n) = A & \Rightarrow & \textbf{\textit{Let}}\ X = \textbf{\textit{Oper}}(S_1)\ldots\textbf{\textit{Oper}}(S_n) = A
\end{array}
$$

and in signatures

$$
\begin{array}{lcl}
x_1, \ldots, x_n\ {:}A & \Rightarrow & x_1\ {:}A \ldots x_n\ {:}A \\
\textbf{\textit{var}}\ x_1, \ldots, x_n\ {:}A & \Rightarrow & \textbf{\textit{var}}\ x_1\ {:}A \ldots \textbf{\textit{var}}\ x_n\ {:}A \\
\textbf{\textit{Dyn}}\ X_1, \ldots, X_n <{:}A & \Rightarrow & \textbf{\textit{Dyn}}\ X_1 <{:}A \ldots \textbf{\textit{Dyn}}\ X_n <{:}A \\
x(S_1)\ldots(S_n)\ {:}A & \Rightarrow & x\ {:}\textbf{\textit{Fun}}(S_1)\ldots{:}\textbf{\textit{Fun}}(S_n)\ {:}A \\
X(S_1)\ldots(S_n) <{:}A & \Rightarrow & X\ {:}\textbf{\textit{Oper}}(S_1)\ldots\textbf{\textit{Oper}}(S_n)\ A
\end{array}
$$

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

$$
\begin{array}{lcl}
x_1\ \text{infix}\ x_2 & \Rightarrow & \{\text{infix}(x_1\ x_2)\} \\
x_1\ \text{colonInfix}\ x_2 & \Rightarrow & \{\text{colonInfix}(x_1\ x_2)\} \\
f\ \textbf{\textit{of}}\ x_1 \ldots x_n\ \textbf{\textit{end}} & \Rightarrow & f(\textbf{\textit{array}}\ x_1 \ldots x_n\ \textbf{\textit{end}})
\end{array}
$$

▹ transforms infix type operator applications into prefix notation

$$
\begin{array}{lcl}
X_1\ \text{infix}\ X_2 & \Rightarrow & \{\text{infix}(X_1\ X_2)\} \\
X_1\ \text{colonInfix}\ X_2 & \Rightarrow & \{\text{colonInfix}(X_1\ X_2)\}
\end{array}
$$

▹ replaces missing **else** branches in **if** and **try** expressions (but not in **case** expressions) by an empty expression sequence (that evaluates to **ok**, see [Value seq empty] in § 5.11), for example,

$$\textbf{\textit{if}}\ a\ \textbf{\textit{then}}\ b\ \textbf{\textit{end}} \quad \Rightarrow \quad \textbf{\textit{if}}\ a\ \textbf{\textit{then}}\ b\ \textbf{\textit{else end}}$$

▹ normalizes **elsif** branches into nested **if** expressions

$$
\begin{array}{lcl}
\begin{array}{l}
\textbf{\textit{if}}\ a_1\ \textbf{\textit{then}}\ b_1 \\
\textbf{\textit{elsif}}\ a_2\ \textbf{\textit{then}}\ b_2 \\
\\
\quad\ldots \\
\\
\textbf{\textit{end}}
\end{array}
& \Rightarrow &
\begin{array}{l}
\textbf{\textit{if}}\ a_1\ \textbf{\textit{then}}\ b_1 \\
\textbf{\textit{else}} \\
\quad \textbf{\textit{if}}\ a_2\ \textbf{\textit{then}}\ b_2 \\
\quad\quad \ldots \\
\quad \textbf{\textit{end}} \\
\textbf{\textit{end}}
\end{array}
\end{array}
$$

▹ replaces assertions by conditionals

$$
\textbf{assert}\ a \quad \Rightarrow \quad
\begin{array}{l}
\textbf{\textit{if}}\ a\ \textbf{\textit{then ok}} \\
\textbf{\textit{else}} \\
\quad \textbf{\textit{raise exception}} \\
\textit{``Assertion in file \ldots at line \ldots failed''} \\
\quad \textbf{\textit{end}} \\
\textbf{\textit{end}}
\end{array}
$$

$S ::=$	$\emptyset$	empty signature
$\mid$	$S, x : A$	value signature
$\mid$	$S, X <: A$	type signature
$\mid$	$S, X = A$	type definition
$\mid$	$S, \mathbf{Repeat}(A)$	signature inheritance
;		
$D ::=$	$\emptyset$	empty binding
$\mid$	D, E	sequential binding
$\mid$	$D, E_1 \mid \ldots \mid E_n$	parallel binding
$\mid$	$D, \mathbf{open}(x)$	binding inheritance
$\mid$	$D, \mathbf{open}(x, A)$	binding projection
;		
$E ::=$	$x = a$	value binding
$\mid$	$x : A = a$	constrained value binding
$\mid$	$X = A$	type binding
$\mid$	$X <: A = B$	constrained type binding
;		
$A, B ::=$	$\mathbf{Ok}$	supertype of all types (*top*)
$\mid$	$\mathbf{Nok}$	subtype of all types (*bottom*)
$\mid$	X	type identifier
$\mid$	$p.X$	abstract type identifier
$\mid$	$\mathbf{Bool} \mid \mathbf{Int} \mid \mathbf{String}$	base type identifiers
$\mid$	$\mathbf{Arr}(A)$	array type
$\mid$	$\mathbf{Fun}(S) : A$	(polymorphic) function type
$\mid$	$\mathbf{Tup}(S; C)$	tuple type (with variants)
$\mid$	$\mathbf{Rcd}(S)$	record type
$\mid$	$\mathbf{Exc}(S)$	exception type
$\mid$	$\mathbf{Rec}(X, X_1 <: A_1 = B_1 \ldots X_n <: A_n = B_n)$	recursive type ($1 \leq n$)
$\mid$	$\mathbf{Var}(A)$	type of mutable values
$\mid$	$\mathbf{Dyn}(A)$	dynamic type
$\mid$	$\mathbf{Oper}(X_1 <: A_1, \ldots, X_n <: A_n)B$	type operator
$\mid$	$A(D)$	type operator application
;		
$C ::=$	$\mathbf{Case}(z_1, S_1) \ldots \mathbf{Case}(z_n, S_n)$	variants in tuple types ($0 \leq n$)
;		
$p ::=$	x	value signature selector
$\mid$	$p.x$	field signature selector
;		

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

Let Rec *X<:**Ok** = F(Y)* ***and*** *Y<:**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:

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

and

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

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 (<:**Ok** in the example above) are required to simplify the checking of type operator applications within the recursive binding, for example

Let Rec *A1 = A2(A2)* ***and*** *A2 = A3(A3)* ***and*** ...
and *An =* ***Oper****(X(Y <:**Ok**) <:**Ok**) X(Int)*

has to be written as

Let *Unary =* ***Oper****(X(Y <:**Ok**) <:**Ok**)* ***Ok***
Let *A1 <:Unary = A2(A2)* ***and*** *A2 <:Unary = A3(A3)* ***and*** ...
and *An <:Unary =* ***Oper****(X(Y <:**Ok**) <:**Ok**) X(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.

$a, b ::=$	**ok**	the canonical value of type **Ok**
$\mid$	*int*	integer literals (s. § 3.1)
$\mid$	*real*	floating point literals (s. § 3.1)
$\mid$	*longreal*	double precision floating point literals (s. § 3.1)
$\mid$	*char*	character literals (s. § 3.1)
$\mid$	*string*	string literals (s. § 3.1)
$\mid$	x	value identifier
$\mid$	$\mathbf{fun}(S)a$	(polymorphic) function constructor
$\mid$	$a(D)$	function application
$\mid$	$\mathbf{arr}(D)$	array constructor
$\mid$	$a[b]$	array element selection
$\mid$	$\mathbf{tup}(D, z, D')$	tuple constructor (with variants)
$\mid$	$\mathbf{rcd}(D)$	record constructor
$\mid$	$a.x$	tuple, record and execption field selection
$\mid$	$a!z$	variant projection
$\mid$	$a?z$	variant test
$\mid$	$\mathbf{extend}(a, D)$	record extension
$\mid$	$\mathbf{exc}(a, D)$	exception value generation
$\mid$	$\mathbf{seq}(D)$	sequential evaluation
$\mid$	$\mathbf{if}(a, b_1, b_2)$	conditional
$\mid$	$\mathbf{case}(a,$ $(z_{11} \ldots z_{1k_1}, y_1, b_1) \ldots$ $(z_{n1} \ldots z_{nk_n}, y_n, b_n), b)$	variant analysis $(0 \leq n, 1 \leq k_i)$
$\mid$	$\mathbf{case}(a,$ $(z_{11} \ldots z_{1k_1}, y_1, b_1) \ldots$ $(z_{n1} \ldots z_{nk_n}, y_n, b_n))$	exhaustive variant analysis $(1 \leq n, 1 \leq k_i)$
$\mid$	$\mathbf{loop}(a_1)$	loop
$\mid$	**exit**	loop exit
$\mid$	$\mathbf{while}(a, b)$	pre-check loop
$\mid$	$\mathbf{for}(x, a_1, a_2, a_3)$	integer iteration
$\mid$	$\mathbf{try}(a,$ $(a_1, y_1, b_1) \ldots (a_n, y_n, b_n), b)$	exception handling $(0 \leq n)$
$\mid$	$\mathbf{raise}(a, D)$	exception generation
$\mid$	**reraise**	exception propagation
$\mid$	$\mathbf{andif}(a, b)$	conditional conjunction
$\mid$	$\mathbf{orif}(a, b)$	conditional disjunction
;		

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:[2]

$:= (A <:\textbf{\textit{Ok}}\ \textbf{var}\ lhs :A\ \ rhs :A)$

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 **else** branch.

[2] Different implementations of the assignment function may attach recovery or concurrency control functionality to the plain destructive update operator.

5 The Static Semantics of Tl

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 TL 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 TL are implicitly defined by a mapping of TL 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 TL:

$A \downarrow X$	The type A is contractive in variable X	§ 5.4
$S\ sig$	The signatures S are well-formed	§ 5.5
$S\ rcdsig$	The signatures S are well-formed record signatures	§ 5.5
$A\ type$	The type A is well-formed	§ 5.6
$A \overset{Sig}{::} S$	The type A has signatures S	§ 5.7
$p \overset{Sig}{::} S$	The selector p identifies a value with signatures S	§ 5.7
$S <:: S'$	S are subsignatures of S'	§ 5.8
$S \overset{Tup}{<::} S'$	S are tuple subsignatures of S'	§ 5.8
$S \overset{Rcd}{<::} S'$	S are record subsignatures of S'	§ 5.8
$A <: B$	A is a subtype of B	§ 5.9
$D :: S$	The bindings D have signatures S	§ 5.10
$a : A$	The value a has type A	§ 5.11

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' the signatures in S followed by signatures in S'
D, D' the bindings in D 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:

$x \in Dom(S)$ the value variable x is defined in S
$X \in Dom(S)$ the type variable X is defined in S

The type rules of TL 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:

$S \vdash J$ in the static environment S the judgement J 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:

$A\{X \leftarrow B\}$ A where each free occurence of X is substituted by B

This substitution operation could be made more explicit by replacing the applied occurence of a variable name by the (relative) index of its defining occurence witin the static environment and thereby reducing the substitution operation to a mere index manipulation [de Bruijn 1972; Abadi *et al.* 1990][3].

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:

$A\{\Leftarrow D\}$ A where all $X, x \in Dom(D)$ are substituted as follows

[Subst empty]

$$\frac{}{S \vdash A\{\Leftarrow \emptyset\} = A}$$

[Subst Ide]

$$\frac{S \vdash D :: S' \qquad S, S' \vdash D' :: X <: B}{S \vdash A\{\Leftarrow D, D'\} = A\{\Leftarrow D\}\{X \leftarrow B\}}$$

[Subst Let]

$$\frac{S \vdash D :: S' \qquad S, S' \vdash D' :: X = B}{S \vdash A\{\Leftarrow D, D'\} = A\{\Leftarrow D\}\{X \leftarrow B\}}$$

[Subst ide]

$$\frac{S \vdash D :: S' \qquad S, S' \vdash D' :: x : B}{S \vdash A\{\Leftarrow D, D'\} = A\{\Leftarrow D\}}$$

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.

$\mathbf{qualify}(T; p; S)$ substitute in T each occurence of $X \in Dom(S)$ by $p.X$

where $T \in \mathit{Sig} \cup \mathit{Type}$. This qualification is defined inductively as follows:

$$\mathbf{qualify}(T; p; \emptyset) = T$$

$$\mathbf{qualify}(T; p; S, x : B) = \mathbf{qualify}(T\{x \leftarrow p.x\}; p; S)$$

$$\mathbf{qualify}(T; p; S, X <: B) = \mathbf{qualify}(T\{X \leftarrow p.X\}; p; S)$$

$$\mathbf{qualify}(T; p; S, X = B) = \mathbf{qualify}(T\{X \leftarrow p.X\}; p; S)$$

[3] This is also the mechanism used in the TL type checker.

5.4 Contractive Types

A final auxiliary notation is required in rule [Type Rec] in § 5.6 to restrict the structure of recursive types to *contractive* type expressions.

$S \vdash A \downarrow X$ In environment S type A is contractive in variable X

Systems of recursive type equations $\mathbf{Rec}(X_i, X_1 <: A_1 = B_1 \ldots X_n <: A_n = B_n)$ 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 occuring in the the static environment:

$$Y \downarrow X \Leftrightarrow Y \neq X$$

$$\mathbf{Ok} \downarrow X \quad \mathbf{Nok} \downarrow X \quad \mathbf{Bool} \downarrow X \quad \mathbf{Int} \downarrow X \quad \mathbf{String} \downarrow X$$

$$x.Y \downarrow X \quad \mathbf{Fun}(S') : B \downarrow X \quad \mathbf{Tup}(S'; C) \downarrow X \quad \mathbf{Rcd}(S') \downarrow X \quad \mathbf{Exc}(S') \downarrow X \quad \mathbf{Arr}(A) \downarrow X$$

$$\mathbf{Var}(A) \downarrow X \Leftrightarrow A \downarrow X$$

$$\mathbf{Dyn}(A) \downarrow X \Leftrightarrow A \downarrow X$$

The last three judgements refer to the static environment S:

[Contractive Oper]

$$\frac{S, S' \vdash B \downarrow X}{S \vdash \mathbf{Oper}(S')B \downarrow X}$$

[Contractive Rec]

$$\frac{S \vdash \mathbf{Rec}(X_i, X_1 <: A_1 = B_1 \ldots X_n <: A_n = B_n)\ type}{S \vdash \mathbf{Rec}(X, X_1 <: A_1 = B_1 \ldots X_n <: A_n = B_n) \downarrow Y}$$

[Contractive Apply]

$$\frac{S \vdash A <: \mathbf{Oper}(S')B \quad S \vdash D :: S' \quad S \vdash B\{\Leftarrow D\} \downarrow X}{S \vdash A(D) \downarrow X}$$

The above definition is well-founded although rule [Contractive Rec] itself requires the definition of well-founded types, since the syntax of TL does not allow statically nested recursive type declarations.

5.5 Well-formed Signatures

[Sig empty]

$$\frac{}{\vdash \emptyset\ sig}$$

[Sig ide]

$$\frac{S \vdash A\ type}{\vdash S, x : A\ sig}$$

[Sig Ide]

$$\frac{S \vdash A\ type}{\vdash S, X <: A\ sig}$$

[Sig Ide Let]

$$\frac{S \vdash A\ type}{\vdash S, X = A\ sig}$$

[Sig Repeat]

$$\frac{S \vdash A \overset{Sig}{::} S' \quad \vdash S'\ sig}{\vdash S, \mathbf{Repeat}(A)\ sig}$$

[Rcdsig]

$$\frac{\vdash S, S'\ sig \quad (S' = S_1, S_2 \wedge X \in Dom(S_1) \Rightarrow X \notin Dom(S_2))}{S \vdash S'\ rcdsig}$$

5.6 Well-formed Types

[Type Builtin]

$$\frac{\vdash S\ sig \quad A \in \{\mathbf{Ok}, \mathbf{Nok}, \mathbf{Bool}, \mathbf{Int}, \mathbf{String}\}}{S \vdash A\ type}$$

[Type Ide]

$$\frac{\vdash S, X <: A, S'\ sig \quad X \notin Dom(S')}{S, X <: A, S' \vdash X\ type}$$

[Type Ide Let]

$$\frac{\vdash S, X <: A, S'\ sig \quad X \notin Dom(S')}{S, X = A, S' \vdash X\ type}$$

[Type Dot]

$$\frac{S \vdash p \overset{Sig}{::} S' \quad S, S' \vdash X\ type \quad X \in Dom(S')}{S \vdash p.X\ type}$$

[Type Fun]

$$\frac{\vdash S, S'\ sig \quad S, S' \vdash B <: \mathbf{Ok}}{S \vdash \mathbf{Fun}(S') : B\ type}$$

[Type Tup]

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

[Type Rcd]

$$\frac{S \vdash S'\ rcdsig}{S \vdash \mathbf{Rcd}(S')\ type}$$

[Type Exc]

$$\frac{\vdash S, S'\ sig}{S \vdash \mathbf{Exc}(S')\ type}$$

[Type Rec]

$$\frac{\begin{array}{c} S, X_1 <: A_1, \ldots, X_n <: A_n \vdash B_i <: A_i \quad i = 1 \ldots n \qquad 1 \leq j \leq n \\ S, X_1 = B_1, \ldots, X_n = B_n \vdash B_i \downarrow X_i \quad i = 1 \ldots n \end{array}}{S \vdash \mathbf{Rec}(X_j, X_1 <: A_1 = B_1 \ldots X_n <: A_n = B_n)\ type}$$

[Type Var]

$$\frac{S \vdash A <: \mathbf{Ok}}{S \vdash \mathbf{Var}(A)\ type}$$

[Type Dyn]

$$\frac{S \vdash A\ type}{S \vdash \mathbf{Dyn}(A)\ type}$$

[Type Arr]

$$\frac{S \vdash A <: \mathbf{Ok}}{S \vdash \mathbf{Arr}(A)\ type}$$

[Type Oper]

$$\frac{\vdash S, S'\ sig \quad S, S' \vdash B\ type}{S \vdash \mathbf{Oper}(S')B\ type}$$

[Type Apply]

$$\frac{S \vdash A <: \mathbf{Oper}(S')B \quad S \vdash D :: S'}{S \vdash A(D)\ type}$$

5.7 Signatures of Types and Values

[Select ide]
$$\frac{S \vdash x : A \quad S \vdash A \overset{Sig}{::} S'}{S \vdash x \overset{Sig}{::} S'}$$

[Select dot]
$$\frac{S \vdash p \overset{Sig}{::} S' \quad x \in Dom(S') \quad S, S' \vdash x \overset{Sig}{::} S''}{S \vdash p.x \overset{Sig}{::} S''}$$

[Select Tup]
$$\frac{S \vdash A : \mathbf{Tup}(S'; C)}{S \vdash A \overset{Sig}{::} S'}$$

[Select Rcd]
$$\frac{S \vdash A : \mathbf{Rcd}(S')}{S \vdash A \overset{Sig}{::} S'}$$

[Select Exc]
$$\frac{S \vdash A : \mathbf{Exc}(S')}{S \vdash A \overset{Sig}{::} S'}$$

[Select Fun]
$$\frac{S \vdash A : \mathbf{Fun}(S') : B}{S \vdash A \overset{Sig}{::} S'}$$

[Select Oper]
$$\frac{S \vdash A : \mathbf{Oper}(S')B}{S \vdash A \overset{Sig}{::} S'}$$

5.8 Subsignatures

[Subsig reflexive]
$$\frac{\vdash S, S' \ sig}{S \vdash S' <:: S'}$$

[Subsig ide]
$$\frac{S \vdash S' <:: S'' \quad S, S' \vdash A <: B}{S \vdash S', x : A <:: S'', x : B}$$

[Subsig Ide]
$$\frac{S \vdash S' <:: S'' \quad S, S' \vdash A <: B}{S \vdash S', X <: A <:: S'', X <: B}$$

[Subsig Ide Let]
$$\frac{S \vdash S' <:: S'' \quad S, S' \vdash A <: B}{S \vdash S', X = A <:: S'', X <: B}$$

[Subsig Repeat]
$$\frac{S \vdash S' <:: S'' \quad S, S' \vdash A \overset{Sig}{::} S'''}{S \vdash S', \mathbf{Repeat}(A) <:: S'', S'''}$$

[Subsig Tup]
$$\frac{\vdash S, S_1, S_2 \ sig \quad S \vdash S_1 <:: S_1'}{S \vdash S_1, S_2 \overset{Tup}{<::} S_1'}$$

[Subsig Rcd]
$$\frac{S \vdash S_1, S_2, S_3 \ rcdsig \quad S \vdash S_1, S_3 \overset{Rcd}{<::} S_1' \quad S, S_1' \vdash S_2 \overset{Tup}{<::} S_2'}{S \vdash S_1, S_2, S_3 \overset{Rcd}{<::} S_1', S_2'}$$

5.9 Subtypes

[Subtype reflexive]
$$\frac{S \vdash A \ type}{S \vdash A <: A}$$

[Subtype transitive]
$$\frac{S \vdash A <: A' \quad S \vdash A' <: A''}{S \vdash A <: A''}$$

[Subtype Nok]
$$\frac{S \vdash A <: \mathbf{Ok}}{S \vdash \mathbf{Nok} <: A}$$

[Subtype Ok Builtin]
$$\frac{\vdash S \ sig \quad A \in \{\mathbf{Bool}, \mathbf{Int}, \mathbf{String}\}}{S \vdash A <: \mathbf{Ok}}$$

[Subtype Ok Fun]
$$\frac{S \vdash \mathbf{Fun}(S') : B \ type}{S \vdash \mathbf{Fun}(S') : B <: \mathbf{Ok}}$$

[Subtype Ok Tup]
$$\frac{S \vdash \mathbf{Tup}(S';C)\ \mathit{type}}{S \vdash \mathbf{Tup}(S';C) <: \mathbf{Ok}}$$

[Subtype Ok Rcd]
$$\frac{S \vdash \mathbf{Rcd}(S')\ \mathit{type}}{S \vdash \mathbf{Rcd}(S') <: \mathbf{Ok}}$$

[Subtype Ok Exc]
$$\frac{S \vdash \mathbf{Exc}(S')\ \mathit{type}}{S \vdash \mathbf{Exc}(S') <: \mathbf{Ok}}$$

[Subtype Ok Arr]
$$\frac{S \vdash \mathbf{Arr}(A)\ \mathit{type}}{S \vdash \mathbf{Arr}(A) <: \mathbf{Ok}}$$

[Subtype Ok Var]
$$\frac{S \vdash \mathbf{Var}(A)\ \mathit{type}}{S \vdash \mathbf{Var}(A) <: \mathbf{Ok}}$$

[Subtype Ok Dyn]
$$\frac{S \vdash A <: \mathbf{Ok}}{S \vdash \mathbf{Dyn}(A) <: \mathbf{Ok}}$$

[Subtype Ide]
$$\frac{\vdash S, X <: A, S'\ \mathit{sig} \quad X \notin \mathit{Dom}(S')}{S, X <: A, S' \vdash X <: A}$$

[Subtype Ide Let]
$$\frac{\vdash S, X <: A, S'\ \mathit{sig} \quad X \notin \mathit{Dom}(S')}{S, X = A, S' \vdash X <: A}$$

[Subtype Ide Let2]
$$\frac{\vdash S, X <: A, S'\ \mathit{sig} \quad X \notin \mathit{Dom}(S')}{S, X = A, S' \vdash A : X}$$

[Subtype Dot]
$$\frac{S \vdash p \overset{\mathit{Sig}}{::} S', X <: A, S'' \quad X \notin \mathit{Dom}(S'')}{S \vdash p.X <: \mathbf{qualify}(A; p; S')}$$

[Subtype Fun]
$$\frac{S \vdash S'' <:: S' \quad S, S'' \vdash A <: B}{S \vdash \mathbf{Fun}(S') : A <: \mathbf{Fun}(S'') : B}$$

[Subtype Arr]
$$\frac{S \vdash A <: B}{S \vdash \mathbf{Arr}(A) <: \mathbf{Arr}(B)}$$

[Subtype Tup]
$$\frac{\begin{array}{c} S \vdash S_0, S_i \overset{\mathit{Tup}}{<::} S'_0, S'_i \quad i = 1 \ldots n \quad 1 \leq n \\ S \vdash S'_0, S''_j\ \mathit{sig} \quad j = 1 \ldots m \quad 1 \leq m \end{array}}{\begin{array}{ll} S \vdash & \mathbf{Tup}(S_0; \mathbf{Case}(z_1, S_1) \ldots \mathbf{Case}(z_n, S_n)) <: \\ & \mathbf{Tup}(S'_0; \mathbf{Case}(z_1, S'_1) \ldots \mathbf{Case}(z_n, S'_n)\, \mathbf{Case}(z'_1, S''_1) \ldots \mathbf{Case}(z'_m, S''_m)) \end{array}}$$

[Subtype Rcd]
$$\frac{S \vdash S' \overset{\mathit{Rcd}}{<::} S''}{S \vdash \mathbf{Rcd}(S') <: \mathbf{Rcd}(S'')}$$

[Subtype Exc]
$$\frac{S \vdash S' \overset{\mathit{Tup}}{<::} S''}{S \vdash \mathbf{Exc}(S') <: \mathbf{Exc}(S'')}$$

[Subtype Var]
$$\frac{S \vdash A\ \mathit{type}}{S \vdash \mathbf{Var}(A) <: A}$$

[Subtype Dyn Elim]
$$\frac{S \vdash A\ \mathit{type}}{S \vdash \mathbf{Dyn}(A) <: A}$$

[Subtype Dyn]
$$\frac{S \vdash A <: B}{S \vdash \mathbf{Dyn}(A) <: \mathbf{Dyn}(B)}$$

[Subtype Non-Rec Rec]
$$\frac{\begin{array}{c} A \neq \mathbf{Rec}(Y, D') \\ S \vdash A <: B_j\{\Leftarrow X_1 = \mathbf{Rec}(X_1, D), \ldots, X_n = \mathbf{Rec}(X_n, D)\} \end{array}}{S \vdash A <: \mathbf{Rec}(X_j, X_1 <: A_1 = B_1 \ldots X_n <: A_n = B_n)}$$

[Subtype Rec Non-Rec]
$$\frac{\begin{array}{c} A \neq \mathbf{Rec}(Y, D') \\ S \vdash B_j\{\Leftarrow X_1 = \mathbf{Rec}(X_1, D), \ldots, X_n = \mathbf{Rec}(X_n, D)\} <: A \end{array}}{S \vdash \mathbf{Rec}(X_j, X_1 <: A_1 = B_1 \ldots X_n <: A_n = B_n) <: A}$$

[Subtype Rec Rec]

$$\frac{S'' \vdash A_i <: A'_j \qquad S'', S', S \vdash B_i <: B'_j}{S'' \vdash \mathbf{Rec}(X_i, X_1 <: A_1 = B_1 \ldots X_n <: A_n = B_n) <: \mathbf{Rec}(X'_j, X'_1 <: A'_1 = B'_1 \ldots X'_m <: A'_m = B'_m)}$$

where

$$\begin{aligned} S' &= X'_1 <: \mathbf{Rec}(X'_1, D'), \ldots X'_{j-1} <: \mathbf{Rec}(X'_{j-1}, D'), X'_j <: A'_j, \\ &\quad X'_{j+1} <: \mathbf{Rec}(X'_{j+1}, D'), \ldots X'_m <: \mathbf{Rec}(X'_m, D') \\ S &= X_1 <: \mathbf{Rec}(X_1, D), \ldots X_{i-1} <: \mathbf{Rec}(X_{i-1}, D), X_i <: X'_j, \\ &\quad X_{i+1} <: \mathbf{Rec}(X_{i+1}, D), \ldots X_n <: \mathbf{Rec}(X_n, D) \\ D' &= X'_1 <: A'_1 = B'_1, \ldots, X'_m <: A'_m = B'_m \\ D &= X_1 <: A_1 = B_1, \ldots, X_n <: A_n = B_n \end{aligned}$$

[Subtype Oper]

$$\frac{S \vdash S'' <:: S' \qquad S, S'' \vdash A <: B}{S \vdash \mathbf{Oper}(S')A <: \mathbf{Oper}(S'')B}$$

[Subtype Apply]

$$\frac{S \vdash A <: \mathbf{Oper}(S')B \qquad S \vdash D :: S'}{S \vdash A(D) <: B\{\Leftarrow D\}}$$

5.10 Signatures of Bindings

[Bind empty]

$$\frac{\vdash S \ \mathit{sig}}{S \vdash \emptyset :: \emptyset}$$

[Bind ide]

$$\frac{S \vdash D :: S' \qquad S, S' \vdash a : A}{S \vdash D, x = a :: S', x : A}$$

[Bind ide restrict]

$$\frac{S \vdash D :: S' \qquad S, S' \vdash a : A}{S \vdash D, x : A = a :: S', x : A}$$

[Bind Ide]

$$\frac{S \vdash D :: S' \qquad S, S' \vdash A \ \mathit{type}}{S \vdash D, X = A :: S', X = A}$$

[Bind Ide restrict]

$$\frac{S \vdash D :: S' \qquad S, S' \vdash B <: A}{S \vdash D, X <: A = B :: S', X = B}$$

[Bind and]

$$\frac{S \vdash D :: S' \qquad S, S' \vdash E_i :: S_i \quad i = 1 \ldots n \quad 1 \leq n}{S \vdash D, E_1 | \ldots | E_n :: S', S_1, \ldots, S_n}$$

[Bind rec]

$$\frac{S \vdash D :: S' \qquad S, S', x_1 : A_1, \ldots, x_n : A_n \vdash a_i : A_i \quad i = 1 \ldots n \quad 1 \leq n}{S \vdash D, \mathbf{rec}(x_1 : A_1 = a_1 | \ldots | x_n : A_n = a_n) :: S', x_1 : A_1, \ldots, x_n : A_n}$$

[Bind open]

$$\frac{S \vdash D :: S' \qquad S, S' \vdash p \overset{Sig}{::} S''}{S \vdash D, \mathbf{open}(p) :: S', \mathbf{qualify}(S''; p; S'')}$$

[Bind open restrict]

$$\frac{S \vdash D :: S' \qquad S, S' \vdash p \overset{Sig}{::} S'' \qquad S \vdash A \overset{Sig}{::} S''' \qquad S \vdash S'' <:: S'''}{S \vdash D, \mathbf{open}(p, A) :: S', \mathbf{qualify}(S'''; p; S'')}$$

5.11 Types of Values

[Value subsumption]

$$\frac{S \vdash a : A \quad S \vdash A <: B}{S \vdash a : B}$$

[Value ok]

$$\frac{\vdash S \ sig}{S \vdash \mathbf{ok} : \mathbf{Ok}}$$

[Value ide]

$$\frac{\vdash S, x : A, S' \ sig \quad x \notin Dom(S')}{S, x : A, S' \vdash x : A}$$

[Value fun]

$$\frac{S, S' \vdash a : A}{S \vdash \mathbf{fun}(S')a : \mathbf{Fun}(S') : A}$$

[Value apply]

$$\frac{S \vdash a : \mathbf{Fun}(S') : A \quad S \vdash D :: S'}{S \vdash a(D) : A\{\Leftarrow D\}}$$

[Value array]

$$\frac{S \vdash D :: x_1 : A \ldots x_n : A \quad 1 \leq n}{S \vdash \mathbf{arr}(D) : \mathbf{Arr}(A)}$$

[Value array empty]

$$\frac{\vdash S \ sig}{S \vdash \mathbf{arr}() : \mathbf{Arr}(\mathbf{Nok})}$$

[Value index]

$$\frac{S \vdash a : \mathbf{Arr}(A) \quad S \vdash b : \mathbf{Int}}{S \vdash a[b] : A}$$

[Value tuple]

$$\frac{S \vdash D :: S'}{S \vdash \mathbf{tup}(D) : \mathbf{Tup}(S'; \mathbf{Case}(?, \oslash))}$$

[Value tuple variant]

$$\frac{S \vdash A : \mathbf{Tup}(S'; \mathbf{Case}(z_1, S_1) \ldots \mathbf{Case}(z_n, S_n)) \quad S \vdash D :: S', S_i \quad 1 \leq z_i \leq n}{S \vdash \mathbf{tup}(D; z_i; A)) : A}$$

[Value tuple dot]

$$\frac{S \vdash a : \mathbf{Tup}(S', x : A, S''; C) \quad x \notin Dom(S'')}{S \vdash a.x : \mathbf{qualify}(A; a; S')}$$

[Value variant project]

$$\frac{S \vdash x : \mathbf{Tup}(S'; \mathbf{Case}(z_1, S_1) \ldots \mathbf{Case}(z_n, S_n))}{S \vdash x!z_i : \mathbf{Tup}(S', S_i; \mathbf{Case}(?, \oslash))}$$

[Value variant test]

$$\frac{S \vdash x : \mathbf{Tup}(S'; \mathbf{Case}(z_1, S_1) \ldots \mathbf{Case}(z_n, S_n))}{S \vdash x?z_i : \mathbf{Bool}}$$

[Value record]

$$\frac{S \vdash D :: S' \quad S \vdash S' \ rcdsig}{S \vdash \mathbf{rcd}(D) : \mathbf{Rcd}(S')}$$

[Value record dot]

$$\frac{S \vdash a : \mathbf{Rcd}(S', x : A, S'')}{S \vdash a.x : \mathbf{qualify}(A; a; S')}$$

[Value extend]

$$\frac{S \vdash a : \mathbf{Rcd}(S') \quad S, S' \vdash D :: S'' \quad S \vdash S', S'' \ rcdsig}{S \vdash \mathbf{extend}(a, D) : \mathbf{Rcd}(S', S'')}$$

[Value exception]

$$\frac{S \vdash a : \mathbf{String} \quad S \vdash D :: S'}{S \vdash \mathbf{exc}(a, D) : \mathbf{Exc}(S')}$$

[Value seq empty]

$$\frac{\vdash S \ sig}{S \vdash \mathbf{seq}(\oslash) : \mathbf{Ok}}$$

[Value seq ide]

$$\frac{S \vdash D :: S' \quad S, S' \vdash a : A}{S \vdash \mathbf{seq}(D, x = a) : A}$$

[Value seq ide restrict]

$$\frac{S \vdash D :: S' \quad S, S' \vdash a : A}{S \vdash \mathbf{seq}(D, x : A = a) : A}$$

[Value seq Ide]

$$\frac{S \vdash D :: S' \quad S, S' \vdash A \ type}{S \vdash \mathbf{seq}(D, X = A) : \mathbf{Ok}}$$

[Value seq Ide restrict]

$$\frac{S \vdash D \ :: \ S' \qquad S, S' \vdash B <: A}{S \vdash \mathbf{seq}(D, X <: A = B) : \mathbf{Ok}}$$

[Value if]

$$\frac{S \vdash a : \mathbf{Bool} \qquad S \vdash b : B \qquad S \vdash b' : B}{S \vdash \mathbf{if}(a, b, b') : B}$$

[Value case]

$$\frac{\begin{array}{c} S \vdash a : \mathbf{Tup}(S'; \mathbf{Case}(z'_1, S_1) \ldots \mathbf{Case}(z'_m, S_m)) \\ S \vdash b : B \\ S, y_i : \mathbf{Tup}(S', S''; \mathbf{Case}(?, \oslash)) \vdash b_i : B \quad 0 \leq n \quad i = 1 \ldots n \\ S, S' \vdash S'' \ <:: \ S_{z_{ij}} \quad j = 1 \ldots k_i \\ \bigcup_{i=1}^{n} \bigcup_{j=1}^{k_i} z_{ij} \subseteq \bigcup_{l=1}^{m} z'_l \quad z_{ij} \neq z_{i'j'} \ \ i \neq i', j \neq j' \end{array}}{S \vdash \mathbf{case}(a, (z_{11} \ldots z_{1k_1}, y_1, b_1) \ldots (z_{n1} \ldots z_{nk_n}, y_n, b_n), b) : B}$$

[Value case exhaustive]

$$\frac{\begin{array}{c} S \vdash a : \mathbf{Tup}(S'; \mathbf{Case}(z'_1, S_1) \ldots \mathbf{Case}(z'_m, S_m)) \\ S, y_i : \mathbf{Tup}(S', S''; \mathbf{Case}(?, \oslash)) \vdash b_i : B \quad 1 \leq n \quad i = 1 \ldots n \\ S, S' \vdash S'' \ <:: \ S_{z_{ij}} \quad j = 1 \ldots k_i \\ \bigcup_{i=1}^{n} \bigcup_{j=1}^{k_i} z_{ij} = \bigcup_{l=1}^{m} z'_l \quad z_{ij} \neq z_{i'j'} \ \ i \neq i', j \neq j' \end{array}}{S \vdash \mathbf{case}(a, (z_{11} \ldots z_{1k_1}, y_1, b_1) \ldots (z_{n1} \ldots z_{nk_n}, y_n, b_n)) : B}$$

[Value loop]

$$\frac{S \vdash a : \mathbf{Ok}}{S \vdash \mathbf{loop}(a) : \mathbf{Ok}}$$

[Value exit]

$$\frac{\vdash S \ \ sig}{S \vdash \mathbf{exit} : \mathbf{Nok}}$$

[Value while]

$$\frac{S \vdash a : \mathbf{Bool} \qquad S \vdash b : \mathbf{Ok}}{S \vdash \mathbf{while}(a, b) : \mathbf{Ok}}$$

[Value for]

$$\frac{S \vdash a : \mathbf{Int} \qquad S \vdash b : \mathbf{Int} \qquad S, x : \mathbf{Int} \vdash c : \mathbf{Ok}}{S \vdash \mathbf{for}(x, a, b, c) : \mathbf{Ok}}$$

[Value try]

$$\frac{\begin{array}{c} S \vdash a : B \\ S \vdash b : B \\ S \vdash a_i : \mathbf{Exc}(S_i) \quad i = 1 \ldots n \quad 0 \leq n \\ S, x_i : \mathbf{Tup}(S_i; \mathbf{Case}(?, \oslash)) \vdash b_i : B \quad i = 1 \ldots n \end{array}}{S \vdash \mathbf{try}(a, (a_1, x_1, b_1) \ldots (a_n, x_n, b_n), b) : B}$$

[Value raise]

$$\frac{S \vdash a : \mathbf{Exc}(S') \qquad S \vdash D \ :: \ S'}{S \vdash \mathbf{raise}(a, D) : \mathbf{Nok}}$$

[Value reraise]

$$\frac{\vdash S \ \ sig}{S \vdash \mathbf{reraise} : \mathbf{Nok}}$$

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

Abadi et al. *1989:* Abadi, M., Cardelli, L., Pierce, B. C., and Plotkin, G.D. Dynamic typing in a statically typed language. Digital Systems Research Center Reports 47, DEC SRC Palo Alto, Juni 1989.

Abadi et al. *1990:* Abadi, M., Cardelli, L., Curien, P.-L., and Lévy, J.-J. Explicit substitutions. Digital Systems Research Center Reports 54, DEC SRC Palo Alto, Februar 1990.

Abadi et al. *1992:* Abadi, M., Cardelli, L., Pierce, B., and Rémy, D. Dynamic typing in polymorphic languages. In *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications*, Juni 1992.

Atkinson and Morrison 1988: Atkinson, M.P. and Morrison, R. Types, bindings and parameters in a persistent environment. In Atkinson, M.P., Buneman, P., and Morrison, R., editors, *Data Types and Persistence*, Topics in Information Systems. Springer-Verlag, 1988.

Brown and Rosenberg 1991: Brown, A.L. and Rosenberg, J. Persistent object stores: An implementation technique. In *Proceedings of the Fourth International Workshop on Persistent Object Systems, Martha's Vineyard, Massachusetts.* Morgan Kaufmann Publishers, Januar 1991.

Burstall and Lampson 1984: Burstall, R. and Lampson, B. A kernel language for abstract data types and modules. In *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*. Springer-Verlag, 1984.

Cardelli and Mitchell 1989: Cardelli, L. and Mitchell, J.C. Operations on records. Digital Systems Research Center Reports 48, DEC SRC Palo Alto, August 1989.

Cardelli et al. *1991:* Cardelli, L., Martini, S., Mitchell, J.C., and Scedrov, A. An extension of system F with subtyping. In Ito, T. and Meyer, A.R., editors, *Theoretical Aspects of Computer Software, TACS'91*, Lecture Notes in Computer Science, pages 750–770. Springer-Verlag, 1991.

Cardelli 1989: Cardelli, L. Typeful programming. Digital Systems Research Center Reports 45, DEC SRC Palo Alto, May 1989.

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 1992a: Cardelli, L. Extensible records in a pure calculus of subtyping. Digital Systems Research Center Reports 81, DEC SRC Palo Alto, Januar 1992.

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).

Connor et al. *1991:* Connor, R., McNally, D., and Morrison, R. Subtyping and assignment in database programming languages. In *Database Programming Languages: Bulk Types and Persistent Data*, pages 363–382, Nafplion, Greece, 1991. Morgan Kaufmann Publishers.

de Bruijn 1972: de Bruijn, N.G. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.

Dearle et al. *1989:* Dearle, A., Connor, R., Brown, F., and Morrison, R. Napier88 – a database programming language? In *Proceedings of the Second International Workshop on Database Programming Languages, Salishan, Oregon*, Juni 1989.

MacQueen et al. *1986:* MacQueen, D.B., Plotkin, G.D., and Sethi, R. An ideal model for recursive polymorphic types. *Information and Control*, 71:95–130, 1986.

Matthes and Schmidt 1991a: Matthes, F. and Schmidt, J.W. Bulk types: Built-in or add-on? In *Proceedings of the Third International Workshop on Database Programming Languages, Nafplion, Greece*. Morgan Kaufmann Publishers, September 1991. (also appeared as TR FIDE/91/27).

Matthes and Schmidt 1991b: Matthes, F. and Schmidt, J.W. Towards database application systems: Types, kinds and other open invitations. In *Proceedings of the Kiev East/West Workshop on Next Generation Database Technology*, volume 504 of *Lecture Notes in Computer Science*, April 1991.

Matthes et al. *1991:* Matthes, F., Ohori, A., and Schmidt, J.W. Typing schemes for objects with locality. In *Proceedings of the Kiev East/West Workshop on Next Generation Database Technology*, volume 504 of *Lecture Notes in Computer Science*, April 1991. (also appeared as TR FIDE/91/12).

Matthes 1992: Matthes, F. *Generic Database Programming: A Linguistic and Architectural Framework*. PhD thesis, Fachbereich Informatik, Universität Hamburg, Germany, September 1992. (in German).

Morrison et al. *1987:* Morrison, R., Atkinson, M.P., and Dearle, A. Flexible incremental bindings in a persistent object store. Persistent Programming Research Report 38, Univ. of St. Andrews, Dept. of Comp. Science, Juni 1987.

Ohori et al. *1989:* Ohori, A., Buneman, P., and Breazu-Tannen, V. Database programming in Machiavelli – a polymorphic language with static type inference. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Portland, Oregon*, pages 46–57, 1989.

Ohori 1989: Ohori, A. *A Study of Semantics, Types and Languages for Databases and Object-Oriented Programming*. PhD thesis, University of Pennsylvania, 1989.

Pierce 1992: Pierce, B. C. Bounded quantification is undecidable. In *Proceedings of the Nineteenth ACM Symposium on Principles of Programming Languages*, pages 305–315, Januar 1992.

Rémy 1991: Rémy, D. Type inference for records in a natural extension of ml. Rapport de Recherche 1431, INRIA, Domaine de Voluceau Rocquencourt 78153 Le Chesnay Cedex - France, May 1991.

Schmidt and Matthes 1991: Schmidt, J.W. and Matthes, F. Naming schemes and name space management in the DBPL persistent storage system. In *Proceedings of the Fourth International Workshop on Persistent Object Systems, Martha's Vineyard, Massachusetts.* Morgan Kaufmann Publishers, Januar 1991.

Schröder and Matthes 1992: Schröder, G. and Matthes, F. Using the Tycoon compiler toolkit. DBIS Tycoon Report 061-92, Fachbereich Informatik, Universität Hamburg, Germany, May 1992.

Solomon 1978: Solomon, M. Type definitions with parameters. In *Proceedings of the Fifth ACM Symposium on Principles of Programming Languages, Tucson, Arizona*, pages 31–38, Januar 1978.

Stansifer 1988: Stansifer, R. Type inference with subtypes. In *Proc. 15th ACM Symposium on Principles of Programming Languages*, pages 88–97, 1988.

Wand 1987: Wand, M. Complete type inference for simple objects. In *Proceedings of the Second Annual Symposium on Logic in Computer Science*, pages 37–44, Ithaca, New York, Juni 1987.