Fachbereich Informatik
Universität Kaiserslautern
Postfach 3049
D-6750 Kaiserslautern

SEKI - REPORT

TEL (Version 0.9) Report and User Manual

Gert Smolka

SEKI Report SR-87-11

SEKI Report SR-87-11, Universität Kaiserslautern, West Germany, February 1988

TEL (Version 0.9)

Report and User Manual

Gert Smolka

FB Informatik, Universität Kaiserslautern
6750 Kaiserslautern, West Germany

smolka@uklirb.uucp

Abstract

TEL is a second generation logic programming language integrating types and functions with relational programming à la Prolog. Relations are defined as in Prolog and are executed by typed resolution and backtracking. Functions are defined with conditional equations and are executed by typed innermost rewriting.

The most innovative aspect of TEL is its type system, which accommodates parametric polymorphism as in ML and subtypes as in OBJ2. Variables need not be declared since TEL's type checker infers their most general types automatically. Types are present at runtime through typed matching and unification: values are tested for membership in subtypes and variables are constrained to subtypes.

TEL is not a toy language. Almost the entire TEL system has been written in TEL. TEL has a module facility supporting the incremental construction of large programs. Furthermore, TEL supports type-safe file handling and other extra-logical operations.

Acknowledgments: This research was funded by the Bundesminister für Forschung und Technologie under grant ITR8501A. Version 0.9 of TEL is being implemented by Costa Moissiadis, Werner Nutt, Reinhard Praeger, Ralf Scheidhauer, and Georg Seul. The first version of TEL was implemented by Michael Schmitt.

Contents

1 Introduction 1

2 Types 6

3 Functions 15

4 Built-in Types 20

5 Relations 26

6 Modules 36

7 Open Variables 53

8 Type Checking 63

9 Streams and Procedures 71

10 More on Conditions 79

11 Data Bases 84

Appendices

A Built-ins 86

B Syntax 94

C Manager Commands 103

D Limitations of the Current Implementation 107

1 Introduction

TEL, an acronym for types, equations and logic, is a second generation logic programming language. It is the practical outcome of a research effort aimed at the integration of types and functions with logic programming à la Prolog. Here are some highlights of TEL:

- TEL is a functional language. Functions are defined with conditional equations and are executed by innermost rewriting.
- TEL is a relational language. Relations are defined with Horn clauses and are executed by resolution and backtracking.
- Relations are declared with fixed input and output arguments, the consistent use of which is checked automatically at compile time. These data flow declarations provide for a simple and clean operational interaction between functions and relations.
- The data flow discipline can be weakened by declaring variables as open. Thus the full generality of logical variables in Prolog is available if needed.
- TEL is a typed language. It is the first language supporting both subtypes (as in OBJ2) and polymorphic type constructors (as in ML). Every well-typed term has a unique least type depending functionally on the types of the variables occurring in the term.
- TEL computes with types. Types are present at run-time through typed matching and unification: values are tested for membership in subtypes and open variables are constrained to subtypes.
- TEL has a module facility supporting the incremental construction of large programs. After the interface structure of a system has been fixed, every module can be compiled separately.
- TEL is a logic programming language. TEL's kernel language is based on a first-order, typed, definite clause logic with equality giving an initial algebra semantics to programs.

- TEL is a practical language. It supports type-safe file handling and other extra-logical operations. Almost the entire TEL system is written in TEL.

- TEL is an interactive language. The user enters queries, which are type checked, compiled, and executed. The results of a query are reported together with their least types.

Most of the theoretical and practical effort was devoted to the development of TEL's type system. So far TEL is the only language integrating parametric polymorphism à la ML [Harper et al. 86] with subtypes à la OBJ2 [Futatsugi 85]. This combination regains much of the flexibility of untyped languages such as Lisp and Prolog while providing the classical advantages of typed languages:

- The data structures used by a program can be defined explicitly. This leads to clearer, much easier to understand programs. The explicit definition of data structures is particularly beneficial if they are complex, as it is typically the case in Artificial Intelligence.

- Type checking detects many programming errors at compile time, a feature whose importance is proportional to the size of the program under development.

The presence of subtypes makes TEL's type system more than a syntactic discipline merely visible at compile time. TEL actually computes with types: at run time values are tested for membership in subtypes and variables are constrained to subtypes. Constraining variables to subtypes rather than binding them tentatively to particular elements (as in Prolog) avoids expensive backtracking.

The combination of parametric polymorphism with subtypes poses many interesting research problems: the design of a logic supporting these features, the development of the necessary type checking algorithms (which are nontrivial), and the development of an operational semantics having typed rewriting and unification as its major components. These problems are adressed in my thesis [Smolka 88], which provides the theoretical foundation for TEL. Another paper contributing to the theoretical foundation of TEL is [Smolka et al. 87], which studies computational aspects of an equational logic with subsorts.

The goal in designing TEL was to come up with a practical language that is a significant improvement over Prolog and can be implemented efficiently right now. The quest for practicability strongly constrained the design of TEL:

- Functions are executed by innermost rewriting rather than by the more general narrowing. If there is a need to solve for variables, this still can be done with relations. One advantage of executing functions with rewriting is that the programmer doesn't need to worry about their control. Furthermore, executing functions with innermost rewriting can compete with the efficiency of pure Lisp, which gains a magnitude in speed over current Prolog implementations.

- Relations must be declared with fixed input and output arguments. This ensures a clean operational interaction between functions and relations and is in accordance with common Prolog programming style. Having explicit data flow declarations and checking their consistent use at compile time contributes significantly to the clarity of programs. If the full generality of logical variables is needed, which is typically the case only at a few places in a large program, it can be obtained by bypassing the data flow discipline by declaring variables as open. While this approach is quite unsatisfactory from a theoretician's point of view, our programming experience in TEL suggests that it is very practical. One major use of logical variables is the implementation of open data structures, for instance, tables that are created incrementally at run time. In TEL open data structures can be implemented as abstract data types, thus making it possible to hide the use of open variables.

- Logic alone does not suffice for a practical programming language. Hence TEL has several extra-logical features including modules, control structures, stream-based file handling and data bases. All of TEL's extra-logical features are type safe.

To ensure TEL's practicability, I decided to implement TEL with a bootstrapped approach so that we could write most of the TEL system in TEL.

This still provides an excellent test case for TEL and many of the features of version 0.9 of TEL grew out of the experience made when implementing earlier versions of TEL.

At the time this report is published (February 1988), we have almost finished an implementation of version 0.9 of TEL [Nutt/Smolka 88]. This implementation runs on Quintus Prolog under UNIX 4.2 BSD on Apollo workstations and will be distributed freely including the program sources. Most of the implementation is written in TEL. The frontend of the compiler produces an intermediate language, which the backend translates to Quintus Prolog enhanced with a small run-time system. TEL programs that are comparable to Prolog programs run at the same speed as their Prolog equivalents.

Since the current implementation employs Prolog as the target language, the backend of the compiler is simple and we could concentrate on the frontend and the programming environment, which turned out to be complex due to type and module checking. To achieve a reasonably efficient execution, we were forced to map TEL's typed unification more or less directly to Prolog's untyped unification. Consequently, the current implementation cannot constrain open variables to subtypes. The solution to this problem will be the development of an abstract machine tailored to TEL's needs. The abstract machine will also allow for many optimizations exploiting the presence of functions and types that aren't possible at the level of Prolog.

Currently we are investigating several extensions that could be part of version 1.0 of TEL. Among them are feature types and inheritance hierarchies [Smolka/Aït-Kaci 87], which would provide record notation and feature unification. Another line of research tries to accommodate types as first-class citizens and to allow for dependent types. Finally, we would like to have the possibility to pass functions and relations as arguments.

This report describes version 0.9 of TEL from the viewpoint of a programmer who has programming experience in Prolog but is not necessarily interested in TEL's theoretical foundations. It is complemented by my thesis [Smolka 88], which provides the theoretical foundations and develops, in a more general setting, the employed type checking and unification algorithms.

References

K. Futatsugi, J.A. Goguen, J.-P. Jouannaud and J. Meseguer, Principles of OBJ2. POPL 1985, 52–66.

R. Harper, D. MacQueen, and R. Milner, Standard ML. Report ECS-LFCS-86-2, Edinburgh University, Scotland, March 1986.

W. Nutt and G. Smolka, Implementing TEL. SEKI Report, Universität Kaiserslautern, West Germany, 1988, forthcoming.

G. Smolka, Classified Logic: Semantics, Deduction, Type Checking and Computation. Dissertation, Universität Kaiserslautern, West Germany, 1988, forthcoming.

G. Smolka and H. Aït-Kaci, Inheritance Hierarchies: Semantics and Unification. To appear in Symbolic Computation, Special Issue on Unification Theory, 1988. Report AI-057-87, MCC, Austin, Texas, May 1987.

G. Smolka, W. Nutt, J.A. Goguen and J. Meseguer, Order-Sorted Equational Computation. Presented at the Colloquium on the Resolution of Equations in Algebraic Structures, Austin, Texas, May 1987. SEKI Report SR-87-14, Universität Kaiserslautern, West Germany, December 1987.

2 Types

About the most simple kind of type definition you can write in TEL is

```
    color := {red, blue, green}.
```

This definition introduces the *type constructor* `color` together with the three *value constructors* `red`, `blue` and `green`. The definition states that the type `color` has exactly three elements which are denoted by the given value constructors.

Definitions cannot be given directly to TEL but must be part of a module. After you have activated TEL, you can enter the command

```
    TEL> #edit_module(test).
```

and an editor window will pop up on the screen containing an empty module with the name `test`:

```
    module test.
    endmodule.
```

Now you can add definitions, for instance:

```
    module test.
    color := {red, blue, green}.
    endmodule.
```

After you have saved the editor window you can enter the command

```
    TEL> #open(test).
```

and TEL will type-check, compile, load, and open the module `test`. Now you can type the term

```
    TEL> red.
```

and TEL will respond

```
    red : color
```

which means that the term `red` reduces to the term `red` having the least type `color`. In TEL every well-typed ground term, that is, a well-typed term not containing variables, has a unique least type.

To obtain a type with infinitely many elements you need to use recursion. For instance, try something like

```
tree := {etree,
         netree: tree x  %left subtree
                 tree}.  %right subtree
```

The elements of `tree` are the closed terms that can be build with the value constructors `etree` and `netree`, where the two arguments of `netree` must be of type `tree`. The recursion comes in through the binary value constructor

```
netree: tree x tree --> tree.
```

After you have opened a module with the definition of `tree` you can type the term

```
TEL> netree(netree(etree, etree), etree).
```

and TEL will respond with:

```
netree(netree(etree, etree), etree) : tree
TEL>
```

An important feature of TEL is that types can be defined as the union of types. For instance, you can define binary trees also as

```
tree := empty_tree ++ nonempty_tree.
empty_tree := {etree}.
nonempty_tree := {netree: tree x tree}.
```

With this definition `empty_tree` and `nonempty_tree` are *subtypes* of `tree`. After you have opened a module containing this definition of `tree` you can type

```
TEL> etree.
```

and TEL will respond with

```
    etree : empty_tree.
```

If you type

```
    TEL> netree(netree(etree, etree), etree).
```

TEL will respond with:

```
    netree(netree(etree, etree), etree) : nonempty_tree.
```

The possibility to define a type as the union of subtypes contributes significantly to the expressive power of TEL's type system. You can mix subtype definitions with constructor definitions. For instance, you can write

```
    strange_type := color ++
                    tree ++
                    {other,
                     strange_tree: strange_type x strange_type}.
```

After you have opened a module containing the definition of strange_type you can type the following queries:

```
    TEL> red.
    red : color
    TEL> etree.
    etree:empty_tree
    TEL> strange_tree(blue, other).
    strange_tree(blue, other) : strange_type
```

The query

```
    TEL> red : strange_type.
    succeeded.
```

is a so-called *containment*, which tests whether the value of the term given at its left-hand side is an element of the type given at its right-hand side. If you type

```
red:tree.
```

TEL will respond with

```
*** type error in condition 1:
    least type of red is color;
    color and tree don't have a common subtype
```

since the type checker determines that the given containment cannot hold.

Trees labelled with colors can be defined as follows:

```
tree := empty_tree ++ nonempty_tree.
empty_tree := {etree}.
nonempty_tree := {netree: tree x tree x color}.
```

TEL also provides for polymorphic type definitions. The following is a polymorphic definition of labelled trees:

```
tree(T) := empty_tree ++ nonempty_tree(T).
empty_tree := {etree}.
nonempty_tree(T) := {netree: tree x tree x T}.
```

The letter T is a variable that ranges over types and parameterizes the definition of tree and non_emptytree with respect to the type of the labels. This polymorphic definition introduces infinitely many types, for instance,

```
tree(color), tree(tree(color)), tree(tree(tree(color))), ... .
```

After you have opened a module containing the polymorphic definition of labelled trees and the definition of `color`, you can type the following queries:

```
TEL> etree.
etree : empty_tree
TEL> netree(etree, etree, red).
netree(etree, etree, red) : nonempty_tree(color)
TEL> netree(etree, etree, etree).
netree(etree, etree, etree) : nonempty_tree(empty_tree)
TEL> netree(etree, etree, netree(etree, etree, red)).
netree(etree, etree, netree(etree, etree, red)) :
     nonempty_tree(nonempty_tree(color)).
```

The syntax of variables that range over types is the same as for variables that range over elements of types: they must start with a capital letter and can then continue with capital and small letters, digits, and the underline character '_' .

All type constructors you can define in TEL are *monotonic* with respect to the subtype order. For instance, `tree(empty_tree)` is a subtype of `tree(tree(color))` since `empty_tree` is a subtype of `tree(color)`. Furthermore, `empty_tree` is a subtype of `tree(tree(color))` since `empty_tree` is a subtype of `tree(`*t*`)` for every type *t*.

The following polymorphic definition of lists is built-in in TEL:

```
list(T) := elist ++ nelist(T).
elist := {nil}.
nelist(T) := {. : T x list(T)}.
```

For syntactical convenience, TEL treats the value constructor '`.`' as a right-associative infix operator. For instance, if you have opened module with the definition of `color`, you can type the query

```
TEL> red.blue.green.nil
```

and TEL will respond

```
red.blue.green.nil : nelist(color).
```

Pairs are another built-in polymorphic type of TEL:

```
L##R := {# : L x R}.
```

For syntactical convenience, TEL treats the binary type constructor '`##`' and the binary value constructor '`#`' as right-associative infix operators.

A type definition whose right-hand side consists of a single type term defines a *type abbreviation*. For instance,

```
assoc_list(Key,T) := list(Key##T).
```

introduces the type abbreviation `assoc_list`. You can now write `assoc_list(color,bool)` for `list(color##bool)`. Type abbreviations are syntactic sugar that is eliminated at compile-time.

Consider the type definition

```
ty(T) := {foo: list(T) x color}.
```

What do you think is the least type of `foo(nil,red)`? TEL will give you the answer if you open a module with the definition of `ty` and pose the query:

```
TEL> foo(nil,red).
foo(nil,red) : ty(void).
```

TEL solves the problem with the internal type `void`, which has no elements and is a subtype of every type. TEL won't allow you to explicitly use `void` in your programs.

By now you know TEL's basic machinery for type definitions. In later sections we will discuss a few further built-in types and TEL's facility for defining abstract types. In the rest of this section, we will state some restrictions that type definitions in TEL must observe.

The *closedness condition* applies to all definitions you can write in TEL and requires that in a module every occurring designator (a name for an object, for instance, a type or value constructor) must have one and only one definition. In particular, TEL will complain if you use the same name for a type and a constructor or if the same constructor occurs in the right-hand sides of two different type definitions.

The *minimality condition* requires that

1. the variables occurring in the left-hand side of a type definition must be pairwise distinct
2. every variable occurring in the left-hand side of a type definition must occur in the right-hand side of the type definition, and every variable occurring in the right-hand side of a type definition must occur in the left-hand side of the type definition.

The *completeness condition* requires that two types have a *greatest common subtype* if they have a common subtype. Thus TEL will complain if you write

```
tya := {a}.
tyb := {b}.
tyc := tya ++ tyb ++ {c}.
tyd := tya ++ tyb ++ {d}.
```

since `tyc` and `tyd` have `tya` and `tyb` as common subtypes but do not have a greatest common subtype. If you *complete* the above definition to

```
tya := {a}.
tyb := {b}.
tyab := tya ++ tyb.
tyc := tyab ++ {c}.
tyd := tyab ++ {d}.
```

TEL will be happy since now `tyab` is the greatest common subtype of `tyc` and `tyd`. Figure 2.1 gives a graphical representation of the two type hierarchies.

The *well-foundedness condition* requires that no type has infinitely many subtypes. In contrast to the preceding conditions, which are more or less of a

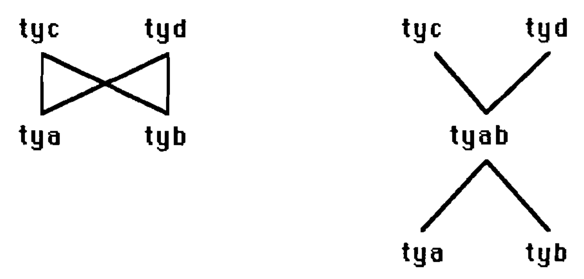

Figure 2.1. An incomplete type hierarchy and its completion.

cosmetic nature, this condition unfortunately excludes quite interesting type definitions. For instance, TEL will scream at you if you write

```
mylist(T) := T##mylist(T) ++ {mynil}.
```

since, for instance, `mylist(color)` has infinitely many subtypes:

`mylist(color)` $\geq$ `color##mylist(color)` $\geq \cdots$.

The well-foundedness condition is needed so that TEL's type checker and unification algorithm can work properly.

The *coherence condition* requires that two type terms are equal if their outermost type constructors are equal and they both can be reached by following subtype specifications starting from the righthand side of some type definition. For instance, TEL will complain if you write the definitions

```
tyc(T) := tya(color) ++ tyb(T).
tya(T) := list(T) ++ {a: T}.
tyb(T) := list(tree(T)) ++ {b: T}.
```

since, starting from `tyc(T)`, one can reach both `list(color)` and `list(tree(T))`:

```
tyc(T) ⇒ tya(color) ⇒ list(color)
tyc(T) ⇒ tyb(T) ⇒ list(tree(T)).
```

Like the well-foundedness condition, the coherence condition is needed so that TEL's type checker and unification algorithm can work properly.

This gives you a good idea of the restrictions type definitions in TEL must satisfy. All these restrictions are checked automatically by TEL.

In TEL it is possible to define types having no elements, for instance,

```
empty_type := {foo: empty_type x color}.
```

TEL checks for each type constructor whether it has elements and prints a warning if it discovers an empty type constructor. We will see later that empty types can make sense in conjunction with open data structures.

3 Functions

Functions in TEL are defined by conditional equations and are executed by typed rewriting. The following examples are functions for list processing, so you may want to look again at the built-in definition of lists:

```
list(T) := elist ++ nelist(T).
elist := {nil}.
nelist(T) := {.: T x list(T)}.
```

A function that appends two lists can be defined as follows:

```
app: list(T) x list(T) --> list(T).
   app(nil, L) = L.
   app(H.T, L) = H.app(T,L).
```

The definition consists of three sentences: a function declaration stating the types of the arguments and the result, and two equations defining app by induction on the list structure of the first argument. A *sentence* is a sequence of characters ending with a *full stop*, that is, a period followed by a layout character, for instance, a space or a newline character. The *scope of a variable* is always limited to the sentence in which it appears. Variables start with a capital letter and can continue with letters, digits or the underline character '_' . Thus T, L, H, and T are the variables that occur in the definition of app.

Since TEL derives the types of variables automatically, you don't have to declare variables. The types TEL derives for the variables in the second equation of app are:

```
TT:type, H:TT, T:list(TT), L:list(TT).
```

The type variable TT doesn't appear in the clause but is an auxiliary variable generated by TEL's type checker. Note tat the occurrence of T in the declaration of app is unrelated to the occurrence of T in the second equation of app, since the scope of a variable is always limited to the sentence in which it occurs.

TEL also supports a second, more compact syntax for function definitions:

```
app: list(T) x list(T) --> list(T).
   nil, L  |> L.
   H.T, L  |> H.app(T,L).
```

Since TEL has subtypes, it makes often sense to declare more than one rank for a function, for instance:

```
app: list(T) x list(T) --> list(T),
     nelist(T) x list(T) --> nelist(T),
     list(T) x nelist(T) --> nelist(T).
   nil, L  |> L.
   H.T, L  |> H.app(T,L).
```

With this definition you can use app(*s*,*t*) as an argument for a function that requires a nonempty list, provided *s* or *t* is a nonempty list. A fourth rank one could declare for app is

```
elist x elist --> elist
```

but this rank will be of little use in practice.

After you have opened a module containing the definitions of color and app, you can enter the query

```
app(red.blue.nil, green.blue.nil).
```

and TEL will respond:

```
red.blue.green.blue.nil : nelist(color).
```

The query is executed by rewriting the given term with the equations defining app, that is, by applying them from left to right:

```
app(red.blue.nil, green.blue.nil)
red.app(blue.nil, green.blue.nil)  %by the 2nd equation
red.blue.app(nil, green.blue.nil)  %by the 2nd equation
red.blue.green.blue.nil.           %by the 1st equation
```

Rewriting is done in an innermost order, that is, the arguments of a function are *reduced* or *evaluated* before the function is *applied*. TEL's well-typedness conditions ensure that rewriting never increases the least type of the term being rewritten.

Another list function is the membership test

```
member: T x list(T) --> bool
   _, nil   |> false.
   X, X._   |> true.
   X, Y.R   |> member(X,R)   <-- X \= Y.
```

where

```
bool := {true, false}.
```

is a built-in type of TEL. This example illustrates several further features of TEL. First, the underline character '_' can be used as a *wildcard variable*, that is, as a variable that occurs only once in a sentence. It is good style to use the underline character for every such variable. Second, the third equation of `member` is conditional. Its condition is the disequation `X\=Y`, which is satisfied if `X` and `Y` are different. Finally, the left-hand sides of the equations defining a function need not be linear—for instance, `X` appears twice in the second equation of `member`.

When a function is executed, its equations are considered in top down order. An equation applies if its left-hand side matches and its conditions are satisfied. The first equation that applies determines the result of the function. Note that `member` is defined such that always exactly one equation applies. Since the equations are tried in top down order, `member` could also be written as

```
member: T x list(T) --> bool
   _, nil   |> false.
   X, X._   |> true.
   X, Y.R   |> member(X,R).  %<-- X \= Y.
```

where the operationally redundant test `X \= Y` is omitted. The drawback of this optimization is that the declarative semantics of the definition of `member` is not correct anymore, that is, we have traded clarity for efficiency. It is good style to list the conditions that are optimized away as comments.

The following defines a function computing the list of all sublists of a list:

```
powerlist: list(T) --> list(list(T)).
   nil  |> nil.nil.
   H.T  |> app(listcons(H,PL), PL)  <-- PL = powerlist(T).


listcons: T x list(list(T)) --> list(list(T)).
   _, nil   |> nil.
   X, H.T   |> (X.H).listcons(X,T).
```

The second clause of `powerlist` shows that you can introduce new variables in the condition part of a clause by binding them at the left-hand side of an equation.

The canonical example of a recursive function is the factorial function for the natural numbers. Since integers are built-in in TEL, one possibility is:

```
fac: nat --> posint.
   0  |> 1.
   N  |> N*fac(N-1)  <-- N>0.
```

Another possibility is:

```
fac: nat --> posint.
   0  |> 1.
   N  |> N*fac(N-1)  <-- N:posint.
```

Here the condition of the second clause is the containment `N:posint`, which is satisfied if `N` is a positive integer. The type `posint` is a built-in subtype of `nat`, which in turn is a built-in subtype of `int`.

In TEL a function must have at least one argument. Constants can be defined as so-called *parameters*, for instance:

```
par length : nat = 22.
par width : nat = 56.
par area : nat = length*width.
```

The value of a parameter is computed exactly once when the module in which the parameter is defined is loaded.

The left-hand side of an equation defining a function f must have the form $f(s_1, \ldots, s_n)$, where the formal arguments $s_1, \ldots, s_n$ must be *canonical terms*, that is, terms that only consist of variables and value constructors. For the |>-syntax this means that every term that appears left from the |>-symbol must be canonical.

For every well-typed tuple of arguments, at least one of the equations defining a function should apply. Since TEL allows for conditional equations, this property is undecidable. If at run-time a situation occurs in which no equation of a function applies, TEL will print an error message and abort execution.

4 Built-in Types

This section presents most of TEL's built-in types.

4.1 Booleans

The following type definition is built-in:

```
bool := {true, false}.
```

Furthermore, the following boolean connectives are built-in:

```
and: bool x bool --> bool.  %and is a right-associative
   true, true  |> true.     %infix operator
   false, _    |> false.
   _, false    |> false.


or: bool x bool --> bool.   %or is a right-associative
   false, false  |> false.  %infix operator
   true, _       |> true.
   _, true       |> true.

not: bool --> bool.  %not is a prefix operator
   true   |> false.
   false  |> true.
```

4.2 Integers

Integers are built-in as follows:

```
int := negint ++ nat.
nat := zero ++ posint.
negint := {~1, ~2, ~3, ... }.
zero := {0}.
posint := {1, 2, 3, ... }.
par minnegint : negint = <implementation dependent>.
par maxposint : posint = <implementation dependent>.
```

The following arithmetic functions are built-in:

```
+ : int x int --> int,  % + is a right-associative
    nat x nat --> nat,  % infix operator
    posint x nat --> posint,
    nat x posint --> posint,
    negint x negint --> negint.

- : int x int --> int,        % - is a left-associative
    nat x negint --> posint,  % infix operator
    negint x nat --> negint.

~ : int --> int,  %unary minus,  ~ is a prefix operator
    posint --> negint,
    negint --> posint.

* : int x int --> int,  % * is a right-associative
    nat x nat --> nat,  % infix operator
    posint x posint --> posint,
    posint x negint --> negint,
    negint x posint --> negint,
    negint x negint --> posint.

mod: int x int --> nat.  % mod is an infix operator

// : int x int >-> int,  % // is an infix operator
     nat x nat >-> nat,
     posint x posint --> posint,
     posint x negint --> negint,
     negint x posint --> negint,
     negint x negint --> posint.
```

The first two ranks of the integer division function '//' are partial since division by zero is undefined. You should use partial ranks for all functions that are not defined for all arguments. Operationally, it makes no difference whether you use total or partial ranks, but the correct use of partial ranks makes it easier to understand your programs.

The usual comparisons for integers are built-in:

```
< : int x int --> bool.  % < is an infix operator
=< : int x int --> bool.  % =< is an infix operator
> : int x int --> bool.  % > is an infix operator
>= : int x int --> bool.  % >= is an infix operator
```

4.3 Characters

Characters are built-in as follows:

```
char := layout_char ++ alpha_char ++ symbol_char.
alpha_char := letter ++ digit ++ {"_"}.
letter := capital_letter ++ small_letter.
symbol_char := grouping_symbol ++ operator_symbol ++ {"%"}.

layout_char := {"bell", "eof", "nl",
                 " any character with ASCII-code less than 33"}.

capital_letter := {"A", "B", ... , "Z"}.
small_letter := {"a", "b", ... , "z"}.
digit := {"0", "1", ... , "9"}.

grouping_symbol := {"(", ")", "[", "]", "{", "}",
                    """", "'", ","}.

operator_symbol := {"+", "-", "*", "/", "|", "\", "^",
                    "<", ">", "`", "~", "=", ":", ".",
                    "?", "@", "#", "$", "&", "!", ";"}.
```

Every character has a natural number equivalent:

```
natequiv: char --> nat.
charequiv: nat >-> char.
```

4.4 Lists

The following definition of lists is built-in:

```
list(T) := elist ++ nelist(T).
elist := {nil}.
nelist(T) := {. : T x list(T)}.
```

Furthermore, a few list functions are built-in:

```
| : list(T) x list(T) --> list(T),      % | is a right-assoc.
    nelist(T) x list(T) --> nelist(T),  % infix operator
    list(T) x nelist(T) --> nelist(T).
   nil, L  |> L.
   H.T, L  |> H.T|L.


in: T x list(T) --> bool.  % in is an infix operator
   _, nil  |> false.
   X, X._  |> true.
   X, Y.T  |> X in T.  %<-- X \= Y.


length: list(T) --> nat,
        nelist(T) --> posint.
   nil  |> 0.
   _.T  |> 1 + length(T).
```

4.5 Pairs

Pairs are built-in as follows:

```
S##T := {# : S x T}.  % ## and # are right-associative
                      % infix operators
```

4.6 Strings

Strings are built-in as follows:

```
string := estring ++ nestring.
estring := {''}.
nestring := a nonempty string starts with ', continues with at least one character,
        where ' is written as '', and ends with '
```

Strings are ordered lexiographically and the following comparisons are built-in:

```
@< : string x string --> bool.  % @< is an infix operator
@=< : string x string --> bool.  % @=< is an infix operator
@> : string x string --> bool.  % @> is an infix operator
@>= : string x string --> bool.  % @>= is an infix operator
```

Strings can be converted to character lists and character lists can be converted to strings:

```
chartrans: string --> list(char),
           nestring --> nelist(char).


stringtrans: list(char) --> string,
             nelist(char) --> nestring.
```

A function that concatenates two strings is built-in:

```
^ : string x string --> string.
   S1, S2  |> stringtrans(chartrans(S1)|chartrans(S2)).
```

Furthermore, a function that converts natural numbers into strings is built-in:

```
genstring: string x nat --> nestring.
   S, N  |> S^stringtrans(genstring1(N,nil)).
```

The auxiliary functions genstring1 and gen_equiv are not built-in, but they are included here to give two more examples for functional programming in TEL.

```
    genstring1: nat x list(char) --> list(char).
       N, L  |> gen_equiv(N).L  <-- N < 10.
       N, L  |> genstring1(N//10, gen_equiv(N mod 10).L).
                      %<-- N >= 10.


    gen_equiv: nat >-> char.  %only defined for 0..9
       0  |> "0".
       1  |> "1".
       2  |> "2".
       3  |> "3".
       4  |> "4".
       5  |> "5".
       6  |> "6".
       7  |> "7".
       8  |> "8".
       9  |> "9".
```

5 Relations

As in Prolog, relations in TEL are defined by a sequence of Horn clauses, which are logical implications of the form

$$P \leftarrow C_1 \ \& \ \ldots \ \& \ C_n$$

and are read: P holds if $C_1, \ldots, C_n$ hold. Since TEL is typed, you must declare a type for every argument of a relation. Furthermore, you must declare for every argument of a relation whether, operationally, it is used as an input or an output argument. Compared to Prolog, this data flow declarations certainly restrict the things one can do with relations, but, on the other hand, they make it easier to understand the operational semantics of a program. Furthermore, data flow declarations are needed for a clean and simple integration of the functional and relational parts of the language. In a later section we will discuss so-called open variables, which provide a means to bypass data flow declarations and thus allow to regain the full power of Prolog if it is actually needed.

A simple example is the definition of a membership relation for lists:

```
rel member: ?T x list(T).
   member(X, X._).
   member(X, _.T)  <-- member(X,T).
```

This definition can be read as follows: `X` is a member of a list whose head is `X`, and `X` is a member of a list if it is a member of its tail. The first argument of `member` is declared as an output argument and the second argument is declared as an input argument. When a relational condition is executed, the terms appearing as input arguments must be ground, that is, must not contain variables. After a relational condition is executed, the terms appearing as output arguments will be ground. TEL's type checker ensures that you can define and use relations only in such a way that these conditions are always satisfied at run-time.

If you have opened a module containing the definition of `member`, you can type the following query:

```
TEL> member(X, 1.2.3.nil).
```

Tel will compute the first solution for the variable X and respond with:

```
X = 1 : posint
more answers? (y/n).
```

If you now type 'n' , TEL will be ready for the next query. However, if you type 'y' , TEL computes a further answer to your query:

```
X = 2 : posint
more answers? (y/n) y
X = 3 : posint
more answers? (y/n) y
failed.
```

Here are two further queries:

```
TEL> member(2, 1.2.3.nil).
succeeded
TEL> member(4, 1.2.3.nil)
failed.
```

TEL won't accept the query

```
TEL> member(X, 1.2.3.Y).
```

since the second argument of `member` is an input argument and thus must not contain an unbound variable. TEL will respond with the error message:

```
*** mode error in condition 1:
    second argument 1.2.3.Y of member is declared input;
    Y is not bound.
```

In TEL relations are executed as in Prolog. To execute a relational condition $r(s_1, \ldots, s_n)$, the clauses defining r are tried in top down order. A clause applies if its head unifies with $r(s_1, \ldots, s_n)$ and all its conditions, which are executed from left to right, succeed. If a clause applies, the unification with its head and the execution of its conditions will bind all variables occurring in the output arguments of $r(s_1, \ldots, s_n)$ to ground terms. If no clause of r applies, the execution of $r(s_1, \ldots, s_n)$ fails. In contrast to functional conditions, a relational condition can be reactivated through backtracking and can thus produce more than one set of bindings for the variables occurring in its arguments. I won't offer more information on the execution of relations since careful explanations can be found in textbooks on Prolog and logic programming.

5.1 Example: A Tautology Checker

This example shows how one can implement a tautology checker for propositional formulas in TEL. It illustrates determinate relations, negation as failure, and the combination of relations and functions.

Propositional formulas are defined as follows:

```
propform := bool ++ propvar ++
            {a: propform x propform,  % and connective
             o: propform x propform,  % or connective
             n: propform}.            % not connective


propvar := {v: string}.  % propositional variable
```

Furthermore, we need assignments that assign truth values to propositional variables:

```
assignment := list(propvar##bool).
```

Note that `assignment` is not a type constructor but a type abbreviation for the type term `list(propvar##bool)`. The definition of `assignment` reveals a weakness that TEL shares with other typed programming languages: the type definition cannot express the requirement that an assignment should assign only one truth value to a propositional variable. All we can do is to define a test that checks whether an assignment is consistent:

```
consistent: assignment --> bool.
   nil   |> true.
   H.A   |> consistent1(H,A) and consistent(A).


consistent1: propvar##bool x assignment --> bool.
   _, nil          |> true.
   V#_, V#_.A      |> false.
   V1#B1, V2#_.A   |> consistent1(V1#B1, A).  % <-- V1 \= V2.
```

Next we define a relation `truthvalue(F, A, CA, B)` that holds if `A` can be extended to `CA` such that `F` has the truthvalue `B` under `CA`:

```
rel truthvalue: propform x
                assignment x
                ?assignment x  % extended assignment
                ?bool.  % truth value under extended ass.


   truthvalue(B, A, A, B)  <-- B:bool.


   truthvalue(V, A, CA, B)  <-- V:propvar &
         extends(V, A, CA, B).


   truthvalue(a(F1,F2), A, CA, B)  <--
         truthvalue(F1, A, CA1, B1) &
         truthvalue(F2, CA1, CA, B2) &
         B = (B1 and B2).


   truthvalue(o(F1,F2), A, CA, B)  <--
         truthvalue(F1, A, CA1, B1) &
         truthvalue(F2, CA1, CA, B2) &
         B = (B1 or B2).
   truthvalue(n(F), A, CA, B)  <--
         truthvalue(F, A, CA, B1) &
         B = not B1.


rel extends: propvar x assignment x ?assignment x ?bool.
   extends(V, nil, V#true, true).
   extends(V, nil, V#false, false).
   extends(V, A, A, B)  <-- V#B._ = A.
   extends(V, W#BW.A, W#BW.CA, B)  <-- V \= W &
         extends(V, A, CA, B).
```

If you have opened a module containing these definitions, you can type the query:

```
TEL> F = a(v('x'), v('y')) &
     A = v('x')#true .nil &
     truthvalue(F, A, CA, B).
CA = v('x')#true .v('y')#true .nil : nelist(propvar##bool)
B = true : bool
more answers? (y/n) y
CA = v('x')#true .v('y')#false .nil : nelist(propvar##bool)
B = false : bool
more answers? (y/n) y
failed.
```

We can now define a relation that holds if its argument is a satisfiable propositional formula:

```
rel satisfiable: propform.
satisfiable(F)  <-- truthvalue(F, nil, _, true).
```

A relation is called *determinate* if it produces at most one collection of output arguments for any well-typed collection of input arguments. Since `satisfiable` has no output arguments, it is necessarily determinate. In TEL you can declare relations to be determinate by using drel instead of rel. Thus

```
drel satisfiable: propform.
satisfiable(F)  <-- truthvalue(F, nil, _, true).
```

is a better definition of satisfiable since it makes explicit that satisfiable is determinate. Furthermore, declaring satisfiable as determinate will speed up the execution of your program, since it prevents backtracking the execution of satisfiable, which would unnecessarily force truthvalue to search for a further solution.

From what I have said it is clear that you can force relations to be determinate by declaring them determinate. This use of a drel-declaration corresponds to a weak form of Prolog's cut.

A boolean test for satisfiability can be defined as follows:

```
issatifiable: propform --> bool.
   F  |> true  <-- truthvalue(F, nil, _, true).
   F  |> false  <-- naf truthvalue(F, nil, _, true).
```

This example illustrates that TEL offers the possibility to negate relational conditions. This is done by the reserved identifier naf, which stands for negation as failure. A condition naf *C* succeeds if *C* fails and fails if *C* succeeds. As in Prolog, TEL's negation as failure is in general not logical negation.

You can make the following optimization without changing the operational semantics of issatisfiable:

```
issatifiable: propform --> bool.
   F  |> true  <-- truthvalue(F, nil, _, true).
   F  |> false.  % <-- naf truthvalue(F, nil, _, true).
```

Finally, you can write a tautology test as follows:

```
istautology: propform --> bool.
   F  |> not issatisfiable(n(F)).
```

5.2 Example: A Precedence Parser

This example implements a precedence parser for expressions built from integers and prefix and infix operators. Operators and precedences are defined as follows:

```
operator := prefix_operator ++ postfix_operator.


prefix_operator := {preop: string x    % operator name
                        precedence x   % prec. of operator
                        precedence}.   % max. prec. of arg.
%the argument precedence must be =< the operator precedence


infix_operator := {inop: string x     % operator name
                       precedence x   % prec. of operator
                       precedence x   % max. prec. left arg.
                       precedence}.   % max. prec. right arg.
%the argument precedences must be =< the operator precedence


precedence := nat.  % must be =< maxprecedence
par maxprecedence : precedence = 10000.
```

The parser translates lists of operators and integers to groups, which are defined as follows:

```
group := int ++
         {pgroup: prefix_operator x group,
          igroup: infix_operator x group x  % left arg.
                                   group}.  % right arg.


op_or_int := operator ++ int.
op_or_group := op_or_int ++ group.
group_or_error := group ++ {error}.
```

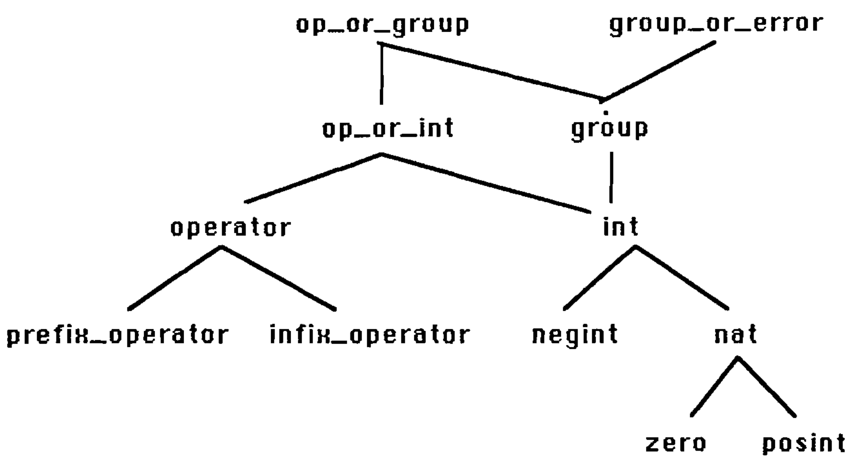

Figure 5.1. The subtype hierarchy of the precedence parser example.

These type definitions provide a nice example for a nontrivial subtype hierarchy, which is shown graphically in Figure 5.1. Note that op_or_int and group have int as greatest common subtype. Next we define a function that yields the precedence of operators and groups:

```
pre: op_or_group --> nat.
   preop(_,P,_)    |> P.
   inop(_,P,_,_)   |> P.
   I               |> 0  <-- I:int.
   pgroup(O,_)     |> pre(O).
   igroup(O,_,_)   |> pre(O).
```

Now we are ready to define the parser:

```
parse: list(op_or_int) --> group_or_error.
   L  |> G  <-- parse1(L, maxprecedence, G, nil).
   L  |> error.  % <-- naf parse1(L, maxprecedence, _, nil).
```

```
drel parse1: list(op_or_group) x
             precedence x  % current precedence
             ?group x
             ?list(op_or_group).  % unparsed tokens


  parse1(G.nil, P, G, nil)  <-- G:group & P >= pre(G).


  parse1(O.R, P, G, R2)  <--
        preop(_,OP,AP) = O &
        P >= OP &
        parse1(R, AP, G1, R1) &
        parse1(pgroup(O,G1).R1, P, G, R2).


  parse1(G.O.R, P, G2, R2)  <--
        G:group &
        inop(_,OP,LP,RP) = O &
        OP =< P & pre(G) =< LP &
        parse1(R, RP, G1, R1) &
        parse1(igroup(O,G,G1).R1, P, G2, R2).


  parse1(G.O.R, P, G, O.R)  <--
        G:group & O:infix_operator &
        P < pre(O) & pre(G) =< P.
```

If you have opened a module with these definitions, you can enter the term

```
TEL> parse(4
            .inop('-',5,5,4)
            .preop('~',2,1)
            .5
            .inop('-',5,5,4)
            .7 .nil).          % 4 - ~5 - 7
```

and TEL will respond with

```
= igroup(inop('-',5,5,4),
         igroup(inop('-',5,5,4),
                4,
                pgroup(preop('~',2,1), 5)),
         7).
```

5.3 Total Relations

Every function can be formulated as a determinate relation by transferring the result of the function through an output argument. Usually you won't want to do this since functional notation allows for nesting. However, relational notation can be convenient for functions whose result is a compound term that must be decomposed for further processing. For instance,

```
foo: nat --> nat##nat.
   N   |> N//5 # N mod 5.
```

can be written as the total determinate relation

```
tdrel roo: nat x ?nat x ?nat.
   roo(N, D, M)  <-- D = N//5 & M = N mod 5.
```

A relation is *total*, if it produces at least one output for every well-typed input. If you know that a relation is total, you should make this explicit by declaring it as `trel` or, if you also know that it is determinate, as `tdrel`. If the execution of a relation declared as total fails to produce at least one output, TEL prints an error message and aborts execution. These error messages are helpful during debugging.

If you want to write programs that are executed very efficiently, you should know that the condition `roo(N,D,M)` is executed more efficiently than `D#M = foo(N)` since in the functional case TEL has to construct a pair and then to decompose it again, which costs time as well as memory.

6 Modules

TEL has a simple, nonparametric module facility supporting the incremental construction of large systems. The module facility provides for information hiding, abstract data types and separate compilation. A module consists of an *interface* defining which modules are imported and which objects are exported, and a *body* implementing the exported objects that aren't transferred from imported modules. Modules must be organized hierarchically, that is, a module cannot be imported by one of its submodules. To implement a system, one starts by writing interfaces and continues by implementing the corresponding bodies. After a hierarchy of interfaces has been compiled, the corresponding bodies can be compiled separately.

6.1 An Example

Figure 6.1 shows the simplified module structure of a compiler. The module `abs_syntax_and_table` defines the abstract syntax and the definition table of the compiler. To make things easy, we assume that the definition table need not be extended during compilation.

```
interface abs_syntax_and_table.
   entry := abstract.
   name: entry --> string.
   address: entry --> nat.
   drel entryof: string x ?entry.
   term := {ter: entry x list(term)}.
endinterface.
```

Note that `entry` is exported as an abstract type, that is, an importing module does not know which constructors and subtypes `entry` has. For every abstract type equality and containment (for instance, `X:entry`) are available.

Views are special modules that don't have a body. The view `frontend_import` defines which of the objects exported by the module

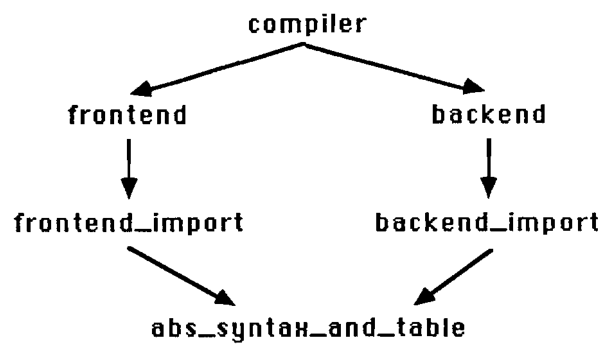

Figure 6.1. A module hierarchy.

abs_syntax_and_table can be seen by the front end of the compiler.

```
view frontend_import.
   imports abs_syntax_and_table.
   from abs_syntax_and_table: entry, entryof, term.
endview.
```

The relation entryof is used to obtain the entry of an identifier. Since it is a relation that can fail, entryof can also be used to check whether an identifier is defined in the table.

The view backend_import defines which of the objects exported by abs_syntax_and_table can be seen by the back end of the compiler.

```
view backend_import.
   imports abs_syntax_and_table.
   from abs_syntax_and_table: entry, name, address, term.
endview.
```

Now we are ready for the interface of the front end:

```
interface frontend.
   imports frontend_import.
   from abs_syntax_and_table: term abstract.
   error := abstract.
   term_or_error := term ++ error.
   parse: list(char) --> term_or_error.
   error_msg: error --> string.
endinterface.
```

Since the top module of the compiler is not supposed to inspect the details of the parser output, `term` is transferred as an abstract type although it is imported as a nonabstract type from `frontend_import`. Note that the transfer declaration

```
from abs_syntax_and_table: term abstract.
```

gives the name of the module where `term` is actually defined and not the name of the module from which `term` is imported.

The interface of the back end is

```
interface backend.
   imports backend_import.
   from abs_syntax_and_table: term abstract.
   code: term --> list(char).
endinterface.
```

and the interface of the top module of the compiler is

```
interface compiler.
   imports frontend, backend.
endinterface.
```

Note that the type `term` is imported twice, once from `frontend` and once from `backend`. This is the so-called *sharing problem*. To make sure that the multiple import of an identifier is okay, TEL must find out in which module an identifier is actually defined. A multiple import of an identifier is okay if all imports refer to the same module.

To compile all these interfaces it suffices to type

```
TEL> compile_interface(compiler).
```

After you have compiled the interfaces you can compile and recompile the module bodies in any order you like. However, before you can open a module, all its submodules must have been compiled.

The body of the top module could be defined as follows:

```
module compiler.
   output := {error_str: string,
              code_list: list(char)}.
   compile: list(char) --> output.
      L |> compile1(parse(L)).
   compile1: term_or_error --> output.
      T |> code_list(code(T))  <-- T:term.
      E |> error_str(error_msg(E))  <-- E:error.
endmodule.
```

The function `compile1` uses containments for the abstract types `term` and `error` to find out whether the parser has detected an error.

6.2 Signatures

In a TEL module the following objects can exist:

- *type constructors*, for instance, `bool` or `list`
- type abbreviations
- value constructors, for instance, `true`, `#`, 63, `'a string'`, or `"eof"`
- parameters, for instance, `maxposint` or `minnegint`

- functions, for instance, `+`, `mod`, or `natequiv`
- relations
- procedures, which will be introduced in a later section.

For the sake of a short name, the term function is used in TEL only for functions that, more exactly, might be called extending value functions. Mathematically, type constructors, value constructors and parameters are functions as well.

Every object has a name. Objects that can be defined in TEL must be named by so-called *designators*, which are either identifiers or operators. The syntactic details of identifiers and operators are spelled out in Appendix B.

Objects that are not built-in are introduced by definitions. Part of a definition is a declaration, which states the kind of the object, fixes the designator that names the object, and possibly states type and data flow information. TEL allows for the following declarations:

declaration $\longrightarrow$
  type_declaration
  | *parameter_declaration*
  | *function_declaration*
  | *relation_declaration*
  | *procedure_declaration*.

Declarations of value constructors appear as part of type declarations.

The syntax of declarations is as follows:

type_declaration $\longrightarrow$
  abstract_type_declaration
  | *type_abbreviation*
  | *type_definition*

abstract_type_declaration $\longrightarrow$
  type_dec_lhs ‘`:=`’ ‘`abstract`’ ‘`.`’

type_dec_lhs ⟶

identifier ['(' { *variable* } ')']

| *prefix_operator variable*

| *variable infix_operator variable*

| *variable postfix_operator*

the occurring variables must be pairwise distinct

type_abbreviation ⟶

type_dec_lhs ':=' *nonvariable_type_term*

every variable that occurs in the left-hand side must occur in the right-hand side and vice versa

type_definition ⟶

type_dec_lhs ':=' *type_def_rhs* '.'

every variable that occurs in the left-hand side must occur in the right-hand side and vice versa

type_def_rhs ⟶

(*subtype_specification* '++')*

subtype_specification '++'

subtype_specification

| (*subtype_specification* '++')*

'{' { *constructor_definition* } '}'

subtype_specification ⟶

nonvariable_type_term

constructor_definition ⟶

designator [':' *domain*]

domain ⟶

type_term ['x' *domain*]

designator ⟶

identifier | *operator*

A nonabstract type declaration is called a definition since it completely defines the declared type constructor. Note that the right-hand side of a type definition contains further declarations, namely, constructor declarations. Analogous to type declarations, constructor declarations are called definitions since they define the declared constructor completely.

parameter_declaration $\longrightarrow$
  'par' *identifier* ':' *ground_type_term* '.'

function_declaration $\longrightarrow$
  designator ':' { *rank* } '.'
    all ranks must specify the same number of arguments

rank $\longrightarrow$
  domain '-->' *type_term*
  | *domain* '>->' *type_term*
    every variable occurring in the codomain of a rank must occur in the domain of the rank

relation_declaration $\longrightarrow$
  rel_class designator ':' *io_domain* '.'

rel_class $\longrightarrow$
  'tdrel' | 'drel' | 'trel' | 'rel'

io_domain $\longrightarrow$
  ['?'] *type_term* ['x' *io_domain*]
    every variable occurring in the type term of an output argument must occur in the type term of an input argument

procedure_declaration $\longrightarrow$
  proc_class designator ':' [*io_domain*] '.'

proc_class $\longrightarrow$
  'tproc' | 'proc'

A *signature* is a set of declarations containing the declarations of all built-in objects, which are listed in Appendix A.

Given a signature, one can built two kinds of terms—type terms and value terms. *Type terms* are terms that are built from type constructors and variables. *Types* are type terms that do not contain variables. *Value terms* are terms that are built from value constructors, functions, parameters, and variables. *Values* are value terms that are built from value constructors only, that is, do not contain functions, parameters, or variables. The built-in values of `integer`, `char` and `string` are nullary value constructors.

Every module comes with three signatures:

- an *export signature* defined by the interface of the module
- an *import signature* defined by the interface of the module
- a *local signature* defined by the body of the module.

Since views don't have a body, they have only an import and an export signature.

There are several consistency requirements for the signatures of modules and views, which are checked automatically. To define these requirements, we need several technical definitions. This definitions make sense only with respect to a given signature.

We write $s \Rightarrow t$ (read: s is *directly outermost above* t) if s and t are type terms and the pair (s,t) is an instance of a pair (u,v), where the signature contains a type definition whose left-hand side is u and whose right-hand side contains v as a subtype specification.

We write $s \Rightarrow^* t$ (read: s is *outermost above* t) if t is a type term and there exist $n > 0$ type terms $s_1, \ldots, s_n$ such that

$$s = s_1 \Rightarrow s_2 \Rightarrow \cdots \Rightarrow s_n = t.$$

Given the signature that belongs to the subtype hierarchy in Figure 5.1, we have, for instance, `int`$\Rightarrow^*$`int`, `group`$\Rightarrow$`int`, `group`$\Rightarrow^*$`posint`, `list(group)`$\Rightarrow$`nelist(group)`, and `list(group)`$\Rightarrow$`elist`.

We say that a type constructor f is a *subconstructor* of a type constructor g, if there exist terms $s_1, \ldots, s_m$ and variables $x_1, \ldots, x_n$ such that $g(x_1, \ldots, x_n) \Rightarrow^* f(s_1, \ldots, s_m)$ and $g(x_1, \ldots, x_n)$ is the left-hand side of a type definition. We say that a type constructor f is a *superconstructor* of a type constructor g if g is a subconstructor of f.

We write $s \longrightarrow t$ (read: s is directly above t) if t is a type term and s can be obtained from t by replacing a subterm u with v, where $u \Rightarrow v$. We write $s \geq t$ (read: s is above t) if s and t are type terms and there exist $n > 0$ type terms $s_1, \ldots, s_n$ such that

$$s = s_1 \longrightarrow s_2 \longrightarrow \cdots \longrightarrow s_n = t.$$

We say that a type s is a *subtype* of a type t if t is above s. We say that a type s is a *supertype* of a type t if t is a subtype of s. Every type is a subtype and a supertype of itself.

The *infimum* $s \sqcap t$ of two type terms s and t is the greatest type term u such that $s \geq u$ and $t \geq u$. The *supremum* $s \sqcup t$ of two type terms s and t is the least type term u such that $u \geq s$ and $u \geq t$. The consistency requirements we will discuss below ensure that $\geq$ is a partial order on type terms and that $s \sqcap t$ [$s \sqcup t$] exist if s and t have a common lower [upper] bound.

Most of the following consistency requirements for signatures were already discussed informally in Section 2, which also gives counterexamples.

A signature is *closed*, if every designator occurring in it has one and only one declaration, and every term occurring in one of its declarations as a type term is in fact a type term.

A signature is *well-founded*, if there are no infinite chains

$$s \longrightarrow s_1 \longrightarrow s_2 \longrightarrow s_3 \longrightarrow \cdots$$

issuing from the left-hand side s of a type definition. In a well-founded signature, every type has only finitely many subtypes.

A signature is *coherent* if for every left-hand side u of a type definition and every two type terms $f(s_1, \ldots, s_m)$ and $g(t_1, \ldots, t_n)$ that are outermost below u we have $f(s_1, \ldots, s_m) = g(t_1, \ldots, t_n)$ if $f = g$.

A signature is *complete* if every two type constructors that have a common subconstructor have a greatest common subconstructor.

A function definition is *regular* if for every two ranks (total or partial) $s_1 \cdots s_n \rightarrow s$ and $t_1 \cdots t_n \rightarrow t$ one of the following two conditions is satisfied:

1. $s_i \sqcap t_i$ exists for all i, $s_1 \sqcap t_1 \cdots s_n \sqcap t_n \rightarrow u$ is one of the declared ranks, $s \geq u$, and $t \geq u$
2. there is an i such that $s_i = f(\cdots)$, $t_i = g(\cdots)$, and there is no type constructor that is below f and g.

A signature is *regular* if each of its function declarations is regular.

Given a closed signature, the *corresponding abbreviation-free signature* is obtained by deleting all abbreviation declarations and repeatedly expanding the remaining occurrences of the abbreviations until all abbreviations are eliminated. Of course, this elimination process only terminates if abbreviations are used nonrecursively, a property that is checked by TEL.

A signature is *consistent* if it is closed and its corresponding abbreviation-free signature exists and is well-founded, coherent, complete, and regular.

6.3 Views

The syntax of views is as follows:

view $\longrightarrow$
  '`view`' *module_name* '`.`'
  '`imports`' { *module_name* } '`.`'
  transfer_declaration
  (*transfer_declaration*)*
  '`endview`' '`.`'

transfer_declaration $\longrightarrow$
  '`from`' *module_name* '`:`' { *designator* ['`abstract`'] } '`.`'

The `imports`-sentence defines the *import signature* of the view. The module names listed in the `imports`-sentence must be pairwise distinct. If a designator is exported by more than one of the imported modules, all exports must be defined in the same module. The import signature of the view is obtained as the union of the export signatures of the imported modules, where for a designator that is exported abstract by some of the imported modules and nonabstract by some other imported modules the nonabstract declaration is taken. The thus obtained import signature of the view must be consistent.

An example of a view whose import signature is inconsistent is `mod3`:

```
interface mod1.
   tya := {a}.  tyb := {b}.  tyc := tya ++ tyb ++ {c}.
endinterface


interface mod2.
   imports mod1.
   from mod1: tya, tyb.
   tyd := tya ++ tyb ++ {d}.
endinterface


view mod3.
   imports mod1, mod2.
   from mod1: tya, tyb.
   from mod2: tyd.
endview.
```

The import signature of the view `mod3` violates the completeness condition since `tyc` and `tyd` don't have a greatest common subtype although they have `tya` and `tyb` as common subtypes.

The `from`-sentences of a view define the *export signature* of the view. There can be at most one `from`-sentence for a module. The designators listed in a `from`-sentence for a module M must be pairwise distinct and actually be defined and exported by M. Thus the view

```
view mod4.
   imports mod2.
   from mod2: tya, tyb.
endview.
```

is inconsistent, while the view

```
view mod5.
   imports mod2.
   from mod1: tya, tyb.
endview.
```

is consistent.

Every designator listed in a from-sentence must be exported by at least one of the modules imported by the view. If a designator in a from-sentence for a module M is exported nonabstract by M but is declared abstract in the import signature of the view, it must be qualified abstract in the from-sentence. On the other hand, a designator listed in a from-sentence can be qualified abstract although it is declared nonabstractly in the import signature of the view. The interface frontend discussed before gives you an example of this kind of information hiding.

The export signature of a view is obtained from the import signature of the view by deleting the declarations of the designators that are not listed in a from-sentence. If a designator is qualified as abstract in a from-sentence of the view but is declared nonabstractly in the import signature of the view, it is declared as abstract in the export signature of the view. The thus obtained export signature of the view must be consistent.

6.4 Interfaces

The syntax of interfaces is defined as follows:

```
interface ⟶
    'interface' module_name '.'
    [ 'imports' { module_name } '.'
      [ (transfer_declaration)*
        declaration ] ]
    (declaration)*
    'endinterface' '.'
```

The imports-sentence of an interface must satisfy the same conditions as the imports-sentence of a view and the *import signature* of an interface is defined analogously to the import signature of a view. Furthermore, the requirements for the from-sentences of an interface are the same as for the from-sentences of a view.

The *export signature* of an interface is the signature defined by the from-sentences of the interface together with the other declarations appearing in the interface. The export signature of an interface must be consistent. The signature defined by the from-sentences of an interface is obtained in the same way it is obtained for views and must be consistent.

A declaration of an interface must not declare a designator that is declared in the import signature of the interface. For this reason the interface

```
interface mod6.
   imports mod1.
   tye := {a}.
endinterface.
```

is inconsistent although its import and export signature are consistent.

A type constructor is called *abstract* if it is declared as abstract, or its definition has the form

$$f(\cdots) := f_1(\cdots) \texttt{++} \cdots \texttt{++} f_n(\cdots),$$

where $f_1, \ldots, f_n$ are abstract type constructors. A type constructor is called *concrete* if every type constructor appearing in the right-hand side of its definition is concrete (taking the greatest fixed point of this inductive definition). A type constructor is called *mixed* if it is neither abstract nor concrete. For instance, in the export signature of the interface

```
interface mixed.
   aty1 := abstract.
   aty2 := abstract.
   aty3 := aty1 ++ aty2.
   cty := {a, f: cty}.
   mty := {g: aty, h: cty}.
endinterface.
```

aty1, aty2 and aty3 are abstract, cty is concrete, and mty is mixed.

We need an additional requirement to ensure that abstract types cannot be inspected. To see this, consider the illegal interfaces

```
interface mod7.
   tya := abstract.
   foo: int --> tya
endinterface.


interface mod8.
   imports mod7.
   tyb := tya ++ {b}.
endinterface.
```

If TEL would accept these interfaces, then a programmer could write the equation foo(5)=b, which would allow him to find out whether the value of foo(5) is b.

A type definition is called *protecting* if it has either the form

$$f(\cdots) := f_1(\cdots) \texttt{++} \cdots \texttt{++} f_n(\cdots),$$

where either all or none of the f_i are abstract, or it has the form

$$f(\cdots) := f_1(\cdots) \texttt{++} \cdots \texttt{++} f_n(\cdots) \texttt{++} \{\cdots\},$$

where none of the f_i is abstract.

Every type definition of the corresponding abbreviation-free signature of the export signature of an interface must be protecting.

6.5 Module Bodies

The syntax of module bodies is defined as follows:

```
module_body  ⟶
    'module' module_name '.'
    (definition)*
    'endmodule' '.'

definition  ⟶
     type_definition
    | type_abbreviation
    | parameter_definition
    | function_definition
    | relation_definition
    | procedure_definition

parameter_definition  ⟶
    'par' identifier ':' ground_type_term '=' term
        [ '<--' condition_part ] '.'

function_definition  ⟶
    function_declaration
        (functional_clause)*

relation_definition  ⟶
    relation_declaration
        (relational_clause)*

procedure_definition  ⟶
    procedure_declaration
        (relational_clause)*
```

The import signature of a module is the import signature of its interface and the export signature of a module is the export signature of its interface.

A module body can be compiled only after its interface has been compiled. If the interface of a module is empty, both the import and the export signature of the module are the signature consisting of all built-in declarations. The *local signature* of a module is its import signature joined with the declarations appearing in its body. The local signature of a module must be consistent.

If the corresponding abbreviation-free signature of the local signature of a module contains a type definition

$$f(x_1, \ldots, x_m) \text{ := } \cdots \text{ ++ } g(s_1, \ldots, s_n) \text{ ++ } \cdots \ .$$

such that f is not declared in the import signature of the module, then g must not be an abstract type constructor. This requirement corresponds to the protection requirement for interfaces.

Every declaration that appears in the interface of a module must appear in exactly the same form in the body of the module. The only exception to this rule are abstract type declarations, where the body must contain a type definition or a type abbreviation whose left-hand side equals the left-hand side of the abstract type declaration in the interface.

6.6 Compiling Views, Interfaces and Module Bodies

Appendix D lists the commands for compiling views, interfaces and module bodies.

A module body can be compiled only after its interface has been compiled. However, if you ask TEL to compile a module body for which you haven't written an interface yet, TEL will assume that the module has an empty interface and compile the body under this assumption.

If you ask TEL to compile an interface or a view, it will first compile all imported interfaces and views that haven't been compiled yet. Only after all imported interfaces have been compiled successfully, TEL will attempt the compilation of the importing interface or view.

If you recompile a module body, possibly after you have changed it, no other module body or interface needs to be recompiled.

If you recompile an interface or a view M, the body of M (if M isn't a view) and all interfaces, bodies, and views importing M directly or indirectly will be marked as uncompiled.

6.7 Opening Modules

A module can be opened only if it either has no interface or its interface and all imported modules have already been compiled successfully. If these requirements are met but the body of the module to be opened hasn't been compiled successfully yet, TEL will attempt to compile the body. Once the body of the module to be opened is compiled successfully, TEL starts loading all imported modules that aren't loaded already and then loads the module to be opened. Finally, after all modules have been loaded, TEL loads the local signature of the module to be opened and prompts you for the next query.

Once a module is loaded (not necessarily opened), it remains loaded as long as it is not marked as uncompiled. Of course, a module will be unloaded if you recompile its body.

Parameter definitions are executed only once when the module in which they are defined is loaded. At any one time, at most one instance of a module is loaded. Thus, if the value of a parameter is a data base (an imperative concept that will be discussed later), all importing modules will use the same data base.

7 Open Variables

Relations in TEL must be declared with fixed input and output arguments. TEL checks the consistent use of these data flow declarations and thus ensures a clean integration of functions and relations. However, data flow declarations restrict the possibilities of how one can compute with relations. To regain the full power of Prolog, TEL offers the possibility to declare variables as open. Since TEL's type checker considers variables declared as open as bound to ground terms, open variables provide a means to bypass TEL's data flow discipline. This provides for all the advanced programming techniques that were developped for Prolog.

The philosophy behind data flow declarations is that programs without open variables are far easier to understand than programs with open variables. Since in large programs open variables are only used in a few places, declaring them is not much effort.

In the literature on logic programming, open variables are called logical variables. Since the practical use of "logical variables" almost always involves the use of nonlogical operations (for instance, testing at run time whether a variable is bound), this name is somewhat misleading.

Suppose you have opened a module containing the definition:

```
rel append: list(T) x list(T) x list(T).
   append(nil, L, L).
   append(H.T, L, H.TL) <-- append(T, L, TL).
```

Then you can pose the following query:

```
TEL> !L1 & !L2 & append(L1, L2, 1.2.nil).
L1 = nil : elist
L2 = 1.2.nil : list(posint)
more answers? (y/n) y
L1 = 1.nil : list(posint)
L2 = 2.nil : list(posint)
more answers? (y/n) y
```

```
L1 = 1.2.nil : list(posint)
L2 = nil : elist
more answers? (y/n) y
failed.
```

Since all three arguments of append are input arguments, TEL won't accept the query `append(L1, L2, 1.2.nil)`. However, since the variables `L1` and `L2` are declared as open in the query above, TEL's type checker considers them as bound to ground terms. A term is ground or closed if it doesn't contain variables, and a term is open, if it contains variables. The execution of append binds `L1` and `L2` since the formal arguments in the clause heads are unified with the actual arguments. Since append is not declared as determinate, TEL can compute more than one answer. If append were declared as a `drel`, TEL would compute only the first answer.

Here is another query you can pose using open variables:

```
TEL> !L1 & !L2 & !T & append(L1, L2, 1.T).
L1 = nil : elist
L2 = 1.T : list(posint)
more answers? (y/n) y
L1 = 1.nil : list(posint)
L2 = T : list(posint)
more answers? (y/n) y
L1 = 1._1.nil : list(posint)
L2 = _2 : list(posint)
T  = _1._2 : list(posint)
more answers? (y/n) y
L1 = 1._1._3.nil : list(posint)
L2 = _2 : list(posint)
T  = _1._3._2 : list(posint)
more answers? (y/n).
```

For this query TEL could in fact compute infinitely many answers. The open variables _1, _2 and _3 were not given in the query but were generated during execution.

Another interesting query that makes use of TEL's typed unification is:

```
TEL> !L1 & !L2 & L2:list(negint) &
     append(L1, L2, 1.2.0.~1.~2.nil).
L1 = 1.2.0.nil : list(nat)
L2 = ~1.~2.nil : list(negint)
more answers? (y/n) y
L1 = 1.2.0.~1.nil : list(int)
L2 = ~2.nil : list(negint)
more answers? (y/n) y
L1 = 1.2.0.~1.~2.nil : list(int)
L2 = nil : elist
more answers? (y/n) y
failed.
```

7.1 Example: Tables as Open Data Structures

Without open variables the efficient implementation of tables allowing for insertion and deletion of entries is impossible. However, such tables can be implemented quite efficiently by using open variables. Here we will implement tables as an abstract data type with the interface:

```
interface table.
   table(Entry) := abstract.
   tdrel insert: string x Entry x table(Entry).
   drel lookup: string x ?Entry x table(Entry).
   drel remove: string x table(Entry).
   allkeys: table(Entry) --> list(string).
   compress: table(Entry) --> table(Entry).
endinterface.
```

Note that table is declared as a unary abstract type constructor and Entry is used as a type variable. Furthermore, note that no function or relation is exported with which you can create an empty table. The reason for this has to do with the fact that empty tables are implemented as open variables and will be discussed thoroughly at the end of the section. From the above interface definition you can see that the same name can be used for the module and an object in the module.

We will implement tables with binary search trees, where empty trees are represented as open variables. Every node of the tree comes with a delete flag that is an open variable as long as the node is not deleted.

```
table(Entry) := {node: string x        % key
                       Entry x
                       table(Entry) x  % left subtree
                       table(Entry) x  % right subtree
                       zero}           % delete flag
```

Insertion of a new entry is defined as follows:

```
tdrel insert: string x Entry x table(Entry).


   insert(K, E, T) <-- var(T) &
        node(K, E, _, _, _) = T.


% the remaining clauses assume that the third argument
% is not a variable


   insert(K, E, node(K,_,L,_,0)) <--
        insert(K, E, L).
   % an already existing entry for K will be deleted


   insert(K, E, node(CK,_,L,_,_)) <-- K @< CK &
        insert(K, E, L).
        insert(K, E, node(CK,_,_,R,_)) <-- % CK @< K &
             insert(K, E, R).
```

The built-in relation var

```
    drel var: T
```

succeeds if and only if its argument is a variable. The equational condition `node(K, E, _, _, _) = T` of the first clause of `insert` will always succeed since it is executed only if T is a variable. Since equational conditions are executed by unifying their left-hand with their right-hand side, execution of this equation will bind T to the term `node(K, E, _, _, _)`. Note that this term contains three new open variables for the left subtree, the right subtree and the delete flag. You don't have to declare these variables as open since TEL's type checker believes that T, which appears as an input argument, is bound to a ground term. Since `insert` is forced by declaration to be determinate, the clauses following the first clause will only be used if the third argument is not a variable.

The lookup relation is defined as follows:

```
    drel lookup: string x ?Entry x table(Entry).
       lookup(K, E, T) <--  naf var(T) &
            lookup1(K, E, T).


    drel lookup1: string x ?Entry x table(Entry).
    % the third argument must not be a variable


       lookup1(K, E, node(K,E,_,_,D)) <-- var(D).


       lookup1(K, E, node(K,E,L,_,D)) <-- % naf var(D) &
            lookup(K,E,L).


       lookup1(K, E, node(CK,_,L,_,_)) <-- K @< CK &
            lookup(K, E, L).
```

```
   lookup1(K, E, node(CK,_,_,R,_)) <-- % CK @< K &
        lookup(K, E, R).
```

The relation `lookup` fails if the table doesn't contain an undeleted entry for the given key. Thus it can be used to test whether a table contains an entry for a key.

The definition of `remove` is quite similar to the definition of `lookup`:

```
drel remove: string x table(Entry).
   remove(K, T) <-- naf var(T) &
        remove1(K, T).

drel remove1: string x table(Entry).
% the second argument must not be  a variable

   remove1(K, E, node(K,E,_,_,D)) <-- var(D) &
        D = 0.

   remove1(K, E, node(K,E,L,_,D)) <-- % naf var(D) &
        remove(K,E,L).

   remove1(K, E, node(CK,_,L,_,_)) <-- K @< CK &
        remove(K, E, L).

   remove1(K, E, node(CK,_,_,R,_)) <-- % CK @< K &
        remove(K, E, R).
```

Like `lookup` `remove` fails if there is no undeleted entry for the given key in the table.

The function `allkeys` returns the list of all keys for which the argument table contains an undeleted entry.

```
allkeys: table(Entry) --> list(string).
   T                |> nil
                    <-- var(T).
```

```
% the remaining clauses assume that the argument
% is not a variable

   node(K,_,L,R,D)   |> K.(allkeys(L)|allkeys(R))
                     <-- var(D).  % entry is not deleted


   node(K,_,L,R,D)   |> allkeys(L)|allkeys(R).
                     %<-- naf var(D). % entry is deleted
```

The function compress builds a new table that doesn't contain deleted entries.

```
compress: table(Entry) --> table(Entry).
   T                 |> T
                     <-- var(T).


% the remaining clauses assume that the argument
% is not a variable

   node(K,E,L,R,D)   |> node(K, E, compress(L), compress(R), D)
                     <-- var(D).  % entry is not deleted


   node(K,_,L,R,D)   |> compress1(L,compress(R)).
                     %<-- naf var(D).  % entry is deleted


compress1: table(Entry) x table(Entry) --> table(Entry).
   T, T1                 |> T1
                         <-- var(T).


% the remaining clauses assume that the argument
% is not a variable
```

```
    node(K,E,L,R,D), T  |> CT
                       <-- var(D) &  % entry is not deleted
                           CT = compress1(L, compress1(R,T)) &
                           insert(K,E,CT).


    node(_,_,L,R,D), T  |> compress1(L, compress1(R,T)).
                       %<-- naf var(D).  % entry is deleted
```

Next we write a module importing table:

```
interface table_test.
   imports table.
endinterface.


module table_test.
   tdrel empty_int_table: ?table(int).
      empty_int_table(T) <-- !T.
endmodule.
```

After you have opened the module table_test, you can, for instance, enter the query

```
TEL> empty_int_table(T) &
     insert('five', 5, T) & insert('four', 4, T) &
     lookup('five', E, T).
```

and TEL will respond:

```
T = abstract : table(int)
E = 5 : posint.
```

Since the type of T is abstract, TEL doesn't print the actual value of T but just tells you that it is abstract.

This example should give you a rough idea of what you can do with open variables. In Prolog textbooks you can find further examples. Basically, open variables are single-assignment pointers that become invisible once they are bound. Since you can bind an open variable to a term containing further open variables, open variables give you a means for building data structures incrementally. For large applications, like the TEL system itself, the efficiency gained from using open variables can be of vital importance. However, since open data structures require much more care for the operational semantics than closed data structures, I recommend the use of open data structures only if a solution using closed data structures is significantly more complicated or significantly less efficient.

Now let's discuss why I didn't equip the interface of the `table` module with a relation that creates empty tables. Since `table` is an abstract data type, it is in fact rather awkward to not hide the information that empty tables are variables. The problem is that in TEL it is impossible to write a relation

```
tdrel empty_table: ?table(T).
   empty_table(Table) <-- !Table.
```

since every variable that occurs in an output position of a relational domain must occur in at least one input position of the domain. This restriction is essential for the operational semantics of TEL since at run time every open variable must have a unique type not containing variables.

Introducing a dummy input argument

```
tdrel empty_table: T x ?table(T).
   empty_table(_, Table) <-- !Table.
```

won't satisfy TEL's type checker either since this still doesn't allow to infer a ground type term for `Table` at compile time. The type checker will accept an open variable X only if at least one of the following conditions is satisfied:

- X occurs in the output argument of a relational condition
- X occurs at the left-hand side of an equational condition
- it is possible to infer a ground type term for X.

The empty_table example shows a weakness of TEL's type system that needs to be resolved in the future. The most promising solution seems to make types first-class objects, which would allow the following elegant solution:

```
tdrel empty_table: T:type x ?T.
   empty_table(ETYPE, T) <-- !T & T:table(ETYPE).
```

8 Type Checking

This section defines how the clauses of a module body are type checked. Before you read this section you should be familiar with the notion of a consistent signature and the definitions introduced in Subsection 6.2.

To type check the clauses of a module body, TEL uses the corresponding abbreviation-free signature of the local signature of the module. All the following definitions are made with respect to a given consistent and abbreviation-free signature.

To type check the clauses of a module body, TEL extends the type terms defined by the corresponding abbreviation-free signature of the local signature of the module by the special nullary type constructor $\bot$. The above order "$s \geq t$" on type terms is extended such that $\bot$ becomes the least type term, that is, $s \geq \bot$ for every type term s. You may think of $\bot$ as an empty type that is a subtype of every type. A program cannot explicitly use $\bot$, but $\bot$, which is printed as `void` by TEL, can occur in the answer to a query (Section 2 gives an example for such a query).

In this section, we will use the term *value function* for value constructors, parameters and functions. To ease our notation, we will use uniform ranks for all value functions. These ranks have the form $s_1 \cdots s_n \rightarrow s$, where $n \geq 0$ and $s_1, \ldots, s_n$ and s are type terms. No distinction is made between partial and total ranks of functions.

We use $\mathcal{V}(s)$ to denote the set of all variables occurring in a term s.

A value term is called *canonical* if it consists only of variables and value constructurs.

A *variable qualification* is a pair $x{:}\,s$ consisting of a variable x and a type term s. A *prefix* is a set of variable qualifications such that no variable is qualified more than once. We use $\mathcal{D}(P)$ to denote the set of all variables qualified by a prefix P.

8.1 Type Checking Terms

Given a value function f and $n \geq 0$ type terms $s_1, \ldots, s_n$, where n is the arity of f, the *least codomain of f for $s_1, \ldots, s_n$* is defined as follows:

$$\mathbf{least\_codomain}(f, (s_1, \ldots, s_n)) :=$$

$$\min\{\theta t \mid t_1 \cdots t_n \to t \text{ is a rank of } f \text{ and } \theta t_1 \geq s_1, \ldots, \theta t_n \geq s_n\}.$$

The letter θ ranges over substitutions that replace variables with type terms. The minimum is taken with respect to the above order "$s \geq t$" for type terms. Of course, the least codomain of f for $s_1, \ldots, s_n$ doesn't always exist. However, the regularity condition for signatures ensures that the least codomain exists if and only if there is at least one rank $t_1 \cdots t_n \to t$ of f and a substitution θ such that $\theta t_i \geq s_i$ for $i = 1, \ldots, n$.

In TEL, every well-typed value term has a unique *least type term.* The partial function $P \uparrow s$ yields the least type term of a value term s under a prefix P and is defined as follows:

1. $P \uparrow x = t$

   if $(x{:}\, t) \in P$

2. $P \uparrow x = \perp$

   if x is not qualified in P

3. $P \uparrow f(s_1, \ldots, s_n) = \mathbf{least\_codomain}(f, P \uparrow s_1, \ldots, P \uparrow s_n)$

A value term s is *well-typed under a prefix* P if $\mathcal{V}(s) \subseteq \mathcal{D}(P)$ and $P \uparrow s$ exists.

A type term t is called *proper* if there exist a canonical value term s and a prefix P not containing $\perp$ such that $\mathcal{V}(s) = \mathcal{D}(P)$ and $t \geq (P \uparrow s)$. For instance, `list`($\perp$) is proper since $\emptyset \uparrow$`nil` = `elist` and `list`($\perp$) $\geq$ `elist`, while the type term `nat##`$\perp$ is not proper.

A prefix is called *proper* if each of its type terms is proper. If a value term s is well-typed under a proper prefix P, then $P \uparrow s$ is proper.

The partial function $P \uparrow s$ is a central component of TEL's type checker. It is used for checking whether value terms are well-typed and for computing their least type terms.

8.2 Inferring the Types of Variables

The *noncanonical variables* of a value term are defined as follows:

1. $\mathcal{NCV}(x) = \emptyset$
2. $\mathcal{NCV}(f(s_1, \ldots, s_n)) = \mathcal{NCV}(s_1) \cup \cdots \cup \mathcal{NCV}(s_n)$

   if f is a value constructor
3. $\mathcal{NCV}(f(s_1, \ldots, s_n)) = \mathcal{V}(s_1) \cup \cdots \cup \mathcal{V}(s_n)$

   if f is not a value constructor.

A variable occurring in a value term s is called a *canonical variable of* s if it is not a noncanical variable of s.

Given a type term s and a value constructor f with rank $t_1 \cdots t_n \rightarrow t$, the greatest domain of f for s is

$$\textbf{greatest\_domain}(f, s) := \max\{\theta(t_1, \ldots, t_n) \mid s \geq \theta t\},$$

where the maximum is taken with respect to the order obtained by extending the above order componentwise to tuples of type terms. The greatest domain of f for s exists if and only if there exists a substitution θ such that $s \geq \theta t$. If s is proper, then the greatest domain of f for s is a tuple of proper terms since the codomains of value constructors are linear, that is, no variable occurs twice.

The partial function $P \downarrow M$ takes two arguments: P must be a proper prefix and M must be a set of containments $s\!: t$ such that s is a value term and t is a type term. If $P \downarrow M$ is defined, it yields a proper prefix that extends the given prefix P by adding and strengthening qualifications for the canonical variables occurring in the value terms of M. The definition of $P \downarrow M$ is as follows:

1. $P{\downarrow}\emptyset = P$

2. $P{\downarrow}(\{s{:}\,t\} \uplus M) = (P{\downarrow}\{s{:}\,t\}){\downarrow}M$

3. $P{\downarrow}\{x{:}\,s\} = P \cup \{x{:}\,s\}$

   if x is not qualified in P and s is proper

4. $(P \uplus \{x{:}\,t\}){\downarrow}\{x{:}\,s\} = P \cup \{x{:}\,(t \sqcap s)\}$

   if $t \sqcap s$ is proper

5. $P{\downarrow}\{f(s_1,\ldots,s_n){:}\,t\} = P{\downarrow}\{s_1{:}\,t_1,\ldots,s_n{:}\,t_n\}$

   if f is a value constructor and $(t_1,\ldots,t_n) = \mathbf{greatest\_domain}(f,t)$

6. $P{\downarrow}\{f(s_1,\ldots,s_n){:}\,t\} = P$

   if f is not a value constructor.

If s is a term, P is a proper prefix such that $\mathcal{NCV}(s) \subseteq \mathcal{D}(P)$, and $P{\uparrow}s$ is defined and proper, then $P \downarrow \{s{:}\,(P \uparrow s)\}$ is defined and is a proper prefix qualifying all variables in s and satisfying $P{\uparrow}s = (P \downarrow \{s{:}\,(P \uparrow s)\}){\uparrow}s$. Thus $P{\downarrow}\{s{:}\,t\}$ is a function that infers types for the canonical variables of a term s.

8.3 Typechecking Conditions

The type checker for conditions is a partial function $F.O.P[C]$ that takes four arguments: a set of variables F, a set of variables O, a prefix P, and a condition C. The argument F is the set of "forbidden variables", that is, variables that must not occur in C. The argument O is the set of variables that have been declared as open in preceding conditions. The prefix P qualifies all variables for which types have been already derived. And C is the condition to be type checked under F, O and P. If $F.O.P[C]$ is defined, then C is well-typed under F, O and P. The result of $F.O.P[C]$ is a triple $F'.O'.P'$, which extends the input triple $F.O.P$ with the information obtained from the condition C.

The empty condition and conjunctions are the trivial cases:

- $F.O.P[\emptyset] = F.O.P$

- $F.O.P[C\&C'] = (F.O.P[C])[C']$.

  Negation as failure is checked as follows:

- $F.\emptyset.P[\texttt{naf}\ \ C] = (F' \cup (\mathcal{D}(P') - \mathcal{D}(P))).\emptyset.P$

  if $F'.\emptyset.P' := F.O.P[C]$ is defined.

The type checking rule for negation as failure shows the purpose of the list F of forbidden variables. Since variable bindings produced during the execution of C are not propagated outside of `naf` C, variables introduced in C must not be used outside of `naf` C.

Now we come to the type checking rules for primitive conditions. Declarations of open variables are easy to check:

- $F.O.P[\,!\,x] = F.O \cup \{x\}.P$

  if $x \notin F \cup O \cup \mathcal{D}(P)$.

Declaring a variable x as open is okay only if x did not appear so far.

Containments are checked as follows:

- $F.O.P[s:t] = F.\emptyset.(P\downarrow\{s:t\})$

  if t is a type,
  $O \subseteq \mathcal{V}(s) \subseteq O \cup \mathcal{D}(P)$,
  $\mathcal{NCV}(s) \subseteq \mathcal{D}(P)$, and
  $((P\downarrow\{s:t\})\uparrow s) \geq t$.

The right-hand side of a containment must be a type term not containing variables.

Discontainments are checked as follows:

- $F.\emptyset.P[s\backslash:t] = F.O.P$

  if $\mathcal{V}(s) \subseteq \mathcal{D}(P)$ and $F.O.P[s:t]$ is defined.

Equations are checked as follows:

- $F.\emptyset.P[s\texttt{=}t] = F.\emptyset.P\!\downarrow\!\{s\colon(P\!\uparrow\! t)\}$

  if $\mathcal{NCV}(s) \cup \mathcal{V}(t) \subseteq \mathcal{D}(P)$,
  F and $\mathcal{V}(s)$ are disjoint, and
  $(P\!\downarrow\!\{s\colon(P\!\uparrow\! t)\})\!\uparrow\! s \sqcap P\!\uparrow\! t$ exists and is proper.

The type checker treats equations asymmetrically to enforce a certain programming style: only the left-hand side can contain variables for which types have not been derived so far. Of course, the logical semantics and the execution of equations are symmetric.

Furthermore, an equation type checks only if the least type terms of the left and the right-hand side have a proper infimum. The reason for this requirement is that in TEL two terms can denote the same value only if their least type terms have a proper common lower bound.

Disequations are checked as follows:

- $F.\emptyset.P[s\texttt{\textbackslash=}t] = F.\emptyset.P$

  if $\mathcal{V}(s) \subseteq \mathcal{D}(P)$ and $F.O.P[s\texttt{=}t]$ is defined.

Boolean conditions are abbreviations for equations:

- $F.\emptyset.P[s] = F.\emptyset.P[\texttt{true=}s]$

  if the top symbol of s is a variable or a function.

To check relational conditions, we need a further auxiliary function. Given a relation p with the domain

$$t_1 \times \cdots \times t_k \times\ ?t_{k+1} \times \cdots \times\ ?t_n$$

and type terms $s_1, \ldots, s_k$ $(k \geq 0)$, the *least domain* of p for $s_1, \ldots, s_k$ is defined as follows:

$$\textbf{least\_domain}(p, (s_1, \ldots, s_k)) :=$$

$$\min\{\theta(t_1, \ldots, t_n) \mid \theta t_1 \geq s_1 \wedge \cdots \wedge \theta t_k \geq s_k\}.$$

Of course, the least domain of p for $s_1, \ldots, s_k$ doesn't always exist. To ease the notation, we assume in this section that in a relational domain the input arguments always appear before the output arguments. This assumption is purely for notational convenience; in TEL one is of course free to arrange input and output arguments in any order.

Relational and procedural conditions are checked as follows:

- $F.O.P[p(s_1, \ldots, s_n)] = F.\emptyset.(P \downarrow \{s_1 \colon t_1, \ldots, s_n \colon t_n\})$

  if p is a relation or a procedure having the positions $1, \ldots, k$ as input and the positions $k+1, \ldots, n$ as output arguments,
  $(t_1, \ldots, t_n) = \mathbf{least\_domain}(p, (P \uparrow s_1, \ldots, P \uparrow s_k))$ exists,
  $t_1, \ldots, t_k$ are proper,
  $(P \uparrow s_i) \sqcap t_i$ exists and is proper for $i = k+1, \ldots, n$,
  $\mathcal{NCV}(s_1) \cup \cdots \cup \mathcal{NCV}(s_n) \subseteq \mathcal{D}(P)$,
  $O \subseteq \mathcal{V}(s_1) \cup \cdots \cup \mathcal{V}(s_k) \subseteq O \cup \mathcal{D}(P)$,
  $\mathcal{V}(s_{k+1}) \cup \cdots \cup \mathcal{V}(s_n)$ and F are disjoint, and
  $P \downarrow \{s_1 \colon t_1, \ldots, s_k \colon t_k\}$ qualifies every variable in O with a ground type term.

8.4 Type Checking Clauses

A functional clause $f(s_1, \ldots, s_n)$ = s <-- C is well-typed if

- f is a function and $s_1, \ldots, s_n$ are canonical value terms
- there exists a rank $t_1 \cdots t_n \to t$ of f such that $(\emptyset \downarrow \{s_i \colon t_i\}_{i=1}^{n})$ is defined
- for every rank $t_1 \cdots t_n \to t$ of f such that $(\emptyset \downarrow \{s_i \colon t_i\}_{i=1}^{n})$ is defined:
  - $F.O.P := \emptyset.\emptyset.(\emptyset \downarrow \{s_i \colon t_i\}_{i=1}^{n})[C]$ is defined and $O = \emptyset$
  - $\mathcal{V}(s) \subseteq \mathcal{D}(P)$ and $t \geq P \uparrow s$.

A relational clause $p(s_1, \ldots, s_n)$ <-- C, where p is a relation with the domain

t_1 x $\cdots$ x t_kx ?t_{k+1} x $\cdots$ x ?t_n, is well-typed if

- $s_1, \dots, s_n$ are canonical value terms
- $F.O.P := \emptyset.\emptyset.(\emptyset\!\downarrow\!\{s_i\!: t_i\}_{i=1}^{k})[C]$ is defined
- $O \subseteq \mathcal{V}(s_{k+1}) \cup \dots \cup \mathcal{V}(s_n) \subseteq O \cup \mathcal{D}(P)$
- $t_i \geq P\!\uparrow\! s_i$ for $i = k+1, \dots, n$
- $P\!\downarrow\!\{s_i\!: t_i\}_{i=k+1}^{n}$ qualifies every variable in O with a ground type term.

  A parameter definition par p : t = s <-- C is well-typed if

- $F.O.P := \emptyset.\emptyset.\emptyset[C]$ is defined and $O = \emptyset$
- $\mathcal{V}(s) \subseteq \mathcal{D}(P)$ and $t \geq P\!\uparrow\! s$.

TEL's type checker runs with polynomial complexity with respect to the length of a clause. Furthermore, for every rank or domain, TEL's type checker goes only once from left to right through a clause and decides immediately whether a primitive condition is well-typed. Such a local and deterministic strategy is crucial for the ability to give precise and localized error messages in case a clause is not well-typed.

8.5 Type Checking Queries

Queries have the same syntax and the same operational semantics as conditions of clauses. A query C is well-typed if $F.O.P := \emptyset.\emptyset.\emptyset[C]$ is defined and $O = \emptyset$.

9 Streams and Procedures

Streams are internal representations of files opened for reading or writing. Streams are values of so-called stream types, which are obtained by two built-in abstract type constructors:

```
instream(T) := abstract.
outstream(T) := abstract.
```

There are three operations for opening a file and binding it to a newly created stream:

```
proc open_instream: string x T:type x ?instream(T).
proc open_outstream: string x T:type x ?outstream(T).
proc append_outstream: string x T:type x ?outstream(T).
```

The first argument specifies the name of the file to be opened. The second argument specifies the element type of the file. TEL considers a file to be a list of values all belonging to the same type. The third argument returns a new stream that is connected to the file that was opened.

Files whose elements are characters are text files and can be edited with any text editor the system provides. All other files are kept in a special format and should only be written and read by TEL.

The procedure `open_instream` opens a file for reading. If the file to be opened doesn't exist, `open_instream` fails. The procedure `open_outstream` opens a file for writing. If the file to be opened exists, `open_outstream` will delete all elements of the file so that the file becomes empty. If the file to be opened doesn't exist, `open_outstream` creates a new file with the given name. The procedure `append_outstream` opens a file for writing without overwriting its existing elements. If the file to be opened doesn't exist, `append_outstream` creates a new file with the given name.

After input from or output to a stream is finished, a stream must be closed with one of the built-in procedures:

```
tproc close_instream: instream(T).
tproc close_outstream: outstream(T).
```

The closing operations free the file connected to the given stream, so that the file can be used again by other programs. After a stream is closed, an attempt to access this stream will cause a run-time error.

The principal procedures for reading and writing are:

```
proc get: instream(T) x ?T.
tproc put: outstream(T) x T.
```

The procedure get fails if the given input stream contains no further element, that is, the end of the file connected to the stream is reached. If get fails on an input stream, a further call of get on this stream will cause a run-time error and abort execution.

If you type the query

```
TEL> open_outstream('myfile', string#int ,SO) &
     put(SO, 'Time is money'#3) &
     put(SO, 'and love is honey.'#4) &
     close_outstream(SO) &
     open_instream('myfile', string#int, SI) &
     get(SI, E1) & get(SI, E2) &
     close_instream(SI).
```

TEL will answer:

```
SO = abstract : outstream(string#int)
SI = abstract : instream(string#int)
E1 = 'Time is money'#3 : nestring#posint
E2 = 'and love is honey.'#4 : nestring#posint.
```

It is possible to write terms containing open variables on a file. If such terms are read in again, the occurring variables are replaced consistently with new variables, where the scope of variables is limited to the term read by get.

For instance, the query

```
TEL> open_outstream('test', list(int) ,SO) &
     !X & !Y & put(SO, X.Y) & put(SO, X.X.Y) &
     close_outstream(SO) &
     open_instream('test', list(int), SI) &
     get(SI, E1) & get(SI, E2) &
     close_instream(SI).
```

will be answered by TEL with:

```
SO = abstract : outstream(list(int))
SI = abstract : instream(list(int))
X  = _1 : int
Y  = _2 : list(int)
E1 = _3._4 : list(int)
E2 = _5._5._6 : list(int).
```

There are three character streams that are always open and cannot be closed:

```
par user_input: instream(char).
par user_output: outstream(char).
par user_error: outstream(char).
```

You can use them to read from and to write on your TEL window. If you write on a stream, the information is usually not immediately transferred to the connected file but is kept in a buffer. With the procedure

```
tproc flush: outstream(T).
```

you can force TEL to actually write the buffer of the given stream on the file connected to the stream. This is particulary useful for the standard streams `user_output` and `user_error`.

The following functions, which cannot be used on the standard streams user_input, user_output and user_error, return information about the state of character streams:

```
lineno: instream(char) --> nat,
        outstream(char) --> nat.
charno: instream(char) --> nat,
        outstream(char) --> nat.
linepos: instream(char) --> nat,
         outstream(char) --> nat.
```

The function lineno yields the current line number of the stream. The function charno yields the number of characters read from or written to a stream so far. The function linepos yields the number of characters read from or written to the current line of the stream.

The procedure

```
tproc print: outstream(char) x T.
```

can write values of every type on a character stream. For instance, if you enter the query

```
TEL> print(user_output, 'I don''t be'^'lieve it!'#5*7) &
     flush(user_output).
```

TEL will answer:

```
'I don''t believe it'#35.
```

A use of print is only okay if TEL can infer a ground type term for the second argument. Hence, the procedure

```
proc doesnt_work: T.
   doesnt_work(X) <-- print(user_output, X).
```

won't type check.

Of course, print does not print abstract values. For instance, the query

```
TEL> open_outstream('test', char ,SO) &
     print(user_output, SO) &
     flush(user_output) &
     close_outstream(SO).
```

results in the answer:

```
abstract : outstream(list(int))
SO = abstract : outstream(list(int)).
```

The procedure

```
proc parse: instream(char) x T:type x ?T.
```

is the inverse to print: it reads characters until it reaches a full stop, that is, a period followed by a layout character, and then tries to build a ground term of the required type. If parse can't build a ground term of the required type from the characters read, it fails. Furthermore, parse fails if the end of the file is reached. Analogous to get, a second attempt to read after the end of the file has been reached will cause a run-time error.

Given the query

```
TEL> parse(user_input, list(int), L).
```

TEL prints the prompt > and waits until you type in a sentence, that is, a sequence of characters followed by a full stop. For instance, if you type

```
> 6.7.8.nil.
```

TEL will give the answer

```
L = 6.7.8.nil.
```

Of course, parse cannot read abstract values.

Now we have seen all built-ins for stream handling. Many of them are procedures, which were not discussed so far. Procedures are determinate relations that possibly change the state of the TEL system. TEL treats and executes procedures exactly like determinate relations, except for the following points:

- a procedure may have no argument, while a relation always must have at least one argument
- clauses of functions and relations cannot have procedural conditions.

Starting from the built-in procedures you can define further procedures. The following procedures, which are actually built-in, are examples for defined procedures:

```
tproc nl: outstream(char).
   nl(S) <-- put(S, "nl").


tproc put_string: outstream(char) x string.
   put_string(OS, S) <-- put_chars(chartrans(S), OS).

tproc put_chars: outstream(char) x list(char).
   put_chars(S, nil).
   put_chars(S, H.T) <-- put(S, H) & put_chars(S, T).
```

The rest of this section spells out how stream types and the special procedures `open_instream`, `open_outstream`, `append_outstream`, and `parse` are type checked.

Stream types are obtained by the unary abstract type constructors `instream` and `outstream`. These two type constructors are treated differently from the other abstract type constructors in the following respects:

- `instream`(s) and `outstream`(s) are type terms if and only if s is a ground type term
- $s \leq$ `instream`(t) if and only if there exists a type u such that $s =$ `instream`(u) and $u \leq t$

- `instream`$(s) \leq t$ if and only if there exists a type u such that $t =$ `instream`(u) and $s \leq u$
- $s \leq$ `outstream`(t) if and only if there exists a type u such that $s =$ `outstream`(u) and $t \leq u$
- `outstream`$(s) \leq t$ if and only if there exists a u such that $t =$ `outstream`(u) and $u \leq s$.

This means that the stream type constructors cannot be applied to type terms containing variables. Furthermore, the type constructor `instream` is monotonic, while the type constructor `outstream` is anti-monotonic.

The built-in procedures

`open_instream, open_outstream, append_outstreamand parse`

take a ground type term as argument. TEL won't allow you to use this kind of domains for relations or procedures you define yourself. Procedural conditions using these special procedures are type checked as follows:

- $F.\emptyset.P[\texttt{open\_instream}(s,t,x)] = F.\emptyset.(P \cup \{x\colon \texttt{instream}(t)\})$

  if $\mathcal{V}(S) \subseteq \mathcal{D}(P)$, $P{\uparrow}s \leq$ `string`,
  t is a type, and
  x is a variable not contained in $F \cup \mathcal{D}(P)$

- $F.\emptyset.P[\texttt{open\_outstream}(s,t,x)] = O.(P \cup \{x\colon \texttt{outstream}(t)\})$

  if $\mathcal{V}(S) \subseteq \mathcal{D}(P)$, $P{\uparrow}s \leq$ `string`,
  t is a type, and
  x is a variable not contained in $F \cup \mathcal{D}(P)$

- `append_outstream` is type checked like `open_outstream`

- $F.\emptyset.P[\texttt{parse}(s,t,u)] = F.\emptyset.P{\downarrow}\{u\colon t\}$

  if $\mathcal{V}(S) \subseteq \mathcal{D}(P)$, $P{\uparrow}s \leq$ `instream(char)`,
  t is a type,

$(P{\upharpoonleft}u) \sqcap t$ exists and is proper,
$\mathcal{NCV}(u) \subseteq \mathcal{D}(P)$ and $F \cap \mathcal{V}(u) = \emptyset$.

10 More on Conditions

So far we have not discussed all constructs that can be used in the condition part of a clause. Let's start with the complete syntax of condition parts.

```
condition_part  ⟶
    condition [ '&' condition_part ]

condition  ⟶
      conditional
    | simple_condition

conditional  ⟶
    'if' simple_conjunction 'then' cond_condition
    ('elsif' simple_conjunction 'then' cond_condition)*
    [ 'else' cond_condition 'fi' ]

simple_conjunction  ⟶
    simple_condition [ '&' simple_conjunction ]

cond_condition  ⟶
      'succeed'
    | 'fail'
    | condition_part

simple_condition  ⟶
      term '=' term
    | term '\=' term
    | term ':' closed_type_term
    | term '\:' closed_type_term
    | primitive_condition
    | 'naf' primitive_condition
    | '!' variable
    | 'do' primitive_condition
    | term 'islistof' term 'where' primitive_condition

primitive_condition  ⟶
    term
```

The above syntax also applies to the condition part of parameter definitions. Since the condition part of a parameter definition must always succeed, it must not contain the reserved identifier `fail`. The condition part of a parameter definition is executed when the module in which it appears is loaded. If the execution of a parameter definition fails, the module cannot be loaded and TEL prints an error message.

A conditional

`if` C_1 `then` C_2 `else` C_3 `fi`

is executed by first executing the condition C_1. If the execution of C_1 succeeds, C_1 is left determinate, that is, it cannot be backtracked, and the condition C_2 is executed. If the execution of C_1 fails, all variable bindings produced during its execution are retracted and the condition C_3 is executed. The execution of `succeed` always succeeds and the execution of `fail` always fails.

A one-handed conditional

`if` C_1 `then` C_2 `fi`

is an abbreviation for

`if` C_1 `then` C_2 `else succeed fi.`

A conditional with `elseif`

`if` C_1 `then` C_2
`elsif` C_3 `then` C_4
`else` C_5 `fi`

is an abbreviation for

`if` C_1
`then` C_2
`else if` C_3 `then` C_4 `else` C_5 `fi fi.`

The type checking rules for conditionals are:

- $F.\emptyset.P$[`if` C_0 `then` C_1 `else` C_2 `fi`] $= F'.\emptyset.P'$

  if neither C_1 nor C_2 is `fail`,
  $F_1.\emptyset.P_1 := F.\emptyset.P[C_0 \& C_1]$ is defined,
  $F_2.\emptyset.P_2 := F.\emptyset.P[C_2]$ is defined,
  $(F_1 \cup \mathcal{D}(P_1)) \cap (F_2 \cup \mathcal{D}(P_2)) \subseteq F \cup \mathcal{D}(P)$,
  $P' := \{(x{:}\, s_1 \sqcup s_2) \mid (x{:}\, s_1) \in P_1 \ \wedge \ (x{:}\, s_2) \in P_2 \}$ exists, and
  $F' := (F_1 \cup F_2 \cup \mathcal{D}(P_1) \cup \mathcal{D}(P_2)) - \mathcal{D}(P')$

- $F.\emptyset.P$[`if` C_0 `then fail else` C_2 `fi`] $= F'.\emptyset.P_2$

  if C_2 is not `fail`,
  $F_1.\emptyset.P_1 := F.\emptyset.P[C_0]$ is defined,
  $F_2.\emptyset.P_2 := F.\emptyset.P[C_2]$ is defined,
  $(F_1 \cup \mathcal{D}(P_1)) \cap (F_2 \cup \mathcal{D}(P_2)) \subseteq F \cup \mathcal{D}(P)$, and
  $F' := (F_1 \cup F_2 \cup \mathcal{D}(P_1)) - \mathcal{D}(P_2)$

- $F.\emptyset.P$[`if` C_0 `then` C_1 `else fail fi`] $= F.\emptyset.P[C_0 \& C_1]$

  if C_1 is not `fail`

- $F.\emptyset.P$[`succeed`] $= F.\emptyset.P$.

A `do`-condition

`do` C

is equivalent to the condition

`if naf` C `then succeed else succeed fi.`

All stack memory allocated during the execution of `do` C is released immediately after its execution. TEL has `do`-conditions since in large systems that run for a long time it can be essential to release memory. For instance, the implementation of TEL, which is written in TEL itself, does the compilation of modules within a `do`-condition. This works since all results of the compilation are written on files within the `do`-condition.

With an islistof-condition one can collect into a list all answers the execution of a relational condition can produce. For instance, if a module with the definitions

```
person := {dick, harry, tom, cathy}.
beverage := {beer, wine, cider, water, brandy, coffee}.


rel likes: ?person x ?beverage.
likes(dick, beer).
likes(dick, cider).
likes(cathy, water)
likes(cathy, wine).
likes(harry, beer).
likes(tom, brandy).
```

is opened, the query

```
TEL> L islistof B where likes(P, B).
```

will produce the answer:

```
L = beer.cider.water.wine.beer.brandy.nil  : list(beverage).
```

Furthermore, for the query

```
TEL> L islistof P#B where likes(P, B).
```

TEL will compute the answer:

```
L =dick#beer .dick#cider .cathy#water .cathy#wine
     .harry#beer .tom#brandy .nil : list(person ## beverage).
```

The execution of an islistof-condition always succeeds. For instance, the query

```
TEL> L islistof P where likes(P, coffee).
```

produces the answer

```
    L = nil : elist
```

since no person likes coffee.

The type checking rule for `islistof`-conditions is:

- $F.\emptyset.P[x$ `islistof` s `where` $r(t_1,\ldots,t_n)] = F.\emptyset.(P \cup \{x\colon$`list`$(Q\uparrow s)\})$

  if x is a variable not occurring in $F \cup \mathcal{D}(P)$,
  r is a relation,
  $F.\emptyset.Q := F.\emptyset.P[r(t_1,\ldots,t_n)]$ exists,
  $\mathcal{V}(s) \subseteq \mathcal{D}(Q)$, and
  $F' := F \cup (\mathcal{D}(Q) - \mathcal{D}(P))$.

Operationally, a disequation s`\=`t is equivalent to `naf` s`=`t. As long as s and t don't contain variables at execution time, `naf` s`=`t will in fact be disequality in the initial model, provided none of the involved relations is forced to be determinate.

Operationally, a discontainment s`\:`t is equivalent to `naf` $s\!:\!t$. As long as s doesn't contain variables at execution time, `naf` $s\!:\!t$ will in fact be discontainment in the initial model, provided none of the involved relations is forced to be determinate.

11 Data Bases

Data Bases are lists of canonical value terms, where additional elements can be inserted and existing elements can be deleted. Data bases are elements of so-called data base types, which are obtained by the abstract type constructor

```
    database(T) := abstract.
```

An empty data base is obtained with the procedure

```
    tproc emptydb: T:type x ?database(T).
```

The procedure

```
    tproc assert: T x database(T).
```

inserts a term at the end of a data base. The procedure

```
    proc retract: ?T x database(T).
```

deletes the first element of the given data base that unifies with the term given as first argument. The relation

```
    rel indb: ?T x database(T).
```

can be used to enumerate the elements of a data base. For instance, if you type the query

```
    TEL> emptydb(int##int, D) &
         assert(1#1, D) & assert(2#2, D) &assert(3#1, D) &
         assert(4#1, D) & retract(4#X, D) &
         Y#X indb D.
```

TEL computes the following answers:

```
    D = abstract : database(int##int)
    X = 1 : posint
    Y = 1 : posint
    more answers? (y/n) y
    D = abstract : database(int##int)
    X = 1 : posint
    Y = 3 : posint
    more answers? (y/n) y
    failed.
```

If a term in a database contains variables, the variables of the term are replaced consistently by new variables each time the term is accessed with `retract` or `indb`.

A counter can be realized as follows:

```
natcounter := database(nat).
par counter: natcounter = D <-- emptydb(nat, D) &
                                assert(0, D).


tproc increment: natcounter x ?nat.
   increment(C, N) <-- retract(N, C) & assert(N+1, C).
```

Data bases can be useful in large systems to store information that is accessed by many components of the system but is changed by few components of the system. Furthermore, data bases can be used to communicate information outside that is computed within a do-condition.

The type constructor `database` is treated differently from other abstract type constructors in the following respects:

- `database(`s`)` is a type term if and only if s is a ground type term
- if the outermost type constructor of s or t is `database`, then $s \leq t$ if and only if $s = t$.

The type checking rule for the procedure `emptydb`, which takes a type as argument, is:

- $F.\emptyset.P[\texttt{emptydb}(t,x)] = F.\emptyset.(P \cup \{x\colon \texttt{database}(t)\})$

  if t is a type and
  x is a variable not contained in $F \cup \mathcal{D}(P)$.

A Built-ins

This section lists all built-in objects of TEL.

A.1 Booleans

```
bool := {true, false}.

and: bool x bool --> bool.
or: bool x bool --> bool.
not: bool --> bool.
```

A.2 Integers

```
int := negint ++ nat.
nat := zero ++ posint.
negint := {~1, ~2, ~3, ... }.
zero := {0}.
posint := {1, 2, 3, ... }.
par minnegint : negint.
par maxposint : posint.

+ : int x int --> int,
    nat x nat --> nat,
    posint x nat --> posint,
    nat x posint --> posint,
    negint x negint --> negint.

- : int x int --> int,
    nat x negint --> posint,
    negint x nat --> negint.
```

```
~ : int --> int,
    posint --> negint,
    negint --> posint.

* : int x int --> int,
    nat x nat --> nat,
    posint x posint --> posint,
    posint x negint --> negint,
    negint x posint --> negint,
    negint x negint --> posint.

mod: int x int --> nat.

// : int x int >-> int,
     nat x nat >-> nat,
     posint x posint --> posint,
     posint x negint --> negint,
     negint x posint --> negint,
     negint x negint --> posint.

< : int x int --> bool.
=< : int x int --> bool.
> : int x int --> bool.
>= : int x int --> bool.
```

A.3 Characters

```
char := layout_char ++ alpha_char ++ symbol_char.
alpha_char := letter ++ digit ++ {"_"}.
letter := capital_letter ++ small_letter.
symbol_char := grouping_symbol ++ operator_symbol ++ {"%"}.
```

```
layout_char := {"bell", "eof", "nl",
                    " any character with ASCII-code less than 33"}.

capital_letter := {"A", "B", ... , "Z"}.
small_letter := {"a", "b", ... , "z"}.
digit := {"0", "1", ... , "9"}.

grouping_symbol := {"(", ")", "[", "]", "{", "}",
                    """", "'", ","}.

operator_symbol := {"+", "-", "*", "/", "|", "\", "^",
                    "<", ">", "`", "~", "=", ":", ".",
                    "?", "@", "#", "$", "&", "!", ";"}.

natequiv: char --> nat.
charequiv: nat >-> char.
```

A.4 Lists

```
list(T) := elist ++ nelist(T).
elist := {nil}.
nelist(T) := {. : T x list(T)}.

| : list(T) x list(T) --> list(T),
    nelist(T) x list(T) --> nelist(T),
    list(T) x nelist(T) --> nelist(T).

in: T x list(T) --> bool.

length: list(T) --> nat,
        nelist(T) --> posint.
```

A.5 Pairs

```
S##T := {# : S x T}.
```

A.6 Strings

```
string := estring ++ nestring.
estring := {''}.
nestring :=  a nonempty string starts with ', continues with at least one character,
             where ' is written as '', and ends with '


@< : string x string --> bool.
@=< : string x string --> bool.
@> : string x string --> bool.
@>= : string x string --> bool.


chartrans: string --> list(char),
           nestring --> list(char).


stringtrans: list(char) --> string,
             nelist(char) --> nestring.


^ : string x string --> string.


genstring: string x nat --> nestring.
```

A.7 Streams

```
instream(T) := abstract.
outstream(T) := abstract.
```

```
proc open_instream: string x T:type x ?instream(T).
proc open_outstream: string x T:type x ?outstream(T).
proc append_outstream: string x T:type x ?outstream(T).

tproc close_instream: instream(T).
tproc close_outstream: outstream(T).

proc get: instream(T) x ?T.
tproc put: outstream(T) x T.

par user_input: instream(char).
par user_output: outstream(char).
par user_error: outstream(char).

tproc flush: outstream(T).

lineno: instream(char) --> nat,
        outstream(char) --> nat.
charno: instream(char) --> nat,
        outstream(char) --> nat.
linepos: instream(char) --> nat,
         outstream(char) --> nat.

tproc print: outstream(char) x T.

proc parse: instream(char) x T:type x ?T.

tproc nl: outstream(char).
tproc put_string: outstream(char) x string.
tproc put_chars: outstream(char) x list(char).
```

A.8 Data Bases

```
database(T) := abstract.
```

```
tproc emptydb: T:type x ?database(T).
tproc assert: T x database(T).
proc retract: ?T x database(T).
rel indb : ?T x database(T).
```

A.9 Variable Test

```
drel var: T.
```

A.10 Unix and Quintus Access

The current implementation of TEL has two built-in procedures for accessing the UNIX operating system and the Quintus Prolog System on which TEL is running. If your system needs to use these procedures, I recommend that you use them only in a special module, say, `unix_quintus_interface`, so that it is easy to see which low level features are used by your system. Since in future implementations of TEL these two procedures may change, isolating them in a single module will make it easier to port your TEL application.

The procedure

```
proc unix: string.
```

passes its argument to a newly created UNIX shell process for execution as a shell command. The shell run depends on the current UNIX environment. If the execution of the command fails, `unix` fails.

For the Quintus Prolog access TEL supports a type prolog_term allowing to express arbitrary Prolog terms in TEL.

```
prolog_term := refl_variable ++ refl_integer ++
               {pterm: string x list(prolog_term)}.
refl_variable := {rvar: varname}.
varname := string. % must satisfy Quintus Prolog Syntax
refl_integer := {rint: int}.
```

The procedure

```
proc quintus: prolog_term x list(refl_variable) x
              ?list(refl_variable##prolog_term).
```

executes its first argument as a goal in the Quintus Prolog system on which TEL is currently running. If the execution of the goal given in the first argument succeeds, quintus returns the computed bindings for the variables given in the second argument through the third argument. If the execution of the given goal fails, quintus fails. Since quintus is a procedure, it is determinate, that is, it cannot be backtracked.

With quintus you can use all the goodies provided by Quintus Prolog. You can even define your own Prolog predicates. If you do this, use only names that (1) are not the names of Quintus Prolog built-ins, (2) do not start with tel_, which is the prefix for the predicates comprising TEL's run time system, and (3) are atoms that can be written without quotes.

For convenience, TEL has the following two procedures built-in although they can be defined with quintus. The procedure

```
tproc statistics.
statistics <-- quintus(pterm('statistics',nil), nil, _).
```

prints information about the current memory allocation and the used time on user_output. The procedure

```
    tdrel time: ?nat.
    time(T) <--  %statistics(runtime, [_,T])
       quintus(pterm('statistics',
                     pterm('runtime',nil)
                    .pterm('.',
                           rvar('_')
                           .pterm('.',
                                  rvar('T')
                                  .pterm('[]',nil)
                                  .nil)
                           .nil)
                     .nil),
                rvar('T').nil,
                _#rint(T)._).
```

returns the seconds of CPU-time used since the last call of time or statistics.

B Syntax

This section defines TEL's syntax using the following notation:

- Syntactic categories are printed slanted, for instance, *type_definition.*
- Every syntactic category is defined by a syntactic rule, which takes the form

  $$C \longrightarrow S_1|S_2|\cdots|S_n$$

  and states that the syntactic category C can take one of the alternative forms $F_1, \ldots, F_n$.
- A terminal form 'T' means that the token T must appear physically.
- An optional form [F] means that the form F is optional.
- A list form {F} means that the form F appears either once or more than once separated by commas ',' .
- A star form $(F)^*$ denotes a possibly empty sequence of Fs.

B.1 Modules

```
view ⟶
    'view' module_name '.'
    'imports' { module_name } '.'
        transfer_declaration
        (transfer_declaration)*
    'endview' '.'

interface ⟶
    'interface' module_name '.'
    [ 'imports' { module_name } '.'
      [ (transfer_declaration)*
       declaration ] ]
    (declaration)*
    'endinterface' '.'
```

module_body $\longrightarrow$
    ‘`module`’ *module_name* ‘`.`’
    (*definition*)*
    ‘`endmodule`’ ‘`.`’

module_name $\longrightarrow$
    identifier

transfer_declaration $\longrightarrow$
    ‘`from`’ *module_name* ‘`:`’ { *designator* [‘`abstract`’] } ‘`.`’

declaration $\longrightarrow$
    type_declaration
    | *parameter_declaration*
    | *function_declaration*
    | *relation_declaration*
    | *procedure_declaration*

definition $\longrightarrow$
    type_definition
    | *type_abbreviation*
    | *parameter_definition*
    | *function_definition*
    | *relation_definition*
    | *procedure_definition*

B.2 Declarations and Definitions

type_declaration $\longrightarrow$
    abstract_type_declaration
    | *type_abbreviation*
    | *type_definition*

abstract_type_declaration $\longrightarrow$
    type_dec_lhs ‘`:=`’ ‘`abstract`’ ‘`.`’

type_dec_lhs $\longrightarrow$

  identifier ['(' { *variable* } ')']

 | *prefix_operator variable*

 | *variable infix_operator variable*

 | *variable postfix_operator*

   the occurring variables must be pairwise distinct

type_abbreviation $\longrightarrow$

 type_dec_lhs ':=' *nonvariable_type_term*

  every variable that occurs in the left-hand side must occur in the right-hand side and vice versa

type_definition $\longrightarrow$

 type_dec_lhs ':=' *type_def_rhs* '.'

  every variable that occurs in the left-hand side must occur in the right-hand side and vice versa

type_def_rhs $\longrightarrow$

  (*subtype_specification* '++')*

   subtype_specification '++'

   subtype_specification

 | (*subtype_specification* '++')*

   '{' { *constructor_definition* } '}'

subtype_specification $\longrightarrow$

 nonvariable_type_term

constructor_definition $\longrightarrow$

 designator [':' *domain*]

designator $\longrightarrow$

 identifier | *operator*

domain $\longrightarrow$

 type_term ['x' *domain*]

parameter_definition $\longrightarrow$

 'par' *identifier* ':' *ground_type_term* '=' *term*

   ['<--' *condition_part*] '.'

parameter_declaration $\longrightarrow$

 'par' *identifier* ':' *ground_type_term* '.'

function_definition $\longrightarrow$
    function_declaration
        (*functional_clause*)*

function_declaration $\longrightarrow$
    designator ':' { *rank* } '.'
        all ranks must specify the same number of arguments

rank $\longrightarrow$
    domain '-->' *type_term*
    | *domain* '>->' *type_term*
        every variable occurring in the codomain of a rank must occur in the domain of the rank

relation_definition $\longrightarrow$
    relation_declaration
        (*relational_clause*)*

relation_declaration $\longrightarrow$
    rel_class designator ':' *io_domain* '.'

rel_class $\longrightarrow$
    'tdrel' | 'drel' | 'trel' | 'rel'

io_domain $\longrightarrow$
    ['?'] *type_term* ['x' *io_domain*]
        every variable occurring in the type term of an output argument must occur in the type term of an input argument

procedure_definition $\longrightarrow$
    procedure_declaration
        (*relational_clause*)*

procedure_declaration $\longrightarrow$
    proc_class designator ':' [*io_domain*] '.'

proc_class $\longrightarrow$
    'tproc' | 'proc'

B.3 Clauses

functional_clause $\longrightarrow$
    nonvariable_term '=' *term*
        ['<--' *condition_part*] '.'
    | { *term* } '|>' *term*
        ['<--' *condition_part*] '.'

```
relational_clause  ⟶
      nonvariable_term
            [ '<--'  condition_part ] '.'

condition_part  ⟶
      condition [ '&'  condition_part ]

condition  ⟶
        conditional
      | simple_condition

conditional  ⟶
      'if'  simple_conjunction 'then'  cond_condition
      ('elsif'  simple_conjunction 'then'  cond_condition)*
      [ 'else'  cond_condition 'fi' ]

simple_conjunction  ⟶
      simple_condition [ '&'  simple_conjunction ]

cond_condition  ⟶
        'succeed'
      | 'fail'
      | condition_part

simple_condition  ⟶
        term '='  term
      | term '\='  term
      | term ':'  ground_type_term
      | term '\:'  ground_type_term
      | primitive_condition
      | 'naf'  primitive_condition
      | '!'  variable
      | 'do'  primitive_condition
      | term 'islistof'  term 'where'  primitive_condition

primitive_condition  ⟶
      term
```

B.4 Terms

term ⟶
integer
| *character*
| *string*
| *variable*
| *identifier* [‘(’ { *term* } ‘)’]

there must be no character between the identifier and ‘(’

| *prefix_operator term*
| *term infix_operator term*
| *term postfix_operator*
| ‘(’ *term* ‘)’

nonvariable_term ⟶

a term that is not a variable

type_term ⟶

a term not containing integers, characters or strings

ground_type_term ⟶

a type term not containing variables

nonvariable_type_term ⟶

a type term that is not a variable

B.5 Tokens

integer ⟶
[‘~’] *natural_number*

natural_number ⟶
digit (*digit*)*

character ⟶
‘"bell"’ | ‘"eof"’ | ‘"nl"’
| ‘"0"’ | ... | ‘"9"’ | ‘"a"’ | ... | ‘"z"’ | ‘"A"’ | ... | ‘"Z"’
| ‘"_"’ | ‘" "’ | ‘"%"’
| ‘"("’ | ‘")"’ | ‘"["’ | ‘"]"’ | ‘"{"’ | ‘"}"’ | ‘""""’ | ‘"’"’ | ‘","’
| ‘"+"’ | ‘"-"’ | ‘"*"’ | ‘"/"’ | ‘"~"’ | ‘"<"’ | ‘">"’ | ‘"="’
| ‘":"’ | ‘"?"’ | ‘"!"’ | ‘";"’ | ‘"."’
| ‘"$"’ | ‘"&"’ | ‘"@"’ | ‘"#"’ | ‘"|"’ | ‘"\"’ | ‘"^"’ | ‘"‘"’

string ⟶

‘,,’

| *nonempty_string*

nonempty_string ⟶

starts with ’, contains at least one character, ’ is written as ’’, and ends with ’

variable ⟶

capital_letter (*alpha_character*)*

| *wildcard*

wildcard ⟶

‘_’

identifier ⟶

small_letter (*alpha_character*)*

must not be a *reserved_identifier* or an *operator*

alpha_character ⟶

digit | *capital_letter* | *letter* | ‘_’

layout_token ⟶

comment

| any nonempty sequence of ASCII characters with code ≤32

comment ⟶

starts with % and ends with newline

end_of_sentence_token ⟶

a period ‘.’ followed by an ASCII character with code ≤32

operator ⟶

prefix_operator | *infix_operator* | *postfix_operator*

prefix_operator ⟶

user_defined_prefix_operator

| ‘not’ precedence 900

| ‘~’ precedence 200

infix_operator $\longrightarrow$

user_defined_infix_operator

| `indb`   precedence 1200

| `and`   precedence 1100, right-associative

| `or`   precedence 1000, right-associative

| `in`   precedence 900

| `<` | `=<` | `>` | `>=`   precedence 900

| `|`   precedence 800, right-associative

| `.`   precedence 700, right-associative

| `#`   precedence 600, right-associative

| `+`   precedence 500, right-associative

| `-`   precedence 500, left-associative

| `*`   precedence 400, right-associative

| `//` | `mod`   precedence 400

| `##`   precedence 300, right-associative

| `@<` | `@=<` | `@>` | `@>=`   precedence 200

| `^`   precedence 100, right-associative

postfix_operator $\longrightarrow$

user_defined_postfix_operator

reserved_identifier $\longrightarrow$

`interface` | `endinterface` | `module` | `endmodule` | `view`
| `endview` | `imports` | `from` | `abstract` | `par`
| `rel` | `drel` | `tdrel` | `trel` | `proc` | `tproc` | `do` | `naf`
| `if` | `then` | `elsif` | `else` | `fi` | `succeed` | `fail`
| `islistof` | `where` | `void`

To see how operators are parsed, consider the text

```
X + 5 + ~4 - 7 - 6* ~Y
```

which is parsed as the term

```
((X + (5 + ~4)) - 7) - (6 * (~Y)).
```

B.6 User-defined Operators

When TEL is invoked, it looks in the current working directory for a file `myoperators`. If this file exists, TEL will treat the operators defined in it just as it treats the built-in operators. The file `myoperators` must have the following format:

```
my_operators  ⟶
    (operator_definition)*

operator_definition  ⟶
      'prefix' operator_text precedence [ 'right' ] '.'
    | 'infix' operator_text precedence [ associativity ] '.'
    | 'postfix' operator_text precedence [ 'left' ] '.'

operator_text  ⟶
      identifier
          must not be a reserved_identifier or a built-in operator
    | any nonempty sequence of the characters
              + - * / ~ < > = : ? ! ; . $ & @ # | \ ^ `
          but not a built-in operator or any of the following:
              := ++ --> >-> |> <-- & ! = \= : \:.

precedence  ⟶
    natural_number

associativity  ⟶
    'left' | 'right'
```

C Manager Commands

After you have invoked the TEL system, TEL's manager prints the prompt `TEL>` and waits for your input. You can enter commands or queries. Commands are used, among other things, to request that a module be edited, compiled or opened. Queries request TEL computations and are type checked and executed in the environment defined by the local signature of the module currently opened. If no module is opened, the signature consisting of all built-in objects is taken as environment. The manager accepts *functional queries*, which consist of a term not containing variables, and *relational queries*, which have the same form as clause bodies.

Commands start with the character # and end with a period followed by the newline key. Here are the commands available in our implementation:

`#halt`. Ends the TEL session.

`#help`. Lists the available commands.

`#show_definition d ...` . Prints the definitions of the designators `d,...` in the current environment.

`#show_module m ...` . Prints information about the modules `m,...` .

`#show_system`. Prints an alphabetical list of all known modules. For instance, the first line of the list

```
C I  B   abstract_syntax_and_table  %1
  I (B)  backend  %7
  V      backend_import
L I  B   frontend  %8
  V      frontend_import

module abstract_syntax_and_table is opened
```

says that the module `abstract_syntax_and_table` is consulted, its interface and body have been compiled successfully, and the Prolog code for its objects is disambiguated with the prefix `%1`. The second line says that the interface of the module backend has been compiled successfully, while the draft of the body of backend has not been compiled successfully. The third line says that the view backend_import has been compiled successfully. The fourth line says that the interface and body of the module frontend have both been compiled successfully and that the module is loaded.

#edit_interface m. Creates an edit window for the interface of module m. If the interface has been compiled successfully, the manager asks whether you want the compilation of the interface to be retracted. If a compilation is retracted, the compilation of all dependent module components is retracted. Don't forget to save the editor buffer after you have finished editing, otherwise TEL won't be able to access the interface file.

#edit_body m. Creates an edit window for the body of module m.

#edit_view m. Creates an edit window for the view m.

#delete m. Deletes the module (interface and body) or view m.

#compile_interface m. First, the compilation of all module components depending on m is retracted. Then the compilation of the interface of module m is attempted. If the interface of an imported module or an imported view has not been compiled successfully so far, its compilation is attempted recursively.

#compile_body m. Attempts the compilation of the body of module m. If m is loaded or consulted and the compilation turns out to be successful, the manager asks whether you want m to be reloaded or reconsulted.

#compile_view m. First, the compilation of all module components depending on m is retracted. Then the compilation of the view m is attempted. If the interface of an imported module or an imported view has not been compiled successfully so far, its compilation is attempted recursively.

#open m. Attempts to open the module m. If the body of m has not been compiled successfully so far, its compilation is attempted. If debugging mode is on, m is consulted rather than loaded.

#show_switches. Prints the settings of the switches of the TEL system. The default settings are:

```
noise        2    (1, 2, 3, 4, 5)
time         off  (on, off)
types        off  (on, off)
debug        off  (on, off)
print_depth  30   (1, 2, 3, ... )
```

The switch noise determines how much TEL tells you about what it is doing. If the switch time is on, TEL tells you how much CPU seconds it needs for its actions. If the switch types is on, TEL prints the types it infers for variables when it type checks clauses. If the switch debug is on, TEL is in debugging mode. The switch print_depth determines up to which depth TEL prints terms that appear as answers to queries.

#switch s v. Sets switch s to value v.

#save f. Saves the current state of the TEL system in a file f. You can restart the TEL system in this state by typing f to the UNIX shell.

#generate f m p. Generates a user system on file f using the nullary total procedure p defined in module m as start-up procedure. You can start the generated system by typing f to the UNIX shell. The generated system contains TEL's run-time system but not its manager and compiler.

#spy o ... . Sets spypoints on the objects o,... and turns the debugging mode on. An object in the module currently opened is specified by its designator, while an object d in another module m is specified by m:d. Spypoints can be put on functions, relations and procedures. If you enter a query and debugging mode is on, execution stops at every spypoint and the relevant information is printed. You will be quite amazed at first since you are actually debugging the Prolog code generated by TEL using the excellent Quintus Prolog debugger. Don't worry, this works quite well in practice although it may not seem so. Before you start debugging TEL programs, you better get acquainted with the Quintus Prolog debugger.

#nospy o ... . Removes the spypoints from the objects o,... .

#nospyall. Removes all spypoints.

#show_spypoints. Prints all existing spypoints.

#consult m ... . Consults the modules m,... , which must have been compiled successfully. If a module is loaded, its Prolog code is compiled, while the Prolog code is interpreted if the module is consulted. Interpreted Prolog code is much slower than compiled Prolog code, but the Prolog debugger can do much more with interpreted code. If debugging mode is on, the open command consults rather than loads the requested module.

#deconsult m ... . Deconsults and loads the modules m,... , which must be consulted currently.

#prolog. Starts a Prolog break shell from which you can return to TEL.

D Limitations of the Current Implementation

Our current implementation of TEL Version 0.9 has the following limitations (in order of their significance):

- Open variables cannot be constrained to subtypes. This is due to the fact that for reasons of effiency TEL's typed unification is mapped more or less directly to Prolog's untyped unification.
- No subtype of the built-in types `char` and `string` (including `char` and `string`) can be a subtype of a user-defined type. This limitation is due to the fact that characters are implemented as Prolog numbers and strings are implemented as Prolog atoms. Without this limitation, integers could not be distinguished from characters and nullary value constructors could not be distinguished from strings.
- Relations declared with `trel` don't produce a run-time error if they fail to yield at least one answer. We don't know how to implement this feature efficiently in Prolog.
- User-defined operators are not implemented (yet).
- Of course, TEL inherits all limitations of Quintus Prolog.