

Chapter 15

Static typing, polymorphism and type synthesis

We now want to perform static typechecking of ASL programs, that is, to complete typechecking *before* evaluation, making run-time type tests unnecessary during evaluation of ASL programs.

Furthermore, we want to have *polymorphism* (i.e. allow the identity function, for example, to be applicable to any kind of data).

Type synthesis may be seen as a game. When learning a game, we must:

- learn the rules (what is allowed, and what is forbidden);
- learn a winning strategy.

In type synthesis, the rules of the game are called a *type system*, and the winning strategy is the typechecking algorithm.

In the following sections, we give the ASL type system, the algorithm and an implementation of that algorithm. Most of this presentation is borrowed from [7].

15.1 The type system

We study in this section a type system for the ASL language. Then, we present an algorithm performing the type synthesis of ASL programs, and its Caml Light implementation. Because of subtle aspects of the notation used (inference rules), and since some important mathematical notions, such as unification of first-order terms, are not presented here, this chapter may seem obscure at first reading.

The type system we will consider for ASL has been first given by Milner [27] for a subset of the ML language (in fact, a superset of λ -calculus). A *type* is either:

- the type `Number`;
- or a type variable (α, β, \dots) ;
- or $\tau_1 \rightarrow \tau_2$, where τ_1 and τ_2 are types.

In a type, a type variable is an *unknown*, i.e. a type that we are computing. We will use $\tau, \tau', \tau_1, \dots$, as *metavariables*¹ representing types. This notation is important: we shall use other greek letters to denote other notions in the following sections.

Example $(\alpha \rightarrow \text{Number}) \rightarrow \beta \rightarrow \beta$ is a type.

□

A *type scheme*, is a type where some variables are distinguished as being *generic*. We can represent type schemes by:

$$\forall \alpha_1, \dots, \alpha_n. \tau \text{ where } \tau \text{ is a type.}$$

Example $\forall \alpha. (\alpha \rightarrow \text{Number}) \rightarrow \beta \rightarrow \beta$ and $(\alpha \rightarrow \text{Number}) \rightarrow \beta \rightarrow \beta$ are type schemes.

□

We will use $\sigma, \sigma', \sigma_1, \dots$, as metavariables representing type schemes. We may also write type schemes as $\forall \vec{\alpha}. \tau$. In this case, $\vec{\alpha}$ represent a (possibly empty) set of generic type variables. When the set of generic variables is empty, we write $\forall. \tau$ or simply τ .

We will write $FV(\sigma)$ for the set of *unknowns* occurring in the type scheme σ . Unknowns are also called *free variables* (they are not bound by a \forall quantifier).

We also write $BV(\sigma)$ (*bound type variables of σ*) for the set of type variables occurring in σ which are not free (i.e. the set of variables universally quantified). Bound type variables are also said to be *generic*.

Example If σ denotes the type scheme $\forall \alpha. (\alpha \rightarrow \text{Number}) \rightarrow \beta \rightarrow \beta$, then we have:

$$FV(\sigma) = \{\beta\}$$

and

$$BV(\sigma) = \{\alpha\}$$

□

A *substitution instance* σ' of a type scheme σ is the type scheme $S(\sigma)$ where S is a substitution of types for *free* type variables appearing in σ . When applying a substitution to a type scheme, a renaming of some bound type variables may become necessary, in order to avoid the capture of a free type variable by a quantifier.

Example

- If σ denotes $\forall \beta. (\beta \rightarrow \alpha) \rightarrow \alpha$ and σ' is $\forall \beta. (\beta \rightarrow (\gamma \rightarrow \gamma)) \rightarrow (\gamma \rightarrow \gamma)$, then σ' is a substitution instance of σ because $\sigma' = S(\sigma)$ where $S = \{\alpha \leftarrow (\gamma \rightarrow \gamma)\}$, i.e. S substitutes the type $\gamma \rightarrow \gamma$ for the variable α .
- If σ denotes $\forall \beta. (\beta \rightarrow \alpha) \rightarrow \alpha$ and σ' is $\forall \delta. (\delta \rightarrow (\beta \rightarrow \beta)) \rightarrow (\beta \rightarrow \beta)$, then σ' is a substitution instance of σ because $\sigma' = S(\sigma)$ where $S = \{\alpha \leftarrow (\beta \rightarrow \beta)\}$. In this case, the renaming of β into δ was necessary: we did not want the variable β introduced by S to be captured by the universal quantification $\forall \beta$.

¹A metavariable should not be confused with a *variable* or a *type variable*.

□

The type scheme $\sigma' = \forall\beta_1 \dots \beta_m.\tau'$ is said to be a *generic instance* of $\sigma = \forall\alpha_1 \dots \alpha_n.\tau$ if there exists a substitution S such that:

- the domain of S is included in $\{\alpha_1, \dots, \alpha_n\}$;
- $\tau' = S(\tau)$;
- no β_i occurs free in σ .

In other words, a generic instance of a type scheme is obtained by giving more precise values to some generic variables, and (possibly) quantifying some of the new type variables introduced.

Example If $\sigma = \forall\beta.(\beta \rightarrow \alpha) \rightarrow \alpha$, then $\sigma' = \forall\gamma.((\gamma \rightarrow \gamma) \rightarrow \alpha) \rightarrow \alpha$ is a generic instance of σ . We changed β into $(\gamma \rightarrow \gamma)$, and we universally quantified on the newly introduced type variable γ .

□

We express this type system by means of *inference rules*. An inference rule is written as a fraction:

- the numerator is called the *premisses*;
- the denominator is called the *conclusion*.

An inference rule:

$$\frac{P_1 \dots P_n}{C}$$

may be read in two ways:

- “**If** P_1, \dots **and** P_n , **then** C ”.
- “**In order to prove** C , **it is sufficient to prove** P_1, \dots **and** P_n ”.

An inference rule may have no premise: such a rule will be called an *axiom*. A complete proof will be represented by a *proof tree* of the following form:

$$\frac{\frac{P_1^m \dots \dots \dots P_l^k}{\vdots} \dots \dots}{\frac{P_1^1 \dots P_n^1}{C}}$$

where the leaves of the tree (P_1^m, \dots, P_l^k) are instances of axioms.

In the premisses and the conclusions appear *judgements* having the form:

$$\Gamma \vdash e : \sigma$$

Such a judgement should be read as “under the typing environment Γ , the expression e has type scheme σ ”. Typing environments are sets of *typing hypotheses* of the form $x : \sigma$ where x is an identifier name and σ is a type scheme: typing environments give types to the variables occurring free (i.e. unbound) in the expression.

When typing λ -calculus terms, the typing environment is managed as a *stack* (because identifiers possess local scopes). We represent that fact in the presentation of the type system by *removing* the typing hypothesis concerning an identifier name x (if such a typing hypothesis exists) before adding a new typing hypothesis concerning x .

We write $\Gamma - \Gamma(x)$ for the set of typing hypotheses obtained from Γ by removing the typing hypothesis concerning x (if it exists).

Any numeric constant is of type `Number`:

$$\frac{}{\Gamma \vdash \text{Const } n : \text{Number}} \quad (\text{NUM})$$

We obtain type schemes for variables from the typing environment Γ :

$$\frac{}{\Gamma \cup \{x : \sigma\} \vdash \text{Var } x : \sigma} \quad (\text{TAUT})$$

It is possible to instantiate type schemes. The “GenInstance” relation represents generic instantiation.

$$\frac{\Gamma \vdash e : \sigma \quad \sigma' = \text{GenInstance}(\sigma)}{\Gamma \vdash e : \sigma'} \quad (\text{INST})$$

It is possible to generalize type schemes with respect to variables that do not occur free in the set of hypotheses:

$$\frac{\Gamma \vdash e : \sigma \quad \alpha \notin FV(\Gamma)}{\Gamma \vdash e : \forall \alpha. \sigma} \quad (\text{GEN})$$

Typing a conditional:

$$\frac{\Gamma \vdash e_1 : \text{Number} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash (\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \text{ fi}) : \tau} \quad (\text{IF})$$

Typing an application:

$$\frac{\Gamma \vdash e_1 : \tau \rightarrow \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash (e_1 \ e_2) : \tau'} \quad (\text{APP})$$

Typing an abstraction:

$$\frac{(\Gamma - \Gamma(x)) \cup \{x : \tau\} \vdash e : \tau'}{\Gamma \vdash (\lambda x . e) : \tau \rightarrow \tau'} \quad (\text{ABS})$$

The special rule below is the one that introduces polymorphism: this corresponds to the ML `let` construct.

$$\frac{\Gamma \vdash e : \sigma \quad (\Gamma - \Gamma(x)) \cup \{x : \sigma\} \vdash e' : \tau}{\Gamma \vdash (\lambda x . e') \ e : \tau} \quad (\text{LET})$$

This type system has been proven to be *semantically sound*, i.e. the semantic value of a well-typed expression (an expression admitting a type) cannot be an *error value* due to a type error. This is usually expressed as:

Well-typed programs cannot go wrong.

This fact implies that a clever compiler may produce code without any dynamic type test for a well-typed expression.

Example Let us check, using the set of rules above, that the following is true:

$$\emptyset \vdash \text{let } f = \lambda x.x \text{ in } f \ f : \beta \rightarrow \beta$$

In order to do so, we will use the equivalence between the **let** construct and an application of an immediate abstraction (i.e. an expression having the following shape: $(\lambda v.M)N$). The (LET) rule will be crucial: without it, we could not check the judgement above.

$$\frac{\frac{\frac{}{\{x : \alpha\} \vdash x : \alpha} \text{(TAUT)}}{\emptyset \vdash (\lambda x.x) : \alpha \rightarrow \alpha} \text{(ABS)} \quad \frac{\frac{}{\Gamma \vdash f : \forall \alpha. \alpha \rightarrow \alpha} \text{(TAUT)}}{\Gamma \vdash f : (\beta \rightarrow \beta) \rightarrow (\beta \rightarrow \beta)} \text{(INST)} \quad \frac{\frac{}{\Gamma \vdash f : \forall \alpha. \alpha \rightarrow \alpha} \text{(TAUT)}}{\Gamma \vdash f : \beta \rightarrow \beta} \text{(INST)}}{\frac{\emptyset \vdash (\lambda x.x) : \forall \alpha. \alpha \rightarrow \alpha \text{(GEN)} \quad \frac{\Gamma \vdash f : (\beta \rightarrow \beta) \rightarrow (\beta \rightarrow \beta) \quad \Gamma \vdash f : \beta \rightarrow \beta}{\Gamma = \{f : \forall \alpha. \alpha \rightarrow \alpha\} \vdash f \ f : \beta \rightarrow \beta} \text{(APP)}}{\emptyset \vdash \text{let } f = \lambda x.x \text{ in } f \ f : \beta \rightarrow \beta} \text{(LET)}$$

□

This type system does not tell us how to find the best type for an expression. But what is the best type for an expression? It must be such that any other possible type for that expression is more specific; in other words, the best type is the *most general*.

15.2 The algorithm

How do we find the most general type for an expression of our language? The problem with the set of rules above, is that we could instantiate and generalize types at any time, introducing type schemes, while the most important rules (application and abstraction) used only types.

Let us write a new set of inference rules that we will read as an algorithm (close to a Prolog program):

Any numeric constant is of type Number:

$$\frac{}{\Gamma \vdash \text{Const } n : \text{Number}} \text{(NUM)}$$

The types of identifiers are obtained by taking generic instances of type schemes appearing in the typing environment. These generic instances will be *types* and not type schemes: this restriction appears in the rule below, where the type τ is expected to be a generic instance of the type scheme σ .

As it is presented (belonging to a deduction system), the following rule will have to anticipate the effect of the equality constraints between types in the other rules (multiple occurrences of a type metavariable), when choosing the instance τ .

$$\frac{\tau = \text{GenInstance}(\sigma)}{\Gamma \cup \{x : \sigma\} \vdash \text{Var } x : \tau} \text{(INST)}$$

When we read this set of inference rules as an algorithm, the (INST) rule will be implemented by:

1. taking as τ the “most general generic instance” of σ that is a type (the rule requires τ to be a type and not a type scheme),

2. making τ more specific by *unification* [32] when encountering equality constraints.

Typing a conditional requires only the test part to be of type `Number`, and both alternatives to be of the same type τ . This is an example of equality constraint between the types of two expressions.

$$\frac{\Gamma \vdash e_1 : \text{Number} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash (\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \text{ fi}) : \tau} \quad (\text{COND})$$

Typing an application produces also equality constraints that are to be solved by unification:

$$\frac{\Gamma \vdash e_1 : \tau \rightarrow \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash (e_1 \ e_2) : \tau'} \quad (\text{APP})$$

Typing an abstraction “pushes” a typing hypothesis for the abstracted identifier: unification will make it more precise during the typing of the abstraction body:

$$\frac{(\Gamma - \Gamma(x)) \cup \{x : \forall \tau. \tau\} \vdash e : \tau'}{\Gamma \vdash (\lambda x . e) : \tau \rightarrow \tau'} \quad (\text{ABS})$$

Typing a `let` construct involves a generalization step: we generalize as much as possible.

$$\frac{\Gamma \vdash e : \tau' \quad \{\alpha_1, \dots, \alpha_n\} = FV(\tau') - FV(\Gamma) \quad (\Gamma - \Gamma(x)) \cup \{x : \forall \alpha_1 \dots \alpha_n. \tau'\} \vdash e' : \tau}{\Gamma \vdash (\lambda x . e') e : \tau} \quad (\text{LET})$$

This set of inference rules represents an algorithm because there is exactly one conclusion for each syntactic ASL construct (giving priority to the (LET) rule over the regular application rule). This set of rules may be read as a Prolog program.

This algorithm has been proven to be:

- *syntactically sound*: when the algorithm succeeds on an expression e and returns a type τ , then $e : \tau$.
- *complete*: if an expression e possesses a type τ , then the algorithm will find a type τ' such that τ is an instance of τ' . The returned type τ' is thus the most general type of e .

Example We compute the type that we simply checked in our last example. What is drawn below is the result of the type synthesis: in fact, we run our algorithm with type variables representing unknowns, modifying the previous applications of the (INST) rule when necessary (i.e. when encountering an equality constraint). This is valid, since it can be proved that the correction of the whole deduction tree is preserved by substitution of types for type variables. In a real implementation of the algorithm, the data structures representing types will be submitted to a unification mechanism.

$$\frac{\frac{\frac{}{\{x : \alpha\} \vdash x : \alpha} \text{(INST)}}{\emptyset \vdash (\lambda x.x) : \alpha \rightarrow \alpha} \text{(ABS)} \quad \frac{\frac{}{\Gamma \vdash f : (\beta \rightarrow \beta) \rightarrow (\beta \rightarrow \beta)} \text{(INST)} \quad \frac{}{\Gamma \vdash f : \beta \rightarrow \beta} \text{(INST)}}{\Gamma = \{f : \forall \alpha. \alpha \rightarrow \alpha\} \vdash f \ f : \beta \rightarrow \beta} \text{(APP)}}{\emptyset \vdash \text{let } f = \lambda x.x \text{ in } f \ f : \beta \rightarrow \beta} \text{(LET)}$$

Once again, this expression is not typable without the use of the (LET) rule: an error would occur because of the type equality constraints between all occurrences of a variable bound by a “ λ ”. In an effective implementation, a unification error would occur.

□

We may notice from the example above that the algorithm is *syntax-directed*: since, for a given expression, a type deduction for that expression uses exactly one rule per sub-expression, the deduction possesses the same structure as the expression. We can thus reconstruct the ASL expression from its type deduction tree. From the deduction tree above, if we write upper rules as being “arguments” of the ones below and if we annotate the applications of the (INST) and (ABS) rules by the name of the subject variable, we obtain:

$$\text{LET}_f(\text{ABS}_x(\text{INST}_x), \text{APP}(\text{INST}_f, \text{INST}_f))$$

This is an illustration of the “types-as-propositions and programs-as-proofs” paradigm, also known as the “Curry-Howard isomorphism” (cf. [14]). In this example, we can see the type of the considered expression as a proposition and the expression itself as the proof, and, indeed, we recognize the expression as the deduction tree.

15.3 The ASL type-synthesizer

We now implement the set of inference rules given above.

We need:

- a Caml representation of ASL types and type schemes,
- a management of type environments,
- a unification procedure,
- a typing algorithm.

15.3.1 Representation of ASL types and type schemes

We first need to define a Caml type for our ASL type data structure:

```
#type asl_type = Unknown
#           | Number
#           | TypeVar of vartype
#           | Arrow of asl_type * asl_type
#and vartype = {Index:int; mutable Value:asl_type}
#and asl_type_scheme = Forall of int list * asl_type ;;
Type asl_type defined.
Type vartype defined.
Type asl_type_scheme defined.
```

The `Unknown` ASL type is not really a type: it is the initial value of fresh ASL type variables. We will consider as abnormal a situation where `Unknown` appears in place of a regular ASL type. In such situations, we will raise the following exception:

```
#exception TypingBug of string;;
Exception TypingBug defined.
```

Type variables are allocated by the `new_vartype` function, and their global counter (a local reference) is reset by `reset_vartypes`.

```
#let new_vartype, reset_vartypes =
#(* Generating and resetting unknowns *)
#   let counter = ref 0
#   in (function () -> counter := !counter + 1;
#       {Index = !counter; Value = Unknown}),
#       (function () -> counter := 0);;
new_vartype : unit -> vartype = <fun>
reset_vartypes : unit -> unit = <fun>
```

15.3.2 Destructive unification of ASL types

We will need to “shorten” type variables: since they are indirections to ASL types, we need to follow these indirections in order to obtain the type that they represent. For the sake of efficiency, we take advantage of this operation to replace multiple indirections by single indirections (shortening).

```
#let rec shorten t =
#   match t with
#   | TypeVar {Index=_; Value=Unknown} -> t
#   | TypeVar ({Index=_;
#               Value=TypeVar {Index=_;
#                               Value=Unknown} as tv}) -> tv
#   | TypeVar ({Index=_; Value=TypeVar tv1} as tv2)
#     -> tv2.Value <- tv1.Value; shorten t
#   | TypeVar {Index=_; Value=t'} -> t'
#   | Unknown -> raise (TypingBug "shorten")
#   | t' -> t';;
shorten : asl_type -> asl_type = <fun>
```

An ASL type error will be represented by the following exception:

```
#exception TypeClash of asl_type * asl_type;;
Exception TypeClash defined.
```

We will need unification on ASL types with *occur-check*. The following function implements occur-check:

```
#let occurs {Index=n;Value=_} =
#   let rec occrec = function
#     TypeVar{Index=m;Value=_} -> (n=m)
#     | Number -> false
#     | Arrow(t1,t2) -> (occrec t1) or (occrec t2)
#     | Unknown -> raise (TypingBug "occurs")
#   in occrec
#;;
occurs : vartype -> asl_type -> bool = <fun>
```

The unification function: implements destructive unification. Instead of returning the most general unifier, it returns the unificand of two types (their most general common instance). The two arguments are physically modified in order to represent the same type. The unification function will detect type clashes.

```
#let rec unify (tau1,tau2) =
# match (shorten tau1, shorten tau2)
# with (* type variable n and type variable m *)
#   (TypeVar({Index=n; Value=Unknown} as tv1) as t1),
#   (TypeVar({Index=m; Value=Unknown} as tv2) as t2)
#   -> if n <> m then tv1.Value <- t2
# | (* type t1 and type variable *)
#   t1, (TypeVar ({Index=_;Value=Unknown} as tv) as t2)
#   -> if not(occurs tv t1) then tv.Value <- t1
#       else raise (TypeClash (t1,t2))
# | (* type variable and type t2 *)
#   (TypeVar ({Index=_;Value=Unknown} as tv) as t1), t2
#   -> if not(occurs tv t2) then tv.Value <- t2
#       else raise (TypeClash (t1,t2))
# | Number, Number -> ()
# | Arrow(t1,t2), (Arrow(t'1,t'2) as t)
#   -> unify(t1,t'1); unify(t2,t'2)
# | (t1,t2) -> raise (TypeClash (t1,t2));;
unify : asl_type * asl_type -> unit = <fun>
```

15.3.3 Representation of typing environments

We use `asl_type_scheme` list as typing environments, and we will use the encoding of variables as indices into the environment.

The initial environment is a list of types (`Number -> (Number -> Number)`), which are the types of the ASL primitive functions.

```
#let init_typing_env =
# map (function s ->
#   forall([],Arrow(Number,
#     Arrow(Number,Number))))
#   init_env;;
```

The global typing environment is initialized to the initial typing environment, and will be updated with the type of each ASL declaration, after they are type-checked.

```
#let global_typing_env = ref init_typing_env;;
```

15.3.4 From types to type schemes: generalization

In order to implement generalization, we will need some functions collecting types variables occurring in ASL types.

The following function computes the list of type variables of its argument.

```

#let vars_of_type tau =
# let rec vars vs = function
#   Number -> vs
#   | TypeVar {Index=n; Value=Unknown}
#       -> if mem n vs then vs else n::vs
#   | TypeVar {Index=_; Value= t} -> vars vs t
#   | Arrow(t1,t2) -> vars (vars vs t1) t2
#   | Unknown -> raise (TypingBug "vars_of_type")
# in vars [] tau
#;;
vars_of_type : asl_type -> int list = <fun>

```

The `unknowns_of_type(bv,t)` application returns the list of variables occurring in `t` that do not appear in `bv`. The `subtract` function returns the difference of two lists.

```

#let unknowns_of_type (bv,t) =
#   subtract (vars_of_type t) bv;;
unknowns_of_type : int list * asl_type -> int list = <fun>

```

We need to compute the list of unknowns of a type environment for the generalization process (unknowns belonging to that list cannot become generic). The set of unknowns of a type environment is the union of the unknowns of each type. The `flat` function flattens a list of lists.

```

#let flat = it_list (prefix @) [];;
flat : 'a list list -> 'a list = <fun>

#let unknowns_of_type_env env =
#   flat (map (function Forall(gv,t) -> unknowns_of_type (gv,t)) env);;
unknowns_of_type_env : asl_type_scheme list -> int list = <fun>

```

The generalization of a type is relative to a typing environment. The `make_set` function eliminates duplicates in its list argument.

```

#let rec make_set = function
#   [] -> []
#   | x::l -> if mem x l then make_set l else x :: make_set l;;
make_set : 'a list -> 'a list = <fun>

#let generalise_type (gamma, tau) =
#   let genvars =
#       make_set (subtract (vars_of_type tau)
#                         (unknowns_of_type_env gamma))
#   in Forall(genvars, tau)
#;;
generalise_type : asl_type_scheme list * asl_type -> asl_type_scheme = <fun>

```

15.3.5 From type schemes to types: generic instantiation

The following function returns a generic instance of its type scheme argument. A generic instance is obtained by replacing all generic type variables by new unknowns:

```
#let gen_instance (Forall(gv,tau)) =
# (* We associate a new unknown to each generic variable *)
# let unknowns = map (function n -> n, TypeVar(new_vartype())) gv in
# let rec ginstance = function
#   (TypeVar {Index=n; Value=Unknown} as t) ->
#     (try assoc n unknowns
#      with Not_found -> t)
#   | TypeVar {Index=_; Value= t} -> ginstance t
#   | Number -> Number
#   | Arrow(t1,t2) -> Arrow(ginstance t1, ginstance t2)
#   | Unknown -> raise (TypingBug "gen_instance")
# in ginstance tau
#;;
gen_instance : asl_type_scheme -> asl_type = <fun>
```

15.3.6 The ASL type synthesizer

The type synthesizer is the `asl_typing_expr` function. Each of its match cases corresponds to an inference rule given above.

```
#let rec asl_typing_expr gamma =
# let rec type_rec = function
#   Const _ -> Number
#   | Var n ->
#     let sigma =
#       try nth n gamma
#       with Failure _ -> raise (TypingBug "Unbound")
#     in gen_instance sigma
#   | Cond (e1,e2,e3) ->
#     unify(Number, type_rec e1);
#     let t2 = type_rec e2 and t3 = type_rec e3
#     in unify(t2, t3); t3
#   | App((Abs(x,e2) as f), e1) -> (* LET case *)
#     let t1 = type_rec e1 in
#       let sigma = generalise_type (gamma,t1)
#       in asl_typing_expr (sigma::gamma) e2
#   | App(e1,e2) ->
#     let u = TypeVar(new_vartype())
#     in unify(type_rec e1,Arrow(type_rec e2,u)); u
#   | Abs(x,e) ->
#     let u = TypeVar(new_vartype()) in
```

```
#      let s = Forall([],u)
#      in Arrow(u,asl_typing_expr (s::gamma) e)
# in type_rec;;
asl_typing_expr : asl_type_scheme list -> asl -> asl_type = <fun>
```

15.3.7 Typing, trapping type clashes and printing ASL types

Now, we define some auxiliary functions in order to build a “good-looking” type synthesizer. First of all, a printing routine for ASL type schemes is defined (using a function `tvar_name` which computes a decent name for type variables).

```
#let tvar_name n =
# (* Computes a name "'a", ... for type variables, *)
# (* given an integer n representing the position *)
# (* of the type variable in the list of generic *)
# (* type variables *)
# let rec name_of n =
#   let q,r = (n / 26), (n mod 26) in
#   let s = make_string 1 (char_of_int (96+r)) in
#   if q=0 then s else (name_of q)^s
# in ""^(name_of n)
#;;
tvar_name : int -> string = <fun>
```

Then a printing function for type schemes.

```
#let print_type_scheme (Forall(gv,t)) =
# (* Prints a type scheme. *)
# (* Fails when it encounters an unknown *)
# let names = let rec names_of = function
#   (n,[]) -> []
#   | (n,(v1::Lv)) -> (tvar_name n)::(names_of (n+1, Lv))
#   in names_of (1,gv) in
# let tvar_names = combine (rev gv, names) in
# let rec print_rec = function
#   TypeVar{Index=n; Value=Unknown} ->
#     let name = try assoc n tvar_names
#       with Not_found ->
#         raise (TypingBug "Non generic variable")
#     in print_string name
#   | TypeVar{Index=_;Value=t} -> print_rec t
#   | Number -> print_string "Number"
#   | Arrow(t1,t2) ->
#     print_string "("; print_rec t1;
#     print_string " -> "; print_rec t2;
#     print_string ")"
```

```
#   | Unknown -> raise (TypingBug "print_type_scheme")
# in print_rec t
#;;
Toplevel input:
>           | (n,(v1::Lv)) -> (tvar_name n)::(names_of (n+1, Lv))
>           ^^^
Warning: the variable Lv starts with an upper case letter in this pattern.
print_type_scheme : asl_type_scheme -> unit = <fun>
```

Now, the main function which resets the type variables counter, calls the type synthesizer, traps ASL type clashes and prints the resulting types. At the end, the global environments are updated.

```
#let typing (Decl(s,e)) =
# reset_vartypes();
# let tau = (* TYPING *)
#   try asl_typing_expr !global_typing_env e
#   with TypeClash(t1,t2) -> (* A typing error *)
#     let vars=vars_of_type(t1)@vars_of_type(t2) in
#     print_string "ASL Type clash between ";
#     print_type_scheme (Forall(vars,t1));
#     print_string " and ";
#     print_type_scheme (Forall(vars,t2));
#     print_newline();
#     raise (Failure "ASL typing") in
# let sigma = generalise_type (!global_typing_env,tau) in
# (* UPDATING ENVIRONMENTS *)
# global_env := s::!global_env;
# global_typing_env := sigma::!global_typing_env;
# reset_vartypes ();
# (* PRINTING RESULTING TYPE *)
# print_string "ASL Type of ";
# print_string s;
# print_string " is ";
# print_type_scheme sigma; print_newline();;
typing : top_asl -> unit = <fun>
```

15.3.8 Typing ASL programs

We reinitialize the parsing environment:

```
#global_env:=init_env; ();;
- : unit = ()
```

Now, let us run some examples through the ASL type checker:

```
#typing (parse_top "let x be 1;");;
ASL Type of x is Number
```

```

- : unit = ()
#typing (parse_top "+ 2 ((\x.x) 3);");;
ASL Type of it is Number
- : unit = ()
#typing (parse_top "if + 0 1 then 1 else 0 fi;");;
ASL Type of it is Number
- : unit = ()
#typing (parse_top "let id be (\x.x);");;
ASL Type of id is ('a -> 'a)
- : unit = ()
#typing (parse_top "+ (id 1) (id id 2);");;
ASL Type of it is Number
- : unit = ()
#typing (parse_top "let f be (\x.x x) (\x.x);");;
ASL Type of f is ('a -> 'a)
- : unit = ()
#typing (parse_top "+ (\x.x) 1;");;
ASL Type clash between Number and ('a -> 'a)
Uncaught exception: Failure "ASL typing"

```

15.3.9 Typing and recursion

The Z fixpoint combinator does not have a type in Milner's type system:

```

#typing (parse_top
# "let fix be \f.((\x.f(\z.(x x)z)) (\x.f(\z.(x x)z)));");;
ASL Type clash between 'a and ('a -> 'b)
Uncaught exception: Failure "ASL typing"

```

This is because we try to apply x to itself, and the type of x is not polymorphic. In fact, no fixpoint combinator is typable in ASL. This is why we need a special primitive or syntactic construct in order to express recursivity.

If we want to assign types to recursive programs, we have to predefine the Z fixpoint combinator. Its type scheme should be $\forall\alpha.((\alpha \rightarrow \alpha) \rightarrow \alpha)$, because we take fixpoints of functions.

```

#global_env := "fix"::init_env;
#global_typing_env:=
# (Forall([1],
# Arrow(Arrow(TypeVar{Index=1;Value=Unknown},
# TypeVar{Index=1;Value=Unknown}),
# TypeVar{Index=1;Value=Unknown})))
# ::init_typing_env;
#();;
- : unit = ()

```

We can now define our favorite functions as:

```
#typing (parse_top
#   "let fact be fix (\\f.(\\n. if = n 0 then 1
#                       else * n (f (- n 1))
#                       fi));");;
ASL Type of fact is (Number -> Number)
- : unit = ()

#typing (parse_top "fact 8;");;
ASL Type of it is Number
- : unit = ()

#typing (parse_top
#   "let fib be fix (\\f.(\\n. if = n 1 then 1
#                       else if = n 2 then 1
#                               else + (f(- n 1)) (f(- n 2))
#                               fi
#                       fi));");;
ASL Type of fib is (Number -> Number)
- : unit = ()

#typing (parse_top "fib 9;");;
ASL Type of it is Number
- : unit = ()
```

