Type Reconstruction

Recall that by a *type system* we mean a set of rules and mechanisms used in a programming language to organize, build and handle the types accepted in the language. We have seen how types are built in Caml and in the Typed Lambda Calculus. This section discusses additional aspects of a type system: type inference, type equivalence and type compatibility. Caml is the anchor of the discussion.

Type inference

In strongly typed functional programming languages types need not be explicitly declared. They are automatically inferred. The synthesis process of the type of an expression is based on the types of the component elements of the expression and on the lexical context the expression is part of. Since a type is represented as an expression build up with constants (predefined types), type variables and type constructors (as those illustrated in figure1), the synthesis process can be seen as combining a set of type expressions according to precise rules and conventions. These rules and conventions together with the type constants and the type constructors in the language make the main part of the type system of the language.

Type constructor	*Arity*	*Precedence*	*Position*	*Associativity*
Sum type constructor	≥ **0**	**highest**	**Postfix**	**Left**
list (List)	**1**	**|**	**Postfix**	**Left**
***** (Cartesian product)	> **1**	**|**	**Infix**	**-**
-> (Function)	**2**	**lowest**	**Infix**	**Right**

Table 1. The type constructors in Caml

A *type schema* is a type expression that can contain bound type variables and free type variables. A *free type variable* from within a type schema can be substituted once only by a unique type. Afterwards it stays bound to that type. A *bound type variable* stands for a generic type that can be substituted any number of times by any type. There are two equivalent notations for describing a type schema: one - at the abstract level - that is used to explain how the type inference works, the other - at the language level - that is used to encode a type schema in a format accessible to the programmer.

- At the abstract level we represent the type variables by Greek letters different than τ and σ. A type schema that may contain bound variables is designated by σ whereas a type schema that does not contain bound variables is designated by τ. The set of type schemas of kind σ includes the set of schemas of kind τ. A type schema that contains free type variables is *weak polymorphic*. A type schema that contains bound type variables only is *strong polymorphic*. For example, the type schema τ **= β-> int -> β*int** is weak polymorphic, whereas σ **=** $\forall\alpha$**.(**α **->** α **list)** is a strong polymorphic type schema. The quantifier $\forall$ binds type variables in a type schema. It works as λ does in λ-expressions. For instance, in the type schema $\forall\alpha$**.(**α*****β **->** β*****α**)** the variable α is bound whereas β is free.

- At the language level the bound type variables are represented as **'*identifier*** whereas free type variables are represented as **'_*identifier***, where ***identifier*** is a valid identifier in the language. Moreover, the binding quantifier $\forall$ is dropped from the encoding of a type schema. For instance, the abstract type schemas $\forall\alpha$**.(**α **->** α **list)** and **β -> int -> β*int** may be encoded at the language level as **'a -> 'a list** and,

respectively, **'_a -> int-> '_a*int**. The abstract type schema ∀α.(α*β -> β*α), which contains both free and bound variables, could be encoded as **'a*'_b -> '_b*'a** at the language level.

The *typing environment* of an expression is noted Γ. It designates the set of the program identifiers that exist in the lexical scope of the expression and the types associated with these identifiers. The pair **x:σ**, where **x** is an identifier and **σ** is a type schema, is called a typing hypothesis. The typing environment contains typing hypothesis.

In statically scoped languages (such as Caml) the typing environment is organized as a stack. It is modified each time the type inference process enters or exits a referential context. The LIFO modification of Γ is described using two operations. The operation of removing all the typing hypotheses of a variable **x** from Γ is represented by $\Gamma\text{-}\Gamma_x$. Adding a typing hypothesis **x:σ** to Γ is represented by $\Gamma \cup \{x:\sigma\}$. The notation $(\Gamma\text{-}\Gamma_x) \cup \{x:\sigma\}$ stands for pushing the typing hypothesis **{x:σ}** at the top of the Γ stack, seen here as a set. In addition, the notation **FV(σ)** designates the set of the free type variables from the type schema **σ**, whereas **FV(Γ)** is the set of the free type variables from the typing hypotheses contained in Γ.

Notice that during the process of type inference we are talking of various kind of expressions: first, there is the expression to be typed; second, there are type schemas. In addition, there are different kind of variables: on one hand the variables from the expression to be typed (i.e. program variables); on the other hand type variables, bound and free. It is very important to distinguish between these objects of different kind. Moreover, when we say that a type variable **x** is bound we may understand either that **x** appears in a type schema ∀**x . ...**, or that **x** is bound to a type schema.

Type inference rules and operations

Type schemas are derived according to *type inference rules*. A type inference rule has the format

$$\frac{P_1\ P_2\ .\ .\ .\ P_n}{C}\ \textit{rule\_id}$$

The elements P_i are the premises of the rule whereas **C** is the conclusion of the rule. Usually, premises have the form $\Gamma \vdash E_i:\sigma_i$ and read: in the typing environment Γ the expression E_i has the type σ_i. The format of **C** is similar. The ***rule_id*** is the identifier of the rule, usually the identifier of an operator form the language. The rule reads: considering that the premises are true, the conclusion of the rule is true. A procedural interpretation of the rule is: to prove the conclusion of the rule, the premises of the rule have to be proved. An inference rule with no premises is an *axiom*.

The inference process of the type of an expression unfolds as a proof tree. Each inference rule in the tree takes as premises the conclusions of rules placed higher in the tree. Each operator and predefined expression of the language is associated with a type inference rule. For instance, the type of the expression **1+2** can be derived as follows:

$$\dfrac{\dfrac{}{\Gamma \vdash 1:int}\ Const_C \qquad \dfrac{}{\Gamma \vdash 2:int}\ Const_C}{\Gamma \vdash 1+2 : int}\ +$$

Apart from synthesis rules as above, type synthesis uses additional operations on type schemas: type instantiation, type generalization and type unification.

Type unification

Unifying two type schemas τ and τ' means to compute the (most general) substitution $S=\{\tau_1/\alpha_1,\tau_2/\alpha_2,...,\tau_n/\alpha_n\}$ for the type variables $\alpha_1,\alpha_2,...,\alpha_n$ within τ and τ' such that after performing the substitution of α_i by τ_i within both τ and τ', τ and τ' are the same expression. For example, assume that τ = (α * β list) and τ'= (int * γ). We can take the substitution S={int/α, β list/γ}. Subject to the substitution S the type schemas σ and σ' are the same, i.e. $\tau/S \equiv \tau'/S \equiv$ int * β list. Observe that type variables may be bound to type schemas as result of unification.

The unification complicates when the type schemas contain bound variables. Both schemas are first instantiated and then unified. For example, unifying $\sigma = \forall\alpha\ \beta.(\alpha * \beta\ \text{list})$ with $\tau = \forall\delta.(\delta * \gamma)$ leads to the unification of (α' * β' list) with (δ' * γ) and produces the substitution S = {δ'/α', β' list/γ}. The substitutions σ/S and τ/S are ∀δ'.(δ' * β' list).

Since the unification of two types computes the most general substitution of those types, the type inference process will usually find the most general type of an expression. Such a type is called the principal type of the expression.

Unification is NP-complete and therefore one may expect the process of type inference to be extremely slow. In practice this seldom happens. On the other hand, unification can produce cyclic results. In the unification process α *unify*(S) α->β the type variable α must be bound to the type α->β which now stands for (((α...->β)->β)->β)->β, where α... designates the process of non-terminating substitutions of α by α->β. Such type schemas are not allowed in the type system. The unification algorithm must perform a so called *occurrence check* in order to avoid producing cyclic objects and for avoiding the non termination of the unification itself. The occurrence check searches for the occurrences of a variable within the value bound to that variable as result of the unification process.

Type instantiation and generalization

Type instantiation and generalization use the concept of *local typing environment* of an expression and the concept of *local type variable*. Consider the typing process of an expression E and let Γ' designate the typing context when the typing process starts. All the new typing hypotheses pushed onto Γ' during the typing of E designate the local typing environment Γ_E of E. Γ_E contains the typing hypotheses for the variables local (or bound) in E. Therefore, just before the end of typing E the overall typing environment is $\Gamma = \Gamma' \cup \Gamma_E$. For instance, the typing of E = fun x -> (x,y) from within let y = [] in fun x -> (x,y) produces the typing environment $\Gamma = \Gamma' \cup \Gamma_E$, where Γ_E = {x:β}, Γ' = {y: ∀α. α list,...}.

A type variable α is *local to the typing of an expression* E if the following constraints apply:

1. $\alpha \in FV(\Gamma_E)$, ie. α is a free (non universally quantified) type variable in the type hypotheses that are in Γ_E.
2. α is unbound or it is bound to a type schema whose free type variables are local to the typing of E. Such bindings may result from the process of type unification performed while typing E.

3. There is no type variable $\beta \in \Gamma$ and $\beta \notin \Gamma_E$ that is bound, directly or indirectly, to a type schema that contains α.

Type instantiation. Let $E:\sigma$, $\sigma = \forall\alpha_1\alpha_2...\alpha_n \ . \ T$, designate an already typed value used in the typing environment Γ. The instantiation of the type schema σ computes a weak polymorphic type schema $\tau = T_{[...\alpha_i / \beta_i...]}$, $\beta_i \notin \Gamma$, $i=1,n$, by replacing the binding variables $\alpha_1, \alpha_2,...,\alpha_n$ by new type variables (variables that are not in $FV(\Gamma)$).

Type generalization. Let $E:\sigma$, $\sigma = \forall\alpha_1\alpha_2...\alpha_n \ . \ T$, designate a value typed in the typing environment Γ. The generalization of the type schema σ computes a schema $\sigma' = \forall\alpha_1\alpha_2...\alpha_n \ \beta_1\beta_2... \ \beta_q \ . \ T$ by universally quantifying the type variables β_j from σ that are local to the typing of E.

Examples of type inference rules

The following inference rules comply somewhat with the type system of Caml 0.73. These rules use the type instantiation operation implicitly. Whenever the type of a value used in the premise of a rule is denoted by $v:\tau$ it means that type instantiation has been applied to v.

$$\frac{}{\Gamma \vdash c:int}\ Const_c$$

The rule above is an axiom template. Provided that $c \in int$ it stands for an infinity of axioms, one for each integer constant c. It simply says that an integer constant has the type int in any typing environment. Similar axioms exist for the constants of other types in the language.

$$\frac{}{(\Gamma - \Gamma x) \cup \{x:\alpha\} \vdash x:\alpha}\ var$$

The rule var is an axiom for asserting typing hypotheses for variables. It asserts that in a typing context Γ, in which all the typing hypotheses of a variable x are replaced by a new typing hypothesis $\{x:\alpha\}$, the variable x has the type specified in the hypothesis. In addition it is considered that $\alpha \notin FV(\Gamma)$.

$$\frac{\Gamma \vdash P:\tau_p \quad \Gamma \vdash E_1:\sigma \quad \Gamma \vdash E_2:\sigma' \quad \Gamma \vdash \sigma \ unify(S) \ \sigma' \quad \Gamma \vdash \tau_p \ unify(S_p) \ bool}{\Gamma/S \cup S_p \vdash (\text{if } P \text{ then } E_1 \text{ else } E_2):\sigma/S \cup S_p}\ if$$

The if rule above types a conditional expression. Observe that both expressions E_1 and E_2 must have types that *unify*. The premise $\Gamma \vdash \sigma \ unify(S) \ \sigma'$ says that type schemas σ and σ' need not be identical but must *unify* according to the substitution S.

$$\frac{\Gamma \vdash E_1:\tau_1 \quad \Gamma \vdash E_2:\tau_2 \quad \Gamma \vdash \{\tau_1,\tau_2\} \ unify(S) \ int}{\Gamma/S \vdash E_1 \ \# \ E_2:int}\ \#$$

The rule # is a template for rules used to type integer arithmetic expressions. Here # stands for an arithmetic operator (+, -, *, /, ...). The notation $\{\tau_1,\tau_2\} \ unify \ (S) \ int$ states that both

type schemas τ_1 and τ_2 must unify with **int** and the resulting substitution is **S**. Rules of the same kind exist for other predefined types in the language.

$$\frac{\Gamma \vdash E_1:\tau_1\text{->}\tau \qquad \Gamma \vdash E_2:\tau_2 \qquad \Gamma \vdash \tau_1 \ unify(S)\ \tau_2}{\Gamma/S \vdash (E_1\ E_2):\tau/S}\ \textbf{app}$$

The **app** rule types a function application. Since the type schemas of E_1 and E_2 are implicitly instantiated and, therefore, turned into weak polymorphic type schemas, the following results can be explained.

```
let pair = fun x -> fun y -> (x,y);;
pair : 'a -> 'b -> 'a*'b = <fun>   ➔strong polymorphic

let s = pair 1;;
s : '_a -> int*'_a = <fun>         ➔weak polymorphic

s 2;;
- : int*int = 1,2                  ➔monomorphic

s;;
s : int -> int*int = <fun>         ➔monomorphic
```

Although the signature of **pair** is strong polymorphic, the application **(pair 1)** generates a weak polymorphic function. The type variable **'_a** is free in the signature of **s**. When first particularized to a type **'_a** will remain bound to that type. When the type of the application **(s 2)** is inferred the variable **'_a** is bound to **int**. The type of **s** freezes to **int -> int*int**.

$$\frac{(\Gamma\text{-}\Gamma_X)\cup\{x:\tau\} \vdash E:\sigma \qquad \alpha_1,\alpha_2,...,\alpha_n \textit{ in } \tau\text{->}\sigma \textit{ are local to the typing of } \textbf{fun x->E}}{\Gamma \vdash (fun\ x\ \text{->}\ E):\forall\alpha_1\alpha_2...\alpha_n.(\tau\text{->}\sigma)}\ \textbf{fun}$$

The **fun** rule corresponds to the typing of a functional abstraction. The rule tries to generalize as much as possible the type schema of a function by turning to bound variables all free type variables that are *local to the typing* of the function.

The rule reads: if we can prove that in the typing environment Γ extended with the typing hypothesis $x:\tau$ the type of the expression **E** is σ and $\alpha_1,\alpha_2,...,\alpha_n$ are local type variables, then we can infer that the type of the expression **fun x -> E** is described by the type schema $\forall\alpha_1\alpha_2...\alpha_n.(\tau\text{->}\sigma)$. Observe that if the type variables $\alpha_1,\alpha_2,...,\alpha_n$ are the only free type variables in τ and σ then the resulting function is strong plymorphic. The rule can be used only if the formal parameter of the function is a variable.

Applying type inference rules

As a first example of type inference consider the typing of the small program above. The type inference for **fun x -> fun y -> (x,y)** goes as illustrated in the following proof tree, where the type variables α and β are not in **FV(Γ)**. This property is general: all free type variables created in a given typing environment are *new*.

$$\dfrac{\dfrac{\dfrac{}{(\Gamma\text{-}\Gamma_x)\cup\{x:\alpha\} \vdash x:\alpha}\ \textbf{var}}{(((\Gamma\text{-}\Gamma_x)\cup\{x:\alpha\})\text{-}\Gamma_y)\cup\{y:\beta\} \vdash y:\beta}\ \textbf{var}}{(((\Gamma\text{-}\Gamma_x)\cup\{x:\alpha\})\text{-}\Gamma_y)\cup\{y:\beta\} \vdash (x,y): \alpha*\beta \qquad \beta \textit{ is local}}\ \textbf{(\_,\_)}$$

$$\dfrac{(\Gamma\text{-}\Gamma_x)\cup\{x:\alpha\} \vdash \textbf{fun y -> (x,y)}:\forall\beta.(\beta\text{->}\alpha*\beta) \qquad \alpha \textit{ is local}}{\Gamma \vdash \textbf{fun x ->fun y ->(x,y)}:\forall\alpha.(\alpha\text{-> }\forall\beta.(\beta\text{->}\alpha*\beta))}\ \textbf{fun}\ \textbf{fun}$$

The Caml encoding of the type schema is **'a -> ('b -> ('a*'b))**. Taking into account the precedence and the associativity of the type constructors the type schema can be rewritten **'a->'b->'a*'b**. Finally, the expression **let pair = fun x -> fun y -> (x,y)** extends Γ with the typing hypothesis for **pair**. Note $\Gamma'= \Gamma\text{-}\Gamma_{\textbf{pair}}\cup\{\textbf{pair}:\forall\alpha.(\alpha\text{->}\forall\beta.(\beta\text{->}\alpha*\beta))\}$.

The typing of the application **(pair 1)** is described by the inference tree below. Recall that, as a program variable, **pair** is bound to the value **fun x ->fun y ->(x,y)**.

$$\dfrac{\dfrac{}{\Gamma' \vdash \textbf{1:int}}\ \textbf{Const}_1 \qquad \Gamma' \vdash \textbf{pair}:\alpha\text{->}\beta\text{->}\alpha*\beta \qquad \Gamma' \vdash \alpha \textit{ unify}\textbf{(S) int}}{\Gamma'/S \vdash \textbf{(pair 1): }\beta\textbf{->int*}\beta \textit{ and } \textbf{S=\{int/}\alpha\textbf{\}} \textit{ and } \textbf{x:int}}\ \textbf{app}$$

Notice that the type schema of the variable **pair** is modified. All bound type variables within the type schema are turned into new free variables, not present in Γ'. In addition, observe that the type schema of the parameter **x** (i.e. the type variable α) unifies with the type schema **int** of the argument of the function. The unifying substitution is **S={int/α}**.

The result of the inference process above is **(pair 1):β->int*β**. The type schema is encoded in Caml as **'_a->int*'_a**. The result of the application **(pair 1)** is the weak polymorphic function **fun y ->(1,y): β->int*β**. Notice that **fun y ->(1,y)** is a functional closure which holds the binding **x:int←1**. The closure mechanism complicates the inference process since, when applied, the typing environment of the application must be extended with the typing saved within the closure. To avoid this problem, here we consider that **x** is substituted by its corresponding value in the body of the function.

Finally, as the effect of the expression **let = pair 1**, the typing hypothesis **s: β->int*β** is inserted in the typing environment $\underline{\Gamma}'= \Gamma'/\{\textbf{int/}\alpha\}$, and the type variable β is registered as a free type variable in the resulting typing environment. Note $\Gamma''= \underline{\Gamma}'\text{-}\underline{\Gamma}'_{\textbf{s}}\cup\{\textbf{s:}\beta\textbf{->int*}\beta\}$. The value of **s** is **fun y ->(1,y)**.

The type inference for the application **(s 2)** is trivial. However, it has an important side effect.

$$\dfrac{\dfrac{}{\Gamma'' \vdash \textbf{2: int}}\ \textbf{Const}_2 \qquad \Gamma'' \vdash \textbf{\{s:}\beta\textbf{->int*}\beta\textbf{\}} \qquad \Gamma'' \vdash \beta \textit{ unify}\textbf{(S) int}}{\Gamma''/S \vdash \textbf{(s 2):int*int} \textit{ and } \textbf{S=\{int/}\beta\textbf{\}} \textit{ and } \textbf{y:int}}\ \textbf{app}$$

The value of **(s 2)** is **(1,2)**. Since the type variable **β** is free in **Γ"**, the side effect of binding **β** to the type **int** will be preserved. The type schema of **s** is changed to **int->int*int** and, therefore, the type schemas of all the expressions using **s** may indirectly be changed. That sort of side effect, due to the way the weak polymophism is tackled, may be difficult to control in Caml programs. Some functional programming languages, such as Haskell and Standard ML, simply rule out the weak polymorphism.

What could be the cure if we want a strong typed function resulted from the application **(pair 1)**? It is simple: we have to build a functional abstraction.

```
let s = fun z -> (pair 1) z;;
s : 'a -> int*'a = <fun>
```

The explanation of the strong typed signature of **s** is found in the following inference tree.

$$\dfrac{\dfrac{}{\Gamma' \vdash 1:int}\ Const_1 \qquad \Gamma' \vdash \{pair:\alpha\text{->}\beta\text{->}\alpha*\beta\}^{1} \qquad \Gamma' \vdash \alpha\ unify(S)\ int}{\Gamma'/S \vdash (pair\ 1):\ \beta\text{->}int*\beta \quad and \quad S=\{int/\alpha\}}\ app \qquad \dfrac{}{(\Gamma'/S\text{-}\Gamma'_z)\cup\{z:\beta\} \vdash z:\beta}\ var$$

$$\dfrac{(\Gamma'/S\text{-}\Gamma'_z)\cup\{z:\beta\} \vdash ((pair\ 1)\ z):int*\beta \qquad \beta\ is\ local}{\Gamma'/S \vdash fun\ z\text{->}\ ((pair\ 1)\ z):\ \forall\beta.(\beta\text{->}int*\beta)}\ fun\ \ app$$

The value bound to the variable **s** is the function **fun z -> ((fun y -> (1,y)) z)** and it is strong polymorphic. The application **(s 2)** will not have any side effect on the signature of **s**.

An example of type inference

We consider the type synthesis of a fixed-point combinator for writing **1**-ary recursive functions.

```
type 'a T = FUN of 'a T -> 'a;;

let Fix =
    fun f -> ((fun (FUN g) -> fun x -> f (g (FUN g)) x)
              (FUN (fun (FUN g) -> fun x -> f (g (FUN g)) x)));;
Fix: (('a->'b)->'a->'b)->'a->'b = <fun>
```

For simplifying the inference tree, we use abbreviating notations for the typing environments corresponding to the variables that appear within the definition of **Fix**:

- Γ^{f} is the typing environment of **f**;
- Γ^{g} is the typing environment of **g**;
- Γ^{x} is the typing environment of **x**;

[1] **α** and **β** are not contained in **FV(Γ')**. They are *new* free type variables.

Moreover, we consider that the structure of the type schema for the variables **f** and **g** is already known:

- **f** is a curried function with two parameters **f:α->β->γ**;
- **g** is a function that takes an argument of type **δT** and returns a result of type **δ**.

Therefore, we skip the initial part of the type inference tree that derives the typing hypotheses for the variables **f**, **g**, and **x** and create the nested typing environments $\Gamma^{x} \subset \Gamma^{g} \subset \Gamma^{f}$.

Γ^{f} **= (Γ-Γ$_{f}$) ∪{f:α->β->γ}**

Γ^{g} **= (Γ^{f}-Γ$_{g}$)∪{g:δT->δ}**

Γ^{x} **= (Γ^{g}-Γ$_{x}$)∪{x:μ}**

As an additional simplification, we write $\alpha \leftarrow \tau$ to show that - as a result of the unification process - the type variable α is bound to the type τ. In addition, the alteration of a typing context Γ by a unification process is not mentioned explicitly. Therefore, instead of **Γ/S**, where **S** is a substitution, we will write Γ.

$$\dfrac{\dfrac{\dfrac{}{\Gamma^{x} \vdash g: \delta T \to \delta}}{\Gamma^{x} \vdash FUN\ g: \delta T} \quad \Gamma^{x} \vdash g: \delta T \to \delta \quad \delta T\ unify\ \delta T}{\Gamma^{x} \vdash g\ (FUN\ g): \delta}\ app$$

$$\dfrac{\Gamma^{x} \vdash f: \alpha \to \beta \to \gamma \quad \Gamma^{x} \vdash x: \mu \quad \Gamma^{x} \vdash \alpha\ unify\ \delta \quad \Gamma^{x} \vdash \beta\ unify\ \mu}{\Gamma^{x} \vdash (f\ (g\ FUN\ g)\ x): \gamma \quad and\ \delta \leftarrow \alpha,\ \mu \leftarrow \beta}\ app$$

$$\dfrac{\Gamma^{x} \vdash x: \beta \quad (\alpha,\beta,\gamma \text{ are non local to } \Gamma^{g} \text{ and cannot be generalized})}{\Gamma^{g} \vdash (fun\ x \to f\ (g\ (FUN\ g))\ x): \beta \to \gamma}\ fun$$

$$\dfrac{\Gamma^{g} \vdash FUN\ g: \alpha T \quad (\alpha,\beta,\gamma \text{ are non local to } \Gamma^{g} \text{ and cannot be generalized})}{\Gamma^{f} \vdash fun\ (FUN\ g) \to fun\ x \to f\ (g\ (FUN\ g))\ x: \alpha T \to \beta \to \gamma}\ fun$$

$$\dfrac{\Gamma^{f} \vdash fun\ (FUN\ g) \to fun\ x \to f\ (g\ (FUN\ g))\ x: \alpha T \to \beta \to \gamma \quad \Gamma^{f} \vdash \alpha\ unify\ \beta \to \gamma}{\Gamma^{f} \vdash FUN(fun\ (FUN\ g) \to fun\ x \to f\ (g\ (FUN\ g))\ x): (\beta \to \gamma)T \quad and \quad \alpha \leftarrow \beta \to \gamma}\ T$$

$$\dfrac{\Gamma^{f} \vdash fun\ (FUN\ g) \to fun\ x \to f\ (g\ (FUN\ g)\ x: (\beta \to \gamma)T \to \beta \to \gamma}{\Gamma^{f} \vdash ((fun\ (FUN\ g) \to fun\ x \to f\ (g\ (FUN\ g))\ x)\ (FUN(fun\ (FUN\ g) \to fun\ x \to f\ (g\ (FUN\ g))\ x))): \beta \to \gamma}\ app$$

$$\dfrac{\Gamma^{f} \vdash f: (\beta \to \gamma) \to \beta \to \gamma \quad \beta,\gamma \text{ are local to } \Gamma^{f} \text{ and can be generalized}}{\Gamma \vdash fun\ f \to ((fun\ (FUN\ g) \to fun\ x \to f\ (g\ (FUN\ g))\ x)\ (FUN\ (fun\ (FUN\ g) \to fun\ x \to f\ (g\ (FUN\ g))\ x))): \forall \beta,\gamma.(((\beta \to \gamma) \to \beta \to \gamma) \to \beta \to \gamma)}\ fun$$

The type schema $\forall \beta,\gamma.(((\beta \to \gamma) \to \beta \to \gamma) \to \beta \to \gamma)$ is encoded at the language level by the type expression **(('a->'b)->'a->'b)->'a->'b**.

Type equivalence

Deciding on type equivalence is to determine when types associated to different language constructs are the same. This can be done in two ways:

- *Structural equivalence* considers that two types are equivalent if they are built in the same way from equivalent types. The terminal point of the recursion resides in the equivalence of simple, predefined types.

- *Name based equivalence* considers that two types are equivalent if they have the same name or alias names. If the alias types are considered distinct the equivalence is *strict*; in the case alias types are considered the same the equivalence is *loose*.

Since a type carry a meaning, and the name of the type usually reflect that meaning, name based type equivalence seems more reasonable than structural equivalence. It is used in most commercial languages, including C++ and Java.

In functional languages such as ML or Caml, both forms of type equivalence are used. For instance, in Caml the named types with alias names are considered equivalent. The record types are equivalent if they contain the same fields, regardless of the order in which they appear in the records. In addition, two types are considered equivalent if they have the same type schema (they have the same structure and contents) although they might have different meanings.

Since type names can be optionally declared, the user has the freedom to choose one or the other way to decide the equivalence of types. For example, consider that the radiation of a source over a time interval **[1,n]** depends on the temperature of the source and on its distance from the measuring point, according to the formula:

$$radiation = \frac{1}{n} \sum_{i=1}^{n} \frac{temp_i}{dist_i^2}.$$

In the process of encoding the radiation formula we can work with specific type names, according to the meaning of the values that occur within the formula.

```
type Distance = float
and  Temperature = float;;

type Evolution == (Distance * Temperature) list;;

let (radiation: Evolution -> Temperature) =
fun e -> let rec sum =
             fun [] -> 0.0
               | ((dist,temp)::rest)-> (temp/.(dist *. dist)) +. (sum rest)
         in
            if e = [] then 0.0
                      else (sum e)/.(float_of_int (list_length e));;
radiation: Evolution -> Temperature = <fun>

let r = radiation [(1.0,2.0); (3.3,4.3); (5.0,6.0)];;
r: Temperature = 0.878285889195
```

The types **Distance** and **Temperature** are aliases of **float**. As far as we process a **Temperature** with functions that explicitly work with the type **Temperature** the alias is seen in the system echo. However, since loose name equivalence is used in Caml, we can use functions that work with the type **float** to process temperatures. The problem is that in this case the alias **Temperature** is discarded in favor of **float** and the meaning of the value computed is lost.

```
let max t1 t2 = if t1 >. t2 then t1 else t2;;
max: float -> float -> float = <fun>

max r r;;
-: float = 0.878285889195
(* The type of (max r r) should be a Temperature rather than a float *)
```

Type equivalence has no perfect solution. A compiler should be equally smart as the programmer to be able to reason about the equivalence taking into account the meaning of the processed values. Indeed, a program is an encoding of an algorithm that, in turn, is the end result of reasoning about solving a problem. The meaning of the problem objects is an essential part of this reasoning process. It is used to prove that specific combinations and transformations of the problem objects lead to meaningful results. The reasoning process that ends with an algorithm is not captured in the algorithm.

An algorithm cannot analyze its own correctness, including termination. It simply works as planned. This implicitly means that the compiler of a language used to encode the algorithm is not able to decide rightfully on type equivalence simply because the necessary information is not part of the program. While types are explicit at the level of programs the rationale behind the way they are used is in the mind of the programmer. From this point of view the type equivalence rules used in the language cannot always fit the type equivalence rationale in the mind of the programmer.

Stricter type systems, of the sort Pascal is providing, avoid most of type equivalence traps by simplifying the issue, but they deny the power and convenience of generic processing templates. On the other hand, languages with more "flexible" type systems that support generic processing are subject to type equivalence pitfalls as in the example above. It is hard to say if a value of type **(float * float) list** can be rightfully accepted by a function with the signature **(Distance * Temperature) list -> Temperature** without knowing the meaning of **float**, **Temperature**, and **Distance** and the meaning of the processing itself.

Type compatibility

Checking type compatibility means to find out if a value of a given type can be correctly used in a given processing context. Type compatibility is not always reduced to type equivalence. Depending on the language it may happen that values with different types qualify for a given monomorphic operation. In Pascal, for example, **integer** and **real** are distinct types yet integers can be used with real arithmetic operators.

In a language where types can be inferred, type compatibility is implicitly absorbed into type inference process. If a unique type can be synthesized for an expression it means that the types of the expression elements are compatible. Here, the term unique type means a type that satisfies given criteria, for example the most general type. In other languages type compatibility is subject to specific rules. To ensure type compatibility the solutions mentioned below can be adopted.

- Explicit type conversion, using type casts. A type cast can be:
  - Non-converting, if the purpose of the conversion is to interpret the underlying machine representation of a value of a type **`T`** as the representation of a value of a different type **`T'`**. For example, a non-converting cast in C is **`(float *)calloc(n, sizeof(float))`**. In ANSI C **`calloc`** returns a **`void *`** which needs be taken as a pointer to whatever value the space is allocated for. There is no code generated to modify the machine representation of the pointer.
  - Converting, if the purpose is to convert a value of a type **`T`** into a value of a type **`T'`** which has a different underlying machine representation. For instance, consider the C declarations: **`int n=1; double r=(double)n;`** The cast **`(double)n`** generates code for converting the integer value of **`n`** to the equivalent value of type **`double`**.
- Automatic type conversion called *coercion*. For example, in Pascal if the operands of a real arithmetic operation are integers they are automatically converted to floating point.

Automatic type conversions can sometimes prove dangerous, since the programmer may get something else than expected. There are languages, such as Caml, which rule out coercion. The safer, explicit type conversions are adopted. Indeed, in a strong type system such that of Caml, explicit casts are validated by the type inference engine. In more permissive languages, such as C, the programmer can fairly well shoot into his legs by using faulty casts as in the program below.

```
#include <stdio.h>

int main() {
    double z = 3; int x = 1; double y = 2;

    printf("x=%d, y=%lf, z=%lf\n",x,y,z);
    *(double*)&x = y;
    printf("x=%d, y=%lf, z=%lf\n",x,y,z);
    return 0;
}
```

The number of bytes used to hold a **`double`** is greater than the number of bytes necessary to hold an integer. If the variables **`x`**, **`y`** and **`z`** are allocated contiguously, then storing a **`double`** into the location of **`x`**, and pretending **`x`** is a **`double`**, implicitly overwrites part of the store allocated to **`y`** or **`z`**. This may be the happy case, since program code could be overwritten as well. Indeed, using the Borland 3.1 compiler, the output of the program is:

```
x=1, y=2.000000, z=3.000000
x=0, y=2.000000, z=3.031250
```