Typing of Functional Programs

Christian Sternagel

January 18, 2019

> *A type system is the most cost effective unit test you'll ever have.*
>
> Peter Hallam

There are two important tasks concerning types in a functional language: type checking and type inference.

Type checking is the process of verifying given constraints on types and may either occur at compile-time (static type checking) or at run-time (dynamic type checking). Since with static type checking type correctness is already established during the compilation process, there is no need to store any type information in the running program. Haskell uses static type checking, hence this is also what is discussed in the following.

Type inference is the process of computing a (most general) type for a given expression. A language where types are inferred automatically (like Haskell) makes some programming tasks easier. For example, types of variables need not be declared explicitly, but still type safety is maintained.[1]

Before giving the details of type checking and type inference, some typed language is needed. Two obvious choices would be λ-calculus extended by types (also called *simply typed lambda-calculus*) and Haskell itself. Since the former is inconvenient to use and the latter is more complex than necessary, a mixture of both is considered.

1 Core FP

The language used to demonstrate type checking and type inference, residing somewhere between λ-calculus and Haskell, is called *core FP*. Its expressions (e) are defined by the

[1] A program is *type safe* if a certain class of errors—namely *type errors*—is prevented by the compiler. An example of a type error would be the application of a list length function to an integer.

following BNF grammar

$$\begin{array}{rcll} e & ::= & x \mid e\ e \mid \lambda x.\, e & (\lambda\text{-terms}) \\ & \mid & c & \\ & \mid & \textbf{let}\ x = e\ \textbf{in}\ e & \\ & \mid & \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e & \end{array}$$

where *constants* (c) stand for primitives like True, False, $<$, $>$, $=$, $\times$, $+$, $\div$, $-$, 0, 1, Pair, fst, snd, Nil, Cons, head, tail, ... (think "predefined constants and functions").

2 Type Checking

Before going into type checking, some formal definitions are required. In the following a type τ is of the form

$$\tau ::= \alpha \mid \tau \to \tau \mid C(\tau, \ldots, \tau)$$

where α (α_0, α_1, ...) is a *type variable*, '$\to$' is the (right-associative) *function space constructor* (in the end just a special case of the next construct), and C (C_1, C_2, ...) an arbitrary type constructor (like List for the type of lists). Every type constructor has a fixed *arity*, that is, the number of arguments it takes. For example, List is unary, hence applications of the type constructor for lists are of the form List(α), List(Int), List(Bool), etc. Note that the base types (Bool, Int, ...) are just a special case of type constructors, namely those of arity zero (that is, without arguments). Instead of Bool() or Int(), as indicated by the BNF grammar, such *nullary* type constructors are written without (), like Bool, Int.

A *(typing) environment* is a set of pairs, mapping variables and constants to types. Instead of (e, τ) these pairs are written $e :: \tau$, denoting "*e has type* τ." For example, the typing environment where the variable x is of type Bool and the variable y of type List(α), is written

$$\{x :: \mathsf{Bool}, y :: \mathsf{List}(\alpha)\}.$$

In the following, let

$$P = \{\mathsf{True} :: \mathsf{Bool}, \mathsf{False} :: \mathsf{Bool}, + :: \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int}, 0 :: \mathsf{Int}, 1 :: \mathsf{Int}, \mathsf{Nil} :: \mathsf{List}(\alpha), \ldots\}$$

denote the *primitive (typing) environment* (containing type information for every primitive constant).

A *(type) substitution* σ is a mapping from type variables to types such that only for finitely many type variables α we have $\sigma(\alpha) \neq \alpha$. Therefore, type substitutions may be represented by sets of bindings $\{\alpha_0 \mapsto \tau_0, \ldots, \alpha_n \mapsto \tau_n\}$. The application of a substitution σ to a type τ (written $\tau\sigma$) is defined by

$$\begin{aligned} \alpha\sigma &= \sigma(\alpha) \\ (\tau_1 \to \tau_2)\sigma &= \tau_1\sigma \to \tau_2\sigma \\ C(\tau_1, \ldots, \tau_n)\sigma &= C(\tau_1\sigma, \ldots, \tau_n\sigma) \end{aligned}$$

$$\frac{e :: \tau \in E}{e :: \tau\sigma}\ (\mathsf{ins}) \qquad \frac{e_1 :: \tau_2 \to \tau_1 \qquad e_2 :: \tau_2}{e_1\ e_2 :: \tau_1}\ (\mathsf{app})$$

$$\frac{\boxed{\begin{array}{c} x :: \tau_1 \\ \vdots \\ e :: \tau_2 \end{array}}}{\lambda x.\, e :: \tau_1 \to \tau_2}\ (\mathsf{abs}) \qquad \frac{e_1 :: \tau_1 \qquad \boxed{\begin{array}{c} x :: \tau_1 \\ \vdots \\ e_2 :: \tau_2 \end{array}}}{\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 :: \tau_2}\ (\mathsf{let})$$

$$\frac{e :: \tau}{e :: \tau}\ (\mathsf{copy}) \qquad \frac{e_1 :: \mathsf{Bool} \qquad e_2 :: \tau \qquad e_3 :: \tau}{\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 :: \tau}\ (\mathsf{ite})$$

Table 1: The inference rules for type checking.

The application of a type substitution to another type substitution is their *composition*

$$\sigma_1\sigma_2 = (\lambda x.\, \sigma_1(x)\sigma_2)$$

The following example familiarizes the reader with type substitutions.

Example 1. Consider the type τ and the type substitutions σ and σ_2:

$$\begin{aligned} \tau &= \alpha \to (\alpha_1 \to \alpha_3) \\ \sigma &= \{\alpha \mapsto \mathsf{Int} \to \mathsf{Int}, \alpha_1 \mapsto \mathsf{List}(\alpha_2)\} \\ \sigma_2 &= \{\alpha_3 \mapsto \alpha_4, \alpha_2 \mapsto \alpha, \alpha \mapsto \alpha_1\} \end{aligned}$$

Then we have

$$\begin{aligned} \tau\sigma &= (\mathsf{Int} \to \mathsf{Int}) \to (\mathsf{List}(\alpha_2) \to \alpha_3) \\ \sigma\sigma_2 &= \{\alpha \mapsto \mathsf{Int} \to \mathsf{Int}, \alpha_1 \mapsto \mathsf{List}(\alpha), \alpha_3 \mapsto \alpha_4, \alpha_2 \mapsto \alpha\} \end{aligned}$$

A *typing judgment* is written $E \vdash e :: \tau$ (akin to a sequent in natural deduction) for some typing environment E, core FP expression e and type τ.

Such a typing judgment reads:

From the typing environment E, type τ can be derived for the expression e.

The *inference rules* that are used to check types (that is, prove typing judgments by natural deduction) are given in Table 1.

The part of an inference rule above the line consists of a number of *premises*. Below the line are its *conclusions*.

Now type checking of a typing judgment $E \vdash e :: \tau$ corresponds to giving a natural deduction proof using the rules from Table 1.

Let us have a closer look at the different inference rules.

(ins) The *instantiation* rule states that for elements of the typing environment arbitrary type instances (obtained by applying the type substitution σ) of their original type τ can be derived. In other words, those elements are *polymorphic* (for example, if a length function was in E, we could apply it to lists of Ints, Bools, ...).

(app) The *application* rule states that in order to prove that the type τ_1 can be derived for the application $e_1\ e_2$ we have to show that it is the case that (1) the type $\tau_2 \to \tau_1$ can be derived for e_1 and (2) the type τ_2 can be derived for e_2. This rule captures the intuition that function application does only make sense for functions, that is, expressions of some *arrow type* $\tau \to \tau'$, and further, the argument of a function has to have the correct type. (When arrow types are read as implication, this rule resembles *modus ponens*—aka *implication elimination*—for propositional logic.)

(abs) The *abstraction* rule states that a function $\lambda x.\, e$ has type $\tau_1 \to \tau_2$ if under the assumption that the variable x has type τ_1 we can derive type τ_2 for the function body e. (Again reading arrow types as implication, this rule corresponds to *implication introduction*—on the level of types—for propositional logic.)

(let) The *let(-binding)* rule states that the expression **let** $x = e_1$ **in** e_2 has type τ_2 if (1) the type τ_1 can be derived for e_1 and (2) assuming type τ_1 for variable x, type τ_2 can be derived for e_2. (You may think of the let rule as a combination of the abstraction rule and the application rule, since **let** $x = e_1$ **in** e_2 is equivalent to $(\lambda x.\, e_2)\ e_1$ in our setting.)[2]

(copy) The *copy* rule allows us to reuse previously derived results.

(ite) The *if-then-else* rule captures the intuition that the conditional expression e_1 of **if** e_1 **then** e_2 **else** e_3 has to be of type Bool and the *then*-branch as well as the *else*-branch have to be of the same type.

Often, thinking in a goal-oriented manner, that is, reading inference rules from bottom to top (or put differently, thinking about the premises of an inference rule as the subgoals we have to prove in order to arrive at its conclusion), will help you in finding a proof of a typing judgment. The only guesswork has to be done for the type τ_2 of the argument in the (app) rule as well as the type τ_1 (which also corresponds to a function argument) in the (let) rule.

Example 2. Consider the typing environment $E = \{\mathsf{True} :: \mathsf{Bool}, + :: \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int}\}$. Then the judgment $E \vdash (\lambda x.\, x)\ \mathsf{True} :: \mathsf{Bool}$ can be proved by

1	$\mathsf{True} :: \mathsf{Bool}$	ins E
2	$x :: \mathsf{Bool}$	assumption
3	$\lambda x.\, x :: \mathsf{Bool} \to \mathsf{Bool}$	abs 2
4	$(\lambda x.\, x)\ \mathsf{True} :: \mathsf{Bool}$	app 3, 1

and the judgment $E \vdash \lambda x.\, x + x :: \mathsf{Int} \to \mathsf{Int}$ by

[2] Note that this variant of **let** is *not* polymorphic, that is, it is not possible to use different type instances of τ_1 for x inside e_2.

1	$x :: \mathsf{Int}$	assumption
2	$+ :: \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int}$	ins E
3	$(+)\ x :: \mathsf{Int} \to \mathsf{Int}$	app 2, 1
4	$x + x :: \mathsf{Int}$	app 3, 1
5	$\lambda x.\, x + x :: \mathsf{Int} \to \mathsf{Int}$	abs 1–4

In the second example the '+' is used infix. This is just for convenience. By the grammar for core FP expressions it would be prefix, which is used as in Haskell, that is, $(+)\ x\ y$ instead of $x + y$.

3 Type Inference

Inferring the most general type of a given expression is known as *type inference*. It is a bit more complicated than type checking. Hence some further definitions are needed.

The set of *type variables* of a type τ is given by

$$\begin{aligned} \mathcal{V}(\alpha) &= \{\alpha\} \\ \mathcal{V}(\tau_1 \to \tau_2) &= \mathcal{V}(\tau_1) \cup \mathcal{V}(\tau_2) \\ \mathcal{V}(C(\tau_1, \ldots, \tau_n)) &= \mathcal{V}(\tau_1) \cup \ldots \cup \mathcal{V}(\tau_n) \end{aligned}$$

Example 3. Recall the type τ and the substitution σ from Example 1. Then we have:

$$\begin{aligned} \mathcal{V}(\tau) &= \{\alpha, \alpha_1, \alpha_3\} \\ \mathcal{V}(\tau\sigma) &= \{\alpha_2, \alpha_3\} \end{aligned}$$

3.1 Unification Problems

A *unification problem* is represented by a (finite) sequence of equations between types $\tau_1 \approx \tau_1'; \ldots; \tau_n \approx \tau_n'$. Here, an empty sequence is represented by $\square$. *Unification* is the process of finding a substitution σ such that the types in each equation become syntactically identical, that is, $\tau_1\sigma = \tau_1'\sigma; \ldots; \tau_n\sigma = \tau_n'\sigma$. Such a substitution is then called *a solution to the unification problem* or just a *unifier*. If a unification problem admits a solution then it is found by any exhaustive sequence of applications of the inference rules of Table 2

(d) The *decomposition* rules capture the facts that (d_1) two applications of type constructors are unifiable if and only if the type constructors are equal *and* their respective parameters are unifiable, and (d_2) two arrow types are unifiable if and only if their respective components are unifiable.

(**d**) **decomposition**

$$\frac{E_1; C(\tau_1, \ldots, \tau_n) \approx C(\tau'_1, \ldots, \tau'_n); E_2}{E_1; \tau_1 \approx \tau'_1; \ldots; \tau_n \approx \tau'_n; E_2} \ (\mathsf{d}_1)$$

$$\frac{E_1; \tau_1 \to \tau_2 \approx \tau'_1 \to \tau'_2; E_2}{E_1; \tau_1 \approx \tau'_1; \tau_2 \approx \tau'_2; E_2} \ (\mathsf{d}_2)$$

(**t**) **removal of trivial equations**

$$\frac{E_1; \tau \approx \tau; E_2}{E_1; E_2} \ (\mathsf{t})$$

(**v**) **variable elimination**

$$\frac{E_1; \alpha \approx \tau; E_2 \quad \alpha \notin \mathcal{V}(\tau)}{(E_1; E_2)\{\alpha \mapsto \tau\}} \ (\mathsf{v}_1) \qquad \frac{E_1; \tau \approx \alpha; E_2 \quad \alpha \notin \mathcal{V}(\tau)}{(E_1; E_2)\{\alpha \mapsto \tau\}} \ (\mathsf{v}_2)$$

Table 2: The inference rules for unification.

(v) The *variable* rules state that as soon as either the left-hand side (for v_1) or the right-hand side (for v_2) of an equation is a type variable α, the extracted information can be used (in form of a substitution) to refine the remaining problem, but only if the type variable does not occur on the other side of the equation. This is called the *occur*-check.

(t) The *trivial equations removal* rule does exactly that, it removes trivial equations (which do neither pose further constraints nor give additional information), that is, equations where left-hand side and right-hand side are identical.

We apply the rules of unification from top to bottom. We depict unification derivations as sequences. If E is the premise and E' the conclusion of an inference rule r (with $r \in \{\mathsf{d}_1, \mathsf{d}_2, \mathsf{v}_1, \mathsf{v}_2, \mathsf{t}\}$), the application of r is written as $E \Rightarrow_\sigma^{(r)} E'$, where σ indicates a substitution (for $r \in \{\mathsf{d}_1, \mathsf{d}_2, \mathsf{t}\}$ the substitution ι, that is, the *empty substitution*, with $\iota(\alpha) = \alpha$ for all type variables α, is used). To solve a given unification problem E_1 the inference rules are applied repeatedly. The inference rules are designed such that this process stops after finitely many, say n, steps:

$$E_1 \Rightarrow_{\sigma_1}^{(r_1)} E_2 \Rightarrow_{\sigma_2}^{(r_2)} \cdots \Rightarrow_{\sigma_n}^{(r_n)} E_{n+1}.$$

If $E_{n+1} = \square$ then E_1 has the solution $\sigma = \sigma_1\sigma_2\cdots\sigma_n$.[3] If $E_n \neq \square$ then E_1 does not have a solution.

[3]The order of applying the unification rules to equations in E may have an effect on the unifier. However, every computed unifier σ is *most general*. This means that any other unifier τ can be obtained from σ. Formally this means that there exists a substitution μ such that $\tau = \sigma\mu$.

$$\frac{E, e :: \tau_0 \rhd e :: \tau_1}{\tau_0 \approx \tau_1}\ (\mathsf{con}) \qquad \frac{E \rhd e_1\ e_2 :: \tau}{E \rhd e_1 :: \alpha \to \tau; E \rhd e_2 :: \alpha}\ (\mathsf{app})$$

$$\frac{E \rhd \lambda x.\, e :: \tau}{\tau \approx \alpha_1 \to \alpha_2; E, x :: \alpha_1 \rhd e :: \alpha_2}\ (\mathsf{abs}) \qquad \frac{E \rhd \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 :: \tau}{E \rhd e_1 :: \alpha; E, x :: \alpha \rhd e_2 :: \tau}\ (\mathsf{let})$$

$$\frac{E \rhd \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 :: \tau}{E \rhd e_1 :: \mathsf{Bool}; E \rhd e_2 :: \tau; E \rhd e_3 :: \tau}\ (\mathsf{ite})$$

Table 3: The inference rules for computing typing constraints.

Example 4. The types $\mathsf{List}(\mathsf{Bool})$ and $\mathsf{List}(\alpha)$ are unifiable as can be seen by the derivation

$$\begin{array}{lll} \mathsf{List}(\mathsf{Bool}) \approx \mathsf{List}(\alpha) & \Rightarrow_{\iota}^{(\mathsf{d}_1)} & \mathsf{Bool} \approx \alpha \\ & \Rightarrow_{\{\alpha \mapsto \mathsf{Bool}\}}^{(\mathsf{v}_2)} & \square \end{array}$$

The unifier is $\iota\{\alpha \mapsto \mathsf{Bool}\} = \{\alpha \mapsto \mathsf{Bool}\}$.

3.2 Typing Constraints

Before unification can be used to implement type inference, a translation from type inference problems to unification problems is needed. This is the purpose of the inference rules in Table 3.

A *type inference problem* is given by $E \rhd e :: \tau$. This reads: "Transform the given problem into a unification problem using the inference rules of Table 3. Afterwards solve the resulting unification problem (if possible)." If unification succeeds, we obtain a substitution σ such that $E \vdash e :: \tau\sigma$.

Note that in the typing constraint rules of Table 3, the type variables α, α_1 and α_2 are required to be *fresh*, that is, they do not occur in the preceding derivations.

To solve the type inference problem $E \rhd e :: \alpha$ we apply the typing constraint rules from top to bottom. If no further application of a typing constraint rule is possible before having a unification problem, then statement e cannot be typed with respect to the type environment E. Otherwise, at some point the given type inference problem is translated into a unification problem (representing typing constraints). If the resulting unification problem has a solution, applying this to the initial type τ, represents the most general type of the original type inference problem, otherwise the original type inference problem is *not typable*.

Example 5. Consider the primitive environment P as defined above and application of the identity function as given by $\mathbf{let}\ id = \lambda x.\, x\ \mathbf{in}\ id\ 1$. The resulting type inference

problem is

$$P \rhd \textbf{let } id = \lambda x.\, x \textbf{ in } id\ 1 :: \alpha_0$$

where α_0 is a fresh type variable. Using the typing constraint rules this is transformed into the unification problem:

$$\underline{P \rhd \textbf{let } id = \lambda x.\, x \textbf{ in } id\ 1 :: \alpha_0}$$
$$\overset{\mathsf{let}}{\Rightarrow}$$
$$\underline{P \rhd \lambda x.\, x :: \alpha_1}; P, id :: \alpha_1 \rhd id\ 1 :: \alpha_0$$
$$\overset{\mathsf{abs}}{\Rightarrow}$$
$$\underline{P, x :: \alpha_2 \rhd x :: \alpha_3}; \alpha_1 \approx \alpha_2 \to \alpha_3; P, id :: \alpha_1 \rhd id\ 1 :: \alpha_0$$
$$\overset{\mathsf{con}}{\Rightarrow}$$
$$\alpha_2 \approx \alpha_3; \alpha_1 \approx \alpha_2 \to \alpha_3; \underline{P, id :: \alpha_1 \rhd id\ 1 :: \alpha_0}$$
$$\overset{\mathsf{app}}{\Rightarrow}$$
$$\alpha_2 \approx \alpha_3; \alpha_1 \approx \alpha_2 \to \alpha_3; \underline{P, id :: \alpha_1 \rhd id :: \alpha_4 \to \alpha_0}; P, id :: \alpha_1 \rhd 1 :: \alpha_4$$
$$\overset{\mathsf{con}}{\Rightarrow}$$
$$\alpha_2 \approx \alpha_3; \alpha_1 \approx \alpha_2 \to \alpha_3; \alpha_1 \approx \alpha_4 \to \alpha_0; \underline{P, id :: \alpha_1 \rhd 1 :: \alpha_4}$$
$$\overset{\mathsf{con}}{\Rightarrow}$$
$$\alpha_2 \approx \alpha_3; \alpha_1 \approx \alpha_2 \to \alpha_3; \alpha_1 \approx \alpha_4 \to \alpha_0; \mathsf{Int} \approx \alpha_4.$$

Afterwards we use unification to obtain a solution:

$$\begin{array}{ll}
 & \alpha_2 \approx \alpha_3; \alpha_1 \approx \alpha_2 \to \alpha_3; \alpha_1 \approx \alpha_4 \to \alpha_0; \mathsf{Int} \approx \alpha_4 \\
\Rightarrow^{(\mathsf{v}_1)}_{\{\alpha_2 \mapsto \alpha_3\}} & \alpha_1 \approx \alpha_3 \to \alpha_3; \alpha_1 \approx \alpha_4 \to \alpha_0; \mathsf{Int} \approx \alpha_4 \\
\Rightarrow^{(\mathsf{v}_1)}_{\{\alpha_1 \mapsto \alpha_3 \to \alpha_3\}} & \alpha_3 \to \alpha_3 \approx \alpha_4 \to \alpha_0; \mathsf{Int} \approx \alpha_4 \\
\Rightarrow^{(\mathsf{d}_2)}_{\iota} & \alpha_3 \approx \alpha_4; \alpha_3 \approx \alpha_0; \mathsf{Int} \approx \alpha_4 \\
\Rightarrow^{(\mathsf{v}_1)}_{\{\alpha_3 \mapsto \alpha_4\}} & \alpha_4 \approx \alpha_0; \mathsf{Int} \approx \alpha_4 \\
\Rightarrow^{(\mathsf{v}_1)}_{\{\alpha_4 \mapsto \alpha_0\}} & \mathsf{Int} \approx \alpha_0 \\
\Rightarrow^{(\mathsf{v}_2)}_{\{\alpha_0 \mapsto \mathsf{Int}\}} & \square.
\end{array}$$

The resulting unifier is

$$\sigma = \{\alpha_0 \mapsto \mathsf{Int}, \alpha_1 \mapsto \mathsf{Int} \to \mathsf{Int}, \alpha_2 \mapsto \mathsf{Int}, \alpha_3 \mapsto \mathsf{Int}, \alpha_4 \mapsto \mathsf{Int}\}.$$

Since the type variable α_0 was used for the initial type inference problem and $\sigma(\alpha_0) = \mathsf{Int}$, the most general type for **let** $id = \lambda x.\, x$ **in** $id\ 1$ is Int.

Note: To compute the unifier σ we look how the type variables change by applying all substitutions appearing in the unification proof sequence. For α_0 this is easy since α_0 is mapped to Int. For α_1 this is a bit more involved since α_1 is first mapped to

$\alpha_3 \to \alpha_3$, but afterwards α_3 is mapped to α_4 and finally α_4 is mapped to Int. Hence $\sigma(\alpha_1) = \mathsf{Int} \to \mathsf{Int}$.

Example 6. As a second example consider the expression $\lambda x.\, x\ x$ and the primitive environment P. The resulting type inference problem is

$$P \rhd \lambda x.\, x\ x :: \alpha_0$$

where α_0 is a fresh type variable. This is transformed into the unification problem:

$$\underline{P \rhd \lambda x.\, x\ x :: \alpha_0}$$
$$\overset{\mathsf{abs}}{\Rightarrow}$$
$$\underline{P, x :: \alpha_1 \rhd x\ x :: \alpha_2}; \alpha_0 \approx \alpha_1 \to \alpha_2$$
$$\overset{\mathsf{app}}{\Rightarrow}$$
$$\underline{P, x :: \alpha_1 \rhd x :: \alpha_3 \to \alpha_2}; P, x :: \alpha_1 \rhd x :: \alpha_3; \alpha_0 \approx \alpha_1 \to \alpha_2$$
$$\overset{\mathsf{con}}{\Rightarrow}$$
$$\alpha_1 \approx \alpha_3 \to \alpha_2; \underline{P, x :: \alpha_1 \rhd x :: \alpha_3}; \alpha_0 \approx \alpha_1 \to \alpha_2$$
$$\overset{\mathsf{con}}{\Rightarrow}$$
$$\alpha_1 \approx \alpha_3 \to \alpha_2; \alpha_1 \approx \alpha_3; \alpha_0 \approx \alpha_1 \to \alpha_2$$

Afterwards we use unification as follows:

$$\begin{array}{l} \alpha_1 \approx \alpha_3 \to \alpha_2; \alpha_1 \approx \alpha_3; \alpha_0 \approx \alpha_1 \to \alpha_2 \\ \alpha_3 \to \alpha_2 \approx \alpha_3; \alpha_0 \approx (\alpha_3 \to \alpha_2) \to \alpha_2 \end{array} \quad \Rightarrow^{(\mathsf{v}_1)}_{\{\alpha_1 \mapsto \alpha_3 \to \alpha_2\}}$$

where after the first step the occur-check fails and hence the given unification problem is not unifiable. This means that $\lambda x.\, x\ x$ is not typable.

Example 7. As a further example consider the Y-combinator and the empty environment $\varnothing$. The type inference problem $\varnothing \rhd \mathsf{Y} :: \alpha_0$ is transformed into a unification problem as follows:

$$\underline{\varnothing \rhd \lambda f.\, (\lambda x.\, f\ (x\ x))\ (\lambda x.\, f\ (x\ x)) :: \alpha_0}$$
$$\overset{\mathsf{abs}}{\Rightarrow}$$
$$\underline{f :: \alpha_1 \rhd (\lambda x.\, f\ (x\ x))\ (\lambda x.\, f\ (x\ x)) :: \alpha_2}; \alpha_0 \approx \alpha_1 \to \alpha_2$$
$$\overset{\mathsf{app}}{\Rightarrow}$$
$$\underline{f :: \alpha_1 \rhd \lambda x.\, f\ (x\ x) :: \alpha_3 \to \alpha_2}; f :: \alpha_1 \rhd \lambda x.\, f\ (x\ x) :: \alpha_3; \alpha_0 \approx \alpha_1 \to \alpha_2$$
$$\overset{\mathsf{abs}}{\Rightarrow}$$
$$\underline{\{f :: \alpha_1, x :: \alpha_4\} \rhd f\ (x\ x) :: \alpha_5}; \alpha_3 \to \alpha_2 \approx \alpha_4 \to \alpha_5;$$
$$f :: \alpha_1 \rhd \lambda x.\, f\ (x\ x) :: \alpha_3; \alpha_0 \approx \alpha_1 \to \alpha_2$$
$$\overset{\mathsf{app}}{\Rightarrow}$$

$$\overset{\mathsf{app}}{\Rightarrow}$$

$$\underline{\{f :: \alpha_1, x :: \alpha_4\} \rhd f :: \alpha_6 \to \alpha_5}; \{f :: \alpha_1, x :: \alpha_4\} \rhd x\ x :: \alpha_6;$$
$$\alpha_3 \to \alpha_2 \approx \alpha_4 \to \alpha_5; f :: \alpha_1 \rhd \lambda x.\ f\ (x\ x) :: \alpha_3; \alpha_0 \approx \alpha_1 \to \alpha_2$$

$$\overset{\mathsf{con}}{\Rightarrow}$$

$$\alpha_1 \approx \alpha_6 \to \alpha_5; \underline{\{f :: \alpha_1, x :: \alpha_4\} \rhd x\ x :: \alpha_6;}$$
$$\alpha_3 \to \alpha_2 \approx \alpha_4 \to \alpha_5; f :: \alpha_1 \rhd \lambda x.\ f\ (x\ x) :: \alpha_3; \alpha_0 \approx \alpha_1 \to \alpha_2$$

$$\overset{\mathsf{app}}{\Rightarrow}$$

$$\alpha_1 \approx \alpha_6 \to \alpha_5; \underline{\{f :: \alpha_1, x :: \alpha_4\} \rhd x :: \alpha_7 \to \alpha_6}; \{f :: \alpha_1, x :: \alpha_4\} \rhd x :: \alpha_7;$$
$$\alpha_3 \to \alpha_2 \approx \alpha_4 \to \alpha_5; f :: \alpha_1 \rhd \lambda x.\ f\ (x\ x) :: \alpha_3; \alpha_0 \approx \alpha_1 \to \alpha_2$$

$$\overset{\mathsf{con}}{\Rightarrow}$$

$$\alpha_1 \approx \alpha_6 \to \alpha_5; \alpha_4 \approx \alpha_7 \to \alpha_6; \underline{\{f :: \alpha_1, x :: \alpha_4\} \rhd x :: \alpha_7};$$
$$\alpha_3 \to \alpha_2 \approx \alpha_4 \to \alpha_5; f :: \alpha_1 \rhd \lambda x.\ f\ (x\ x) :: \alpha_3; \alpha_0 \approx \alpha_1 \to \alpha_2$$

$$\overset{\mathsf{con}}{\Rightarrow}$$

$$\alpha_1 \approx \alpha_6 \to \alpha_5; \alpha_4 \approx \alpha_7 \to \alpha_6; \alpha_4 \approx \alpha_7; \alpha_3 \to \alpha_2 \approx \alpha_4 \to \alpha_5;$$
$$\underline{f :: \alpha_1 \rhd \lambda x.\ f\ (x\ x) :: \alpha_3}; \alpha_0 \approx \alpha_1 \to \alpha_2$$

$$\overset{\mathsf{abs}}{\Rightarrow}$$

$$\alpha_1 \approx \alpha_6 \to \alpha_5; \alpha_4 \approx \alpha_7 \to \alpha_6; \alpha_4 \approx \alpha_7; \alpha_3 \to \alpha_2 \approx \alpha_4 \to \alpha_5;$$
$$\underline{\{f :: \alpha_1, x :: \alpha_8\} \rhd f\ (x\ x) :: \alpha_9}; \alpha_3 \approx \alpha_8 \to \alpha_9; \alpha_0 \approx \alpha_1 \to \alpha_2$$

$$\overset{\mathsf{app}}{\Rightarrow}$$

$$\alpha_1 \approx \alpha_6 \to \alpha_5; \alpha_4 \approx \alpha_7 \to \alpha_6; \alpha_4 \approx \alpha_7; \alpha_3 \to \alpha_2 \approx \alpha_4 \to \alpha_5;$$
$$\underline{\{f :: \alpha_1, x :: \alpha_8\} \rhd f :: \alpha_{10} \to \alpha_9}; \{f :: \alpha_1, x :: \alpha_8\} \rhd x\ x :: \alpha_{10}; \alpha_3 \approx \alpha_8 \to \alpha_9;$$
$$\alpha_0 \approx \alpha_1 \to \alpha_2$$

$$\overset{\mathsf{con}}{\Rightarrow}$$

$$\alpha_1 \approx \alpha_6 \to \alpha_5; \alpha_4 \approx \alpha_7 \to \alpha_6; \alpha_4 \approx \alpha_7; \alpha_3 \to \alpha_2 \approx \alpha_4 \to \alpha_5;$$
$$\alpha_1 \approx \alpha_{10} \to \alpha_9; \underline{\{f :: \alpha_1, x :: \alpha_8\} \rhd x\ x :: \alpha_{10}}; \alpha_3 \approx \alpha_8 \to \alpha_9; \alpha_0 \approx \alpha_1 \to \alpha_2$$
$$\overset{\mathsf{app}}{\Rightarrow} \alpha_1 \approx \alpha_6 \to \alpha_5; \alpha_4 \approx \alpha_7 \to \alpha_6; \alpha_4 \approx \alpha_7; \alpha_3 \to \alpha_2 \approx \alpha_4 \to \alpha_5;$$
$$\alpha_1 \approx \alpha_{10} \to \alpha_9; \underline{\{f :: \alpha_1, x :: \alpha_8\} \rhd x :: \alpha_{11} \to \alpha_{10}};$$
$$\{f :: \alpha_1, x :: \alpha_8\} \rhd x :: \alpha_{11}; \alpha_3 \approx \alpha_8 \to \alpha_9; \alpha_0 \approx \alpha_1 \to \alpha_2$$

$$\overset{\mathsf{cons}}{\Rightarrow}$$

$$\alpha_1 \approx \alpha_6 \to \alpha_5; \alpha_4 \approx \alpha_7 \to \alpha_6; \alpha_4 \approx \alpha_7; \alpha_3 \to \alpha_2 \approx \alpha_4 \to \alpha_5;$$
$$\alpha_1 \approx \alpha_{10} \to \alpha_9; \alpha_8 \approx \alpha_{11} \to \alpha_{10}; \underline{\{f :: \alpha_1, x :: \alpha_8\} \rhd x :: \alpha_{11}};$$
$$\alpha_3 \approx \alpha_8 \to \alpha_9; \alpha_0 \approx \alpha_1 \to \alpha_2$$

$$\overset{\mathsf{cons}}{\Rightarrow}$$

$$\alpha_1 \approx \alpha_6 \to \alpha_5; \alpha_4 \approx \alpha_7 \to \alpha_6; \alpha_4 \approx \alpha_7; \alpha_3 \to \alpha_2 \approx \alpha_4 \to \alpha_5;$$
$$\alpha_1 \approx \alpha_{10} \to \alpha_9; \alpha_8 \approx \alpha_{11} \to \alpha_{10};$$
$$\alpha_8 \approx \alpha_{11}; \alpha_3 \approx \alpha_8 \to \alpha_9; \alpha_0 \approx \alpha_1 \to \alpha_2$$

It is left as an exercise to show that this unification problem is not solvable and hence Y is not typable.

4 Recursion

An interesting result for the simply typed lambda calculus (that will not be proved in this course, however), is that every typable λ-term is guaranteed to terminate. This is also true for core FP. As has been seen in the last section, Y is not typable. That is, of course, due to the mentioned result. But if Y is not typable, there is no possibility to define recursive functions (since all other thinkable fixed point combinators are also not typable). The trick is, to include Y as a primitive constant and just assign a type that suffices to make applications of Y well-typed.

The idea of the fixed point combinator was that given some function t (expecting itself as first argument), it replicates this function and applies it to the computed fixed point $\mathsf{Y}\ t$, that is, $t\ (\mathsf{Y}\ t)$.

Example 8. Again, consider the function length. As a first approach for its implementation consider

$$\mathsf{length} = \lambda x.\, \mathbf{if}\ \mathsf{null}\ x\ \mathbf{then}\ 0\ \mathbf{else}\ 1 + \mathsf{length}\ (\mathsf{tail}\ x)$$

where $\mathsf{null} :: \mathsf{List}(\alpha) \to \mathsf{Bool}$, $\mathsf{tail} :: \mathsf{List}(\alpha) \to \mathsf{List}(\alpha)$, $0 :: \mathsf{Int}$, $1 :: \mathsf{Int}$, and $+ :: \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int}$ are constants contained in P. As already mentioned for the corresponding λ-term, this definition is not well-defined, due to the recursive reference to length.

Again the problem is solved by introducing an additional argument and applying Y to the resulting expression, that is,

$$\mathsf{length} = \mathsf{Y}\ (\lambda f x.\, \mathbf{if}\ \mathsf{null}\ x\ \mathbf{then}\ 0\ \mathbf{else}\ 1 + f\ (\mathsf{tail}\ x)).$$

It can be seen that Y expects some function (that is, an expression having an arrow type) as its argument. The function that is supplied to Y in turn expects another function (namely the one that is about to be defined). This process should be generally applicable, hence Y does need some not too restricted type. The result after some investigation will be similar to

$$\mathsf{Y} :: (\alpha \to \alpha) \to \alpha.$$

Now let P_μ denote the primitive typing environment P extended by a type assignment for Y

$$P_\mu = P \cup \{\mathsf{Y} :: (\alpha \to \alpha) \to \alpha\}$$

5 Further Remarks

The type inference algorithm presented in this chapter follows the *Hindley-Milner type inference* algorithm for the simply typed lambda calculus. It was first presented by Hindley [1] and independently conceived by Milner [2]. Sometimes it is also referred to as *Algorithm W*.

References

[1] Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969. doi:10.2307/1995158.

[2] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Science*, 17(3):348–375, 1978. doi:10.1016/0022-0000(78)90014-4.