2-4-2 / Type systems
Extensions

François Pottier

December 9, 2008

Two presentations

Two presentations of type inference for Damas and Milner's type system are possible:

- *one of* Milner's classic algorithms [1978], $\mathcal{W}$ or $\mathcal{J}$; see my old course notes for details [Pottier, 2002, §3.3];
- a constraint-based presentation [Pottier and Rémy, 2005];

I prefer the latter, but review the former first.

Contents

This algorithm *expects* a pair $\Gamma \vdash t$, *produces* a type T, and uses two global variables, V and φ.

V is an infinite *fresh supply* of type variables:

$$\begin{aligned} \text{fresh} \quad = \quad & do\ X \in V \\ & do\ V \leftarrow V \setminus \{X\} \\ & \text{return } X \end{aligned}$$

φ is a *substitution* (of types for type variables), initially the identity.

Here is the algorithm in monadic style:

$$
\begin{aligned}
\mathcal{J}(\Gamma \vdash x) \quad &= \quad \text{let } \forall X_1 \ldots X_n.T = \Gamma(x) \\
&\quad\quad \textit{do } X'_1, \ldots, X'_n = \text{fresh}, \ldots, \text{fresh} \\
&\quad\quad \text{return } [X_i \mapsto X'_i]_{i=1}^{n}(T) \; \textit{— take a fresh instance} \\
\mathcal{J}(\Gamma \vdash \lambda x.t_1) \quad &= \quad \textit{do } X = \text{fresh} \\
&\quad\quad \textit{do } T_1 = \mathcal{J}(\Gamma; x : X \vdash t_1) \\
&\quad\quad \text{return } X \rightarrow T_1 \; \textit{— form an arrow type} \\
&\ldots
\end{aligned}
$$

$$\begin{array}{rcl}
& \dots & \\
\mathcal{J}(\Gamma \vdash t_1\ t_2) & = & \text{do } T_1 = \mathcal{J}(\Gamma \vdash t_1) \\
& & \text{do } T_2 = \mathcal{J}(\Gamma \vdash t_2) \\
& & \text{do } X = \text{fresh} \\
& & \text{do } \varphi \leftarrow \text{mgu}(\varphi(T_1) = \varphi(T_2 \rightarrow X)) \circ \varphi \\
& & \text{return } X \ \text{ -- } \textit{solve } T_1 = T_2 \rightarrow X \\
\mathcal{J}(\Gamma \vdash \text{let } x = t_1 \text{ in } t_2) & = & \text{do } T_1 = \mathcal{J}(\Gamma \vdash t_1) \\
& & \text{let } \sigma = \overline{\forall}\, \text{ftv}(\varphi(\Gamma)).\varphi(T_1) \ \text{ -- } \textit{generalize} \\
& & \text{return } \mathcal{J}(\Gamma; x : \sigma \vdash t_2)
\end{array}$$

($\overline{\forall}\bar{X}.T$ quantifies over all type variables *other than* $\bar{X}$.)

Theorem (Correctness)

If $\mathcal{J}(\Gamma \vdash t)$ *terminates in state* (φ, V) *and returns* T*, then* $\varphi(\Gamma) \vdash t : \varphi(T)$ *is a valid judgement.*

Theorem (Completeness)

Let Γ be an environment. Let (φ_0, V_0) be a state that satisfies the algorithm's invariant. Let θ_0 and T_0 be such that $\theta_0\varphi_0(\Gamma) \vdash t : T_0$ is a judgement. Then, the execution of $\mathcal{I}(\Gamma \vdash t)$ out of the initial state (φ_0, V_0) succeeds. Let (φ_1, V_1) be its final state and T_1 be its result. Then, there exists a substitution θ_1 such that $\theta_0\varphi_0$ and $\theta_1\varphi_1$ coincide outside V_0 and such that T_0 equals $\theta_1\varphi_1(T_1)$.

Proof.

[...] We have

$$\theta_1\varphi_1(\gamma) = \theta_1\psi\varphi'_2(\gamma) = \theta''_2\varphi'_2(\gamma).$$

Since a is fresh for γ and φ'_2, we can pursue with

$$\theta''_2\varphi'_2(\gamma) = \theta'_2\varphi'_2(\gamma) = \theta'_1\varphi'_1(\gamma) = \theta_0\varphi_0(\gamma).$$

Thus, $\theta_1\varphi_1$ and $\theta_0\varphi_0$ coincide outside V_0 [...] □

A typing (Γ', T) is *relative to Γ* if and only if its first component Γ' is an instance of Γ.

A typing of t is *principal relative to Γ* if and only if it is relative to Γ and every typing of t relative to Γ is an instance of it.

Corollary (Relative principal typings)

The execution of $\mathcal{J}(\Gamma \vdash t)$ succeeds if and only if t admits a typing relative to Γ.

Furthermore, if φ_1 and T_1 are the algorithm's results, then $(\varphi_1(\Gamma), \varphi_1(T_1))$ is a typing of t and is principal relative to Γ.

This is also known as the *principal types* property.

See [Jim, 1995, Wells, 2002] for more details on principal typings and principal types.

Algorithm $\mathcal{J}$ mixes *generation* and *solving* of equations. This lack of modularity leads to several weaknesses:

- proofs are more difficult;
- correctness and efficiency concerns are not clearly separated;
- generalizations, such as the introduction of subtyping, are not easy.

Algorithm $\mathcal{J}$ works with *substitutions*, instead of *constraints*.

Substitutions are an approximation to solved forms for unification constraints.

Working with substitutions means using *most general unifiers*, *composition*, and *restriction*.

Working with constraints means using *equations*, *conjunction*, and *existential quantification*.

Road map

Type inference for Damas and Milner's type system involves slightly more than first-order unification: there is also *generalization* and *instantiation* of type schemes.

So, the constraint language must be enriched.

I proceed in two steps:

- still within simply-typed λ-calculus, I present a variation of the constraint language;
- building on this variation, I introduce polymorphism.

How about letting the constraint solver, instead of the constraint generator, deal with *environment access* and *lookup?*

Let's enrich the syntax of constraints:

$$C ::= \ldots \mid x = T \mid \mathit{def}\ x : T \text{ in } C$$

The idea is to interpret constraints in such a way as to validate the equivalence law:

$$\mathit{def}\ x : T \text{ in } C \equiv [x \mapsto T]C$$

The *def* form is an *explicit substitution* form.

More precisely, here is the new interpretation of constraints.

As before, a valuation ϕ maps type variables X to ground types.

In addition, a valuation ψ maps variables x to ground types.

The satisfaction judgement now takes the form $\phi, \psi \vdash C$. The new rules of interest are:

$$\frac{\psi x = \phi T}{\phi, \psi \vdash x = T} \qquad \frac{\phi, \psi[x \mapsto \phi T] \vdash C}{\phi, \psi \vdash \mathrm{def}\ x : T\ \mathrm{in}\ C}$$

(All other rules are modified to just transport ψ.)

Constraint generation is now a mapping of an expression t and a type T to a constraint $[\![t : T]\!]$. There is no longer a need for the parameter Γ.

$$
\begin{array}{rcl}
[\![x : T]\!] & = & x = T \\
[\![\lambda x.t : T]\!] & = & \exists X_1 X_2.(\mathsf{def}\ x : X_1\ \mathsf{in}\ [\![t : X_2]\!] \wedge X_1 \rightarrow X_2 = T) \\
 & & \text{if } X_1, X_2 \# t, T \\
[\![t_1\ t_2 : T]\!] & = & \exists X.([\![t_1 : X \rightarrow T]\!] \wedge [\![t_2 : X]\!]) \\
 & & \text{if } X \# t_1, t_2, T
\end{array}
$$

Look ma, *no environments!*

Theorem (Soundness and completeness)

Let $\mathrm{fv}(t) = \mathrm{dom}(\Gamma)$*. Then,* $\phi, \phi\Gamma \vdash [\![t : T]\!]$ *if and only if* $\phi\Gamma \vdash t : \phi T$*.*

Corollary

Let $\mathrm{fv}(t) = \emptyset$*. Then, t is well-typed if and only if* $\exists X.[\![t : X]\!] \equiv \mathrm{true}$*.*

This variation shows that there is *freedom* in the design of the constraint language, and that altering this design can *shift work* from the constraint generator to the constraint solver, or vice-versa.

To permit polymorphism, we must extend the syntax of constraints so that a variable x denotes not just a ground type, but a *set of ground types*.

However, these sets cannot be represented as type schemes $\forall \bar{X}.T$, because constructing these simplified forms requires constraint solving.

To avoid mingling constraint generation and constraint solving, we use type schemes that incorporate constraints: *constrained type schemes*.

The syntax of *constraints* and of *constrained type schemes* is:

$$
\begin{array}{rcl}
C & ::= & T = T \mid C \wedge C \mid \exists X.C \\
 & \mid & x \preceq T \\
 & \mid & \varsigma \preceq T \\
 & \mid & \mathsf{def}\ x : \varsigma\ \mathsf{in}\ C \\
\varsigma & ::= & \forall \bar{X}[C].T
\end{array}
$$

$x \preceq T$ and $\varsigma \preceq T$ are *instantiation constraints.* The latter form is introduced so as to make the syntax stable under substitutions of constrained type schemes for variables.

As before, $\mathsf{def}\ x : \varsigma\ \mathsf{in}\ C$ is an *explicit substitution* form.

Enriching constraints

The idea is to interpret constraints in such a way as to validate the *equivalence laws*:

$$\text{def } x : \varsigma \text{ in } C \equiv [x \mapsto \varsigma]C$$

$$(\forall \bar{X}[C].T) \preceq T' \equiv \exists \bar{X}.(C \wedge T = T') \quad \text{if } \bar{X} \# T'$$

Using these laws, a closed constraint can be rewritten to a unification constraint (with a possibly exponential increase in size).

The new constructs do not add much expressive power. They add just enough to allow a stand-alone formulation of constraint generation.

A type variable X still denotes a ground type.

A variable x now denotes a *set* of ground types.

Instantiation constraints are interpreted as *set membership.*

$$\frac{\psi x \ni \phi T}{\phi, \psi \vdash x \preceq T} \qquad \frac{\binom{\psi}{\phi}\varsigma \ni \phi T}{\phi, \psi \vdash \varsigma \preceq T} \qquad \frac{\phi, \psi[x \mapsto \binom{\psi}{\phi}\varsigma] \vdash C}{\phi, \psi \vdash \mathsf{def}\ x : \varsigma\ \mathsf{in}\ C}$$

The interpretation of $\forall\bar{X}[C].T$ under ϕ and ψ is the set of all $\phi' T$, where ϕ and ϕ' coincide outside $\bar{X}$ and where ϕ' and ψ satisfy C.

$$\binom{\psi}{\phi}(\forall\bar{X}[C].T) = \{\phi' T \mid (\phi' \setminus \bar{X} = \phi \setminus \bar{X}) \wedge (\phi', \psi \vdash C)\}$$

For instance, the interpretation of $\forall X[\exists Y.X = Y \rightarrow Z].X \rightarrow X$ under ϕ and ψ is the set of all ground types of the form $(t \rightarrow \phi Z) \rightarrow (t \rightarrow \phi Z)$, where t ranges over ground types.

This is also the interpretation of $\forall Y.(Y \rightarrow Z) \rightarrow (Y \rightarrow Z)$. Every constrained type scheme is equivalent to a standard type scheme.

In the following, I use a variant of the *def* construct:

$$\text{let } x : \varsigma \text{ in } C \quad \equiv \quad \text{def } x : \varsigma \text{ in } ((\exists X.x \preceq X) \wedge C)$$

It would be equivalent to provide a direct interpretation of it:

$$\frac{\binom{\psi}{\phi}\varsigma \neq \emptyset \qquad \phi, \psi[x \mapsto \binom{\psi}{\phi}\varsigma] \vdash C}{\phi, \psi \vdash \text{let } x : \varsigma \text{ in } C}$$

Constraint generation is now as follows:

$$
\begin{array}{rcl}
[\![x : T]\!] & = & x \preceq T \\
[\![\lambda x.t : T]\!] & = & \exists X_1 X_2.(\text{def } x : X_1 \text{ in } [\![t : X_2]\!] \wedge X_1 \rightarrow X_2 = T) \\
 & & \text{if } X_1, X_2 \# t, T \\
[\![t_1\ t_2 : T]\!] & = & \exists X.([\![t_1 : X \rightarrow T]\!] \wedge [\![t_2 : X]\!]) \\
 & & \text{if } X \# t_1, t_2, T \\
[\![\text{let } x = t_1 \text{ in } t_2 : T]\!] & = & \text{let } x : (\!|t_1|\!) \text{ in } [\![t_2 : T]\!] \\
(\!|t|\!) & = & \forall X[[\![t : X]\!]].X
\end{array}
$$

$(\!|t|\!)$ is a *principal constrained type scheme* for t: its intended interpretation is the set of all ground types that t admits.

Lemma

$$\exists X.(\llbracket t : X \rrbracket \wedge X = T) \quad \equiv \quad \llbracket t : T \rrbracket \textit{ if } X \# T.$$

Lemma

$$(\!| t |\!) \preceq T \quad \equiv \quad \llbracket t : T \rrbracket.$$

Lemma

$$[x \mapsto (\!| t_1 |\!)]\llbracket t_2 : T \rrbracket \quad \equiv \quad \llbracket [x \mapsto t_1]t_2 : T \rrbracket.$$

Lemma

$$\llbracket \text{let } x = t_1 \text{ in } t_2 : T \rrbracket \quad \equiv \quad \llbracket t_1; [x \mapsto t_1]t_2 : T \rrbracket.$$

The constraint associated with a let construct is *equivalent* to the constraint associated with its let-normal form.

Lemma

The size of $[\![t : T]\!]$ *is linear in the sum of the sizes of* t *and* T.

Constraint generation can be implemented in linear time and space.

The statement keeps its previous form, back but Γ now contains Damas-Milner type schemes.

Theorem (Soundness and completeness)

Let $\mathrm{fv}(t) = \mathrm{dom}(\Gamma)$*. Then,* $\phi, \phi\Gamma \vdash [\![t : T]\!]$ *if and only if* $\phi\Gamma \vdash t : \phi T$*.*

Note that

- constraint generation has *linear complexity;*
- constraint generation and constraint solving are *separate;*
- the constraint language remains *small* as the programming language grows.

This makes constraints suitable for use in an efficient and modular implementation.

Contents

Let Γ_0 stand for $assoc : \forall XY.X \longrightarrow \mathsf{list}\,(X \times Y) \longrightarrow Y$.

We take Γ_0 to be the *initial environment*, so that the constraints considered next are implicitly wrapped within the context $\mathsf{def}\ \Gamma_0\ \mathsf{in}\ []$.

Let t stand for the term

$$\begin{array}{l}\lambda x.\lambda l_1.\lambda l_2. \\ \quad \text{let } \mathit{assocx} = \mathit{assoc}\ x \text{ in} \\ \quad (\mathit{assocx}\ \ l_1, \mathit{assocx}\ \ l_2)\end{array}$$

One anticipates that *assocx* receives a polymorphic type scheme, which is instantiated twice at different types...

Let Γ stand for $x : X_0; l_1 : X_1; l_2 : X_2$. Then, the constraint $[\![t : X]\!]$ is (with a few minor simplifications):

$$\exists X_0 X_1 X_2 Y. \left(\begin{array}{l} X = X_0 \rightarrow X_1 \rightarrow X_2 \rightarrow Y \\ \text{def } \Gamma \text{ in} \\ \qquad \text{let } \mathit{assocx} : \forall Z_1 [\exists Z_2. \left(\begin{array}{l} \mathit{assoc} \preceq Z_2 \rightarrow Z_1 \\ x \preceq Z_2 \end{array} \right)].Z_1 \text{ in} \\ \qquad \exists Y_1 Y_2. \left(\begin{array}{l} Y = Y_1 \times Y_2 \\ \forall i \quad \exists Z_2.(\mathit{assocx} \preceq Z_2 \rightarrow Y_i \wedge l_i \preceq Z_2) \end{array} \right) \end{array} \right)$$

(The index i ranges over $\{1, 2\}$.)

Constraint solving can be viewed as a *rewriting process* that exploits *equivalence laws*. Because equivalence is, by construction, a *congruence*, rewriting is permitted within an arbitrary context.

For instance, environment access is allowed by the law

$$\text{let } x : \varsigma \text{ in } \mathcal{C}[x \preceq T] \quad \equiv \quad \text{let } x : \varsigma \text{ in } \mathcal{C}[\varsigma \preceq T]$$

where $\mathcal{C}$ is a context that does not bind x.

Thus, within the context def $\Gamma_0; \Gamma$ in [], the constraint:

$$\begin{pmatrix} assoc \preceq Z_2 \rightarrow Z_1 \\ x \preceq Z_2 \end{pmatrix}$$

is *equivalent* to:

$$\begin{pmatrix} \exists XY.(X \rightarrow \text{list } (X \times Y) \rightarrow Y = Z_2 \rightarrow Z_1) \\ X_0 = Z_2 \end{pmatrix}$$

By first-order unification, the constraint:

$$\exists Z_2.(\exists XY.(X \to \text{list } (X \times Y) \to Y = Z_2 \to Z_1) \wedge X_0 = Z_2)$$

simplifies down successively to:

$$\exists Z_2.(\exists XY.(X = Z_2 \wedge \text{list } (X \times Y) \to Y = Z_1) \wedge X_0 = Z_2)$$

$$\exists Z_2.(\exists Y.(\text{list } (Z_2 \times Y) \to Y = Z_1) \wedge X_0 = Z_2)$$

$$\exists Y.(\text{list } (X_0 \times Y) \to Y = Z_1)$$

The constrained type scheme:

$$\forall Z_1[\exists Z_2.(\mathit{assoc} \preceq Z_2 \rightarrow Z_1 \wedge x \preceq Z_2)].Z_1$$

is thus *equivalent* to:

$$\forall Z_1[\exists Y.(\text{list } (X_0 \times Y) \rightarrow Y = Z_1)].Z_1$$

which can also be written:

$$\forall Z_1 Y[\text{list } (X_0 \times Y) \rightarrow Y = Z_1].Z_1$$

$$\forall Y.\text{list } (X_0 \times Y) \rightarrow Y$$

The initial constraint has now been simplified down to:

$$\exists X_0 X_1 X_2 Y. \left(\begin{array}{l} X = X_0 \rightarrow X_1 \rightarrow X_2 \rightarrow Y \\ \text{def } \Gamma \text{ in} \\ \quad \text{let } \mathit{assocx} : \forall Y.\text{list } (X_0 \times Y) \rightarrow Y \text{ in} \\ \quad \exists Y_1 Y_2. \left(\begin{array}{l} Y = Y_1 \times Y_2 \\ \forall i \quad \exists Z_2.(\mathit{assocx} \preceq Z_2 \rightarrow Y_i \wedge l_i \preceq Z_2) \end{array} \right) \end{array} \right)$$

The simplification work spent on *assocx*'s type scheme was well worth the trouble, because we are now going to *duplicate* the simplified type scheme.

The sub-constraint:

$$\exists Z_2.(assocx \preceq Z_2 \to Y_i \wedge I_i \preceq Z_2)$$

where $i \in \{1, 2\}$, is rewritten:

$$\exists Z_2.(\exists Y.(\text{list } (X_0 \times Y) \to Y = Z_2 \to Y_i) \wedge X_i = Z_2)$$

$$\exists Y.(\text{list } (X_0 \times Y) \to Y = X_i \to Y_i)$$

$$\exists Y.(\text{list } (X_0 \times Y) = X_i \wedge Y = Y_i)$$

$$\text{list } (X_0 \times Y_i) = X_i$$

The initial constraint has now been simplified down to:

$$\exists X_0 X_1 X_2 Y. \left(\begin{array}{l} X = X_0 \rightarrow X_1 \rightarrow X_2 \rightarrow Y \\ \mathit{def}\ \Gamma\ \mathsf{in} \\ \quad \mathsf{let}\ \mathit{assocx} : \forall Y.\mathsf{list}\ (X_0 \times Y) \rightarrow Y\ \mathsf{in} \\ \quad \exists Y_1 Y_2. \left(\begin{array}{l} Y = Y_1 \times Y_2 \\ \forall i \quad \mathsf{list}\ (X_0 \times Y_i) = X_i \end{array} \right) \end{array} \right)$$

Now, the context $\mathit{def}\ \Gamma\ \mathsf{in}\ \mathsf{let}\ \mathit{assocx} : \ldots\ \mathsf{in}\ []$ can be dropped, because the constraint that it applies to contains no occurrences of assoc, x, l_1, or l_2.

The constraint becomes:

$$\exists X_0 X_1 X_2 Y. \left(\begin{array}{l} X = X_0 \rightarrow X_1 \rightarrow X_2 \rightarrow Y \\ \exists Y_1 Y_2. \left(\begin{array}{l} Y = Y_1 \times Y_2 \\ \forall i \quad \text{list}\,(X_0 \times Y_i) = X_i \end{array} \right) \end{array} \right)$$

that is:

$$\exists X_0 X_1 X_2 Y Y_1 Y_2. \left(\begin{array}{l} X = X_0 \rightarrow X_1 \rightarrow X_2 \rightarrow Y \\ Y = Y_1 \times Y_2 \\ \forall i \quad \text{list}\,(X_0 \times Y_i) = X_i \end{array} \right)$$

and, by eliminating a few auxiliary variables:

$$\exists X_0 Y_1 Y_2. (X = X_0 \rightarrow \text{list}\,(X_0 \times Y_1) \rightarrow \text{list}\,(X_0 \times Y_2) \rightarrow Y_1 \times Y_2)$$

We have shown the following equivalence between constraints:

$$\begin{array}{ll} & \text{def } \Gamma_0 \text{ in } [\![t : X]\!] \\ \equiv & \exists X_0 Y_1 Y_2. (X = X_0 \rightarrow \text{list } (X_0 \times Y_1) \rightarrow \text{list } (X_0 \times Y_2) \rightarrow Y_1 \times Y_2) \end{array}$$

That is, the *principal type scheme* of t relative to Γ_0 is

$$\forall X_0 Y_1 Y_2. X_0 \rightarrow \text{list } (X_0 \times Y_1) \rightarrow \text{list } (X_0 \times Y_2) \rightarrow Y_1 \times Y_2$$

Again, constraint solving can be explained in terms of a *small-step rewrite system*. Again, one checks that every step is meaning-preserving, that the system is normalizing, and that every normal form is either literally "false" or satisfiable.

Different constraint solving *strategies* lead to different behaviors in terms of complexity, error explanation, etc.

See ATTAPL for details on constraint solving [Pottier and Rémy, 2005]. See Jones [1999] for a different presentation of type inference, in the context of Haskell.

In all reasonable strategies, the left-hand side of a let constraint is simplified *before* the let form is expanded away.

This corresponds, in Algorithm $\mathcal{J}$, to computing a principal type scheme before examining the right-hand side of a let construct.

Type inference for ML is DEXPTIME-complete [Kfoury et al., 1990, Mairson, 1990], so any constraint solver has exponential complexity.

Nevertheless, under the hypotheses that *types have bounded size* and let forms have bounded left-nesting depth, constraints can be solved in linear time [McAllester, 2003].

This explains why ML type inference *works well in practice.*

Contents

Damas and Milner's type system has *principal types:* at least in the core language, no type information is required.

This is very lightweight, but a bit *extreme:* sometimes, it is useful to write types down, and use them as *machine-checked documentation.*

Let us, then, allow programmers to *annotate* a term with a type:

$$t ::= \ldots \mid (t : T)$$

Typing and constraint generation are obvious:

$$\text{Annot}\quad \frac{\Gamma \vdash t : T}{\Gamma \vdash (t : T) : T} \qquad\qquad [\![(t : T) : T']\!] = [\![t : T]\!] \wedge T = T'$$

Type annotations are *erased* prior to runtime, so the operational semantics is not affected. (Why is erasure sound?)

The constraint $[\![(t : T) : T']\!]$ *implies* the constraint $[\![t : T']\!]$.

That is, in terms of type inference, *type annotations are restrictive:* they lead to a principal type that is less general, and possibly even to ill-typedness.

For instance, $\lambda x.x$ has principal type scheme $\forall X.X \rightarrow X$, whereas $(\lambda x.x : \mathsf{int} \rightarrow \mathsf{int})$ has principal type scheme $\mathsf{int} \rightarrow \mathsf{int}$.

Type variables within type annotations?

Does it make sense for a type annotation to contain a type variable, as in, say:

$$(\lambda x.x : X \longrightarrow X)$$
$$(\lambda x.x + 1 : X \longrightarrow X)$$
$$\text{let } f = (\lambda x.x : X \longrightarrow X) \text{ in } (f\ 0, f\ \text{true})$$

If so, what does it mean?

Type variables within type annotations?

Does it make sense for a type annotation to contain a type variable, as in, say:

$$(\lambda x.x : X \rightarrow X)$$
$$(\lambda x.x + 1 : X \rightarrow X)$$
$$\text{let } f = (\lambda x.x : X \rightarrow X) \text{ in } (f\ 0, f\ \text{true})$$

If so, what does it mean?

Short answer: it does not mean anything, because X is unbound. *"There is no such thing as a free variable"* (Alan Perlis).

A longer answer is, it is necessary to specify *how* and *where* type variables are bound.

How is X bound?

If X is *existentially* bound, or *flexible,* then both $(\lambda x.x : X \longrightarrow X)$ and $(\lambda x.x + 1 : X \longrightarrow X)$ should be well-typed.

If it is *universally* bound, or *rigid,* only the former should be well-typed.

Where is X bound?

If X is bound *within* the left-hand side of this "let" construct, then this code:

$$\text{let } f = (\lambda x.x : X \longrightarrow X) \text{ in } (f\ 0, f \text{ true})$$

should be well-typed.

On the other hand, if X is bound *outside* this "let" form, then this code should be ill-typed, since no *single* ground value of X is suitable.

Let's allow programmers to *explicitly bind* type variables:

$$t ::= \ldots \mid \exists\bar{X}.t \mid \forall\bar{X}.t$$

It now makes sense for a type annotation $(t : T)$ to contain free type variables.

Terms t can now contain free type variables, so some side conditions have to be updated (e.g., $\bar{X} \# \Gamma, t$ in *Gen*).

The typing rules are as follows:

$$\text{Exists}\quad \frac{\Gamma \vdash [\vec{X} \mapsto \vec{T}]t : T}{\Gamma \vdash \exists \bar{X}.t : T} \qquad \text{Forall}\quad \frac{\Gamma \vdash t : T \qquad \bar{X} \# \Gamma}{\Gamma \vdash \forall \bar{X}.t : \forall \bar{X}.T} \qquad \left(\text{Gen}\quad \frac{\Gamma \vdash t : T \qquad \bar{X} \# \Gamma, t}{\Gamma \vdash t : \forall \bar{X}.T} \right)$$

Again, these constructs are erased prior to runtime. (Why is this sound?)

Constraint generation for the existential form is straightforward:

$$[\![(\exists\bar{X}.t) : T]\!] \quad = \quad \exists\bar{X}.[\![t : T]\!] \quad \text{if } \bar{X} \# T$$

The type annotations inside t contain free occurrences of $\bar{X}$. Thus, the constraint $[\![t : T]\!]$ contains such occurrences as well. They are bound by the existential quantifier.

For instance, the expression:

$$\lambda x_1.\lambda x_2.\exists X.((x_1 : X), (x_2 : X))$$

has principal type scheme $\forall X.X \rightarrow X \rightarrow X \times X$. Indeed, the generated constraint contains the pattern:

$$\exists X.([\![x_1 : X]\!] \wedge [\![x_2 : X]\!] \wedge \ldots)$$

which requires x_1 and x_2 to *share* a common (unspecified) type.

Constraint generation: universal case

A term t has type scheme, say, $\forall X.X \rightarrow X$ if and only if t has type $X \rightarrow X$ *for every instance of X*, or, equivalently, for an abstract X.

To express this in terms of constraints, we introduce *universal quantification* in the constraint language:

$$C ::= \ldots \mid \forall X.C$$

Its interpretation is standard.

The need for universal quantification arises when polymorphism is *required* by the programmer, as opposed to *inferred* by the system.

Constraint generation: universal case

Constraint generation for the universal form is somewhat more subtle. A naïve definition *fails:*

$$[\![\forall \bar{X}.t : T]\!] = \forall \bar{X}.[\![t : T]\!] \qquad \text{if } \bar{X} \# T$$

This requires T to be simultaneously equal to *all* of the types that t assumes when $\bar{X}$ varies.

For instance, with this incorrect definition, one would have:

$$\begin{array}{rcl} [\![\forall X.(\lambda x.x : X \rightarrow X) : \mathsf{int} \rightarrow \mathsf{int}]\!] & = & \forall X.[\![(\lambda x.x : X \rightarrow X) : \mathsf{int} \rightarrow \mathsf{int}]\!] \\ & \equiv & \forall X.([\![\lambda x.x : X \rightarrow X]\!] \wedge X = \mathsf{int}) \\ & \equiv & \forall X.(\mathsf{true} \wedge X = \mathsf{int}) \\ & \equiv & \mathsf{false} \end{array}$$

A correct definition is:

$$[\![\forall \bar{X}.t : T]\!] = \forall \bar{X}.\exists Z.[\![t : Z]\!] \wedge \exists \bar{X}.[\![t : T]\!]$$

This requires t to be well-typed *for all* instances of $\bar{X}$ and requires T to be a valid type for t under *some* instance of $\bar{X}$.

A problem with this definition is...

A correct definition is:

$$[\![\forall\bar{X}.t : T]\!] = \forall\bar{X}.\exists Z.[\![t : Z]\!] \wedge \exists\bar{X}.[\![t : T]\!]$$

This requires t to be well-typed *for all* instances of $\bar{X}$ and requires T to be a valid type for t under *some* instance of $\bar{X}$.

A problem with this definition is...

The term t is duplicated! This can lead to exponential complexity. Fortunately, this can be avoided modulo a slight extension of the constraint language [Pottier and Rémy, 2003, p. 112].

Annotating a term with a *type scheme*, rather than just a type, is now just syntactic sugar:

$$(t : \forall \bar{X}.T) \quad \text{stands for} \quad \forall \bar{X}.(t : T) \quad \text{if } \bar{X} \# t$$

In that particular case, constraint generation is in fact simpler:

$$[\![(t : \forall \bar{X}.T) : T']\!] \equiv \forall \bar{X}.[\![t : T]\!] \wedge (\forall \bar{X}.T) \preceq T'$$

(Exercise: check this equivalence.)

A *correct* example:

$$\begin{array}{rl} & [\![(\exists X.(\lambda x.x + 1 : X \to X)) : \mathsf{int} \to \mathsf{int}]\!] \\ = & \exists X.[\![(\lambda x.x + 1 : X \to X) : \mathsf{int} \to \mathsf{int}]\!] \\ \equiv & \exists X.(X = \mathsf{int}) \\ \equiv & \mathit{true} \end{array}$$

The system *infers* that X must be int. Because X is a local type variable, it *does not* appear in the final constraint.

An incorrect example:

$$
\begin{array}{ll}
 & [\![(\forall X.(\lambda x.x+1 : X \longrightarrow X)) : \mathsf{int} \longrightarrow \mathsf{int}]\!] \\
\Vdash & \forall X.\exists Z.[\![(\lambda x.x+1 : X \longrightarrow X) : Z]\!] \\
\equiv & \forall X.\exists Z.(X = \mathsf{int} \wedge X \longrightarrow X = Z) \\
\equiv & \forall X.X = \mathsf{int} \\
\equiv & \mathsf{false}
\end{array}
$$

The system *checks* that X is used in an abstract way, which is not the case here, since the code implicitly assumes that X is int.

A *correct* example:

$$
\begin{aligned}
& [\![(\forall X.(\lambda x.x : X \to X)) : \mathrm{int} \to \mathrm{int}]\!] \\
= \quad & \forall X.\exists Z.[\![(\lambda x.x : X \to X) : Z]\!] \wedge \exists X.[\![(\lambda x.x : X \to X) : \mathrm{int} \to \mathrm{int}]\!] \\
\equiv \quad & \forall X.\exists Z.X \to X = Z \wedge \exists X.X = \mathrm{int} \\
\equiv \quad & \mathrm{true}
\end{aligned}
$$

The system *checks* that X is used in an abstract way, which is indeed the case here.

It also checks that, if X is appropriately instantiated, the *code* admits the expected type $\mathrm{int} \to \mathrm{int}$.

An incorrect example:

$$\begin{array}{ll} & [\![\exists X.(\mathsf{let}\ f = (\lambda x.x : X \rightarrow X)\ \mathsf{in}\ (f\ 0, f\ \mathsf{true})) : Z]\!] \\ \equiv & \exists X.(\mathsf{let}\ f : X \rightarrow X\ \mathsf{in}\ \exists Z_1 Z_2.(f \preceq \mathsf{int} \rightarrow Z_1 \wedge f \preceq \mathsf{bool} \rightarrow Z_2 \wedge Z_1 \times Z_2 = Z)) \\ \equiv & \exists X Z_1 Z_2.(X \rightarrow X = \mathsf{int} \rightarrow Z_1 \wedge X \rightarrow X = \mathsf{bool} \rightarrow Z_2 \wedge Z_1 \times Z_2 = Z) \\ \Vdash & \exists X.(X = \mathsf{int} \wedge X = \mathsf{bool}) \\ \equiv & \mathsf{false} \end{array}$$

X is bound *outside* the let construct; f receives the monotype $X \rightarrow X$.

A *correct* example:

$$
\begin{array}{rl}
& [\![\text{let } f = \exists X.(\lambda x.x : X \rightarrow X) \text{ in } (f\ 0, f\ \text{true}) : Z]\!] \\
\equiv & \text{let } f : \forall Y[\exists X.(X \rightarrow X = Y)].Y \text{ in} \\
& \quad \exists Z_1 Z_2.(f \preceq \text{int} \rightarrow Z_1 \wedge f \preceq \text{bool} \rightarrow Z_2 \wedge Z_1 \times Z_2 = Z) \\
\equiv & \text{let } f : \forall X.X \rightarrow X \text{ in} \\
& \quad \exists Z_1 Z_2.(\ldots) \\
\equiv & \exists Z_1 Z_2.(\text{int} = Z_1 \wedge \text{bool} = Z_2 \wedge Z_1 \times Z_2 = Z) \\
\equiv & \text{int} \times \text{bool} = Z
\end{array}
$$

X is bound *within* the let construct; the term $\exists X.(\lambda x.x : X \rightarrow X)$ has the same principal type scheme as $\lambda x.x$, namely $\forall X.X \rightarrow X$; f receives the type scheme $\forall X.X \rightarrow X$.

Type annotations in the real world

For historical reasons, in Objective Caml, type variables are not explicitly bound. (In my opinion, that's *bad!*) They are implicitly *existentially* bound at the nearest enclosing toplevel let construct.

In Standard ML, type variables are implicitly *universally* bound at the nearest enclosing toplevel let construct.

In Glasgow Haskell, type variables are implicitly existentially bound within patterns: *'A pattern type signature brings into scope any type variables free in the signature that are not already in scope'* [Peyton Jones and Shields, 2004].

Constraints help understand these varied design choices uniformly.

Recall the typing rule for recursive functions (November 25, 2008):

$$\text{FixAbs}\quad \frac{\Gamma; f : T \vdash \lambda x.t : T}{\Gamma \vdash \mu f.\lambda x.t : T}$$

It leads to the following derived typing rule:

$$\text{LetRec}\quad \frac{\Gamma; f : T_1 \vdash \lambda x.t_1 : T_1 \qquad \bar{X} \# \Gamma, t_1 \qquad \Gamma; f : \forall \bar{X}.T_1 \vdash t_2 : T_2}{\Gamma \vdash \text{let rec } f\ x = t_1 \text{ in } t_2 : T_2}$$

Any comments?

These rules require occurences of f to have *monomorphic type* within the recursive definition (that is, within $\lambda x.t_1$).

This is visible also in terms of type inference. The constraint

$$[\![\text{let rec } f\ x = t_1 \text{ in } t_2 : T]\!]$$

is *equivalent* to

$$\text{let } f : \forall XY[\text{let } f : X \longrightarrow Y; x : X \text{ in } [\![t_1 : Y]\!]].X \longrightarrow Y \text{ in } [\![t_2 : T]\!]$$

This is problematic in some situations, most particularly when defining functions over *nested algebraic data types* [Bird and Meertens, 1998, Okasaki, 1999].

This problem is solved by introducing *polymorphic recursion*, that is, by allowing μ-bound variables to receive a polymorphic type scheme:

$$\text{FixAbsPoly}\quad \frac{\Gamma; f : S \vdash \lambda x.t : S}{\Gamma \vdash \mu f.\lambda x.t : S} \qquad \text{LetRecPoly}\quad \frac{\Gamma; f : S \vdash \lambda x.t_1 : S \qquad \Gamma; f : S \vdash t_2 : T}{\Gamma \vdash \text{let rec } f\ x = t_1 \text{ in } t_2 : T}$$

This extension is due to Mycroft [1984].

Polymorphic recursion alters, to some extent, Damas and Milner's type system.

Now, not only *let-bound*, but also *μ-bound* variables receive type schemes. The type system is no longer equivalent, up to reduction to let-normal form, to simply-typed λ-calculus.

This has two consequences:

- *monomorphization*, a technique employed in some ML compilers [Tolmach and Oliva, 1998, Cejtin et al., 2007], is no longer possible;

Polymorphic recursion alters, to some extent, Damas and Milner's type system.

Now, not only *let-bound*, but also *μ-bound* variables receive type schemes. The type system is no longer equivalent, up to reduction to let-normal form, to simply-typed λ-calculus.

This has two consequences:

- *monomorphization*, a technique employed in some ML compilers [Tolmach and Oliva, 1998, Cejtin et al., 2007], is no longer possible;
- *type inference* becomes problematic!

Type inference for ML with polymorphic recursion is undecidable [Henglein, 1993]. It is *equivalent* to the undecidable problem of *semi-unification.*

Yet, type inference in the presence of polymorphic recursion can be made simple. (How?)

Yet, type inference in the presence of polymorphic recursion can be made simple. (How?)

By relying on a *mandatory type annotation*. The rules become:

FixAbsPoly

$$\frac{\Gamma; f : S \vdash \lambda x.t : S}{\Gamma \vdash \mu(f : S).\lambda x.t : S}$$

LetRecPoly

$$\frac{\Gamma; f : S \vdash \lambda x.t_1 : S \qquad \Gamma; f : S \vdash t_2 : T}{\Gamma \vdash \text{let rec } (f : S) = \lambda x.t_1 \text{ in } t_2 : T}$$

The type scheme S no longer has to be guessed.

The constraint generation rule becomes:

$$[\![\text{let rec } (f : S) = \lambda x.t_1 \text{ in } t_2 : T]\!] = ?$$

The constraint generation rule becomes:

$$[\![\text{let rec } (f : S) = \lambda x.t_1 \text{ in } t_2 : T]\!] = \text{let } f : S \text{ in } ([\![\lambda x.t_1 : S]\!] \wedge [\![t_2 : T]\!])$$

It is clear that f receives type scheme S both *inside and outside* of the recursive definition.

Contents

We extend the basic unification algorithm with support for universal quantification.

The solved forms are unchanged: universal quantifiers are always *eliminated.*

In short, in order to reduce $\forall \bar{X}.C$ to a solved form, where C is itself a solved form:

- if a rigid variable is equated with a constructed type, fail;
- if two rigid variables are equated, fail;
- if a free variable dominates a rigid variable, fail;
- otherwise, one can decompose C as $\exists \bar{Y}.(C_1 \wedge C_2)$, where $\bar{X}\bar{Y} \# C_1$ and $\exists \bar{Y}.C_2 \equiv \text{true}$; in that case, $\forall \bar{X}.C$ reduces to just C_1.

See [Pottier and Rémy, 2003, p. 109] for details.

Objective Caml implements a form of unification under a mixed prefix:

```
$ ocaml
# let module M : sig val id : 'a -> 'a end
               = struct let id x = x + 1 end
  in M.id;;
Values do not match: val id : int -> int
is not included in val id : 'a -> 'a
```

This *example gives rise to a constraint of the form* $\forall X.X = \mathrm{int}$.

Here is another *example*:

```
$ ocaml
# let r = ref (fun x -> x) in
  let module M : sig val id : 'a -> 'a end
                = struct let id = !r end
  in M.id;;
Values do not match: val id : '_a -> '_a
is not included in val id : 'a -> 'a
```

This *example gives rise to a constraint of the form* $\exists Y.\forall X.X = Y$.

Contents

Product and sum types alone do not allow describing *data structures* of *unbounded size*, such as lists and trees.

Indeed, if the grammar of types is $T ::= \mathrm{unit} \mid T \times T \mid T + T$, then it is clear that every type describes a *finite* set of values.

For every k, the type of lists of length at most k is expressible using this grammar. However, the type of lists of unbounded length is not.

The following definition is inherently *recursive:*

> *"A list is either empty or a pair of an element and a list."*

We need something like this:

$$\text{list}\ X \quad \diamond \quad \text{unit} + X \times \text{list}\ X$$

But what does $\diamond$ stand for? Is it *equality,* or some kind of *isomorphism?*

Equi- versus iso-recursive types

There are two standard approaches to recursive types, dubbed the *equi-recursive* and *iso-recursive* approaches.

In the equi-recursive approach, a recursive type is *equal* to its unfolding.

In the iso-recursive approach, a recursive type and its unfolding are related via explicit *coercions*.

In the *equi-recursive* approach, the usual syntax of types:

$$T ::= X \mid F\ \vec{T}$$

is no longer interpreted inductively. Instead, types are the *regular trees* built on top of this signature.

Finite syntax for equi-recursive types

If desired, it is possible to use *finite syntax* for recursive types:

$$T ::= X \mid \mu X.(F\ \vec{T})$$

I do not allow the seemingly more general $\mu X.T$, because $\mu X.X$ is meaningless, and $\mu X.Y$ or $\mu X.\mu Y.T$ are useless. If I write $\mu X.T$, it should be understood that T is *contractive*, that is, T is a type constructor application.

For instance, the type of lists of elements of type X is:

$$\mu Y.(\text{unit} + X \times Y)$$

Each type in this syntax denotes a unique regular tree, sometimes known as its *infinite unfolding*. Conversely, every regular tree can be expressed in this notation (possibly in more than one way).

If one builds a type-checker on top of this finite syntax, then one must be able to *decide* whether two types are *equal*, that is, have identical infinite unfoldings.

This can be done efficiently by unification.

One can also prove [Brandt and Henglein, 1998] that equality is the least congruence generated by the following two rules:

Fold/Unfold

$$\mu X.T = [X \mapsto \mu X.T]T$$

Uniqueness

$$\frac{T_1 = [X \mapsto T_1]T \qquad T_2 = [X \mapsto T_2]T}{T_1 = T_2}$$

In both rules, T must be contractive.

This axiomatization does not directly lead to an efficient algorithm for deciding equality, though.

Type soundness for equi-recursive types

In the presence of equi-recursive types, structural induction on types is no longer permitted — but *we never used it* anyway.

It remains true that $F\,\vec{T}_1 = F\,\vec{T}_2$ implies $\vec{T}_1 = \vec{T}_2$ — this was used in our Subject Reduction proofs.

It remains true that $F_1\,\vec{T}_1 = F_2\,\vec{T}_2$ implies $F_1 = F_2$ — this was used in our Progress proofs.

So, the reasoning that leads to *type soundness* is unaffected.

How is type inference adapted for equi-recursive types?

The *syntax* of constraints is unchanged: they remain systems of equations between finite first-order types, without μ's. Their *interpretation* changes: they are now interpreted in a universe of regular trees.

As a result,

- constraint generation is *unchanged;*
- constraint solving is adapted by *removing the occurs check.*

Here is a function that measures the length of a list:

$$\begin{array}{l} \mu length.\lambda xs.\text{case } xs \text{ of} \\ \quad \lambda().0 \\ \quad [] \; \lambda(x, xs).1 + length\ xs \end{array}$$

Type inference gives rise to the *cyclic equation:*

$$Y = \text{unit} + X \times Y$$

where *length* has type $Y \rightarrow \text{int}$.

That is, *length* has *principal type scheme:*

$$\forall X.(\mu Y.\text{unit} + X \times Y) \rightarrow \text{int}$$

or, equivalently, principal constrained type scheme:

$$\forall X[Y = \text{unit} + X \times Y].Y \rightarrow \text{int}$$

The cyclic equation that characterizes lists was never provided by the programmer, but was inferred.

Objective Caml implements *equi-recursive* types upon explicit request:

```
$ ocaml -rectypes
# type ('a, 'b) sum = Left of 'a | Right of 'b;;
type ('a, 'b) sum = Left of 'a | Right of 'b
# let rec length xs =
    match xs with
    | Left () -> 0
    | Right (x, xs) -> 1 + length xs
  ;;
val length : ((unit, 'b * 'a) sum as 'a) -> int = <fun>
```

Quiz: *why is* `-rectypes` *only an option?*

Equi-recursive types are simple and powerful. In practice, however, they are perhaps *too expressive:*

```
$ ocaml -rectypes
# let rec map f = function
  | [] -> []
  | x :: xs -> map f x :: map f xs;;
val map : 'a -> ('b list as 'b) -> ('c list as 'c) = <fun>
# map (fun x -> x + 1) [ 1; 2 ];;
This expression has type int but is used with type 'a list as 'a
# map () [[];[[]]];;
- : 'a list as 'a = [[]; [[]]]
```

Equi-recursive types allow this nonsensical version of map to be accepted, thus delaying the detection of a programmer error.

Quiz: why is this *accepted?*

```
$ ledit ocaml
# let f x = x#hello x;;
val f : (< hello : 'a -> 'b; .. > as 'a) -> 'b = <fun>
```

In the iso-recursive approach, the user is allowed to introduce new *type constructors* D via (possibly mutually recursive) *declarations:*

$$D\,\vec{X} \approx T \qquad \text{(where } \mathrm{ftv}(T) \subseteq \bar{X}\text{)}$$

Each such declaration adds two new *term constants,* whose semantics is the identity:

$$\begin{aligned} \mathsf{fold}_D &: \forall \bar{X}.T \rightarrow D\,\vec{X} \\ \mathsf{unfold}_D &: \forall \bar{X}.D\,\vec{X} \rightarrow T \end{aligned}$$

A parameterized, iso-recursive type of lists is:

$$\text{list}\ X \approx \text{unit} + X \times \text{list}\ X$$

The empty list is:

$$\text{fold}_{\text{list}}\ (\text{inj}_1\ ()) : \forall X.\text{list}\ X$$

A function that measures the length of a list is:

$$\left(\begin{array}{l} \mu length.\lambda xs.case\ (\text{unfold}_{\text{list}}\ xs)\ of \\ \qquad \lambda().0 \\ \qquad [\!]\ \lambda(x, xs).1 + length\ xs \end{array}\right) : \forall X.\text{list}\ X \longrightarrow \text{int}$$

One *folds upon construction* and *unfolds upon deconstruction.*

Type inference for iso-recursive types

In the iso-recursive approach, *types remain finite.* The type list X is just an application of a type constructor to a type variable.

As a result, *type inference is unaffected.* The occurs check remains.

Contents

Algebraic data types result of the fusion of iso-recursive types with structural, labelled products and sums.

This suppresses the *verbosity* of explicit folds and unfolds as well as the *fragility* and inconvenience of numeric indices — instead, named *record fields* and *data constructors* are used.

For instance,

$$\mathsf{fold}_{\mathsf{list}}\ (\mathsf{inj}_1\ ()) \quad \text{is replaced with} \quad \mathit{Nil}\ ()$$

An algebraic data type constructor D is introduced via a *record type* or *variant type* definition:

$$D\,\vec{X} \approx \prod_{\ell \in L} \ell : T_\ell \qquad \text{or} \qquad D\,\vec{X} \approx \sum_{\ell \in L} \ell : T_\ell$$

L denotes a finite set of record labels or data constructors.

Algebraic data type definitions can be mutually recursive.

Effects of a record type declaration

The record type definition $D\,\vec{X} \approx \prod_{\ell \in L} \ell : T_\ell$ introduces syntax for *constructing* and *destructing* records:

$$t ::= \ldots \mid \{\ell = t_\ell\}_{\ell \in L} \mid t.\ell$$

The typing rules are:

$$\textsc{Record}\quad \frac{\forall \ell \in L, \quad \Gamma \vdash t_\ell : [\vec{X} \mapsto \vec{T}]T_\ell}{\Gamma \vdash \{\ell = t_\ell\}_{\ell \in L} : D\,\vec{T}}$$

$$\textsc{Get}\quad \frac{\Gamma \vdash t : D\,\vec{T}}{\Gamma \vdash t.\ell : [\vec{X} \mapsto \vec{T}]T_\ell}$$

The variant type definition $D\,\vec{X} \approx \sum_{\ell \in L} \ell : T_\ell$ introduces this syntax:

$$t ::= \ldots \mid \ell\; t \mid \mathsf{case}\; t\; \mathsf{of}\; [v_\ell]_{\ell \in L}$$

The typing rules are:

Data

$$\frac{\Gamma \vdash t : [\vec{X} \mapsto \vec{T}]T_\ell}{\Gamma \vdash \ell\; t : D\,\vec{T}}$$

Case

$$\frac{\Gamma \vdash t : D\,\vec{T} \qquad \forall \ell \in L, \quad \Gamma \vdash v_\ell : [\vec{X} \mapsto \vec{T}]T_\ell \rightarrow T}{\Gamma \vdash \mathsf{case}\; t\; \mathsf{of}\; [v_\ell]_{\ell \in L} : T}$$

Here is an algebraic data type of lists:

$$\text{list } X \approx \mathit{Nil} : \text{unit} + \mathit{Cons} : X \times \text{list } X$$

This gives rise to:

$$\Gamma \vdash \mathit{Nil}\ () : \text{list } T \qquad \frac{\Gamma \vdash t_1 : T \qquad \Gamma \vdash t_2 : \text{list } T}{\Gamma \vdash \mathit{Cons}\ (t_1, t_2) : \text{list } T}$$

$$\frac{\begin{array}{c}\Gamma \vdash t : \text{list } T_1 \\ \Gamma \vdash v_1 : \text{unit} \rightarrow T_2 \qquad \Gamma \vdash v_2 : T_1 \times \text{list } T_1 \rightarrow T_2\end{array}}{\Gamma \vdash \mathit{case}\ t\ \mathit{of}\ (\mathit{Nil} : v_1 \ [\!]\ \mathit{Cons} : v_2) : T_2}$$

A function that measures the length of a list is:

$$\left(\begin{array}{l} \mu length.\lambda xs.\mathsf{case}\ xs\ \mathsf{of} \\ \qquad Nil : \lambda().0 \\ \quad [\!]\ Cons : \lambda(x, xs).1 + length\ xs \end{array}\right) : \forall X.\mathsf{list}\ X \longrightarrow \mathsf{int}$$

In Objective Caml, a record field can be marked *mutable*. This introduces extra syntax for writing this field:

$$\text{Set}\quad \frac{\Gamma \vdash t_1 : D\,\vec{T} \qquad \Gamma \vdash t_2 : [\vec{X} \mapsto \vec{T}]T_\ell}{\Gamma \vdash t_1.\ell \leftarrow t_2 : \mathsf{unit}}$$

This also makes $\{\ell = t_\ell\}_{\ell \in L}$ a memory allocation expression, *not a value*, so, due the value restriction, the type of such an expression can never be generalized.

(Most titles are clickable links to online versions.)

Bird, R. and Meertens, L. 1998.
Nested datatypes.
In *International Conference on Mathematics of Program Construction (MPC)*. Lecture Notes in Computer Science, vol. 1422. Springer Verlag, 52–67.

Brandt, M. and Henglein, F. 1998.
Coinductive axiomatization of recursive type equality and subtyping.
Fundamenta Informaticæ 33, 309–338.

Cejtin, H., Fluet, M., Jagannathan, S., and Weeks, S. 2007.
The MLton compiler.

Bibliography]Bibliography

Henglein, F. 1993.
Type inference with polymorphic recursion.
ACM Transactions on Programming Languages and Systems 15, 2 (Apr.), 253–289.

Jim, T. 1995.
What are principal typings and what are they good for?
Tech. Rep. MIT/LCS TM-532, Massachusetts Institute of Technology. Aug.

Jones, M. P. 1999.
Typing Haskell in Haskell.
In *Haskell workshop*.

[][

Kfoury, A. J., Tiuryn, J., and Urzyczyn, P. 1990.
ML typability is DEXPTIME-complete.
In *Colloquium on Trees in Algebra and Programming*. Lecture Notes in Computer Science, vol. 431. Springer Verlag, 206–220.

Mairson, H. G. 1990.
Deciding ML typability is complete for deterministic exponential time.
In *ACM Symposium on Principles of Programming Languages (POPL)*. 382–401.

[][

McAllester, D. 2003.
A logical algorithm for ML type inference.
In *Rewriting Techniques and Applications (RTA)*. Lecture Notes in Computer Science, vol. 2706. Springer Verlag, 436–451.

Milner, R. 1978.
A theory of type polymorphism in programming.
Journal of Computer and System Sciences 17, 3 (Dec.), 348–375.

Mycroft, A. 1984.
Polymorphic type schemes and recursive definitions.
In *International Symposium on Programming*. Lecture Notes in Computer Science, vol. 167. Springer Verlag, 217–228.

[][

Okasaki, C. 1999.
Purely Functional Data Structures.
Cambridge University Press.

Peyton Jones, S. and Shields, M. 2004.
Lexically-scoped type variables.
Manuscript.

Pottier, F. 2002.
Notes du cours de DEA "Typage et Programmation".

Pottier, F. and Rémy, D. 2003.
The essence of ML type inference.
Draft of an extended version. Unpublished.

[][

Pottier, F. and Rémy, D. 2005.
The essence of ML type inference.
In *Advanced Topics in Types and Programming Languages*, B. C. Pierce, Ed. MIT Press, Chapter 10, 389–489.

Tolmach, A. and Oliva, D. P. 1998.
From ML to Ada: Strongly-typed language interoperability via source translation.
Journal of Functional Programming 8, 4 (July), 367–412.

[][

Wells, J. B. 2002.
The essence of principal typings.
In *International Colloquium on Automata, Languages and Programming*. Lecture Notes in Computer Science, vol. 2380. Springer Verlag, 913–925.