The Language of Types

In modern programming languages the types are rich enough to require a little language of their own. Here we describe such a type language for an ML-like language.

Basic Types: int / bool / string / char / real

Now we can build compound types; in order to do this we use type constructors: these are constructs to create new types from old ones:

$*$: product $\to$: function space

list | : sum

I am ignoring array, record and other things for now.

Let us consider $*$: this is the counterpart of the term constructor (,).

Given two types say int & bool we define a new type int $*$ bool. How do we connect terms of the language & types? Through typing rules. I will use letters like f, e, t etc for terms & greek letters α, β for types.

A typing rule has the form

$$\frac{term_1 : typ_1, \; \cdots \; term_k : typ_k}{term : typ}$$

We use type variables to have generic rules. Here is the rule for $*$:

$$\frac{t_1 : \tau_1, \quad t_2 : \tau_2}{(t_1, t_2) : \tau_1 * \tau_2}$$

How do we read such a rule? In words: __if__ you have __shown__ that t_1 has type τ_1 & that t_2 has type τ_2 __then__ you can conclude that the pair (t_1, t_2) has type (τ_1, τ_2).

How do we start off? We need some basic assumptions for the basic types. Here, for example, are the rules for __int__:

$$\frac{}{0:\text{int}} \quad \frac{}{1:\text{int}} \quad \frac{}{2:\text{int}} \quad \cdots \quad \frac{}{\sim 1:\text{int}} \quad \cdots$$

Why draw a line with nothing on top? To indicate that we need no further assumptions: 0 has type int, no assumptions are needed to conclude this. We can now type more complicated expressions with more rules:

$$\frac{x:\text{int}\ ,\ y:\text{int}}{x+y:\text{int}} \qquad \frac{x:\text{int}\ ,\ y:\text{int}}{x*y:\text{int}} \quad \cdots$$

this is not the complete set of rules for int, but it gives the idea. Here are some rules for bool

$$\frac{}{\text{true}:\text{bool}} \qquad \frac{}{\text{false}:\text{bool}} \qquad \frac{b:\text{bool}}{\text{not } b:\text{bool}} \quad \cdots$$

$$\frac{x:\text{int},\ y:\text{int}}{x=y:\text{bool}}$$

Here is the rule for ~~to~~ conditionals

$$\frac{e_1:\tau\ ,\ e_2:\tau,\ b:\text{bool}}{\text{if } b \text{ then } e_1 \text{ else } e_2 : \tau}$$

This says that both branches must have the same type τ, the conditions must be a boolean and then the overall conditional will have type τ.

Now some rules for tuples and projections

$$\frac{t_1 : \tau_1 \quad , \quad t_2 : \tau_2}{(t_1, t_2) : \tau_1 * \tau_2}$$

We can use this with any types, e.g.

$$\frac{17 : int \quad , \quad false : bool}{(17, false) : int * bool}$$

We can make nested pairs too

$$\frac{(17, false) : int * bool \qquad 2+3 : int}{((17, false), 2+3) : (int * bool) * int}$$

Note we are assigning types to expressions not just to values. We can understand pattern matching as follows

match (x, y) with (17, false)

$$\frac{(17, false) : int * bool}{17 : int \quad , \quad false : bool}$$

We have destructors in F# (not much used)

$$\frac{(1729, \text{"Hardy"}) : int * string}{\begin{array}{c} fst\ (1729, \text{"Hardy"}) : int \\ snd\ (1729, \text{"Hardy"}) : string \end{array}}$$

Note the type system does not spell out the computation rules. For this we need a different kind of presentation called operational semantics.

e.g. snd (1729, "Hardy") $\longrightarrow$ "Hardy"

We will not discuss operational semantics.

Here is a little typing "tree"

$$\frac{\dfrac{17 : int \quad true : bool}{(17, bool) : int * bool} \qquad \dfrac{2 : int \quad 3 : int}{2+3 : int}}{((17, bool), 2+3) : (int * bool) * int}.$$

Conceptually the type-checker builds such trees.

Lists : list is a type constructor, :: is a term constructor, nil or [] is a constant and hd, tl are the destructors. Here are the rules

$$\frac{e : \tau \qquad l : \tau\text{-list}}{e :: l : \tau\text{-list}}$$

$$\frac{l : \tau\text{-list}}{hd(l) : \tau} \qquad \frac{l : \tau\text{-list}}{tl(l) : \tau\text{-list}}$$

Note the type system will not tell you what happens if you apply hd to nil. The type rule says this is OK but in reality an exception is raised.

Now for the most important type constructor: $\rightarrow$ However we need to deal with variables first because a parameter is a crucial part of f^n. How does a variable get a type? For now, we assume that variables are typed by declarations. Later we will see that polymorphic types can be inferred. We will need to keep track of these declarations when making typing judgments. We have a set of assumptions (or declarations) called a context. For example we might have

$n : int,\ x : bool,\ y : int * int, \ldots$

We will use Γ for a context. Our judgement will look like $\Gamma \vdash e : \tau$

Using the type assumptions (or declarations) in Γ we conclude that the expression e has the type τ. We can add these contexts to our previous rules & use them otherwise unchanged.

$$\frac{\Gamma \vdash e : \text{bool} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : \tau}$$

Before we deal with functions let us consider let expression which introduce new bindings. We need to manipulate Γ

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

This says the obvious thing: if $x : \tau$ is one of the assumptions in Γ then one can conclude $x : \tau$.

Now for let

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 \ \cancel{\text{end}} : \tau_2}$$

x must be fresh in Γ

Here is an example:

$$\dfrac{\dfrac{}{\vdash 5 : \text{int}} \qquad \dfrac{\dfrac{x : \text{int} \vdash x : \text{int} \qquad x : \text{int} \vdash 2 : \text{int}}{x : \text{int} \vdash x + 2 : \text{int}} \qquad \dfrac{\overbrace{x : \text{int}, y : \text{int}}^{\Gamma} \vdash x : \text{int} \qquad \Gamma \vdash y : \text{int}}{x : \text{int}, y : \text{int} \vdash x + y : \text{int}}}{x : \text{int} \vdash \text{let } y = x + 2 \text{ in } x + y \ \cancel{\text{end}} : \text{int}}}{\vdash \text{let } x = 5 \text{ in } (\text{let } y = x + 2 \text{ in } x + y \ \cancel{\text{end}}) \ \cancel{\text{end}} : \text{int}}$$

<u>Note</u> order of assumptions does not matter, we can rearrange the order at will. We can reuse assumptions as often as we want. We may have unused assumptions. For example, in let $x = 5$ in 3 ~~end~~ : int we do not need any assumption about x to type check the body $3 : \text{int}$.

Now for the all important $\rightarrow$ constructor. If τ_1, τ_2 are types then $\tau_1 \rightarrow \tau_2$ ~~are types~~ is a type. We have a <u>term</u> constructor fun $x \rightarrow \cdots$ and a "destructor" i.e. function application.

Here are the rules:

$$\frac{\Gamma, x:\tau_1 \vdash e:\tau_2}{\Gamma \vdash \text{fun } x \rightarrow e : \tau_1 \rightarrow \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\, e_2 : \tau_2}$$

Here are some examples of type derivations

$$\frac{x:\text{int} \vdash x:\text{int}}{\vdash \text{fun } x \rightarrow x : \text{int} \rightarrow \text{int}} \qquad \frac{x:\text{string} \vdash x:\text{string}}{\vdash \text{fun } x \rightarrow x : \text{string} \rightarrow \text{string}}$$

The same expression has multiple types! We will discuss polymorphism later. For now we will think monomorphically.

Some more examples:

let $x = 1$
let $f =$ fun $u \rightarrow u + x$
let $y = 2$
$f\ y$

$\Gamma = x:\text{int},\ y:\text{int},\ u:\text{int},\ f:\text{int} \rightarrow \text{int}$

I will show the typing derivation in pieces

$$\cfrac{\cfrac{\Gamma \vdash u:\text{int} \qquad \Gamma \vdash x:\text{int}}{\Gamma \vdash u + x : \text{int}}}{x,\cdots, y, \cdots f \cdots \vdash \text{fun } u \rightarrow u + x : \text{int} \rightarrow \text{int}} \qquad \frac{\Gamma \vdash f:\text{int} \rightarrow \text{int} \quad \Gamma \vdash y:\text{int}}{\Gamma \vdash f\, y : \text{int}}$$

ABBREVIATED TO FIT HERE $\longrightarrow$

$$\frac{y:\text{int},\ f:\text{int} \rightarrow \text{int},\ x:\text{int} \vdash f\, y:\text{int} \qquad \Gamma \vdash 2:\text{int}}{f:\text{int} \rightarrow \text{int},\ \cancel{y}\ x:\text{int} \vdash \text{let } y = 2 \text{ in } f\, y : \text{int}}$$

$$\vdots$$

$$\frac{}{x:\text{int} \vdash \begin{array}{l}\text{let } f = \text{fun } u \rightarrow u + x \\ \text{let } y = 2 \text{ in } f\,y\end{array} : \text{int} \ \cancel{\rightarrow \text{int}}}$$