The two Camlp4s

Jeremy Yallop

Copenhagen
September 2012

Camlp4 the code generator

$$\textbf{type}\ \alpha\ \text{tree} = \text{Empty} \mid \text{Branch}\ \textbf{of}\ (\alpha\ \text{tree} \times \alpha \times \alpha\ \text{tree})$$
$$\quad \textbf{with}\ \text{sexp}$$

Camlp4 the code generator

```
type α tree = Empty | Branch of (α tree × α × α tree)
  with sexp
```

```
let rec sexp_of_tree of_a = function
| Empty → Sexp.Atom "Empty"
| Branch (v1, v2, v3) → let v1 = (let v1 = sexp_of_tree of_a v1
                                  and v2 = of_a v2
                                  and v3 = sexp_of_tree of_a v3
                                  in Sexp.List [ v1; v2; v3 ])
                        in Sexp.List [ Sexp.Atom "Branch"; v1 ]
```

Camlp4 the syntax extender

```
let rec cons : α. α × α queue → α queue = function
  | x, Deep {f = Three (a, b, c); m = lazy m;                    r} →
      Deep {f = Two (x, a);       m = lazy (cons ((b, c), m)); r}
  | ...
```

Camlp4 the syntax extender

```
let rec cons : α. α × α queue → α queue = function
  | x, Deep {f = Three (a, b, c); m = lazy m;                    r} →
      Deep {f = Two (x, a);       m = lazy (cons ((b, c), m)); r}
  | ...
```

```
let cons = object (self)
 method cons : α. α × α queue → α queue = function
 | x, Deep {f = Three (a, b, c); m = m;                                  r = r}
  → Deep {f = Two (x, a);       m = lazy (self#cons ((b, c), force m)); r = r}
 | ...
end
```

Extending syntax is hard

$$\texttt{e} \quad \rightsquigarrow \quad \textbf{let}\ \texttt{x} = \texttt{force}\ \texttt{y}\ \textbf{in}\ \texttt{e}[\texttt{y} := \texttt{x}]$$

Extending syntax is hard

$$\texttt{e} \quad \rightsquigarrow \quad \textbf{let}\ \texttt{x} = \texttt{force}\ \texttt{y}\ \textbf{in}\ \texttt{e}[\texttt{y} := \texttt{x}]$$

$$\texttt{y + 1} \quad \rightsquigarrow \quad \textbf{let}\ \texttt{x} = \texttt{force}\ \texttt{y}\ \textbf{in}\ \texttt{x + 1}$$

Extending syntax is hard

$$\texttt{e} \quad \rightsquigarrow \quad \textbf{let}\ \texttt{x} = \texttt{force}\ \texttt{y}\ \textbf{in}\ \texttt{e}[\texttt{y} := \texttt{x}]$$

$$\texttt{y + 1} \quad \rightsquigarrow \quad \textbf{let}\ \texttt{x} = \texttt{force}\ \texttt{y}\ \textbf{in}\ \texttt{x + 1}$$

$$\texttt{(y, }\ \textbf{fun}\ \texttt{y} \to \texttt{y)} \quad \rightsquigarrow \quad \textbf{let}\ \texttt{x} = \texttt{force}\ \texttt{y}\ \textbf{in}\ \texttt{(x, }\ \textbf{fun}\ \texttt{y} \to \texttt{y)}$$

Extending syntax is hard

$$\texttt{e} \rightsquigarrow \textbf{let}\ \texttt{x} = \texttt{force}\ \texttt{y}\ \textbf{in}\ \texttt{e}[\texttt{y} := \texttt{x}]$$

$$\texttt{y + 1} \rightsquigarrow \textbf{let}\ \texttt{x} = \texttt{force}\ \texttt{y}\ \textbf{in}\ \texttt{x + 1}$$

$$\texttt{(y, }\textbf{fun}\ \texttt{y} \to \texttt{y)} \rightsquigarrow \textbf{let}\ \texttt{x} = \texttt{force}\ \texttt{y}\ \textbf{in}\ \texttt{(x, }\textbf{fun}\ \texttt{y} \to \texttt{y)}$$

```
let module M =
struct
 open External
 let z = y
end
in M.z
```

$\rightsquigarrow$ **?**

Lessons from Coq notation

Integrated into **culture**

Integrated into **tools**

Integrated into **language**