Université de Montréal

Typer
An infix statically typed Lisp

Stefan Monnier

monnier@iro.umontreal.ca

Université de Montréal

with Pierre Delaunay & Vincent Archambault-Bouffard

Infix statically typed Lisp

Lisp $\simeq$ prefix dynamically typed functional programming

$\Longrightarrow$

ML $\simeq$ infix statically typed Lisp

¡BUT!

ML and friends lack macros!

As a consequence, ML can't be minimalist like Scheme

Less support for DSLs

I want it all:

simple core, flexible syntax, dependent types, seamless macros

A taste of Typer

Simple function definition:

```
add1 : Int -> Int;
add1 = lambda x -> x + 1;
```

Can be shortened to:

```
add1 x = x + 1;
```

New type definition:

```
type List (a : Type)
  | nil
  | cons a (List a);
```

Syntactic complexity of macros

Multiple syntactic classes:

expression, instruction, declaration, formal argument, sequence of ...

Example in OCaml:

```
let x = a; b in x = a; b
```

$\simeq$

```
let (x = (a; b)) in ((x = a); b)
```

[And yet, we call it “context free grammar”!]

How then should we parse a macro invocation like:

```
mymacro (x = a; b)
```

Syntactic complexity of macros (cont.)

Options to parse `mymacro (x = a; b)`:

- Delay parsing `mymacro` arguments
  - May still need to parse enough to find boundaries of arguments
- Let `mymacro` specify the class of each argument
  - Imposes tight bond between macros and syntax
- Disallow the problem

Last choice is more restrictive ... more in the spirit of Lisp syntax:

> The relative precedence of = and `;` is always the same, regardless if it's an expression, declaration, ...

[Well, "restriction" or "feature" is in the eye of the beholder]

Operator Precedence Grammar

Old and weak parsing technology (Floyd 1963)

Parsing based on a table of precedences

- Each *keyword* gets two precedences: and left and a right one
- When faced with kw_1 *exp* kw_2 attach *exp* to the higher precedence
- The surrounding context is not taken into account

Constructs made of several keywords

- When right precedence of kw_1 is equal to left precedence of kw_2

Similar to Agda's mixfix parsing

[Fun fact: Can parse backward just as easily!]

Example use of OPG

`[`	•,1	`]`	1,•
`if`	•,2	`then`	2,3
`+`	6,7	`=`	4,5

```
if x = y + 1 then [ 5 ]
if x = y + 1 then ([_] 5)
if x = (_+_ y 1) then ([_] 5)
if (_=_ x (_+_ y 1)) then ([_] 5)
(if_then_ (_=_ x (_+_ y 1)) ([_] 5))
```

S-expressions

Typer's front end is like that of Lisp:

1. Parse the source text using the *reader*
2. Returns an S-expression

```
type Sexp
  | symbol String
  | immediate Imm
  | node Sexp (Lisp Sexp)
```

3. Expand macros and see if the S-expression is a meaningful program

The same *reader* can be used to read non-programs ...

... or programs in other languages (e.g. DSL)

Typer's reader

A declaration like

```
type List (a : Type)
  | nil
  | cons a (List a)
```

is processed by the reader exactly like

```
type_ (_|_ (List (_:_ a Type))
           nil
           (cons a (List a))
```

Contrary to Lisp, parentheses are only used for grouping:

`(nil)` is identical to `nil` rather than to `(_) nil`

Extending the syntax

Typer separates syntactic extensions from macros

Simple primitive to set/change the precedence of operators:

```
define-operator if () 2;
define-operator then 2 3;
define-operator else 3 66;
```

Only affects code that's not yet been parsed!

Syntactic extensions are mixfix syntactic sugar:

$$e_1 + e_2 \equiv \texttt{\_+\_}\ e_1\ e_2$$

$$\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \equiv \texttt{if\_then\_else\_}\ e_1\ e_2\ e_3$$

Defining macros

Like in Lisp, macros take a list of `Sexp` and return a new `Sexp`

```
if_then_else_ =
  macro (lambda args ->
         let e1 = List_nth 0 args Sexp_error;
             e2 = List_nth 1 args Sexp_error;
             e3 = List_nth 2 args Sexp_error;
         in quote (case (uquote e1)
                    | true => (uquote e2)
                    | false => (uquote e3)));
```

where `macro` is a data constructor:

```
macro : (List Sexp -> Sexp) -> Macro;
```

Invoking macros

Macros are expanded from the outside, like in Lisp

Macro calls distinguished by type:

1. For $e_1 \ldots e_n$, lookup type of e_1
2. If type is `Macro`, then it's a macro call

$\Rightarrow$ Need to interleave type inference and macro expansion

Elaboration: Type inference and macro-expansion

Takes an S-expression, returns a core lambda expression with its type:

```
elaborate : Ctx -> Sexp -> Pair Lexp Ltype;
```

Elaboration pseudo-code

```
elaborate : Ctx -> Sexp -> Pair Lexp Ltype;
elaborate c sexp =
case sexp
 | symbol s => elab_variable_reference c s
 | immediate v => elab_immediate_value v
 | node head args =>
   let (e1, t1) = elaborate c head in
   case t1
    | "Macro"
      => elaborate c (macroexpand c e1 args)
    | "Special-Form"
      => elab_special_form c e1 args
    | _ => elab_funcall c t1 e1 args;
```

Bidirectional type-inference

To better propagate existing type information

Usually done by handling core-constructs in either `check` or `infer`:

```
infer : Ctx -> Sexp -> Pair Lexp Ltype;
check : Ctx -> Sexp -> Ltype -> Lexp;
```

Core constructs use *special-forms* rather than hard-coded names, so:

```
elaborate : Ctx -> Sexp -> Option Ltype
          -> Pair Lexp (Option Ltype);
```

[Then define `check` and `infer` on top of `elaborate`]

Expanding macros

`macroexpand` takes a `Lexp` which describes the macro

Usually, this `Lexp` is just a variable reference

⇒ Need to turn this `Lexp` into an executable, closed function

1. Check that it is indeed closed
2. Evaluate in turn all the vars transitively referenced
3. Evaluate the macro itself; extract its function, and call it

Typer is pure: those evaluations have no side-effects and can be cached

Supports anonymous macros and more [tho, not a design goal]

Conclusion

ML-style syntax and semantics

Lisp-style syntactic structure and metaprogramming

Simple core

Syntax extensions independent from macros

Simple, seamless, and powerful macros

Declarations and macros

Mutual recursion à la Haskell does not mesh well with macros:

```
p aul = h ud ak;
john (or mccarty (p eterson));
```

- Is `ud` defined in the expansion of the call to *john*?
- Is `p` a macro that should be expanded in the second line?

Typer's mutual recursion needs explicit annotations:

```
ud : ?;
p aul = h ud ak;
john (or mccarty (p eterson));
```

OPG in practice

We can't have both

```
          x : a -> b    =    x : (a -> b)
  lambda x : a -> b     =    lambda (x : a) -> b
```

Anecdotal evidence from Emacs's SMIE:

Modula-2, Octave, Prolog, Ruby, sh, CSS, SML, OCaml, Coq, ...

Example problems for SML syntax: `=` `|` `of` `val`

OPG focuses on *finding* a structure, not *checking* it:

`"if A (C then D"` ≡ `"if_ A (\(_ (_then_ C D))"`

Lexer

Strings, comments, integers, floats, identifiers

Identifiers separated by spaces or comments

A set of *single-char identifiers*:

```
( ) { } , ;
```

Meant to be user-extensible

Expects UTF-8 but does not really care

No distinction between upper and lower case

Structured identifiers

Our OPG parser is sufficient to define a satisfactory ML-style syntax

Only sore point: things like `Str.concat`

```
Str.concat a b ?≡? _._ Str (concat a b)
```

Rather than allow some keywords to bind more tightly than the space

- Parse “identifiers” with a secondary precedence table

```
Str.concat ≡ __.__ Str concat
Str.concat a b ≡ (__.__ Str concat) a b
```

Related work

Mixfix in Agda (and others), Coq's `Notation`, ...

Honu and Star use OPG in a very similar way

- Some additional parsing done by the macros

Prolog for systematic use of an even more restrictive class of grammars

Template Haskell, for the interleaving of expansion and inference

Future work

Rewrite in Typer

Hygiene (we're not in '63 any more, right?)

Tolerable error reporting

Give access to the context and the expected type of macro calls

Something like `syntax-parse`