Typer: An infix statically typed Lisp

Pierre Delaunay
delaunap@iro.umontreal.ca
Université de Montréal
Département d'Informatique et Recherche Opérationnelle
Montréal, QC, Canada

Vincent Archambault-Bouffard
archambv@iro.umontreal.ca
Université de Montréal
Département d'Informatique et Recherche Opérationnelle
Montréal, QC, Canada

Stefan Monnier
monnier@iro.umontreal.ca
Université de Montréal
Département d'Informatique et Recherche Opérationnelle
Montréal, QC, Canada

Abstract

We show how Lisp-style macros and extensible infix syntax are combined in the programming language Typer, which is a combination of Lisp, ML, and Coq. Its name is an homage to Scheme(r) with which it shares the goal of pushing as much functionality as possible outside of the core and into libraries. While it superficially looks more like Haskell and ML, with infix notation and static polymorphic typing, it tries to preserve the syntactic malleability of Lisp by relying on the traditional Lisp-style S-expressions and macros.

Its main tool to this end is the use of an infix notation for S-expressions, which still makes it possible to parse sub-expressions before knowing what role they will play.

CCS Concepts •**Software and its engineering →Functional languages; Extensible languages;** *Control structures; Syntax; Preprocessors;*

Keywords Macros, S-expressions, Pure Type Systems, ML, Lisp

ACM Reference format:
Pierre Delaunay, Vincent Archambault-Bouffard, and Stefan Monnier. 2017. Typer: An infix statically typed Lisp. In *Proceedings of ML Family Workshop, Oxford, UK, September 2017 (ML'2017),* 2 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

Synopsis We show how to extend Lisp's S-expression syntax with infix notations in order to design an ML-style language with Lisp-style macros.

1 Introduction

The defining feature of languages from the Lisp family has been their exclusive reliance on a very simple and regular syntax, basically composed of atomic elements like symbols, strings, numbers, and parenthesized subtrees. While this limitations is what makes Lisp unpalatable to some programmers, it is also the fundamental element that gives Lisp its power for metaprogramming.

There is clearly a desire to extend Lisp with a richer syntax, as evidenced by the various exceptions (which you might call extensions) to the base parenthesized-prefix syntax present in languages from the Lisp family, such as the use of the traditional "`" and "," prefix symbols for quasi-quoting, Common-Lisp's "*pkg*:*id*" qualified identifiers or "*var* `in` *exp*" in the loop macro, `syntax-parse`'s "*var*:*syntaxclass*" shorthand (Culpepper and Felleisen 2010), or TypedRacket's "[*var* : *type*]" notation in formal arguments (Tobin-Hochstadt and Felleisen 2008).

But attempts to add comparable macro systems to languages with a richer syntax have not had the same success in the sense that they apparently do not work well enough to be widely embraced by their users. We believe, to be really successful, a macro system needs to be simple and seamless. By "seamless" here we mean that the syntax of macro calls is just as flexible as that of any other construct of the language, so users of a macro can't really tell if it's implemented as a macro or if it's a primitive construct of the language.

Most macro systems outside of the Lisp-family fail this test. These systems additionally suffer from complexity: Some of that complexity comes from the need to expose how that richer syntax gets mapped to some data representation. But the main source is the presence of various syntactic categories (such as expressions, instructions, types annotations, or declarations). This can force the system to have various kinds of macros, and it introduces further difficulties because the parser needs to decide which syntactic category to use to parse each macro call's actual arguments.

When designing the language Typer, we wanted to have an ML-style language where, just like in Scheme, we can define as much of the language as possible in the form of libraries layered on top of a minimalist core.

This means we wanted to be able to define constructs such as "if e_1 then e_2 else e_3" in the language itself, and we wanted to be able to implement the compilation of pattern matching in a library rather than inside the compiler. This in turn requires the ability to extend the syntax of the language, including adding new infix or mixfix (Danielsson and Norell 2008) constructs.

In Typer we decided to start from a Lisp-style syntax and to generalize it to a more conventional infix notation, while still keeping as much as possible of Lisp's simple yet powerful macro system. We do this by keeping macros decoupled from syntax: we first define a *skeleton syntax tree* (Backrach and Playford 1999) which we call S-expressions (because they are very similar to Lisp's S-expressions) and then we define an infix syntax on top of it as mere syntactic sugar. Like Lisp's S-expressions, the syntax only accepts a single syntactic category. This means that, just as in Lisp, Typer's *reader* (the parser building the S-expression) does not need to know if it's parsing an instruction, or an expression, or a type annotation, because they are all parsed in the same way.

Typer's reader, while more complex than that of Lisp, is still very primitive, since it uses an *operator precedence grammar* (Floyd 1963). This is the sweet spot which gives us just enough power to handle a syntax that should feel familiar to ML and Haskell users, while still being able to parse the code without needing to know if it's in the middle of a type annotation, function declaration, or some as-yet unknown DSL expression.

This work is supported by Canada's National Science and Engineering Research Council grant #298311 / 2012.

ML'2017, Oxford, UK

DOI: 10.1145/nnnnnnn.nnnnnnn

The result is a language whose syntax superficially looks like ML, but is really more like that of Lisp, which lets us define a macro system which is just as seamless and almost as simple as that of Lisp.

Typer macros are defined in Typer, and their expansion is performed during the elaboration phase, which also infers types. This interleaving is similar to the one used in Template Haskell (Sheard and Peyton Jones 2002), and it makes the macro system a bit more complex than that of Lisp since we cannot separate macro expansion from type inference. On the other hand it makes it easy to support things such as lexically scoped macros. More importantly, it enables Typer macros to make use of type information.

2 Primer

Typer's front-end has two important elements: the *reader* which turns an input text into an S-expression tree, and the *elaborator* which turns an S-expression into a typed lambda calculus.

2.1 The Reader

Based on a rudimentary grammar represented as a precedence table, Typer's reader will treat the following expressions as equivalent:

```
          (a) + b  =  (_+_ a b)
        [a, b, c]  =  \[_\] (_\,_ a b c)
lambda (x : t) -> e  =  lambda_->_ (_:_ x t) e
lambda (a - b) -> e  =  lambda_->_ (_-_ a b) e
```

Where the \ characters are needed because `[_]` would be read as 3 separate tokens. Contrary to Lisp, parentheses have no significance other than to group elements. Notice the last line, which is nonsensical in Typer but is still a valid S-expression as far as the reader is concerned, since it applies the same parsing rules to the "lambda argument" as anywhere else, for the simple reason that it does not know yet what "`lambda_->_`" is bound to.

2.2 Examples

New data types are typically defined with:

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

which the grammar treats as equivalent to the prefix form:

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

And "`type_`" is actually a macro which expands the above to:

```
List = typecons (List (_:_ a Type))
                nil
                (cons a (List a)));
```

where `typecons` is a built-in *special form* which defines a new type constructor.

To define a new form like "if e_1 then e_2 else e_3", we need 2 steps. First, setup the grammar to give appropriate precedences to "`if`", "`then`", and "`else`"

```
define-operator if () 2;
define-operator then 2 1;
define-operator else 1 66;
```

then define "`if_then_else_`" as a macro:

```
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)));
```

2.3 The Elaborator

In the above example, "`macro`" is the predefined data constructor of type "`(List Sexp -> Sexp) -> Macro`" which constructs a macro. Macro calls are distinguished from function calls by the fact that the element in function position has type "`Macro`".

So, Typer's elaboration needs to know the type of sub-expressions in order to know how to perform macro expansion. More specifically, the elaborator takes an S-expression and does:

```
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 e1 args)
    | "Special-Form" => elab_special_form c e1 args
    | _ => elab_funcall c t1 e1 args;
```

3 Conclusion and future work

We have presented the syntactic structure of the programming language Typer. It demonstrates how to extend Lisp's S-expressions with an infix syntax without losing the power and flexibility of Lisp macros. To do so, we use an operator precedence grammar, a sweet spot that is flexible enough to provide a familiar infix syntax, yet restricted enough that S-expressions can still be parsed without needing to know anything about macros.

Typer's implementation is available online from http://gitlab.com/monnier/typer. It is currently still rather primitive. Our main focus is on developing its macro facilities to have convenient access to typing information.

Acknowledgments

We would like to thank Christopher League for his comments. The work is supported by the National Science and Engineering Research of Canada under grant No: 298311 / 2012.

References

Johnathan Backrach and Keith Playford. 1999. *D-Expressions: Lisp Power, Dylan Style.* Technical Report. AI-Lab, MIT. http://people.csail.mit.edu/jrb/Projects/dexprs.pdf

Ryan Culpepper and Matthias Felleisen. 2010. Fortifying Macros. In *International Conference on Functional Programming*. Baltimore, MD. http://www.ccs.neu.edu/scheme/pubs/icfp10-cf.pdf

Nils Anders Danielsson and Ulf Norell. 2008. Parsing Mixfix Operators. In *Implementation and Application of Functional Languages (Lecture Notes in Computer Science)*, Vol. 6836. 80–99. https://pdfs.semanticscholar.org/6598/4aee4eccd577f9dd35154b1d24b157050d23.pdf

Robert W. Floyd. 1963. Syntactic Analysis and Operator Precedence. *Journal of the ACM* 10, 3 (July 1963), 316–333. http://dl.acm.org/citation.cfm?doid=321172.321179

Tim Sheard and Simon Peyton Jones. 2002. Template metaprogramming for Haskell. In *Haskell Workshop*. ACM Press, Pittsburgh, Pennsylvania, 1–16. http://dl.acm.org/citation.cfm?id=636528

Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The Design and Implementation of Typed Scheme. In *Symposium on Principles of Programming Languages*. ACM Press, San Francisco, California, 395–406. http://dl.acm.org/citation.cfm?id=1328486