Chamau: an ML dialect with quotations, grammars, and extensible syntax

Daniel de Rauglaudre Michel Mauny

INRIA-Rocquencourt
Projet Cristal
F-78153 Le Chesnay Cedex
France
Email: {Daniel.de_Rauglaudre,Michel.Mauny}@inria.fr
Web: http://pauillac.inria.fr/~{ddr,mauny}

April 10, 1996

Abstract

We present Chamau, a dialect of ML, enriched with support for concrete syntax manipulations. Chamau has the following original features: quotations, allowing the programmer to give a user-defined syntax to arbitrary values, and extensible grammars: a library to ease parser definitions. Moreover, Chamau's concrete syntax is itself extensible, which makes it possible to specialize the language to application domains.

Chamau is based on the Caml Special Light system [Ler95], and therefore features an extremely efficient compiler and an expressive higher order module system.

Introduction

In a programming language, syntactic facilities may allow for writing programs faster, for making them more readable, and therefore easier to maintain. Typical examples include C macros (preprocessed via an external process `cpp`, or `m4`) and Lisp/Scheme macros, processed by the compiler itself. However, these macros permit only simple patterns (typically, macro calls look like function calls). Another need for concrete syntax support is when the programmer wants to introduce his/her own syntactic construct in order to emphasize such or such programming style or dedicated library. Even more useful is the design of such dedicated library together with its syntactic construct to facilitate its usage.

Our system, Chamau, meets these requirements, since it includes the following original features:

- User-programmable quotations and antiquotations.
- A parser generator library, producing parsers from grammar specifications. Grammars are extensible. This library can be included in applications, whose interface language can easily be designed as an extensible language.
- Chamau is bootstrapped, and the Chamau parser itself is generated from an extensible grammar. Chamau's syntax is therefore extensible.

Chamau is based on the Caml Special Light (CSL) compiler and interactive system [Ler95], using only a new front-end, but reusing CSL's compiler and backend. CSL is one of the most efficient ML implementations at the time of this writing.

We describe in the following the quotation mechanism and the grammar library. We terminate with a few indications about the parsing technology that we use, and efficiency considerations.

1 Quotations

In a source program, quotations appear as character strings written inside `<<` and `>>`. A simple usage of quotations could be the same as simple `#define`'d C macros. For instance, the X-window events have integer values, but humans generally recognize them

through their names `keyPressEvent`, `exposeEvent`, etc. Therefore, a program[1] containing:

```
match event.kind with
[ <<keyPressEvent>> -> ...
| <<exposeEvent>> -> ...
| <<configureNotifyEvent>> -> ...
```

is obviously easier to write and to understand than the same one with integer constants replacing event names.

A more sophisticated example is a concrete syntax for λ-calculus, where abstractions are written `[x]M`, instead of λ`x.M`. The type of such values (abstract syntax trees of λ-terms) can be defined in Chamau as:

```
type term =
[ Abs of term
| Ref of int
| App of term and term ] ;
```

Typical values of that type are:

```
Abs (Ref 0)
Abs (Abs (Ref 1))
Abs (App (Ref 0) (Ref 0))
App (Abs (App (Ref 0) (Ref 0)))
    (Abs (App (Ref 0) (Ref 0)))
```

However, λ-calculists would prefer to write them as:

```
<< [x]x >>
<< [x,y]x >>
<< [x](x x) >>
<< ([x](x x) [x](x x)) >>
```

Moreover, one can build the last term by `App`'ing the third one to itself, i.e.:

```
let delta = Abs (App (Ref 0) (Ref 0))
in App delta delta;
```

Using the quotation together with antiquotations (in the second declaration, identifiers following "^" characters):

```
let delta = << [x](x x) >>
in << (^delta ^delta) >>;
```

to obtain the same results as above.

Quotations can be named, allowing the usage of different kinds of quotations in the same source program. For instance, one can write:

```
let delta = <:lambda< [x](x x) >>
in <:lambda< (^delta ^delta) >>;
```

Quotations enables the programmer to give a concrete syntax to arbitrary values (typically, complex data structures). This facilitates both program writing and reading.

Briefly, in order to define one's own quotation, one needs:

- to program a *quotation expander*: a string matching function, or a string parser, producing new strings being the expanded form of the quotation argument;
- to load the code of that expander, or to enrich the compiler's environment (for batch compilation), before compiling source programs. The enriched compiler will call the user's expander when encountering quotations.

We do not describe here the implementation of quotations. More information can be found in [MdR94][2].

2 Grammars and extensibility

Our grammars are precedence grammars for distfix operators, with precedences and associativity annotations. They are inspired from [Aas92], although we adopted a more classical notion of precedence. As an example, a grammar for arithmetic expressions can be created and enriched as:

```
value gram = Grammar.new Lexer.make;
value expr = Grammar.Entry.new gram;
<:extend< expr:
  [ LeftA
    [ e1=expr; "+"; e2=expr << e1 + e2 >>
    | e1=expr; "-"; e2=expr << e1 - e2 >> ]
  | LeftA
    [ e1=expr; "*"; e2=expr << e1 * e2 >>
    | e1=expr; "/"; e2=expr << e1 / e2 >> ]
  | [ e=INT << int_of_string e >>
    | "("; e=expr; ")" << e >> ] ]
>>;
```

The order in which productions occur specifies their relative precedences. The right hand side of each production (inside <<...>>) is the returned value for that

[1] The Chamau "match ... with" construct is similar, although more general, to Pascal's "case" and to C's "switch".

[2] In [MdR94], expanders had to return Chamau abstract syntax trees. Currently, expanders produce strings, asking less knowledge from the programmer.

production, using identifiers bound in the left hand side.

One can now create the associated parser and use it:

```
value calc str =
    Grammar.Entry.parse expr
            (Stream.of_string str);
calc "3+5*2";
- : int = 13
calc "1 + 2 + 3 - 4";
 - : int = 2
```

The `expr` grammar may be extended further with, for instance:

```
<:extend< expr: Like "*"
   [[ e1=expr; "%"; e2=expr
                    << e1 mod e2 >> ]]
>>;
```

We introduced a `%` operator, with the same precedence and associativity as multiplication (this is the meaning of `Like`). We would need then to re-create the parser, for that new production to be taken into account.

In the same way, all productions of a given grammar can be erased.

The `Grammar` module is a library that contains functions enabling the creation and extension of grammars, together with the parser generator. The `Lexer` module is a library that contains a generic lexical analyser. When extending a grammar, newly introduced keywords are taken into account by the lexer associated to the grammar. The `extend` quotation acts as an interface to grammar extension. These libraries are autonomous and can therefore be used in standalone applications.

Chamau's syntax is itself defined as a grammar, and is therefore extensible. For example, we can extend the syntax of expressions with an infix keyword for functional composition:

```
<:extend< expr: After "apply"
   [[ f = expr; "o"; g = expr
        <:expr< fun [x -> ^f (^g x)] >> ]]
>>;
```

The `o` symbol binds tighter than application (`apply` is the name of the production for function application in Chamau's grammar), and "e_1 `o` e_2" gets inlined to the function which, when applied to some x, returns e_1 (e_2 x).

Of course, extending the syntax of an existing language requires a good knowledge of its syntax, but we believe that our grammars are readable enough to act as a documentation. The extensibility of Chamau's syntax make it suitable for specialization to particular applications.

3 Parsing technology and efficiency considerations

The parsers generated from grammars are "mostly functional" recursive-descent parsers presented in [MdR92], with no backtrack.

Parser generation from the grammars presented above does not use sophisticated algorithms. The different phases of parser generation are:

1. collection of new keywords, and update of the lexer associated to the grammar;
2. representation of the grammar as a tree data structure;
3. left-factoring of each precedence level;
4. the data structure representing the grammar is then passed as argument to a generic parser.

It should be noted that grammars and parser generation take advantage of the ML type system: their type is inferred as it is for any other expression, and generated parsers are therefore guaranteed to be type-correct.

Although we did not measure precisely the speed of the generated parsers, the Chamau parser (generated from a grammar) parses (including I/O and construction of abstract syntax trees) approximatively 10K lines per minute on a 33MHz SUN SparcStation 5, when compiled by a non-optimized compiler (that is using bytecode).

Conclusion

Mostly functional parsers, quotations and extensible grammars are integrated as syntactic tools into the Chamau system. Bootstrapping Chamau enables Chamau's syntax extensibility. The whole system has proved being efficient.

A beta version of Chamau is freely available at the following address:

Host	`ftp.inria.fr`
Directory	`INRIA/Projects/cristal/chamau`

References

[Aas92] Annika Aasa. *User Defined Syntax.* PhD thesis, Chalmers University of Technology, 1992.

[Ler95] Xavier Leroy. *The Caml Special Light system, release 1.07 – Documentation and user's manual.* INRIA, September 1995. Documentation distributed with the Caml Special Light system.

[MdR92] Michel Mauny and Daniel de Rauglaudre. Parsers in ML. In *Proceedings of the ACM International Conference on Lisp and Functional Programming*, San Francisco, USA, 1992.

[MdR94] Michel Mauny and Daniel de Rauglaudre. A complete and realistic implementation of quotations for ML. In *Record of the 1994 ACM-SIGPLAN Workshop on ML and its Applications*, June 1994.