

# Parsers in ML

Michel MAUNY

Daniel de RAUGLAUDRE

INRIA-Rocquencourt\*

## Abstract

We present the operational semantics of streams and stream matching as discussed in [12]. Streams are data structures such as lists, but with different primitive operations. Streams not only provide an interface to usual input/output channels, but may be used as a data structure *per se*, holding any kind of element. A special pattern matching construct is dedicated to streams and the actual matching process will be called *parsing*.

The primary parsing semantics that we propose here is predictive parsing, i.e. recursive descent semantics with a one token look-ahead: although this choice seems to restrict us to the recognition of LL(1) languages, we show by examples that full functionality and parameter passing allow us to write parsers for complex languages. The operational semantics of parsers is given by transforming parsers into regular functions.

We introduce a non-strict semantics of streams by translating stream expressions into more classical data structures; we also investigate different sharing mechanisms for some of the stream operations.

## 1 Introduction

Several functional languages provide ways to assign a concrete syntax to data types [18, 1] and to define parsing functions [18, 3].

In [12], the first author proposed an integration of a parser definition facility in functional languages. This proposal was based on pattern matching values of a **stream** data type. The present paper presents an operational semantics for these streams and their pattern-matching (parsing): recursive descent semantics for parsing, with one look-ahead token (predictive parsing). In

---

\* Authors address: INRIA, B.P. 105, F-78153 Le Chesnay Cedex. Email: {Michel.Mauny,Daniel.de.Rauglaudre}@inria.fr

other words, a parsing rule is chosen if the first terminal of the stream argument is accepted by the first component of the stream pattern. This simple semantics combined with the full functionality of ML makes it possible to write in a simple way very powerful parsers.

Generally, the technique used for compiling grammar specifications into parsing functions produces bottom-up parsers and is often limited to LALR(1) languages [2]. This technique has the advantage of allowing the writing of rather high-level specifications (context-free grammars) and of producing efficient parsers. However, the generated parsers are driven by parsing tables, which make debugging of parsers difficult (e.g. one cannot trace a specific non-terminal). Furthermore, these techniques provide no parametricity for parsers.

Depending on the host language (purely functional or admitting side-effects), parsing can have a destructive effect on the stream argument or not. The destructive semantics has been implemented in the Caml Light system [9], and it proved to be efficient.

The remainder of the paper is organized as follows. We first recall the syntax and typing rules of stream expressions and parsers. Section 4 gives an operational semantics for parsing and for stream expressions. Section 5 gives examples of parsers. The current implementation of streams and parsers is described in section 6, where performance is discussed. Finally, the integration of these features in a purely functional language is presented in section 7, and recursive descent parsing with a limited form of backtracking is outlined.

## 2 Syntax

We extend ML with a data type for stream values and functions defined by case on their stream argument. We follow the syntax of the Caml dialect of ML [5, 18]. The syntax of expressions is extended by the following constructs:

$$\begin{aligned} \text{Expr} ::= & \dots \\ & \parallel [< >] \\ & \parallel [< \text{Str. comp.} ; \dots ; \text{Str. comp.} >] \\ & \parallel \text{parser } \text{Str. pat.} \rightarrow \text{Expr} \\ & \dots \\ & \mid \text{Str. pat.} \rightarrow \text{Expr} \end{aligned}$$

where *Str. comp.* is the syntactic category of stream components:

```
type token = INT of int | ID of string | LAM | DOT | LPAR | RPAR | PLUS | MINUS | MULT | DIV
```

The lexical analyzer (omitted) has the type `char stream → token stream`.

```
type lambda = Id of string | Int of int
             | App of lambda * lambda | Abs of string * lambda
             | Plus of lambda * lambda | Minus of lambda * lambda
             | Mult of lambda * lambda | Div of lambda * lambda
```

The `expr` function of type `token stream → lambda` is the entry point:

```
1. let rec expr = parser [< 'LAM; 'ID s; 'DOT; expr e >] → Abs(s,e)
2.   | [< add e >] → e
3. and add = let rec rest e1 = parser
4.   [< 'PLUS; mult e2; (rest (Plus(e1,e2))) e >] → e
5.   | [< 'MINUS; mult e2; (rest (Minus(e1,e2))) e >] → e
6.   | [< >] → e1
7.   in parser [< mult e1; (rest e1) e2 >] → e2
8.
9. and mult = let rec rest e1 = parser
10.  [< 'MULT; appl e2; (rest (Mult(e1,e2))) e >] → e
11.  | [< 'DIV; mult e2; (rest (Div(e1,e2))) e >] → e
12.  | [< >] → e1
13.  in parser [< appl e1; (rest e1) e2 >] → e2
14.
15. and appl = let rec rest e1 = parser
16.  [< atom e2; (rest (App(e1,e2))) e >] → e
17.  | [< >] → e1
18.  in parser [< atom e1; (rest e1) e2 >] → e2
19.
20. and atom = parser [< 'INT n >] → Int n
21.   | [< 'ID s >] → Id s
22.   | [< 'LPAR; expr e; 'RPAR >] → e
```

Figure 1: A parser for  $\lambda$ -terms with arithmetic infix operators.

```
Str. comp. ::= ' Expr
            || Expr
```

The syntax of streams looks like that for lists. A stream component is either a “singleton” stream `'E`, where `E` is an expression, or a substream `E`, where `E` is an expression. For instance, if both `s` and `t` are streams of integers, then `[< '1; s; t; '10 >]` is a stream of integers containing in order the element `1`, the elements of `s`, the elements of `t`, and `10`. The infinite stream of all integers can be built in the following way:

```
(* ints: int stream *)
let ints =
  let rec ints_from n = [<'n; ints_from(n+1)>]
  in ints_from 0
```

Parsers use parse rules composed of a stream pattern (*Str. pat.*) and an expression. Stream pattern components are either “terminal patterns” or “non-terminal” calls on the implicit stream argument:

```
Str. pat. ::= [<>]
            || [< Str. pat. comp. ; ... ; Str. pat. comp. >]
```

```
Str. pat. comp. ::= Expr Pattern
                 || ' Pattern
```

Stream patterns components of the form `'Pattern` are to be matched against stream elements. The expression part of *Expr Pattern* denotes the call of a non-terminal to the actual stream argument, whose result will be matched against the pattern.

Stream patterns are intended to match *initial segments* of streams. Therefore, the empty stream pattern matches any stream, acting as a “wildcard”. For example, the following parser returns the first element of its integer stream argument or the integer 0 if the stream is empty:

```
(* next_int: int stream → int *)
let next_int = parser [< 'n >] → n
                  | [< >] → 0
```

A more complete example appears in figure 1: a parser for  $\lambda$ -terms extended with integers and infix arithmetic operators. We will use it as a running example.

$$\begin{array}{c}
(1) \quad \Gamma \vdash_{\text{Exp}} [< >] : \alpha \text{ stream} \\
(2) \quad \frac{\Gamma \vdash_{\text{SEC}} sc_i : \tau \text{ stream} \quad i = 1, \dots, n}{\Gamma \vdash_{\text{Exp}} [< sc_1; \dots; sc_n >] : \tau \text{ stream}} \\
(3) \quad \frac{\Gamma \vdash_{\text{Exp}} e : \tau}{\Gamma \vdash_{\text{SEC}} 'e : \tau \text{ stream}} \\
(4) \quad \frac{\Gamma \vdash_{\text{Exp}} e : \tau \text{ stream}}{\Gamma \vdash_{\text{SEC}} e : \tau \text{ stream}} \\
(5) \quad \Gamma \vdash_{\text{SPat}} [< >] : \alpha \text{ stream} \Rightarrow \emptyset \\
(6) \quad \frac{\Gamma + \Delta_1 + \dots + \Delta_{i-1} \vdash_{\text{SPC}} sc_i : \tau \text{ stream} \Rightarrow \Delta_i \quad i = 1, \dots, n \quad \text{Dom}(\Delta_i) \cap \text{Dom}(\Delta_j) = \emptyset \quad i \neq j}{\Gamma \vdash_{\text{SPat}} [< sc_1; \dots; sc_n >] : \tau \text{ stream} \Rightarrow \Delta_1 + \dots + \Delta_n} \\
(7) \quad \frac{\Gamma \vdash_{\text{Pat}} p : \tau \Rightarrow \Delta}{\Gamma \vdash_{\text{SPC}} 'p : \tau \text{ stream} \Rightarrow \Delta} \\
(8) \quad \frac{\Gamma \vdash_{\text{Exp}} e : \tau \text{ stream} \rightarrow \sigma \quad \Gamma \vdash_{\text{Pat}} p : \sigma \Rightarrow \Delta}{\Gamma \vdash_{\text{SPC}} e p : \tau \text{ stream} \Rightarrow \text{Clos}(\Delta, FV(\Gamma) \cup FV(\tau))} \\
(9) \quad \frac{\Gamma \vdash_{\text{SPat}} sp_i : \tau \text{ stream} \Rightarrow \Delta_i \quad \Gamma + \Delta_i \vdash_{\text{Exp}} e_i : \sigma \quad i = 1, \dots, n}{\Gamma \vdash_{\text{Exp}} (\text{parser } sp_1 \rightarrow e_1 \mid \dots \mid sp_n \rightarrow e_n) : \tau \text{ stream} \rightarrow \sigma}
\end{array}$$

Figure 2: Typing rules for streams and parsers.

### 3 Types

The inference rules are given in figure 2. The notation should be explained: typing environments  $(\Gamma, \Delta, \dots)$  are partial maps from identifiers to type schemes. We write  $\Gamma + \Delta$  for the typing environment associating  $\Delta(x)$  to the identifier  $x$  if  $\Delta(x)$  is defined,  $\Gamma(x)$  otherwise.

$FV(\tau)$  is the set of free type variables occurring in the type  $\tau$ . For type schemes,  $FV(\forall \alpha_1 \dots \alpha_n. \tau) = FV(\tau) \setminus \{\alpha_1, \dots, \alpha_n\}$ . The  $\text{Clos}$  function is in charge of type generalization. It is defined by  $\text{Clos}(\tau, V) = \forall \alpha_1 \dots \alpha_n. \tau$ , where  $\{\alpha_1, \dots, \alpha_n\}$  is  $FV(\tau) \setminus V$ .  $\text{Clos}$  and  $FV$  are extended to type environments:  $\text{Clos}(\Gamma, V)$  generalizes all type assumptions of  $\Gamma$  with respect to  $V$ , and  $FV(\Gamma)$  is the set of all free type variables occurring in  $\Gamma$ .

Rules 1 to 4 deal with the typing of stream expressions: rules 3 and 4 type stream components (defining the  $\vdash_{\text{SEC}}$  auxiliary predicate), and rules 1 and 2 type stream expressions (extending the typing relation  $\vdash_{\text{Exp}}$  for expressions). As expected, they force all elements of a  $\tau$  **stream** to be of type  $\tau$ , and all its substreams to be of type  $\tau$  **stream**.

Rules 5 to 8 assign types to stream patterns and produce the typing environments (written  $\Delta$ ) that will be used on the right-hand side of parsing rules.

Rules 7 and 8 define the typing relation  $\vdash_{\text{SPC}}$  (typing stream pattern components). Rule 7 deals with patterns matching stream elements (terminals), while rule 8 deal with non-terminal calls. Since non-terminal calls can produce polymorphic values, these values being matched against a pattern, we generalize all typing assumptions produced by the pattern, with respect to type variables occurring free in the current environment or in the type

of the stream argument.

Rules 5 and 6 give types to stream patterns, defining the  $\vdash_{\text{SPat}}$  typing relation. Rule 6 types stream pattern components in sequence, composing the typing environments produced by typing previous components. This allows for the usage of values produced by a stream pattern component in subsequent non-terminal calls in the same stream pattern. For example, in figure 1, line 4,  $\mathbf{e}_2$ , produced by non-terminal **mult** is used in the argument of the recursive call to **rest**.

This implies a left-to-right matching, since a value bound by a stream pattern component can be used in a non-terminal call occurring on its right, in the same stream pattern. However, stream patterns are still linear: a variable cannot have more than one binding occurrence in a stream pattern, hence the side-condition of rule 6<sup>1</sup>.

Finally, rule 9 gives a type to parsers, and is similar to the rule for typing pattern-matching functions.

### 4 Operational semantics

In this section, we give an operational semantics for streams and parsers. We use a translation from a functional language extended with stream expressions and parsers to a conventional functional language. We first describe the semantics of parsers, by introducing two primitive operations on streams. The precise behavior of these

<sup>1</sup>This point can be discussed: the operational semantics given below imply a left-to-right matching of streams, and two binding occurrences of the same variable in the same stream pattern are seen as two consecutive **let**'s on the same variable, the second hiding the first. Therefore, non-linear stream patterns could be accepted, as long as it is clear that variable binding occurs in sequence, from left-to-right.

```

TExpr [[]] : Expression → Expression
TExpr [parser  $c_1$  | ... |  $c_n$ ] = function strm → TRule [[]] (TRule [[]] (...(TRule [[]] (None)) ...))
TExpr [[< >]] = Stream(ref (Value Sempty))
TExpr [[<  $s$  >]] = Stream(ref (Delayed (function () → Sapp(Stream(ref (Value Empty)), TSC [[]] ))))
TExpr [ $e_1$   $e_2$ ] = (TExpr [[]]  $e_1$ ) (TExpr [[]]  $e_2$ )
TExpr [ $x$ ] =  $x$  where  $x$  is a variable name

```

And similarly for other ML constructs.

```

TRule [[]] _ : Parse case → Expression → Expression
TRule [[< '  $P$ ;  $s$  >] →  $E$ ]  $K$  = match (peek strm)
    with Some  $P$  → junk strm ; TRule [[<  $s$  >] →  $E$ ] (raise ParseFailed)
    | _ →  $K$ 
TRule [[<  $f(P)$ ;  $s$  >] →  $E$ ]  $K$  = match (TExpr [[]]  $f$ ) strm
    with Some  $P$  → TRule [[<  $s$  >] →  $E$ ] (raise ParseFailed)
    | None →  $K$ 
    | _ → raise MatchFailed
TRule [[< >] →  $E$ ]  $K$  = Some(TExpr [[]]  $E$ )

TSC [[]] : List of stream components → Expression
TSC [ $E$ ;  $s$  &] = Stream(ref (Delayed (function () → Scons (TExpr [[]]  $E$ , TSC [[]]  $s$  ))))
TSC [ $E$ ;  $s$  &] = Stream(ref (Delayed (function () → Sapp (TExpr [[]]  $E$ , TSC [[]]  $s$  ))))
TSC [[]] = Stream(ref (Value Sempty))

```

Figure 3: Translation of streams and stream matching.

primitives depends on the semantics of stream expressions which will be given in section 4.2.

#### 4.1 Stream pattern matching

We give the operational semantics of parsing by translating parsers into the core ML language, extended with stream expressions and special functions operating on streams. The primitives operating on streams are:

```

peek:  $\alpha$  stream →  $\alpha$  option
junk:  $\alpha$  stream → unit

```

The **option** type is defined as:

```

type  $\alpha$  option = Some of  $\alpha$  | None

```

The **peek** function returns either the first element of its stream argument, or the constant **None** if the argument is the empty stream. In the semantics presented in this section, the **junk** function has a destructive effect on its stream argument: the first component is removed, and the stream is physically updated by its tail. In the generated programs, **junk** is always called after a successful call to **peek**, and thus cannot be applied to the empty stream.

We assume defined the exception **MatchFailed**, provided by the host language and raised when pattern-matching fails. We introduce a new exception **ParseFailed**. The semantics of stream matching is given by the *TExpr* [[]] function, which takes as argument a parser body (i.e. a list of parse cases) and returns a function body with a single case. The **strm** variable

name is supposed to be new and occur nowhere else in the program. The *TExpr* [[]] transformation recursively translates other ML constructs, preserving their original structure. The translation is given in figure 3<sup>2</sup>. The *TRule* [[]]  $K$  transformation takes as arguments a parse case to be translated and a translated program text  $K$ , acting as a continuation.

The translation of a non-terminal call (second case of the definition of *TRule* [[]]  $K$ ) produces a **match** construct, which should generate polymorphism, extending the ML **let** construct<sup>3</sup>.

Type variables of (*TExpr* [[]]  $f$ ) **strm** that do not occur free in the current typing environment (which contains a binding for the stream argument **strm**) can therefore be generalized. This justifies typing rule 8, figure 2.

As an example, the translation of the definition of **expr** (figure 1, lines 1-2) has the following shape:

1. (**\* expr**: **token stream** → **lambda option** \*)
2. ...
3. **and** **expr** = **function** **strm** →
4. **match** **peek strm** **with**
5.     **Some** **LAM** → **junk strm**;
6.     (**match** **peek strm**

<sup>2</sup>The “**match ...with ...**” construct matches an expression against patterns: it can also be written “**case ...of ...**”; the “**raise Exc**” expression raises the exception **Exc**.

<sup>3</sup>Actually, the **let** construct should be considered as a derived form of **match**, inheriting its typing rules. This is true in the Caml implementation [18], but not in the SML Definition [14].

```

7.         with Some(ID s) → (match ...)
8.         | _ → raise ParseFailed)
9. | _ → (match add strm
10.      with Some e → Some e
11.      | None → None
12.      | _ → raise MatchFailed)

```

We can see on this example that the choice of a parsing rule is done according to the result of matching the first element of the stream: the translated function first matches the current token against `LAM` (line 5), and, if the match fails or the stream is empty, calls `add` on the stream, and returns `None` if `add` returns `None` (line 11). Since the translation of `add` will have the same behavior, the choice of the parsing rule is done according to the shape of the first token.

Inner matches either succeed, or produce `ParseFailed`. `MatchFailed` is raised when the result of a successful non-terminal call does not match the specified pattern.

As we expect, left-recursion generally produces non-terminating parsers, since a recursive parser produces a recursive function, and a left-recursive parser may call itself without consuming any input from its stream argument.

The translation given in figure 3 uses the `option` type for the sake of readability. As shown by the example above, this has the effect of producing a function of type  $\alpha \text{ stream} \rightarrow \beta \text{ option}$  for a parser of type  $\alpha \text{ stream} \rightarrow \beta$ . The types of parsers could be preserved by using an exception `NotFound`, raised instead of producing `None`. This is the mechanism used by the current implementation (cf. section 6).

## 4.2 Stream expressions

Streams are preferably submitted to lazy evaluation. One reason is efficiency: a big stream can be constructed while a small part of it is used (in case of parsing error, for example). Furthermore, it is natural to build infinite streams and, in this case, lazy evaluation prevents a program from looping on that stream if the entire stream is not needed. Laziness is even more necessary if we want to model the normal ML input/output features as streams and stream operations: in the case of interactive input, building the input stream must not require a character to be typed on the keyboard before this character is actually needed by a parser.

Moreover, since we chose the destructive semantics for stream matching, streams are “consumed” by parsers, and they are thus similar to references in ML. The destructive semantics of stream matching interacts well with polymorphism: the well-known problem of “polymorphic references” [6, 16, 10] does not occur as long as streams cannot be extended in-place: the only destructive operation allowed on a stream is removing its first element.

We have two possible behaviors for stream concatenation. When substreams appear in streams, we can:

- either share them (i.e. parsing the stream will affect its substreams, and if several occurrences of the same substream appear, that substream, as well as all its other occurrences, will be emptied when the first occurrence will be completely parsed);
- or copy them (i.e parsing the stream will not affect substreams).

We present below two implementations of stream expressions following each of the semantics given above. The data type of streams is defined as:

```

type a stream = Stream of a str delay ref
and a delay = Delayed of (unit → a)
              | Value of a
and a str = Empty
           | Scons of a * a stream
           | Sapp of a stream * a stream

```

The type `delay`, combined with the `ref` type, is used to encode lazy evaluation in a strict language. This encoding is not satisfactory, since the creation of references on polymorphic values will prevent some translated programs from being typable. The availability of lazy data structures in a strict language [13] avoids this problem. However, we still need references to simulate destructive semantics: we thus cannot avoid typing problems for the translated programs.

The basic functions for direct manipulation of values of type `stream` are the `peek` and `junk` functions given below. First of all, we need a function `force` evaluating delayed values:

```

(* force: a delay → a *)
let force s = match !s
              with Delayed f →
                 let v = f() in s:=Value v; v
                 | Value v → v

```

We now come to the translation of stream expressions. We extend the function `TExpr` `[[ ]]` to stream expressions (cf. figure 3). The `TExpr` `[[ ]]` function delays the evaluation of the first element of a stream by appending the empty stream in front of the stream argument. The `TSC` `[[ ]]` auxiliary function translates stream components.

### 4.2.1 Substream sharing

The first semantics presented above can be implemented by the following `peek` and `junk` functions. The `peek` function is defined by:

```

(* peek: a stream → a option *)
let rec peek = function Stream s as str →
  match force s
  with Empty → None
   | Scons(x,_) → Some x
   | Sapp(s1, Stream s2) →
     (match peek s1

```

```
with None → s:=!s2; peek str
  | x → x)
```

and the junk function by:

```
(* junk: α stream → unit *)
let rec junk = function Stream s →
  match force s
  with Scons(_, Stream s') → s:=!s';()
  | Sapp(s',_) → junk s'
  | _ → raise (Failure "Junk: Bug")
```

The `junk` function can only be called after `peek` has returned a value different from `None`, thus cannot receive the empty stream as argument.

#### 4.2.2 Substream copying

Copying substreams can be implemented by making copies of (suspensions of) substreams encountered by the `peek` function. Since we have references to represent suspensions, it is then sufficient to copy those references. Moreover, we can take advantage of the fact that streams can be modified in-place to reorganize the stream by balancing it to the right, in order to optimize further accesses to its head. We omit the precise definition of `peek`.

This balancing guarantees that `junk` is always applied on a `Scons` stream:

```
(* junk: α stream → unit *)
let junk = function Stream s → match force s
  with Scons(_, Stream s') → s:=!s';()
  | _ → raise (Failure "Junk: Bug")
```

Copying substreams implies a loss of substream sharing: they are not emptied when their copies are parsed, but they do not benefit from the computations performed on their copies. This mechanism is thus closer to “call-by-name” than to lazy evaluation.

## 5 Examples

We give a few examples of usage of streams and parsers. All examples run in our implementation.

### 5.1 Parameterized parsers

The next examples show parsers parameterized by data structures or functions. The following parser is a partial rewrite of our running example (figure 1), that produces  $\lambda$ -terms with de Bruijn indices instead of variable names. The sum type `lambda` must be changed to include `Id of int`, encoding variable names by integers. An extra argument `rho`, a list of variable names, has been added to each parser. When parsing an abstraction, its body gets parsed with the abstracted variable added to `rho`, and parsing an identifier returns its index in `rho`.

```
(* expr: string list
  → token stream → lambda *)
let rec expr rho = parser
  [< 'LAM; 'ID s; 'DOT; (expr(s::rho)) e >]
```

```
→ Abs(e)
```

```
| [< (add env) e >] → e
and ...
and atom rho = parser
  [< 'INT n >] → Int n
  | [< 'ID s >] → Id (index s rho)
  | [< 'LPAR; (expr rho) e; 'RPAR >] → e
```

We can also parameterize a parser by another parser. It is common, when writing the grammar of a programming language, to write several times the same parsing scheme used with different non-terminals. In these cases, parameterized parsers are of great utility. The following parser recognizes arbitrary sequences of valid inputs to its parameter `elem`.

```
(* star: (α stream → β) → α stream → β list *)
let rec star elem = parser
  [< elem e; (star elem) l >] → e::l
  | [< >] → []
```

The `star` parser returns the list of results returned by its parameter `elem`. An example of specialization of the `star` parser is:

```
(* int_star: token stream → int list *)
let int_star = star (parser [< 'INT x >] → x)
```

### 5.2 Precedence and associativity

The problem of precedence and associativity of operators in top-down parsing receives simple solutions with functional parsers. For arithmetic operators, precedence must be taken into account. In figure 1, the relative precedences of multiplication and addition are specified by defining `add` in terms of `mult`, imposing a precedence of multiplicative operators over additive operators. Operators having the same precedence appear in consecutive parsing rules of the same parser. In figure 1, we used twice the same parsing scheme to implement left-associative arithmetic expressions (lines 3–13): the local functions `rest` accumulate results of parsing in their first argument.

Higher-order parsers factorize this work: `left_assoc` is a general parser for expressions separated by left associative operators. Parameter `op` is a parser for the operators; it returns a curried function building abstract syntax trees. The `term` parameter is a parser for expressions:

```
1. (* left_assoc: (α stream → β → β → β)
  2.   → (α stream → β)
  3.   → α stream → β *)
4. let left_assoc op term =
5.   let rec rest t1 = parser
6.     [<op f; term t2; (rest(f t1 t2)) t>] → t
7.     | [< >] → t1
8.   in parser [< term t; (rest t) r >] → r
The parsers for arithmetic operators can be written as:
  let op_add = parser
    [< 'PLUS >] → (function x →
```

```

      function y → Plus(x,y))
| [< 'MINUS >] → (function x →
      function y → Minus(x,y))
and op_mult = parser
  [< 'MULT >] → (function x →
      function y → Mult(x,y))
| [< 'DIV >] → (function x →
      function y → Div(x,y))

```

We can now rewrite `add` and `mult` in the following way:

```

let rec expr = ...
and add = left_assoc op_add mult
and mult = left_assoc op_mult appl
...

```

Right associativity and non-associativity can be treated by changing the first parse case of `rest` (line 6 of the definition of `left_assoc`) into:

```

[<op f; term t2; (rest t2) t >] → f t1 t

```

and:

```

[<op f; term t2 >] → f t1 t2

```

respectively.

Notice that the `appl` parser cannot be expressed using `left_assoc` since there is no operator for applications. A parser `op_appl` for application operators does not consume any input and always succeeds. Because of the predictive parsing technique, transforming `appl` using `left_assoc` and `op_appl` produces a parser that always fails.

### 5.3 Parsing a non context-free language

A typical example [2] of non context-free language is:

$$\{wcv \mid w \in (a|b)^*\}$$

where  $a$ ,  $b$  and  $c$  are terminals. In order to write a parser for that language, we need two auxiliary parsers: `wd` (for “word definition”) parses its input until a `c` character is encountered. It returns a list of parsers, each one dedicated to the recognition of a specific character of the word  $w$ . We write ‘`a`’, ‘`b`’ character constants of type `char`.

```

(* wd: char stream
   → (char stream → string) list *)
let rec wd = parser
  [<'a'; wd l>] → (parser [<'a'>] → "a")::l
| [<'b'; wd l>] → (parser [<'b'>] → "b")::l
| [<>] → []

```

The second parser that we need is `wu` (for “word usage”): a function taking as first argument a list of parsers, and returning a parser that applies each parser of the list, in sequence. The result is thus the sequencing of a list of parsers, seen as non-terminals.

```

(* wu: (α stream → string) list
   → α stream → string *)
let rec wu = function
  p::pl → (parser [<p x; (wu pl) l >] → x^l)
| [] → (parser [<>] → "")

```

A parser `wcv` for the language given above is:

```

(* wcv: char stream → string *)
let wcv = parser [<wd pl; 'c'; (wu pl) l>] → l

```

And the expression:

```

wcv (stream_of_string "abaacabaa")

```

evaluates to “`abaa`”.

## 6 Implementation

The second author realized two different implementations of streams and parsers in Caml Light [9]. Both implementations follow the same semantics: parsing has a destructive effect on streams, and stream concatenation share substreams (i.e. do not use copies of substreams).

Both implementations follow closely the operational semantics given here, although the translation of parsers into functions is more sophisticated. The types of parsers do not use the `option` type: instead, the `NotFound` exception is used.

### 6.1 Source to source translation

The first implementation consists of a source to source translation, following the scheme presented above, but with optimizations. For instance, consecutive parsing rules beginning by a terminal are gathered in a single `match` construct. This optimization is crucial when defining lexical analyzers. Typechecking parsers and stream expressions is achieved by typing the produced translation.

### 6.2 Integration to the Caml Light system

The second implementation is an integration of streams and parsers in the Caml Light system. Parsers and stream expressions are typechecked as primitive constructs, and intermediate code is directly generated. No extension of the Caml Light execution model was necessary.

Here also, consecutive parsing rules beginning with a terminal pattern, are gathered in a single match instead of a “cascade” of matches. Furthermore, compile-time  $\beta$ -reduction of anonymous parsers used as non-terminals is also performed. Finally, when the last rule of a parser is of the form  $[< E P >] \rightarrow P$ , where  $P$  is an irrefutable pattern, it is simplified into  $[< >] \rightarrow E$ . If this is the only rule of a parser, the parser is simply replaced by  $E$ . Unfortunately, we cannot optimize tail calls in the general case (i.e. when terminals or non-terminals appear at the left of the call), since in these cases, a possible `None` produced by  $E$  must be changed into `raise Parsefailed`.

### 6.3 Bootstrap of a realistic language

The complete lexical analyzer and parser of Caml Light have been written using streams and stream pattern matching. The new analyzers have been integrated in the Caml Light compiler, and a complete bootstrap of

---


$$TE\text{Expr} \llbracket \text{parser } c_1 \mid \dots \mid c_n \rrbracket = \text{function str} \rightarrow TR\text{ule} \llbracket c_1 \rrbracket (TR\text{ule} \llbracket c_2 \rrbracket (\dots (TR\text{ule} \llbracket c_n \rrbracket (\text{str}, \text{None})) \dots))$$

$$TR\text{ule} \llbracket \langle 'P; spcs \rangle \rightarrow E \rrbracket K = \text{match } (\text{peek str})$$

$$\quad \text{with } \text{Some } P \rightarrow \text{let str} = \text{junk str} \text{ in}$$

$$\quad \quad \quad TR\text{ule} \llbracket \langle spcs \rangle \rightarrow E \rrbracket (\text{raise ParseFailed})$$

$$\quad \quad \quad \mid \_ \rightarrow K$$
  

$$TR\text{ule} \llbracket \langle f(P); spcs \rangle \rightarrow E \rrbracket K = \text{match } (TE\text{Expr} \llbracket f \rrbracket) \text{ str}$$

$$\quad \text{with } (\text{str}, \text{Some } P) \rightarrow TR\text{ule} \llbracket \langle spcs \rangle \rightarrow E \rrbracket (\text{raise ParseFailed})$$

$$\quad \quad \quad \mid (\text{str}, \text{None}) \rightarrow K$$

$$\quad \quad \quad \mid \_ \rightarrow \text{raise MatchFailed}$$
  

$$TR\text{ule} \llbracket \langle \rangle \rightarrow E \rrbracket K = (\text{str}, \text{Some}(TE\text{Expr} \llbracket E \rrbracket))$$


---

Figure 4: Translation for purely functional parsers.

the Caml Light implementation has been done. The result is an implementation of Caml Light including stream expressions and parsers, and using them to define its own syntax. In this implementation, the input channel is itself a stream of characters. This bootstrap shows that writing realistic lexical analyzers and parsers using streams and stream pattern matching is feasible.

The original Caml Light implementation uses a lexical analyzer generator written in Caml Light by X. Leroy following the lines of Lex [11]. The main difference between this generator and Lex is that instead of interpreting tables, the automaton is represented by a set of mutually recursive functions. The original Caml Light parser is produced by Berkeley Yacc<sup>4</sup> [8]. It uses Berkeley Yacc parsing tables in a Caml format, and the parsing engine is the Caml Light translation of the Berkeley Yacc parsing engine. Timing the bootstrapping process showed that the new bootstrap is only 4% slower than the original bootstrap (2mn 15s on a Sun SparcStation 2). We conclude that this relatively simple implementation of streams and stream matching is already competitive with more classical techniques.

#### 6.4 Performances

We give some preliminary results about performances. Two simple programs have been written for that purpose.

	wc	parsing	
		test 1	test 2
Lex-C (cc)	2.27s		
Yacc-Lex-C (cc)		0.64s	1.03s
YL-Caml-Light	10.88s	8.12s	12.87s
PP-Caml-Light	7.66s	5.89s	9.74s

The first one is a simple simulation of the Unix `wc` command (“word count”), written as a lexical analyzer. The command has been run on a Sony 3410 workstation.

---

<sup>4</sup>Bob Corbett’s reimplementaion of Yacc, from the Unix BSD 4.4 distribution.

- **Lex-C** is a Lex generated C program;
- **YL-Caml-Light** is written using the generators of the original Caml Light system;
- **PP-Caml-Light** is implemented using our facilities.

The input was a text file of 128k characters. The timings show that reading and testing on character streams is 42% slower with streams and the `parser` facility than with the lexical analyzer generator, which is itself 3.4 times slower than `Lex-wc` compiled with `cc`. Since Caml Light is a byte-coded implementation, this result is good.

The second example is a lexical analyzer composed with the parser given in figure 1. The parser does not produce any output: it only recognizes or rejects a phrase. The produced commands have been called on two inputs: “test 1” is a sequence of 960 expressions (25k characters), and “test 2” is a single expression (39k characters). The timings show that the “streams” version is about 25% faster than the command produced by Caml Yacc and Caml Lex, but still almost 10 times slower than the C versions.

These timings show that even with a simple compilation technique of streams and stream matching, we obtain reasonable performance. Moreover, the size of code and data for parsing functions is small: we save the size of the parsing stack and tables.

## 7 Purely functional parsers and streams

In order to integrate streams and parsers into purely functional languages, it is necessary to avoid destructive update of streams. The description above has to be slightly modified in order to fit the needs of purely functional languages. The “substream copying” semantics of stream expressions has, of course, to be adopted. But, in order to be able to recover the remaining input to a parser, we have to change the type of parsers: they must also return the remaining stream as part of their result. Moreover, purely functional languages do not have exception handling: the values returned by a parser must be of an

---

```
TExpr [[parser c1 | ... | cn] = function strm → TCase [[c1] (TCase [[c2] (...(TCase [[cn] (strm, None))...))
```

```
TCase [[< ' P; spcs >] → E] K = match match (peek strm)
  with Some P → let strm = junk strm in
    TRest [[< spcs >] → E]
  | _ → (strm, None)
  with (strm, Some x) → (strm, Some x)
  | (_, None) → K

TCase [[< f(P); spcs >] → E] K = match match (TExpr [[f]]) strm
  with (strm, Some P) → TRest [[< spcs >] → E]
  | (strm, None) → (strm, None)
  | _ → raise MatchFailed
  with (strm, Some x) → (strm, Some x)
  | (_, None) → K

TRest [[< ' P; spcs >] → E] = match (peek strm)
  with Some P → let strm = junk strm in
    TRest [[< spcs >] → E]
  | _ → (strm, None)

TRest [[< f(P); spcs >] → E] = match (TExpr [[f]]) strm
  with (strm, Some P) → TRest [[< spcs >] → E]
  | (strm, None) → (strm, None)
  | _ → raise MatchFailed

TRest [[< >] → E] = (strm, Some(TExpr [[E]]))
```

---

Figure 5: Translation for parsers with limited backtrack.

option type. Thus, the type of a parser will be:

```
α stream → (α stream * β option)
```

We assume that we have lazy data structures; the definition of streams can therefore be simplified into:

```
type α stream = Empty
  | Scons of α * α stream
  | Sapp of α stream * α stream
```

We use the following *peek* function:

```
(* peek: α stream → α option *)
let rec peek = function
  Empty → None
  | Scons(x, _) → Some x
  | Sapp(s1, s2) → (match peek s1
    with None → peek s2
    | x → x)
```

The *junk* function is now side-effect free and returns the tail of its argument stream:

```
(* junk: α stream → α stream *)
let junk = function
  Scons(_, s') → s'
  | Sapp(Empty, s2) → junk s2
  | Sapp(Scons(x, s1), s2) → Sapp(s1, s2)
  | Sapp(Sapp(s1, s2), t)
```

```
→ junk (Sapp(s1, (Sapp(s2, t)))
```

```
| Empty → raise (Failure "Junk: Bug")
```

The translation of streams becomes straightforward.

## 7.1 Predictive parsers

The translated parsers rebind the variable *str*m to the tail of the stream, returned by the call *junk* *str*m, hiding its previous binding. The new translation is given in figure 4.

## 7.2 Parsers with limited backtracking

From this purely functional implementation, it is fairly easy to design a version of these parsers using an arbitrarily large look-ahead set of tokens. This semantics implements a restricted form of backtracking: if a non-terminal at position *n* in a parsing rule fails, then the next rule of the calling parser is tried, instead of trying the next rule of the non-terminal at position *n*-1 as would be implied by a full backtracking semantics. This has the advantage of reducing the search space in a uniform way. The translation of parsers with limited backtrack is given in figure 5.

## 8 Related works

Writing parsers in functional languages has been explored by Burge [4] and Fairbairn [7], among others. Their work is based on so-called “parser-builders” [15], consisting mainly in programming the alternation of parsers, corresponding to the “|” separating parsing rules, and (possibly empty) sequencing of parsers, corresponding to the “;” separating stream patterns components. Burge [4] also introduces general higher-order parsers.

These parsers share a common semantics: recursive descent with full backtracking. In [15], it is advised to limit the search space by inserting error functions at appropriate places, e.g. when an **IF** token has been found, but no **ELSE** can be found, there is no need for trying other parsing rules, unless they also wait for an **IF** token. This eliminates obviously useless portions of the search space.

We have concentrated here on more classical semantics: recursive-descent with only one look-ahead token, which is the technique used in conventional programming languages when writing parsers “by hand”. This provides us with an efficient parser definition facility, well-suited to the ML destructive input-output model. Moreover, this semantics allows for writing parsers for interactive languages and is powerful enough to enable the complete bootstrapping of a realistic programming language. Furthermore, this simple technique guarantees that semantic actions will be executed only once during parsing: this is a useful property for mostly-functional programming languages such as ML.

## 9 Conclusion

We have presented the operational semantics for streams and stream matching in ML-like languages. Streams are lazy data structures, and we have considered different semantics for stream matching. We emphasized predictive parsing, which seems to represent a good trade-off between expressivity, efficiency, and ease of debugging.

The **parser** facility works well with purely functional languages, and allows for different semantics: predictive parsing or arbitrary large look-ahead set with limited backtracking semantics can also be designed for these parsers, providing us with useful alternatives to general backtracking techniques that appear in the literature [4, 7, 17, 15].

## 10 Acknowledgements

We thank Ian Jacobs and Xavier Leroy for their merciless reading of a draft of this paper.

## References

- [1] A. Aasa, K. Petersson, and D. Synek. Concrete syntax for data objects in functional languages. In *Proceedings of*

*the 1988 ACM Conference on Lisp and Functional Programming*, pages 96–105, 1988.

- [2] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.
- [3] A. Appel and D. MacQueen. A Standard ML compiler. In G. Kahn, editor, *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*, 1987.
- [4] W.H. Burge. *Recursive Programming Techniques*. Addison-Wesley, 1975.
- [5] G. Cousineau and G. Huet. The CAML primer. Technical Report 122, INRIA, 1990.
- [6] Luis Damas. *Type assignment in programming languages*. PhD thesis, University of Edinburgh, 1985.
- [7] J. Fairbairn. Making form follow function: An exercise in functional programming style. *Software, Practice and Experience*, 17(6):379–386, 1987.
- [8] S.C. Johnson. Yacc: Yet Another Compiler-Compiler. In *Unix Programmer’s Manual*, volume PS1. Usenix Association, 1986.
- [9] X. Leroy. The ZINC experiment: an economical implementation of the ML language. Technical Report 117, INRIA, 1990.
- [10] X. Leroy and P. Weis. Polymorphic type inference and assignment. In *Proc. Symp. Principles of Programming Languages*, pages 291–302. ACM press, 1991.
- [11] M.E. Lesk. Lex — a lexical analyser generator. Technical Report 39, AT&T Bell Laboratories, 1975.
- [12] M. Mauny. Parsers and printers as stream destructors and constructors embedded in functional languages. In *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*. Addison-Wesley, 1989.
- [13] M. Mauny. Integrating lazy evaluation in strict ML. Technical Report 137, INRIA, 1992.
- [14] Robin Milner, Mads Tofte, and Robert Harper. *The definition of Standard ML*. The MIT Press, 1990.
- [15] C. Reade. *Elements of Functional Programming*. International Computer Science Series. Addison-Wesley, 1989.
- [16] M. Tofte. Type inference for polymorphic references. *Information and Computation*, 1(89):1–34, 1990.
- [17] P. Wadler. How to replace failure with a list of successes. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 113–128. Springer-Verlag, 1985.
- [18] P. Weis, M.V. Aponte, A. Laville, M. Mauny, and A. Suárez. The CAML reference manual. Technical Report 121, INRIA, 1990.