MiniML Spec

Fazazi Zeid Luo Yukai Dibassi Brahima

28 avril, 2023

Table des matières

1 Syntaxe MiniML

1.1 Lexing Tokens

1.1.1 Separators

$$\{ \quad \} \qquad [\quad] \qquad (\quad) \qquad ; \qquad , \qquad * \qquad \rightarrow \qquad | \qquad =$$

1.1.2 Mots Clefs

$$let \qquad fun \qquad in \qquad match \qquad with \qquad type \qquad of \qquad rec \qquad if \qquad then \qquad else$$

1.1.3 Types

$$int \qquad bool \qquad unit$$

1.1.4 Operateurs

$$+ \qquad - \qquad \% \qquad / \qquad :: \qquad \&\& \qquad || \qquad * \qquad < \qquad > \qquad \leq \qquad \geq \qquad =$$

1.1.5 Valeurs Atomiques

$$integer = (\ -\ )\ ?\ [\ 0\ -\ 9\ ]$$
$$boolean = (\ true \mid false\ )$$

1.1.6 Identificateur

$$alphanum = [a-z \quad A-Z \quad 0-9 \quad \_]*$$
$$basic\_ident = [a-z \quad \_]\, alphanum$$
$$vartype = [`a]\ [0\,..\,9]*$$

$$constructeur\_ident = [A-Z]\, alphanum$$
$$constructeur\_infixes = [::\quad ,]$$

1.2 Grammaire

$$
\begin{array}{rll}
\textbf{Prog} := & | & \textbf{Def} \\
& | & \textbf{Expr} \\
& | & \textbf{Prog} \; ; ; \; \textbf{Prog}
\end{array}
$$

1.2.1 Definitions

$$
\begin{array}{rll}
\textbf{Def} := & | & \textit{let basic\_ident} = \textbf{Expr} \qquad \textit{Def Variable} \\
& | & \textit{let basic\_ident}_{\textbf{list}} = \textbf{Expr} \qquad \textit{Def Function} \\
& | & \textit{let rec basic\_ident}_{\textbf{list}} = \textbf{Expr} \qquad \textit{Def FunctionRec} \\
& | & \textit{type vartype}_{\textbf{list}} \; \textit{basic\_ident} = \textbf{NewContructor\_Case} \qquad \textit{Def Type}
\end{array}
$$

$$
\begin{array}{rll}
\textbf{NewContructor\_Case} := & | & \textit{constructeur\_ident} \\
& | & \textit{constructeur\_ident of } \textbf{Type} \\
& | & \textit{NewContructor\_Case} \; '|' \; \textit{NewContructor\_Case}
\end{array}
$$

1.2.2 Expressions

$$
\begin{array}{rll}
\textbf{Litteral} := & | & \textit{integer} \\
& | & \textit{boolean} \\
& | & ( ) \qquad \textit{Unit}
\end{array}
$$

$$
\begin{array}{rll}
\textbf{Expr} := & | & (\textbf{Expr}) \\
& | & \textbf{Litteral} \\
& | & \textit{basic\_ident} \\
& | & \textbf{UnaryOperator} \quad \textbf{Expr} \\
& | & \textbf{Expr} \quad \textbf{BinaryOperator} \quad \textbf{Expr} \\
& | & \textbf{Expr} \quad \textbf{Expr} \qquad \textit{Call} \\
& | & \textbf{Expr}; \textbf{Expr} \qquad \textit{Sequence} \\
& | & \textit{let basic\_ident} = \textbf{Expr} \; \textit{in} \; \textbf{Expr} \qquad \textit{Binding} \\
& | & \textit{fun basic\_ident}_{\textbf{list}} \rightarrow \textbf{Expr} \qquad \textit{Lambda} \\
& | & \textbf{Expr} \; \textit{constructeur\_infixes} \; \textbf{Expr} \\
& | & \textit{constructeur\_ident} \; \textbf{Expr} \qquad \textit{Construction} \\
& | & \textit{constructeur\_ident}
\end{array}
$$

	|	*let basic_ident*$_{\textbf{list}}$ = **Expr** *in* **Expr**	*Fonction*
	|	*let rec basic_ident basic_ident*$_{\textbf{list}}$ = **Expr** *in* **Expr**	*Fonction Recusive*
	|	*match* **Expr** *with* **Match_Case**	
	|	*if* **Expr** *then* **Expr** *else* **Expr**	
UnaryOperator	:= |	*not*	
BinaryOperator	:= |	&&	
	|	||	
	|	+	
	|	−	
	|	/	
	|	%	
	|	*	
	|	<	
	|	>	
	|	≤	
	|	≥	
	|	=	

1.2.3 Filtrage et Motifs

Match_Case	:= |	**Pattern** → **Expr**	
	|	**Pattern** → **Expr** ′|′ **Match_Case**	
Pattern	:= |	(**Pattern**)	
	|	**Litteral**	
	|	*basic_ident*	
	|	_	
	|	*constructeur_ident*	
	|	*constructeur_ident* **Pattern**	

1.2.4 Types

Type	:=	| (**Type**)	
		| *int*	
		| *bool*	
		| *unit*	
		| **Type** $*$ **Type**	*Tuple Type*
		| **Type** $\rightarrow$ **Type**	*Lambda Type*
		| *vartype*	*'a*
		| *basic_ident*	*Defined Type*
		| $\mathbf{Type}_{\mathbf{List}}$	*App Type*

2 Semantique de traduction

2.1 Programmes

$$\vdash_{Prog} Prog[cs] \rightarrow Prog'(x)$$

$$\begin{aligned} \text{si } &\vdash_{\text{Cmds}} cs \rightarrow (g, \omega, \upsilon) \\ \text{alors } &\vdash_{\text{Cmds}} Prog[cs] \rightarrow Prog'(\omega; Do(g, \upsilon)) \end{aligned}$$

2.2 Suites de commandes

$$\vdash_{\text{Cmds}} cs \rightarrow (\gamma, \omega, \upsilon)$$

$$\begin{aligned} \text{(VAR DEFS) si } & d \in DEF, \\ \text{et si } & \vdash_{\text{Cmds}} cs \rightarrow (\gamma, \omega, \upsilon) \\ \text{et si } & \vdash_{\text{Def}} d \rightarrow \pi \\ \text{et si } & \pi \in GLB \\ alors & \vdash_{\text{Cmds}} (Def(d); cs) \rightarrow ((\gamma; \pi), \omega, \upsilon) \end{aligned}$$

$$\begin{aligned} \text{(TYPE DEFS) si } & d \in DEF, \\ \text{et si } & \vdash_{\text{Cmds}} cs \rightarrow (\gamma, \omega, \upsilon) \\ \text{et si } & \vdash_{\text{Def}} d \rightarrow \pi \\ \text{et si } & \pi \in TYPE \\ alors & \vdash_{\text{Cmds}} (Def(d); cs) \rightarrow (\gamma, (\omega; \pi), \upsilon) \end{aligned}$$

$$\begin{aligned} \text{(GLB EXPR) si } & b \in EXPR, \\ \text{et si } & \vdash_{\text{Cmds}} cs \rightarrow (\gamma, \omega, \upsilon) \\ \text{et si } & \vdash_{\text{Expr}} b \rightarrow \upsilon' \\ alors & \vdash_{\text{Cmds}} (Expr(b), cs) \rightarrow (\gamma, \omega, \upsilon') \end{aligned}$$

2.3 Définitions

$$\vdash_{\text{Def}} d \to \pi$$

(VARDEF) si $\vdash_{\text{Expr}} e \to e'$
alors $\vdash_{\text{Def}} VariableDef(v, e) \to GLB(InsLet(v, e'))$

(TYPDEF) si $\vdash c1 \to c1' \ldots$ si $\vdash cN \to cN'$
alors $\vdash_{\text{Def}} TypeDef(n, [t1, \ldots, tn], [c1, \ldots, cN])$
$\to \text{TYPE}(Typ\_Def(n, [t1, \ldots, tn], Def\_Datatype[c1', \ldots, cN']))$

2.4 Types

$$\vdash_{\text{Type}} t \to t'$$

(TINT) $\vdash_{Type} TypeInt \to TypInt$
(TBOOL) $\vdash_{Type} TypeBool \to TypBool$
(TUNIT) $\vdash_{Type} TypeUnit \to TypUnit$
(TDEF) $\vdash_{Type} TypeDefined(id) \to TypVar(id)$
(TVAR) $\vdash_{Type} TypeVar(id) \to TypVar(id)$

(TTUPLE) si $\vdash_{Type} t1 \to t_1, \ldots$ et $\vdash_{Type} tN \to t_N$
alors $\vdash_{Type} TypeTuple([t1, \ldots, tN]) \to TypTuple[t_1, \ldots, t_N]$

(TCONS) si $\vdash_{Type} t \to t'$
si $\vdash_{Type} p1 \to p_1, \ldots$ et $\vdash_{Type} pN \to p_N$
alors $\vdash_{Type} TypeConstructor(t, [p1, \ldots, pN]) \to TypApp(t, [p_1, \ldots, p_N])$

(TLAMB) si $\vdash_{Type} a \to a'$
et si $\vdash_{Type} ret \to ret'$
alors $\vdash_{Type} TypeLambda(a, ret)$
$\to TypClosure(Exp,\ (TypFun(TypThunk(ret'), a')))$

2.5 Litteraux et Expressions

$$\vdash_{\text{Expr}} e \to e'$$

(INT) si $i \in \text{NUM}$
alors $\vdash_{Expr} Integer(i) \to ExprInt(i)$
(TRUE) $\vdash_{Expr} Boolean(true) \to ExprConstructor(True, [\,])$
(FALSE) $\vdash_{Expr} Boolean(false) \to ExprConstructor(False, [\,])$
(UNIT) $\vdash_{Expr} Unit \to ExprConstructor(Unit, [\,])$

(TUPLE) si $\vdash_{Expr} e1 \to e_1, \ldots$ si $\vdash_{Expr} eN \to e_N$
alors $\vdash_{Expr} Tuple([e1, \ldots, eN]) \to ExprConstructor(Tuple, [e_1, \ldots, e_N])$

(UNARY1) si $\vdash_{Expr} a \to a'$
alors $\vdash_{Expr} CallUnary(op, [a]) \to ExprMonPrim(op, a')$
(UNARY0) si $\vdash_{Expr} Lambda(a, CallUnary(op, [a])) \to \omega$
alors $\vdash_{Expr} CallUnary(op, [\,]) \to \omega$

(BINARY2) si $\vdash_{Expr} a1 \to a_1$ et $\vdash_{Expr} a2 \to a_2$
alors $\vdash_{Expr} CallBinary(op, [a1 : a2]) \to ExprBinPrim(op, a_1, a_2)$
(BINARY1) si $\vdash_{Expr} Lambda(b, CallUnary(op, [a, b])) \to \omega$
alors $\vdash_{Expr} CallBinary(op, [a]) \to \omega$
(BINARY0) si $\vdash_{Expr} Lambda(a, Lambda(b, CallUnary(op, [a, b]))) \to \omega$
alors $\vdash_{Expr} CallBinary(op, [\,]) \to \omega$

(CONSTR) si $\vdash_{Expr} e \to e'$
alors $\vdash_{Expr} Construct(c, e) \to ExprConstructor(ConsNamed(c), e')$

(BIND) si $\vdash_{Expr} i \to i'$
et si $\vdash_{Expr} c \to c'$
alors $\vdash_{Expr} Binding(v, i, c) \to ExprBlock(Blk([InsLet(v, i')], c'))$

(MATCH) si $\vdash_{Expr} m \to m'$
si $m1 \in CASE \ldots$ et $mN \in CASE$
si $\vdash_{Case} m1 \to m_1, \ldots$ et $\vdash_{Case} mN \to m_N$
alors $\vdash_{Expr} Match(m, [m1, \ldots, mN]) \to ExprMatch(m', [m_1, \ldots, m_N])$

$$\text{(SEQ) si } \vdash_{Expr} e1 \to InsLet(x1, e_1), \ldots \text{ et } \vdash_{Expr} eN-1 \to InsLet(xN-1, e_{N-1})$$
$$\text{et si } \vdash_{Expr} eN \to e_N$$
$$\text{alors} \vdash_{Expr} Sequence([e1, \ldots, eN])$$
$$\to ExprBlock(Blk([InsLet(x1, e_1); \ldots; InsLet(xN-1, e_{N-1})], e_N))$$

$$\text{(CALL) si } \vdash_{Expr} a \to a'$$
$$\text{et si } \vdash_{Expr} f \to f'$$
$$\text{alors} \vdash_{Expr} Call(f, a)$$
$$\to ExprBlock(Blk([InsOpen(Exp, f'), InsForce(ExprMethod(Call, [a']))]))$$

$$\text{(LAMBDA) si } \vdash_{Expr} a \to a'$$
$$\text{et si } \vdash_{Expr} b \to b'$$
$$\text{alors} \vdash_{Expr} Lambda(a, b)$$
$$\to ExprClosure(Exp, ExprGet([GetPatTag(Call, [a'], ExprThunk(b'))]))$$

$$\text{(REC) si } \vdash_{Expr} a \to a'$$
$$\text{si } \vdash_{Expr} v \to v' \text{ et si } \vdash_{Expr} b \to b'$$
$$\text{alors} \vdash_{Expr} FunctionRec(v, a, b)$$
$$\to ExprClosure(Exp, ExprRec(v', ExprGet([GetPatTag(Call, [a'], ExprThunk(b'))])))$$

2.6 Motifs et Filtrage

$$\vdash_{\text{Case}} Case(p, e) \to \alpha$$

$$\text{(INTPAT) si } l \in \text{NUM et } \vdash_{Expr} e \to e'$$
$$\text{alors} \vdash_{Case} Case(LitteralPattern(l), e) \to MatchPatTag(IntLitt\ l, [\,], e')$$
$$\text{(BOOLPAT) si } l = Boolean(\_),$$
$$\text{si } \vdash l \to l' \text{ et } \vdash_{Expr} e \to e'$$
$$\text{alors} \vdash_{Case} Case(LitteralPattern(l), e) \to MatchPatTag(l', [\,], e')$$
$$\text{(UNITPAT) si } l = Unit \text{ et } \vdash_{Expr} e \to e'$$
$$\text{alors} \vdash_{Case} Case(LitteralPattern(l), e) \to MatchPatTag(Unit, [\,], e')$$

(TUPAT) si $p1 \in CASE, \ldots$ si $pN \in CASE$
si $\vdash p1 \to p_1, \ldots,$ si $\vdash pN \to p_N$
et $\vdash_{Expr} e \to e'$
alors $\vdash_{Case} Case(TuplePattern([p1, \ldots, pN]), e) \to MatchPatTag(Tuple, [p_1, \ldots, p_N], e')$

(CONSPAT) si $c \in CASE$
si $\vdash c \to c'$
et $\vdash_{Expr} e \to e'$
alors $\vdash_{Case} Case(ConstructorPattern((n, c)), e) \to MatchPatTag(ConsNamed(n), c', e')$

(VARPAT) si $\vdash_{Expr} e \to e'$
alors $\vdash_{Case} Case(VarPattern(x), e, l) \to MatchPatVar((x, l), e', l)$

(WILDPAT) si $\vdash_{Expr} e \to e'$
alors $\vdash_{Case} Case(WildcardPattern(), e, l) \to MatchPatVar((n, l), e', l)$