

MiniML Spec

Fazazi Zeid

Luo Yukai

Dibassi Brahima

28 avril, 2023

Table des matières

1	Syntaxe MiniML	3
1.1	Jetons de lexing	3
1.1.1	Séparateurs	3
1.1.2	Mots Clefs	3
1.1.3	Types	3
1.1.4	Opérateurs	3
1.1.5	Valeurs Atomiques	3
1.1.6	Identificateur	3
1.2	Grammaire	4
1.2.1	Outils de lecture	4
1.2.2	Programmes	4
1.2.3	Définitions	4
1.2.4	Expressions	5
1.2.5	Filtrage et Motifs	6
1.2.6	Types	6
2	Semantique de traduction	7
2.1	Programmes	7
2.2	Suites de commandes	7
2.3	Définitions	8
2.4	Types	8
2.5	Littéraux et Expressions	9
2.6	Motifs et Filtrage	11

1 Syntaxe MiniML

1.1 Jetons de lexing

1.1.1 Séparateurs

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

1.1.2 Mots Clefs

let fun in match with type of rec if then else

1.1.3 Types

int bool unit

1.1.4 Operateurs

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

1.1.5 Valeurs Atomiques

integer = $(-)?[0-9]$

boolean = $(true | false)$

1.1.6 Identificateur

alphanum = $[a-z A-Z 0-9 _]*$

basic_ident = $[a-z _]$ *alphanum*

vartype = $['a'] [0..9]*$

constructeur_ident = $[A-Z]$ *alphanum*

constructeur_infixes = $::, ,$

1.2 Grammaire

Voici la grammaire BNF de notre langage MiniML.

1.2.1 Outils de lecture

Pour simplifier la lecture de la grammaire, nous avons utilisé les notations suivantes :

- **Litteral** : Utilisation d'une règle définie
- $basic_ident_{list}$: Utilisation de la règle zéro ou plusieurs fois
- *Variable* : Noeud ciblé par le cas

1.2.2 Programmes

$$\begin{array}{lcl} \mathbf{Prog} := & | & \mathbf{Def} \\ & | & \mathbf{Expr} \\ & | & \mathbf{Prog} \ ; \ ; \ ; \ \mathbf{Prog} \end{array}$$

1.2.3 Definitions

$$\begin{array}{lcl} \mathbf{Def} := & | & \textit{let } basic_ident = \mathbf{Expr} \quad \textit{Variable} \\ & | & \textit{let } basic_ident_{list} = \mathbf{Expr} \quad \textit{Fonction} \\ & | & \textit{let rec } basic_ident_{list} = \mathbf{Expr} \quad \textit{Fonction Rec} \\ & | & \textit{type } vartype_{list} \ basic_ident = \mathbf{NewConstructors} \quad \textit{Type} \\ & | & \textit{type } vartype_{list} \ basic_ident \textit{ with } basic_ident_{list} = \mathbf{NewConstructors} \quad \textit{Type Paramétré} \end{array}$$
$$\begin{array}{lcl} \mathbf{NewConstructors} := & | & \textit{constructeur_ident} \\ & | & \textit{constructeur_ident of Type} \\ & | & \mathbf{NewConstructors} \ ' \ ' \ \mathbf{NewConstructors} \end{array}$$
$$\begin{array}{lcl} \mathbf{NewConstructorsParam} := & | & \textit{constructeur_ident of Type with ParamEquation} \\ & | & \mathbf{NewConstructorsParam} \ ' \ ' \ \mathbf{NewConstructorsParam} \end{array}$$
$$\begin{array}{lcl} \mathbf{ParamEquation} := & | & \textit{basic_ident} \\ & | & 0 \\ & | & 1 \\ & | & (\mathbf{ParamEquation}) \\ & | & \mathbf{ParamEquation} + \mathbf{ParamEquation} \\ & | & \mathbf{ParamEquation} * \mathbf{ParamEquation} \end{array}$$

1.2.4 Expressions

Litteral	:=	<i>integer</i>	
		<i>boolean</i>	
		<i>()</i>	<i>Unit</i>
Expr	:=	(Expr)	
		Litteral	
		<i>basic_ident</i>	
		UnaryOperator Expr	
		Expr BinaryOperator Expr	
		Expr Expr	<i>Call</i>
		Expr; Expr	<i>Sequence</i>
		<i>let basic_ident = Expr in Expr</i>	<i>Binding</i>
		<i>fun basic_ident_{list} → Expr</i>	<i>Lambda</i>
		Expr constructeur_infixes Expr	
		<i>constructeur_ident Expr</i>	<i>Construction</i>
		<i>constructeur_ident</i>	
		<i>let basic_ident_{list} = Expr in Expr</i>	<i>Fonction</i>
		<i>let rec basic_ident basic_ident_{list} = Expr in Expr</i>	<i>Fonction Recursive</i>
		<i>match Expr with Match_Case</i>	
		<i>if Expr then Expr else Expr</i>	
UnaryOperator	:=	<i>not</i>	
BinaryOperator	:=	&&	
		 	
		+	
		−	
		/	
		%	
		*	
		<	
		>	
		≤	
		≥	
		=	

1.2.5 Filtrage et Motifs

Match_Case	:=		Pattern \rightarrow Expr	
			Pattern \rightarrow Expr ' ' Match_Case	
Pattern	:=		(Pattern)	
			Litteral	
			<i>basic_ident</i>	
			—	
			<i>constructeur_ident</i>	
			<i>constructeur_ident</i> Pattern	

1.2.6 Types

Type	:=		(Type)	
			<i>int</i>	
			<i>bool</i>	
			<i>unit</i>	
			Type * Type	<i>Tuple Type</i>
			Type \rightarrow Type	<i>Lambda Type</i>
			<i>vartype</i>	<i>'a</i>
			<i>basic_ident</i>	<i>Defined Type</i>
			Type _{List}	<i>App Type</i>

2 Semantique de traduction

2.1 Programmes

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

Un programme MiniML est une suite de commandes $[cs]$ qui est traduite en un programme x en LCBPV. Un programme est dit traduisible si la suite de commandes $[cs]$ qui le compose peut-être traduite.

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

2.2 Suites de commandes

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

Le resultat de la traduction d'une suite de commandes cs est un triplet (γ, ω, v) où:

- γ est le resultat de la traduction des variables globales,
- ω est le resultat de la traduction des definitions de types
- v est la dernière expression traduite.

Ce triplet est rendu nécessaire par la sémantique de LCBPV qui ne permet pas de définir des variables globales comme en MiniML. Cela a aussi pour conséquence un changement de portée entre les déclarations de type en MiniML et en LCBPV.

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

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

(GLB EXPR) si $b \in EXPR$,
 et si $\vdash_{\text{Cmds}} cs \rightarrow (\gamma, \omega, v)$
 et si $\vdash_{\text{Expr}} b \rightarrow v'$
 alors $\vdash_{\text{Cmds}} (Expr(b), cs) \rightarrow (\gamma, \omega, v')$

2.3 Définitions

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

On définit la relation Def selon les cas de construction des définitions. Les cas de construction des définitions sont donnés par les clauses des règles syntaxiques.

Une définition est dite traduisible si chacune de ses clauses peut être traduite. On distingue deux catégories de définitions:

- Les définitions de **Variables Globales**,
- Les définitions de **Types**.

Ces deux catégories de définitions sont traitées différemment par le jugement \vdash_{Cmds}

- π est donc la traduction de la définition d placée dans la bonne catégorie.

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

(TYPDEF) si $\vdash c1 \rightarrow c1' \dots$ si $\vdash cN \rightarrow cN'$
 alors $\vdash_{\text{Def}} TypeDef(n, [t1, \dots, tn], [c1, \dots, cN])$
 $\rightarrow \text{TYPE}(TypeDef(n, [t1, \dots, tn], DefDatatype[c1', \dots, cN']))$

2.4 Types

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

On définit la relation \vdash_{Type} selon les cas de construction des types. Les cas de construction des types sont donnés par les clauses des règles syntaxiques. Un type est dit traduisible si chacune de ses clauses peut être traduite.

(TINT) $\vdash_{\text{Type}} TypeInt \rightarrow TypInt$
 (TBOOL) $\vdash_{\text{Type}} TypeBool \rightarrow TypBool$
 (TUNIT) $\vdash_{\text{Type}} TypeUnit \rightarrow TypUnit$

$$\begin{aligned}
& (\text{TDEF}) \vdash_{Type} TypeDefined(id) \rightarrow TypVar(id) \\
& (\text{TVAR}) \vdash_{Type} TypeVar(id) \rightarrow TypVar(id) \\
\\
& (\text{TTUPLE}) \text{ si } \vdash_{Type} t1 \rightarrow t_1, \dots \text{ et } \vdash_{Type} tN \rightarrow t_N \\
& \quad \text{alors } \vdash_{Type} TypeTuple([t1, \dots, tN]) \rightarrow TypTuple[t_1, \dots, t_N] \\
\\
& (\text{TCONS}) \text{ si } \vdash_{Type} t \rightarrow t' \\
& \quad \text{si } \vdash_{Type} p1 \rightarrow p_1, \dots \text{ et } \vdash_{Type} pN \rightarrow p_N \\
& \quad \text{alors } \vdash_{Type} TypeConstructor(t, [p1, \dots, pN]) \rightarrow TypApp(t, [p_1, \dots, p_N]) \\
\\
& (\text{TLAMB}) \text{ si } \vdash_{Type} a \rightarrow a' \\
& \quad \text{et si } \vdash_{Type} ret \rightarrow ret' \\
& \quad \text{alors } \vdash_{Type} TypeLambda(a, ret) \\
& \quad \rightarrow TypClosure(Exp, (TypFun(TypThunk(ret'), a')))
\end{aligned}$$

2.5 Litteraux et Expressions

$$\vdash_{Expr} e \rightarrow e'$$

On définit la relation \vdash_{Expr} selon les cas de construction des expr. Les cas de construction des expressions sont donnés par les clauses des règles syntaxiques. Une expression est dite traduisible si chacune de ses clauses peut être traduite.

$$\begin{aligned}
& (\text{INT}) \text{ si } i \in \text{NUM} \\
& \quad \text{alors } \vdash_{Expr} Integer(i) \rightarrow ExprInt(i) \\
& (\text{TRUE}) \quad \vdash_{Expr} Boolean(true) \rightarrow ExprConstructor(True, []) \\
& (\text{FALSE}) \quad \vdash_{Expr} Boolean(false) \rightarrow ExprConstructor(False, []) \\
& (\text{UNIT}) \quad \vdash_{Expr} Unit \rightarrow ExprConstructor(Unit, []) \\
\\
& (\text{TUPLE}) \text{ si } \vdash_{Expr} e1 \rightarrow e_1, \dots \text{ si } \vdash_{Expr} eN \rightarrow e_N \\
& \quad \text{alors } \vdash_{Expr} Tuple([e1, \dots, eN]) \rightarrow ExprConstructor(Tuple, [e_1, \dots, e_N]) \\
\\
& (\text{UNARY1}) \text{ si } \vdash_{Expr} a \rightarrow a' \\
& \quad \text{alors } \vdash_{Expr} CallUnary(op, [a]) \rightarrow ExprMonPrim(op, a') \\
& (\text{UNARY0}) \text{ si } \vdash_{Expr} Lambda(a, CallUnary(op, [a])) \rightarrow \omega \\
& \quad \text{alors } \vdash_{Expr} CallUnary(op, []) \rightarrow \omega
\end{aligned}$$

(BINARY2) si $\vdash_{Expr} a1 \rightarrow a_1$ et $\vdash_{Expr} a2 \rightarrow a_2$
 alors $\vdash_{Expr} CallBinary(op, [a1 : a2]) \rightarrow ExprBinPrim(op, a_1, a_2)$

(BINARY1) si $\vdash_{Expr} Lambda(b, CallUnary(op, [a, b])) \rightarrow \omega$
 alors $\vdash_{Expr} CallBinary(op, [a]) \rightarrow \omega$

(BINARY0) si $\vdash_{Expr} Lambda(a, Lambda(b, CallUnary(op, [a, b]))) \rightarrow \omega$
 alors $\vdash_{Expr} CallBinary(op, []) \rightarrow \omega$

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

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

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

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

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

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

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

2.6 Motifs et Filtrage

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

On définit la relation \vdash_{Case} selon les cas de construction des motif de correspondance. Les cas de construction des motif de correspondance sont donnés par les clauses des règles syntaxiques. Un motif de correspondance est dit traduisible si chacune de ses clauses peut être traduite.

- p est un motif
- e est l'expression qui sera évaluée si le motif est vérifié

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

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

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

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

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