MiniML Spec

Fazazi Zeid Luo Yukai Dibassi Brahima

2023-05-06

Table des matières

1 Syntaxe MiniML

1.1 Jetons de lexing

1.1.1 Séparateurs

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

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

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
- $basic\_ident_{\textbf{list}}$: Utilisation de la règle zéro ou plusieurs fois
- *Variable* : Noeud ciblé par le cas

1.2.2 Programmes

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

1.2.3 Definitions

$$
\begin{array}{rll}
\textbf{Def} := & | & let\ basic\_ident = \textbf{Expr} \quad \textit{Variable} \\
& | & let\ basic\_ident_{\textbf{list}} = \textbf{Expr} \quad \textit{Fonction} \\
& | & let\ rec\ basic\_ident_{\textbf{list}} = \textbf{Expr} \quad \textit{Fonction Rec} \\
& | & type\ vartype_{\textbf{list}}\ basic\_ident = \textbf{NewContructors} \quad \textit{Type} \\
& | & type\ vartype_{\textbf{list}}\ basic\_ident\ with\ basic\_ident_{\textbf{list}} = \textbf{NewContructors} \quad \textit{Type Parametré}
\end{array}
$$

$$
\begin{array}{rll}
\textbf{NewContructors} := & | & constructeur\_ident \\
& | & constructeur\_ident\ of\ \textbf{Type} \\
& | & \textbf{NewContructors}\ '|'\ \textbf{NewContructors}
\end{array}
$$

$$
\begin{array}{rll}
\textbf{NewContructorsParam} := & | & constructeur\_ident\ of\ \textbf{Type}\ with\ \textbf{ParamEquation} \\
& | & \textbf{NewContructorsParam}\ '|'\ \textbf{NewContructorsParam}
\end{array}
$$

$$
\begin{array}{rll}
\textbf{ParamEquation} := & | & basic\_ident \\
& | & 0 \\
& | & 1 \\
& | & (\ \textbf{ParamEquation}\ ) \\
& | & \textbf{ParamEquation} + \textbf{ParamEquation} \\
& | & \textbf{ParamEquation} * \textbf{ParamEquation}
\end{array}
$$

1.2.4 Expressions

Litteral	:=	*integer*	
	|	*boolean*	
	|	( )	*Unit*
Expr	:=	(**Expr**)	
	|	**Litteral**	
	|	*basic_ident*	
	|	**UnaryOperator** **Expr**	
	|	**Expr** **BinaryOperator** **Expr**	
	|	**Expr** **Expr**	*Call*
	|	**Expr**; **Expr**	*Sequence*
	|	*let basic_ident* = **Expr** *in* **Expr**	*Binding*
	|	$fun\ basic\_ident_{\textbf{list}} \rightarrow$ **Expr**	*Lambda*
	|	**Expr** $constructeur\_infixes$ **Expr**	
	|	*constructeur_ident* **Expr**	*Construction*
	|	*constructeur_ident*	
	|	$let\ basic\_ident_{\textbf{list}}$ = **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	:=	&&	
	|	||	
	|	+	
	|	−	
	|	/	
	|	%	
	|	*	
	|	<	
	|	>	
	|	≤	
	|	≥	
	|	=	

1.2.5 Filtrage et Motifs

Match_Case	:=	|	**Pattern** $\rightarrow$ **Expr**
		|	**Pattern** $\rightarrow$ **Expr** $'\|'$ **Match_Case**
Pattern	:=	|	(**Pattern**)
		|	**Litteral**
		|	*basic_ident*
		|	_
		|	*constructeur_ident*
		|	*constructeur_ident* **Pattern**

1.2.6 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

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

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.

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

2.2 Suites de commandes

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

Le resultat de la traduction d'une suite de commandes cs est un triplet $(\gamma, \omega, \upsilon)$ où:

- γ est le resultat de la traduction des variables globales,
- ω est le resultat de la traduction des definitions de types
- υ 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.

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

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

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

2.3 Définitions

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

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 $\vdash_{\text{Cmds}}$

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

(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 $\vdash_{\text{Def}} TypeDef(n, [t1, \ldots, tn], [c1, \ldots, cN])$
$\to \text{TYPE}(TypDef(n, [t1, \ldots, tn], DefDatatype[c1', \ldots, cN']))$

2.4 Types

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

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

(TDEF) $\vdash_{Type} TypeDefined(id) \rightarrow TypVar(id)$
(TVAR) $\vdash_{Type} TypeVar(id) \rightarrow TypVar(id)$

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

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

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

2.5 Litteraux et Expressions

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

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.

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

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

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

(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 $\vdash_{Expr} Match(m, [m1, \ldots, mN]) \to ExprMatch(m', [m_1, \ldots, m_N])$

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

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

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

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

2.6 Motifs et Filtrage

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

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