MiniML Spec

Fazazi Zeid Luo Yukai Dibassi Brahima

2023-05-06

Table des matières

1 Grammaire

Voici la grammaire BNF de notre langage MiniML.

1.1 Identificateurs

<Id> ::= ['a - z' 'A - Z' '0 - 9' '_']*
<ConstructeurId> ::= ['A - Z'] <Id>
<Vartype> ::= ''' ['a - z'] ['0 - 9']*

1.2 Programmes

<Prog> ::= | <Expr>
| <Def> ';;' <Prog>

1.3 Definitions

<Def> ::= | 'let' <Id> '=' <Expr>
| 'type' <Vartype>* <Id> '=' <NewContructors>

<NewContructors> ::= | <ConstructeurId> 'of' <Type>
| <ConstructeurId> 'of' <Type> 'with' <ParamAssigns>
| <NewContructors> '|' <NewContructors>

<ParamAssigns> ::= | <VarType> '=' <ParmExpr>
| <ParamAssigns> ',' <ParamAssigns>
<ParmExpr> ::= | '1'
| <VarType>
| <ParmExpr> '+' <ParmExpr>
| <ParmExpr> '*' <ParmExpr>

1.4 Expressions

<Litteral> ::= | ['0 - 9']+
| ['true' | 'false']
| '(' ')'

<Expr> ::= | <Litteral>
| <Id>
| <UnaryOperator>
| <BinaryOperator>
| '(' <Expr> <Expr> ')'
| 'let' <Id> '=' <Expr> 'in' <Expr>
| 'fun' <Id> → <Expr>
| 'fun' 'rec' <Id> <Id> → <Expr>
| <Expr> ',' <Expr>
| <ConstructeurId> <Expr>
| 'match' <Expr> 'with' <MatchCase>

<UnaryOperator> ::= 'not'
<BinaryOperator> ::= ['and' | 'or' | '+' | '-' | '/' | '%' | '*' | '<' | '>' | '=']

1.5 Filtrage et Motifs

<MatchCase> ::= | <Pattern> → <Expr>
| <MatchCase> '|' <MatchCase>

<Pattern> ::= | <Litteral>
| <Id>
| <Pattern> ',' <Pattern>
| <ConstructeurId> <Pattern>

1.6 Types

<Type> ::= | <Vartype>
| <Id>
| <Type> '*' <Type>
| '(' <Type> <Type> ')'

2 Semantique de traduction

2.1 Programmes

$$Prog[cs] \rightarrow [\![Prog[cs]]\!]_{Prog}$$

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.

$$Prog[cs] \rightarrow [\![Prog[cs]]\!]_{Cmds} \rightarrow Prog'([\![cs]\!]_{Cmds})$$

2.2 Suites de commandes

$$cs \rightarrow [\![cs]\!]_{Cmds} \rightarrow (\gamma, \omega, \upsilon)$$

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.

$$\text{(VAR DEFS)} \quad (Def(d); cs) \rightarrow [\![d]\!]_{Def} + [\![cs]\!]_{Cmds} \rightarrow ((\gamma; [\![d]\!]_{Def}), \omega, \upsilon)$$

$$\text{(TYPE DEFS)} \quad (Def(d); cs) \rightarrow [\![d]\!]_{Def} + [\![cs]\!]_{Cmds} \rightarrow (\gamma, (\omega; [\![d]\!]_{Def}), \upsilon)$$

$$\text{(GLB EXPR)} \quad (Expr(b), cs) \rightarrow [\![b]\!]_{Expr} \times [\![cs]\!]_{Cmds} \rightarrow (\gamma, \omega, [\![b]\!]_{Expr})$$

2.3 Définitions

$$d \to [\![d]\!]_{Def} \to \pi$$

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 $Cmds$

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

(VARDEF) $VariableDef(v, e) \to GLB(InsLet(v, [\![e]\!]_{Expr}))$

(TYPDEF) $TypeDef(n, [t_1, \ldots, t_N], [c_1, \ldots, c_N])$
$\to \text{TYPE}(TypDef(n, [t_1, \ldots, t_N], DefDatatype[[\![c_1]\!], \ldots, [\![c_N]\!]]))$

2.4 Types

$$t \to [\![t]\!]_{Type} \to t'$$

On définit la relation $Type$ 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.

(TLITT) $TypeInt \to [\![TypeInt]\!]_{Type} \to TypInt$

(TVAR) $TypeDefined(id) \to [\![TypeDefined(id)]\!]_{Type} \to TypVar(id)$

(TTUPLE) $TypeTuple([t_1, \ldots, t_N]) \to TypTuple[[\![t_1]\!]_{Type}, \ldots, [\![t_N]\!]_{Type}]$

(TAPP) $TypeConstructor(t, [p_1, \ldots, p_N]) \to TypApp([\![t]\!]_{Type}, [[\![p_1]\!]_{Type}, \ldots, [\![p_N]\!]_{Type}])$

(TCLOS) $TypeLambda(a, ret) \to TypClosure(Exp, (TypFun(TypThunk([\![ret]\!]_{Type}), [\![a]\!]_{Type})))$

2.5 Litteraux et Expressions

$$e \to [\![e]\!]_{Expr} \to e'$$

On définit la relation *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.

(CONSTR) $Construct(c, e) \rightarrow ExprConstructor(ConsNamed(c), [\![e]\!]_{Expr})$

(UNARY) $CallUnary(op, [a]) \rightarrow ExprMonPrim(op, [\![a]\!]_{Expr})$

(BINARY) $CallBinary(op, [a_1, a_2]) \rightarrow ExprBinPrim(op, [\![a_1]\!]_{Expr}, [\![a_2]\!]_{Expr})$

(MATCH) $Match(m, [m_1, \ldots, m_N]) \rightarrow ExprMatch([\![m]\!]_{Expr}, [[\![m_1]\!]_{Case}, \ldots, [\![m_N]\!]_{Case}])$

(BLOCK) $Sequence([e_1, \ldots, e_{N-1}, e_N]) \rightarrow ExprBlock(Blk([[\![e_1]\!]_{Expr}; \ldots; [\![e_{N-1}]\!]_{Expr}], e_N))$

(CALL) $Call(f, a) \rightarrow ExprBlock(Blk([InsOpen(Exp, [\![f]\!]_{Expr}),$
$InsForce(ExprMethod(Call, [[\![a]\!]_{Expr}]))]))$

(LAMBDA) $Lambda(a, b)$
$\rightarrow ExprClosure(Exp, ExprGet([GetPatTag(Call, [[\![a]\!]_{Expr}], ExprThunk([\![b]\!]_{Expr}))]))$

(REC) $FunctionRec(v, a, b) \rightarrow ExprClosure(Exp, ExprRec([\![v]\!]_{Expr},$
$ExprGet([GetPatTag(Call, [[\![a]\!]_{Expr}], ExprThunk([\![b]\!]_{Expr}))])))$

2.6 Motifs et Filtrage

$$Case(p, e) \rightarrow [\![Case(p, e)]\!]_{Case} \rightarrow \alpha$$

On définit la relation *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

(PATTAG) $Case(ConstructorPattern((n, c)), e)$
$\rightarrow MatchPatTag(ConsNamed(n), [\![c]\!]_{Case}, [\![e]\!]_{Expr})$

(PATVAR) $Case(VarPattern(x), e, l) \rightarrow MatchPatVar((x, l), [\![e]\!]_{Expr}, l)$