EITQ Project

Marc de Visme
marc.de-visme@inria.fr

For the 17th of December, 2022

The Project

Using Python3 and TatSu library,

1. Provide a grammar for the MiniML language described in Section 1. Then, like in the TP4, use the parse method on a few examples.
2. Provide a walker that compute the (big step) semantics of expressions of MiniML, as described in Section 2, and a use it on a few examples. *Advice : if you struggle, particularly with handling variables (and substitution), please continue past those issues. Most of the third question and all of the fourth question do not rely on you being able to handle the semantics of variables.*
3. Provide another grammar, the associated semantic walker, and a few examples, for CircuitML as defined in Section 3. Note that this second grammar will be slightly modified in the fourth question.
4. Provide a walker that compute the type of expressions of CircuitML. You can modify the second grammar to add type annotations as defined in Section 4. As always provide a few examples too.

Send the project as a single file named LASTNAME_FIRSTNAME.py to marc.de-visme@inria.fr before the Saturday 17th of december 23:59:59. This project is not a team project, and must be individual (you can still ask each other for explanations about the TP and the TatSu library).

1 The MiniML Language

Expressions of **MiniML** are defined by the following syntax:

e	::=	n	(n relative integer, no space between $-$ and the integer for negatives)
	|	$+$	(addition on integers)
	|	$-$	(subtraction on integers)
	|	$*$	(multiplication on integers)
	|	$/$	(euclidean division for integers)
	|	Fst	($(x, y) \mapsto x$)
	|	Snd	($(x, y) \mapsto y$)
	|	(e_1, e_2)	(pair of expressions)
	|	x	(a variable name, only consisting of lowercase letters)
	|	$x = e_1; e_2$	(assigning the value of e_1 to x to use it in e_2)
	|	Lambda $x : e$	(definition of a function)
	|	$e_1\ e_2$	(application of an expression to another)
	|	(e)	(parentheses)

With the following associativity rules[1]:

- $e_1\ e_2\ e_3 \quad \equiv \quad (e_1\ e_2)\ e_3$
- $x = e_1; e_2\ e_3 \quad \equiv \quad x = e_1; (e_2\ e_3)$
- $\text{Lambda}\ x : e_1\ e_2 \quad \equiv \quad \text{Lambda}\ x : (e_1\ e_2)$

In particular:

- No infix operations. So we write $+(1, 2)$ instead of $1 + 2$.
- No Boolean and no if/then/else, contrary to the course. They will be in **CircuitML**.
- No float, contrary to what the TP said.
- We do not impose a limit on the size of integers, and assume that Python is handling that correctly.

You will not be penalized for adding any reasonable feature.

2 Semantics of the MiniML Language

Some of those expressions are considered to be values:

v	::=	n	(n relative integer, no space between $-$ and the integer for negatives)
	|	$+$	(addition on integers)
	|	$-$	(subtraction on integers)
	|	$*$	(multiplication on integers)
	|	$/$	(euclidean division for integers)
	|	Fst	($(x, y) \mapsto x$)
	|	Snd	($(x, y) \mapsto y$)
	|	(v_1, v_2)	(pair of values)
	|	x	(a variable name, only consisting of lowercase letters)
	|	$\text{Lambda}\ x : e$	(definition of a function, e can be a non-value)

In other words, unless they are under a Lambda, no application, no variable definition, and no parentheses (except for pairs).

The semantics of MiniML computes for any expression e a value v, which we write $e \Rightarrow v$. In the project, we expect you to code a semantics, so a Walker (see TP4) that goes through the tree of e and outputs v (and "None" or an error when no such value exists). You might need to first make a function computing the substitution[2], which we recall is defined inductively as follows[3]:

Expression e	**Expression** $e[x \leftarrow v]$	**Condition**
n	n	$n \in \mathbb{Z}$
OP	OP	$\text{OP} \in \{+, -, *, /, \text{Fst}, \text{Snd}\}$
(e_1)	$(e_1[x \leftarrow v])$	
(e_1, e_2)	$(e_1[x \leftarrow v], e_2[x \leftarrow v])$	
$e_1\ e_2$	$e_1[x \leftarrow v]\ e_2[x \leftarrow v]$	
x	v	
$x = e_1; e_2$	$x = e_1[x \leftarrow v]; e_2$	
$\text{Lambda}\ x : e_1$	$\text{Lambda}\ x : e_1$	
y	y	$y \neq x$
$y = e_1; e_2$	$z = e_1[y \leftarrow z][x \leftarrow v]; e_2[y \leftarrow z][x \leftarrow v]$	$y \neq x,\ z \notin e_1, z \notin e_2, z \notin v$
$\text{Lambda}\ y : e_1$	$\text{Lambda}\ z : e_1[y \leftarrow z][x \leftarrow v]$	$y \neq x,\ z \notin e_1, z \notin v$

[1] While the notation method has yet to be determined, getting the associativity wrong will only result to a penalty of at most 20% of the points associated to the question.

[2] Other solutions might exist.

[3] In the last two lines, we rename the variable y to an arbitrary z to avoid variable capture. This is to ensure that $(\text{Lambda}\ y : x)[x \leftarrow y]$ gives the constant function $\text{Lambda}\ z : y$ and not the identity function $\text{Lambda}\ y : y$.

We expect the semantics to satisfy the following rules:

$$\frac{n \in \mathbb{Z}}{n \Rightarrow n} \qquad \frac{\text{OP} \in \{+, -, *, /, \text{Fst}, \text{Snd}\}}{\text{OP} \Rightarrow \text{OP}} \qquad \frac{\text{x lowercase name}}{\text{x} \Rightarrow \text{x}} \qquad \frac{}{\text{Lambda } x : e \Rightarrow \text{Lambda } x : e}$$

$$\frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2}{(e_1, e_2) \Rightarrow (v_1, v_2)} \qquad \frac{e_1 \Rightarrow \text{Fst} \quad e_2 \Rightarrow (v_1, v_2)}{e_1\ e_2 \Rightarrow v_1} \qquad \frac{e_1 \Rightarrow \text{Snd} \quad e_2 \Rightarrow (v_1, v_2)}{e_1\ e_2 \Rightarrow v_2}$$

$$\frac{e \Rightarrow v}{(e) \Rightarrow v} \qquad \frac{e_1 \Rightarrow v_1 \quad e_2[x \leftarrow v_1] \Rightarrow v_2}{x = e_1; e_1 \Rightarrow v_2} \qquad \frac{e_1 \Rightarrow \text{Lambda } x : e_3 \quad e_2 \Rightarrow v_2 \quad e_3[x \leftarrow v_2] \Rightarrow v_2}{e_1\ e_2 \Rightarrow v_2}$$

$$\frac{e_1 \Rightarrow + \quad e_2 \Rightarrow (n_1, n_2) \quad n_1 + n_2 = n_3}{e_1\ e_2 \Rightarrow n_3} \qquad \frac{e_1 \Rightarrow - \quad e_2 \Rightarrow (n_1, n_2) \quad n_1 - n_2 = n_3}{e_1\ e_2 \Rightarrow n_3}$$

$$\frac{e_1 \Rightarrow * \quad e_2 \Rightarrow (n_1, n_2) \quad n_1 * n_2 = n_3}{e_1\ e_2 \Rightarrow n_3} \qquad \frac{e_1 \Rightarrow / \quad e_2 \Rightarrow (n_1, n_2) \quad n_2 \neq 0 \quad \text{floor}(n_1/n_2) = n_3}{e_1\ e_2 \Rightarrow n_3}$$

3 The CircuitML Language

Expressions of **CircuitML** are defined by the following syntax:

e	::=	True	
	|	False	
	|	!	(negation)
	|	&	(conjunction)
	|	|	(disjunction)
	|	*Nand*	(negation of conjunction)
	|	Fst	$((x, y) \mapsto x)$
	|	Snd	$((x, y) \mapsto y)$
	|	(e_1, e_2)	(pair of expressions)
	|	x	(a variable name, only consisting of lowercase letters)
	|	$x = e_1; e_2$	(assigning the value of e_1 to x to use it in e_2)
	|	Lambda $x : e$	(definition of a function)
	|	$e_1\ e_2$	(application of an expression to another)
	|	(e)	(parentheses)
	|	If e_1 Then e_2 Else e_3 Endif	(conditional)

Similarly to **MiniML**, we don't use infix operators (so write &(True, False)), and we have the same associativity rules. We consider a If-Then-Else-Endif syntax, you are free to use the "Opif" operator defined in the course instead. Then, the values and semantics are defined as follows:

v	::=	True	
	|	False	
	|	!	(negation)
	|	&	(conjunction)
	|	|	(disjunction)
	|	Nand	(negation of conjunction)
	|	Fst	$((x, y) \mapsto x)$
	|	Snd	$((x, y) \mapsto y)$
	|	(v_1, v_2)	(pair of values)
	|	x	(a variable name, only consisting of lowercase letters)
	|	Lambda $x : e$	(definition of a function, e can be a non-value)

$$\frac{b \in \{\text{True}, \text{False}\}}{b \Rightarrow b} \qquad \frac{\text{OP} \in \{!, \&, |, \text{Nand}, \text{Fst}, \text{Snd}\}}{\text{OP} \Rightarrow \text{OP}} \qquad \frac{\text{x lowercase name}}{\text{x} \Rightarrow \text{x}} \qquad \frac{}{\text{Lambda } x : e \Rightarrow \text{Lambda } x : e}$$

$$\frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2}{(e_1, e_2) \Rightarrow (v_1, v_2)} \qquad \frac{e_1 \Rightarrow \text{Fst} \quad e_2 \Rightarrow (v_1, v_2)}{e_1\ e_2 \Rightarrow v_1} \qquad \frac{e_1 \Rightarrow \text{Snd} \quad e_2 \Rightarrow (v_1, v_2)}{e_1\ e_2 \Rightarrow v_2}$$

$$\frac{e \Rightarrow v}{(e) \Rightarrow v} \qquad \frac{e_1 \Rightarrow v_1 \quad e_2[x \leftarrow v_1] \Rightarrow v_2}{x = e_1; e_1 \Rightarrow v_2}$$

$$\frac{e_1 \Rightarrow \text{True} \quad e_2 \Rightarrow v_2}{\text{If } e_1 \text{ Then } e_2 \text{ Else } e_3 \text{ Endif} \Rightarrow v_2} \qquad \frac{e_1 \Rightarrow \text{False} \quad e_3 \Rightarrow v_3}{\text{If } e_1 \text{ Then } e_2 \text{ Else } e_3 \text{ Endif} \Rightarrow v_3}$$

$$\frac{e_1 \Rightarrow ! \quad e_2 \Rightarrow b_2 \quad \text{not } b_2 = b_3}{e_1\ e_2 \Rightarrow b_3} \qquad \frac{e_1 \Rightarrow \& \quad e_2 \Rightarrow (b_1, b_2) \quad b_1 \text{ and } b_2 = b_3}{e_1\ e_2 \Rightarrow b_3}$$

$$\frac{e_1 \Rightarrow | \quad e_2 \Rightarrow (b_1, b_2) \quad b_1 \text{ or } b_2 = b_3}{e_1\ e_2 \Rightarrow b_3} \qquad \frac{e_1 \Rightarrow \text{Nand} \quad e_2 \Rightarrow (b_1, b_2) \quad \text{not}(b_1 \text{ and } b_2) = b_3}{e_1\ e_2 \Rightarrow b_3}$$

4 Typing System for CircuitML

Our types are generated by the following syntax:

$$\begin{array}{rcll} \tau & ::= & B & \text{(Boolean)} \\ & | & (\tau_1 * \tau_2) & \text{(Pair)} \\ & | & (\tau_1 -> \tau_2) & \text{(Function)} \end{array}$$

We change a few lines in the syntax of our grammar to add type annotations[4]

$$\begin{array}{rcl} e & ::= & \ldots \\ & | & \text{Fst}\{(\tau_1 * \tau_2) -> \tau_1\} \\ & | & \text{Snd}\{(\tau_1 * \tau_2) -> \tau_2\} \\ & | & \text{Lambda } x\{\tau\} : e \end{array}$$

A context Γ is a list of pairs (x_i, τ_i) with x_i variable names and τ_i types. Typing consists in finding the type τ of an expression e, under a context Γ that specifies the type of its free variables. We write $\Gamma \vdash e : \tau$. In the project, we expect you to write a Walker that goes through the tree of e, takes as an additional input a context Γ, and outputs a τ such that $\Gamma \vdash e : \tau$ (and "None" or an Error if there is none). This typing has to satisfies the following rules:

$$\frac{b \in \{\text{True}, \text{False}\}}{\Gamma \vdash b : B} \qquad \frac{}{\Gamma \vdash ! : B -> B} \qquad \frac{\text{OP} \in \{!, \&, |, \text{Nand}\}}{\Gamma \vdash \text{OP} : (B * B) -> B} \qquad \frac{\text{OP} \in \{\text{Fst}, \text{Snd}\}}{\Gamma \vdash \text{OP}\{\tau\} : \tau} \qquad \frac{(\text{x}, \tau) \in \Gamma}{\Gamma \vdash \text{x} : \tau}$$

$$\frac{(x, \tau_1), \Gamma \vdash e : \tau_2}{\Gamma \vdash \text{Lambda } x\{\tau_1\} : e : \tau_1 -> \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : (\tau_1 * \tau_2)} \qquad \frac{\Gamma \vdash e_1 : \tau_1 -> \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash (e) : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad (x, \tau_1), \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash x = e_1; e_2 : \tau_2} \qquad \frac{\Gamma \vdash e_1 : B \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \text{If } e_1 \text{ Then } e_2 \text{ Else } e_3 \text{ Endif} : \tau}$$

[4]This ensure that terms have at most one type. Otherwise Lambda $x : x$ could have the type $B -> B$ or $(B -> B) -> (B -> B)$, or many others. Adding polymorphic types to handle this situation is doable, but require a lot of work.