Action Semantics of MICROML

Hermano Moura *
University of Glasgow
Department of Computing Science
Glasgow, G12 8QQ, Scotland
moura@dcs.glasgow.ac.uk

20th August 1992

Abstract

This document gives the definition of MICROML, a simple functional language designed to serve as an example language in our research on semantics-directed compiler generation.

Contents

*Supported by CNPq, Brazil. On leave from Caixa Econômica Federal, Brazil.

1 Abstract Syntax

needs: Lexical Syntax .

closed Abstract Syntax .

grammar:

(1) Program = Declaration **|**
Declaration Program .

(2) Declaration = `val` Identifier = Expression **|**
`val rec` Identifier = `fn` Identifier => Expression .

(3) Expression = `true` **|**
`false` **|**
Numeral **|**
Identifier **|**
`fn` Identifier => Expression **|**
Expression (Expression) **|**
`if` Expression `then` Expression `else` Expression **|**
Expression Operator Expression **|**
Operator Expression .

(4) Operator = `not` **|** `and` **|** `or` **|** `~` **|** `+` **|** `-` **|** `*` **|** `/` **|** `=` .

2 Semantic Entities

needs: Action Notation , **Data Notation** .

2.1 Data

introduces: value , operand , operation-result .

needs: Abstractions .

(1) value = truth-value **|** integer **|** function .

(2) operand = truth-value **|** integer .

(3) operation-result = truth-value **|** integer .

2.2 Bindings

needs: Data , **Abstractions** .

(1) token $\leq$ string .

(2) bindable(t:token) = value .

2.3 Abstractions

introduces: function , argument , function-result .

needs: Data .

(1) function = abstraction .

(2) argument = value .

(3) function-result = value .

3 Semantic Functions

needs: Abstract Syntax , Semantic Entities .

3.1 Programs

introduces: run ⟦ _ ⟧ .

needs: Declarations .

- run ⟦ _ ⟧ :: Program → act .

(1) run ⟦ D ⟧ = elaborate ⟦ D ⟧ .

(2) run ⟦ D P ⟧ = elaborate ⟦ D ⟧ before run ⟦ P ⟧ .

3.2 Declarations

introduces: elaborate ⟦ _ ⟧ .

needs: Expressions .

- elaborate ⟦ _ ⟧ :: Declaration → act .

(1) elaborate ⟦ `val` I `=` E ⟧ = evaluate ⟦ E ⟧ then bind token of ⟦ I ⟧ to it .

(2) elaborate ⟦ `val rec` I_1 `=` `fn` I_2 `=>` E ⟧ =
    recursively bind token of ⟦ I_1 ⟧ to closure abstraction
        | | rebind moreover bind token of ⟦ I_2 ⟧ to the argument
        | hence
        | | evaluate ⟦ E ⟧ .

3.3 Expressions

introduces: evaluate ⟦ _ ⟧ , evaluate operator ⟦ _ ⟧ .

3.3.1 Evaluating Expressions

- evaluate ⟦ _ ⟧ :: Expression → act .

(1) evaluate ⟦ `true` ⟧ = give true .

(2) evaluate ⟦ `false` ⟧ = give false .

(3) evaluate ⟦ N ⟧ = give valuation ⟦ N ⟧ .

(4) evaluate ⟦ I ⟧ = give the value bound to token of ⟦ I ⟧ .

(5) evaluate ⟦ `fn` I `=>` E ⟧ =
    give closure abstraction
        | | rebind moreover bind token of ⟦ I ⟧ to the argument
        | hence
        | | evaluate ⟦ E ⟧ .

(6) evaluate ⟦ E_1 (E_2) ⟧ =
    | | evaluate ⟦ E_1 ⟧ then give the abstraction label #1
    | and
    | | evaluate ⟦ E_2 ⟧ then give the argument label #2
    then
    | enact the abstraction#1 with the argument#2 .

(7) evaluate ⟦ `if` E_1 `then` E_2 `else` E_3 ⟧ =
| evaluate ⟦ E_1 ⟧ then give the truth-value
then
| | check (it is true) and then evaluate ⟦ E_2 ⟧
| or
| | check (it is false) and then evaluate ⟦ E_3 ⟧ .

(8) evaluate ⟦ E_1 O E_2 ⟧ =
| | evaluate ⟦ E_1 ⟧ then give the operand label #1
| and
| | evaluate ⟦ E_2 ⟧ then give the operand label #2
then
| evaluate operator ⟦ O ⟧ .

(9) evaluate ⟦ O E ⟧ =
| evaluate ⟦ E ⟧ then give the operand
then
| evaluate operator ⟦ O ⟧ .

3.3.2 Evaluating Operators

- evaluate operator ⟦ _ ⟧ :: Operator → act .

(1) evaluate operator ⟦ `not` ⟧ = give not (the truth-value) .

(2) evaluate operator ⟦ `and` ⟧ = give conjunction (the truth-value#1, the truth-value#2) .

(3) evaluate operator ⟦ `or` ⟧ = give disjunction (the truth-value#1, the truth-value#2) .

(4) evaluate operator ⟦ `~` ⟧ = give negation (the integer) .

(5) evaluate operator ⟦ `+` ⟧ = give sum (the integer#1, the integer#2) .

(6) evaluate operator ⟦ `-` ⟧ = give difference (the integer#1, the integer#2) .

(7) evaluate operator ⟦ `*` ⟧ = give product (the integer#1, the integer#2) .

(8) evaluate operator ⟦ `/` ⟧ = give quotient (the integer#1, the integer#2) .

(9) evaluate operator ⟦ `=` ⟧ = give (the operand#1 is the operand#2) .

4 Lexical Syntax

introduces: token of ⟦ _ ⟧ , valuation ⟦ _ ⟧ .

grammar:

(1) Identifier = letter **|**
Identifier letter **|**
Identifier digit .

(2) Numeral = digit **|**
Numeral digit .

(3) digit = `0` **|** `1` **|** ... **|** `9` .

(4) letter = `A` **|** `B` **|** ... **|** `Z` **|** `a` **|** ... **|** `z` .

4.1 Identifiers

- token of $[\![\_]\!]$ :: Identifier $\rightarrow$ token .

(1) token of $[\![I]\!] = I$.

4.2 Numerals

- valuation $[\![\_]\!]$ :: Numeral $\rightarrow$ integer .

(1) valuation $[\![0]\!] = 0$.

(2) valuation $[\![1]\!] = 1$.

...

(10) valuation $[\![9]\!] = 9$.

(11) valuation $[\![N\ D]\!] =$ sum (product (valuation $[\![N]\!]$, 10), valuation $[\![D]\!]$) .