Action Semantics of microSpecimen

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

2nd September 1992

Abstract

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

Contents

1 Informal Description **1**
1.1 Programs . . . 1
1.2 Declarations . . . 2
1.3 Commands . . . 3
1.4 Expressions . . . 4
1.5 Types . . . 5
1.6 Types and Contextual Constraints . . . 5

2 Abstract Syntax **5**
2.1 Programs . . . 5
2.2 Declarations . . . 5
2.3 Commands . . . 5
2.4 Expressions . . . 6
2.5 Types . . . 6

3 Semantic Entities **6**
3.1 Data . . . 6
3.2 Bindings . . . 6
3.3 Storage . . . 6
3.4 Abstractions . . . 7

4 Semantic Functions **7**
4.1 Programs . . . 7
4.2 Declarations . . . 7
4.2.1 Elaborating Declarations . . . 7
4.2.2 Elaborating Procedure Formal Parameter . . . 8
4.3 Commands . . . 8

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

4.3.1 Executing Commands 8
4.3.2 Evaluating a Procedure Argument 9
4.4 Expressions 9
4.4.1 Evaluating Expressions 9
4.4.2 Evaluating Operators 10

5 Lexical Syntax **10**
5.1 Identifiers 10
5.2 Numerals 10

1 Informal Description

microSpecimen is a simple imperative programming language. It has two types of abstractions: procedures and functions. Functions are free of side effects, although they can access non-local variables. Integers and booleans are the only data types in the language. microSpecimen is a member of a group of four imperative languages used in research on semantics-directed compiler generation. (The others three are nanoSpecimen, milliSpecimen and Specimen.)

1.1 Programs

Syntax

Program = `program` Identifier = Declaration `in` Command `end`

Semantics

A microSpecimen program is a declaration (sequence) followed by a command (sequence). The execution of a program consists of the elaboration of the declaration and then the execution of the command using the bindings produced by the declaration. These bindings are visible throughout the command (global bindings); holes in the scope of global bindings can occur by local declarations of the same global identifier.

Examples

We show below a microSpecimen program that computes the factorial of 4.

```
program fact4 =
    var x : int := 4;
    proc fact (var n : int) =
        local
            var temp : int = 1
        in
            while not (n = 1)
                do
                    temp := temp * n;
                    n := n - 1
                end;
            n := temp
        end
in
    fact (var x)
end
```

The following program, which also computes the factorial of 9, uses a function instead of a procedure. Note the recursive declaration of `fact`.

```
program fact9 =
    var x : int = 9;
    fun fact n : int -> int = if n = 0 then 1 else n * fact (n - 1)
in
    x := fact x
end
```

1.2 Declarations

Syntax

Declaration = `const` Identifier : Type = Expression |
`var` Identifier : Type := Expression |
`proc` Identifier ProcFormal = Command `end` |
`fun` Identifier Identifier : Type -> Type = Expression |
Declaration ; Declaration

ProcFormal = () |
(Identifier : Type) |
(`var` Identifier : Type)

Semantics

Declarations allow the programmer to make bindings. MICROSPECIMEN has four types of declarations (bindings): constant, variable, procedure and function declaration. Procedures and functions declarations can be recursive.

A constant declaration binds an identifier to a constant value. If possible, this value can be evaluated at compile-time.

A variable declaration binds an identifier to a cell. The contents of the cell is the result of the evaluation of the expression.

A procedure declaration binds an identifier to a procedure. A procedure is a command abstraction. Zero or one formal parameter can be defined in the procedure declaration. The body of a procedure is just a command. No result is returned, a procedure works by its side effects only.

A function declaration binds an identifier to a function. A function is an expression abstraction. All MICROSPECIMEN functions are unary (have just one argument). The body of a function is just an expression. Functions always return a value.

Examples

- `const year : int = 1991`
- `const yearPlusOne : int = year + 1`
- `const factOfFour : int = fact 4`
- `var x : bool = false`
- `var y : bool = not x`
- `proc inc (var x : int) = x := x + 1 end`
- `fun isTrue b : bool -> bool = if b then true else false`

1.3 Commands

A command is used to update variables. MICROSPECIMEN has five types of commands: assignment, conditional, while, procedure call, and block.

Syntax

```
Command       = Identifier := Expression |
                if Expression then Command else Command |
                while Expression do Command end |
                local Declaration in Command end |
                Identifier () |
                Identifier ( Proc Argument ) |
                Command ; Command

Proc Argument = Expression |
                var Identifier
```

Semantics

The assignment is the simplest form of a command. It assigns the value of an expression to a variable. The expression is evaluated and the resultant value is assigned to the cell bound to the variable (identifier).

The conditional command is executed as follows: the expression is evaluated; if the evaluation result is true then the `then` branch command is executed otherwise the `else` branch command is executed.

The while-command is executed as follows: (a) the expression is evaluated; (b) if its value is true then the subcommand is executed and then we start from step (a) again; else, if the value is false, the while-command is terminated.

A procedure call, P`(`A`)`, causes the execution of the procedure bound to P. A parameter can be passed by value or by reference (`var` formal). By value, P`(`E`)`, the actual parameter is evaluated and its value is bound to the formal argument; by reference, P`(var` I`)`, the actual parameter is evaluated and its l-value is bound to the formal parameter. In this case the actual parameter must be a variable and the value passed is a cell. Neither procedures nor functions can be passed as parameters.

Examples

- `i := 314`
- `x := x + 1`
- `if x = 0 then x := x + 1 else x := x - 1`
- `while x = true do x := true and true end`
- `if tag then process(true) else process(false)`

1.4 Expressions

Expressions are evaluated yielding a value (an integer or a boolean). Expressions are completely free of side effects.

Syntax

```
Expression = true |
             false |
             Numeral |
             Identifier |
             if Expression then Expression else Expression |
             Identifier Expression |
             Expression Operator Expression |
             Operator Expression |
             ( Expression )

Operator   = not | and | or | + | - | * | / | = | <
```

Semantics

A literal expression is one that does not need additional evaluation (it is still in normal form). MICROSPECIMEN has two types of literal expressions: boolean and integer.

An identifier is an expression. Its evaluation gives the value bound to it, in the case of a constant; or the procedure or function bound to it, in the case of a procedure or function identifier, respectively; or the content of the cell bound to it, in the case of a variable.

The conditional expression has a semantics very similar to the conditional command, except that the `then` and `else` branches are expressions instead of commands.

The function application, IE, evaluates as follows: E is evaluated; each occurrence of the formal parameter inside the function body is replaced by the resultant value; the function body is evaluated and its result is returned as the result of the function application.

Operators...

Examples

- `true`
- `45637384746`
- `x`
- `if tag then process tag else un process tag`
- `g (x + 3) - f (x - 3)`
- `- 3`
- `3 + 4 - 4 * 3 - 9`
- `(3 + 4) - (4 * 3 - 9)`

1.5 Types

MICROSPECIMEN has only two types: `int` and `bool`. Types of all constants, variables, procedures and functions must be explicitly declared.

1.6 Types and Contextual Constraints

Although we only specified the dynamic semantics (a static semantics will be given later), we are going to state informally some types and contextual constraints that help the user to write valid MICROSPECIMEN programs.

2 Abstract Syntax

needs: Lexical Syntax .

closed Abstract Syntax .

2.1 Programs

needs: Declarations , Commands .

grammar:

(1) Program = `program` Identifier = Declaration `in` Command `end` .

2.2 Declarations

needs: Commands , Expressions , Types .

grammar:

(1) Declaration = `const` Identifier : Type = Expression |
`var` Identifier : Type := Expression |
`proc` Identifier ProcFormal = Command `end` |
`fun` Identifier Identifier : Type -> Type = Expression |
Declaration ; Declaration .

(2) ProcFormal = () |
(Identifier : Type) |
(`var` Identifier : Type) .

2.3 Commands

grammar:

(1) Command = Identifier := Expression |
`if` Expression `then` Command `else` Command |
`while` Expression `do` Command `end` |
`local` Declaration `in` Command `end` |
Identifier () |
Identifier (ProcArgument) |
Command ; Command .

(2) ProcArgument = Expression |
`var` Identifier .

2.4 Expressions

grammar:

(1) Expression = `true` |
`false` |
Numeral |
Identifier |
`if` Expression `then` Expression `else` Expression |
Identifier Expression |
Expression Operator Expression |
Operator Expression |
(Expression) .

(2) Operator = `not` | `and` | `or` | `+` | `-` | `*` | `/` | `=` | `<` .

2.5 Types

grammar:

(1) Type = `bool` | `int` .

3 Semantic Entities

needs: **Standard** .

3.1 Data

introduces: value .

(1) value = truth-value | integer .

3.2 Bindings

needs: **Data** , **Storage** , **Abstractions** .

(1) token = symbol .

(2) bindable(T:token) = value | procedure | function | cell .

3.3 Storage

introduces: truth-value-cell , integer-cell , allocate-a-cell-for ⟦ _ ⟧ .

(1) cell = truth-value-cell | integer-cell .

(2) storable(C:truth-value-cell) = truth-value .

(3) storable(C:integer-cell) = integer .

- allocate-a-cell-for ⟦ _ ⟧ :: Type → act .

(4) allocate-a-cell-for ⟦ `bool` ⟧ = allocate a truth-value-cell .

(5) allocate-a-cell-for ⟦ `int` ⟧ = allocate an integer-cell .

3.4 Abstractions

introduces: procedure , function , proc-argument , fun-argument , fun-result .

needs: **Data** , **Storage** .

(1) procedure = abstraction .

(2) function = abstraction .

(3) proc-argument = value | cell .

(4) fun-argument = value .

(5) fun-result = value .

4 Semantic Functions

needs: **Abstract Syntax** , **Semantic Entities** .

4.1 Programs

introduces: run ⟦ _ ⟧ .

needs: **Declarations** , **Commands** .

- run ⟦ _ ⟧ :: Program → act .

(1) run ⟦ `program` I = D `in` C `end` ⟧ =
| elaborate ⟦ D ⟧
hence
| execute ⟦ C ⟧ .

4.2 Declarations

introduces: elaborate ⟦ _ ⟧ ,
elaborate-proc-formal ⟦ _ ⟧ .

needs: **Commands** , **Expressions** , **Types** .

4.2.1 Elaborating Declarations

- elaborate ⟦ _ ⟧ :: Declaration → act .

(1) elaborate ⟦ `const` I : T = E ⟧ =
| evaluate ⟦ E ⟧
then
| bind token-of ⟦ I ⟧ to the value .

(2) elaborate ⟦ `var` I : T = E ⟧ =
| | evaluate ⟦ E ⟧ then give the value label #1
| and
| | allocate-a-cell-for ⟦ T ⟧ then give it label #2
then
| | bind token-of ⟦ I ⟧ to the cell#2
| and
| | store the value#1 in the cell#2 .

(3) elaborate ⟦ `proc` I F = C `end` ⟧ =
recursively bind token-of ⟦ I ⟧ to closure abstraction
| | rebind moreover elaborate-proc-formal ⟦ F ⟧
hence
| | execute ⟦ C ⟧ .

(4) elaborate ⟦ `fun` I_1 I_2 : T_1 -> T_2 = E ⟧ =
recursively bind token-of ⟦ I_1 ⟧ to closure abstraction
| | rebind moreover bind token-of ⟦ I_2 ⟧ to the fun-argument
hence
| | evaluate ⟦ E ⟧ then give the fun-result .

(5) elaborate ⟦ D_1 ; D_2 ⟧ =
| elaborate ⟦ D_1 ⟧
before
| elaborate ⟦ D_2 ⟧ .

4.2.2 Elaborating Procedure Formal Parameter

- elaborate-proc-formal ⟦ _ ⟧ :: ProcFormal → act .

(1) elaborate-proc-formal ⟦ () ⟧ = complete .

(2) elaborate-proc-formal ⟦ I : T ⟧ =
    | give the value label #1 and allocate a cell
    then
    | bind token-of ⟦ I ⟧ to it and store the value#1 in it .

(3) elaborate-proc-formal ⟦ `var` I : T ⟧ = bind token-of ⟦ I ⟧ to the cell .

4.3 Commands

introduces: execute ⟦ _ ⟧ , evaluate-proc-argument ⟦ _ ⟧ .

needs: **Declarations** , **Expressions** , **Literals** .

4.3.1 Executing Commands

- execute ⟦ _ ⟧ :: Command → act .

(1) execute ⟦ I := E ⟧ =
    | evaluate ⟦ E ⟧
    then
    | store the value in the cell bound to token-of ⟦ I ⟧ .

(2) execute ⟦ `if` E `then` C_1 `else` C_2 ⟧ =
    | evaluate ⟦ E ⟧
    then
    | | check (it is true) and then execute ⟦ C_1 ⟧
    | or
    | | check (it is false) and then execute ⟦ C_2 ⟧ .

(3) execute ⟦ `while` E `do` C ⟧ =
    unfolding
    | | evaluate ⟦ E ⟧
    | then
    | | | check the truth-value and then execute ⟦ C ⟧ and then unfold
    | | or
    | | | check not the truth-value .

(4) execute ⟦ `local` D `in` C `end` ⟧ =
    | rebind moreover elaborate ⟦ D ⟧
    hence
    | execute ⟦ C ⟧ .

(5) execute ⟦ I () ⟧ =
    enact the procedure bound to token-of I .

(6) execute ⟦ I (A) ⟧ =
    | evaluate-proc-argument ⟦ A ⟧
    then
    | enact (the procedure bound to token-of ⟦ I ⟧ with the proc-argument) .

(7) execute ⟦ C_1 ; C_2 ⟧ = execute ⟦ C_1 ⟧ and then execute ⟦ C_2 ⟧ .

4.3.2 Evaluating a Procedure Argument

- evaluate-proc-argument ⟦ _ ⟧ :: ProcArgument → act .

(1) evaluate-proc-argument ⟦ E : Expression ⟧ = evaluate E .

(2) evaluate-proc-argument ⟦ `var` I : Identifier ⟧ = give the cell bound to token-of I .

4.4 Expressions

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

4.4.1 Evaluating Expressions

- evaluate ⟦ _ ⟧ :: Expression → act .

(1) evaluate ⟦ `true` ⟧ = give true .

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

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

(4) evaluate ⟦ I ⟧ =
| give the value bound to token-of ⟦ I ⟧
or
| give the value stored in the cell bound to token-of ⟦ I ⟧ .

(5) evaluate ⟦ `if` E `then` E_1 `else` E_2 ⟧ =
| evaluate ⟦ E ⟧
then
| | check (it is true) and then evaluate ⟦ E_1 ⟧
| or
| | check (it is false) and then evaluate ⟦ E_2 ⟧ .

(6) evaluate ⟦ I E ⟧ =
| evaluate ⟦ E ⟧
then
| enact (the function bound to token-of ⟦ I ⟧ with the fun-argument) .

(7) evaluate ⟦ E_1 O E_2 ⟧ =
| | evaluate ⟦ E_1 ⟧ then give the value label #1
| and
| | evaluate ⟦ E_2 ⟧ then give the value label #2
then
| apply-operator ⟦ O ⟧ .

(8) evaluate ⟦ O E ⟧ =
| evaluate ⟦ E ⟧ then give the value
then
| apply-operator ⟦ O ⟧ .

(9) evaluate ⟦ (E) ⟧ = evaluate ⟦ E ⟧ .

4.4.2 Evaluating Operators

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

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

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

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

(4) apply-operator ⟦ + ⟧ = give sum (the integer#1, the integer#2) .

(5) apply-operator ⟦ - ⟧ = give difference (the integer#1, the integer#2) .

(6) apply-operator ⟦ * ⟧ = give product (the integer#1, the integer#2) .

(7) apply-operator ⟦ / ⟧ = give quotient (the integer#1, the integer#2) .

(8) apply-operator ⟦ = ⟧ = give (the value#1 is the value#2) .

(9) apply-operator ⟦ < ⟧ = give is-less-than (the integer#1, the integer#2) .

5 Lexical Syntax

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

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` .

5.1 Identifiers

- token-of ⟦ _ ⟧ :: Identifier → token .

(1) token-of ⟦ I ⟧ = I .

5.2 Numerals

- valuation-of ⟦ _ ⟧ :: Numeral → integer .

(1) valuation-of ⟦ 0 ⟧ = 0 .

(2) valuation-of ⟦ 1 ⟧ = 1 .

(3) valuation-of ⟦ 2 ⟧ = 2 .

(4) valuation-of ⟦ 3 ⟧ = 3 .

(5) valuation-of ⟦ 4 ⟧ = 4 .

(6) valuation-of ⟦ 5 ⟧ = 5 .

(7) valuation-of ⟦ 6 ⟧ = 6 .

(8) valuation-of ⟦ 7 ⟧ = 7 .

(9) valuation-of ⟦ 8 ⟧ = 8 .

(10) valuation-of ⟦ 9 ⟧ = 9 .

(11) valuation-of ⟦ N D ⟧ = sum (product (valuation-of ⟦ N ⟧, 10), valuation-of ⟦ D ⟧) .