Monte Carlo Simulation Language

Reference Manual

Diego Garcia (dg2275)
Eita Shuto (es2908)
Yunling Wang (yw2291)
Chong Zhai (cz2191)

1. Introduction 5
Overview 5
Goal 5
Sub-algorithms: 5
Key feature 6
Hybrid Style 6
2. Lexical Conventions 7
Comments 7
Identifiers 7
Keywords 7
Constants 8
Integer constants 8
Floating constants 8
Character constants 8
Strings 8
3. Conversions 9
Floats and integers 9
Random type resolution 9
4. Expressions 10
Objects and lvalues 10
Primary expressions 10
Identifiers 10
Constants 10
Strings 10
Parentheses 10
Functions 10
Arrays 10
Unary Logical Operator 11
! operator 11
Arithmetic Operators 11
* operator 11
_ operator 11
operator 11
% operator 11

+ operator 11
- operator 11
Comparison operators 12
< operator 12
> operator 12
<= operator 12
>= operator 12
== operator 12
!= operator 12
Logical operators 12
& operator 12
| operator 13
5. Declaration 14
Variables 14
Type specifier 14
Declarator-list 14
Declarator 14
functions 14
6. Statements 16
Expression statement 16
Statement sequence 16
Declaration statement 16
Branching statement 16
Scoping statement 17
Listing statement 17
Looping statement 17
7. Scope 18
General Rules 18
Scope Classifications 18
File Scope 18
Function Scope 18
Assistant Scope 18
Local Scope 19
8. Compiler Control Lines 20

File inclusion.......... 20
9. Examples.......... 21
PI calculation.......... 21
Pollard Monte Carlo factorization method.......... 21
10. Appendix.......... 23
Mc.aggregate (func, input, time).......... 23
Mc.list (func, input, time) 23

1. Introduction

Overview

We are studying O'Caml when design this general purpose simulation language. The language aim to simplify the simulation programming with Monte Carlo method, free the programmers to the programming details about the simulation and focus on the model of particular problems. The discussion on generality provided the theoretical base for the feasibility of this idea.

Varieties and derivatives are introduced. For example Quasi-Monte Carlo method, known as the Halton-Hammersley-Wozniakowski algorithm, uses quasirandom numbers-also called low discrepancy sequences. And it has much faster speed on the evaluation of numerical integrations. It was implemented in Mathematica as NIntegrate[f, ..., Method ->QuasiMonteCarlo?]. Matlab uses this algorithm when calculate t cumulative distribution function for four or more dimensions: mvtcdf uses a quasi-Monte Carlo integration algorithm based on methods developed by Genz and Bretz. In 1992 a research group in the computer Science Department at Columbia University started testing QMC, using improved low discrepancy sequences (LDS), on a 360 dimensional CMO provided by Goldman Sachs. To our surprise QMC always beat MC. Their research turned into a patent for an estimation method and system for complex securities using low-discrepancy deterministic sequences.

Goal

Since we are not experts on Monte Carlo theory which becomes more and more subtle and is still under development. Our goal is not to compare or test the result for different algorithms. Beside, we are going to provide a language which simplifies the process of generate random numbers or low discrepancy sequences, aggregation the simulation results and keeps the track of convergences or variational conditions. Due to the scale of this project, there would been no GUI or any graphics statistics tools. Importing and exporting will be supported, so data could be visualized in other mathematical softwares.

Sub-algorithms:

- Generation of random numbers
- Uniform distribution:

  Mersenne twister: It is designed with Monte Carlo simulations and other statistical simulations in mind. Researchers primarily want good quality numbers but also benefit from its speed and portability. Advantages: It was designed to have a period of $2^{19937} - 1$ (the creators of the algorithm proved this property). In practice, there is little reason to use larger ones, as most applications do not require 2^{19937} unique combinations (2^{19937} is approximately 4.3×10^{6001}). It has a very high order of dimensional equidistribution. It passes numerous stringent tests for statistical randomness.

- Arbitrary distribution:

  Most distribution could be generated by using Uniform[0,1] random numbers. Algorithms is distribution depended, inverse transformation, acceptance-rejection method, composition method and etc.

- Generation of low discrepancy sequences

    Sobol' type, Van der Corput Sequence, Halton Sequence and Faure Sequence.

Key feature

Most calculations are based on random or quasi-random numbers. So we introduce rand as a built-in type for our language, the only thing programmer has to do is to specify the algorithm to be used and the type of distribution.

Hybrid Style

Most parts of the language follows the style of C-language, such as naming of keywords, comments and functional structures. But we also introduced some features from O'Caml which we believe are convenient and elegant, such as List.map, List.fold_left and List.iter.

2. Lexical Conventions

There are six kinds of tokens: identifiers, keywords, constants, strings, expression operators, and other separators. In general blanks, tabs, newlines, and comments as described below are ignored except as they serve to separate tokens. At least one of these characters is required to separate otherwise adjacent identifiers, constants, and certain operator-pairs.

If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters which could possibly constitute a token.

Comments

The characters `/*` introduce a comment, which terminates with the characters `*/`.

Identifiers

An identifier is a sequence of letters and digits; the first character must be alphabetic. The underscore “” *counts as alphabetic. Upper and lower case letters are considered different.*

Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

```
int
float
char
string
bool
list
array
vector
if
else
for
rand
return
continue
break
```

Constants

There are several kinds of constants, as follows:

Integer constants

An integer constant is a sequence of digits.

Floating constants

A floating constant consists of an integer part, a decimal point, a fraction part, an e, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (not both) may be missing; either the decimal point or the e and the exponent (not both) may be missing. Every floating constant is taken to be double-precision. In this language, some mathematical floating constants are referred by their conventional names in capital case, such as: PI, E. Due to the frequency of their usage, it's supported by the language, not math library.

Character constants

A character constant is 1 or 2 characters enclosed in single quotes " ′". Within a character constant a single quote must be preceded by a back-slash "\". The language supports basically alphabetic characters. Certain non-graphic characters, and "\" itself, may be escaped according to the following table:

```
BS ¥b NL ¥n CR ¥r HT ¥t ¥ ¥¥
```

The escape "\ddd" consists of the backslash followed by 1, 2, or 3 octal digits which are taken to specify the value of the desired character. A special case of this construction is "\0" (not followed by a digit) which indicates a null character.

Strings

A string is a sequence of characters surrounded by double quotes " "". A string has the type array-of-characters (see below) and refers to an area of storage initialized with the given characters. The compiler places a null byte (\0) at the end of each string so that programs which scan the string can find its end. In a string, the character " "" must be preceded by a "\" ; in addition, the same escapes as described for character constants may be used.

3. Conversions

In different circumstances, an expression of one type can be used to fulfill the role of a different type. This section lists the type conversions that can be used implicitly

Floats and integers

All integers may be converted without loss of significance to float. Conversion of float to integer takes place with truncation towards 0. Erroneous results can be expected if the magnitude of the result exceeds 2,147,483,647.

Random type resolution

Whenever a randomFloat or a randomInt is used where a float or integer is expected, the variable's value is resolved.

4. Expressions

An expression is a sequence of operators and operands. The precedence of expression operators is the same as the order of this section (highest precedence first).

Objects and lvalues

Objects are a manipulable region of memory, and lvalues are expressions referring to objects. In other word, lvalue can be placed at the left side hand of an assignment statement. Each object has its type.

Primary expressions

Identifiers

An identifier is name of variables and functions. It begins with any alphabet or underscore and is any combination of alphabet, digit & underscore. A variable identifier is an lvalue expression.

Constants

A decimals (integer) is a primary expression for an *int* type, and a floating constant is one for a *float* type. A constants is not an lvalue expression.

Strings

A string is a primary expression for a *string* type. A string is not an lvalue expression.

Parentheses

A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression. The presence of parentheses does not affect whether the expression is an lvalue

```
"(" expression ")"
```

Functions

A function call is a primary expression followed by a list of expression in parentheses. This list is called argument and can be empty. All arguments is passed by value. Even if function changes the values of argument, these changes do not affect the value of actual parameters. A function may return value.

```
lvalue := expression "(" expression expression ... expression ")"
```

Arrays

An index of array is specified by integer constants and arithmetic operators. Arithmetic operators are defined later. It is called a constant expression.

```
lvalue := expression ".[" constant_expression "]"
```

Unary Logical Operator

! operator

This operator is an unary operator and the result is a negation of the operand.

```
lvalue := '!' expression
```

Arithmetic Operators

Arithmetic operators can be used with *int* and *float* type. All operand of an operator must be same type but an auto conversion can be applied. In principle, a type of the result is same as operands.

∗ operator

The result is the multiplication of the expressions. This operator can be used with *vector* type. In this case, The result is the cross product of the vectors.

```
lvalue := expression '∗' expression
```

_ operator

The operator is used only with a *vector* type. The result is the scalar product and a *float* type. The number of dimension of operand should be same.

```
lvalue := expression '_' expression
```

/ operator

The result is the quotient of the division of the expressions.

```
lvalue := expression '/' expression
```

% operator

The result is the reminder of the division of the expressions.

```
lvalue := expression '/' expression
```

+ operator

The result is the sum of the expressions. This operator can be used with a *vector* type, sequence form and a *string* type. In case of a sequence form and a *string* type, the result is the combination of operands.

```
lvalue := expression '+' expression
```

- operator

The result is the difference of the expressions. This operator can be applied to a *vector* type.

```
lvalue := expression '-' expression
```

Comparison operators

The result of comparison operators is a *boolean* type object. The restriction of operands are same as arithmetic operators.

< operator

When the value of the left hand operand is less than one of right hand operand, this operator returns *true*. Otherwise, it returns *false*.

```
lvalue := expression '<' expression
```

> operator

When the value of the left hand operand is greater than one of right hand operand, this operator returns *true*. Otherwise, it returns *false*.

```
lvalue := expression '>' expression
```

<= operator

When the value of the left hand operand is less than or equal to one of right hand operand, this operator returns *false*. Otherwise, it returns *false*.

```
lvalue := expression '<=' expression
```

>= operator

When the value of the left hand operand is greater than or equal to one of right hand operand, this operator returns *false*. Otherwise, it returns *false*.

```
lvalue := expression '>=' expression
```

== operator

When the operands has same value, this operator returns *true*. Otherwise, it returns *false*.

```
lvalue := expression '==' expression
```

!= operator

When the operands has different value, this operator returns *true*. Otherwise, it returns *false*.

```
lvalue := expression '!=' expression
```

Logical operators

Operands of logical operators must be a *boolean* type. The result is also a *boolean* type.

& operator

The result is a conjunction of the operands.

```
lvalue := expression '&' expression
```

| *operator*

The result is a disjunction of the operands.

```
        lvalue := expression '|' expression
```

5. Declaration

Declarations are used to specify the interpretation which MCSL gives to each identifier; The declarations of variables and functions are treated differently.

Variables

All variables should be explicitly declared as below:

```
type-specifier declarator-list;
```

The *type-specifier* specified the datatype of the variables in the declarator-list(see section *DataType_?)*.

The *declarator-list* specified a list of declarators as explained below.

Type specifier

The type specifiers are:

```
type specifier:
        int
        char
        float
        boolean
        string
        vector
        list
        array
        randomint
        randomfloat
```

If the typespecifier is missing from a declaration, it is generally taken to be float.

Declarator-list

The declarator-list is a list of declarators with following format:

```
declarator-list:
       declarator, declarator-list
       declarator
```

Declarator

The declarators are names of the variables that are declared.

functions

The declarations of functions have the form

```
type func function-name (parameter-list) := statement;;
```

The *type* is the return type of the function. The *func* is a keyword indicating what follows is a function declaration. The *function-name* is the name of the function. The *parameter-list* is a list of parameters for the function. They are seperated with comma, and enclosed by "(" and ")".

```
parameter-list:
        type1 parameter1, type2 parameter2, ... typeN parameterN
```

statement is defined in the section *Statement*.

6. Statements

Statements are executed sequentially in the order they are given. For statements with sub-statements, the statement's type and value is the same as the sub-statement's.

Expression statement

The most basic statement is an expression statement:

    expression

This statement has the same type and value as the expression.

Statement sequence

To write a sequence of statements, which are executed left to right, they must be separated by the `;;` token:

    statement-sequence:
        statement `;;` *statement-sequence*
        statement

This statement has the same type and value as the expression.

Declaration statement

Identifier declaration is done through the following statement:

type-specifier identifier parameter-declaration$_{opt}$ `:=` *statement*

Where

    parameter-declaration:
        `(` *parameter-list*$_{opt}$ `)`

And

    parameter-list:
        type-specifier identifier parameter-list
        type-specifier identifier

This form declares identifier as a variable or function. For variables, *parameter-declaration* is omitted. For functions, *parameter-declaration* is mandatory. More on declarations can be read at the ImplicitDeclaration section.

Branching statement

This statement has the following form:

    `if` *expression* `then` *statement else-list* `endif`

Where

    else-list:
        `else` *statement*
        `elseif` *expression* `then` *statement else-list*

In all forms, the expressions must be of type boolean, and all sub-statements must be of the same type. The first expression is evaluated, and if true, the following statement is executed. If false, the process is repeated with the next elseif expression/statement (if any). If all expressions evaluate false, the else sub-statement is executed.

Scoping statement

Scoping statements are used to limit the scope of declarations.

```
with statement-sequence do statement done
```

Any declarations in the expression will only be valid for the sub-statement.

Listing statement

This statement is used to process list elements.

```
for identifier in list-identifier do statement done
```

For each element in the list, the identifier is assigned the element and then the statement is executed. This is done sequentially in the same order the elements are found in. The identifier will have the same type than the list's elements. Also, the identifier's scope is limited to the sub-statement.

Looping statement

Looping statements are used to iterate other statements.

```
loop statement while expression done
```

The expression must be of boolean type. The sub-statement is executed, and then expression will be evaluated. This process will repeat as long as the expression is true.

7. Scope

There are basically two kinds of scopes: lexical scope and scope of externals. Lexical scope is the region of a program during which it may be used without drawing "undefined identifier" diagnostics; the scope of externals is the region within which the the same external identifiers are referenced to the same object. The rules talked here are generally lexical scope rules, as our language assume the preprocessor will include the external source files into the application program where the external identifiers are used, thus transferred all the external identifiers into local ones.

General Rules

Generally, for the different scopes in our language, the variables in the outer scopes are always visible in the inner scopes, while those in inner scopes are invisible in outer scopes. There are circumstances, however, that variables in one scope are visible in another scope that is not overlapped with it. The next section provides detailed explanations for this situation.

Scope Classifications

Below is the typical scope illustration for our language.

```
file-starts-here
    func a(...)={
            _B_
    }
    with{
            _C_
    }
    do{
            _D_
    }
            _A_
file-ends-here
```

There are 4 kinds of different scopes in the illustration above. *A* is called file scope; *B* is called function scope; *C* is called assistant scope; *D* is called local scope;

File Scope

The file-scope variables are so called "global variables". The lexical scope of these kinds of variables is the entire file, which means the variables are visible in all the functions within this file.

Function Scope

This scope is enclosed by the "{" and "}" after the function declaration. The function-scope variables refer to the variables that are only visible in one function, that is, the function within which they are used.

Assistant Scope

This scope is enclosed by the keywords *with* and *do*. The variables within this scope are visble in both this assistant scope and the local scope (see *local scope* that is matched to the current assistant scope.

Local Scope

This scope is enclosed by the "{" and "}" after the keyword *do*. Variables within a certain local scope are only visible within this region.

8. Compiler Control Lines

When a line begins with the character #, it is interpreted not by the compiler itself, but by a preprocessor which is capable of inserting named files into the source program. In order to cause this preprocessor to be invoked, it is necessary that the very first line of the program begin with #. Since null lines are ignored by the preprocessor, this line need contain no other information.

File inclusion

A compiler control line of the form

```
        # include "filename"
```

results in the replacement of that line by the entire contents of the file *filename*.

9. Examples

PI calculation

This approximates pi by getting random points in a square, and calculating how many fall into an inscribed circle.

```
func inCircle (randFloat x, randFloat y) :=
   with
      vector v := <<x, y>>
   do
      if v.size <= 1
      then 1
      else 0
      endif
   done
;;
randFloat domain := Rand.float(0, 1) ;;
func begin(int iterations) :=
   (MC.aggregate (inCircle, (domain, domain), iterations)) / iterations
;;
```

Pollard Monte Carlo factorization method

```
int func rec gcd(int x, int y):=
if  x==0
then   y
else  if y==0
then  x
else
        gcd(x, y%x)
endif
;;


int func fact(int N, int b) :=
with
         k := Math.factorial(b) ;;
          randInt a := Rand.int(2, N-2)
 do
         loop
                f := gcd(mod(a^k-1,N),N)
         while
                 f==1
         done
done
;;

int * int func begin (int N, int b) :=
with
          int f := fact(N, b)
 do
         (f, N/f)
done
;;
```

10. Appendix

We have a series of Library functions called *Monte-Carlo functions* that are used exclusively for Monte-Carlo simulations. All of them are actually loop statements (See the section *Statement*).

Mc.aggregate (func, input, time)

This function performs simulation for *time* times. It takes one element of *input* each time, apply that to the simulation function *func*, and add the simulation result to the final return value.

Mc.list (func, input, time)

This function is similar to *Mc.aggregate*, except that instead of returning the aggregation of all the simulation result, it returns a list with all the simulation results