The C(M) programming language

Stoyan Mihov

April 7, 2016

1 Overview

C(M) is an efficient and powerful programming language, which directly translates mathematical constructions into efficient C programs. It has efficient very high-level structures and expressions, which enable the rapid development of complex algorithms and applications.

The C(M) compiler is freely available in executable form for the major platforms from the C(M) site, `http://lml.bas.bg/~stoyan/lmd/C%28M%29.html`. The compiler translates the C(M) program into a readable C program with the same names of identifiers and similar structure to the original. The C program can be further modified, extended or embedded into other programs.

This paper introduces informally the basic concepts and features of the C(M) language. It does not attempt to be comprehensive. Instead, it introduces many of C(M) 's most noteworthy features, which give a good idea of the languages flavour and style. The appendix provides the formal syntax description.

2 Concept of C(M)

The main idea behind the C(M) language is the usage of the standard mathematical language for the description of algorithms and the construction of data structures. It implements as data structures mathematical objects like tuples, sets, lists, functions, relations and matrices.

The main features of the language are:

- Declarative programming style – only one assignment of an expression to a variable is allowed;
- Strong type checking – the type of each identifier has to be defined at compile time and there is no universal type;
- High-level functional programming – functions can be used as parameters and returned as result, Currying of functions is supported;
- High-level expressions – set builder notation, quantification, function lifting and others are supported;
- Construction of structures by induction – with the notion of mathematical induction we obtain efficient implementations without sacrificing the declarative nature of the language.
- Optimal (in some sense) memory management – the memory allocated by the objects is either reused or freed authomatically after their last usage without performing garbage collection.

C(M) can be regarded as a strictly typed declarative functional language. In addition to the common functional languages C(M) implements mathematical syntax, set-theoretic high-level structures and expressions and construction by induction. The high-level structures and expressions have similarities with the ones in the SETL language but C(M) differs by its declarative nature and strong type checking.

3 Example

1 $\mathcal{R}$ **is** $2^{\mathbb{N}\times\mathbb{N}}$;
2 $\text{closure} : \mathcal{R} \to \mathcal{R}$;
3 $\text{closure}(A) := T$, **where**
4     $T :=$ **induction**
5   **step** $0:$
6             $T^{(0)} := A$;
7   **step** $n+1:$
8             $T^{(n+1)} := T^{(n)} \cup \{(a,c) \mid (a,b) \in T^{(n)}, (b,c) \in A\}$;
9   **until** $\forall (a,b) \in T^{(n)}, (b,c) \in A : ((a,c) \in T^{(n)})$
10   ;
11   ;
12 $\text{dump} \leftarrow \text{closure}(\{(1,2),(2,3),(3,5),(5,10)\})$;

```
1  R is 2^(IN*IN);
2  closure in R -> R;
3  closure(A) := T, where
4     T := induction
5        step 0:
6           T@0 := A;
7        step n+1:
8           T@n+1 := T@n \/ {(a,c) | (a,b) in T@n, (b,c) in A};
9        until forall (a,b) in T@n, (b,c) in A : ((a,c) in T@n)
10       ;
11    ;
12 dump <- closure({(1,2),(2,3),(3,5),(5,10)});
```

Perhaps the best introduction to C(M) is a short example. The following is a complete C(M) program to construct the transitive closure of a finite relation of natural numbers. At the end the closure of a concrete relation is dumped.

The program above is listed twice – first using the mathematical layout, and second, as plain text. The compiler supports an automatic translation from the plain text into LaTeX for producing the mathematical layout.

This program starts with the naming of the type $2^{\mathbb{N}\times\mathbb{N}}$ as $\mathcal{R}$ in Line 1. The largest part of this program is the "closure" function definition in lines 3–11. As stated in Line 2 *closure* is of type $\mathcal{R} \to \mathcal{R}$ i.e. a function which takes a parameter of type $\mathcal{R}$ and returns result of type $\mathcal{R}$. In Line 3 the parameter is named A and the result returned is named T. T is defined in the **where** block in lines 4–11. C(M) uses the semicolon in line 11 to recognise the **where** block end. T is defined by an **induction** statement in lines 4–10. The semicolon in Line 10 marks the end of the **induction** statement. The base – **step** 0 of the induction sets the base of T to A in Line 6. The inductive step defines the $(n+1)^{\text{th}}$ value of T as $T^{(n)} \cup \left\{(a,c) \;\middle|\; (a,b) \in T^{(n)}, (b,c) \in A\right\}$ in Line 8. In this assignment $T^{(n+1)}$ becomes the union of $T^{(n)}$ with the set of all (a,c), where (a,b) runs through $T^{(n)}$ and (b,c) runs through A. The inductive step is performed until the condition in Line 9 is satisfied. This condition states that for every $(a,b) \in T^{(n)}$ and $(b,c) \in A$ it holds that $(a,c) \in T^{(n)}$. This means that the inductive step will not extend $T^{(n)}$ anymore. Line 12 dumps the closure of $\{(1,2),(2,3),(3,5),(5,10)\}$ to the output.

The compiler takes as input the plain text of the program (without the line numbering) and outputs a C source code. Let the above program is presented in a file named `example.cm`. Then the compiler is invoked by:

```
cm example.cm -o example.c
```

The file `example.c` will contain the corresponding C source code. Afterwards the C code has to be compiled with the `gcc` compiler for producing an executable. Currently the C code generated by C(M) contains nested functions, which are supported by `gcc` but are not ANSI C compatible. The compilation is invoked by the following command:

```
gcc -fnested-functions -lm -o example example.c
```

In the newer versions (e.g 4.7) of `gcc` the option `-fnested-functions` has to be omitted. If no floating point functions are used then the `-lm` option is not required.

The LaTeX layout is generated by the compiler in the following way:

```
cm -L example.cm -o example.tex
```

Afterwards the example.tex file has to be compiled with LaTeX.

4 Types

C(M) supports the following basic types: `IN` ($\mathbb{N}$) for natural numbers, `IZ` ($\mathbb{Z}$) for integer numbers, `IR` ($\mathbb{R}$) for real numbers and `IB` ($\mathbb{B}$) for boolean values. Those types are implemented in C as `unsigned long`, `long`, `double` and `unsigned long` correspondingly. The type for matrix of real numbers is `M(IR)` ($\mathcal{M}(\mathbb{R})$).

C(M) supports the following complex types:

- Tuples: if `T_1,T_2,...,T_n` ($T_1, T_2, \ldots, T_n$) are types then `T_1 * T_2 * ... * T_n` ($T_1 \times T_2 \times \ldots \times T_n$) is the type of the n-tuples, whose i-th projection is of type `T_i` (T_i).
- Lists: if T is a type then `T^*` (T^*) is the type of the lists with elements of T.
- Sets: if T is a type then `2^T` (2^T) is the type of the sets with elements of T.
- Functions: if `T_1` (T_1) and `T_2` (T_2) are types then `T_1 -> T_2` ($T_1 \to T_2$) is the type of functions with domain T_1 and range T_2.

Parentheses have to be used for type grouping. For example the type $A \times B \times C$ indicates the type of triples whose first, second and third projections are of types A, B and C correspondingly. $(A \times B) \times C$ indicates the type of pairs whose first projection is a pair of the types A and B and its second projection is of type C.

The type `STRING` is predefined as $\mathbb{N}^*$.

5 Identifiers

The identifiers in C(M) have to start with a letter, can contain digits and can end with apostrophes. The identifiers can also have indices. Some examples of identifiers are given below:

`X`	X
`X1s''`	$X1s''$
`X'_3`	X'_3
`A'_p'_3`	$A'_{p'_3}$

Inside an induction statement the inductive identifiers are followed by the index in the series in parentheses:

`A@0`	$A^{(0)}$
`A'_2@n`	$A_2'^{(n)}$
`A'@n_c+1`	$A'^{(n_c+1)}$

6 Constants

The following types of constants are supported in C(M) :

Booleans	$true$, $false$
Natural numbers	3, 2014
Real numbers	−25.349, 2.1234E−23
Strings	"Example", "This is a sentence."
The empty set	{} ($\emptyset$)
The null list	[] (ε)

The constants are the simplest expressions.

7 Terms

C(M) allows the grouping of identifiers into tuples for supporting of parallel assignments or multiple definitions. Tuples of identifiers can be recursively grouped into terms. Examples of terms are:

A
(a, b)
$(a, ((b, c', d), f_1))$

Terms can be used for running arguments in set builder and quantifier expressions (see next section). In those cases the term can contain constants as well for constraining the running arguments. For example if S is a set of triples then in a set builder $(a, 0, X') \in S$ will run through the triples of S with second projection equal to 0.

8 Expressions

Constants and identifiers which are already defined are the simplest expressions.

Tuples

`(a,b,c,d)`	(a, b, c, d)	tuple construction
`Proj(2,T)`	$Proj_2(T)$	tuple projection
`Proj((2,4),T)`	$Proj_{(2,4)}(T)$	tuple projection to subtuple

Comparisons

`E_1=E_2`	$E_1 = E_2$	equal
`E_1~=E_2`	$E_1 \neq E_2$	not equal
`E_1<E_2`	$E_1 < E_2$	lower (for numbers only)
`E_1<=E_2`	$E_1 \leq E_2$	lower or equal (for numbers only)
`E_1>E_2`	$E_1 > E_2$	greater (for numbers only)
`E_1>=E_2`	$E_1 \geq E_2$	greater or equal (for numbers only)

Sets

`|S|`	$\|S\|$	number of elements in the set S
`{a,b,c,d}`	$\{a, b, c, d\}$	set of items
`{n_1..n_2}`	$\{n_1, \ldots, n_2\}$	set of the numbers from n_1 to n_2
`S_1/\S_2`	$S_1 \cap S_2$	intersection
`S_1\/S_2`	$S_1 \cup S_2$	union
`S_1\S_2`	$S_1 \setminus S_2$	difference
`S_1 subset S_2`	$S_1 \subset S_2$	subset
`S_1 ~subset S_2`	$S_1 \not\subset S_2$	non-subset
`S_1 meets S_2`	$S_1 \cap S_2 \neq \emptyset$	meets
`S_1 ~meets S_2`	$S_1 \cap S_2 = \emptyset$	meets not
`a in S`	$a \in S$	membership of a in S
`a ~in S`	$a \notin S$	non-membership of a in S
`union(S)`	$\bigcup(S)$	union of the element in the sets in S
`2^S`	2^S	the set of all subsets of S
`S_1*S_2*...*S_k`	$S_1 \times S_2 \times \ldots \times S_k$	cartesian product
`{E | t_1 in S_1 & C_1, t_2=E_2 & C_2, ...,t_k in S_k & C_k}`	$\{E \mid t_1 \in S_1 \ \& \ C_1, t_2 = E_2 \ \& \ C_2, \ldots, t_k \in S_k \ \& \ C_k\}$	set builder, where E is an expression, t_i are terms, S_i and resp. E_i are sets or lists and resp. expressions, and C_i are optional conditional expressions.

Relation (set of tuples)

All set expressions plus the following:

Proj(2,R)	$Proj_2(R)$	Relation projection
Proj((2,4),R)	$Proj_{(2,4)}(R)$	Relation projection to subrelation
Func(1,2,R)	$\mathcal{F}_{1\to 2}(R)$	functionalization of R

Function

Functions are sets of pairs, which have unique mapping. All expressions on sets and relations are applicable plus the following:

!f(x)	$!f(x)$	true if $f(x)$ is defined, false otherwise
f|_E	$f\|_E$	restriction of f to the domain given by E

Lists

|L|	$\|L\|$	length of the list L
[A,B,C,D]	$\langle A, B, C, D\rangle$	list of items
[n_1..n_2]	$\langle n_1, \ldots, n_2\rangle$	list of the numbers from n_1 to n_2
L_1.L_2	$L_1 \cdot L_2$	concatenation of lists
L[i]	$(L)_i$	i-th element of a list
L[i..j]	$(L)_{i,\ldots,j}$	sublist from i-th to j-th element
#(a,L)	$\#_L(a)$	the index of a in L
a in L	$a \in L$	membership of a in L
a ~in L	$a \notin L$	non-membership of a in L
flatten(L)	$\bigodot(L)$	list of the elements of the lists in L
[E | t_1 in S_1 & C_1, t_2=E_2 & C_2, ...,t_k in S_k & C_k]	$\langle E \mid t_1 \in S_1 \;\&\; C_1, t_2 = E_2 \;\&\; C_2, \ldots, t_k \in S_k \;\&\; C_k\rangle$	list builder, where E is an expression, t_i are terms, S_i and resp. E_i are sets or lists and resp. expressions, and C_i are optional conditional expressions.

Lists and sets of numbers

min(L)	$\min(L)$	minimal element in the list/set
max(L)	$\max(L)$	maximal element in the list/set
sum(L)	$\sum(L)$	sum of the elements in the list/set
prod(L)	$\prod(L)$	product of the elements in the list/set

Boolean operators

~E	$\neg E$	negation
E_1/\E_2	$E_1 \wedge E_2$	conjunction
E_1\/E_2	$E_1 \vee E_2$	disjunction
E_1->E_2	$E_1 \to E_2$	implication
E_1<->E_2	$E_1 \leftrightarrow E_2$	equivalence
forall t_1 in S_1,...,t_k in S_k:(E)	$\forall t_1 \in S_1, \ldots, t_k \in S_k : (E)$	universal quantifier
exists t_1 in S_1,...,t_k in S_k:(E)	$\exists t_1 \in S_1, \ldots, t_k \in S_k : (E)$	existential quantifier

Expressions on natural, integer and real numbers

|E|	$\|E\|$	absolute value
-E	$-E$	negative value
+E	$+E$	positive value
E_1^E_2	$E_1^{E_2}$	power
E_1*E_2	$E_1 \times E_2$	multiplication
E_1/E_2	E_1/E_2	devision
E_1+E_2	$E_1 + E_2$	addition
E_1-E_2	$E_1 - E_2$	substraction
E_1 rem E_2	$E_1 \operatorname{rem} E_2$	reminder

Expressions on matrices

-M	$-M$	negative value
+M	$+M$	positive value
M`	M^T	transposition
M_1^E_2	$M_1^{E_2}$	power (not implemented yet)
M_1*M_2	$M_1 \times M_2$	multiplication
M_1/M_2	M_1/M_2	devision (not implemented yet)
M_1\M_2	$M_1 \setminus M_2$	left devision (not implemented yet)
M_1+M_2	$M_1 + M_2$	addition
M_1-M_2	$M_1 - M_2$	substraction
M_1.^E_2	$M_1 \cdot \wedge M_2$	dot power
M_1.*M_2	$M_1 \cdot \times M_2$	dot multiplication
M_1./M_2	$M_1 \cdot / M_2$	dot devision
M_1.M_2	$M_1 \cdot M_2$	horizontal concatenation of matrices
M_1\\M_2	$\begin{matrix} M_1 \\ M_2 \end{matrix}$	vertical concatenation of matrices
[:A,B,C,D:]	$[A, B, C, D]$	1-row matrix
M[i,j]	$M_{i,j}$	matrix element
M[L_1,L_2]	M_{L_1,L_2}	submatrix with indices $L_1 \times L_2$
[:E|i=1..n:]	$[E \mid i = 1, \ldots, n]$	1-row matrix builder
[:E|i=1..n,j=1..m:]	$[E \mid i = 1, \ldots, n, j = 1, \ldots, m]$	matrix builder

Conditional expressions

```
?  E_1 if C_1
?  E_2 if C_2
⋮
?  E_k if C_k
?  E_k+1 otherwise
```

$$\begin{cases} E_1 & \textbf{if } C_1 \\ E_2 & \textbf{if } C_2 \\ \vdots & \\ E_k & \textbf{if } C_k \\ E_{k+1} & \textbf{otherwise} \end{cases}$$

E_1 if C_1
E_2 if C_2
E_k if C_k
E_{k+1} otherwise

Build-in functions, constants and objects

Function	Type	Description
$\mathrm{true}, \mathrm{false}$	$\mathbb{B}$	boolean constants
$\mathrm{set}(L)$	$T^* \to 2^T$	the set of the elements in the list L
$\mathrm{elementOf}(S)$	$2^T \to T$	returns one element of S
$\mathrm{argmin}(f, L), \mathrm{argmax}(f, L)$	$(T \to \mathbb{N}) \times T^* \to T$ $(T \to \mathbb{Z}) \times T^* \to T$ $(T \to \mathbb{R}) \times T^* \to T$ $(T \to \mathbb{N}) \times 2^T \to T$ $(T \to \mathbb{Z}) \times 2^T \to T$ $(T \to \mathbb{R}) \times 2^T \to T$	returns a value in L for which the function f is minimal/maximal
$\mathrm{subst}(L, i, v)$	$T^* \times \mathbb{N} \times T \to T^*$	substitution of the i-th element in L with v
$\mathrm{substl}(L, I, V)$	$T^* \times \mathbb{N}^* \times T^* \to T^*$	substitution of the elements with indices I in L with V
$\mathrm{rows}(M), \mathrm{cols}(M)$	$\mathcal{M}(\mathbb{R}) \to \mathbb{N}$	number of rows/columns of M
$\mathrm{substMat}(M, i, j, v)$	$\mathcal{M}(\mathbb{R}) \times \mathbb{N} \times \mathbb{N} \times \mathbb{R} \to \mathcal{M}(\mathbb{R})$	substitution the element at (i, j) in M with v
$\mathrm{substSubMatrix}(M, I, J, M')$	$\mathcal{M}(\mathbb{R}) \times \mathbb{N}^* \times \mathbb{N}^* \times \mathcal{M}(\mathbb{R}) \to \mathcal{M}(\mathbb{R})$	substitution the submatrix at indicies $I \times J$ in M with M'
$\mathrm{AND}(a, b), \mathrm{OR}(a, b), \mathrm{XOR}(a, b)$	$\mathbb{N} \times \mathbb{N} \to \mathbb{N}$	bitwise and/or/exclusive or of a and b
$\mathrm{NOT}(a)$	$\mathbb{N} \to \mathbb{N}$	bitwise negation of A
$\mathrm{SHL}(a, b), \mathrm{SHR}(a, b)$	$\mathbb{N} \times \mathbb{N} \to \mathbb{N}$	shift left/right of a with b bits
$\sin(x), \cos(x), \tan(x)$	$\mathbb{R} \to \mathbb{R}$	trigonometric functions
$\mathrm{asin}(x), \mathrm{acos}(x), \mathrm{atan}(x)$	$\mathbb{R} \to \mathbb{R}$	inverse trigonometric functions
$\sinh(x), \cosh(x), \tanh(x)$	$\mathbb{R} \to \mathbb{R}$	hyperbolic trigonometric functions
$\mathrm{asinh}(x), \mathrm{acosh}(x), \mathrm{atanh}(x)$	$\mathbb{R} \to \mathbb{R}$	inverse hyperbolic trigonometric functions
$\mathrm{sqrt}(x), \exp(x)$	$\mathbb{R} \to \mathbb{R}$	square root and exponent functions
$\log(x), \log 2(x), \log 10(x)$	$\mathbb{R} \to \mathbb{R}$	natural, base 2 and base 10 logarithm functions
$\mathrm{erf}(x), \mathrm{tgamma}(x), \mathrm{lgamma}(x)$	$\mathbb{R} \to \mathbb{R}$	error function, gamma and logarithm of gamma functions
$\mathrm{floor}(x), \mathrm{ceil}(x), \mathrm{trunc}(x)$	$\mathbb{R} \to \mathbb{R}$	floor, ceiling and truncate functions
$\mathrm{round}(x), \mathrm{rint}(x)$	$\mathbb{R} \to \mathbb{R}$	rounding functions
$\mathrm{lrint}(x)$	$\mathbb{R} \to \mathbb{Z}$	rounding to whole number
$\mathrm{str}(i)$	$\mathbb{N} \to STRING$	textual representation of i
$\mathrm{loadBin}(F)$	$STRING \to \mathbb{N}^*$	loads the binary file F and returns the list of its bytes
$\mathrm{loadText}(F)$	$STRING \to STRING$	loads the text file F and returns its UTF-8 symbols
$\mathrm{restore}(F)$	$STRING \to T$	restores an object stored in the file F
$\mathrm{restoreBin}(F)$	$STRING \to T$	restores an object stored in the binary file F
argc	$\mathbb{N}$	number of command line arguments passed to the program
$\mathrm{argv}(i)$	$\mathbb{N} \to STRING$	the i-th command line argument passed to the program
mainResult	$\mathbb{N}$	if assigned it will be the exit status of the program execution

9 Statements

Any program in C(M) is a list of statements. There are only 4 kinds of statements – type definitions, declarations, assignments and supplementary actions. Any statement must end with a semicolon.

9.1 Type definition

A type definition is used for naming a complex type. A type definition is in the form:

```
ID is TYPE;
```

For example the statement in Line 1 of our example is naming the type $2^{\mathbb{N}\times\mathbb{N}}$ to $\mathcal{R}$.

9.2 Declaration

The declaration statements are used for declaring the types of the used terms and identifiers. A declaration is required in the following two cases: when the term is defined by induction[1] or when a function is defined with an expression from its parameters. In case the type of a term can be inferred from the expression, which has been assigned to the term, the declaration is not obligatory. Though in some cases the type inferred from the expression might not be what is needed and a declaration has to correct it. The declaration statement is in the form:

$\texttt{TERM}_1$, $\texttt{TERM}_2$, ..., $\texttt{TERM}_n$ in TYPE;

The statement in Line 2 of the example declares the type of "closure" to $\mathcal{R} \to \mathcal{R}$.

9.3 Assignment

The assignments are the most commonly used statements in C(M) . They are used to calculate and assign a value to a term. The general form of an assignment statement is:

```
TERM := ASSIGNMENT;
```

If the term is a name of a function then the function can be defined on its parameters in the following way:

```
ID(PARAMETER_1,PARAMETER_2,...,PARAMETER_k) := ASSIGNMENT;
```

There are 4 types of assignments – simple assignment, assignment with where block, case assignment and inductive assignment.

9.3.1 Simple assignment

The simple assignment is an expression. for example the statements in Lines 6 and 8 in our example are simple assignments.

```
TERM := EXPRESSION;
```

9.3.2 Assignment with where block

The assignment statements with a where block are like the simple assignments but followed by a block of additional statements after the where keyword. Those additional statements are usually required for defining the objects used in the expression. The general form of such a assignment statement is:

```
TERM := EXPRESSION, where
   STATEMENT_1;
   STATEMENT_2;
   ...
   STATEMENT_k;
;
```

In our example an assignment statements with a where block is given in Lines 3–11.

[1]Unless its type is not implicitly known.

9.3.3 Case assignment

The case assignment consists of a list of pairs and expressions. The expression of the first fulfilled condition is assigned to the term. An otherwise expression is optional and will be assigned in case non of the conditions holds:

```
TERM :=
   case CONDITION_1 :  EXPRESSION_1
   case CONDITION_2 :  EXPRESSION_2
   ...
   case CONDITION_k :  EXPRESSION_k
   otherwise EXPRESSION_k+1
;
```

Each expression might be followed by a where block. The general form is:

```
TERM :=
   case CONDITION_1 :  EXPRESSION_1, where
      STATEMENTS
   case CONDITION_2 :  EXPRESSION_2, where
      STATEMENTS
   ...
```

9.3.4 Inductive assignment

The inductive assignment is used for inductive constructions of terms. The general form is:

```
TERM := induction
   step 0 :
      STATEMENTS
   step n+1 :
      STATEMENTS
   until CONDITION
;
```

The base of the induction is defined by the statements after the step 0 keyword. The inductive step is defined by the statements after the step n+1 keyword. The inductive steps are repeated until the condition holds. In Lines 4–10 of our example T is defined by an inductive statement.

9.4 Supplementary actions

Besides the definitions, declarations and assignments C(M) supports supplementary actions for mainly for outputting and dumping of resulting objects. The general form is:

```
ACTION <- EXPRESSION;
```

In line 12 of the example the closure of the given relation is dumped out. Currently the following actions are implemented:

`dump <- E;`	dumps the given expression to the standard output
`print <- E;`	the expression, which must be a STRING is printed on the standard output
`assert <- E;`	if the expression of type $\mathbb{B}$ does not hold the program halts
`store <- (F,E);`	the expression `E` will be stored in the file with name `F` (must be a STRING)
`storeBin <- (F,E);`	the expression `E` will be stored in binary format in the file with name `F` (must be a STRING)
`saveBin <- (F,L);`	the list of bytes `L` (must be $\mathbb{N}^*$) will be stored in the binary file with name `F` (must be a STRING)
`saveText <- (F,T);`	the text `T` (must be STRING) will be stored in the UTF-8 text file with name `F` (must be a STRING)
`import <- F;`	import the file with name F (must be a STRING constant) to the program (in case of multiple imports of the same file, the file will be imported only once)
`include <- F;`	include the source from file with name F (must be a STRING constant) to the program at the corresponding place. (Multiple inclusion of the same file are allowed.)

10 Comments

In the C(M) programs everything after `//` till the end of the line is regarded as a comment. If the comment is placed between two statements it will appear also in the LaTeX code. Thus the comments can contain LaTeX commands which will be considered in the LaTex compilation.

11 The example revisited

Looking at the example in Section 3 we can find two major sources of inefficiencies.

First, all the elements of $T^{(n)}$ are considered to be extended transitively with elements of A at each step, although it is clear that the elements considered on previous steps can not deliver new pairs. This can be avoided by considering at step $n+1$ only the $n+1$-st element of $T^{(n)}$. The second deficiency is that after choosing a pair (a,b) from T to be extended, we run through all pairs from A to find the ones with first coordinate b. Instead of that we can construct a function A', which for a given b returns the set of all c, such that $(b,c) \in A$. In the new implementation given below we define T as a pair of numbers in Line 4. A' is defined in Line 5. The type of A' will be $\mathbb{N} \to 2^{\mathbb{N}}$. In Lines 6–13 T is constructed inductively. The main inductive step is given in Line 11 where $T^{(n)}$ is extended with the extensions of the $n+1$-st pair in $T^{(n)}$. The induction ends when all elements of $T^{(n)}$ are processed.

1 $\mathcal{R}$ **is** $2^{\mathbb{N}\times\mathbb{N}}$;
2 $\text{closure} : \mathcal{R} \to \mathcal{R}$;
3 $\text{closure}(A) := \text{set}(T)$, **where**
4 $\quad T \in (\mathbb{N} \times \mathbb{N})^*$;
5 $\quad A' := \mathcal{F}_{1\to 2}(A)$;
6 $\quad T :=$ **induction**
7 $\quad\quad$ **step** 0 :
8 $\quad\quad\quad T^{(0)} := A$ **as** $(\mathbb{N} \times \mathbb{N})^*$;
9 $\quad\quad$ **step** $n+1$:
10 $\quad\quad\quad (a,b) := (T^{(n)})_{n+1}$;
11 $\quad\quad\quad T^{(n+1)} := T^{(n)} \cdot \begin{cases} \langle (a,c) \mid c \in \mathrm{A}'(b) \ \& \ (a,c) \notin T^{(n)} \rangle & \textbf{if } ! \mathrm{A}'(b) \\ \varepsilon & \textbf{otherwise} \end{cases}$;
12 $\quad\quad$ **until** $n = \left|T^{(n)}\right|$
13 $\quad\quad$;
14 $\quad$;
15 dump $\leftarrow$ $\text{closure}(\{(1,2),(2,3),(3,5),(5,10)\})$;

A Syntax definiton

The formal syntax is defined using the LLgen notation (`http://www.cs.vu.nl/~ceriel/LLgen.html`).

```
%token ID;                 /* identifier */
%token IDN;                /* inductive identifier */
%token STRING;             /* string constant */
%token NUM;                /* natural number constant */
%token RNUM;               /* real number constant */
%token restr;              /* |_ */
%token in;                 /* in */
%token subset;             /* subset */
%token meets;              /* meets */
%token cap;                /* /\ */
%token cup;                /* \/ */
%token le;                 /* <= */
%token ge;                 /* >= */
%token dd;                 /* .. */
%token forall;             /* forall */
```

```
%token exists;            /* exists */
%token if;                /* if */
%token otherwise;         /* otherwise */
%token where;             /* where */
%token eqv;               /* <-> */
%token until;             /* until */
%token induction;         /* induction */
%token step;              /* step */
%token is;                /* is */
%token def;               /* := */
%token arrow;             /* -> */
%token rem;               /* rem */
%token as;                /* as */
%token case;              /* case */
%token opmat;             /* [: */
%token clmat;             /* :] */
%token newrow;            /* \\ */
%token larrow;            /* <- */
%token proj;              /* Proj */
%token func;              /* Func */

%token defined;

%start setlang, STMTL;

UNOP : '+' | '-'  | '~' | '!' |  cup | '.' ;
MULOP : '/' | '\\' | restr | rem | '.' [ '*' | '/' | '^' ]? ;
ADDOP : '+' | '-' | cap | cup | arrow | eqv ;
EQOP : '=' | '<' | '>' | le | ge | in | subset | meets | '~' [ in | subset | meets | '='];
QUANTOR : forall | exists ;

TYPE : DTYPE [ arrow DTYPE] * ;
DTYPE : CTYPE [ '*' CTYPE ] * ;
CTYPE : BTYPE ['^' '*']  * ;
BTYPE : ID [ '(' ID ')' ] ? | '(' TYPE ')' | NUM '^' BTYPE ;

AEXPR :
        NUM | RNUM | STRING | ID | IDN |
        CASE |
        '|' EXPR '|' |
        '(' EXPRL ')' |
        '#' '(' EXPR ',' EXPR ')' |
        proj '(' EXPR ',' EXPR ')' |
        func '(' EXPR ',' EXPR ',' EXPR ')' |
        if '(' EXPR ',' EXPR ',' EXPR ')' |
        '{' [ EXPR [',' EXPRL | dd EXPR | '|' TERMEXPRL ] ? ] ? '}'|
        '[' [ EXPR [',' EXPRL | dd EXPR | '|' TERMEXPRL ] ? ] ? ']'|
        opmat EXPR [',' EXPRL | '|' [ ID '=' NUM dd EXPR ] +2 ] ? clmat |
        QUANTOR TERMEXPRL ':' '(' EXPR ')' ;


CASE : '?' EXPR [ if [defined | EXPR]  [CASE | otherwise] ] ;

POSOP :  '(' EXPR ')' |  '[' EXPR [ dd EXPR | ',' EXPR  ] ? ']' | '`' ;

POSEXPR : AEXPR POSOP * ;
```

```
UNEXPR  : UNOP * POSEXPR;
POWEXPR : UNEXPR [ '^' UNEXPR ] *;
TMSEXPR : POWEXPR [ '*' POWEXPR ] * ;
MULEXPR : TMSEXPR [ MULOP TMSEXPR ] * ;
ADDEXPR : MULEXPR [ ADDOP MULEXPR ] * ;
MATEXPR : ADDEXPR [ newrow ADDEXPR ] * ;
EQEXPR : MATEXPR [ EQOP MATEXPR ] * ;
EXPR : EQEXPR [ as TYPE ] ? ;

EXPRL : EXPR [ ',' EXPR ] * ;

TERM : ID | IDN | NUM | RNUM | STRING | '(' TERML ')' ;
TERML : TERM [ ',' TERM ] * ;

TERMEXPR : TERM [in | '='] EXPR ['&' EXPR ]? ;
TERMEXPRL : TERMEXPR [ ',' TERMEXPR ]*  ;

ASSGNMNT : EXPR [ ',' where STMTL ] ? |
        [ case EXPR ':' EXPR [ ',' where STMTL]? ] + [otherwise ':' EXPR [ ',' where STMTL]? ] ? |
        induction
                step NUM ':' STMTL
                step ID '+' NUM ':' STMTL
                until EXPR ;

STMT : TERML [  is TYPE  |
                in TYPE  |
                TERM ? def ASSGNMNT |
larrow EXPR
             ] ;

STMTL : [ STMT ';' ]+ ;
```