LFCS

Laboratory for Foundations of Computer Science
Department of Computer Science - The University of Edinburgh

A Language for value-passing CCS

by

Glenn Bruns

A Language for value-passing CCS

LFCS Report Series

ECS-LFCS-91-175

LFCS
Department of Computer Science
University of Edinburgh
The King's Buildings
Edinburgh EH9 3JZ

August 1991

A language for value-passing CCS

Glenn Bruns

August 23, 1991

Abstract

Milner has defined an extension to CCS, called *value-passing CCS*, which allows parameterized agents and actions, and conditional expressions. This notation leaves open the problems of describing data and of associating sets of values with actions. We define a language for value-passing CCS in which sets, sequences, and tuples are built from natural number, boolean, and string constants. Only finite values can be constructed. Our language also contains declarations to associate set-valued expressions with actions. This paper contains a definition of the language and a description of a translator to basic CCS.

1 Introduction

Milner [2] describes CCS as a notation rather than a language, because it contains only the essential core of operators needed for a theoretical study of concurrency. Value-passing CCS, also defined by Milner, provides some of the conveniences a language should have, such as parameterized agents and actions, and conditionals. Unfortunately, tools such as the Concurrency Workbench [1] support only basic CCS, and the definition of value-passing CCS leaves unspecified the types and syntax of data.

We have developed a language for value-passing CCS that provides a concrete syntax and approximately the same set of data types and operations as that found in Milner's text [2]. We have also implemented (with Stuart Anderson) a translator that generates basic CCS agent definitions from declarations in our language.

This paper contains a definition of our language and a discussion of the translator. The semantic definition defines value-passing CCS operationally, without reference to basic CCS. We feel that this approach is more direct than a translational definition and also allows the semantics of agents and data to be handled uniformly.

The data part of our language uses sets, sequences, and tuples as the main data types. Values of these types can be built up from natural number, boolean, and string constants. Since all data operators preserve finiteness, only finite data elements can be constructed. This is necessary since we implement the language by translation to basic CCS.

The only feature found in our language and not in Milner's value-passing CCS is that of indexed actions. The motivation for this feature comes from the common situation in which a process would like to accept an input only if it has a certain value. With parameterized input actions, *any* value must be accepted. An input action in which the index value is the value to be accepted has the desired behavior.

2 Syntax

A program in value-passing CCS consists of three three kinds of definitions: agent definitions, constant definitions, and label definitions.

program	::=	*decllist*
decllist	::=	*decl*
	|	*decl decllist*
decl	::=	*constdef* | *labeldef* | *agentdef* | *pagentdef*

The syntax of definitions is as follows (keywords given in boldface):

constdef	::=	**const** *ident* = *vexpr*
labeldef	::=	**label** *labeltypelist*
agentdef	::=	**agent** *const* = *aexpr*
pagentdef	::=	**agent** *const*(*params*) = *aexpr*
labeltypelist	::=	*labeltype*
	|	*labeltype*, *labeltypelist*
labeltype	::=	*label*
	|	*label*~*vexpr*
	|	*label*(*vexpr*)
params	::=	*param*
	|	*param*, *params*
param	::=	*ident*:*vexpr*

where *ident* is an upper or lower-case letter followed by zero or more alphanumerics and *const* is an

upper-case letter followed by zero or more alphanumerics.

2.1 agent expressions

Agent expressions have the following syntax:

aexpr	::=	**nil**
	|	*t.aexpr*
	|	*label.aexpr*
	|	*label~vexpr.aexpr*
	|	*name(ident).aexpr*
	|	*coname(vexpr).aexpr*
	|	*name~vexpr(ident).aexpr*
	|	*coname~vexpr(vexpr).aexpr*
	|	*aexpr* + *aexpr*
	|	*aexpr* | *aexpr*
	|	*aexpr*\{*namelist*}
	|	*aexpr*[*sublist*]
	|	**if** *vexpr* **then** *aexpr*
	|	**if** *vexpr* **then** *aexpr* **else** *aexpr*
	|	**sum** (*ident*:*vexpr*, *aexpr*)
	|	**comp** (*ident*:*vexpr*, *aexpr*)
	|	*const*
	|	*const*(*vexprlist*)
	|	(*aexpr*)

where *name* is a lower-case letter followed by zero or more alphanumerics and *coname* is a name starting with a apostrophe ('). Note that the letter 't' represents the symbol τ.

The operators on agent expressions have the following precedence, listed in decreasing order: restriction and relabelling, prefix, composition, summation.

Auxiliary syntax for agent expressions:

namelist	::=	*name*
	|	*name*, *namelist*
sublist	::=	*asub*
	|	*asub*, *sublist*
asub	::=	*label* / *label*
label	::=	*name* | *coname*

2.2 value expressions

Value expressions can be basic expressions, set expressions, tuple expressions, or sequence expressions:

vexpr ::= *basicexpr* | *setexpr* | *tupexpr* | *seqexpr*

The syntax for basic expressions:

basicexpr ::= *ident*
| *str*
| *num*
| *bool*
| *unop vexpr*
| *vexpr binop vexpr*
| (*vexpr*)

where *unop* $\in$ {**not**}, *binop* $\in$ {+, −, **mod**, =, <>, <, <=, >=, >, **and**, **or**}, *str* is a string of characters enclosed by double quotes, *num* is one or more decimal digits, and *bool* is **true** or **false**.

The binary operators are all left associative, and are given in order of decreasing precedence. The logical operator **not** has higher precedence than **and** but lower precedence than the relational operators. The syntax for set expressions:

setexpr ::= {}
| {*vexprlist*}
| {*name*:*vexpr* | *vexpr*}
| {*vexpr* | *name*:*vexpr*}
| {*vexpr* | *vexpr* **where** *name*:*vexpr*}
| **prod** (*vexprlist*)
| **pow** (*vexpr*)
| **union** (*vexpr*, *vexpr*)
| **diff** (*vexpr*, *vexpr*)
| **size** (*vexpr*)
| **member** (*vexpr*, *vexpr*)

vexprlist ::= *vexpr*
| *vexpr*, *vexprlist*

The syntax for tuple expressions:

tupexpr ::= << *vexprlist* >>
| *vexpr*#*vexpr*
| **update** (*vexpr*, *vexpr*, *vexpr*)

The syntax for sequence expressions:

$$
\begin{array}{lll}
\textit{seqexpr} & ::= & [\,] \\
& | & \textbf{seqs}\,(\textit{vexpr}, \textit{vexpr}) \\
& | & \textbf{cons}\,(\textit{vexpr}, \textit{vexpr}) \\
& | & \textbf{hd}\,(\textit{vexpr}) \\
& | & \textbf{tl}\,(\textit{vexpr}) \\
& | & \textbf{append}\,(\textit{vexpr}, \textit{vexpr}) \\
& | & \textbf{len}\,(\textit{vexpr})
\end{array}
$$

Comments are allowed anywhere in a program; their syntax is (**comment**). Nested comments are not allowed.

3 Semantics

The semantics of the value-passing language will be defined using structured operational semantics [3].

We will need to evaluate phrases of the language relative to an environment, because in value-passing CCS parameterized actions depend on an association established between names and sets of values. For example $a(x).P \xrightarrow{a(1)} P$ is only possible if the name a is associated with a set containing the value 1. The semantics of basic CCS given in [2] implicitly refer to an environment in the definition of agent constants.

The environment is formalized as a partial function from variables to values. The operation of updating an environment, denoted by EE', is defined as follows:

$$
(EE')(x) = \begin{cases} E'(x) & \text{if defined} \\ E(x) & \text{otherwise} \end{cases}
$$

Sequents in the semantics have two forms. The first form is

$$
E \vdash \textit{phrase} \Longrightarrow \textit{result}
$$

Informally, this sequent is read as: "evaluating the phrase in environment E yields the result". Sequents of this form are used in three different ways. If the phrase is a value expression, then the result is a value. If the phrase is a declaration, then the result is an environment. If the phrase is a CCS constant, then the result is a CCS expression.

The second form of sequent is

$$
E \vdash P \xrightarrow{\alpha} P'
$$

This sequent is read as: "the transition $P \xrightarrow{\alpha} P'$ is possible in environment E".

In the semantic rules that follow, E and E' stand for environments, e and e' stand for value expressions, u and v stand for values, and S and T stand for set values. Conventions are also used for CCS terms and actions. P and Q stand for CCS terms, a and b stand for names, and l and m stand for labels. The letter α stands for a label or an extended label, which is a label that has been indexed or parameterized or both.

The notation $\tilde{v}$ denotes the indexed set $\{v_i \mid i \in I\}$ of values, where I is understood.

declarations

declaration list

$$\frac{E \vdash d \Longrightarrow E_1 \quad E \vdash \mathit{decls} \Longrightarrow E_2}{E \vdash d\ \mathit{decls} \Longrightarrow E_1 E_2}$$

constant

$$\frac{E \vdash e \Longrightarrow v}{E \vdash \textbf{const}\ x = e \Longrightarrow \{(x, v)\}}$$

simple label

$$E \vdash \textbf{label}\ a \Longrightarrow \{(a, \mathit{simple})\}$$

indexed label

$$E \vdash \textbf{label}\ a\tilde{}e \Longrightarrow \{(a, \mathit{index})\}$$

parameterized label

$$\frac{E \vdash e \Longrightarrow S}{E \vdash \textbf{label}\ a(e) \Longrightarrow \{(a, \mathit{param}(S))\}}$$

indexed, parameterized label

$$\frac{E \vdash e' \Longrightarrow S}{E \vdash \textbf{label}\ a\tilde{}e(e') \Longrightarrow \{(a, \mathit{indexparam}(S))\}}$$

agent

$$E \vdash \textbf{agent}\ P = Q \Longrightarrow \{(P, Q)\}$$

parameterized agent

$$E \vdash \textbf{agent}\ P(\tilde{x}{:}\tilde{e}) = Q \Longrightarrow \{(P, (\tilde{x}, Q))\}$$

Label declarations create an association in the environment between actions and label types, where label types are built using the constructors *simple*, *index*, *param*(S), and *indexparam*(S). Note that the constructors *index* and *indexparam* do not make use of the indexing set given in the corresponding label declarations. As a consequence, actions that might be considered as type errors can occur. For example, the action a_0 could occur even if the value 0 was absent from the set given in the label declaration for the indexed action a. Similar type errors are possible with parameterized output actions. Future versions of the language and translator will handle these type errors.

Note also that expressions in declarations are evaluated immediately, giving static binding and requiring that constants be declared before use.

agent expressions

tau action

$$E \vdash t.P \xrightarrow{\tau} P$$

simple prefix

$$\frac{E \vdash a \Longrightarrow \mathit{simple}}{E \vdash l.P \xrightarrow{l} P} \quad a = \mathit{base}(l)$$

indexed prefix

$$\frac{E \vdash a \Longrightarrow \mathit{index} \quad E \vdash e \Longrightarrow v}{E \vdash l\tilde{}e.P \xrightarrow{l_v} P} \quad a = \mathit{base}(l)$$

parameterized input action

$$\frac{E \vdash a \Longrightarrow \mathit{param}\,(S)}{E \vdash a(x).P \xrightarrow{a(v)} P} \quad v \in S$$

parameterized output action

$$\frac{E \vdash a \Longrightarrow \mathit{param}\,(S) \quad E \vdash e \Longrightarrow v}{E \vdash \overline{a}(e).P \xrightarrow{\overline{a}(v)} P}$$

indexed, parameterized input action

$$\frac{E \vdash a \Longrightarrow \mathit{indexparam}(S) \quad E \vdash e \Longrightarrow u}{E \vdash a\tilde{}e(x).P \xrightarrow{a_u(v)} P} \quad v \in S$$

indexed, parameterized output action

$$\frac{E \vdash a \Longrightarrow \mathit{indexparam}(S) \quad E \vdash e \Longrightarrow u \quad E \vdash e' \Longrightarrow v}{E \vdash \overline{a}\tilde{}e(e').P \xrightarrow{\overline{a}_u(v)} P}$$

summation-1

$$\frac{E \vdash P \xrightarrow{\alpha} P'}{E \vdash P + Q \xrightarrow{\alpha} P'}$$

summation-2

$$\frac{E \vdash Q \xrightarrow{\alpha} Q'}{E \vdash P + Q \xrightarrow{\alpha} Q'}$$

composition-1

$$\frac{E \vdash P \xrightarrow{\alpha} P'}{E \vdash P \mid Q \xrightarrow{\alpha} P' \mid Q}$$

composition-2

$$\frac{E \vdash Q \xrightarrow{\alpha} Q'}{E \vdash P \mid Q \xrightarrow{\alpha} P \mid Q'}$$

composition-3

$$\frac{E \vdash P \xrightarrow{l} P' \quad E \vdash Q \xrightarrow{\bar{l}} Q'}{E \vdash P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

restriction

$$\frac{E \vdash P \xrightarrow{\alpha} P'}{E \vdash P\backslash L \xrightarrow{\alpha} P'\backslash L} \quad \mathit{base}(\alpha) \notin L$$

relabelling

$$\frac{E \vdash P \xrightarrow{\alpha} P'}{E \vdash P[f] \xrightarrow{f'(\alpha)} P'[f]}$$

conditional-1
$$\frac{E \vdash \mathbf{if}\ e\ \mathbf{then}\ P\ \mathbf{else\ nil} \xrightarrow{\alpha} P'}{E \vdash \mathbf{if}\ e\ \mathbf{then}\ P \xrightarrow{\alpha} P'}$$

conditional-2
$$\frac{E \vdash e \Longrightarrow \mathit{true} \quad E \vdash P \xrightarrow{\alpha} P'}{E \vdash \mathbf{if}\ e\ \mathbf{then}\ P\ \mathbf{else}\ Q \xrightarrow{\alpha} P'}$$

conditional-3
$$\frac{E \vdash e \Longrightarrow \mathit{false} \quad E \vdash Q \xrightarrow{\alpha} Q'}{E \vdash \mathbf{if}\ e\ \mathbf{then}\ P\ \mathbf{else}\ Q \xrightarrow{\alpha} Q'}$$

sigma
$$\frac{E \vdash e \Longrightarrow S \quad E \vdash \sum_{v \in S} P[v/x] \xrightarrow{\alpha} P'}{E \vdash \mathbf{sum}(x{:}e, P) \xrightarrow{\alpha} P'}$$

pi
$$\frac{E \vdash e \Longrightarrow S \quad E \vdash \prod_{v \in S} P[v/x] \xrightarrow{\alpha} P'}{E \vdash \mathbf{prod}(x{:}e, P) \xrightarrow{\alpha} P'}$$

constant-1
$$E \vdash P \Longrightarrow Q \quad \text{if } E(P) = Q$$

constant-2
$$\frac{E \vdash P \Longrightarrow Q \quad E \vdash Q \xrightarrow{\alpha} Q'}{E \vdash P \xrightarrow{\alpha} Q'}$$

parameterized constant-1
$$E \vdash P \Longrightarrow (\tilde{x}, Q) \quad \text{if } E(P) = (\tilde{x}, Q)$$

parameterized constant-2
$$\frac{E \vdash P \Longrightarrow (\tilde{x}, Q) \quad E\{(x_1, v_1), \ldots, (x_n, v_n)\} \vdash Q \xrightarrow{\alpha} Q'}{E \vdash P(\tilde{v}) \Longrightarrow Q}$$

The function *base* in the simple prefix, indexed prefix, and restriction rules returns the name part of a given action:

$$\begin{aligned} \mathit{base}(\tau) &= \tau \\ \mathit{base}(a) &= a \\ \mathit{base}(l_v) &= \mathit{base}(l) \\ \mathit{base}(l(v)) &= \mathit{base}(l) \\ \mathit{base}(\bar{l}) &= \mathit{base}(l) \end{aligned}$$

For example, $\mathit{base}(a_u(v)) = \mathit{base}(a_u) = \mathit{base}(a) = a$.

The function f' in the relabelling rule is an extension of the relabelling function f so that it also applies to the base part of indexed and parameterized actions:

$$
\begin{array}{rcl}
f'(\tau) & = & \tau \\
f'(a) & = & f(a) \\
f'(l_v) & = & f'(l)_v \\
f'(l(v)) & = & f'(l)(v) \\
f'(\bar{l}) & = & \overline{f'(l)}
\end{array}
$$

value expressions

constant $\quad E \vdash v \Longrightarrow v$

identifier $\quad E \vdash x \Longrightarrow v \quad \text{if } E(x) = v$

set comprehension 1
$$\frac{E \vdash \{x \mid e' \textbf{ where } x{:}e\} \Longrightarrow S}{E \vdash \{x{:}e \mid e'\} \Longrightarrow S}$$

set comprehension 2
$$\frac{E \vdash \{e \mid \textbf{true where } x{:}e'\} \Longrightarrow S}{E \vdash \{e \mid x{:}e'\} \Longrightarrow S}$$

set comprehension 3
$$\frac{E \vdash e'' \Longrightarrow \{\tilde{v}\} \quad E\{(x, v_i)\} \vdash e \Longrightarrow v_i' \quad E\{(x, v_i)\} \vdash e' \Longrightarrow b_i}{E \vdash \{e \mid e' \textbf{ where } x{:}e''\} \Longrightarrow \{y \mid y = v_i' \wedge b_i = \mathit{true}\}}$$

tuple selection
$$\frac{E \vdash e \Longrightarrow \langle \tilde{v} \rangle \quad E \vdash e' \Longrightarrow i}{E \vdash e \# e' \Longrightarrow v_i}$$

tuple update
$$\frac{E \vdash e \Longrightarrow \langle \tilde{v} \rangle \quad E \vdash e' \Longrightarrow i \quad E \vdash e'' \Longrightarrow u}{E \vdash \textbf{update } (e, e', e'') \Longrightarrow \langle \tilde{v}' \rangle} \quad \tilde{v}' = \tilde{v}, \text{ except } v_i' = u$$

sequences
$$\frac{E \vdash e \Longrightarrow n \quad E \vdash e' \Longrightarrow S}{E \vdash \textbf{seqs}(e, e') \Longrightarrow \bigcup_{i=0}^{n} S^i}$$

Most value expressions are not given semantic rules, since the types and operators are given their common interpretation. For example, in the constant rule, the syntactic entity 1 is given as its interpretation the integer 1. As another example, the operator + is interpreted as integer addition.

In case their meaning is not entirely clear, we state here that the set operators *prod*, *pow*, *diff*, *size*, and *member* refer to, respectively, the set product, the power set, the set difference, the cardinality of a set, and set membership. The sequence operators *hd*, *tl*, *append*, and *len* refer to, respectively, the head of a sequence, the tail of a sequence, the appending of two sequences, and the length of a sequence.

To define the relational operators, we make use of a total order (denoted here by $\prec$) over *all* values. This relation is defined between values of different types by ordering the types themselves: booleans $\prec$ naturals $\prec$ strings $\prec$ tuples $\prec$ sequences $\prec$ sets. So, for example, $1 \prec \{\}$. The following rules define the order relation within each type. For booleans, **false** $\prec$ **true**. Natural numbers are ordered by the relation $\leq$. Strings, tuples, and sequences are ordered lexicographically. Sets are ordered as sequences by ordering the elements they contain and then removing duplicates.

4 Translating to basic CCS

Our value-passing language is implemented through a two-step translation to basic CCS. In the first step, a value-passing program is translated to an ml expression. The ml expression contains constructors of datatypes that capture the abstract syntax of value-passing CCS. For example, the abstract syntax of agent expressions is represented by the datatype *Beh*, and a conditional agent expression is represented by the constructor *cond* **of** *Elem* $*$ *Beh* $*$ *Beh*. The ml expression also refers to functions that perform the translation to basic CCS. For example, the function *GenElems* takes as input a set-valued expression and returns a list containing the elements in the set.

The second step of the translation is to execute the ml expression produced in the first step, producing a file of basic CCS definitions in a form readable by the Concurrency Workbench. In certain syntactic respects, the basic CCS produced is not true to the operational semantics we have given. For example, the subscripted actions found in the semantics cannot be represented in the basic CCS produced simply because the Workbench accepts only ASCII characters. Therefore, subscripted actions are encoded into ASCII strings by the translator. The value part of a subscripted action need also be encoded, since mathematical symbols such as $\{$ and $\cup$ are not allowed by the Workbench in an action name.

The encoding scheme is precisely described here so that the basic CCS generated by the translator can be understood:

$$
\begin{array}{rcl}
[\![\tau]\!] & \Rightarrow & t \\
[\![a]\!] & \Rightarrow & a \\
[\![\bar{a}]\!] & \Rightarrow & {}'a \\
[\![l_v]\!] & \Rightarrow & [\![l]\!]\text{\_}[\![v]\!] \\
[\![l(v)]\!] & \Rightarrow & [\![l]\!]\text{-}[\![v]\!] \\
[\![l_u(v)]\!] & \Rightarrow & [\![l_u]\!]\text{-}[\![v]\!] \\
[\![P(\tilde{v})]\!] & \Rightarrow & P\text{\_}[\![<< \tilde{v} >>]\!] \\
[\![\{\tilde{v}\}]\!] & \Rightarrow & \#[\![\tilde{v}]\!]* \\
[\![[\tilde{v}]]\!] & \Rightarrow & \%[\![\tilde{v}]\!]\& \\
[\![\langle\tilde{v}\rangle]\!] & \Rightarrow & < [\![\tilde{v}]\!] > \\
[\![str]\!] & \Rightarrow & str \\
[\![\mathbf{true}]\!] & \Rightarrow & \text{“true”} \\
[\![\mathbf{false}]\!] & \Rightarrow & \text{“false”} \\
[\![int]\!] & \Rightarrow & int
\end{array}
$$

Some tricks are used in the translation process so that the work of maintaining environments and performing substitutions is mostly done by ml. A constant declaration, for example, is translated directly to an ml **let** expression. A parameterized input operator in value-passing CCS is translated to a datatype constructor having the signature

$$
inp\ \mathbf{of}\ Atom * LabelSort * (Elem \rightarrow Beh)
$$

The first parameter, an *Atom*, is the name of the input action. The second parameter gives the label type, which specifies the set over which the input value can range. The third parameter is a function from a value to the agent expression given after the prefix operator. For example, the CCS expression $a(x).P(x)$ generates the ml expression $inp(\text{“}a\text{”}, Param(\{1,2\}), \mathbf{fn}\ x \Rightarrow pconst(\text{“}P\text{”}, x))$, assuming that a has been declared as a parameterized label ranging over $\{1,2\}$, and that *pconst* is a constructor used

to represent parameterized agent constants. Substitution of a value for x within the agent expression P is accomplished by applying the function to the value.

5 Conclusions

The development of our language and translator was motivated by the difficulty in using basic CCS to model real systems. Our experience has confirmed that the language makes the modelling of these systems much more convenient. A secondary benefit is that it is now usually possible to directly express agents in the manner found in Milner's text.

The basic CCS generated by our translator has values encoded into the names of agents and actions, and is therefore awkward to manipulate when using the Concurrency Workbench. Work is currently underway to integrate the translator into the Workbench so that translated terms are never seen by users. In the long term, we plan to have the Workbench manipulate value-passing CCS terms directly.

Indexed actions are treated slightly differently in our language and Milner's. We treat the restriction and relabelling of indexed actions much the same as parameterized actions, so that an action indexed by a particular value cannot be restricted or relabelled independently of the actions indexed with other values. This aspect of indexed actions will probably be changed once a simple definition has been found. Better type checking for indexed actions and for parameterized output actions is another planned improvement.

We have considered extending our language with features such as the pattern-matching of agent parameters, and a "let" facility for making local definitions. We have so far resisted making such additions because the simplicity of our language makes it more suitable for many problems than "heavier" notations such as Lotos.

There is still work to be done in meeting correctness concerns. First, is the semantic definition given here consistent with the translational semantics given by Milner in [2]? Second, is the translator correct with respect to the semantics? We plan to report on these issues in a later paper.

References

[1] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The concurrency workbench: A semantics based tool for the verification of concurrent systems. Technical Report ECS-LFCS-89-83, Laboratory for Foundations of Computer Science, Dept. of Computer Science, University of Edinburgh, 1989.

[2] Robin Milner. *Communication and Concurrency.* Prentice Hall International, 1989.

[3] G.D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI-FN-19, Computer Science Department, Århus University, Denmark, 1981.

A An example

```
(* Scheduler specification from Milner's text, page 115 *)

const n     = 3
const index = {0,1,2}
const iset  = pow(index)

label a~index, b~index

agent Schedspec(i:index,X:iset) =
    if member(i,X) then
        sum(j:X,b~j.Schedspec(i,diff(X,{j})))
    else
        a~i.Schedspec((i + 1) mod n,union(X,{i}))
      + sum(j:X,b~j.Schedspec(i,diff(X,{j})))
```