A Language for Value-Passing CCS

Glenn Bruns
Bell Labs, Lucent Technologies
1000 E. Warrenville Rd., Room 1G-339
Naperville, IL 60540, USA
grb@bell-labs.com

January 13, 1998

Abstract

This paper, taken directly from Appendix C of [1], gives the syntax and semantics for a value-passing CCS. Our language is based on the value-passing version of CCS presented by Milner in [2]. Like Milner, we define the semantics of our notation by translation to basic CCS. Unlike Milner, our notation supports both parametrized and indexed actions, and includes sets, tuples, and lists as built-in data types.

1 Introduction

Milner [2] defines value-passing CCS by translation into basic CCS, thereby showing how values can be reduced to agents. The definition is also interesting as it suggests that tools for the analysis of basic agents can be used for value-passing agents. However, Milner's notation and translation have shortcomings when used for this purpose. First, the translation assumes a fixed but unspecified set of values. In a complete notation a particular syntax and semantics for values must be supplied. Second, value-passing CCS contains only parametrized actions. It is also helpful to have simple actions, indexed actions, and actions that are both indexed and parametrized. Finally, value-passing agents are very weakly typed. For example, an agent can be defined to have two formal parameters but used with three actual parameters. In practice, this discrepancy is the result of an error and should be reported.

Here we describe a language based on value-passing CCS that addresses these issues. The language, and a translator into basic CCS, were developed by the author and Stuart Anderson.

2 Overview

A program is a sequence of declarations, of which there are three kinds. *Constant declarations* bind identifiers to values. For example:

```
const A = pow({0,1})
```

Here A is declared as the power set of $\{0, 1\}$. The language supports finite sets, lists and tuples built from boolean, integer and string constants. *Label declarations* show how actions are parametrized or indexed. For example,

```
label b(A)
```

Here b is declared as a parametrized action with parameter values ranging over set A. *Agent declarations* are like the agent definitions of basic CCS. For example,

```
agent P(X:A) = if X = {} then Q else b(Y).P(diff(X,Y))
```

Here P is declared as an agent with parameter X of type A.

The following sample program is based on the scheduler specification of page 115 in [2].

```
const n    = 3
const I    = {0,1,2}
const Iset = pow(I)

label a_I, b_I

agent Schedspec(i:I,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})))
```

The meaning of a program is given in two steps. First, the declarations are evaluated to form an environment. Two kinds of evaluation relations are used. The first has the form

$$E \vdash e \ \rhd v$$

This relation means that value expression e has value v in environment E. The second relation has the form

$$E \vdash decl \Rightarrow E'$$

This relation means that declaration *decl* extends the environment E to E'.

Second, the agent declarations are translated into basic agent definitions relative to the environment. The translation of a declaration relative to environment E is written

$$[\![decl]\!]_E$$

The language definition is organized as follows. First we define value expressions and their evaluation to values. Then we define declarations and how they are evaluated to extend environments. Finally we present the translation function and show how the parts of the definition fit together.

3 Values

The basic data types of the language are booleans, integers and strings. The type constructors are tuples, lists and finite sets.

Table 1 shows the phrase classes for expressions. Unless otherwise indicated, the lower-case form of the class name serves as a variable over the class. For example, variable *unop* ranges over Unop.

x	$\in$	Var	value variables
		UnOp	unary value operators
		BinOp	binary value operators
		FName	function names
e	$\in$	VExpr	value expressions

Table 1: Phrase classes

A *Var* is an upper- or lower-case letter followed by zero or more alphanumerics, where an alphanumeric is either a decimal digit, an upper- or lower-case letter, or an apostrophe ('). Classes UnOp and BinOp are defined as $\{$`not`$\}$ and $\{+, -,$ `mod`, $=, <>, <, <=, >=, >,$ `and`, `or`$\}$, with the symbols of BinOp listed in order of decreasing binding power. The operator `not` binds more tightly than `and` but less tightly than the relational operators. Class FName is defined as the union of the set function names $\{$`prod`,`pow`,`union`,`diff`,`size`,`member`,`range`,`iter`$\}$, the tuple function name $\{$`update`$\}$, and the list function names $\{$`seqs`,`cons`,`hd`,`tl`,`append`,`len`$\}$.

The syntax of expressions is shown in Table 2. Reserved words are written in `typewriter font` and may not be used as identifiers. The literals are boolean literals, integer literals, and string literals. Their form is given in Section 3.1.

e	::=	*lit*	literal
		x	variable
		unop e	unary operator
		e *binop* e	infix binary operator
		fname$(e_1, \ldots, e_n)$	function application
		`{` $e_1, \ldots, e_n$ `}`	set construction
		`{` e `|` e `where` $x : e$`}`	set comprehension
		`<<` $e_1, \ldots, e_n$ `>>`	tuple construction
		e`#`e	tuple selection
		`[` $e_1, \ldots, e_n$ `]`	list construction
		(e)	parenthesized expression

Table 2: Grammar: value expressions

The semantics of value expressions is defined inductively as the evaluation relation $E \vdash e \triangleright v$, which captures the expression e evaluating to value v, where the free variables of e are defined in the environment E. When the meaning of e is clear we omit its evaluation rule.

In this section an environment E is a finite set of pairs (x, v), which indicates that x is bound to v in E. If E and E' are environments then EE' is the environment in which the variables of both E and E' are defined and the bindings of E' take precedence over those of E. In later sections environments become more complex to take account of other kinds of identifiers.

The evaluation rules for literals and variables are

$$E \vdash \mathit{lit} \ \rhd \mathit{val}(\mathit{lit})$$

and

$$E \vdash x \ \rhd v \quad (x, v) \in E$$

The first rule says that a literal evaluates to its corresponding value. The second says that a variable evaluates to the value to which it is bound in the environment E.

In presenting the evaluation rules for the other expressions the symbol S is used rather than v as a variable over set values. So $E \vdash e \rhd S$ indicates that the result must be set-valued. Similarly, b is used as a variable over booleans.

3.1 Basic Types

The boolean literals are `true` and `false` and their corresponding values are *true* and *false*. The operations are `and`, `or`, `not`. A typical evaluation rule is

$$\frac{E \vdash e_1 \ \rhd b_1 \quad E \vdash e_2 \ \rhd b_2}{E \vdash e_1 \ \texttt{and} \ e_2 \ \rhd b_1 \wedge b_2}$$

Here the symbol $\wedge$ stands for logical conjunction. The other boolean operations also have their usual interpretation.

The integer literals are sequences of digits optionally preceded by a plus or minus sign. The integer literals evaluate to the conventional values. The operations are $+$, $-$ and `mod`. A typical evaluation rule is

$$\frac{E \vdash e_1 \ \rhd n_1 \quad E \vdash e_2 \ \rhd n_2}{E \vdash e_1\texttt{+}e_2 \ \rhd n_1 + n_2}$$

The other integer operations also have their usual interpretation.

The string literals are any sequence of printable ASCII characters between double quotes (`" "`). There are no operations on strings.

3.2 Tuples

If $e_1, \ldots, e_n$ are expressions then `<<` $e_1, \ldots, e_n$ `>>` is an expression which evaluates to a tuple. The evaluation rule is

$$\frac{E \vdash e_i \ \rhd v_i \quad 1 \leq i \leq n}{E \vdash \texttt{<<}e_1, \ldots, e_n\texttt{>>} \ \rhd (v_1, \ldots, v_n)}$$

The following rules define the selecting and updating of tuples. Informally, T`#`i selects the ith element of tuple T, and `update(`T, i, v`)` updates the ith element of tuple T to v.

$$\frac{E \vdash e_1 \ \rhd (v_1, \ldots, v_n) \quad E \vdash e_2 \ \rhd i \quad 1 \leq i \leq n}{E \vdash e_1\texttt{\#}e_2 \ \rhd v_i}$$

$$\frac{E \vdash e_1 \ \rhd (v_1, \ldots, v_n) \quad E \vdash e_2 \ \rhd i \quad E \vdash e_3 \ \rhd v'}{E \vdash \texttt{update(}e_1, e_2, e_3\texttt{)} \ \rhd (v_1, \ldots, v_{i-1}, v', v_{i+1}, \ldots, v_n)}$$

3.3 Lists

If $e_1, \ldots, e_n$ are expressions then $\texttt{[}e_1, \ldots, e_n\texttt{]}$ is an expression which evaluates to a list value. The evaluation rule is similar to that for tuples. The other operations defined for lists are as follows:

- $\texttt{cons}(x, l)$ returns the list l with x added as its first element.
- $\texttt{hd}(l)$ returns the first element of list l.
- $\texttt{tl}(l)$ returns the list l less its first element.
- $\texttt{append}(l_1, l_2)$ returns the list containing the elements of l_1 followed by the elements of l_2.
- $\texttt{len}(l)$ returns the number of elements in l.

3.4 Finite Sets

If $e_1, \ldots, e_n$ are expressions then $\texttt{\{}e_1, \ldots e_n\texttt{\}}$ is an expression which evaluates to a set value. The evaluation rule is

$$\frac{E \vdash e_i \rhd v_i \quad 1 \leq i \leq n}{E \vdash \texttt{\{}e_1, \ldots, e_n\texttt{\}} \rhd \{v_1, \ldots, v_n\}}$$

Other operations defined for sets are as follows:

- $\texttt{prod}(s_1, \ldots, s_n)$ returns the set product $s_1 \times \ldots \times s_n = \{(x_1, \ldots, x_n) \mid x_i \in s_i\}$.
- $\texttt{pow}(s)$ returns the power set of s.
- $\texttt{union}(s_1, s_2)$ returns the set $s_1 \cup s_2$.
- $\texttt{diff}(s_1, s_2)$ returns the set $s_1 \setminus s_2$.
- $\texttt{size}(s)$ returns the number of elements in set s.
- $\texttt{member}(x, s)$ returns *true* if x is a member of set s, else *false*.
- $\texttt{range}(i, j)$ returns the set of integers from i to j, inclusive.
- $\texttt{iter}(s, \texttt{fn}\ x\ \texttt{=>}\ e)$ returns the value obtained by repeatedly applying the monotonic, set-valued function $f(x) = e$, starting with set S, until a fixed point is reached.

There is a set comprehension expression for building sets. Informally, $\{e \,|\, p\ \texttt{where}\ x : A\}$ is the set of values for expression e where the variable x ranges over the elements of set A that satisfy predicate p.

$$\frac{E \vdash e_3 \rhd \{v_1, \ldots, v_k\} \quad E\{(x, v_i)\} \vdash e_1 \rhd v_i' \quad E\{(x, v_i)\} \vdash e_2 \rhd b_i \quad 1 \leq i \leq k}{E \vdash \texttt{\{}e_1\ \texttt{|}\, e_2\ \texttt{where}\ x\,\texttt{:}\,e_3\texttt{\}} \rhd \bigcup_{1 \leq i \leq k} \{v_i' \mid b_i\}}$$

There are two derived forms for set comprehension. The expression $\texttt{\{}\ x\texttt{:}\ e_1\ \texttt{|}\ e_2\ \texttt{\}}$ is an abbreviation for $\texttt{\{}\ x\ \texttt{|}\ e_2\ \texttt{where}\ x\texttt{:}\ e_1\ \texttt{\}}$, and the expression $\texttt{\{}\ e_1\ \texttt{|}\ x\texttt{:}\ e_2\ \texttt{\}}$ is an abbreviation for $\texttt{\{}\ e_1\ \texttt{|}\ \textit{true}\ \texttt{where}\ x\texttt{:}\ e_2\ \texttt{\}}$.

The expression $\texttt{seqs(}n, S\texttt{)}$ is the set of all lists built from set S with length n or less.

$$\frac{E \vdash e_1 \rhd n \quad E \vdash e_2 \rhd S}{E \vdash \texttt{seqs(}e_1, e_2\texttt{)} \rhd \bigcup_{i=0}^{n} S^i}$$

3.5 Relational Expressions

The relational operators (=, <>, <, <=, >, =>), are defined by reference to a total order (denoted by $\prec$) over *all* values. This relation is defined between values of different types by ordering the types themselves: booleans $\prec$ integers $\prec$ strings $\prec$ tuples $\prec$ sequences $\prec$ sets. For example, $1 \prec \{\}$. The following rules define the order relation within each type. For booleans, `false` $\prec$ `true`. For the integers $n \prec m$ just if $n < m$. Strings, tuples and lists are ordered lexicographically. Sets are ordered by considering their elements as an ordered list.

4 Declarations and Environments

This section presents the syntax for declarations and the semantic rules for forming environments from declarations. These environments are used to translate value-passing agents into basic ones.

The phrase classes for declarations and agents appear in Table 3. A *Name* is a lower-case letter followed by zero or more alphanumerics. A *Const* is an upper-case letter followed by zero or more alphanumerics.

		Program	programs
		Decl	declarations
A	$\in$	Const	agent constants
P	$\in$	AExpr	agent expressions
a	$\in$	Name	CCS names
l	$\in$	LName	long names

Table 3: Phrase classes

The syntax for declarations is given in Table 4. The notation $\langle w \rangle$ indicates that w is optional.

program	::=	$decl_1 \ldots decl_n$	
decl	::=	`const` $x = e$	constant declaration
		`label` $l_1, \ldots, l_n$	label declaration
		`agent` $A = P$	agent declaration
		`agent` A`(`x_1:$e_1, \ldots, x_n$:e_n`)`$= P$	parametrized agent declaration
l	::=	$a\langle\_e\rangle$	simple name
		$a\langle\_e\rangle$`(`e`)`	parametrized name

Table 4: Grammar: declarations

The semantic rules for declarations have the form

$$E \vdash decl \Rightarrow E'$$

This rule captures how the current environment E is augmented by the declaration to become environment E'. For each kind of declaration (constant, label and agent) there is a corresponding kind of environment (E_c, E_l and E_a). The total environment is the triple (E_c, E_l, E_a).

The constant environment E_c is a finite set of pairs (x, v). This kind of environment was used in the previous section to evaluate value expressions.

The label environment E_l is a finite set of elements of one of four forms:

- a for simple labels
- a_S for indexed labels
- $a(S)$ for non-indexed value-passing labels
- $a\_S(S')$ for indexed, value-passing labels.

The set values S and S' record the type of the index and of the parameter. For example, if a_S is in E_l then a is a label with an index ranging over set S.

The agent environment E_a is a finite set of elements of one of two forms:

- A for simple (non-parametrized) agent names.
- $A(S_1, \ldots S_n)$ for parametrized agent names. The set values record the types of the parameters.

Again, the notation EE' indicates updating. Often only one component of the environment is updated, so EE'_c stands for $(E_cE'_c, E_l, E_a)$ and similarly for the label and agent environments.

4.1 Labels

To simplify the rules for declarations the evaluation relation of the previous section is extended to label names:

$$E \vdash a \ \triangleright a$$

$$\frac{E \vdash e \ \triangleright S}{E \vdash a\_e \ \triangleright a\_S}$$

$$\frac{E \vdash e \ \triangleright S}{E \vdash a(e) \ \triangleright a(S)}$$

$$\frac{E \vdash e \ \triangleright S \quad E \vdash e' \ \triangleright S'}{E \vdash a\_e(e') \ \triangleright a\_S(S')}$$

4.2 Programs

$$\frac{E \vdash decl_1 \Rightarrow E_1 \quad \ldots \quad E_{n-1} \vdash decl_n \Rightarrow E_n}{E \vdash decl_1 \ldots decl_n \Rightarrow E_n}$$

4.3 Declarations

$$\frac{E_c \vdash e \ \triangleright v}{E \vdash \texttt{const} \ \ x = e \Rightarrow E\{(x, v)\}}$$

$$\frac{E_c \vdash l_i \ \triangleright v_i \quad 1 \leq i \leq n}{E \vdash \texttt{label} \ \ l_1, \ldots, l_n \Rightarrow E\{v_1, \ldots, v_n\}}$$

$$E \vdash \texttt{agent} \ \ A = P \Rightarrow E\{A\}$$

$$\frac{E_c \vdash e_i \ \triangleright S_i \quad 1 \leq i \leq n}{E \vdash \texttt{agent} \ \ A(x_1 : e_1, \ldots, x_n : e_n) = P \Rightarrow E\{A(S_1, \ldots, S_n)\}}$$

5 Translating into Basic CCS

Table 5 gives the translation from value-passing CCS into basic CCS. It is presented in the same style as Milner's translation but relative to an explicit environment. The meaning of a program is written $[\![program]\!]$, and defined using the translation $[\![program]\!]_E$ as follows

$$[\![program]\!] \stackrel{\text{def}}{=} [\![program]\!]_E \quad where\ (\{\},\{\},\{\}) \vdash program \Rightarrow E$$

The translation function $[\![\_]\!]_E$ is now defined. A program is translated into a sequence of agent declarations.

$$[\![decl_1 \ldots decl_n]\!]_E \stackrel{\text{def}}{=} [\![decl_1]\!]_E \ldots [\![decl_n]\!]_E$$

Constant and label declarations are mapped to ϵ, the empty sequence.

$$\begin{aligned} [\![\texttt{const}\ x = e]\!]_E &\stackrel{\text{def}}{=} \epsilon \\ [\![\texttt{label}\ l_1, \ldots, l_n]\!]_E &\stackrel{\text{def}}{=} \epsilon \end{aligned}$$

Non-parametrized agents are translated by translating their right-hand side.

$$[\![\texttt{agent}\ A = P]\!]_E \stackrel{\text{def}}{=} \texttt{agent}\ A = [\![P]\!]_E$$

Parametrized agents are translated into a sequence of agent definitions, one definition for each possible combination of parameter values. Suppose we have declaration

$$\texttt{agent}\ A(x_1{:}e_1, \ldots, x_n{:}e_n) = P$$

Suppose also that $A(S_1, \ldots S_n) \in E_a$ and that $S_1 \times \cdots \times S_n = \{T_1, \ldots, T_k\}$, where T_i is an n-tuple and $T_i \prec T_{i+1}$ for $1 \leq i < n$. Then the translation $[\![\texttt{agent}\ A(x_1{:}e_1, \ldots, x_n{:}e_n) = P]\!]_E$ is a sequence of k declarations, the ith of which is

$$\texttt{agent}\ A_{v_1,\ldots,v_n} = [\![P]\!]_{E\{(x_1,v_1),\ldots,(x_n,v_n)\}}$$

where $(v_1, \ldots, v_n) = T_i$.

The translation of agents with restriction and relabelling is complicated because our language allows labels used with these operators to be indexed or parametrized. To define the operators an auxiliary function is used that translates sets of labels relative to the environment.

$$[\![\texttt{\{}a\texttt{\}}]\!]_E \stackrel{\text{def}}{=} \begin{cases} \texttt{\{}a\texttt{\}} & \text{if } a \in E_l \\ \texttt{\{}a_{v_1}, \ldots, a_{v_n}\texttt{\}} & \text{if } a\_\{v_1, \ldots, v_n\} \in E_l \\ \texttt{\{}a^{v_1}, \ldots, a^{v_n}\texttt{\}} & \text{if } a(\{v_1, \ldots, v_n\}) \in E_l \\ \texttt{\{}a_{v_1}^{w_1}, \ldots, a_{v_n}^{w_m}\texttt{\}} & \text{if } a\_\{v_1, \ldots, v_n\}(\{w_1, \ldots, w_m\}) \in E_l \end{cases}$$

$$[\![\texttt{\{}a\_e\texttt{\}}]\!]_E \stackrel{\text{def}}{=} \begin{cases} \texttt{\{}a_v\texttt{\}} & \text{if } a\_I \in E_l \text{ and } E_c \vdash e \ \triangleright v, v \in I \\ \texttt{\{}a_v^{w_1}, \ldots, a_v^{w_m}\texttt{\}} & \text{if } a\_I(\{w_1, \ldots, w_m\}) \in E_l \\ & \text{and } E_c \vdash e \ \triangleright v, v \in I \end{cases}$$

$$\begin{aligned} [\![\texttt{\{}a\langle\_e\rangle(e')\texttt{\}}]\!]_E \stackrel{\text{def}}{=}\ & \texttt{\{}a\langle_v\rangle^{v'}\texttt{\}} \text{ if } a\langle\_I\rangle(S) \in E_l \text{ and } E_c \vdash e \ \triangleright v, v \in I \\ & \text{and } E_c \vdash e' \ \triangleright v', v' \in S \end{aligned}$$

P	$[\![P]\!]_E$	
`0`	`0`	
$\mathtt{tau}.P$	$\mathtt{tau}.[\![P]\!]_E$	
$a\langle\_e\rangle.P$	$a\langle_i\rangle.[\![P]\!]_E$	$a\langle\_I\rangle \in E_l$, $E_c \vdash e \ \triangleright i, i \in I$
$\text{'}a\langle\_e\rangle.P$	$\text{'}a\langle_i\rangle.[\![P]\!]_E$	$a\langle\_I\rangle \in E_l$, $E_c \vdash e \ \triangleright i, i \in I$
$a\langle\_e\rangle\mathtt{(}x\mathtt{)}.P$	$\Sigma_{v\in V}\, a\langle_i\rangle^v.[\![P]\!]_{E\{(x,v)\}}$	$a\langle\_I\rangle(V) \in E_l$, $E_c \vdash e \ \triangleright i, i \in I$ $v \in V$
$\text{'}a\langle\_e_1\rangle\mathtt{(}e_2\mathtt{)}.P$	$\text{'}a\langle_i\rangle^v.[\![P]\!]_E$	$a\langle\_I\rangle(V) \in E_l$, $E_c \vdash e_1 \ \triangleright i, i \in I$, $E_c \vdash e_2 \ \triangleright v, v \in V$
$P_1 \texttt{ + } P_2$	$[\![P_1]\!]_E \texttt{ + } [\![P_2]\!]_E$	
$P_1 \texttt{ \| } P_2$	$[\![P_1]\!]_E \texttt{ \| } [\![P_2]\!]_E$	
$P\backslash\mathtt{\{}r\mathtt{\}}$	$[\![P]\!]_E\backslash[\![\mathtt{\{}r\mathtt{\}}]\!]_E$	
$P\mathtt{[}r\mathtt{]}$	$[\![P]\!]_E[\![\mathtt{[}r\mathtt{]}]\!]_E$	
$\mathtt{if}\ e\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2$	$[\![P_1]\!]_E$	$E_c \vdash e \ \triangleright \mathit{true}$
$\mathtt{if}\ e\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2$	$[\![P_2]\!]_E$	$E_c \vdash e \ \triangleright \mathit{false}$
$\mathtt{sum(}x\mathtt{:}e\mathtt{,}P\mathtt{)}$	$\Sigma_{v\in S}[\![P]\!]_{E\{(x,v)\}}$	$E_c \vdash e \ \triangleright S$
$\mathtt{comp(}x\mathtt{:}e\mathtt{,}P\mathtt{)}$	$\Pi_{v\in S}[\![P]\!]_{E\{(x,v)\}}$	$E_c \vdash e \ \triangleright S$
A	A	$A \in E_a$
$A\mathtt{(}e_1\mathtt{,}\ \ldots\mathtt{,}\ e_n\ \mathtt{)}$	$A_{v_1,\ldots,v_n}$	$A(V_1,\ldots,V_n) \in E_a$, $E_c \vdash e_i \ \triangleright v_i, 1 \le i \le n$, $v_i \in V_i, 1 \le i \le n$
$\mathtt{(}P\mathtt{)}$	$\mathtt{(}[\![P]\!]_E\mathtt{)}$	

Table 5: The syntax and semantics of agent expressions

The relabelling of parametrized or indexed labels is defined as two cases. Either the range of the relabelling function is a single element, or the domain and range are of equal sizes:

$$[\![\mathtt{[}l\mathtt{/}l'\mathtt{]}]\!]_E \stackrel{\text{def}}{=} \begin{cases} \mathtt{[}l_1\mathtt{/}l'_1,\ldots,l_1\mathtt{/}l'_n\mathtt{]} & \text{if } [\![\mathtt{\{}l_1\mathtt{\}}]\!]_E = \mathtt{\{}l_1\mathtt{\}} \text{ and } [\![\mathtt{\{}l_2\mathtt{\}}]\!]_E = \mathtt{\{}l'_1,\ldots,l'_n\mathtt{\}} \\ \mathtt{[}l_1\mathtt{/}l'_1,\ldots,l_n\mathtt{/}l'_n\mathtt{]} & \text{if } [\![\mathtt{\{}l\mathtt{\}}]\!]_E = \mathtt{\{}l_1,\ldots,l_n\mathtt{\}} \\ & \text{and } [\![\mathtt{\{}l'\mathtt{\}}]\!]_E = \mathtt{\{}l'_1,\ldots,l'_n\mathtt{\}} \end{cases}$$

All that remains to define is the syntax and translation of agent expressions. This is presented in tabular form, similar to that given by Milner. Table 5 has three columns. The first lists the syntactic forms of agents, the second lists the translation of each form, and the third lists the conditions which must be satisfied for the translation to be valid. For simplicity, the conditions that show whether an agent is well typed are included among those listed in the third column.

Conditions involving optional components should be ignored when such a component is omitted. Relabelling and restriction with multiple labels are defined as derived forms. Expression $P\backslash\{l_1 \ldots l_n\}$ is an abbreviation for $P\backslash\{l_1\}\ldots\backslash\{l_n\}$ and expression $P[l_1/l'_1 \ldots l_n/l'_n]$ is an abbreviation for $P[l_1/l'_1]\ldots[l_n/l'_n]$.

v	$\ulcorner v \urcorner$
integer i	ML integer format
boolean b	`true` or `false`
string s	s
$(v_1, \ldots, v_n)$	$\texttt{(}\ulcorner v_1 \urcorner, \ldots, \ulcorner v_n \urcorner\texttt{)}$
$[v_1, \ldots, v_n]$	$\texttt{[}\ulcorner v_1 \urcorner, \ldots, \ulcorner v_n \urcorner\texttt{]}$
$\{v_1, \ldots, v_n\}$	$\texttt{\{}\ulcorner v_1 \urcorner, \ldots, \ulcorner v_n \urcorner\texttt{\}}$ (where $v_i \prec v_{i+1}$ for $1 \leq i < n$)

Table 6: The printed form of values

Label	Printed Form
a	a
a^v	$a\texttt{\%\^{}}\ulcorner v \urcorner\texttt{\%}$
a_i	$a\texttt{\%}\ulcorner i \urcorner\texttt{\%}$
a_i^v	$a\texttt{\%}\ulcorner i \urcorner\texttt{\^{}}\ulcorner v \urcorner\texttt{\%}$

Table 7: The printed form of labels

5.1 Translator Output Format

The language definition gives the meaning of a value-passing agent as a basic CCS agent containing agents and actions with value subscripts and superscripts. The translator prints values, subscripts, and superscripts in plain ASCII characters using a particular representation. Table 6 defines the printed form $\ulcorner v \urcorner$ of value v. Table 7 shows the scheme for printing labels. Finally, the subscripted agent $A_{v_1,\ldots,v_n}$ is printed as

$$A\texttt{\%}\ulcorner v_1 \urcorner, \ldots, \ulcorner v_n \urcorner\texttt{\%}$$

6 Bibliographic Notes

A translator for the language described here is available on the World Wide Web via the following URL:

http://www.dcs.ed.ac.uk/home/cwb/

As for the Concurrency Workbench, the translator is available in pre-compiled form for Sun workstations, but can be used on any system for which Standard ML of New Jersey is available.

References

[1] Glenn Bruns. *Distributed Systems Analysis with CCS.* Prentice Hall, 1997.

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