A Language and Translator for Value-passing CCS

Glenn Bruns

December 5, 1994

Abstract

We define a language for value-passing CCS. The main additions of our language to the notation defined by Milner are: a concrete data language, having set, list, and tuple data types; simple, indexed, parameterized, and indexed/parameterized actions; and action declarations to associate sets of values with actions. This paper contains a definition of the language and a brief description of a translator to basic CCS.

1 Introduction

There are powerful and flexible tools (e.g., [5, 7]) for the analysis of CCS processes, but most only handle basic CCS. They do not handle *value-passing* CCS [10], in which actions and processes can be parameterized. We have found the features of value-passing CCS to be essential in the concise and natural modelling of systems [4, 3].

Milner's definition of value-passing CCS in [10] is given as a function mapping value-passing processes to basic ones. Although the mapping is intended only for the purpose of definition, it suggests that a translator could be written to obtain basic processes from value-passing ones. However, if one is interested in building such a translator, Milner's definition has problems. First, the set of values and value expressions is left unspecified. A concrete language for value-passing CCS must have some particular data language. Secondly, value-passing actions are all parameterized; the simple actions of basic CCS are not included. Also not included are indexed actions, which are used in many examples of [10]. Finally, values for parameterized input actions range over the entire (unspecified) set of values. In translating an input action, one wants to use the smallest set of values actually needed.

Our language for value-passing CCS treats all these issues. For values we provide finite set, tuple, and list data types, built from boolean, integer, and string constants. Simple, indexed, parameterized, and indexed/parameterized actions are provided. Action declarations allow a specific set of values to be associated with each action.

Our language is defined in an operational style, with the agent and value parts defined separately. The definition is organized similarly to the definition of Standard ML [9]. After defining our language, we briefly describe our translator to basic CCS.

2 Overview of the Language and its Definition

The main features of Milner's value-passing CCS are parameterized input actions $a(x).P$ (x is a value variable), parameterized output actions $\overline{a}(e).P$ (e a value expression), parameterized agents definitions $A(x) \stackrel{\mathrm{def}}{=} P$, and conditional statements `if` b `then` P_1 `else` P_2 (b a boolean expression). The definition leaves open the domain of values. For example, $a(x).P$ is defined as the agent that can perform $a(v)$, for *any* value v, and thereby become $P\{v/x\}$. The syntax of value expressions is also left open. For example, the syntax for expression e in $\overline{a}(e).P$ is not defined.

Our version of value-passing CCS has a concrete syntax for values. Furthermore, in our language it is possible to specify a set of values over which a action parameter can range. For example, the program

```
label a({1,2})
agent A(x:{1,2}) = 'a(x+1).0
```

defines agent A having a parameter x ranging over $1, 2$, which performs output action $\overline{a}(x+1)$.

The other main additions of our language are indexed actions and constant definitions. The following example illustrates these features:

```
value N = {1,2}
value B = {true,false}
label a_B(N)
label b_B
agent A(x:N) = b_y.'a_y(x+1).0
```

Here A first synchronizes on indexed action b_y, and then outputs $x+1$ on indexed action a_y.

Our language provide string constants as well as the integer and boolean constants already illustrated. From these one can built sets, lists, and tuples. For our final example we give the scheduler specification from [10].

```
(* Scheduler specification from page 115 of [10] *)

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})))
```

The agent and data parts of the language are quite independent. In the definition of agents we assume only the existence of value variables and value expressions. In the definition of data we make no assumptions at all about agents. Consequently the data part of the language could easily be modified or replaced. The definition of the language reflects this independence: the agent language is defined in sections 3-6, and then the data language is defined in section 7.

The definition of both parts of the language is given by an operational semantics [11]. This style was chosen over a translational semantics because we find it more direct and easier to read. One inference rule of the the definition gives the transitions possible for an agent P in an environment E:

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

Thus our definition replaces the implicit environment of Milner's definition with an explicit one. The updating of the environment is captured by the other main inference rule of the definition:

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

The detailed structure of the definition is based on the definition of Standard ML[9]. We have found that definition easy to use, and hope the similarity between the two definitions will be helpful.

x	$\in$	Var	value variables
a	$\in$	Name	CCS names
A	$\in$	Const	agent constants

Figure 1: Identifiers

		Program	programs
		Decl	declarations
		Param	agent parameters
P	$\in$	AExpr	agent expressions
		LName	long names
		Label	CCS labels
		ReLabFun	relabelling functions
		Lpair	label pairs
e	$\in$	VExpr	value expressions

Figure 2: Phrase Classes

3 Syntax

3.1 Reserved Words

The following are reserved words and may not be used as identifiers:

`agent` `comp` `const` `else` `if` `label` `sum` `t` `then`

3.2 Special Constants

All special constants are data-related and discussed in Section 7. The class of special constants is denoted by SCon; the variable *scon* is used to range over SCon.

3.3 Comments

A comment is any character string enclosed within the brackets `(*` and `*)`. Nested comments are not allowed.

3.4 Identifiers

The classes of identifiers are shown in Figure 1. We use the variables x, a, and A to range over Var, Name, and Const. 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 (`'`). 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.

3.5 Grammar

The phrase classes of the grammar are given in Figure 2. Unless otherwise indicated, the lowercase form of the class will serve as a variable over the class; for example, *program* will be used as a variable ranging over *Program*. The grammatical rules for all phrase classes but VExpr are given in Figures 3 and 4. The binding power of agent operators is as follows, given in order of decreasing binding power: restriction and relabelling, prefix, composition, summation. The `if` and `then` keywords of conditional expressions have lower binding power than all other agent operators.

The following conventions are used in presenting the rules of the grammar:

Program	::=	$decl\langle,declseq\rangle$	
decl	::=	`const` x= e	constant declaration
		`label` $lname\langle,\ lnameseq\rangle$	label declaration
		`agent` A= P	agent declaration
		`agent` $A(param\langle,paramseq\rangle) = P$	parameterized agent declaration
param	::=	x:e	

Figure 3: Grammar: Declarations

P	::=	`0`	inactive agent
		`t`.P	tau prefix
		$a\langle\_e\rangle.P$	simple input (with optional index)
		`'`$a\langle\_e\rangle.P$	simple output
		$a\langle\_e\rangle(x).P$	parameterized input
		`'`$a\langle\_e\rangle(e).P$¡	parameterized output
		P + P¡	summation
		P `|` P	composition
		$P\backslash\{labelseq\}$	restriction
		P`[`$relabfun$`]`	relabelling
		`if` e `then` P `else` P	conditional
		`sum`(x:e,P)	indexed summation
		`comp`(x:e,P)	indexed composition
		A	simple agent constant
		$A(e\ \langle,eseq\rangle)$	parameterized agent constant
		(P)	parenthesized agent expression
lname	::=	$a\langle\_e\rangle$	simple name
		$a\langle\_e\rangle(e)$	parameterized name
label	::=	$\langle$`'`$\rangle$ *lname*	
relabfun	::=	$lpair\ \langle,lpairseq\rangle$	
lpair	::=	*label* `/` *label*	

Figure 4: Grammar: Agent Expressions

- Angled brackets $\langle\rangle$ enclose optional phrases.
- If X is a phrase class, the class $Xseq$ *is defined as follows:*

xseq	*::=*		*(empty sequence)*
		x	*(singleton sequence)*
		$x_1, x_2, \ldots, x_n$	*(sequence, $n \geq 1$)*

4 Static Semantics

Here we informally describe the conditions that must be met for a value-passing CCS program to be considered well-formed:

no multiple declaration No value variable, name, or agent constant may be declared more than once.

declaration before use Value variables and names must be declared before they are used. Note that agent constants *can* be referred to before use.

label usage is consistent with declarations E.g., a label declared as parameterized may not be used as an indexed label.

output action values are consistent with declarations The value expression given with an output action must be consistent with the set given in the corresponding label declaration.

value expressions are consistent with operator types Value expressions must have types consistent with the types of the value operations, which are given in the next section.

agent parameter types are set-valued The value expressions given as types for agent parameters in agent declarations must be set-valued.

pi and sigma types are set-valued The value expressions given as types in sigma and pi agent expressions must be set-valued.

conditions are boolean-valued Value expressions given as conditions in conditional agent expressions must be boolean-valued.

Note that no constraints are placed on the usage of agent constants in agent expressions. For example, we do not require that an agent constant have a single parameter if it is so declared.

5 Semantics

The meaning of value-passing CCS expressions is given as a labelled transition system, as with basic CCS. The general form of a labelled transition system is (S, L, R), where S is a set of states, L is a set of labels, and $R \subseteq S \times L \times S$ is a labelled transition relation. We use $s \xrightarrow{l} s'$ as a shorthand for (s, l, s'). For value-passing CCS, a state is an agent expression P, and a label is a CCS action, possibly parameterized or indexed or both. The semantic rules given below define the transition relation between agent expressions relative to the environment given by program declarations.

5.1 Semantic Objects

The semantic objects are listed in Fig. 5. The simple objects are identifiers,already given in Fig. 1, and values. We use v to stand for a value of any type, and b, s, and t to stand for boolean, set, and tuple values, respectively.

Compound semantic objects are built up from simple semantic objects. To build a finite map from set A to set B, we write $A \xrightarrow{\text{fin}} B$. Finite maps can be written explicitly in the form $\{a_1 \mapsto b_1, \ldots, a_k \mapsto b_k\}$, $k \geq 0$. The finite map $f + g$, where f and g are themselves finite maps, is defined as:

$$(f + g)(a) \stackrel{\text{def}}{=} \text{if } a \in \text{Dom } g \text{ then } g(a) \text{ else } f(a)$$

where Dom gives the domain of a finite map.

5.2 Projection, Injection, and Modification

The operator "of" selects a component of a tuple. For example, "$NEnv$ of E" denotes the name-environment component of E. We will usually apply an environment without explicitly selecting a component, letting the argument indicate the appropriate component. For example, $E(a)$ is shorthand for $(NEnv \text{ of } E)(a)$.

The operator "in" injects a component into a tuple. For example, "$CEnv$ in Env" denotes the environment $(\{\}, CEnv, \{\})$.

The operator + modifies a finite map, as already defined. When modifying environments, we implicitly take unmentioned components to be empty. For example, $E + \{x \mapsto v\}$ is shorthand for $E + (\{\}, \{x \mapsto v\}, \{\})$.

$$
\begin{array}{rcl}
 & & \mathrm{Var} \\
 & & \mathrm{Name} \\
 & & \mathrm{Const} \\
\mathrm{v,b,s,t} & \in & \mathrm{Val} \\
\mathrm{E} & \in & \mathrm{Env} = \mathrm{NEnv} \times \mathrm{CEnv} \times \mathrm{AEnv} \\
\mathrm{NE} & \in & \mathrm{NEnv} = \mathrm{Name} \xrightarrow{\mathrm{fin}} \mathrm{NType} \\
\mathrm{CE} & \in & \mathrm{CEnv} = \mathrm{Var} \xrightarrow{\mathrm{fin}} \mathrm{Val} \\
\mathrm{AE} & \in & \mathrm{AEnv} = \mathrm{Const} \xrightarrow{\mathrm{fin}} \mathrm{AType} \\
\mathrm{nt} & \in & \mathrm{NType} = \{\mathrm{sim}\} \cup (\{\mathrm{idx}\} \times \mathrm{Val}) \cup (\{\mathrm{par}\} \times \mathrm{Val}) \cup (\{\mathrm{idxpar}\} \times \mathrm{Val} \times \mathrm{Val}) \\
 & & \mathrm{AType} = \cup_{k \geq 0}(\mathrm{AExpr} \times \mathrm{Var}^k) \\
 & & \mathrm{EName} = \mathrm{Name} \times \mathrm{NType} \\
\mathrm{l} & \in & \mathrm{Label} = \mathrm{EName} \cup \overline{\mathrm{EName}} \\
\alpha & \in & \mathrm{Action} = \mathrm{Label} \cup \{\tau\} \\
 & & \mathrm{RFunc} = \mathrm{Action} \rightarrow \mathrm{Action} \\
\mathrm{f} & \in & \mathrm{LMap} = \mathrm{Label} \xrightarrow{\mathrm{fin}} \mathrm{Label}
\end{array}
$$

Figure 5: Semantic Objects

5.3 Name Types

Name types are used in two ways. First, they are used to type names in label declarations. For example, the name a in the declaration `label a(1,2)` is typed as $(\mathrm{par}, \{1, 2\})$, indicating that it is an action with a parameter that ranges over values in $\{1, 2\}$. Secondly, they are used to type names in relabellings and restrictions. For example, the name a in the relabelling $[a(2)/b]$ is typed as $(\mathrm{par}, 2)$, indicating that it is a parameterized action with value 2.

The operator $\mathrm{Names}(a, nt)$ gives the set of extended names possible for a assuming set-valued name type nt. For example, $\mathrm{Names}(a, (\mathrm{idx}, \{1, 2\})) = \{(a, (\mathrm{idx}, 1)), (a(\mathrm{idx}, 2))\}$.

The relation $\sim$ is an equivalence relation on name types. Two name types are equivalent, $nt_1 \sim nt_2$, if they are both either simple, indexed, etc., and if their associated values are equal. The definition of equality over values is given in section 7.

5.4 Relabelling Functions

Relabelling functions are total functions that map actions to actions. However, CCS syntax only allows for the description of finite mappings from labels to labels. Such a finite mapping f is extended to a relabelling function f_T according to the following definition:

$$
f_T(\alpha) \stackrel{\mathrm{def}}{=} \begin{cases} \overline{f(\alpha)} & \text{if } \alpha \in \mathrm{Dom}(f) \text{ and } \alpha \in \mathrm{EName} \\ f(\alpha) & \text{if } \alpha \in \mathrm{Dom}(f) \text{ and } \alpha \in \overline{\mathrm{EName}} \\ \alpha & \text{otherwise} \end{cases}
$$

5.5 Syntactic Substitutions

By $P\{v/x\}$ we mean the expression that results from substituting all free occurrences of variable x in P by value v. (An occurrence of a variable x in an agent expression P is *bound* if it is in a part of P with the form $a(x).P'$, $a\_e(x).P'$, $sum(x : e, P')$, or $comp(x : e, P')$; otherwise it is *free*.)

5.6 Set Operations

The set expression $\{x \in S \mid p(x)\}$ stands for the members of set S satisfying the predicate p. The expression S^k stands for the set of sequences of length k that can be built from set S.

5.7 Inference Rules

The infererence rules take two forms. The first shows how declarations or value expressions are evaluated in an environment E:

$$E \vdash phrase \Rightarrow A$$

where A is an updated environment or a value. The second shows the transitions that are possible for an agent expression in an environment:

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

An agent expression P of a program *prog* can perform transition $P \xrightarrow{\alpha} P'$ just if *prog* gives an environment E

$$(\{\}, \{\}, \{\}) \vdash prog \Rightarrow E$$

such that

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

Programs $\boxed{E \vdash program \Rightarrow E'}$

$$\frac{E \vdash decl \Rightarrow E_1 \quad \langle E \vdash declseq \Rightarrow E_2 \rangle}{E \vdash decl\langle, declseq\rangle \Rightarrow E_1 \langle + E_2 \rangle}$$

Declarations $\boxed{E \vdash decl \Rightarrow E'}$

$$\frac{E \vdash e \Rightarrow v}{E \vdash \texttt{const}\ \ x = e \Rightarrow \{x \mapsto v\} \text{ in Env}} \qquad (1)$$

$$\frac{lnameseq = (ln_1, \ldots, ln_k) \quad E \vdash ln_i \Rightarrow (n_i, nt_i)\ (1 \leq i \leq k)}{E \vdash \texttt{label}\ \ lnameseq \Rightarrow \{n_1 \mapsto nt_1, \ldots, n_k \mapsto nt_k\}\ in\ Env}$$

$$E \vdash \texttt{agent}\ \ A = P \Rightarrow \{A \mapsto (P, ())\} \text{ in Env}$$

$$\frac{param\langle, paramseq\rangle = x_1 : e_1, \ldots, x_k : e_k}{E \vdash \texttt{agent}\ \ A(param\langle, paramseq\rangle) = P \Rightarrow \{A \mapsto (P, (x_1, \ldots, x_k))\} \text{ in Env}} \qquad (2)$$

Comments:
(1) Note that expressions in declarations are evaluated immediately, giving static binding and requiring that constants be declared before use.
(2) The expression e_i, which gives the type of parameter i, is not recorded in the agent environment because agent parameter values are not checked.

Agent Expressions $\boxed{E \vdash P \xrightarrow{\alpha} P'}$

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

$$\frac{\langle E \vdash e \Rightarrow v\rangle}{E \vdash a\langle\_e\rangle.P \xrightarrow{a\langle v\rangle} P} \qquad \frac{\langle E \vdash e \Rightarrow v\rangle}{E \vdash \texttt{'}a\langle\_e\rangle.P \xrightarrow{\overline{a}\langle v\rangle} P}$$

$$\frac{E(a) = (\langle\text{idx}\rangle\text{par}\langle, s_1\rangle, s_2) \quad v_2 \in s_2 \quad \langle E \vdash e \Rightarrow v_1\rangle}{E \vdash a\langle\_e\rangle(x).P \xrightarrow{a(v_2)\langle v_1\rangle} P\{v_2/x\}} \qquad \frac{E \vdash e_2 \Rightarrow v_2 \quad \langle E \vdash e_1 \Rightarrow v_1\rangle}{E \vdash \texttt{'}a\langle\_e_1\rangle(e_2).P \xrightarrow{\overline{a}(v_2)\langle v_1\rangle} P\{v_2/x\}} \qquad (3)$$

$$\frac{E \vdash P_1 \xrightarrow{\alpha} P_1'}{E \vdash P_1 + P_2 \xrightarrow{\alpha} P_1'} \qquad \frac{E \vdash P_2 \xrightarrow{\alpha} P_2'}{E \vdash P_1 + P_2 \xrightarrow{\alpha} P_2'}$$

$$\frac{E \vdash P_1 \xrightarrow{\alpha} P_1'}{E \vdash P_1|P_2 \xrightarrow{\alpha} P_1'|P_2} \qquad \frac{E \vdash P_2 \xrightarrow{\alpha} P_2'}{E \vdash P_1|P_2 \xrightarrow{\alpha} P_1|P_2'} \qquad \frac{E \vdash P_1 \xrightarrow{l} P_1' \quad E \vdash P_2 \xrightarrow{\overline{l}} P_2'}{E \vdash P_1|P_2 \xrightarrow{\tau} P'|P_2'}$$

$$\frac{E \vdash P \xrightarrow{\alpha} P' \quad \alpha, \overline{\alpha} \notin \{labelseq\}}{E \vdash P\backslash\{labelseq\} \xrightarrow{\alpha} P\backslash\{labelseq\}}$$

$$\frac{E \vdash P \xrightarrow{\alpha} P' \quad E \vdash relabfun \Rightarrow f}{E \vdash P\,\texttt{[}relabfun\texttt{]} \xrightarrow{f_T(\alpha)} P'\,\texttt{[}relabfun\texttt{]}} \tag{4}$$

$$\frac{E \vdash e \Rightarrow true \quad E \vdash P_1 \xrightarrow{\alpha} P_1'}{E \vdash \texttt{if}\ e\ \texttt{then}\ P_1\ \texttt{else}\ P_2 \xrightarrow{\alpha} P_1'} \qquad \frac{E \vdash e \Rightarrow false \quad E \vdash P_2 \xrightarrow{\alpha} P_2'}{E \vdash \texttt{if}\ e\ \texttt{then}\ P_1\ \texttt{else}\ P_2 \xrightarrow{\alpha} P_2'}$$

$$\frac{E \vdash e \Rightarrow \{v_1, \ldots, v_n\} \quad E \vdash P\{v_1/x\} + \cdots + P\{v_n/x\} \xrightarrow{\alpha} P'}{E \vdash \texttt{sum}(x{:}e, P) \xrightarrow{\alpha} P'}$$

$$\frac{E \vdash e \Rightarrow \{v_1, \ldots, v_n\} \quad E \vdash P\{v_1/x\} \mid \cdots \mid P\{v_n/x\} \xrightarrow{\alpha} P'}{E \vdash \texttt{comp}(x{:}e, P) \xrightarrow{\alpha} P'}$$

$$\frac{E(A) = (P, ()) \quad E \vdash P \xrightarrow{\alpha} P'}{E \vdash A \xrightarrow{\alpha} P'}$$

$$\frac{E(A) = (P, (x_1, \ldots, x_n)) \quad e\langle,eseq\rangle = e_1, \ldots, e_n \quad E \vdash e_i \Rightarrow v_i \quad E \vdash P\{v_1/x_1, \ldots, v_n/x_n\} \xrightarrow{\alpha} P'}{E \vdash A(e\langle,eseq\rangle) \xrightarrow{\alpha} P'}$$

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

Comments:
(3) See Section 5.5.
(4) See Section 5.4.

Long Names $\boxed{E \vdash lname \Rightarrow EName}$

$$E \vdash a \Rightarrow (a, \mathrm{sim})$$

$$\frac{E \vdash e \Rightarrow v}{E \vdash a\_e \Rightarrow (a, (\mathrm{idx}, v))}$$

$$\frac{E \vdash e_2 \Rightarrow v_2 \quad \langle E \vdash e_1 \Rightarrow v_1\rangle}{E \vdash a\langle\_e_1\rangle(e_2) \Rightarrow \{a, (\langle \mathrm{idx}\rangle \mathrm{par}\langle, v_1\rangle, v_2)\}}$$

`if` e `then` P	`if` e `then` P `else 0`

Figure 6: Derived Forms

Labels $\boxed{E \vdash \mathit{label} \Rightarrow \mathit{label}}$

$$\frac{E \vdash \mathit{lname} \Rightarrow l}{E \vdash \langle \text{'} \rangle \mathit{lname} \Rightarrow \overset{\langle - \rangle}{l}}$$

Relabelling Functions $\boxed{E \vdash \mathit{relabfun} \Rightarrow f}$

$$\frac{E \vdash \mathit{lpair} \Rightarrow f \quad \langle E \vdash \mathit{lpairseq} \Rightarrow g \rangle}{E \vdash \mathit{lpair}\ \langle , \mathit{lpairseq} \rangle \Rightarrow f \langle + g \rangle}$$

Label Pairs $\boxed{E \vdash \mathit{lpair} \Rightarrow f}$

$$\frac{E \vdash \mathit{label}_i \Rightarrow (a_i, \mathit{nt}_i)\ (i \in \{1,2\}) \quad E(a_2) \neq \mathrm{sim}}{E \vdash \mathit{label}_1 / \mathit{label}_2 \Rightarrow \{(a_2, \mathit{nt}_2) \mapsto (a_1, \mathit{nt}_1)\}} \tag{5}$$

$$\frac{\begin{array}{c} E \vdash \mathit{label}_i \Rightarrow (a_i, \mathit{nt}_i)\ (i \in \{1,2\}) \quad E(a_1) \neq \mathrm{sim} \quad E(a_2) = \mathrm{sim} \\ \mathrm{Names}(a_2, E(a_2)) = \{l_1, \ldots, l_n\} \end{array}}{E \vdash \mathit{label}_1 / \mathit{label}_2 \Rightarrow \{l_1 \mapsto (a_1, \mathit{nt}_1), \ldots, l_n \mapsto (a_1, \mathit{nt}_1)\}} \tag{6}$$

$$\frac{\begin{array}{c} E \vdash \mathit{label}_i \Rightarrow (a_i, \mathit{nt}_i)\ (i \in \{1,2\}) \quad E(a_1) = \mathrm{sim} \quad E(a_2) = \mathrm{sim} \\ \mathrm{Names}(a_1, E(a_1)) = \{(a_1, \mathit{nt}'_1), \ldots, (a_1, \mathit{nt}'_n)\} \end{array}}{E \vdash \mathit{label}_1 / \mathit{label}_2 \Rightarrow \{(a_2, \mathit{nt}'_1) \mapsto (a_1, \mathit{nt}'_1), \ldots, (a_2, \mathit{nt}'_n) \mapsto (a_1, \mathit{nt}'_n)\}} \tag{7}$$

Comments: (5 – 7) The rules for label pairs differ slightly from those given in [10], to account for the presence of simple, parameterized, and indexed actions. Informally, the rules allow a name appearing simply (i.e. without parameter or index) to stand for the set of all actions in the name's type whenever possible. The first rule handles the case where the second name does not appear simply; here the first name cannot be expanded to all names in its type. The second rule handles the case where both names appear simply and both have the same type; here both names are expanded. The third rule handles the case where the second name appears simply but the first does not; here the second name is expanded.

See section 5.3 for details on name types.

6 Derived Forms

For convenience, a derived syntactic form in Figure 6. This form should be regarded as a rewriting rule that is applied repeatedly until no further applications are possible.

7 The Data Language

The following data language is a simple one of finite sets, tuples, and lists, built from boolean, integer, and string constants. Notice that the finiteness of data values is guaranteed by the syntax of the language.

The data language introduces the following additional reserved words:

```
and  append  cons  diff  false  hd  len  member  mod  not  or  pow
prod  seqs  size  tl  true  union  update  where
```

UnOp	unary value operators
BinOp	binary value operators
FName	function names

Figure 7: New Phrase Classes

e	::=	scon	special constant
		x	variable
		$\mathit{unop}\ \ e$	unary operator
		$e\mathit{binop}\ e$	infix binary operator
		$\mathit{fname}(e\langle,\mathit{eseq}\rangle\ )$	function application
		`{` eseq `}`	enumerated set
		`{` e `|` e `where` $x : e$`}`	set comprehension
		`<<` eseq `>>`	tuple
		e`#`e	tuple selection
		`[` eseq `]`	list
		(e)	parenthesized expression

Figure 8: Grammar: Value Expressions

There are three new special constants. A boolean constant is the string `true` or the string `false`. An integer constant is a non-empty string of decimal digits, optionally preceded by a plus sign (+) or a minus sign (−). A string constant is any string of printable ASCII characters between quotes (`" "`).

Three new phrase classes are introduced; they are given in Figure 7. Classes UnOp and BinOp are defined as {`not`} and $\{+, -,$ `mod`$, =, <>, <, <=, >=, >,$ `and`$,$ `or`$\}$ respectively, where the symbols in BinOp are shown in order of decreasing binding power. The operator `not` has higher binding power then `and` but lower binding power than the relational operators. Class FName is defined as the union of the set function names {`prod`,`pow`,`union`,`diff`,`size`}, the tuple function name {`update`}, and the list function names {`seqs`,`cons`,`hd`,`tl`,`append`,`len`}.

The syntax of value expressions is given by the grammatical rules of Figure 8.

Value Expressions $\boxed{E \vdash \mathit{vepxr} \Rightarrow v}$

$$E \vdash \mathit{scon} \Rightarrow \mathrm{val}(\mathit{scon})$$

$$\frac{E(x) = v}{E \vdash x \Rightarrow v}$$

$$\frac{\begin{array}{c}E \vdash e_3 \Rightarrow \{v_1, \ldots, v_k\} \\ E + \{x \mapsto v_i\} \vdash e_1 \Rightarrow v_i' \quad E + \{x \mapsto v_i\} \vdash e_2 \Rightarrow b_i \quad b_i \in \{\mathit{true}, \mathit{false}\}\ (1 \leq i \leq k)\end{array}}{E \vdash \{\ e \mid e_1\ \texttt{where}\ x : e_2\ \} \Rightarrow \{v_i' \mid b_i\}}$$

$$\frac{E \vdash e_1 \Rightarrow (v_1, \ldots, v_k) \quad E \vdash e_2 \Rightarrow i}{E \vdash e_1\texttt{\#}e_2 \Rightarrow v_i}$$

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

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

`{` x`:` e_1 `|` e_2 `}`	`{` x`|` e_2 `where` x`:` e_1 `}`
`{` e_1 `|` x`:` e_2 `}`	`{` e_1 `|` *true* `where` x`:` e_2 `}`

Figure 9: Derived Forms

$$\frac{E \vdash e \Rightarrow v}{E \vdash (e) \Rightarrow v}$$

Semantic rules are not given for most value expressions, since their meaning is standard:

- The boolean functions `not`, `and`, and `or` return the result of the usual boolean operators.
- The integer functions $+, -$, and x `mod` y return the result of the usual integer operations.
- `{` $x_1, \ldots, x_n$ `}` returns the set $\{x_1, \ldots, x_n\}$.
- `prod`$(s_1, \ldots, s_n)$ returns the set product $s_1 \times \ldots \times s_n = \{(x_1, \ldots, x_n) \mid x_i \in s_i\}$.
- `pow`(s) returns the power set of s.
- `union`(s_1, s_2) returns the set $s_1 \cup s_2$.
- `diff`(s_1, s_2) returns the set $s_1 \setminus s_2$.
- `size`(s) returns the number of elements in set s
- `member`(x, s) returns *true* if x is a member of set s, else *false*.
- `<<` $x_1, \ldots, x_n$ `>>` returns the tuple $(x_1, \ldots, x_n)$.
- `[` $x_1, \ldots, x_n$ `]` returns the list $[x_1, \ldots, x_n]$.
- `cons`(x, l) returns the list l with x added as its first element.
- `hd`(l) returns the first element of list l.
- `tl`(l) returns the list l less its first element.
- `append`(l_1, l_2) returns the list containing the elements of l_1 followed by the elements of l_2.
- `len`(l) returns the number of elements in l.

To define the relational operators $(<>, <, <=, >, =>)$, we use 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. So, for example, $1 \prec \{\}$. The following rules define the order relation within each type. For booleans, `false` $\prec$ `true`. Integers are ordered by the relation $<$. Strings, tuples, and sequences are ordered lexicographically. Sets are ordered as sequences by ordering the elements they contain and then removing duplicates.

Finally, two derived forms for value expressions are given in Figure 9.

8 The Translator

We have written an ML program that takes a value-passing CCS program as input, and produces a basic CCS program as output. The translation can be regarded as consisting of two steps. In the first step, function $\mathcal{T}$ uses information in the constant and label declarations to produce a *variable-free* program.

$$\mathcal{T} : Program_{vp} \rightarrow Program_{vf}$$

$Program_{vp}$ is the class of all value-passing programs (given previously simply as $Program$), and $Program_{vf}$ is the class of variable-free programs. By variable-free, we mean programs that contain no value variables or constants. Note that variable-free programs contain only agent declarations, not label or constant declarations.

A variable-free program may contain terms like $P(1,2)$ and $a_{\{1\}}$, and is therefore not a basic CCS program. In the second step of the translation, function $\mathcal{E}$ encodes variable-free phrases as basic CCS phrases.

$$\mathcal{E} : CCS_{vf} \rightarrow CCS_b$$

CCS_{vf} denotes the variable-free subset of the full value-passing language, and CCS_b denotes basic CCS. The complete translation of a value-passing CCS program can now be written as $\mathcal{E}(\mathcal{T}(program))$.

Although we have not proved our translator correct with respect to the language definition, we can state the required correctness condition once we introduce a bit more notation. For the two forms of inference rules in our semantics we will write $E \vdash_{vp} phrase \Rightarrow E'$ and $E \vdash_{vp} P \xrightarrow{\alpha} P'$. Although an environment of agent declarations are given only implicitly in Milner's definition of basic CCS, we could derive from his definition corresponding inference rules, which we will write as $E \vdash_b phrase \Rightarrow E'$ and $E \vdash_b P \xrightarrow{\alpha} P'$. The expressions in these two forms are restricted to those of basic CCS.

The translator is *correct* if the following condition holds for all $program \in Program_{vp}$, and for all $P, P' \in P_{vf}$:

> If $() \vdash_{vp} program \Rightarrow E$ and $() \vdash_b \mathcal{E}(\mathcal{T}(program)) \Rightarrow F$,
> then $(E \vdash_{vp} P \xrightarrow{\alpha} P'$ iff $F \vdash_b \mathcal{E}(P) \xrightarrow{\mathcal{E}(\alpha)} \mathcal{E}(P'))$.

It is important for a user of our translator to know what our encoding function does, so that basic CCS terms produced by the translator can be related to value-passing CCS terms in the user program. For conciseness, we present here only the "interesting" parts of the encoding. The parts left undefined do no real work. For example: $\mathcal{E}(P_1 + P_2) = \mathcal{E}(P_1) + \mathcal{E}(P_1)$.

$$\begin{aligned} \mathcal{E}(P(v_1,\ldots,v_n)) &= P_{v_1,\ldots,v_n} \\ \mathcal{E}(a_v) &= a_v \\ \mathcal{E}(a(v)) &= a^v \\ \mathcal{E}(a_u(v)) &= a_u^v \end{aligned}$$

Unfortunately, the Concurrency Workbench is not able to accept all names and constants of this form. The translator performs a final stage of encoding, putting basic CCS terms into Concurrency Workbench format. This function is given in a technical note describing operation of the translator.

9 Conclusions

This is a major revision of an earlier report. Adopting the style of ML's definition has improved the organisation and helped uncover some areas of ambiguity. Although experience suggests that our language is easy to understand and use, the lack of a definition facility for functions and local values limits its flexibility. A pattern-matching facility for agent parameters would also be useful.

There are other process languages with richer data sub-languages. The data part of Lotos [2], for example, is based on the algebraic specification language ACT ONE [6]. However, the introduction of verification tools for full Lotos may have been slowed by its relative complexity. Another process language, μCRL [8], has a process part based on ACP [1] and an algebraic data language based on equational axioms. In μCRL, infinite data types and processes can be defined, but a subset, *effective* μCRL, has been identified for which all specifications have finite behaviour. Our language, on the other hand, influenced perhaps too strongly by our translator, allows the definition of only finite values, but infinite-state processes.

By implementing the language with a translator we have taken advantage of the Concurrency Workbench, which provides a wide range of features for simulating and analysing basic CCS agents. A disadvantage of translation to basic CCS is that the generated agent definitions can be long and strange in appearance, making them awkward to work with. A potential solution is to incorporate the translation directly into the Workbench. Alternatively, the Workbench could be re-designed to handle value-passing agents directly. In this way proof techniques for value-passing agents could be added to the Workbench. Another important drawback of the translator is that it has yet to be proved correct with respect to the semantics given here.

A separate report on the translator might be written one day, but presently the language is not rich enough to have a really independent existence. For example, only finite data types are provided, and there is no function definition facility. On the other hand, experience has shown that the simplicity of the language makes it easy to learn and use, and that it allows for the convenient modelling of many real-world systems.

Acknowledgements

Thanks to Zvi Schreiber, Stuart Anderson, and Andy Galloway for comments and corrections, and to Faron Moller for suggestions about the language syntax.

References

[1] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Computation*, 60:109–137, 1984.

[2] T. Bolognesi and E. Brinksma. Introduction to the specification language LOTOS. In van Eijk, Vissars, and Diaz, editors, *The Formal Description Technique LOTOS*. Elsevier, 1989.

[3] Glenn Bruns. A case study in safety-critical design. In *Proceedings of CAV '92*. Springer Verlag, 1992. Lecture Notes in Computer Science, 354.

[4] Glenn Bruns and Stuart Anderson. The formalization and analysis of a communications protocol. *Formal Aspects of Computing*, 6:92–112, 1994.

[5] 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, University of Edinburgh, 1989.

[6] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1*. Springer-Verlag, 1985.

[7] Jens C. Godskesen, Kim G. Larsen, and Michael Zeeberg. TAV (Tools for Automatic Verification) users manual. Technical Report R 89-19, Institute for Electronic Systems, Department of Mathematics and Computer Science, University of Aalborg, Denmark, August 1989.

[8] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. Technical Report CS-R9076, Centre for Mathematics and Computer Science, CWI, 1990.

[9] Robert Harper, Robin Milner, and Mads Tofte. The definition of standard ML: Version 3. Technical Report ECS-LFCS-89-81, Laboratory for Foundations of Computer Science, University of Edinburgh, 1989.

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

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