A Framework for Representing Parameterised Processes*

Ed Harcourt **Paweł Pączkowski** **K.V.S. Prasad**

Department of Computing Science
Chalmers University of Technology
412 96 Göteborg, Sweden

e-mail: {harcourt,pawel,prasad}@cs.chalmers.se
fax: +46 31 16 56 55

November 15, 1995

Abstract

We describe a faithful representation of value-passing recursive parametric CCS processes in Alf, an implementation of Martin-Löf's constructive type theory. The representation is interesting because we borrow as much as possible from Alf including the domain of value and state expressions and the ability to evaluate them. Usually substitution of either channel values for channel variables and processes for process variables play a necessary role in the semantics. However, substitution is also borrowed from Alf by using higher-order functions. The main importance of this representation is that it allows us to borrow Alf's off-the-shelf theorems about data types and provides a uniform setting for doing various kinds machine assisted proofs, such as bisimulation proofs, equational reasoning, verification of Hennessy-Milner logic formulas.

1 Introduction

Machine checked proofs of CCS processes require finite representations of the processes. However, for infinite value and state domains, the traditional semantics of CCS reduces a value-passing process to a potentially infinite sum of processes and a parametric process (a process parameterised by a state variable) to a potentially infinite family of processes [Mil89]. Our goal is to find a representation of value-passing parametric CCS processes suitable for use with Alf, a proof checker for Martin Löf's constructive type theory [CNSvS95]. While we use Alf as our implementation language it is important to realize that the problems we address are not created by Alf but would arise, to some extent, in all logical frameworks such as Coq, HOL, LEGO, Nuprl, Isabelle, or PVS.

One of the best reasons for using Alf is that it lets us combine various proof techniques into a single system. First, our representation lets us use Alf to do proofs about data domains (both channel and states). For example, without introducing any of our own equivalences, Alf can equate the process $\texttt{out!}(x+y).p$ with the process $\texttt{out!}(y+x).p$ by using an already proved theorem about he commutativity of addition. Another important source of Alf's power is its ability to do induction over any inductively defined data type. We can also introduce our own equivalences. For example we have introduced three different relations in Alf; bisimulation, an equational proof system, and Hennessy-Milner logic that relates processes with formula from a modal logic.

*Funding from the Swedish Government agency TFR, and ESPRIT BRA CONCUR2

For the present paper we view Alf as the typed lambda calculus extended with dependent types. So that we can write "useful" processes we allow both the domains of values and states to be infinite (*e.g.,* integers, lists, etc.). The aim of this paper is not to do to equivalence proofs about processes in type theory — first we need an adequate way of expressing value-passing parametric processes.

However, in preliminary experiments we have been successful in doing bisimulation proofs and also model checking infinite state processes against formulae of Hennessy-Milner logic. In particular, having represented CBS, the calculus of broadcasting systems, [Pra9x, Pra93] in Alf and defined strong bisimulation, Prasad has shown that strong bisimulation is a congruence relation with respect to the operators of CBS. Also, in a prioritised version of CBS [Pra94] Jørgen Andersen has verified in Alf that a distributed sorting algorithm (written in CBS) is correct w.r.t. an abstract specification of sorting (which is expressed as an Alf type). In these experiments Alf's ability to do proofs by induction was invaluable. Both of these experiments are yet to be reported.

Motivation. If we want machine representation, we often need to worry about technical aspects that are usually ignored (*e.g.,* the language of data/state expressions, value and process substitution, renaming data and process variables, etc.). Such details are understandably elided in normal discourse as they are usually well understood by the general readership. As we will see in our representation, employing a suitably powerful framework such as Alf allows us to address aspects such as substitution, variable renaming, and data expressions by *appropriately* borrowing from the framework. Care must be taken, however, not to borrow too much as the representation can become unsound. This borrowing is similar to using *Higher-Order Abstract Syntax* [EP88, DFH95].

Our representation is general because we take the value and state domains from Alf by parameterising the representation on a type variable for each, greatly simplifying our syntax of processes as we no longer specify a syntax and semantics for channel and state expressions. In this manner, channel and state expressions are taken from Alf and CCS variables are identified with Alf variables. Also, substitution of values for these variables is also borrowed using higher-order functions. However, care must be taken to ensure that this borrowing is done in a sound manner. For example, it is not sound in Alf to naively use Alf's recursion for process recursion as this creates infinite objects, which can lead to an inconsistent theory [Coq93]. This is because in Alf all functions must terminate and processes usually do not.

Subsequently, we attempt to implement process recursion using an explicit process fix-point construct $\mathbf{fix}(X = E)$. The operational rule for recursion involves a process substitution $E\{P/X\}$ which we would also like to borrow from Alf through function application. However, this too is unsound as the normal type of the fix-point construct is impredicative and the syntax of processes will not be inductively defined in the normal way. These considerations lead us to a "correct" solution which will be to implement parametric defining equations.

We call our representation a *framework* because it is parameterised on two type variables, one for the type of values that may be passed on channels and the other for the type of state. This paper discusses the Alf representation for CCS. We have also implemented CBS, the Calculus of Broadcasting Systems [Pra93].

Outline of the paper. The rest of the paper is outlined as follows. Section 2 introduces the concrete syntax and operational semantics of value-passing CCS that we implement and discusses how functions model value-passing. Section 3 discusses process recursion with a fix-point construct and the problems created when implemented in Alf. Section 4 provides a brief introduction to Alf, and shows how the syntax of CCS is represented in type theory. With Alf now at hand, we show how defining equations are implemented soundly, using Alf functions. Section 5 shows how

the operational semantics of CCS is implemented as an inductively defined Alf relation. Section 6 discusses how we can already do some proofs about value-passing CCS processes in Alf. Section 7 outlines what an "interpreter" for CCS is and how one is written as an Alf function. Related work is briefly discussed in Section 8. Section 9 concludes.

2 CCS Preliminaries

CCS is a formalism where processes are terms of a simple language of process constructors. The behaviour of a process is usually given in terms of the possible transitions of the process. Transitions of the form $p \xrightarrow{\alpha} q$ express process p's ability to perform some action α and evolve to a new process q.

The syntax of CCS we study is given by the following grammar:

$$p ::= ch!e_v.p \mid ch?x.p \mid p \mid p \mid p + p \mid p\backslash\ell \mid p[f] \mid \mathbf{0} \mid A(e_s) \mid \textbf{if } b \textbf{ then } p \quad (1)$$

where ch comes from a countable set of channel names *Chan*, e_v from a language of channel expressions, x from a countable set of channel variables *Var*, A from a set of process variables, e_s from a language of state expressions, and b from a language of boolean expressions. (Precisely speaking, e_v,e_s, b belong to a many-sorted language of expressions which has sorts of channel, state and boolean expressions.) A term $A(e_s)$ refers to a process constant A parameterised by e_s. Associated with A is a definition of the form $A(z) \stackrel{\text{def}}{=} p$ where z is a state variable and $A(e_s)$ may occur in p. To model multiple process parameters we allow the state be a tuple type. The function f is a channel relabelling function and ℓ is a set of channel names. Let $\mathbf{P}$ be the set of processes generated by (1). *Pure CCS* is the sub-calculus without value-passing, the conditional **if**, and process parameters. We include only the **if-then** construct as the **if-then-else** can be encoded using summation (see [Mil89]). We then use the **if-then-else** construct freely.

Channel variables range over a set of channel values *Value* and closed channel expressions evaluate to a value in *Value*. State variables range over a set of state values *State* and closed state expressions evaluate to a value in *State*. Thus, we assume some evaluation scheme for expressions and we will identify closed expressions with their values. The same applies to boolean expressions which evaluate to **true** or **false**. The set of actions Act_τ is defined by expression $(Chan \times \{!\} \times Value) \cup (Chan \times \{?\} \times Var) \cup \{\tau\}$. Example members of Act_τ are $ch?x$, $ch!5$, and τ (without using tuple notation).

The operational semantics of CCS is given in Figure 1 and describes a transition relation $\longrightarrow \subseteq \mathbf{P} \times Act_\tau \times \mathbf{P}$. We only consider closed processes. That is, processes where all variables in channel and state expressions are bound by an input prefix.

Channel Variable Substitution. The rule for reading a channel, **In**, says that the process $ch?x.p$ may receive a value v on channel ch and become $p[v/x]$ which means v is substituted for all free occurrences of x in p. The value substitution $p[v/x]$ is standard and defined as in [CY94]. The input prefixing rule describes a potentially infinite number of possible transitions, one for each possible value v. Under this interpretation, assuming $v \in \mathbb{N}$, the process $ch?x.p$ is equivalent to an infinite summation of processes in pure CCS $ch?0.p[0/x] + ch?1.p[1/x] + ch?2.p[2/x] + \cdots$, usually written $\sum_{v \in \mathbb{N}} ch?x.p[v/x]$. However, this creates infinite terms in CCS, which we need to avoid.

Moreover, we would like to avoid implementing our own substitution used in the rule **In** because implementing explicit channel variable substitution (that avoids possible capture of the channel variables) commits us to implementing our own language of data expressions. We avoid channel

$$\textbf{Out}\quad \frac{}{\alpha.p \xrightarrow{\alpha} p}\quad \alpha = \tau \text{ or } \alpha = ch!v \qquad \textbf{In}\quad \frac{}{ch?x.p \xrightarrow{ch?v} p[v/x]}$$

$$\textbf{SumL}\quad \frac{p \xrightarrow{\alpha} p'}{p + q \xrightarrow{\alpha} p'} \qquad \textbf{SumR}\quad \frac{q \xrightarrow{\alpha} q'}{p + q \xrightarrow{\alpha} q'}$$

$$\textbf{ParL}\quad \frac{p \xrightarrow{\alpha} p'}{p \mid q \xrightarrow{\alpha} p' \mid q} \qquad \textbf{ParR}\quad \frac{q \xrightarrow{\alpha} q'}{p \mid q \xrightarrow{\alpha} p \mid q'} \qquad \textbf{ParT}\quad \frac{p \xrightarrow{a!v} p' \quad q \xrightarrow{a?v} q'}{p \mid q \xrightarrow{\tau} p' \mid q'}$$

$$\textbf{Rel}^{\dagger}\quad \frac{p \xrightarrow{\alpha} p'}{p[f] \xrightarrow{f(\alpha)} p'[f]} \qquad \textbf{Res}^{\ddagger}\quad \frac{p \xrightarrow{\alpha} p'}{p\backslash\ell \xrightarrow{\alpha} p'\backslash\ell}\quad \alpha \in S$$

$$\textbf{Def}\quad \frac{p[v/z] \xrightarrow{\alpha} p'}{A(v) \xrightarrow{\alpha} p'}\quad A(z) \stackrel{\text{def}}{=} p \qquad \textbf{Cond}\quad \frac{p \xrightarrow{\alpha} p'}{(\textbf{if true then } p) \xrightarrow{\alpha} p'}$$

† The function $f : Act \to Act$ only changes channel names and leaves τ and values unaltered.

‡ S is a list of channels and $\alpha \in S$ means α uses a channel in S.

Figure 1: Operational semantics of CCS.

variable substitution and infinite summation by letting CCS channel variables be represented by Alf variables using lambda abstraction. We replace rule **In** with the following rule $\textbf{In}_\lambda$

$$\textbf{In}_\lambda\quad \frac{}{ch?\lambda x.p \xrightarrow{ch?v} (\lambda x.p)v} \tag{2}$$

where x may occur free in p and the input prefixing operator ? has the type $Chan \times (Value \to \mathbf{P})$. Open processes are represented using lambda abstraction and variable substitution $p[v/x]$ is implemented using function application. Using Alf in this manner we identify Alf variables with channel variables and channel expressions with expressions of some Alf's type. We justify this in section 5.

3 Recursion

Consider CCS without process constants and conditionals and assume we have a type of process in Alf called $\mathsf{Proc}(V)$ where V is the type of channel values. It is tempting to try and borrow both recursion and the conditional from Alf by representing a recursive process p by a recursive function p of type $\mathsf{Proc}(V)$. For example, consider a one-place buffer cell that loses even integers.

$$Cell \stackrel{\text{def}}{=} \texttt{in?}x.\textbf{if } even\ x \textbf{ then } Cell \textbf{ else } \texttt{out!}x.Cell \tag{3}$$

In Alf, *Cell* is a recursive function of type $\mathsf{Proc}(V)$ and the **if**-statement is also Alf's.

A parameterised process can be represented as function from a state of type S to a process. For example, equation 4 defines a simple read/write register that stores natural numbers.

$$Reg(y) \stackrel{\text{def}}{=} \texttt{write}?x.Reg(x) + \texttt{read}!y.Reg(y) \tag{4}$$

In pure CCS the usual semantics of *Reg* is that it represents an infinite family of processes $\{Reg_i \mid i \in \mathbb{N}\}$ which we represent functionally by making *Reg* a recursive function of type $\mathbb{N} \to \mathsf{Proc}(\mathbb{N})$. The case without parameters in the beginning of this section is really just the special case where *Cell* is of type $\mathsf{Unit} \to \mathsf{Proc}(V)$ where Unit is the unit data type with one value unit.

If all we wanted from a process definition was to determine its transitions then borrowing Alf's recursion would work (see [Pet94] for a related theorem about implementing a CBS interpreter for Haskell). However, our main goal is to be able to machine check proofs about processes in Alf and in Alf, every recursive function must be defined in terms of a structurally smaller argument. That is, the recursion must be guaranteed to terminate. However, as can be seen in equations 3 and 4, processes are often non-terminating and using Alf's recursion as process recursion creates "infinite objects" which, in Alf (and constructive type theory in general), are not sound [Coq93].

3.1 A Fix-point Construct

The obvious way to avoid using Alf's recursion thereby inhibiting the creation of infinite objects is to implement recursion explicitly. We can do this either by implementing defining equations or by extending the syntax of processes with a fix-point constructor $\mathbf{fix}(X = E)$, where X is a process constant that may occur in a process E. First, we consider the case of adding an explicit fix-point construct.

With the fix-point construct we need to extend the semantics with a transition rule for **fix** which, intuitively, "unwinds the recursive definition once".

$$\textbf{Rec} \quad \frac{E\{\mathbf{fix}(X = E)/X\} \stackrel{\alpha}{\longrightarrow} E'}{\mathbf{fix}(X = E) \stackrel{\alpha}{\longrightarrow} E'}$$

This rule requires a form of process substitution $E\{P/X\}$. Keeping to our goal of trying to borrow as much from Alf as possible, we would like to identify process variables with Alf variables and borrow Alf's substitution for process substitution (as we did in the case for channel variables). In such case, in the rule **Rec**, E would be a function from a process to a new "unwound" process; $E : \mathbf{P} \to \mathbf{P}$ and **fix** would have the usual type of a fixed-point operator; $(\mathbf{P} \to \mathbf{P}) \to \mathbf{P}$. Using function application instead of substitution, the rule **Rec** would become

$$\textbf{Rec'} \quad \frac{E(\mathbf{fix}\ E) \stackrel{\alpha}{\longrightarrow} E'}{\mathbf{fix}\ E \stackrel{\alpha}{\longrightarrow} E'}$$

This avoids creating infinite objects, but we have a new problem. The type of **fix** is ill-defined in Alf (and other implementations of constructive type theory such as Coq and LEGO) because the first occurrence of $\mathbf{P}$ in $(\mathbf{P} \to \mathbf{P}) \to \mathbf{P}$ occurs negatively and the set of processes $\mathsf{Proc}(V)$ would not be inductively defined in the normal sense. The situation is no better for parameterised processes. If $\mathcal{S}$ is the type of a state then the fix-point operator would have the type

$$\mathbf{fix}_{\mathcal{S}} : ((\mathcal{S} \to \mathbf{P}) \to (\mathcal{S} \to \mathbf{P})) \to (\mathcal{S} \to \mathbf{P})$$

which is also ill-defined. If we insist on implementing the fix-point construct then the only way out would be to implement our own process substitution which, though not difficult, does add the usual

complexity of variable capture and renaming and would require proofs of correctness. Implementing process constants will be easier. We should also explain that types of the form $(\mathbf{P} \to \mathbf{P}) \to \mathbf{P}$ are sound in Isabelle [Pau94] and Elf [Pfe91] but they are weaker logics which don't provide induction.

4 Representation in Alf

Before we proceed with our solution for representing recursion we informally present Alf through a few examples on the natural numbers (This short discussion of type theory is taken from [CNSvS95]). We view Alf as the typed lambda calculus extended with dependent types. Alf is a proof editor and all of the Alf code below appears as it does on the screen. There are two kinds of terms in Alf — types and objects, which are inductively defined sets and functions, respectively.

Natural Numbers. The type (set) of natural numbers is introduced with the definition $\mathsf{N} \in \mathsf{Set}$, $0 \in \mathsf{N}$, and $\mathsf{s} \in (\mathsf{N})\mathsf{N}$. Here the type $(\mathsf{N})\mathsf{N}$ is Alf notation for the function type $\mathsf{N} \to \mathsf{N}$. An object (*i.e.*, function) Add that adds two natural numbers and a set (or type) Le representing a relation for $\leq$ are defined by the following (which is how they actually appear in the Alf proof editor).

$$\begin{array}{l}\mathsf{Add} \in (\mathsf{N};\mathsf{N})\mathsf{N} \\ \quad \mathsf{Add}(0,y) = y \\ \quad \mathsf{Add}(\mathsf{s}(x),y) = \mathsf{s}(\mathsf{Add}(x,y))\end{array} \qquad \begin{array}{l}\mathsf{Le} \in (m,n \in \mathsf{N})\mathsf{Set} \\ \quad \mathsf{leO} \in (n \in \mathsf{N})\mathsf{Le}(0,n) \\ \quad \mathsf{leS} \in (m,n \in \mathsf{N};\mathsf{Le}(m,n))\mathsf{Le}(\mathsf{s}(m),\mathsf{s}(n))\end{array}$$

The definition of Le follows the normal relational definition. Notice the use of the dependent function type. In Le the constructor leO is a function whose result type $\mathsf{Le}(0,n)$ depends on the object n. This allows us to define Le as would be done in an operational semantics and hints at how the operational semantics of CCS directly will be specified in Alf. That is, leO encodes the rule $\frac{}{0 \leq n}$ and leS encodes $\frac{n < m}{\mathsf{succ}(n) \leq \mathsf{succ}(m)}$.

Types as Propositions. A function in Alf can be viewed as a proof of a proposition in first-order logic where the type of the function represents the proposition to be proved. For example, the following function is a proof that Le is transitive. In the definition the first three parameters $i, j, k \in \mathsf{N}$ have been hidden along with the declarations of m and n in the constructors for Le. This feature of Alf makes proofs more readable.

$$\begin{array}{l}\mathsf{LeTrans} \in (\mathsf{Le}(i,j);\mathsf{Le}(j,k))\mathsf{Le}(i,k) \\ \quad \mathsf{LeTrans}(\mathsf{leO},h) = \mathsf{leO} \\ \quad \mathsf{LeTrans}(\mathsf{leS}(h_2),\mathsf{leS}(h)) = \mathsf{leS}(i,k,\mathsf{LeTrans}(h_2,h))\end{array}$$

The type of the function represents the proposition to be proved and the body of the function represents the proof. The function is recursive, which represents a proof by induction.

4.1 Representing Finite Processes

We are now in a position to present CCS in Alf. We assume a library of predefined types for natural numbers, lists, and the unit type Nat, List, and Unit.

First, we define a type of processes $\mathsf{Proc}(V,S)$ parameterised on two type variables V and S which represent the types of channel and state expressions, respectively. That is, we abstract from the implementation of channel, state, and boolean expressions that were used in the definition of

Construct name	CCS Syntax	Alf Syntax
Inactive process	$\phi(\mathbf{0})$	$\mathsf{Nil}(V,S)$
Input prefixing	$\phi(ch?x.p)$	$\mathsf{Input}(V,S,ch,\lambda x.\phi(p))$
Conditional	**if** b **then** p	$\mathsf{If}(V,S,b,\phi(p))$
Output prefixing	$\phi(ch!e_v.p)$	$\mathsf{Output}(V,S,ch,\ e_v,\phi(p))$
Tau prefixing	$\phi(\tau.p)$	$\mathsf{OutTau}(V,S,\phi(p))$
Summation	$\phi(p+q)$	$\mathsf{Sum}(V,S,\phi(p),\phi(q))$
Parallel Composition	$\phi(p\|q)$	$\mathsf{Par}(V,S,\phi(p),\phi(q))$
Restriction[a]	$\phi(p\backslash\ell)$	$\mathsf{Res}(V,S,\phi(p),\ell')$
Process Constants[b]	$A(e_s)$	$\mathsf{Var}(V,S,\mathsf{A}(e_s))$
Relabelling[c]	$\phi(p[f])$	$\mathsf{Rel}(V,S,\phi(p),f')$

[a] Where ℓ' is ℓ represented as an Alf list.
[b] Where $\mathsf{A} \in (E)S$ and E is the type of e_s.
[c] Where f' is an obvious Alf representation of f.

Table 1: Translation function ϕ maps CCS to Alf type $\mathsf{Proc}(V,S)$.

P assuming them to be simply Alf expressions. There is a constructor in Proc for each operator in CCS. For example, the parallelism constructor Par has type

$$\mathsf{Par} \in (V, S \in \mathsf{Set}; \mathsf{Proc}(V,S); \mathsf{Proc}(V,S))\mathsf{Proc}(V,S)$$

The type of the input prefixing constructor is

$$\mathsf{Input} \in (V, S \in \mathsf{Set}; \mathsf{Channel}; (V)\mathsf{Proc}(V,S))\mathsf{Proc}(V,S)$$

where Channel is a set of channel names isomorphic to the natural numbers. In examples, we continue to use named channels such as `read`, `write`, `in`, and `out` and assume they are given different values in Channel.

Proc is an inductively defined set definable in any typed functional language such as Haskell or ML. Table 1 describes a function ϕ that shows how a term in the concrete syntax **P** maps to a term of type $\mathsf{Proc}(V,S)$ where V and S are the types of channel values and state; $\phi : \mathbf{P} \to \mathsf{Proc}(V,S)$. By an inductive argument it is easy to show that every term in **P** maps to a term in Proc. However, the converse is not true. There are processes in Proc not in **P**. Let the domain of channel values be the type of channels Channel (*i.e.,* $\mathsf{Proc}(\mathsf{Channel},S)$). Now we can write a process that can pass channels.

$$\mathsf{Q} \in \mathsf{Proc}(\mathsf{Channel},\mathsf{Unit})$$
$$\mathsf{Q} = \mathsf{Input}(\mathsf{Channel},\mathsf{Unit},\mathtt{chan},\lambda y.\mathsf{Output}(\mathsf{Channel},\mathsf{Unit},y,y,\mathsf{Nil}(\mathsf{Channel},\mathsf{Unit})))$$

The process Q reads a value into y on channel `chan` and then writes y on channel y. It is clear that process Q has no representation in **P**.

4.2 Parametric Defining Equations

To avoid creating an ill-defined **fix** construct we could implement our own form of explicit process substitution. It is important that this process substitution be able to handle parameterised processes over infinite state domains (such as the read/write register example above). However,

instead of trying to implement a "fix-point construct with state", it will be easier to implement process constants with parameters where each constant has a declaration of the form $A(z) \stackrel{\text{def}}{=} p$.

Consider first a single defining equation with one parameter. For example, a counter can be defined by $C(x) \stackrel{\text{def}}{=} \texttt{out}!x.C(x+1)$. We can represent this functionally. As we are considering a process definition of only one constant, the name "C" plays no role in the process definition and can be eliminated. We replace each occurrence of constant $C(e_s)$ by a new process constructor $\mathsf{Var}(e_s)$. The process $C(x)$ above is $\lambda x.\texttt{out}!x.\mathsf{Var}(x+1)$ which we would write in Alf as

$$\boxed{\mathsf{C}(x) = \mathsf{Output}(\mathsf{Nat}, \mathsf{Nat}, \texttt{out}, x, \mathsf{Var}(\mathsf{Nat}, \mathsf{Nat}, \mathsf{s}(x)))}$$

As another example, recall the definition of $Reg(x)$ in equation (4). Since there is only one defining equation, to represent this in Alf we need to define an Alf object of type $(\mathsf{Nat})\mathsf{Proc}(\mathsf{Nat}, \mathsf{Nat})$ where the ports `read` and `write` are assigned unique channel names.

$$\begin{array}{l} \mathsf{Reg} \in (\mathsf{Nat})\mathsf{Proc}(\mathsf{Nat}, \mathsf{Nat}) \\ \mathsf{Reg}(x) = \mathsf{Sum}(\mathsf{Nat}, \mathsf{Nat}, \mathsf{Input}(\mathsf{Nat}, \mathsf{Nat}, \texttt{read}, \lambda y.\mathsf{Var}(\mathsf{Nat}, \mathsf{Nat}, y)), \\ \qquad\qquad \mathsf{Output}(\mathsf{Nat}, \mathsf{Nat}, \texttt{write}, x, \mathsf{Var}(\mathsf{Nat}, \mathsf{Nat}, x))) \end{array}$$

This definition may seem a little verbose because of all the type information that is included. However, in Alf, most type information is inferred automatically by unification and does not have to be explicitly entered by the user. Also, we can hide much of the redundant type information.

Multiple sets of defining equations are created by using disjoint unions of states $S = S_1 \uplus \cdots \uplus S_n$. A process parameterised on several state variables is modelled by letting S_i be a product of component substates $S_i = S_{i1} \times \cdots \times S_{in}$. For example, consider the following process P that inputs two integers and outputs their sum (forgetting for the moment that these three equations can easily be rewritten as one).

$$\begin{array}{rcl} P & \stackrel{\text{def}}{=} & \texttt{in}(x)?Q(x) \\ Q(x) & \stackrel{\text{def}}{=} & \texttt{in}(y)?Sum(x,y) \\ Sum(x,y) & \stackrel{\text{def}}{=} & \texttt{out}!(x+y).P \end{array}$$

To represent this in Alf, define a set State in Alf to have the constructors $\mathsf{P} \in \mathsf{State}$, $\mathsf{Q} \in (\mathsf{Nat})\mathsf{State}$, and $\mathsf{Sum} \in (\mathsf{Nat};\mathsf{Nat})\mathsf{State}$. The following Alf object represents the above equation system.

$$\begin{array}{l} env \in (\mathsf{State})\mathsf{Proc}(\mathsf{Nat},\mathsf{State}) \\ env(\mathsf{P}) = \mathsf{Input}(\mathsf{Nat}, \mathsf{State}, \mathsf{read}, \lambda x.\mathsf{Var}(\mathsf{Nat}, \mathsf{State}, \mathsf{Q}(x))) \\ env(\mathsf{Q}(x)) = \mathsf{Input}(\mathsf{Nat}, \mathsf{State}, \mathsf{read}, \lambda y.\mathsf{Var}(\mathsf{Nat}, \mathsf{State}, \mathsf{Sum}(x,y))) \\ env(\mathsf{Sum}(x,y)) = \mathsf{Output}(\mathsf{Nat}, \mathsf{State}, \mathsf{write}, \mathsf{add}(x,y), \mathsf{Var}(\mathsf{Nat}, \mathsf{State}, \mathsf{P})) \end{array}$$

In general, a set of defining equations $A_i(z) \stackrel{\text{def}}{=} p_i$ is encoded in an Alf type State and a function env of type $\mathsf{State} \to \mathsf{Proc}(V,\mathsf{State})$, called an environment, that are defined as follows: **1)** State has constructor A_i of type t_i for each process constant A_i with a parameter of type t_i, **2)** env is defined by cases: $env(\mathsf{A}_i(z)) = \phi(p_i)$. Under this encoding of defining equations, rule **Def** of the operational semantics is replaced by the following rule in our implementation of CCS semantics, which we soon define in Alf.

$$\textbf{Var} \quad \frac{env(\mathsf{A}(v)) \stackrel{\alpha}{\longrightarrow} p'}{A(v) \stackrel{\alpha}{\longrightarrow} p'}$$

Notice that the function application $env(v)$ takes the place of process substitution.

5 Transition Relation in Alf

The transition relation is represented by a set Step with the following type.

$$\boxed{\mathsf{Step} \in (V, S \in \mathsf{Set}; (S)\mathsf{Proc}(V,S); \mathsf{Proc}(V,S); \mathsf{Act}(V); \mathsf{Proc}(V,S))\mathsf{Set}}$$

where $\mathsf{Act}(V)$ is a set of actions with constructors In,$\mathsf{Out} \in (\mathsf{Channel}; V)\mathsf{Act}(V)$ and $\mathsf{Tau} \in \mathsf{Act}(V)$. The constructors for Step are direct translations of the rules of the operational semantics, except for input prefixing and process constants, which are translations of the rules $\mathbf{In}_\lambda$ and **Var**.

$$\boxed{\begin{array}{lcl} \mathsf{InPref} & \in & (V, S \in \mathsf{Set}; \mathit{chan} \in \mathsf{Channel}; v \in V; f \in (V)\mathsf{Proc}(V,S)) \\ & & \mathsf{Step}(V, S, \mathsf{Input}(V, S, \mathit{chan}, f), \mathsf{In}(V, \mathit{chan}, v), f(v)) \end{array}}$$

The rule for process constants must be parameterised on an environment.

$$\boxed{\begin{array}{lcl} \mathsf{Pvar} & \in & (V, S \in \mathsf{Set}; a \in \mathsf{Act}(V); p \in \mathsf{Proc}(V,S); s \in S; \mathit{env} \in (S)\mathsf{Proc}(V,S); \\ & & \mathsf{Step}(V, S, \mathit{env}(s), a, p))\mathsf{Step}(V, S, \mathsf{Var}(V, S, s), a, p) \end{array}}$$

We show just two more constructors for Step corresponding to the rules **ParT** and **Cond** of Figure 1.

$$\boxed{\begin{array}{lcl} \mathsf{ParTau} & \in & (V, S \in \mathsf{Set}; \mathit{chan} \in \mathsf{Channel}; v \in V; p, p', q, q' \in \mathsf{Proc}(V,S); \\ & & \mathsf{Step}(V, S, p, \mathsf{Out}(V, \mathit{chan}, v), p'); \mathsf{Step}(V, S, q, \mathsf{In}(V, \mathit{chan}, v), q')) \\ & & \mathsf{Step}(V, S, \mathsf{Par}(V, S, p, q), \mathsf{Tau}(V), \mathsf{Par}(V, S, p', q')) \\ \mathsf{IfT} & \in & (V, S \in \mathsf{Set}; a \in \mathsf{Act}(V); p, p' \in \mathsf{Proc}(V,S); \mathsf{Step}(V, S, p, a, p')) \\ & & \mathsf{Step}(V, S, \mathsf{If}(V, S, \mathsf{true}, p), a, p') \end{array}}$$

Notice that since the relation $\longrightarrow$ is not recursive because of input prefixing [Vaa93], Step could not be defined in a standard functional language (unless we change the semantics to a "late" operational semantics [HL95]).

We are now in a position to show the soundness of the representation which we will call *adequacy*. First, we need a theorem that justifies using Alf's function application for channel variable substitution. We use the equality sign below to relate Alf terms not distinguishable by Alf. This equality is defined using the notion of Alf's canonical forms and boils down to $\alpha\beta\eta$-reduction for closed terms.

Proposition 1 [Substitution Lemma] If $p \in \mathbf{P}$, $v \in \mathit{Value}$ ($v \in \mathit{State}$) and x is a channel (state) variable then $\phi(p[v/x]) = (\lambda x.\phi(p))v$.

Proof First note that substitution on channel and state expressions (which are just Alf expressions) is defined by $e[v/x] = (\lambda x.e)v$. Then, the proposition follows by induction on the structure of p. For example, for the output prefix we have

$$(\lambda x.\phi(ch!e.p))v = (\lambda x.\mathsf{Output}(ch, e, \phi(p))v = \mathsf{Output}(ch, (\lambda x.e)v, (\lambda x.\phi(p))v)$$

By induction hypothesis and the assumed representation of substitution on expressions, the latter term can be rewritten as $\mathsf{Output}(ch, e[v/x], \phi(p[v/x]))$, which equals $\phi((ch!e.p)[v/x])$. □

Let us extend the translating function ϕ to actions: $\phi(ch!v) = \mathsf{Out}(ch, v)$, $\phi(ch?v) = \mathsf{In}(ch, v)$, $\phi(\tau) = \mathsf{Tau}$.

Proposition 2 [Adequacy] Given a set of defining equations and our encoding of them in an environment ρ

(a) for every closed $p, q \in \mathbf{P}$ *if* $p \xrightarrow{\alpha} q$ *then* $\mathsf{Step}(V, S, \rho, \phi(p), \phi(\alpha), \phi(q))$

(b) for every closed $p \in \mathbf{P}$, $q' \in \mathsf{Proc}(V, S)$, $\alpha' \in \mathsf{Act}(V)$ *if* $\mathsf{Step}(V, S, \rho, \phi(p), \alpha', q')$ *then* $p \xrightarrow{\alpha} q$ for some α, q such that $\phi(\alpha) = \alpha'$ and $\phi(q) = q'$

Proof (a) We proceed by induction on the derivation of $p \xrightarrow{\alpha} q$. We examine here a few cases, the inductive argument is straightforward for the remaining ones. To simplify the notation we will omit the obvious parameters V, S and ρ.

If $ch?x.p \xrightarrow{ch?v} p[v/x]$ then we have to construct $\mathsf{Step}(\mathsf{Input}(ch, \lambda x.\phi(p)), \mathsf{In}(ch, v), \phi(p[v/x]))$. By definition of Step we have $\mathsf{Step}(\mathsf{Input}(ch, \lambda x.\phi(p)), \mathsf{In}(ch, v), (\lambda x.\phi(p))v)$ and the argument is completed by Proposition 1.

If $A(v) \xrightarrow{\alpha} p'$ for some process constant defined by equation $A(z) \stackrel{\text{def}}{=} p$ then, by induction hypothesis, we have

$$\mathsf{Step}(\phi(p[v/z]), \phi(\alpha), \phi(p')). \tag{5}$$

We have to construct $\mathsf{Step}(\mathsf{Var}(\mathsf{A}(v)), \phi(\alpha), \phi(p'))$. According to our encoding of defining equations for processes $\mathsf{A}(v)$ is a (closed) expression of type S (states) and $\rho(\mathsf{A}(v)) = (\lambda z.\phi(p))v$. Using Proposition 1 we get $\rho(\mathsf{A}(v)) = \phi(p[v/z])$. By (5) we have $\mathsf{Step}(\rho(\mathsf{A}(v)), \phi(\alpha), \phi(p'))$, which allows us to construct $\mathsf{Step}(\mathsf{Var}(\mathsf{A}(v)), \phi(\alpha), \phi(p'))$ as required.

If $(\mathbf{if\ true\ then}\ p) \xrightarrow{\alpha} p'$ then, by induction hypothesis, we have $\mathsf{Step}(\phi(p), \phi(\alpha), \phi(p'))$ and constructing the required $\mathsf{Step}(\mathsf{If}(\mathsf{true}, \phi(p)), \phi(\alpha), \phi(p'))$ is straightforward from definition of Step.

(b) We proceed by induction on the derivation of $\mathsf{Step}(p', \alpha', q')$, where $p' = \phi(p)$ for some closed $p \in \mathbf{P}$. The base cases are input, output and τ prefixes. For example, consider the case of the input prefix, i.e. $\mathsf{Step}(\mathsf{Input}(ch, \lambda x.\phi(p)), \mathsf{In}(ch, v), (\lambda x.\phi(p))v)$, where $\mathsf{Input}(ch, \lambda x.\phi(p)) = \phi(ch?x.p)$. But $ch?x.p \xrightarrow{ch?v} p[v/x]$ and $\phi(p[v/x]) = (\lambda x.\phi(p))v$ by Proposition 1 .

For the inductive step, let us again examine just two example cases. Assume $\mathsf{Step}(\mathsf{Var}(\mathsf{A}(v)), \alpha', q')$, where $\mathsf{Var}(\mathsf{A}(v)) = \phi(A(v))$ for some process constant defined by equation $A(z) \stackrel{\text{def}}{=} p$. According to definition of Step we must have had $\mathsf{Step}(\rho(\mathsf{A}(v)), \alpha', q')$. Just as in (a) above $\rho(\mathsf{A}(v)) = \phi(p[v/z])$, so we can use the induction hypothesis to obtain $p[v/z] \xrightarrow{\alpha} q$ for some α and q such that $\phi(\alpha) = \alpha'$ and $\phi(q) = q'$. Now, we can derive $A(v) \xrightarrow{\alpha} q$ as required.

Assume $\mathsf{Step}(\mathsf{If}(b, \phi(p)), \alpha', q')$, where $\mathsf{If}(b, \phi(p)) = \phi(\mathbf{if}\ b\ \mathbf{then}\ p)$. Then we must have had $b = \mathsf{true}$ and $\mathsf{Step}(\phi(p), \alpha', q')$. By induction hypothesis, $p \xrightarrow{\alpha} q$ for some α and q such that $\phi(\alpha) = \alpha'$ and $\phi(q) = q'$, hence $(\mathbf{if}\ b\ \mathbf{then}\ p) \xrightarrow{\alpha} q$. □

Notice that the proposition avoids the problem of terms that are in Proc but not in $\mathbf{P}$ by insisting that all terms in Proc be translated by ϕ.

6 Some Machine Checked Proofs

We can, in Alf, define the notion of a bisimulation and do proofs about bisimulation. Similarly, we can define a logic such as the modal μ-calculus and the satisfaction relation between terms in the logic and processes. As we mentioned in the introduction, we have been experimenting with this. However, without defining our own equivalences it is interesting to examine the kind of proofs can be done in Alf directly.

Transition Proofs. We can do some simple proofs about the transitions a process may have. That is, we can show that certain items are in the Step relation. For example, in the register example it should be true that for any x and y, $Reg(x)$ can input y and evolve to $\mathsf{Var}(y)$. That is, we should be able to build an object with the Alf type:

$$\mathsf{proof} \in (n, m \in \mathsf{Nat})\ \mathsf{Step}(\mathsf{Nat}, \mathsf{Nat}, \mathsf{Reg}(n), \mathsf{In}(\mathsf{Nat}, \mathtt{read}, m), \mathsf{Var}(\mathsf{Nat}, \mathsf{Nat}, m)) \tag{6}$$

which is easily provable in Alf.

Id Proofs. It is easy to define in Alf a relation Id, where $\mathsf{Id} \in (A \in \mathsf{Set}; A; A)$, that relates objects indistinguishable by Alf (equivalent up to Alf's $\alpha\beta\eta$-reduction, in the case of closed terms). The relation Id plays an important role in reasoning about data in our processes. For example, we would like CCS processes to be equivalent up to renaming of data variables and indeed Id provides this. For example, $\mathtt{in}?x\mathtt{out}!(x).\mathbf{0}$ and $\mathtt{in}?y.\mathtt{out}!y.\mathbf{0}$ are in Id. In fact, the following congruence holds.

Proposition 3 [Id Congruence] Id is a congruence with respect to all of the operators of CCS.

This congruence gives us the capability to use off-the-shelf theorems about Alf types (*e.g.*, natural numbers, lists, etc.) to reason about data within our processes. For example, for any x and y the process $\mathtt{out}!(x+y).p$ should be equal, in any reasonable behavioural process equivalence, to the term $\mathtt{out}!(y+x).p$ by using a lemma about the commutativity of $+$. In Alf it is straightforward to show these terms are in Id. This leads us to the next result that if two processes are identical (that is, in Id) then they are strongly bisimilar.

Proposition 4 [Id preserves bisimulation] For all $p, q \in \mathsf{Proc}$ if $\mathsf{Id}(p, q)$ then $p \sim q$.

Equational Reasoning. It is straightforward to define an equality relation by encoding the algebraic laws of CCS. To do this we define a set Equals of the type

$$\boxed{\mathsf{Equals} \in (V, S \in \mathsf{Set}; \mathsf{Proc}(V, S); \mathsf{Proc}(V, S))\mathsf{Set}}$$

where each algebraic law of CCS is encoded as a constructor in Equals. For example, the law for commutativity of $+$ is encoded as

$$\boxed{\mathsf{plus}_{\mathsf{comm}} \in (V, S \in \mathsf{Set}; p, q \in \mathsf{Proc}(V, S))\mathsf{Equals}(V, S, \mathsf{Sum}(p, q), \mathsf{Sum}(q, p))}$$

Alf's type inferencing and unification provide a substantial amount of assistance when doing equational proofs.

Induction Proofs. Alf's ability to allow the user to do proofs by induction is an important source of Alf's power. In Alf, every inductively defined set comes equipped with an induction principle generated by Alf. For example, the two inductively defined sets described so far are Proc and Step. Induction over Proc amounts to structural induction over the syntax of processes and induction over Step is induction over the structure of the derivation (or what Milner calls "transition induction" [Mil89, page 58 section 2.1]). As an example of induction over the structure of a derivation consider the set Sort which describes the syntactic sort of a process (the set of channel names that appear in a process' syntax). The type of Sort is

$$\boxed{\mathsf{Sort} \in (V, S \in \mathsf{Set}; env \in (S)\mathsf{Proc}(V, S); \mathsf{Proc}(V, S); \mathsf{List}(\mathsf{Channel}))\mathsf{Set}}$$

and the introduction rules for Sort are straightforward to implement. We can prove a theorem about sorts [Mil89, proposition 2.1 page 58] that says that if $p \xrightarrow{\alpha} p'$ and $(\alpha \neq \tau)$ then $\alpha \in \mathsf{Sort}(p)$. The corresponding statement in Alf is

$$\begin{array}{l}\mathsf{SortLemma} \in (V, S \in \mathsf{Set}; a \in \mathsf{Act}(V); env \in (S)\mathsf{Proc}(V,S); p, p' \in \mathsf{Proc}(V,S); \\ \qquad\qquad sp \in \mathsf{List}(\mathsf{Channel}); nottau \in (\mathsf{Id}(a, \mathsf{Tau}))\mathsf{Empty}; \mathsf{Sort}(p, sp); \mathsf{Step}(env, p, a, p')) \\ \quad \mathsf{InList}(GetChan(a, nottau), sp)\end{array}$$

$\mathsf{InList}(x, l)$ is a predicate read $x \in l$ and Empty is the null set which corresponds to false so $(\mathsf{Id}(a, \mathsf{Tau}))\mathsf{Empty}$ means $a \neq \tau$. *GetChan* is a function that returns the channel name of an action but can only do so when it knows that the action is not τ (all Alf functions must be total). Some of the type parameters in the arguments to SortLemma have been hidden for readability. SortLemma is a function whose type represents the proposition and whose body represents a proof. When constructing the proof (function) Alf allows the user to do pattern matching on any argument and choosing the argument corresponding to the proof for $\mathsf{Step}(env, p, a, p')$ Alf generates a proof obligation for each of the constructors in Step.

Hennessy-Milner Logic. As a more interesting example, we have encoded the syntax and satisfaction relation of Hennessy-Milner logic [Sti93] and do some proofs about value passing processes by induction (proofs that elude finite model checkers for two reasons, **1)** infinite value domains and **2)** infinite state due to recursion and parallelism in the process). For example, consider the following counter in CCS (slipping back into normal CCS syntax)

$$\begin{array}{rcl} Count(0) & \stackrel{\text{def}}{=} & \mathtt{up}.Count(1) \\ Count(i+1) & \stackrel{\text{def}}{=} & \mathtt{up}.Count(i+2) + \mathtt{down}.Count(i) \end{array}$$

Using induction on n we can show that $\forall i.Count(i) \models [\mathtt{up}]^n \langle \mathtt{down} \rangle^n \mathbf{true}$ (this proof has been carried out in Alf). We have not yet extended HML with the recursive operators μ and ν as we can frequently use the defined operators $[\mathtt{up}]^n$ and $\langle \mathtt{up} \rangle^n$ and induction on n.

Alf provides us with many data types to do induction on and indeed some data types that we normally would not consider. For example, since the sort of a process is defined as an inductively defined set Sort we can now do induction on the sort of a process. In the next section we will encounter another inductively defined set Guarded which describes the set of guarded processes. This will allow us to use induction over the "proof of guardedness" of a process (or more intuitively, induction over the syntax of a guarded process).

7 A CCS Interpreter

The relation Step described in the previous section is defined for all process in Proc, even unguarded processes. Consequently Step is potentially infinitely branching on a process p if p is unguarded. On infinite value domains, Step is also infinitely branching because of the rule for input prefixing (as discussed in section 2). Because of these two problems Step is not effective.

The question arises, under what circumstances can we write a function Step_f that, given a process, returns a finite set (list) of the transitions of the process. If we use the "late" operational semantics we avoid the problem caused by input prefixing. And if we only allow Step_f to operate on guarded processes we avoid the problem of unguarded recursion. But to do this we must either have a syntax of guarded processes (which is difficult to write) or Step_f must also take a *proof* that the process is guarded. The problem is that if p is unguarded then $\mathsf{Step}_f(p)$ will not terminate

and the list of possible transitions may be infinite. Remember, however, that we can only write terminating function in Alf.

To see this, in Alf Step_f would be a function from a process and an environment to a list of transitions

$$\mathsf{Step}_f \in (V, S \in \mathsf{Set}; \mathsf{Proc}(V,S); (\mathsf{S})\mathsf{Proc}(V,S))\mathsf{List}(\mathsf{Transition}(V))$$

Here, a transition is either an input, output, or τ transition and the set $\mathsf{Transition}$ is introduced in Alf by

$$\begin{array}{l}\mathsf{Transition} \in (V, S \in \mathsf{Set})\mathsf{Set} \\ \quad \mathsf{In}_t \in (V, S \in \mathsf{Set}; \mathsf{Channel}; (V)\mathsf{Proc}(V,S))\mathsf{Transition}(V,S) \\ \quad \mathsf{Out}_t \in (V \in \mathsf{Set}; \mathsf{Channel}; V; \mathsf{Proc}(V,S))\mathsf{Transition}(V,S) \\ \quad \mathsf{Tau}_t \in (V, S \in \mathsf{Set}; \mathsf{Proc}(V,S))\mathsf{Transition}(V,S)\end{array}$$

In_t is essentially a pair; a channel and a function that evaluates to the derivative process when the input value is known. Out_t is a triple; a channel, value, and the derivative process. Tau_t is a singleton, just the derivative process.

Normally, Step_f would be defined recursively on the structure of the process. The problem arises when we try to determine the transitions of a process constant. The definition of Step_f on process constants would be

$$\mathsf{Step}_f(\mathsf{Var}(s)) = \mathsf{Step}_f(env(s))$$

However, Alf rejects this as Step_f is called recursively on an argument that is not structurally smaller and implying that Step_f is potentially non-terminating. We must, then, find another parameter which we can use to structure the recursive calls.

Guardedness. The list of transitions of a process p is guaranteed to be finite (assuming a "late" semantics) if p is guarded. This suggests that if we pass Step_f only guarded processes then Step_f is guaranteed to terminate. But this is not quite enough. We also need to pass Step_f a *proof* that the process is guarded. Step_f is then defined, not on the structure of a process, but on the structure of the proof of guardedness.

It is straightforward to define the set of guarded processes in an environment.

$$\mathsf{Guarded} \in (V, S \in \mathsf{Set}; (S)\mathsf{Proc}(V,S); \mathsf{Proc}(V,S))\mathsf{Set}$$

Now we define Step_f so that it takes a proof of guardedness.

$$\mathsf{Step}_f \in (V, S \in \mathsf{Set}; p \in \mathsf{Proc}(V,S); env \in (S)\mathsf{Proc}(V,S); \mathsf{Guarded}(V,S,env,p))\mathsf{List}(\mathsf{Transition}(V))$$

Step_f is now dependently typed on both p and env. Now when we attempt to determine the transitions of a process we have the the following definition for evaluating process constants

$$\mathsf{Step}_f(\mathsf{Var}(s), \mathsf{Var}_g(h)) = \mathsf{Step}_f(env(s), h)$$

where Var_g is the introduction rule which specifies that a process constant is guarded if the process that it is naming is guarded and h is a proof of this guardedness. The recursion in function Step_f is now well-founded and this guarantees that Step_f always terminates.

We can prove the correctness of Step_f in relation to the definition of Step. For brevity we mention here just one associated lemma: if $\mathsf{Step}(V, S, p, \mathsf{Out}, (ch, v), p')$ (i.e. $p \xrightarrow{ch!v} p'$ in CCS notation) then the list returned by Step_f contains the transition $\mathsf{Out}_t(ch, v, p')$. We prove this in Alf by induction on the transition.

8 Related Work

There have been several implementations of process algebras in type theory but as far as we know this is the first that addresses value-passing CCS with parametric recursion in such a way that we borrow data domains from the type theory itself in a sound manner. [CP88] implements pure (*i.e.,* without value-passing or parametric recursion) CCS in Nuprl using the denotational model of non-well-founded set theory. [Sel93] describes the implementation of μCRL in Coq. μCRL is algebraic and is related to ACP. The approach differs in that we represent semantics operationally (as an inductively defined set and can therefore do induction over the relation) and μCRL is defined axiomatically. [Gim95] describes an implementation of CBS in Coq but does not implement recursion explicitly. Instead, Coq is extended with co-inductive types that allow one to express processes as infinite objects.

9 Conclusions

We have described an implementation of CCS in Alf, an implementation of type theory. The version of CCS we studied was value-passing and allowed recursive defining equation which could carry parameters. The language of channel expressions and state expressions was borrowed from Alf in a sound manner consequently allowing the possible types of the channel values and process parameter to be any Alf data type including infinite data domains. Channel expression and process recursion usually require some form of substitution. However, this too was borrowed from Alf thus simplifying the representation as we no longer need to implement substitution (and alpha conversion) nor proofs of correctness. Also, borrowing data types from the framework allows us to use off-the-shelf theorems.

Constraining the types of value and states. As we mentioned, the type $\mathsf{Proc}(V, S)$ contains more processes than can be translated from concrete syntax $\mathbf{P}$ using ϕ. The extra processes in $\mathsf{Proc}(V, S)$ are formed by instantiating either V or S to Channel, or to the type of processes themselves (*i.e.,* $\mathsf{Proc}(\mathsf{Proc}(V, S'), S)$). This situation occurs because V and S are very general and can be instantiated to any Alf type. However, it is straightforward to define a general type Value which disallows problem types above and includes all the combinations of the types of lists, tuples, disjoint unions, and base types. Given the following definition of a channel or state value,

```
Value ∈ Set
    list ∈ (List(Value))Value
    tuple ∈ (Prod(Value, Value))Value
    disjSum ∈ (Plus(Value, Value))Value
    nat ∈ (Nat)Value
    bool ∈ (Bool)Value
```

all of the problems disappear if we instantiate V and S to Value so that we have $\mathsf{Proc}(\mathsf{Value}, \mathsf{Value})$. Also, nothing has changed in the definition of Step and we still identify Alf variables with channel and state variables and use Alf substitution for channel variable and process substitution.

References

[CNSvS95] Thierry Coquand, Bengt Nordström, Jan Smith, and Björn von Sydow. Type the-

ory and programming. Extended Abstracts in Theoretical Computer Science bulletin number 52, February 1995. Also available Chalmers University of Technology technical report 81.

[Coq93] Thierry Coquand. Infinite objects in type theory. In *Proceedings of the 1993 TYPES Workshop*, number 806 in Lecture Notes in Computer Science. Springer-Verlag, 1993.

[CP88] Rance Cleaveland and Prakash Panangaden. Type theory and concurrency. *International Journal of Parallel Programming*, 17(2):153–206, 1988.

[CY94] Rance Cleaveland and Daniel Yankelevich. An operational framework for value-passing processes. In *POPL'94, Proceedings of the* 21^{st} *annual symposium on principles of programming languages*, 1994.

[DFH95] Jöelle Despeyroux, Amy Felty, and André Hirschowitz. Higher-order abstract syntax in Coq. In *TLCA 95 Typed Lambda Calculus and Applications*, number 902 in Lecture Notes in Computer Science. Springer Verlag, 1995.

[EP88] C. Elliot and F. Pfenning. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN'88 International Conference on Programming Language Desighn and Implementation*, Atlanta Georgia, June 1988. ACM, ACM Press.

[Gim95] Eduardo Giménez. Implementation of co-inductive types in Coq: An experiment with the Alternating Bit Prototcol. Technical report, INRIA Lyon, "June" 1995. Available at `ftp.lip.ens-lyon.fr/pub/Rapports/RR/RR95`.

[HL95] Mathew Hennessy and Hu Min Lin. Symbolic bisimulations. *Theoretical Computer Science*, 1995.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[Pau94] Lawrence Paulson. *Isabelle: A Generic Theorem Prover*. Springer-Verlag, 1994. LNCS 828.

[Pet94] Jenny Petersson. Tools for a calculus of broadcasting systems. Licentiate thesis, Department of Computer Science, Chalmers University of Technology, 1994.

[Pfe91] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*. Cambridge University Press, 1991.

[Pra93] K. V. S. Prasad. Programming with broadcasts. In *CONCUR*, August 1993. Springer Verlag LNCS 715.

[Pra94] K. V. S. Prasad. Broadcasting with priority. In *ESOP*, April 1994. Springer Verlag LNCS 788.

[Pra9x] K.V.S. Prasad. A calculus of broadcasting systems. *The Science of Computer Programming*, 199x. To apprear.

[Sel93] M.P.A. Sellink. Verifying process algebra proofs in type theory. Technical Report 87, Deptartment of Philosohy, Utrecht University, "March" 1993. Available at `http://www.phil.ruu.nl/home/marco/preprints.html`.

[Sti93] Colin Stirling. Modal and temporal logics. In Samson Abramsky, Don Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, pages 478–563. Oxford: Clarendon Press, 1993.

[Vaa93] Frits Vaandrager. Expressiveness results for process algebras. Technical Report CS-9301, CWI, University of Amsterdam, Programming Research Group, Amsterdam, The Netherlands, 1993. Appeared in Proceedings of the REX Workshop: "Semantics: Foundations and Applications". LNCS, Springer-Verlag.