Verifying Process Algebra Proofs in Type Theory

M.P.A. Sellink

Department of Philosophy, Utrecht University

P.O. Box 80.126, 3508 TC Utrecht, The Netherlands

email: alex@phil.ruu.nl

Abstract

In this paper we study automatic verification of proofs in process algebra. Formulas of process algebra are represented by types in typed λ-calculus. Inhabitants (terms) of these types represent proofs. The specific typed λ-calculus we use is the *Calculus of Inductive Constructions* as implemented in the interactive proof construction program COQ.

Introduction

Automatic verification will, as we expect, have a beneficial influence on the application of process theory. We believe that it is the only way to reach an acceptable level of correctness of proofs for programs and protocols of realistic size.

An earlier attempt of automatic verification of propositions of process theory is from Rance Cleaveland and Prakash Panangaden [4], who gave an implementation of Milner's *Calculus of Communicating Systems* (CCS) [13] in NuPrl [5]. They constucted a model of CCS in non-well-founded set theory $ZFC^- + AFA$ (ZFC where the foundation-axiom is replaced by the anti-foundation-axiom AFA) and then implemented $ZFC^- + AFA$ in NuPrl.

In 1991, Urban Engberg, Peter Grønning and Leslie Lamport published a paper on mechanical verification of concurrent systems. They started from the *Temporal Logic of Actions* (TLA) which is a logic for specifying and reasoning about concurrent systems. They made use of the verification system LP [8]. In order to avoid errors in the encoding of TLA-expressions into LP, a translator from TLA to LP is written.

We adopt the algebraic approach of Jan Bergstra and Jan Willem Klop [2]. This approach is known as *Algebra of Communicating Processes* (ACP). Jan Friso Groote and Alban Ponse developed a formal language [10] and an accompagnying proof theory [11] for ACP (+ data). The formal language is called μCRL. This paper builds upon the proof theory for μCRL, which was designed to facilitate automatic proof verification.

We indicate how μCRL properties can be transformed to types, how μCRL axioms can be represented, how the logical deduction rules are treated, and so on. There are numerous ways to do this things. At this stage the best (correct, efficient) way to follow is not yet clear to us. This paper must be seen as a first attempt. Proofs for Milner's Scheduler and the good old alternating bit protocol are computer-checked using the representation introduced here [12, 3]. Recently, Jaco van de Pol checked a μCRL proof of the Bounded Retransmission Protocol (BRP). Different from [12] and [3] the BRP proof was checked by building large tacticals that automatically rewrite the huge μCRL process terms into some kind of normal form. This seems to save a lot of work.

This paper is organized as follows. In Section 1 we give a short, informal introduction to type theory. Section 2 is an overview of the COQ system. ACP and μCRL are briefly explained in Section 3. The implementation of μCRL is exposed in sections 4, 6 and 7. Section 5 discusses the deduction rules of the proof theory for μCRL and their implementations in COQ. Three examples of the proposed implementation are added in an appendix.

Acknowledgements. First of all I would like to thank Christine Paulin-Mohring for her hospitality during my visit to Lyon and answering my questions about COQ.

This paper benefitted much from discussions with Marc Bezem, Jan Friso Groote, Jaco van de Pol and Jan Springintveld.

1 Type theory

Types (also called *sorts*) are frequently used in mathematics, logic and computer science. We just mention many sorted predicate logic. In strongly typed programming languages, every expression has a type. Primitive expressions, such as variables, have a type by declaration. These declarations usually classify the variables. Such classes (INTEGER, REAL, CHAR,...) are called *types*. 'x is of type A' is denoted as $x : A$. In Subsection 1.2 we treat type forming connectives more extensively. For the moment we just mention that $A \to B$ is a type if A and B are types.

In the λ-*calculus* [1] types are used to restrict application: a term F may only be applied to a term x if the types of F and x satisfy some restriction. More specifically, $Fx : B$ if $F : A \to B$ and $x : A$. (This is formalized in the inference rule *App* of Table 1.) Expressions of the form $b : B$ are called *statements*. The naive interpretation of a statement is a set theoretic one. This means that types are interpreted as sets and $b : B$ is interpreted as $b \in B$ (b is an element of set B). The type $A \to B$ is interpreted as the set of functions from A to B. When b is an element of B (that possibly contains x), then $\lambda x : A \,.\, b$ is the function from A to B that maps a to $(\lambda x : A \,.\, b)a =_\beta b[a/x]$ for all $a \in A$. The assumption $\langle x : A \rangle$ can be dropped when one abstracts from x in b, because x becomes a bound variable. The type of this bound variable is stored in the term $\lambda x : A \,.\, b$.

1.1 The Curry-Howard-interpretation

The λ-calculus consists of two basic operations on terms, *application* and *abstraction*, both with their corresponding typing rules. These rules are exposed in Table 1. The set Γ is called the *context*. It is a sequence of *statements* of the form $x_i : A_i$ with all the x_i's mutually distinct variables. The types A_i in Γ represent the assumptions (declarations) that are made. The axiom Ax encodes the idea that the statements of the context are assumptions. In Table 1 we give rules for deriving statements.

$$\frac{}{\Gamma \vdash x : A}\, Ax \qquad \langle x : A \rangle \in \Gamma$$

$$\frac{\Gamma \vdash F : A \to B \qquad \Gamma \vdash x : A}{\Gamma \vdash Fx : B}\, App$$

$$\frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda x : A \,.\, b : A \to B}\, Abs$$

Table 1: Rules of inference

Curry and Howard introduced another interpretation of the statement $b : B$, often referred to as 'propositions as types'. In this interpretation the type B represents a proposition instead of a set and b represents a proof of that proposition B. In this case the arrow is interpreted as the logical implication. *App* is understood as 'Fx proves B when F proves $A \to B$ and x proves A'. *Abs* is understood as '$\lambda x : A \,.\, b$ proves $A \to B$ when b proves B'. $\lambda x : A \,.\, b$ builds the proof $b[a/x]$ of B from the proof a of A. This is exactly the intuitionistic idea that a proof of $A \to B$ is an object that transforms a proof of A into a proof of B. In the Curry-Howard interpretation the application rule *App* corresponds to the arrow-elimination rule ($\to E$) of propositional logic, and the abstraction rule *Abs* rule corresponds to the arrow-introduction rule ($\to I$).

We give some examples of the Curry-Howard interpretation. The first example shows that $B \to A$ is provable in the context $\langle x : A \rangle$ with constructive proof $\lambda y : B \,.\, x$.

$$\dfrac{\dfrac{}{\langle x : A\ ,\ y : B\rangle \vdash x : A}\,Ax}{\langle x : A\rangle \vdash \lambda y{:}B\,.\,x : B \to A}\,Abs$$

The second example proves a well-known tautology. We omit brackets according to the convention that $A \to B \to C$ stands for $A \to (B \to C)$ and xyz stands for $(xy)z$. Define

$$\Gamma \equiv \langle x : A \to B \to C\ ,\ y : A \to B\ ,\ z : A\rangle$$

then we have the following derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash x : A \to B \to C}\,Ax \quad \dfrac{}{\Gamma \vdash z : A}\,Ax}{\Gamma \vdash xz : B \to C}\,App \quad \dfrac{\dfrac{}{\Gamma \vdash y : A \to B}\,Ax \quad \dfrac{}{\Gamma \vdash z : A}\,Ax}{\Gamma \vdash yz : B}\,App}{\Gamma \vdash xz(yz) : C}\,App}{\langle x : A \to B \to C\ ,\ y : A \to B\rangle \vdash \lambda z{:}A\,.\,xz(yz) : A \to C}\,Abs}{\langle x : A \to B \to C\rangle \vdash \lambda y{:}A \to B\,.\,\lambda z{:}A\,.\,xz(yz) : (A \to B) \to A \to C}\,Abs}{\langle\,\rangle \vdash \lambda x{:}A \to B \to C\,.\,\lambda y{:}A \to B\,.\,\lambda z{:}A\,.\,xz(yz) : (A \to B \to C) \to (A \to B) \to A \to C}\,Abs$$

proving the tautology $(A \to B \to C) \to (A \to B) \to A \to C$.

The typed λ-calculus given by the inference rules of Table 1 is known as $\lambda^{\to}$ in the literature. In the following subsections we will extend $\lambda^{\to}$ in order to improve the expressive power of the system.

1.2 The type forming connectives $\to$ and Π

In the preceding subsection we already mentioned that $A \to B$ is a type whenever A and B are. In Backus-Naur-form we could express this as follows:

$$\mathbb{T} ::= \mathbb{C} \mid \mathbb{T} \to \mathbb{T}$$

$\mathbb{T}$ stands for the set of all possible types. $\mathbb{C}$ is a set of constants.

In the previous subsection we proved $B \to A$ is provable in context $\langle x : A\rangle$. In fact we also have the 'meta assumption' that A and B are propositions. One can introduce a constant $*^p \in \mathbb{C}$ representing the set of all propositions. This enables us to declare the assumption that A is a proposition by $\langle A : *^p\rangle$. Now it is no longer a meta assumption. For instance the proposition $B \to A$ is not provable in context $\langle x : A\rangle$ but in context $\langle A : *^p\ ,\ B : *^p\ ,\ x : A\rangle$. Similarly we can define a constant $*^s$ representing the set of all sets.

$x : A$	and	$A : *^s$	is interpreted as x is an element of set A
$x : A$	and	$A : *^p$	is interpreted as x is a proof of proposition A

A is called a *type* and x is called a *term* of type A. The following example illustrates that the rules of Table 1 are not satisfactory anymore when $*^p$ and $*^s$ are added to the system.

EXAMPLE 1.2.1 Consider the context $\Gamma \equiv \langle A : *^p\ ,\ B : *^p\ ,\ x : A\rangle$. This must be understood as 'Let A and B are propositions and x a proof of A'. Obviously

$$\dfrac{}{\Gamma \vdash x : A}\,Ax$$

When we abstract from x, we construct a proof of $A \to A$.

$$\dfrac{\Gamma \vdash x : A}{\langle A : *^p\ ,\ B : *^p\rangle \vdash \lambda x{:}A\,.\,x : A \to A}\,Abs$$

It is perfectly allowed to abstract from A. When we use Abs we find:

$$\dfrac{\langle A : *^p\ ,\ B : *^p\rangle \vdash \lambda x{:}A\,.\,x : A \to A}{\langle B : *^p\rangle \vdash \lambda A{:}*^p\,.\,\lambda x{:}A\,.\,x : *^p \to A \to A}\,Abs$$

Now a problem arises when we apply $\lambda A{:}*^p\,.\,\lambda x{:}A\,.\,x$ to B. Note that $(\lambda A{:}*^p\,.\,\lambda x{:}A\,.\,x)\ B =_\beta \lambda x{:}B\,.\,x$.

$$\dfrac{\langle B:*^p\rangle \vdash \lambda A{:}*^p\,.\,\lambda x{:}A\,.\,x : *^p \to A \to A \qquad \dfrac{}{\langle B:*^p\rangle \vdash B:*^p}\,Ax}{\langle B:*^p\rangle \vdash \lambda x{:}B\,.\,x : A \to A}\,App$$

Something is going wrong here: $\lambda x{:}B\,.\,x$ should not be a proof of $A \to A$. When we replace $*^p$ by $*^s$ in this example then exactly the same problem arises: $\lambda x{:}B\,.\,x$ should not be a mapping from A to A. Apparently it is not correct to say that the type of $\lambda A{:}*\,.\,\lambda x{:}A\,.\,x$ is $* \to A \to A$ (where $*$ ranges over $\{*^p, *^s\}$).

The problem is caused by the fact that the variable from which has been abstracted, occurs in the type of the term. In this example: A occurs in $A \to A$. This was not the case when we abstracted from x. What we need is another type forming operation, which allows the dependence of $A \to A$ on A. For this new operation the symbol Π is used, so:

$$\lambda A{:}*^p\,.\,\lambda x{:}A\,.\,x : \Pi A{:}*^p\,.\,A \to A$$

We have to modify our set $\mathbb{T}$ of all possible types. The basic types A and B have become type variables (i.e. abstractions from A and B are allowed) but $*^p$ and $*^s$ remain constants for no abstractions from $*^p$ or $*^s$ are allowed. Furthermore we have to add the new type forming operator Π.

$$\mathbb{T} ::= \mathbb{V} \mid \mathbb{C} \mid \mathbb{T} \to \mathbb{T} \mid \Pi\mathbb{V}{:}\mathbb{T}\,.\,\mathbb{T}$$

From now on the set $\mathbb{T}$ is only a set of *pseudo*-types. For instance $\Pi v{:}v\,.\,t$ with $v \in \mathbb{V}$ and $t \in \mathbb{T}$ is a type that one would like to exclude. Such types will be called *illegal*.

Note that the choice of Π is a natural choice when we interpret types as sets again. $F : \Pi x{:}A\,.\,B(x)$ is an object that gives an element of set $B(a)$ when applied to an element $a \in A$. Such object can be identified with an element of the product of sets $B(x)$ with x ranging over set A, usually denoted by

$$\prod_{x \in A} B(x)$$

The interpretation of the type forming operator Π in the case of 'propositions as types' is well illustrated by the term $I_{poly} \equiv \lambda A{:}*^p\,.\,\lambda x{:}A\,.\,x$. This term constructs a proof of $A \to A$ from an arbitrary proposition A. In other words: it is a proof of $\forall A.A \to A$. This suggests that in the Curry-Howard interpretation Π should be read as $\forall$.

Another observation is that a function from A to B is in fact a tuple $(\ldots, b_x, \ldots)$ with x ranging over A. This tuple is an element of

$$\prod_{x \in A} B$$

(The product of A copies of the same set B.) Now x is a variable that does not occur in B. Therefore, $A \to B$ is nothing more than a special case of $\Pi x{:}A\,.\,B$. Consequently we can skip $\mathbb{T} \to \mathbb{T}$ in the definition of $\mathbb{T}$. The type $A \to B$ is seen as $\Pi x{:}A\,.\,B$ with the bound variable x chosen such that x does not occur free in B. The rules given in Table 1 are replaced by the more general rules given in Table 2. Note that ΠE (resp. ΠI) generalizes $\to E$ (resp. $\to I$) as $B[a/x] \equiv B$ whenever x does not occur in B.

So far we are able to type *terms* depending on *terms* as well as *terms* depending on *types*. For instance, the well-typed term $I_{poly}\ A\ x$ (with $A : *^p$ and $x : A$) of type A is a term with dependence on *type* A and *term* x. Under the Curry-Howard-interpretation this corresponds to second order propositional logic. Since we are also interested in predicate logic we need the facility to type *types* depending on *terms*. Defining a predicate over some set A, corresponds to permitting dependency of propositions on specific elements of A. In other words: When $P\ x : *^p$ for some $x : A$ and $A : *^s$ then $P\ x$ is a *type* depending on the *term* x. Such a construction is not possible without types of the form $\Pi x{:}A\,.\ *^p$ $(\equiv A \to *^p)$ which should be the type of P. However, such types are not allowed because $x \notin \mathbb{V}$. (x is not a *type* variable.) This problem is solved when we extend $\mathbb{T}$ with elements of the form $\lambda\mathbb{V}{:}\mathbb{T}\,.\,\mathbb{T}$ and $\mathbb{T}\ \mathbb{T}$. Both *types* and *terms* then can come from the same set $\mathbb{T}$. Types

$$\frac{}{\Gamma \vdash x : A}\, Ax \qquad \langle x : A\rangle \in \Gamma$$

$$\frac{\Gamma \vdash F : \Pi x{:}A\,.\,B \qquad \Gamma \vdash a : A}{\Gamma \vdash Fa : B[a/x]}\, \Pi E$$

$$\frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda x{:}A\,.\,b : \Pi x{:}A\,.\,B}\, \Pi I$$

Table 2: Π-introduction and Π-elimination

and terms are not mixed up, because types have type $*^p$ or $*^s$. The system we defined by the rules of Table 2 together with the modification that both terms and types come from the set

$$\mathbb{T} ::= \mathbb{C} \mid \mathbb{V} \mid \mathbb{T}\ \mathbb{T} \mid \lambda \mathbb{V}{:}\mathbb{T}\,.\,\mathbb{T} \mid \Pi \mathbb{V}{:}\mathbb{T}\,.\,\mathbb{T},$$

and the context does not contain illegal types, will be denotated by us as λLC. (L stands for Logic.) When $*^p$ and $*^s$ are identified in λLC then we obtain the *Calculus of Constructions* (λC) [6].

1.3 The calculus of inductive constructions

The *Calculus of Inductive Constructions* (λIC) is an extension of λLC. The main difference between λIC and λLC is the presence of so called *inductive types* in λIC. We illustrate the motivations for extending λLC with inductive types by an example.

EXAMPLE 1.3.1 Assume that we want to have a type *nat*, representing the natural numbers. The first (naive) suggestion is to declare $nat : *^s$, $O : nat$ and $S : nat \to nat$. Now $\underline{n} \stackrel{\text{def}}{=} \underbrace{S(\cdots(S}_{n}\ O)\cdots)$ represent $n \in \mathbb{N}$. The problem with this implementation is that the numbers are too passive. Therefore it is impossible to construct for instance a term *sum* satisfying the property

$$sum\ \underline{n}\ \underline{m} =_\beta \underline{n+m} \qquad \text{for all } n, m \in \mathbb{N}$$

This problem can be solved by using the following impredicative definition for the natural numbers.

$$\begin{array}{ll} nat & \equiv \Pi X{:}*^s\,.\,X \to (X \to X) \to X \\ O & \equiv \lambda X{:}*^s\,.\,\lambda x{:}X\,.\,\lambda f{:}X \to X\,.\,x \\ S & \equiv \lambda n{:}nat\,.\,\lambda X{:}*^s\,.\,\lambda x{:}X\,.\,\lambda f{:}X \to X\,.\,f(n\ X\ x\ f) \end{array}$$

Obviously we have that $O : nat$ and $S : nat \to nat$ again. In this case

$$\underline{n} \stackrel{\text{def}}{=} \lambda X{:}*^s\,.\,\lambda x{:}X\,.\,\lambda f{:}X \to X\,.\,\underbrace{f(\cdots(f}_{n}\ x)\cdots)$$

Note that $S\ \underline{n} =_\beta \underline{n+1}$. ($\underline{n}$ is called a *Church numeral*.) It is easy to verify that

$$sum \equiv \lambda n{:}nat\,.\,\lambda m{:}nat\,.\,\lambda X{:}*^s\,.\,\lambda x{:}X\,.\,\lambda f{:}X \to X\,.\,n\ X\ (m\ X\ x\ f)\ f$$

satisfies $sum\ \underline{n}\ \underline{m} =_\beta \underline{n+m}$ for all $n, m \in \mathbb{N}$.

Although the latter encoding is considerably better than the first one, it still has some serious disadvantages:

- We don't have a proof for the induction principle on the natural numbers. Consequently we can not prove properties like $\Pi x{:}\,nat\,.\,less\ x\ (S\ x)$ although we can prove $less\ \underline{n}\ (S\ \underline{n})$ for every $n \in \mathbb{N}$. ($less : nat \to nat \to *^p$ represents the $<$-relation.)
- We are not able to prove $\underline{0} \neq \underline{1}$
- Any predecessor function takes at least linear time. (The predecessor of $\underline{n+1}$ is computed in at least n steps.)

All these problems can be solved by extending the system with inductive types. The extension is too complicated to be explained here in detail. We only illustrate the basic idea by some representative examples.

DEFINITION 1.3.2 The set $\mathbb{T}$ of *pseudo types* is defined by the following abstract syntax:

$$\begin{array}{rcl}\mathbb{T} & ::= & \mathbb{C} \mid \mathbb{V} \mid \mathbb{T}\ \mathbb{T} \mid \lambda\mathbb{V}{:}\mathbb{T}\,.\,\mathbb{T} \mid \Pi\mathbb{V}{:}\mathbb{T}\,.\,\mathbb{T} \mid \mathsf{Ind}(\mathbb{V}:\mathbb{T})\{\mathbb{T} \mid \cdots \mid \mathbb{T}\} \mid \\ & & \mathsf{Constr}(\mathbb{N},\mathbb{T}) \mid \mathsf{Elim}(\mathbb{T},\mathbb{T})\{\mathbb{T} \mid \cdots \mid \mathbb{T}\}\end{array}$$

Note that λIC consists of extra rules defining which Ind-types, which Constr-types and which Elim-types are legal. All these rules can be found in [14]. We do not expose them here. Types of the form $\mathsf{Ind}(\mathbb{V}:\mathbb{T})\{\mathbb{T} \mid \cdots \mid \mathbb{T}\}$ are called *inductive types*.

For instance

$$\mathsf{Ind}(X : *^s)\{X \mid X \to X\}$$

is an inductive type. X is a bound variable in this type. This inductive type should be understood as 'the smallest set X that is closed under two constructors, one of type X and one of type $X \to X$. More general $\mathsf{Ind}(X : A)\{C_1 \mid \cdots \mid C_p\}$ is the smallest set (weakest proposition) of type A that is closed under its p constructors, that have types $C_1, \ldots, C_p$. Under suitable conditions (that are satisfied in all our examples of inductive types) this smallest set does exist. $\mathsf{Constr}(i, \mathsf{Ind}(X : A)\{C_1 \mid \cdots \mid C_p\})$ stands for the i-th constructor of $\mathsf{Ind}(X : A)\{C_1 \mid \cdots \mid C_p\}$. $(1 \leq i \leq p)$. Together with an inductive type comes a computational device:ι-reduction. All together, we have the following inductive encoding for the natural numbers.

EXAMPLE 1.3.3 Encoding of the natural numbers in λIC.

$$\begin{array}{lcllcll}nat & \equiv & \mathsf{Ind}(X : *^s)\{X \mid X \to X\} & & & & \\ O & \equiv & \mathsf{Constr}(1, \mathsf{Ind}(X : *^s)\{X \mid X \to X\}) & : & X & [nat/X] & \equiv\ nat \\ S & \equiv & \mathsf{Constr}(2, \mathsf{Ind}(X : *^s)\{X \mid X \to X\}) & : & X \to X & [nat/X] & \equiv\ nat \to nat\end{array}$$

EXAMPLE 1.3.4 Let $\Gamma \equiv \langle A : *^p\ ,\ B : *^p\rangle$. Another interesting inductive type is

$$\mathsf{Ind}(X : *^p)\{A \to X \mid B \to X\}$$

It is natural to abbreviate this term with $or(A, B)$ because it is true (inhabited) whenever A or B is true (inhabited). This follows directly from the *constr*-rule [14]:

$$\begin{array}{lcllll}\mathsf{Constr}(1, or(A,B)) & : & A \to X & [or(A,B)/X] & \equiv & A \to or(A,B) \\ \mathsf{Constr}(2, or(A,B)) & : & B \to X & [or(A,B)/X] & \equiv & B \to or(A,B)\end{array}$$

In example 1.3.7 at the end of this subsection we explain that $or(A, B)$ is also the smallest inhabitant of $*^p$ (weakest proposition) that has this property. It is perfectly allowed to construct a polymorphic 'or' via λ-abstractions:

$$or \equiv \lambda A{:}*^p\,.\,\lambda B{:}*^p\,.\,or(A, B)$$

For the construction of inhabitants of the different induction principles we need terms of the form $\mathsf{Elim}(\mathbb{T},\mathbb{T})\{\mathbb{T} \mid \cdots \mid \mathbb{T}\}$. In general the term $\mathsf{Elim}(a, B)\{f_1 \mid \cdots \mid f_p\}$ is not legal. One of the constraints is that the type A of a must be of the form $I\ t_1 \cdots t_m$ for some inductive type I. The type of f_i must be F_i. The type F_i can be derived from the i-th constructor of I as explained in [14] (for all $1 \leq i \leq p$, $p \geq 0$ is the

number of constructors of I). When $\mathsf{Elim}(a, B)\{f_1 \mid \cdots \mid f_p\}$ is constructed from a then the inductive type I is eliminated. This explains the name Elim. When a is build from the i-th constructor of I, then the type of $\mathsf{Elim}(a, B)\{f_1 \mid \cdots \mid f_p\}$ is an application of f_i. We restrict ourselves to some examples.

EXAMPLE 1.3.5 Define $sum \equiv \lambda n{:}nat.\, \lambda m{:}nat.\, \mathsf{Elim}(n, nat)\{m \mid \lambda y{:}nat.\, S\}$ then

$$sum\ \underline{n}\ \underline{m} =_\beta \underline{n+m}$$

where $\underline{n}$ abbreviates $\underbrace{S(\cdots(S}_{n}\ O)\cdots)$. The term $\mathsf{Elim}(n, nat)\{m \mid \lambda y{:}nat.\, S\}$ is a recursively defined *sum*. Its reduction behaviour is as follows:

$$\begin{array}{lll} \mathsf{Elim}(O, nat)\{m \mid \lambda y{:}nat.\, S\} & \twoheadrightarrow_\iota & m \\ \mathsf{Elim}(S\ k, nat)\{m \mid \lambda y{:}nat.\, S\} & \twoheadrightarrow_\iota & (\lambda y{:}nat.\, S)\ k\ \mathsf{Elim}(k, nat)\{m \mid \lambda y{:}nat.\, S\} \\ & \twoheadrightarrow_\beta & S\ \mathsf{Elim}(k, nat)\{m \mid \lambda y{:}nat.\, S\} \end{array}$$

More generally, when $b : B$ and $f : nat \to B \to B$, then

$$\begin{array}{lll} \mathsf{Elim}(O, B)\{b \mid f\} & \twoheadrightarrow_\iota & b \\ \mathsf{Elim}(S\ k, B)\{b \mid f\} & \twoheadrightarrow_\iota & f\ k\ \mathsf{Elim}(k, B)\{b \mid f\} \end{array}$$

EXAMPLE 1.3.6 Assume $\langle P : nat \to *^p\ ,\ f_O : (P\ O)\ ,\ f_S : \Pi x{:}nat.\, (P\ x) \to (P(S\ x))\rangle \subseteq \Gamma$ then

$$\Gamma \vdash \mathsf{Elim}(n, P)\{f_O \mid f_S\} : (P\ n)$$

An inhabitant of the induction principle on *nat* is constructable now by simple λ-abstractions:

$$\Pi P{:}nat \to *^p.\, (P\ O) \to (\Pi x{:}nat.\, (P\ x) \to (P(S\ x))) \to \Pi n{:}nat.\, (P\ n)$$

is inhabited by $\lambda P{:}nat \to *^p.\, \lambda f_O{:}(P\ O).\, \lambda f_S{:}\Pi x{:}nat.\, (P\ x) \to (P(S\ x)).\, \lambda n{:}nat.\, \mathsf{Elim}(n, P)\{f_O \mid f_S\}$

EXAMPLE 1.3.7 Assume $\langle A : *^p\ ,\ B : *^p\ ,\ P : *^p\ ,\ f_L : A \to P\ ,\ f_R : B \to P\ ,\ H : or\ A\ B\rangle \subseteq \Gamma$, then

$$\Gamma \vdash \mathsf{Elim}(H, P)\{f_L \mid f_R\} : P$$

Via λ-abstractions we can construct an inhabitant of

$$\Pi P{:}*^p.\, (A \to P) \to (B \to P) \to (or\ A\ B) \to P$$

In other words: when we have a proposition P that can be proved from A as well as from B, then $(or\ A\ B) \to P$ is inhabited. This means that $or\ A\ B$ is the weakest proposition with these properties.

2 The system COQ

The system COQ is a proof construction system, based on type theory. Formulas are represented by types of the underlying typesystem λIC. Inhabitants represent proofs. The construction of such a proof is an interactive process. The user can communicate with the system via *commands*. He determines which strategy should be followed. COQ only executes the calculations and checks conditions. At the end of this section we give a simple example of a COQ proof session.

Notations. We use *italic style* for expressions of λIC and expressions occurring in a μCRL specification. Expressions in COQ are written in `type style`. For instance, when x is a type in μCRL, then x is the corresponding type in λIC which is written as `x` in the COQ system.

In COQ you write `[x:A]B` for $\lambda x{:}A.\, B$ and `(x:A)B` for $\Pi x{:}A.\, B$. When x does not occur in B then $\Pi x{:}A.\, B$ is written as `A->B`. Furthermore $*^p$ is written as `Prop` and $*^s$ is written as `Set`.

Bound variables are renamed automatically when a name conflict occurs. Brackets are omitted according to the following conventions

$$\begin{array}{lll} (\Pi x{:}A.\, B) \to C & \equiv (A \to B) \to C & \text{brackets are written, } x \notin \mathrm{FV}(B) \\ \Pi x{:}A.\, (B \to C) & \equiv A \to (B \to C) & \text{brackets are omitted, } x \notin \mathrm{FV}(B) \cup \mathrm{FV}(C) \end{array}$$

Furthermore xyz means $(xy)z$, $\lambda x{:}A.\, B \to C$ means $\lambda x{:}A.\, (B \to C)$ and $\lambda x{:}A.\, b\ c$ means $(\lambda x{:}A.\, b)c$.

2.1 Contexts

During a COQ session two different contexts are distinguished. The *global* context and the *local* context. The global context contains assumptions about process algebra in general. Such assumptions
(like `ALT:proc->proc->proc`, see Subsection 4.5) are global in the sense that they are not depending on a specific process algebra proof session. The local context contains one or more assumptions that should not be made in general. For instance, when you want to prove some property $\phi \rightarrow \psi$ in context Γ, the strategy (tactic) will be to prove ψ in context Γ, ϕ. It's obvious that the assumption ϕ may not be used in the next proof session. Therefore this assumption ϕ should be droppped from Γ, ϕ after the proof session. So Γ, ϕ is a local context in the sense that some of its assumptions are depending on the specific process algebra proof session.

One can add statements of the form $x : A$, with x a fresh variable, to the global context Γ by saying

```
Parameter x:A.
```

COQ checks for you whether $\Gamma \vdash A : s$ holds, for some context Γ and sort s (e.g. $s = *^p$ or $s = *^s$), or not. The statement is refused in the latter case. COQ overwrites $\langle x : A' \rangle$ when $\langle x : A' \rangle$ was already in Γ. When $s \equiv *^p$ then

```
Axiom x.  Assumes A.
```

has the same effect but is maybe better to read. This instruction enables you to build up a context simply by adding the statements one by one in the right order. A proof session starts with inserting the type `P`, that encodes your proposition, by saying

```
Goal P.
```

The global context is the initial local context. The local context is dropped after every proof session. The global context is dropped when you leave COQ.

2.2 Abbreviations

Assume we want to abbreviate `[x:nat]S(S x)` by `plus_two` and `[P:Prop]P->False` by `not`. This can be done as follows:

```
Definition plus_two  :nat->nat    = [x:nat]S(S x).
Definition not       :Prop->Prop  = [P:Prop]P->False.
```

The explicit typing may be omitted. You can change the syntax of an abbreviation with the help of the *Syntax* instruction. We give two examples:

```
Syntax plus_two  = "_+2".
Syntax not       = "~_".
```

The result of this instructions is that `plus_two x` is written as `x+2` and `not P` as $\sim$ `P`.

2.3 Inductive types

In COQ we have the facility to declare that for instance 'the set of natural numbers' is the smallest collection of terms that contains `O` and is closed under `S`. The syntax for this declaration is as follows:

```
Inductive Definition nat:Set = O:nat | S:nat->nat.
```

The result of this declaration is

```
nat  ≡ Ind(X:Set){X | X->X}
O    ≡ Constr(1,nat)
S    ≡ Constr(2,nat)
```

When you declare an inductive type `M`, COQ automatically introduces an abbreviation `M_ind` of an inhabitant of the induction principle on that type (see subsection 1.3). For instance `nat_ind` is an abbreviation of

```
[P:nat->Prop][a:(P O)][f:(x:nat)(P x)->(P(S x))][x:nat] (<P>Match x with a f)
```

where `<P>Match x with a f` is the COQ denotation for the term $\mathsf{Elim}(x, P)\{a \mid f\}$. The type of `nat_ind` is

```
(P:nat->Prop)(P O)->((x:nat)(P x)->(P (S x)))->(x:nat)(P x)
```

A new kind of reduction called *ι-reduction* reduces an Elim-term to an application of its constructors. We recall the reductions given in Subsection 1.3, this time in COQ-notation.

```
<B>Match O with b f       →ι  b
<B>Match (S k) with b f   →ι  f k (<B>Match k with b f)
```

where `b:B` and `f:nat->B->B`. From now on we will write $\twoheadrightarrow$ for the transitive symmetric closure of $\twoheadrightarrow_\beta \cup \twoheadrightarrow_\iota$.

The inductive definitions have a large variety of applications. For instance a notion of equality can be defined with the same technique:

```
Syntax eq = "<_>_=_".
Inductive Definition eq [A:Set;a:A]:A->Prop = refl_equal:<A>a=a.
```

The result of this declaration is:

```
eq          ≡ [A:Set][a:A]Ind(X:A->Prop){X a}
refl_equal  ≡ [A:Set][a:A]Constr(1,eq A a)
```

The idea behind the latter definition is that `eq A a:A->Prop` is the smallest predicate on `A` that holds in `a`. The abbreviation `eq_ind` is of type

```
(A:Set)(a:A)(P:A->Prop)(P a)->(b:A)(e:<A>a=b)->(P b).
```

In fact, `eq` defines ordinary *Leibniz equality* (identity of indiscernables). An alternative formulation for Leibniz equality would be:

```
Syntax L = "{_}_=_".
Definition L = [A:Set][a,b:A](P:A->Prop)(P a)->(P b).
```

The equivalence of these two versions of equality is established in the following lemma.

LEMMA 2.3.1 For all contexts Γ we have

$\Gamma \vdash$ `<A>x=y` is inhabited $\Longleftrightarrow \Gamma \vdash$ `{A}x=y` is inhabited

PROOF: We give the constructions in both directions.

($\Longrightarrow$) Let `H` be a proof for `<A>x=y`, then
`eq_ind A x` is of type `(P:A->Prop)(P x)->(b:A)(e:<A>x=b)->(P b)`.
For `P` we choose the predicate `[a:A]({A}x=a)`.
This gives us a term of type `({A}x=x)->(b:A)(H:<A>x=b)->{A}x=b`.
`{A}x=x` is proved by `[P:A->Prop][h:(P x)]h`.
Now `y` is substituted for `b` and `H` is substituted for `e`. We find that
`eq_ind A x [a:A]({A}x=a) [P:A->Prop][h:(P x)]h y e` inhabits `{A}x=y`.

($\Longleftarrow$) Let `H` be a proof for `{A}x=y`, then `H:(P:A->Prop)(P x)->(P y)`. Now
`H [a:A](<A>x=a) (refl_equal A x)`
inhabits `<A>x=y` because `refl_equal A x` inhabits `<A>x=x`.

■

Below we give the other inductive definitions that play a role in this paper. We give the matching induction principles without further explanation.

```
Syntax prod = "_*_".
Syntax pair = "<_,_>(_,_)".
Syntax or = "_\/_".
Syntax and = "_/\_".
Inductive Definition True:Prop = I:True.
Inductive Definition False:Prop = .
Inductive Definition bool:Set = true:bool | false:bool.
Inductive Definition prod [A,B:Set]:Set = pair:A->B->(A*B).
Inductive Definition or [A,B:Prop]:Prop = or_introl:A->(A\/B) | or_intror:B->(A\/B).
Inductive Definition and [A,B:Prop]:Prop = conj:A->B->(A/\B).
```

The induction principles are

```
True_ind  :  (P:Prop)P->True->P
False_ind :  (P:Prop)False->P                                          (1)
bool_ind  :  (P:bool->Prop)(P true)->(P false)-> (b:bool)(P b)         (2)
prod_ind  :  (A,B:Set)(P:(A*B)->Prop)
             ((a:A)(b:B)(P <A,B>(a,b)))->(y:A*B)(P y)
or_ind    :  (A,B,P:Prop)(A->P)->(B->P)-> (A \/ B)->P
and_ind   :  (A,B,P:Prop)(A->B->P)-> (A /\ B)->P
```

Remark that we can define any finite set inductively by enumerating its elements as constructors of that set. Let $A = \{a_1, \ldots, a_n\}$ be an arbitrary set, then we can represent A as follows.

```
Inductive Definition A:Set = a1:A | ··· | an:A.
```

The advantage of this inductive definition is that the elements of A are provably distinct, i.e. we have proofs for

$$\sim\texttt{<A>a}_i = \texttt{a}_j \tag{3}$$

for all $i \neq j$, expressing that $\neg(a_i = a_j)$. This is one of the reasons for using an inductive type to represent actions (see Subsection 4.4).

LEMMA 2.3.2 $\Gamma \vdash \sim$`<A>a`$_i$`=a`$_j$ for all contexts Γ and for all $i \neq j$

PROOF: Remark that `<Prop>Match a`$_i$ `with Q`$_1 \cdots$ `Q`$_n \twoheadrightarrow$ `Q`$_i$ for all `Q`$_k$`:Prop`. Let $i, j \in \{1, \ldots, n\}$ such that $i \neq j$ and let `H:<A>a`$_i$`=a`$_j$. We have to construct an inhabitant of `False`. Note that `eq_ind A a`$_i$ is of type `(P:A->Prop)(P a`$_i$`)->(b:A)(e:<A>a`$_i$`=b)->(P b)`. For `P` we choose the predicate `[x:A](<Prop>Match x with Q`$_1 \cdots$ `Q`$_n$`)` where `Q`$_1, \ldots,$ `Q`$_n$ an arbitrary sequence of types such that $\Gamma \vdash$ `Q`$_k$`:Prop` for all $k = 1, \ldots, n$ and at least `Q`$_i$ is inhabited. For instance, choose `Q`$_i \equiv$ `(C:Prop)C->C` and `q`$_i \equiv$ `[C:Prop][h:C]h` then $\Gamma \vdash$ `q`$_i$`:Q`$_i$. Now (`P a`$_k$`)` $\twoheadrightarrow$ `Q`$_k$ for all $k = 1, \ldots, n$ and `eq_ind A a`$_i$ `P q`$_i$ `a`$_j$ `H` is of type `Q`$_j$. Choose `Q`$_j \equiv$ `False`. This is a legal choise in every context because $\langle\ \rangle \vdash$ `False:Prop`. ■

2.4 The tactics of COQ

Using COQ means using so called *tactics*. For instance 'Assume A and try to prove B' is a tactic for proving $A \to B$. This tactic corresponds with $\to I$. There is a tactic behind every reduction rule of λIC. Tactics are executed via *commands*. We give a list of the commands we use for our proof sessions.

`Intro x.` corresponds to the Π-introduction rule

$$\frac{\Gamma \vdash \Pi x{:}A\,.\,B : s \qquad \Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda x{:}A\,.\,b : \Pi x{:}A\,.\,B}$$

Instead of proving $\Pi x\!:\!A\,.\,B$ in (local) context Γ, you can prove B in context $\Gamma, x:A$. The left premise was checked by COQ automatically when $\Pi x:A\,.\,B$ was introduced. The proof of $\Pi x:A\,.\,B$ will be $\lambda x\!:\!A\,.\,b$ where b is the proof of B that you have to construct. When we look for instance to *Second Order Implicational Logic* then this corresponds to the rules $(\forall I)$ and $(\rightarrow I)$. We can describe the behaviour of `Intro x` schematically as follows:

```
(y:A)φ(y)                          φ(x)
================  Intro x ⇝  ================
H  :  ψ                        x  :  A
                               H  :  ψ
```

`Intros` $\mathtt{x}_1 \cdots \mathtt{x}_n$. is a shorthand for `Intro` $\mathtt{x}_1; \ldots;$ `Intro` $\mathtt{x}_n$.

`Apply F.` corresponds to n times the Π-elimination rule, where n is the arity of F. For $i = 1, \ldots, n$:

$$\frac{\Gamma \vdash Fa_1 \cdots a_{i-1} : \Pi x_i\!:\!A_i\,.\,\cdots \Pi x_n\!:\!A_n\,.\,B[a_1/x_1]\cdots[a_{i-1}/x_{i-1}] \qquad \Gamma \vdash a_i : A_i}{\Gamma \vdash Fa_1 \cdots a_i : \Pi x_{i+1}\!:\!A_{i+1}\,.\,\cdots \Pi x_n\!:\!A_n\,.\,B[a_1/x_1]\cdots[a_i/x_i]} \tag{4}$$

Instead of proving $B[a_1/x_1]\cdots[a_n/x_n]$ in (local) context Γ, you can prove $A_1, \ldots, A_n$ in the same context. The proof of $B[a_1/x_1]\cdots[a_n/x_n]$ will be $Fa_1 \cdots a_n$ where a_i is the inhabitant (proof) of A_i that you have to construct. When we look to *Second Order Implicational Logic* again then this corresponds to n times the rules $(\forall E)$ and $(\rightarrow E)$. Schematically:

```
φ(y1, ..., yn)                                  A1
================                  Apply F ⇝     ================
H  :  ψ                                         H  :  ψ
F  :  (x1:A1)···(xn:An)φ(x1, ..., xn)           F  :  (x1:A1)···(xn:An)φ(x1, ..., xn)
                                                subgoal 2 is:  A2
                                                ⋮
                                                subgoal n is:  An
```

`Cut A.` corresponds to *Modus Ponens*. Instead of proving B in context Γ you can prove A and $A \rightarrow B$ in the same context. Proving $A \rightarrow B$ is usually done by assuming a proof of A and trying to construct a proof of B. This explains the name `Cut`, since

$$\dfrac{\begin{array}{c}\vdots\,\mathcal{D}_1\\ A\end{array} \qquad \dfrac{\begin{array}{c}[A]\\ \vdots\,\mathcal{D}_2\\ B\end{array}}{A \rightarrow B}\,(\rightarrow I)}{B}\,(\rightarrow E)$$

is a prototypical example of a *cut*. (The cut is eliminated when all occurences of A in the hypotheses of $\mathcal{D}_2$ are replaced by the prooftree $\mathcal{D}_1$.)

`Assumption.` corresponds to the fact that $\Gamma \vdash A : B$ when $A : B \in \Gamma$. (See [1] for a proof) The assumption tactic searches in the (local) context for a proof of the current goal.

`Undo` ***n***`.` cancels the last n commands. The default value of n is $n = 1$. `Undo` ***n*** is not stored in the history list of commands.

`Show` ***n***`.` shows the n-th subgoal. The default value of n is $n = 1$.

`Abort.` aborts the goal.

`Hint a.` places a into the list of hints. (See `Auto.`)

`Save a.` adds $\langle a : A\rangle$ to the global context, where A is the goal you just proved. a becomes an abbreviation of the (just found) inhabitant of that goal.

`Unfold` $\boldsymbol{n_1^1 \cdots n_1^{p_1} a_1 \cdots n_q^1 \cdots n_q^{p_q} a_q}$`.` replaces successively the occurrences $n_i^1, \ldots, n_i^{p_i}$, $n_i^j \in \mathbb{N}$ of a_i by its definition and reduces to normal form.
For instance : `Unfold 1 plus_two 2 3 not.` changes
`(not (not (not (<nat>x=(plus_two x)))))` into
`(not ((((<nat>x=(S (S x)))->False)->False))`

`Simpl.` corresponds to the ι-conversion rule [14]. It executes as many ι-reductions as possible (until the inhabitant of the inductive type is a variable or all the Elim-terms are disappeared).

`Elim h.` is defined only when $h : I\ t_1 \cdots t_m$ for some inductive type I. It is a shorthand for `Apply (I_ind` *"suitable terms"*`)`. COQ substitutes h for the matched type and automatically drops the type of h as a new subgoal. Later we will see some examples of this.

`Rewrite h.` is defined only when h proves the equality of a and b for some a and b of the same type. All the occurences of a in the current goal are replaced by b.

`Induction h.` is a shorthand for `Intro h ; Elim h.` The constraints follow from the ones given by `Elim h`.

`Pattern` $\boldsymbol{n}$ `x.` changes ϕ`(x)` to `([y:A]`ϕ`(y)) x`. Only the n-th occurence of `x` in ϕ`(x)` is replaced by `y`.

`Auto` $\boldsymbol{n}$`.` tries to construct a proof of the current goal by using only the suggestions of the list of hints. The depth of the search is n. (the default value for n is $n = 5$.)

When you run COQ it is convenient to have an input file consisting of a list of commands. COQ ignores data from the input file when it is placed between $(*\ldots*)$. You can store your commands in a file by saying `Open file.` You can close such file by saying `Close.` COQ creates a file `file.v` consisting of all the commands that are given between `Open file.` and `Close.` When you omit `Close.` then all the commands of the current goal are lost when you leave COQ because commands are added to `file.v` only when you save your goal or when you say `Close.`

EXAMPLE 2.4.1 The following proof session finds a proof for the proposition
`(A:Prop)(B:Prop)(C:Prop)(A->B->C)->(A->B)->A->C.`

```
coq < Goal (A:Prop)(B:Prop)(C:Prop)(A->B->C)->(A->B)->A->C.

1 subgoal
 (A:Prop)(B:Prop)(C:Prop)(A->B->C)->(A->B)->A->C

coq < Intros A B C x y z.

1 subgoal
 C
 ============================
 z  :  A
 y  :  A->B
 x  :  A->B->C
 C  :  Prop
 B  :  Prop
 A  :  Prop

coq < Apply x.

2 subgoals
```

```
 A
 ============================
 z  :  A
 y  :  A->B
 x  :  A->B->C
 C  :  Prop
 B  :  Prop
 A  :  Prop
subgoal 2 is:
 B
```

```
coq < Assumption.
```

```
1 subgoal
 B
 ============================
 z  :  A
 y  :  A->B
 x  :  A->B->C
 C  :  Prop
 B  :  Prop
 A  :  Prop
```

```
coq < Apply y.
```

```
1 subgoal
 A
 ============================
 z  :  A
 y  :  A->B
 x  :  A->B->C
 C  :  Prop
 B  :  Prop
 A  :  Prop
```

```
coq < Assumption.
```

```
Goal proved!
```

```
coq < Save S.
```

We can expose the proof term `S` by saying

```
coq < Print S.
```

```
[A:Prop][B:Prop][C:Prop][x:A->B->C][y:A->B][z:A] x z (y z)
((A->B->C)->(A->B)->A->C
```

3 The specification language μCRL

The syntax and semantics of μCRL are described in the style of ACP, which stands for Algebra of Communicating Processes [2]. We give a brief overview of the most important features.

3.1 Specifications in μCRL

The specification language μCRL is based on CRL (*Common Representation Language*). It has been developed by J.F. Groote and A. Ponse in 1990. The language μCRL consists of process algebra extended with abstract data types.

Assume a set $\mathcal{N}$ of so called *names*. These names are used to denote sorts, variables, functions, processes and labels of actions. An element of $\mathcal{N}$ is a word over an alphabet not containing the following symbols:

$$\bot \quad + \quad \| \quad \lfloor\!\lfloor \quad | \quad \lhd \quad \rhd \quad \cdot \quad \partial \quad \delta \quad \tau \quad \rho \quad \sum \quad \surd \quad \times \quad \rightarrow \quad : \quad = \quad , \quad) \; (\quad \} \; \{ \tag{5}$$

Moreover, $\mathcal{N}$ does not contain "a space", "a newline" and the reserved keywords **sort, proc, var, act, func, comm, rew** and **from**.

In μCRL *data terms* and *process terms* are distinguished. In Backus-Naur notation the definition of process terms is as follows:

DEFINITION 3.1.1 An expression p is called a *process-term* iff p has the following syntax:

$$\begin{aligned} p \quad ::= \quad & (p+p) \mid (p \cdot p) \mid (p \,\|\, p) \mid (p \lfloor\!\lfloor p) \mid (p \mid p) \mid (p \lhd t \rhd p) \mid \textstyle\sum(d:D,p) \mid \partial(\{n_1,\ldots,n_m\},p) \mid \\ & \tau(\{n_1,\ldots,n_m\},p) \mid \rho(\{n_1 \rightarrow n'_1,\ldots,n_m \rightarrow n'_m\},p) \mid \delta \mid \tau \mid n(t_1,\ldots,t_k). \end{aligned}$$

where $k \geq 0$, $n, n_i, n'_i \in \mathcal{N}$. The t, t_i stand for data terms (explained later), d is a variable and D denotes a sort name. Brackets are omitted according to the convention that $\cdot$ binds strongest, the conditional construct binds stronger than the parallel operators which in turn bind stronger than $+$. We briefly describe the behaviour of the different processes.

- $p + q$ behaves like p or like q.
- $p \cdot q$ behaves like q after p.
- $p \,\|\, q$ denotes the interleaving of p and q, except that actions of both arguments may communicate if explicitly allowed in a communication specification.
- $p \lfloor\!\lfloor q$ behaves like $p \,\|\, q$ except that the first step must originate from p.
- $p \mid q$ behaves like $p \,\|\, q$ except that the first step is a communication action between p and q.
- $p \lhd t \rhd q$ behaves like p when t evaluates to *true* and as q when t evaluates to *false*.
- $\sum(d:D,p(d))$ behaves like $p(d_1)+p(d_2)+\cdots$, where $p(d)$ denotes a process term which possibly contains d and $D = \{d_1, d_2, \ldots\}$.
- $\partial(\{n_1,\ldots,n_m\},p)$ renames n_i to δ.
- $\tau(\{n_1,\ldots,n_m\},p)$ renames n_i to τ.
- $\rho(\{n_1 \rightarrow n'_1,\ldots,n_m \rightarrow n'_m\},p)$ renames n_i to n'_i.
- δ describes the process that cannot do anything, in particular it cannot terminate.
- τ represents internal activity that cannot be observed.
- $n(t_1,\ldots,t_k)$ stands for an action or process with k parameters.

The syntax of *specifications in μCRL* is illustrated in Tables 3,4 and 5. For details we refer to [10].

Consider the μCRL specification *ALTERNATE* in Table 3. Here, *true*, *false* and O are called *constants* and S, *even* and $\neg$ are called *functions*.

sort	**Bool**		
	nat		
var	x	$: nat$	
func	$true, false$	$:\rightarrow \mathbf{Bool}$	
	$\neg$	$: \mathbf{Bool} \rightarrow \mathbf{Bool}$	
	O	$:\rightarrow nat$	
	S	$: nat \rightarrow nat$	
	$even$	$: nat \rightarrow \mathbf{Bool}$	
act	a	$: \mathbf{Bool}$	
rew	$\neg(true) = false$		B1
	$\neg(false) = true$		B2
	$even(O) = true$		E1
	$even(S(x)) = \neg(even(x))$		E2
proc	$p(y : nat) = a(even(y)) \cdot p(S(y))$		P
	$q(b : \mathbf{Bool}) = a(b) \cdot q(\neg(b))$		Q

Table 3: The μCRL specification *ALTERNATE*.

sort	**Bool**	
	D	
act	$r_1, r_2, c_2, s_2, s_3 \quad : D$	
proc	$B_{12} = \Sigma(d : D, r_1(d) \cdot s_2(d)) \cdot B_{12}$	B12
	$B_{23} = \Sigma(d : D, r_2(d) \cdot s_3(d)) \cdot B_{23}$	B23
	$B_{123} = \tau(\{c_2\}, \partial(\{r_2, s_2\}, B_{12} \parallel B_{23}))$	B123
comm	$r_2 \mid s_2 = c_2$	

Table 4: The μCRL specification *BUFFER*.

3.2 Data- and process variables

Let E be a μCRL specification.

DEFINITION 3.2.1 E is called *well-formed* if

1. E is *statically semantically correct* [10],
2. E has no empty sorts,
3. The communication function is associative, i.e. : $(p_1 \mid p_2) \mid p_3 = p_1 \mid (p_2 \mid p_3)$,
4. The booleans are defined.

In order to express general properties that a specification may have μCRL consists of two kinds of *variables*.

DEFINITION 3.2.2 A finite set V_d containing elements of the form $\langle d : D \rangle$ with d some name is called a set of *data variables over* E iff

1. the name D is declared as a sort in E,
2. d is not a constant, or an unparameterized action or process from E,

sort	**Bool**		
	D		
	bag		
var	b_1, b_2	$: \mathbf{Bool}$	
	b, c	$: bag$	
	d, e	$: D$	
func	$true, false$	$:\rightarrow \mathbf{Bool}$	
	if_bool	$: \mathbf{Bool} \times \mathbf{Bool} \times \mathbf{Bool} \rightarrow \mathbf{Bool}$	
	if_bag	$: \mathbf{Bool} \times bag \times bag \rightarrow bag$	
	eq	$: D \times D \rightarrow \mathbf{Bool}$	
	$\varnothing$	$:\rightarrow bag$	
	$test$	$: D \times bag \rightarrow \mathbf{Bool}$	
	$in,\ rem$	$: D \times bag \rightarrow bag$	
	con	$: bag \times bag \rightarrow bag$	
act	r, s	$: D$	
rew	$if\_bool(true, b_1, b_2) = b_1$		IF1
	$if\_bool(false, b_1, b_2) = b_2$		IF2
	$if\_bag(true, b, c) = b$		IF3
	$if\_bag(false, b, c) = c$		IF4
	$eq(d, d) = true$		EQ
	$test(d, \varnothing) = false$		BAG1
	$test(d, in(e, b)) = if\_bool(eq(d, e), true, test(d, b))$		BAG2
	$in(d, in(e, b)) = in(e, in(d, b))$		BAG3
	$rem(d, \varnothing) = \varnothing$		BAG4
	$rem(d, in(e, b)) = if\_bag(eq(d, e), b, in(e, rem(d, b)))$		BAG5
	$con(\varnothing, b) = b$		BAG6
	$con(in(d, b), c) = in(d, con(b, c))$		BAG7
proc	$Bag(x : bag) =$	$\Sigma(d : D, r(d) \cdot Bag(in(d, x)))+$	
		$\Sigma(d : D, s(d) \cdot Bag(rem(d, x)) \lhd test(d, x) \rhd \delta)$	BAG

Table 5: The μCRL specification BAG.

3. for each sort $D' \not\equiv D$ of E it holds that $\langle d : D' \rangle \notin V_d$.

Let V_d be a set of data variables over E.

DEFINITION 3.2.3 A finite set V_p of names is called a set of *process variables over E and V_d* iff none of its elements occur as a variable in V_d.

We write E, V_d, V_p when E is a well-formed specification with data variables V_d and process variables V_p.

DEFINITION 3.2.4 A *data term over E, V_d, V_p* is either a constant from E, a variable from V_d or an application of a function from E to data terms over E, V_d, V_p of the appropiate sort. A data term is called *closed* iff it does not contain any variables from V_d.

DEFINITION 3.2.5 A *process term over E, V_d, V_p* is defined inductively over the syntax given in Definition 3.1.1

- $p \circ q$ with $\circ \in \{+, \cdot, \|, \between, |, \lhd\ t\ \rhd\}$ and t a data term over E, V_d, V_p of sort **Bool**, is a process term over E, V_d, V_p if both p and q are,
- $\Sigma(d : D, p)$ is a process term over E, V_d, V_p if p is a process term over

$$E, (V_d \backslash \{\langle d : n \rangle | n \in \mathcal{N}\}) \cup \{\langle d : D \rangle\}, V_p \backslash \{d\},$$

- $C(\{n_1, \ldots, n_k\}, p)$ with $C \in \{\partial, \tau\}$ is a process term over E, V_d, V_p if p is, and the n_i are labels of actions from E,
- $\rho(\{n_1 \rightarrow n'_1, \ldots, n_m \rightarrow n'_m\}, p)$ is a process term over E, V_d, V_p if p is, and the n_i are labels of actions from E such that if $n_i : S_1 \times \cdots \times S_k$ is an action declaration in E then so is $n'_i : S_1 \times \cdots \times S_k$,
- δ and τ are process terms over E, V_d, V_p, from E or if $n \in V_p$,
- $n(t_1, \ldots, t_k)$ is a process term over E, V_d, V_p if either E contains an action declaration of the form $n : S_1 \times \cdots \times S_k$ or a process declaration of the form $n(x_1 : S_1, \ldots, x_k : S_k) = q$ and any t_i is a data term over E, V_d, V_p of sort S_i,

A process term is called *closed* iff it does not contain any variables from $V_d \cup V_p$.

Note: $k = 0$ is allowed in Definition 3.1.1 and Definition 3.2.5.

DEFINITION 3.2.6 A *formula over* E, V_d, V_p is defined inductively in the following way:

- $\perp$ is a formula over E, V_d, V_p.
- $t = u$ is a formula over E, V_d, V_p iff
  - either t and u are data terms over E, V_d, V_p that are of the same sort,
  - or t and u are process terms over E, V_d, V_p.
- $\phi \rightarrow \psi$ is a formula over E, V_d, V_p iff both ϕ and ψ are formulas over E, V_d, V_p.

ϕ is a formula over E, V_d, V_p will be denoted as

$$\phi \quad \textbf{from } E, V_d, V_p$$

The following abbreviations are used:

$$\begin{aligned} \neg\phi &\stackrel{\text{def}}{=} \phi \rightarrow \perp, \\ \phi \vee \psi &\stackrel{\text{def}}{=} (\neg\phi) \rightarrow \psi, \\ \phi \wedge \psi &\stackrel{\text{def}}{=} \neg(\phi \rightarrow \neg\psi), \end{aligned} \tag{6}$$

4 The implementation of a μCRL specification

In the following sections we encode process algebra notions in COQ, which is based on higher order typed λ-calculus. We frequently use the higher order features of COQ. Consequently we actually work in a much stronger and more expressive formalism than μCRL. This raises two problems. First, we would like to be able to prove every theorem of μCRL also in COQ. This appears to be unproblematic since COQ is so much stronger than μCRL. The second problem, which is the converse of the first, is not so easily solved by handwaving. The question is the following: if a type of COQ encodes a formula of μCRL, is this formula then necessarily also a theorem of μCRL? One could weaken this question as follows: if a theorem of COQ encodes a formula of μCRL, is this formula true in the preferred semantic of μCRL (e.g. weak bisimulation semantics [...])? We need at least a positive answer to the weaker question (which of course would follow from a positive answer to the stronger question). In this paper we leave these questions aside, since we first want to explore whether our type theoretic approach is at all feasible, trusting that both questions have positive answers. If not in general, then at least under suitable conditions that can be met.

The encoding is divided in two parts, i.e.

- The general declarations that are independent of the specification. These are explained in Section 6 and Subsection 4.5.
- The specific declarations that are depending on your specification. These are explained in this section.

In order to make the action- and the process declarations more readable we introduce some abbreviations on the meta level. We recall the definition of `prod` from Subsection 2.3.

```
Syntax prod = "_*_".
Syntax pair = "<_,_>(_,_)".
Inductive Definition prod [A,B:Set]:Set = pair:A->B->(A*B).
```

Now we define the following abbreviations for n-tuples:

$$\begin{aligned}
\texttt{prod } \texttt{s}_1 \cdots \texttt{s}_n &\stackrel{\text{def}}{=} \texttt{prod ((prod } \texttt{s}_1 \cdots \texttt{s}_{n-1}\texttt{) s}_n\texttt{)} && \text{for all } n \geq 3\\
(\texttt{x}_1, \texttt{x}_2) &\stackrel{\text{def}}{=} \texttt{<s}_1, \texttt{s}_2\texttt{>(x}_1, \texttt{x}_2\texttt{)}\\
(\texttt{x}_1, \ldots, \texttt{x}_n) &\stackrel{\text{def}}{=} \texttt{<prod s}_1 \cdots \texttt{s}_{n-1}, \texttt{s}_n\texttt{>((x}_1, \ldots, \texttt{x}_{n-1}\texttt{), x}_n\texttt{)} && \text{for all } n \geq 3
\end{aligned}$$

Note that $(\texttt{x}_1, \ldots, \texttt{x}_n)$ is of type $(\texttt{prod } \texttt{s}_1 \cdots \texttt{s}_n)$ when $\texttt{x}_i$ is of type $\texttt{s}_i$ for $i = 1, \ldots, n$. In COQ you can define projection functions by using Elim-terms and ι-reductions.

```
Syntax fst = "<_,_>Fst(_)".
Syntax snd = "<_,_>Snd(_)".
Definition fst = [A,B:Set][x:A*B](<A>Match x with [a:A][b:B]a).
Definition snd = [A,B:Set][x:A*B](<B>Match x with [a:A][b:B]b).
```

Let `x` be of type $\texttt{prod } \texttt{s}_1 \cdots \texttt{s}_n$. For $n \geq 2$ we define

$$\begin{aligned}
\texttt{x}^0 &\stackrel{\text{def}}{=} \texttt{x}\\
\texttt{x}^{i+1} &\stackrel{\text{def}}{=} \texttt{<(prod } \texttt{s}_1 \cdots \texttt{s}_{n-(i+1)}\texttt{), s}_{n-i}\texttt{>Fst(x}^i\texttt{)} && \text{for all } i = 0, \ldots, n-3\\
\pi_1(\texttt{x}) &\stackrel{\text{def}}{=} \texttt{<s}_1, \texttt{s}_2\texttt{>Fst(x}^{n-2}\texttt{)}\\
\pi_2(\texttt{x}) &\stackrel{\text{def}}{=} \texttt{<s}_1, \texttt{s}_2\texttt{>Snd(x}^{n-2}\texttt{)}\\
\pi_i(\texttt{x}) &\stackrel{\text{def}}{=} \texttt{<(prod } \texttt{s}_1 \cdots \texttt{s}_{i-1}\texttt{), s}_i\texttt{>Snd(x}^{n-i}\texttt{)} && \text{for all } i = 3, \ldots, n
\end{aligned}$$

Obviously we have that $\pi_i(\texttt{x}_1, \ldots, \texttt{x}_n) = \texttt{x}_i$ for all $i = 1, \ldots, n$.

4.1 The declaration of the sorts

In μCRL one has a notion of *constructors*. This notion is defined on a meta level. The proof theory consists of a reduction rule *IND* which enables you to reduce the problem to its constructors. In μCRL the notion of constructor is defined as follows.

DEFINITION 4.1.1 Let E be a well-formed specification. Let s be a sort of E, and C_s a subset of the function declarations occuring in E, then C_s is called a *set of constructors of* s iff

1. all functions in C_s have target sort s,
2. any closed data term of sort s can be proved equal to a data term that is obtained from appplications of the functions in C_s.

When $c \in C_s$ then c is allowed as a constructor for s in COQ but not vice versa, Function types in μCRL are of the form $A \times B \to C$. We can represent functions of type $A \to (B \to C)$ in μCRL by identifying $A \times B \to C$ with $A \to (B \to C)$. However, functions of type $(A \to B) \to C$ are not possible in μCRL. For instance

$$\{root : nat \to nat\_tree\ ,\ join : nat \to (\mathbf{Bool} \to nat\_tree) \to nat\_tree\}$$

makes sense as a set of constructors for the sort of the binary trees, with nodes labeled with natural numbers, but is not allowed in μCRL. The induction principle would be

$\Pi P : nat\_tree \to *^p .$
$(\Pi x : nat . P\ (root\ x)) \to$
$(\Pi x : nat . \Pi J : \mathbf{Bool} \to nat\_tree . (\Pi b : \mathbf{Bool} . J\ b) \to P\ (join\ x\ J)) \to$
$\Pi t : nat\_tree . P\ t$

When we want to make us of the meta property that C_s is a set of constructors of sort s then we have to declare s as an inductive type.

A sort declaration is of the following form:

sort **Bool**
s

We chose `Set` for the representation of the μCRL notion **sort**. This means that the sorts of the specification are declared as inhabitants of `Set`.

```
Parameter s:Set.
```

When $\{c_1, \ldots, c_k\}$ is a set of constructors of s, and c_i is of type s_i for $i = 1, \ldots, k$, then we define:

```
Inductive Set s:Set = c1 : s1 | ··· | ck : sk .
```

The sort **Bool** is represented by the inductive type `bool` given in Formula 2. In Subsection 2.4 we promised to give some examples of the `Elim`-instruction. We fulfil this promise in Example 4.1.2.

EXAMPLE 4.1.2 Let $\langle b : \mathbf{Bool} \rangle \in V_d$ and assume we want to prove a proposition $\phi(b)$.

```
1 subgoal
 φ(b)
 ============================
 b         :  bool
 bool_ind  :  (P:bool->Prop)(P true)->(P false)->(x:bool)(P x)

Coq < Elim b.          In this case means:  Apply (bool_ind [z:bool](φ(z))).

2 subgoals
 φ(true)
 ============================
 b         :  bool
 bool_ind  :  (P:bool->Prop)(P true)->(P false)->(x:bool)(P x)

subgoal 2 is:
 φ(false)
```

In COQ-sessions the automatically generated inhabitants of the different induction principles (`bool_ind` in this example) are not exposed.

When we run COQ we may omit the declaration of the booleans because it is already done in the file *prelude.v* [7] which is automatically loaded in the initial state of the system.

4.2 The declaration of the functions and constants

A function- or constant declaration in E has the following form:

func $f : s_1 \times \cdots \times s_k \to s_0$

where s_i is a sort of E for $i = 0, \ldots, k$. (when $k = 0$ then f is a constant, else f is a function)
The declaration in COQ (assuming that f is not seen as a constructor of s_0) is:

```
Parameter f:s1->...->sk->s0.
```

The μCRL notion $\rightarrow$ is represented by `->`. In fact we view f as a function of type $s_1 \rightarrow \cdots \rightarrow s_k \rightarrow s_0$. This way we avoid the complexity that comes with the use of

```
prod:Set->Set->Set
```

which is the term we use to implement the μCRL operator

$$\times : \mathbf{sort} \rightarrow \mathbf{sort} \rightarrow \mathbf{sort}$$

expressing the cartesian product between sets.

4.3 The declaration of the variables and the rewrite rules

The inductive definition of equality is used for the implementation of the μCRL symbol $=$. We recall this definition which is declared in the initial file prelude.v.

```
Syntax eq = "<_>_=_".
Inductive Definition eq [A:Set;a:A]:A->Prop = refl_equal:<A>a=a.
```

Assume a μCRL specification E and let

rew $x = y$ REW

be a rewrite rule of E, then x and y are data terms of sort s, for some sort s declared in E. Furthermore, let

var $v : s'$

be a variable declaration of E and then we define:

```
Section REWRITE.
Variable v:s'.
Axiom REW. Assumes <s>x=y.
End REWRITE.
```

EXAMPLE 4.3.1 Assume we want to prove some formula $\phi(y)$ in a context containing a proof for $x = y$ (where x and y are data terms of sort s).

```
1 subgoal
 φ(y)
 ============================
 x       :  s
 y       :  s
 H       :  <s>x=y
 eq_ind  :  (A:Set)(a:A)(P:A->Prop)(P a)->(b:A)(<A>a=b)->(P b)
```

`Coq < Elim H.` In this case means: `Apply (eq_ind s y [z:s](φ(z))).`

```
1 subgoal
 φ(x)
 ============================
 x       :  s
 y       :  s
 eq_ind  :  (A:Set)(a:A)(P:A->Prop)(P a)->(b:A)(<A>a=b)->(P b)
```

4.4 The declaration of the actions

We introduce the following denotations.

$$\begin{aligned} \mathcal{A}^E &\stackrel{\text{def}}{=} \{a \mid a \text{ is an atomic action of } E\} \\ \mathcal{A}^E_{\delta,\tau} &\stackrel{\text{def}}{=} \mathcal{A}^E \cup \{\delta, \tau\}. \end{aligned}$$

usually we will omit the superscript E. Assume that E has the following action declaration.

$$\begin{aligned} \textbf{act} \quad & a_1 : s_{1,1} \times \cdots \times s_{1,k_1} \\ & \vdots \\ & a_m : s_{m,1} \times \cdots \times s_{m,k_m} \end{aligned}$$

We implement this actions as an inductive set. One of the advantages of an inductive type `act` is that we have a proof for

$$\sim \texttt{<act>a}_i = \texttt{a}_j$$

for $i \neq j$. This property is needed for evaluating whether the property (`in_ehlist L a`), stating that action `a` is in the list of actions `L`, holds or not. (The definition for `in_ehlist` is given in Subsection 6.7) Another advantage of the inductive type for actions is that we can use an Elim-term for the implementation of the communication function γ.

One would expect

$\texttt{a}_i : (\texttt{prod s}_{i,1} \cdots \texttt{s}_{i,k_i})\texttt{->act}$ or
$\texttt{a}_i : \texttt{s}_{i,1}\texttt{->} \cdots \texttt{->s}_{i,k_i}\texttt{->act}$

for the implementation of action a_i for $i = 1, \ldots, m$. However, the communication function is easier to define when only the *names* of the actions are declared. (When $a(d)$ is an action then a is called the *name* of that action.) The data is added when we interpret our action as a process (see Subsection 4.5). This makes the communication function easier to define. We declare the actions as follows:

$\texttt{Inductive Definition act:Set = a}_1\texttt{:act | } \cdots \texttt{ | a}_m\texttt{:act | DELTA:act | TAU:act.}$

The action `DELTA` represents the process δ and the action `TAU` represents the process τ. It is more convenient to see δ and τ as actions because many axioms (see for instance Table 12) are quantified over $\mathcal{A}_{\delta.\tau}$.

4.5 The declaration of the processes

It is natural to implement the processes as an inductive set, for the set of processes PROC is the smallest set such that $x + y \in$ PROC, $x \cdot y \in$ PROC, etc. whenever x and y are processes. (One has to add a constructor that constructs infinite processes from their defining equations.) The inductive definition for processes would have the following form:

```
Inductive Definition proc:Set =  alt:proc->proc->proc |
                                 seq:proc->proc->proc |
                                          ⋮
```

where (`alt x y`) stands for `(x + y)` and (`seq x y`) stands for $(\texttt{x} \cdot \texttt{y})$.

However, a strong reason to use a non-inductive definition is that we don't want to be able to have a case analysis on process constructors. Together with the use of Leibniz equality by the implementation of the axioms of ACP, given in subsection 6.5, it leads to inconsistenty, as the following example shows.

EXAMPLE 4.5.1 Assume `z:proc` and define `P` of type `proc->Prop` as follows:
$\texttt{P} \equiv \texttt{[p:proc](<Prop>Match p with C}_1 \cdots \texttt{C}_{12}$ where
$\texttt{C}_1 \equiv \texttt{[q}_1\texttt{:proc][Q}_1\texttt{:Prop][q}_2\texttt{:proc][Q}_2\texttt{:Prop]False}$,
$\texttt{C}_2 \equiv \texttt{[q}_1\texttt{:proc][Q}_1\texttt{:Prop][q}_2\texttt{:proc][Q}_2\texttt{:Prop]True}$ and

$C_3, \ldots, C_{12}$ are terms corresponding to the other constructors of `proc`. Then
`P (alt (seq z z) (seq z z))` $\twoheadrightarrow$
C_1 `(seq z z) (P (seq z z)) (seq z z) (P (seq z z))` $\twoheadrightarrow$ `False` and
`P (seq z z)` $\twoheadrightarrow$ C_2 `z (P z) z (P z)` $\twoheadrightarrow$ `True`.
When we combine this with the inhabitant `A3` of type `(x:proc)<proc>x=(alt x x)`, as declared in subsection 6.5, we can construct the following inhabitant of `False`:
`eq_ind proc P (seq z z) I (alt (seq z z) (seq z z)) (A3 (seq z z))`

The non-inductive implementation we used is given below. Note that this part of the the implementation is not depending on any specification.

```
Syntax alt   = "_ + _".
Syntax seq   = "_ * _".
Syntax mer   = "_ || _".
Syntax Lmer  = "_ |L _".
Syntax comm  = "_ | _".
Syntax cond  = "_ < _ > _".
Syntax sum   = "+{ _ , _ }".
Parameter proc  :Set.
Parameter ALT   :proc->proc->proc.
Parameter SEQ   :proc->proc->proc.
Parameter MER   :proc->proc->proc.
Parameter LMER  :proc->proc->proc.
Parameter COMM  :proc->proc->proc.
Parameter COND  :proc->bool->proc->proc.
Parameter SUM   :(D:Set)(D->proc)->proc.
Parameter hide  :(list name)->proc->proc.
Parameter enc   :(list name)->proc->proc.
Parameter ren   :(name*name)->proc->proc.
Parameter ia    :(D:Set)D->act->proc.
Definition alt   = [x,y:proc](ALT x y).
Definition seq   = [x,y:proc](SEQ x y).
Definition mer   = [x,y:proc](MER x y).
Definition Lmer  = [x,y:proc](LMER x y).
Definition comm  = [x,y:proc](COMM x y).
Definition cond  = [x:proc][b:bool][y:proc](COND x b y).
Definition sum   = [D:Set][x:D->proc](SUM D x).
```

Now (`ia D a d`) stands for the interpretation of action $a(d)$ for some action a, parameterized with data d of type D, as a process. Note that the type of `ia` forces that every action is parameterized with exactly one data type. This is achieved when $(D_1 \times \cdots \times D_k)$ is declared as (`prod` $\mathtt{D}_1 \cdots \mathtt{D}_k$) and unparameterized actions are parameterized with a dummy. This dummy is declared as an inductive set containing only one element.

```
Inductive Definition one:Set = i:one.
```

For reasons of readability we introduce the following abbreviations:

```
Definition delta  = (ia one DELTA i).
Definition tau    = (ia one TAU i).
```

The only difference between for instance `ALT` and `alt` is that the latter is a definition which makes it possible to introduce a syntax. Note that we have the same syntax `"_*_"` for `prod` and `seq`. This means that `A*B` stands for `(prod A B)` when `A` and `B` are of type `Set` and `A*B` stands for `(seq A B)` when `A` and `B` are of type `proc`.

The specification-dependent part of the process declaration implements the process declarations of E. Let

proc $p(x_1 : s_1, \ldots, x_n : s_n) = \phi(p, x_1, \ldots, x_n)$ P

be a process declaration of E for $n \geq 0$. $\phi(p, x_1, \ldots, x_n)$ is a process possibly containing $x_1, \ldots, x_n$ and p. For $n = 0$ the implementation will be:

```
Section PROC.
Axiom P. Assumes <proc>p = φ(p).
End PROC.
```

For $n = 1$ the implementation will be:

```
Section PROC.
Variable x : s_1.
Axiom P. Assumes <proc>(p x) = φ(p, x).
End PROC.
```

For $n \geq 2$ the implementation will be:

```
Section PROC.
Variable x:(prod s_1 ··· s_n).
Axiom P. Assumes <proc>(p x) = φ(p, π_1(x), ..., π_n(x)).
End PROC.
```

4.6 The declaration of the communication function γ

Assume the following communication declarations in the specification E.

$$\begin{array}{ll} \textbf{comm} & \alpha_1^1 \mid \alpha_1^2 = \alpha_1^3 \\ & \quad \vdots \\ & \alpha_r^1 \mid \alpha_r^2 = \alpha_r^3 \end{array}$$

where α_j^i ranges over the set of atomic actions $\{a_1, \ldots, a_m\}$. In fact the communication function is a total function

$$\gamma : \mathcal{A} \times \mathcal{A} \longrightarrow \mathcal{A} \cup \{\delta\}$$

because $\gamma(a, a') = \delta$ when $a \mid a'$ is not mentioned in the specification. When we extend γ to a function on $\mathcal{A}_{\delta,\tau} \times \mathcal{A}_{\delta,\tau}$ by stating

$$\gamma(a, a') \stackrel{\text{def}}{=} \delta \quad \text{when } \{a, a'\} \cap \{\delta, \tau\} \neq \varnothing$$

then we can represent γ by gamma of type (act->act->act) which is defined as follows:

```
Definition gamma =  [a,b:act](<act>Match a with
                      (<act>Match b with γ(a_1, a_1) ··· γ(a_1, a_m) DELTA DELTA)
                                          ⋮
                     (<act>Match b with γ(a_m, a_1) ··· γ(a_m, a_m) DELTA DELTA)
                      DELTA
                      DELTA
```

where $\mathtt{a}_1, \ldots, \mathtt{a}_m$ are the actions of E. The ι-reduction on Elim-terms satisfies

$$(\mathtt{gamma}\ \mathtt{a}_i\ \mathtt{a}_j) \twoheadrightarrow \gamma(\mathtt{a}_i, \mathtt{a}_j)$$

for all $\mathtt{a}_i, \mathtt{a}_j \in \mathcal{A}_{\delta,\tau}$.

4.7 Remarks

(i): Atomic formulas of μCRL are of the form $t = u$, where t and u are process terms or data terms of the same sort. The implementation of this formula is of type `Prop`:

`<proc>t=u`	if t and u are process terms.
`<s>t=u`	if t and u are data tems of sort s.

The μCRL formula $\bot$ is implemented as `False` which is also of type `Prop`. When `P:Prop` and `Q:Prop` are implementations of the μCRL formulas ϕ and ψ, then `P->Q:Prop` is the implementation of $\phi \to \psi$. By induction we have that all μCRL formulas are implemented as inhabitants of `Prop`.

(ii): Let E be a μCRL specification, V_d a set of data variables and V_p a set of process variables, then we will write

$$\Gamma(E, V_d, V_p)$$

for the smallest context in λIC that contains all the encodings off the the declarations of E as well as the encodings of the elements of V_d and V_p as given in this section.

5 The rules for logical deduction

The proof theory for μCRL is given in a natural deduction format. A deduction is seen as a tree of which each node is labelled with a formula. The leaves of the tree are the *hypotheses* of the deduction. Cancelled hypotheses are placed between brackets. We give the definitions and the proofs of their implementations in COQ.

5.1 Logical deductions in μCRL

Logical deductions are defined in a recursive way.

DEFINITION 5.1.1 $\mathcal{D}$ is called a *logical deduction* if

- $\mathcal{D}$ is a single node tree with as label a formula

$$\phi \quad \textbf{from } E, V_d, V_p. \tag{7}$$

- $\mathcal{D}$ is constructed out of natural deductions $\mathcal{D}_1$ and $\mathcal{D}_2$ using the rules given in Table 6 under the following restrictions:
  - In applications of the rule $(\to I)$ and the rule *RAA* (Reductio Ad Absurdum) all open assumptions of the form indicated by $[\ldots]$ are cancelled.
  - In applications of $(\to I)$, *RAA, REFL, VAR* and *SUB* the conclusion must be a formula.
  - In applications of *SUB* the variable x may not be free in any hypothesis of $\mathcal{D}_1$.
  - Each application of *VAR* is restricted to one of the following cases:
    - $V_d \subseteq V_d'$ or $V_d' \subseteq V_d$, and $V_p = V_p'$.
    - $V_p \subseteq V_p'$ or $V_p' \subseteq V_p$, and $V_d = V_d'$.

5.2 The proofterms corresponding with the logical deduction rules

We briefly explain the relation between the different rules of Table 6 and corresponding COQ features.

- The rule $(\to I)$ of Table 6 corresponds with the $(\forall I)$ rule of λIC.
- Similar, the rule $(\to E)$ corresponds with $(\forall E)$.

$$\frac{\begin{array}{c}[\phi \quad \textbf{from } E, V_d, V_p] \\ \mathcal{D}_1 \\ \psi \quad \textbf{from } E, V_d, V_p\end{array}}{\phi \to \psi \quad \textbf{from } E, V_d, V_p}\,(\to I) \qquad \frac{}{\phi \quad \textbf{from } E, V_d, V_p}\,\text{name of } \phi$$

$$\frac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ \phi \quad \textbf{from } E, V_d, V_p & \phi \to \psi \quad \textbf{from } E, V_d, V_p\end{array}}{\psi \quad \textbf{from } E, V_d, V_p}\,(\to E)$$

$$\frac{\begin{array}{c}[\neg\phi \quad \textbf{from } E, V_d, V_p] \\ \mathcal{D}_1 \\ \bot \quad \textbf{from } E, V_d, V_p\end{array}}{\phi \quad \textbf{from } E, V_d, V_p}\,RAA \qquad \frac{}{t = t \quad \textbf{from } E, V_d, V_p}\,REFL$$

$$\frac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ \phi[t/x] \quad \textbf{from } E, V_d, V_p & t = u \quad \textbf{from } E, V_d, V_p\end{array}}{\phi[u/x] \quad \textbf{from } E, V_d, V_p}\,REPL$$

$$\frac{\begin{array}{c}\mathcal{D}_1 \\ \phi \quad \textbf{from } E, V_d, V_p\end{array}}{\phi[t/x] \quad \textbf{from } E, V_d, V_p}\,SUB \qquad \frac{\begin{array}{c}\mathcal{D}_1 \\ \phi \quad \textbf{from } E, V_d, V_p\end{array}}{\phi \quad \textbf{from } E, V'_d, V'_p}\,VAR$$

Table 6: Rules for logical deductions

- There is a rule for every axiom of μCRL. These axioms are declared in COQ as given in Section 6.
- The μCRL formula $\bot$ is implemented as `False` (see Formula 1). When we have a proof `f` of `False` then `(False_ind` ϕ `f)` is a proof of ϕ. The dropping of $\neg\phi$ in the context makes RAA a classical rule. In general the term `f` can contain a free variable of type $\neg\phi$. In that case we cannot drop the hypothesis $\neg\phi$. In our implementation of μCRL we adapt an axiom `CL` of type `(P:Prop)(P \/ ~P)` which enables us to form

  `M` $\equiv$ `[P:Prop][H:~~P](or_ind P ~P P [x:P]x [x:~P](False_ind P (H x)) (CL P))`

  of type `(P:Prop)(~~P)->P`. Now we can take `(M` ϕ `([H:~` ϕ`]f))` instead of `(False_ind` ϕ `f)` as a proof for ϕ. The λ-abstraction from `H` binds all the free occurences of `H` in `f` and allows us to drop $\langle$`H :~` $\phi\rangle$ in the context.
- Assume that t is a data term of some sort D declared in specification E. Then we have a proof

  `(refl_equal D t)`

  of type `<D>t=t` which is the COQ interpretation for the formula

  $t = t \quad \textbf{from } E, V_d, V_p$

This corresponds with the rule *REFL* of Table 6.

- When we have a proof `e` of type `<D>t=u` and a proof `H` of ϕ(`t`) then

  `(eq_ind D t ([d:D]`ϕ`(d)) H u e)`

  is of type ϕ(`u`). This corresponds with rule *REPL* of Table 6.

- The rule *SUB* corresponds with $(\forall E)$ in λIC. When `H` is a proof for `(d:D)`ϕ`(d)` and `t` is of type `D`, then `(H t)` inhabits ϕ(`t`).

- The rule *VAR* is covered by the *START* rule [6] and the *WEAKENING* rule [6] of λIC.

6 The implementation of the modules

The proof theory for μCRL is based on *modules*. These modules are 'building blocks' consisting of axioms and rules that describe a feature of concurrency in a certain semantical setting. Some of the modules can be omitted in the declarations because all their axioms are provable in the context that follows from the implementation of the specification.

The declarations introduced in this section are not depending on the specification. Some preparatory definitions that are independent of the specification have to be declared first. We start with `ehlist` (Encapsulation-Hide-List) which is a list of actions (names) and `rlist` (Rename-list) which is a list of 'action pairs'. `ehlist` is used to describe the behaviour of `hide` and `enc`. `rlist` is used to describe the behaviour of `ren`.

```
Inductive Definition ehlist:Set  = ehnil:ehlist | ehcons:act->ehlist->ehlist.
Inductive Definition rlist:Set   = rnil:rlist | rcons:act->act->rlist->rlist.
```

Furthermore we need terms `In_ehlist:ehlist->act->Prop` and `In_rlist:act->act->rlist->Prop`. We want the following properties to hold: (informally exposed)

$$\begin{array}{lllll} \texttt{In\_ehlist a L} & \Longleftrightarrow & \texttt{a} \in \texttt{L} & \Longleftrightarrow & \texttt{L} \equiv \texttt{ehcons}(\cdots(\texttt{ehcons a L'})\cdots) \\ \texttt{In\_rlist } \texttt{a}_1\ \texttt{a}_2\ \texttt{R} & \Longleftrightarrow & (\texttt{a}_1 \rightarrow \texttt{a}_2) \in \texttt{R} & \Longleftrightarrow & \texttt{R} \equiv \texttt{rcons}(\cdots(\texttt{rcons } \texttt{a}_1\ \texttt{a}_2\ \texttt{R'})\cdots) \end{array}$$

for all `L:ehlist`, `R:rlist` and `a`, $\texttt{a}_1$, $\texttt{a}_2$`:act`. This is achieved when we introduce the following definitions:

```
Definition In_ehlist =  [a:act][L:ehlist](<Prop>Match L with False
                        [b:act][H:ehlist][P:Prop](<act>a=b)\/P)
Definition In_rlist =   [a1,a2:act][L:rlist] (<Prop>Match L with False
                        [b1,b2:act][H:rlist][P:Prop] ((<act>a1 = b1)/\(<act>a2 = b2))\/P)
```

Finally we introduce an axiom `EXT` which expresses that functions are determined by their applicative behaviour. Axiom `EXT` enables us to prove SUM11 as we will see in Subsection 6.9. Until now it is the only application for `EXT` so we could replace `EXT` by an interpretation `SUM11` of SUM11.

```
Axiom EXT. Assumes (A:Set)(x,y:A->proc)((a:A)<proc>(x a)=(y a))-><A->proc>x=y.
```

6.1 The Module BOOL

The module BOOL consists of two rules, B1 and B2, given in Table 7. The COQ implementations of this axioms are

```
B1:   ~<bool>true=false
B2:   (b:bool)(~<bool>b=true)-><bool>b=false
```

The first type is a special case of `~<A>a=b` which is inhabited as shown in Subsection 2.3. An inhabitant `B2` can be found by using the following tactics.

```
Intro ; Elim b ; Intro ; Apply (absurd <bool>true=true) ; Auto.
Assumption.
Auto.
```

B1:	$\neg(true = false)$	**from** $E, \varnothing, \varnothing$
B2:	$\neg(b = true) \rightarrow b = false$	**from** $E, \langle b : bool\rangle, \varnothing$

Table 7: The module BOOL

where `absurd` is an abbreviation for the term `[P,Q:Prop][x:P][f:~P](False_ind Q (f x))` of type `(P,Q:Prop)P->(~P)->Q`. This gives us the following term for `B2`:

```
(bool_ind [x:bool](~<bool>x=true)->(<bool>x=false)
[H:~<bool>true=true] (absurd <bool>true=true <bool>true=false (refl_equal bool true) H)
[H:~<bool>false=true](refl_equal bool false)
```

6.2 The Modules FACT and REC

The module FACT contains only one axiom. It states that the basic identities on data terms are those declared

FACT:	$t = u$	**from** $E, V_d, \varnothing$

Table 8: The module FACT

in the specification. V_d (in Table 8) is the set of data variables occurring in t and u. Assume that $t = u$ is a rewrite rule of specification E.
According to Subsection 4.3 we have `REW:<D>t=u` $\in \Gamma(E, V_d, \varnothing)$. Consequently $\Gamma(E, V_d, \varnothing) \vdash$ `REW:<D>t=u`.

The module REC states that the basic identities on process terms are those declared in the specification.

REC:	$p(x_1, \ldots, x_k) = q$	**from** $E, \{\langle x_1 : s_1\rangle, \ldots, \langle x_k : s_k\rangle\}, \varnothing$

Table 9: The module REC

Assume that $p(x_1 : s_1\ , \ldots,\ x_k : s_k) = q$ is a process declaration of specification E, then `P:<proc>(p x)=q` $\in \Gamma(E, \{\langle x_1 : s_1\rangle, \ldots, \langle x_k : s_k\rangle\}, \varnothing)$. Consequently $\Gamma(E, \{\langle x_1 : s_1\rangle, \ldots, \langle x_k : s_k\rangle\}, \varnothing) \vdash$ `P:<proc>(p x)=q`. (`x` is of type `prod` $\mathtt{s}_1 \cdots \mathtt{s}_n$.)

6.3 The Module IND

This module consists of rules *IND*(C) where C is a set of constructors some sort s. Assume that $\{\langle x : s\rangle\} \subseteq V_d$. Let

$$C \stackrel{\text{def}}{=} \{f_j : s_j^1 \times \cdots \times s_j^{l_j} \longrightarrow s \mid 1 \leq j \leq k\}$$

be a constructor set for the sort s of cardinality $k > 0$. We will call $l_j \geq 0$ the arity of constructor f_j. Assume we want to prove some formula

$$\phi \quad \textbf{from } E, V_d, V_p$$

that contains a variable x of type s. The rule *IND*(C) reduces the proof of ϕ to showing that property

IND:	$\left(\bigwedge_{j=1}^{k}\left(\bigwedge_{\sigma\in I_j}\sigma(\phi)\right)\rightarrow\phi[f_j(z_j^1,\ldots,z_j^{l_j})/x]\right)\rightarrow\phi$	**from** $E, V_d\cup\{\langle z_j^r : s_j^r\rangle \mid 1\leq r\leq l_j\}, V_p$

Table 10: The module IND. $0\leq j\leq k$

$\lambda x:s\,.\,\phi$ is preserved under the constructors of s. For each j the set I_j is a set of data substitutions over $E, V_d\cup\{\langle z_j^r : s_j^r\rangle \mid 1\leq r\leq l_j\}, V_p$ satisfying for $1\leq k\leq m$:

$$\sigma\in I_j \iff \sigma \text{ is the identity, except that it maps } x \text{ to some } y\in\{z_j^r \mid 1\leq r\leq l_j\}$$

Assume that the premise of the formula above is satisfied, i.e. we have proofs $\mathtt{h}_j$ for each $1\leq j\leq k$ satisfying the COQ implementation $\mathtt{H}_j$ of $\left(\bigwedge_{\sigma\in I_j}\sigma(\phi)\right)\rightarrow\phi[f_j(z_j^1,\ldots,z_j^{l_j})/x]$. Now

$$\mathtt{s\_ind\ [x:s]}\phi\mathtt{(x)\ h}_0\cdots\mathtt{h}_k$$

is of type $\mathtt{(x:s)}\phi\mathtt{(x)}$. This proves the rule *IND*(C) because (by convention) all the universal quantifiers are omitted in a μCRL formula.

6.4 The Module COND

The behaviour of the conditional operator is described in the module COND. It contains two axioms. Obvi-

COND1:	$x\lhd \mathit{true}\rhd y = x$	**from** $E, \varnothing, \{x, y\}$
COND2:	$x\lhd \mathit{false}\rhd y = y$	**from** $E, \varnothing, \{x, y\}$

Table 11: The module COND

ously the types `(x,y:proc)<proc>x=(cond x true y)` and `(x,y:proc)<proc>y=(cond x false y)` are not yet inhabited. We have to declare proofs for this propositions.

```
Section CONDITION.
Variable x,y:proc.
Axiom COND1.  Assumes <proc>x=(cond x true y).
Axiom COND2.  Assumes <proc>y=(cond x false y).
End CONDITION.
```

We give some examples of COQ proofs that are based on this module. Lemma 6.4.1 corresponds with Lemma 4.3.1. of [11].

LEMMA 6.4.1 In context ⟨`x,y,z:proc,b:bool`⟩ we have

(i) `<proc>(x + (x < b > delta)) = x`

(ii) `((<bool>b=true)->(<proc>x=y))->(<proc>(x < b > z)=(y < b > z))`

PROOF: (*i*):
```
bool_ind [b:bool](<proc>(x + (x < b > delta))=x)
(eq_ind proc x [p:proc](<proc>(x + p)=x)
  (eq_ind proc x [p:proc](<proc>p=x) (refl_equal proc x) (x + x) (A3 x))
  (x < true > delta) (COND1 x delta))
(eq_ind proc delta [p:proc](<proc>(x + p)=x)
  (eq_ind proc x [p:proc](<proc>p=x) (refl_equal proc x) (x + delta) (A6 x))
  (x < false > delta) (COND2 x delta)) b
```
(*ii*):
```
bool_ind [b:bool]((<bool>b=true)->(<proc>x=y))->(<proc>(x < b > z)=(y < b > z))
  [H:(<bool>true=true)->(<proc>x=y)](eq_ind proc y (eq proc (x < true > z))
    (eq_ind proc x [p:proc](<proc>p=y) (H (refl_equal bool true))
      (x < true > z) (COND1 x z))
    (y < true > z) (COND1 y z))
  [H:(<bool>false=true)->(<proc>x=y)](eq_ind proc z (eq proc (x < false > z))
    (eq_ind proc z [p:proc](<proc>p=z) (refl_equal proc z)
      (x < false > z) (COND2 x z))
    (y < false > z) (COND2 y z))
b
```
■

Even in this very simple example it is not really easy to read the proof. An alternative could be to give the list of tactics in stead of the proof term itself. This might be slightly better but it is not so easy to see on which subgoal a tactic is applied. (In this paper we always start on a new line when a subgoal is proved, but not vice versa). A solution for the denotational problem could be the use of *proof constellations* as proposed in [9].

The proof of 6.4.1 can be found using the following tactics:

(i):
```
Elim b.
Elim COND1 ; Elim A3 ; Auto.
Elim COND2 ; Elim A6 ; Auto.
```
(ii):
```
Elim b.
Intro ; Elim COND1 ; Elim COND1 ; Auto.
Intro ; Elim COND2 ; Elim COND2 ; Auto.
```

Note that the level of reasoning is quite satisfactory in this example. The tactics have a direct correspondence with the rules needed for the proof. In more complex examples (not illustrated in this paper) the level of reasoning sometimes was unbearably low. For instance, proving that x is not in the list $x_1, \ldots, x_k$ is done by proving $x \neq x_1, \ldots, x \neq x_k$.

LEMMA 6.4.2 In context ⟨x,y,z:proc,b:bool⟩ we have
`<proc>((x < b > y) |L z)=((x |L z) < b > (y |L z)).`

PROOF:
```
bool_ind [b:bool] (<proc>((x < b > y) |L z)=((x |L z) < b > (y |L z)))
(eq_ind proc (x |L z) (eq proc ((x < true > y) |L z))
  (eq_ind proc x [p:proc](<proc>(p |L z)=(x |L z))
    (refl_equal proc (x |L z)) (x < true > y) (COND1 x y))
  ((x |L z) < true > (y |L z)) (COND1 (x |L z) (y |L z)))
(eq_ind proc (y |L z) (eq proc ((x < false > y) |L z))
  (eq_ind proc y [p:proc](<proc>(p |L z)=(y |L z))
    (refl_equal proc (y |L z)) (x < false > y) (COND2 x y))
  ((x |L z) < false > (y |L z)) (COND2 (x |L z) (y |L z)))
b)
```
■

This proof can be found using the following tactics:

```
Elim b.
Elim COND1 ; Elim COND1 ; Auto.
Elim COND2 ; Elim COND2 ; Auto.
```

6.5 The module ACP

The module ACP consists of all the axioms that are standard in ACP. These axioms are divided in what we will call A-axioms, CM-axioms, CF-axioms and D-axioms. Let E be a μCRL specification containing actions a and b. The A-axiom describe the behaviour of the operators $\cdot$ and $+$. The CM-axioms describe the behaviour of the

A1:	$x + y = y + x$	**from** $E, \varnothing, \{x, y\}$
A2:	$x + (y + z) = (x + y) + x$	**from** $E, \varnothing, \{x, y, z\}$
A3:	$x + x = x$	**from** $E, \varnothing, \{x, y, z\}$
A4:	$(x + y) \cdot z = x \cdot z + y \cdot z$	**from** $E, \varnothing, \{x, y, z\}$
A5:	$(x \cdot y) \cdot z = x \cdot (y \cdot z)$	**from** $E, \varnothing, \{x, y, z\}$
A6:	$x + \delta = x$	**from** $E, \varnothing, \{x\}$
A7:	$\delta \cdot x = \delta$	**from** $E, \varnothing, \{x\}$
CM1:	$x \parallel y = x \between y + y \between x + x \mid y$	**from** $E, \varnothing, \{x, y\}$
CM2:	$a \between x = a \cdot x$	**from** $E, \varnothing, \{x, y\}$
CM3:	$a \cdot x \between y = a \cdot (x \parallel y)x$	**from** $E, \varnothing, \{x, y\}$
CM4:	$x + y \between z = x \between z + y \between z$	**from** $E, \varnothing, \{x, y, z\}$
CM5:	$a \cdot x \mid b = (a \mid b) \cdot x$	**from** $E, \varnothing, \{x\}$
CM6:	$a \mid b \cdot x \between y = (a \mid b) \cdot x$	**from** $E, \varnothing, \{x\}$
CM7:	$a \cdot x \mid b \cdot y = (a \mid b) \cdot (x \parallel y)$	**from** $E, \varnothing, \{x, y\}$
CM8:	$(x + y) \mid z = x \mid z + y \mid z$	**from** $E, \varnothing, \{x\}$
CM9:	$x \mid (y + z) = x \mid y + x \mid z$	**from** $E, \varnothing, \{x\}$
CF1:	$n_1(t_1, \ldots, t_m) \mid n_2(t_1, \ldots, t_m) = n_3(t_1, \ldots, t_m)$	**from** $E, V_d, \varnothing$, $n_1 \mid n_2 = n_3 \in \mathit{Comm}(E)$
CF2:	$a \mid b = \delta$	**from** $E, \varnothing, \varnothing$, $\mathit{label}(a) \mid \mathit{label}(b) \notin \mathit{Comm}(E)$
CF2':	$n_1(t_1, \ldots, t_m) \mid n_2(t'_1, \ldots, t'_m) = \delta$	**from** $E, V_d \cup V'_d, \varnothing$, $\exists i \in \{1, \ldots, m\}.\neg(t_i \neq t'_i)$
CF2":	$n_1(t_1, \ldots, t_m) \mid n_2(t'_1, \ldots, t'_{m'}) = \delta$	**from** $E, V_d \cup V'_d, \varnothing$, $m \neq m'$
D1:	$\partial(\{n_1, \ldots, n_m\}, a) = a$	**from** $E, \varnothing, \varnothing$, $\mathit{label}(a) \notin \{n_1, \ldots, n_m\}$
D2:	$\partial(\{n_1, \ldots, n_m\}, a) = \delta$	**from** $E, \varnothing, \varnothing$, $\mathit{label}(a) \in \{n_1, \ldots, n_m\}$
D3:	$\partial(nl, x + y) = \partial(nl, x) + \partial(nl, y)$	**from** $E, \varnothing, \{x, y\}$
D4:	$\partial(nl, x \cdot y) = \partial(nl, x) \cdot \partial(nl, y)$	**from** $E, \varnothing, \{x, y\}$

Table 12: The module ACP. $a, b \in \mathcal{A}_{\delta,\tau}$, nl is a list of names, $m \geq 0$ and $n_i \in \mathcal{N}$. V_d (resp. V'_d) contains all the data variables that occur in t_i (resp. t'_i).

parallel operators. The CF-axioms decribe the communication between the different actions. Let $\mathit{Comm}(E)$ be the commutative and associative closure of all communications declared in E ($\mathit{Comm}(E)$ is finite because E is well-formed). The D-axioms describe the behaviour of the encapsulation operator ∂.

The COQ implementations of these axioms are:

```
Section ACP.
Variable x,y,z  :proc.
Variable a,b    :act.
Variable D,E    :Set.
Variable d      :D.
Variable e      :E.
Variable a      :act.
Variable L      :ehlist.
```

```
Axiom A1.  Assumes <proc>(alt x y)=(alt y x).
Axiom A2.  Assumes <proc>(alt x (alt y z))=(alt (alt x y) z).
Axiom A3.  Assumes <proc>x=(alt x x).
Axiom A4.  Assumes <proc>(alt (seq x z) (seq y z))=(seq (alt x y) z).
Axiom A5.  Assumes <proc>(seq x (seq y z))=(seq (seq x y) z).
Axiom A6.  Assumes <proc>x=(alt x Delta).
Axiom A7.  Assumes <proc>Delta=(seq Delta x).
Axiom CM1.  Assumes <proc>(alt (alt (Lmer x y) (Lmer y x)) (comm x y))= (mer x y).
Axiom CM2.  Assumes <proc>(seq (ia D a d) x)=(Lmer (ia D a d) x).
Axiom CM3.  Assumes <proc>(seq (ia D a d) (mer x y))= (Lmer (seq (ia D a d) x) y).
Axiom CM4.  Assumes <proc>(alt (Lmer x z) (Lmer y z))=(Lmer (alt x y) z).
Axiom CM5.  Assumes <proc>(seq(comm (ia D a d) (ia E b e)) x)= (comm (seq (ia D a d) x) (ia E b e)).
Axiom CM6.  Assumes <proc>(seq (comm (ia D a d) (ia E b e)) x)= (comm (ia D a d) (seq (ia E b e) x)).
Axiom CM7.  Assumes <proc>(seq (comm (ia D a d) (ia E b e)) (mer x y))=
                     (comm (seq (ia D a d) x) (seq (ia E b e) y)).
Axiom CM8.  Assumes <proc>(alt (comm x z) (comm y z))=(comm (alt x y) z).
Axiom CM9.  Assumes <proc>(alt (comm x y) (comm x z))=(comm x (alt y z)).
Axiom D1.  Assumes (~(In_ehlist a L))-><proc>(ia D a d)= (enc L (ia D a d)).
Axiom D2.  Assumes (In_ehlist a L)-><proc>Delta=(enc L (ia D a d)).
Axiom D3.  Assumes <proc>(alt (enc L x) (enc L y))=(enc L (alt x y)).
Axiom D4.  Assumes <proc>(seq (enc L x) (enc L y))=(enc L (seq x y)).
End ACP.
```

6.6 The module SC

The proof theory for μCRL consists of a module SC (Standard Concurrency) to describe properties of the merge operators that are not standard in ACP. These axioms are derivable for process terms that are constructed from

SC1:	$(x \between y) \between z = x \between (y \parallel z)$	**from** $E, \varnothing, \{x, y, z\}$
SC2:	$x \between \delta = x$	**from** $E, \varnothing, \{x\}$
SC3:	$x \mid y = y \mid x$	**from** $E, \varnothing, \{x, y\}$
SC4:	$(x \mid y) \mid z = x \mid (y \mid z)$	**from** $E, \varnothing, \{x, y, z\}$
SC5:	$x \mid (y \between z) = (x \mid y) \between z$	**from** $E, \varnothing, \{x, y, z\}$

Table 13: The module SC

elements of $\mathcal{A}_{\delta,\tau}$.

The implementation of this module is straightforward.

```
Section STANDARD_COMMUNICATION.
Variable x,y,z:proc.
Axiom SC1.  Assumes <proc>(Lmer x (mer y z))=(Lmer (Lmer x y) z).
Axiom SC2.  Assumes <proc>(Lmer x Delta)=(seq x Delta).
Axiom SC3.  Assumes <proc>(comm y x)=(comm x y).
Axiom SC4.  Assumes <proc>(comm x (comm y z))=(comm (comm x y) z).
Axiom SC5.  Assumes <proc>(Lmer (comm x y) z)=(comm x (Lmer y z)).
End STANDARD_COMMUNICATION.
```

6.7 The module HIDE

The module HIDE describes the behaviour of the hiding operator τ. Let nl be a list of names, $n_i \in \mathcal{N}$ and let a range over δ, τ and the actions of E. The implementation is similar to the implementation of the D-axioms

TI1:	$\tau(\{n_1,\ldots,n_m\},a)=a$	**from** $E,\varnothing,\varnothing$, $label\,(a)\notin\{n_1,\ldots,n_m\}$
TI2:	$\tau(\{n_1,\ldots,n_m\},a)=\tau$	**from** $E,\varnothing,\varnothing$, $label\,(a)\in\{n_1,\ldots,n_m\}$
TI3:	$\tau(nl,x+y)=\tau(nl,x)+\tau(nl,y)$	**from** $E,\varnothing,\{x,y\}$
TI4:	$\tau(nl,x\cdot y)=\tau(nl,x)\cdot\tau(nl,y)$	**from** $E,\varnothing,\{x,y\}$

Table 14: The module HIDE

of ACP.

```
Section HIDE.
Variable D     :Set.
Variable d     :D.
Variable x,y   :proc.
Variable a     :act.
Variable L     :ehlist.
Axiom TI1.  Assumes (~(In_ehlist a L))-><proc>(ia D a d)= (hide L (ia D a d)).
Axiom TI2.  Assumes (In_ehlist a L)-><proc>Delta=(hide L (ia D a d)).
Axiom TI3.  Assumes <proc>(alt (hide L x) (hide L y))=(hide L (alt x y)).
Axiom TI4.  Assumes <proc>(seq (hide L x) (hide L y))=(hide L (seq x y)).
End HIDE.
```

6.8 The module REN

The module REN describes the behaviour of the renaming operator ρ. Let nl be a list of names, $n_i\in\mathcal{N}$, $m\geq 1$, $m'\geq 0$ and let a range over δ, τ and the actions of E. The COQ implementation:

RN1:	$\rho(\{n_1\to n'_1,\ldots,n_m\to n'_m\},a)=a$	**from** $E,\varnothing,\varnothing$, $label\,(a)\notin\{n_1,\ldots,n_m\}$
RN2:	$\rho(\{n_1\to n'_1,\ldots,n_m\to n'_m\},n_i(t_1,\ldots,t_{m'}))=n'_i(t_1,\ldots,t_{m'})$	**from** $E,\varnothing,\varnothing$, $1\leq i\leq m$
RN3:	$\rho(nl,x+y)=\rho(nl,x)+\rho(nl,y)$	**from** $E,\varnothing,\{x,y\}$
RN4:	$\rho(nl,x\cdot y)=\rho(nl,x)\cdot\rho(nl,y)$	**from** $E,\varnothing,\{x,y\}$

Table 15: The module RENAME

```
Section RENAME.
Variable D       :Set.
Variable d       :D.
Variable x,y     :proc.
Variable a,b     :act.
Variable L       :rlist.
Axiom RN1.  Assumes ((b:act)(~(In_rlist a b L)))-><proc>(ia D a d)= (ren L (ia D a d)).
Axiom RN2.  Assumes (In_rlist a b L)-><proc>(ia D b d)=(ren L (ia D a d)).
Axiom RN3.  Assumes <proc>(alt (hide L x) (hide L y))=(hide L (alt x y)).
Axiom RN4.  Assumes <proc>(seq (hide L x) (hide L y))=(hide L (seq x y)).
End RENAME.
```

6.9 The module SUM

The axiom SUM2 states that the name of a bound variable is irrelevant. This rule may be omitted because COQ accepts `H:[x:A]`ϕ`(x)` as an inhabitant of `[y:A]`ϕ`(y)`. The σ's are added for a technical reason. We ignore

SUM1:	$\Sigma(d:D,x)=x$	**from** $E,\{\langle d:D\rangle\},\{x\}$
SUM2:	$\Sigma(d:D,\sigma(x))=\Sigma(e:D,\sigma(x)[e/d])$	**from** $E,\{\langle d:D\rangle\},\{x\}$
SUM3:	$\Sigma(d:D,\sigma(x))=\Sigma(d:D,\sigma(x))+\sigma(x)$	**from** $E,\{\langle d:D\rangle\},\{x\}$
SUM4:	$\Sigma(d:D,\sigma(x)+\sigma(y))=\Sigma(d:D,\sigma(x))+\Sigma(d:D,\sigma(y))$	**from** $E,\{\langle d:D\rangle\},\{x,y\}$
SUM5:	$\Sigma(d:D,\sigma(x)\cdot y)=\Sigma(d:D,\sigma(x))\cdot y$	**from** $E,\{\langle d:D\rangle\},\{x,y\}$
SUM6:	$\Sigma(d:D,\sigma(x)\between y)=\Sigma(d:D,\sigma(x))\between y$	**from** $E,\{\langle d:D\rangle\},\{x,y\}$
SUM7:	$\Sigma(d:D,\sigma(x)\mid y)=\Sigma(d:D,\sigma(x))\mid y$	**from** $E,\{\langle d:D\rangle\},\{x,y\}$
SUM8:	$\Sigma(d:D,\partial(nl,\sigma(x)))=\partial(nl,\Sigma(d:D,\sigma(x)))$	**from** $E,\{\langle d:D\rangle\},\{x\}$
SUM9:	$\Sigma(d:D,\tau(nl,\sigma(x)))=\tau(nl,\Sigma(d:D,\sigma(x)))$	**from** $E,\{\langle d:D\rangle\},\{x\}$
SUM10:	$\Sigma(d:D,\rho(nl,\sigma(x)))=\rho(nl,\Sigma(d:D,\sigma(x)))$	**from** $E,\{\langle d:D\rangle\},\{x\}$
SUM11:	$\Sigma(d:D,\sigma(x))=\Sigma(d:D,\sigma(y))$ provided $\sigma(x)=\sigma(y)$ **from** $E,\varnothing,\{x,y\}$	**from** $E,\{\langle d:D\rangle\},\{x,y\}$

Table 16: The module SUM, e is not free in $\sigma(x)$

them here. The implementation of SUM is realized as follows.

```
Section SUM.
Variable D    :Set.
Variable d,e  :D.
Variable x,y  :D->proc.
Variable p    :proc.
Variable L    :ehlist.
Variable R    :rlist.
Axiom SUM1.   Assumes <proc>p=(sum D ([d:D]p)).
Axiom SUM3.   Assumes <proc>(alt (sum D x) (x d))=(sum D x).
Axiom SUM4.   Assumes <proc>(alt (sum D x) (sum D y))= (sum D ([d:D](alt (x d) (y d)))).
Axiom SUM5.   Assumes <proc>(sum D ([d:D](seq (x d) p)))= (seq (sum D x ) p).
Axiom SUM6.   Assumes <proc>(sum D ([d:D](Lmer (x d) p)))= (Lmer (sum D x) p).
Axiom SUM7.   Assumes <proc>(sum D ([d:D](comm (x d) p)))= (comm (sum D x) p).
Axiom SUM8.   Assumes <proc> (sum D ([d:D](hide L (x d))))=(hide L (sum D x)).
Axiom SUM9.   Assumes <proc>(sum D ([d:D](enc L (x d))))=(enc L (sum D x)).
Axiom SUM10.  Assumes <proc> (sum D ([d:D](ren R (x d))))= (ren R (sum D x)).
End SUM.
```

The implementation of axiom SUM11 would be

```
Axiom SUM11.  Assumes ((d:D)<proc>(x d)=(y d))-><proc>(sum D x)=(sum D y).
```

The type of `SUM11` is inhabited by

```
[H:(d:D)<proc>(x d)=(y d)]eq_ind D->proc x [p:D->proc](<proc>+{D,x}=+{D,p})
(refl_equal proc +{D,x}) y (KSI D x y H)
```

and hence Axiom `SUM11` can be omitted.

7 The recursive specification principle

In order to derive identities between infinite processes, μCRL consists of a rule *RSP* (Recursive Specification Principle). This principle (axiom) states something like "every process equation has a unique solution". A process equation in E is an expression $\mathcal{E}$ of the form

$$n(t_1,\ldots,t_k)=\phi(n(f_1(t_1),\ldots,f_k(n_k)))$$

where $t_i, f_i(t_i) : s_i$ for some sort s_i declared in E, $n \in \mathcal{N}$ and ϕ is a process term possibly containing $t_1, \ldots, t_k$ and n. The arity k satisfies $k \geq 0$. Note the syntactical difference from

$$n(t_1 : s_1, \ldots, t_k : s_k) = \phi(n(f_1(t_1), \ldots, f_k(n_k))) \tag{8}$$

which would be the defining equation of a process n. We say that a process p satisfies equation $\mathcal{E}$ when p is a process term with free variables $x_1 : s_1, \ldots, x_k : s_k$ such that $\mathcal{E}[\lambda x_1, \ldots, x_n.p/n]$ holds, i.e.

$$(\lambda x_1, \ldots, x_k.p)(t_1, \ldots, t_k) = \phi((\lambda x_1, \ldots, x_k.p)(f_1(t_1), \ldots, f_k(n_k)))$$

holds. The expressions of the form $(\lambda \vec{x}.p)(\vec{\alpha})$ reduce to $p[\vec{\alpha}/\vec{x}]$. Of course process n satisfies $\mathcal{E}$ when it would be defined as in 8.

EXAMPLE 7.0.1 Consider the following equation in the specification $ALTERNATE, \langle x : nat \rangle, \varnothing$. Let $\mathcal{E}$ be the following equation.

$$n(x) = a(even(x)) \cdot n(S(x)) \tag{9}$$

Note that $p(y)$ and $q(even(y))$ are both process terms containing a free variable $y : nat$. Substitution of this processes in $\mathcal{E}$ gives us

$$\begin{aligned}
(i): \quad (\lambda y.p(y))(x) &= a(even(x)) \cdot (\lambda y.p(y))(S(x)) \twoheadrightarrow \\
p(x) &= a(even(x)) \cdot p(S(x)) \qquad (10) \\
(ii): \quad (\lambda y.q(even(y)))(x) &= a(even(x)) \cdot (\lambda y.q(even(y)))(S(x)) \twoheadrightarrow \\
q(even(x)) &= a(even(x)) \cdot q(even(S(x))) \qquad (11)
\end{aligned}$$

The validity of Formula 10 follows directly from P of Table 3. Equation 11 follows from E2 of Table 3:

$$\dfrac{\dfrac{Q \;\equiv\; q(b) = a(b) \cdot q(\neg(b))}{q(even(x)) = a(even(x)) \cdot q(\neg(even(x)))}\, b := even(x)}{q(even(x)) = a(even(x)) \cdot q(even(S(x))) \;\equiv\; (11)}\, \text{E2}$$

Now $p(x) = q(even(x))$ is derivable with RSP for they both satisfy $\mathcal{E}$.

Obviously we have to have some restrictions on the process equation. For instance B12 and B123 from Table 4 both satisfy the process equation $n = n$, buth we don't want to derive equality between those two processes. For this purpose the notion of *guardedness* is developped. Guardedness is a property on process equations. The idea is that those equations that have solutions which shouldn't be equal (e.g. $n = n$) are unguarded. Then, RSP can be formulated as "every guarded process equation has a unique solution". In formula:

$$\frac{G[\lambda \vec{x}.p(\vec{x})/n] \quad \textbf{from } E, V_d, V_p \qquad G[\lambda \vec{x}.q(\vec{x})/n] \quad \textbf{from } E, V_d, V_p}{p(\vec{x}) = q(\vec{x}) \quad \textbf{from } E, V_d, V_p}\, RSP$$

where $p(\vec{x})$ and $q(\vec{x})$ are process terms over E, V_d, V_p and G is a guarded process equation.

7.1 Guardedness

There are several definitions of guardedness. You can define guardedness as a property of sets of equations. See for instance [11] for a definition. We restrict ourselves to single equations. When you view single equations as sets of equations then our definition coincides with the more general definition in [11].

DEFINITION 7.1.1 A process equation $n = \phi(n)$ is *guarded* iff

- $\phi(n) \equiv \phi_1(n) \circ \phi_2(n)$ with $\circ \in \{+, \parallel, \mid, \lhd\, b \,\rhd\}$ and $n = \phi_i(n)$ is guarded for $i = 1, 2$,
- $\phi(n) \equiv \phi_1(n) \circ \phi_2(n)$ with $\circ \in \{\cdot, \between\}$ and $n = \phi_1(n)$ is guarded,
- $\phi(n) \equiv \Sigma(d : D, \phi_1(n))$ and $n = \phi_1(n)$ is guarded,

- $\phi(n) \equiv C(nl, \phi_1(n))$ with $C \in \{\partial, \tau, \rho\}$ and nl being a list of names (or in the case of ρ a renaming scheme), and $n = \phi_1(n)$ is guarded,
- $n = \delta$ is guarded,
- $n = \tau$ is guarded,
- $n = m(t_1, \ldots, t_k)$, with $k \geq 0$, is guarded for all $m \in \mathcal{N} \setminus \{n\}$.

This single equation version of guardedness is represented by `Section GUARDED`.

```
Section GUARDED.
Variable D    :Set.
Variable d    :D.
Variable x,y  :proc.
Variable z    :D->proc.
Variable a    :act.
Variable B    :bool.
Variable L    :ehlist.
Variable R    :rlist.
Axiom G1.   Assumes (grd (ia D a d)).
Axiom G2.   Assumes ((d:D)(grd (z d)))->(grd (sum D z)).
Axiom G3.   Assumes (grd x)->(grd (seq x y)).
Axiom G4.   Assumes (grd x)->(grd (Lmer x y)).
Axiom G5.   Assumes (grd x)->(grd y)->(grd (alt x y)).
Axiom G6.   Assumes (grd x)->(grd y)->(grd (mer x y)).
Axiom G7.   Assumes (grd x)->(grd y)->(grd (comm x y)).
Axiom G8.   Assumes (grd x)->(grd y)->(grd (cond x B y)).
Axiom G9.   Assumes (grd x)->(grd (hide L x)).
Axiom G10.  Assumes (grd x)->(grd (enc L x)).
Axiom G11.  Assumes (grd x)->(grd (ren R x)).
End GUARDED.
```

7.2 The implementation of RSP

One of the reasons to declare all processes as inhabitants of `A->proc` for some `A:Set` is the possibility to declare a parameter `rsp` that can be applied on an arbitrary pair of processes with the same parameterization.

```
Parameter rsp :  (A:Set)(x,y:A->proc)(G:(A->proc)->A->proc)
                 ((p:A->proc)(a:A)(grd (G p a)))->
                 ((a:A)<proc>(x a)=(G x a))->
                 ((a:A)<proc>(y a)=(G y a))->
                 (a:A)<proc>(x a)=(y a).
```

The inhabitant `(rsp A x y G H Gx Gy a)` of `<proc>(x a)=(y a)` is build from the following proofs:

`H`	proving that process equation `G` is guarded,
`Gx`	proving that `x` satisfies process equation `G`,
`Gy`	proving that `y` satisfies process equation `G`.

EXAMPLE 7.2.1 We assume that we have declared the μCRL specification *ALTERNATE* given in Table 3. (See appendix for the implementation of this specification.) The guarded equation $\mathcal{E}$ given in Formula 9 is implemented as

```
G ≡ [n:nat->proc][x:nat]((ia bool a (even x)) * (n (S x)))
```

of type `(nat->proc)->nat->proc`. The term `Gx` in this case should be an inhabitant of

`<proc>(p x)=(G p x)` stating that `p` satisfies `G`. The proof of the guardedness of `G` is

```
 [n:nat->proc][x:nat](G3 (ia bool a (even x)) (n (S x)) (G1 bool a (even x)))
```

of type `(n:nat->proc)(x:nat)(grd ((ia bool a (even x)) * (n (S x))))`.

We end this subsection with a complete proof session for the formula $p(x) = q(even(x))$.

```
Coq < Goal (x:nat)<proc>(p x)=(q (even x)).

Coq < Intro.

1 subgoal
 <proc>(p x)=(q (even x))
 ============================
 x  :  nat

Coq < Apply (RSP nat p [z:nat](q (even z)) [n:nat->proc][m:nat]((ia bool a (even m)) * (n (s m)))).


3 subgoals
 (p:nat->proc)(d:nat)(grd ((ia bool a (even d)) * (p (s d))))
 ============================
 x  :  nat

subgoal 2 is:
 (d:nat)(<proc>(p d)=((ia bool a (even d)) * (p (s d))))

subgoal 3 is:
 (d:nat)(<proc>(q (even d))=((ia bool a (even d)) * (q (even (s d)))))

Coq < Auto.

Use :  Intro ; Intro ; Apply G5 ; Apply G1

2 subgoals
 (d:nat)(<proc>(p d)=((ia bool a (even d)) * (p (s d))))
 ============================
 x  :  nat

subgoal 2 is:
 (d:nat)(<proc>(q (even d))=((ia bool a (even d)) * (q (even (s d)))))

Coq < Auto.

Use :  Intro ; Apply PROC1

1 subgoal
 (d:nat)(<proc>(q (even d))=((ia bool a (even d)) * (q (even (s d)))))
 ============================
 x  :  nat

Coq < Intro.

1 subgoal
 <proc>(q (even d))=((ia bool a (even d)) * (q (even (s d))))
 ============================
 d  :  nat
 x  :  nat

Coq < Elim E2.

1 subgoal
```

```
 <proc>(q (even d))=((ia bool a (even d)) * (q (neg (even d))))
 ============================
 d  :  nat
 x  :  nat

Coq < Auto.

Use :  Apply PROC2

Goal proved!

Coq < Save pq_equal.

pq_equal is defined

Coq < Print pq_equal.

 [x:nat](RSP nat p [z:nat](q (even z))
   [n:nat->proc][m:nat]((ia bool a (even m)) * (n (s m)))
   [p:nat->proc][d:nat](G5 (ia bool a (even d)) (p (s d)) (G1 (a (even d))))
   [d:nat](PROC1 d)
   [d:nat](eq_ind bool (neg (even d))
     [b:bool](<proc>(q (even d))=((ia bool a (even d)) * (q b)))
     (PROC2 (even d)) (even (s d)) (E2 d))
   x)
```

Appendix

We give the implementation of the three specifications given in Tables 3, 4 and 5. The declarations of *ALTERNATE* are labeled with ALT. The declarations of *BUFFER* are labeled with BUF and the declarations of *BAG* are labeled with BAG. The declarations that are not dependent of the specifications are labeled with GEN (general). When you ignore for instance all the rules labeled with ALT or BUF, then you have the complete set of declarations for specification BAG, etc.

```
ALT  Inductive Set act  = a:act | DELTA:act | TAU:act.

BUF  Inductive Set act  = r1:act | r2:act | c2:act | s2:act | s3:act | DELTA:act | TAU:act.


BAG  Inductive Set act  = r:act | s:act | DELTA:act | TAU:act.

GEN  Inductive Definition one     :Set  = i:one.
GEN  Inductive Definition ehlist  :Set  = ehnil:ehlist | ehcons:act->ehlist-ehlist.
GEN  Inductive Definition rlist   :Set  = rnil:rlist | rcons:act->act->rlist->rlist.
GEN
GEN  Definition In_ehlist  =  [L:ehlist][a:act](<Prop>Match L with False
GEN                           [b:act][P:Prop](<act>a=b)\/P.
GEN  Definition In_rlist   =  [L:rlist][a,b:act](<Prop>Match L with False
GEN                           [n,m:act][P:Prop]((<act>a=n)/\(<act>b=m))\/P.
GEN
```

```
GEN  Parameter proc    :Set.
GEN  Parameter ia      :(A:Set)A->act->proc.
GEN  Parameter grd     :proc->proc->Prop.
GEN  Parameter rsp     :(A:Set)(x,y:A->proc)(G:(A->proc)->A->proc)
GEN                     ((p:A->proc)(a:A)(grd (G p a)))->
GEN                     ((a:A)<proc>(x a)=(G x a))->
GEN                     ((a:A)<proc>(y a)=(G y a))->
GEN                     (a:A)<proc>(x a)=(y a).
GEN
GEN  Section GUARDED.   (see Subsection 7.1)
GEN
GEN  Axiom CL.    Assumes (P:Prop)(P\/~P).
GEN  Axiom EXT.   Assumes (A:Set)(x,y:A->proc)((a:A)<proc>(x a)= (y a))-><A->proc>x=y.
GEN
GEN  Syntax alt    "(_ + _)".
GEN  Syntax seq    "(_ * _)".
GEN  Syntax mer    "(_ || _)".
GEN  Syntax Lmer   "(_ |L _)".
GEN  Syntax comm   "(_ | _)".
GEN  Syntax cond   "(_ < _ > _)".
GEN  Syntax sum    "+{_,_}".
GEN
GEN  Parameter ALT    :proc->proc->proc.
GEN  Parameter SEQ    :proc->proc->proc.
GEN  Parameter MER    :proc->proc->proc.
GEN  Parameter LMER   :proc->proc->proc.
GEN  Parameter COMM   :proc->proc->proc.
GEN  Parameter COND   :proc->bool->proc->proc.
GEN  Parameter SUM    :(A:Set)(A->proc)->proc.
GEN  Parameter enc    :ehlist->act->proc.
GEN  Parameter hide   :ehlist->act->proc.
GEN  Parameter ren    :rlist->act->proc.
GEN
GEN  Definition alt    = [p,q:proc](ALT p q).
GEN  Definition seq    = [p,q:proc](SEQ p q).
GEN  Definition mer    = [p,q:proc](MER p q).
GEN  Definition Lmer   = [p,q:proc](LMER p q).
GEN  Definition comm   = [p,q:proc](COMM p q).
GEN  Definition cond   = [p:proc][b:bool][q:proc](COND p b q).
GEN  Definition sum    = [A:Set][p:A->proc](SUM A p).
GEN  Definition delta  = (ia one DELTA i).
GEN  Definition tau    = (ia one TAU i).
GEN
GEN  Section ACP.                      (see Subsection 6.5)
GEN  Section STANDARD_COMMUNICATION.   (see Subsection 6.6)
GEN  Section HIDE.                     (see Subsection 6.7)
GEN  Section RENAME.                   (see Subsection 6.8)
GEN  Section SUM.                      (see Subsection 6.9)
```

The next step is the implementation of the sorts. A sort is represented by an inductive type when there exists a set of constructors for that sort. The sorts *nat* and **Bool** are represented by the inductive types `nat` and `bool` of prelude.v.

```
BUF  Parameter D  :Set.

BAG  Parameter D  :Set.
BAG
BAG  Inductive Definition bag  :Set = o:bag | cons:D->bag->bag.
```

Now we can declare those functions and constants that are not a constructor of any sort of the specification. We declare a polymorphic `If` to represent the two *BAG*-functions *If_bool* and *If_bag*.

```
ALT  Parameter neg   :bool->bool.
ALT  Parameter even  :nat->bool.

BAG  Parameter If    :(A:Set)bool->A->A->A.
BAG  Parameter Eq    :D->D->bool.
BAG  Parameter test  :D->bag->bool.
BAG  Parameter rem   :D->bag->bag.
BAG  Parameter con   :bag->bag->bag.
```

We used sections to implement the rewrite rules. Each property is quantified over all the variables (of the section in which the property is declared) that occur in that property. The variable declarations are local, they are dropped at the end of the section.

```
ALT  Section REW.
ALT  Variable x  :nat.
ALT  Axiom TF.   Assumes <bool>false=(neg true).
ALT  Axiom FT.   Assumes <bool>true=(neg false).
ALT  Axiom e0. Assumes <bool>true=(even 0).
ALT  Axiom eS. Assumes <bool>(neg (even x))=(even (S x)).
ALT  End REW.
ALT

BAG  Section REW.
BAG  Variable b1,b2  :bool.
BAG  Variable d,e    :D.
BAG  Variable b,c    :bag.
BAG  Axiom IF1.   Assumes <bool>b1=(If bool true b1 b2).
BAG  Axiom IF2.   Assumes <bool>b2=(If bool false b1 b2).
BAG  Axiom IF3.   Assumes <bag>b=(If bag true b c).
BAG  Axiom IF4.   Assumes <bag>c=(If bag false b c).
BAG  Axiom EQ.    Assumes <bool>true=(Eq d d).
BAG  Axiom Ei.    Assumes (~<D>d=e)->(<bool>false=(Eq d e)).
BAG  Axiom BAG1.  Assumes <bool>false=(test d o).
BAG  Axiom BAG2.  Assumes <bool>(If bool (Eq d e) true (test d b))= (test d (In e b)).
BAG  Axiom BAG3.  Assumes <bag>(In e (In d b))=(In d (In e b)).
BAG  Axiom BAG4.  Assumes <bag>o=(rem d o).
BAG  Axiom BAG5.  Assumes <bag>(If bag (Eq d e) b (In e (rem d b)))= (rem d (In e b)).
BAG  Axiom BAG6.  Assumes <bag>b=(con o b).
BAG  Axiom BAG7.  Assumes <bag>(In d (con b c))=(con (In d b) c).
BAG  End REW.
```

The processes are also declared in sections.

```
ALT  Parameter p  :nat->proc.
ALT  Parameter q  :bool->proc.
ALT
```

```
ALT  Section PROC.
ALT  Variable y  :nat.
ALT  Variable b  :bool.
ALT  Axiom P.  Assumes <proc>(seq (ia bool a (even y)) (p (S y)))=(p y).
ALT  Axiom Q.  Assumes <proc>(seq (ia bool a b) (q (neg b)))= (q b).
ALT  End PROC.

BUF  Parameter B12,B23,B123  :proc.
BUF
BUF  Section PROC.
BUF  Axiom P12.   Assumes <proc>(seq (sum D [d:D](seq (ia D r1 d) (ia D s2 d))) B12)=B12.
BUF  Axiom P23.   Assumes <proc>(seq (sum D [d:D](seq (ia D r2 d) (ia D s3 d))) B23)=B23.
BUF  Axiom P123.  Assumes <proc>(enc (ehcons c2 ehnil) (hide (ehcons r2 (ehcons s2 ehnil))
BUF               (mer B12 B23)))=B123.
BUF  End PROC.

BAG  Parameter Bag  :D->proc.
BAG
BAG  Section PROC.
BAG  Variable x  :D.
BAG  Axiom BAG.  Assumes <proc>(alt (sum D [d:D](seq (ia D r d) (Bag (In d x))))
BAG              (sum D [d:D](cond (seq (ia D s d) (Bag (rem d x))) (test d x) delta)))=(Bag x).
```

When we follow the general concept of Subsection 4.6 then we obtain the following declarations for the communication function.

```
ALT  Definition gamma =  [x,y:act]DELTA.

BUF  Definition gamma =  [x,y:act](<act>Match x with
BUF                      DELTA
BUF                      (<act>Match y with DELTA DELTA DELTA s2 DELTA DELTA DELTA)
BUF                      DELTA
BUF                      (<act>Match y with DELTA r2 DELTA DELTA DELTA DELTA DELTA)
BUF                      DELTA
BUF                      DELTA
BUF                      DELTA

BAG  Definition gamma =  [x,y:act]DELTA.
```

References

[1] H.P. Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 117–309. Oxford Science Publications, 1992.

[2] J.A. Bergstra and J.W. Klop. The algebra of recursively defined processes and the algebra of regular processes. In *Proceedings* 11^{th} *ICALP,* Antwerp, volume 172 of *Lecture Notes in Computer Science*, pages 82–95. Springer-Verlag, 1984.

[3] M.A. Bezem and J.F. Groote. A formal verification of the alternating bit protocol in the calculus of constructions. Technical Report 88, Logic Group Preprint Series, Utrecht University, March 1993.

[4] R. Cleaveland and P. Panangaden. Type theory and concurrency. *International Journal of Parallel Programming*, 17:153–206, 1988.

[5] R.L. Constable, S.F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, D.J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, J.T. Sasaki, and S.F. Smith. *Implementing Mathematics with the NuPrl Development System.* Prentice-Hall, inc., Englewood Cliffs, New Jersey, first edition, 1986.

[6] T. Coquand and G. Huet. The calculus of constructions. *Information and Control*, 76:95–120, 1988.

[7] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Paulin-Mohring, and B. Werner. The Coq proof assistant version 5.6 user's guide. Technical report, INRIA – Rocquencourt, 1991.

[8] S.J. Garland and J.V. Guttag. An overview of LP, the Larch prover. In N. Dershowitz, editor, *Proceedings of the 3rd International Conference on Rewriting Techniques and Applications*, volume 355 of *Lecture Notes in Computer Science*, pages 137–155. Springer-Verlag, 1989.

[9] J.F. Groote. Towards a formal mathematical vernacular. Technical Report 84, Logic Group Preprint Series, Utrecht University, December 1992.

[10] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. Technical Report CS-R9076, CWI, Amsterdam, December 1990.

[11] J.F. Groote and A. Ponse. Proof theory for μCRL. Technical Report CS-R9138, CWI, Amsterdam, August 1991.

[12] H. Korver and J. Springintveld. A computer-checked verification of Milner's scheduler. Technical Report 101, Logic Group Preprint Series, Utrecht University, November 1993.

[13] R. Milner. *A Calculus of Communicating Systems*, volume 92. Springer-Verlag, Berlin, 1980.

[14] C. Paulin-Mohring. Inductive definitions in the system Coq. Rules and properties. In M. Bezem and J.F. Groote, editors, *Proceedings of the* 1^{st} *International Conference on Typed Lambda Calculi and Applications, TLCA '93,* Utrecht, The Netherlands, volume 664 of *Lecture Notes in Computer Science*, pages 328–345. Springer-Verlag, 1993.