A verification of the Bakery protocol combining algebraic and model-oriented techniques

C. Brovedani and A.S. Klusener

Computer Science/Department of Software Technology

CS-R9619 1996

Report CS-R9619
ISSN 0169-118X

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

A Verification of the Bakery Protocol Combining Algebraic and Model-oriented Techniques

Claudia Brovedani

Poker S.R.L. Corso Garibaldi 167, 10078 Venaria Reale (TO), Italy

Steven Klusener

CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

e-mail: stevenk@cwi.nl

Abstract

In this paper we give a specification of the so called Bakery protocol in an extension of the process algebra ACP with abstract datatypes. We prove that this protocol is equal to a Queue, modulo branching bisimulation equivalence.

The verification is as follows. First we give a linear specification of the Bakery, that is a specification without parallelism. Then we introduce an invariant and encorporate this invariant into the linear specification of the Bakery and the specification of the Queue. Finally, we give a boolean function on the arguments of the resulting specification of the Bakery and the Queue, and we prove that by its equations it defines a branching bisimulation.

This paper can be considered as an alternative to the proof of Groote and Korver [GK94], that proves the correctness of the Bakery protocol modulo weak bisimulation (or observational congruence) completely within the proof system of μCRL.

AMS Subject Classification (1991): 68Q10, 68Q40, 68Q45, 68Q55.
CR Subject Classification (1991): D.1.3, D.3.1, D.4.1, F.1.2, F.3.2.
Keywords & Phrases: process algebra, ACP, protocol verification, branching bisimulation.
Note: While writing this paper the first author stayed at CWI, funded by scholarships of the University of Torino and CNR.

Introduction

Process algebras, such as ACP [BK84, BW90] and CCS [Mil89], provide a simple but expressive framework for specifying and analyzing distributed systems, see for example [Bae90]. However, in their pure form these process algebras are not very well suited for dealing with systems in which the process behavior depends on the underlying data. Therefore, Groote and Ponse have developed the language μCRL[GP94], a combination of ACP with abstract datatypes [EM85]. For this language they also gave a proof system [GP91].

In order to exercise with different proof methods and techniques, that concern processes with data, various verifications of the so called Bakery protocol have been given. This protocol requires that each customer takes a number when entering the bakery shop, and that the baker serves the customers in order of their number. This protocol is correct as the customers are served in order of their entrance, in other words, the external behavior of the protocol corresponds with a queue. Although this seems rather obvious this paper, and others, show that a detailed analysis of the correctness depends subtly on a mixture of process behavior and data properties.

In [GK94] Groote and Korver have proven the bakery protocol purely algebraically, using the axioms of μCRL, the proofrule RSP and the τ-laws for weak bisimulation (which is a more identifying equivalence than branching bisimulation). Lately Griffioen and Korver have proven the bakery protocol in the I/O-automata model [GK95], this proof has been checked formally within the proof system LP.

When Groote and Korver started their work on the verification of the bakery protocol in μCRL, it was not completely clear how it could be done. For example, it was not yet known how one could deal with invariants, as it has been proposed later by Bezem and Groote in [BG93]. Therefore, Jan Bergstra suggested to try to verify the bakery protocol in a more model oriented way. In this paper this idea is worked out in detail.

Our verification consists of the following steps:

- First we define the bakery protocol formally; it is a parallel composition of an in–counter, a shop–floor (modelled as a bag), and the serve desk, or out–counter. Moreover, we define a queue process.

  The goal of the paper is to prove that these two processes are equal, modulo branching bisimulation.

- Then, we give a linear specification of the bakery protocol, i.e., a process definition without parallelism. This definition is based on the four states in which the bakery can evolve, similar definitions of these four states can be found in [GK94] as well.

  This definition contains a rather complex sum, based on the fact that if the baker asks for the next customer, he asks for *some* customer that holds a label that corresponds with his counter.

- We introduce a so called invariant that formalizes the intuition that says that there will be exactly a unique customer with that label. We encorporate this invariant into the linear specification of the bakery, by which the above mentioned sum can be simplified.

- In order to relate the resulting definition with the one of the queue, we encorporate the invariant into the definition of the queue as well.

- Finally we define a relation as a boolean funtion and we show that it can be considered as a branching bisimulation.

  So, we do not use the axioms for branching bisimulation, nor the proof rule RSP.

The language we use consists of ACP constructs and a prefix summation, it is derived from real time ACP with prefix summation [FK95], and it can be considered as a subcalculus of μCRL.

Acknowledgements

The first author wants to thank the CWI, and Jan Willem Klop in particular, for the offered hospitality. Furthermore, both authors want to thank Jan Willem Klop, Henri Korver, and Alban Ponse for the technical discussions on earlier versions of this paper.

Jan Bergstra is acknowledged for providing the original research proposal.

1 Some remarks on the syntax of ACP with prefix Summation

1.1 The Datatype part

The datatypes that we deal with are defined in detail in the Appendix, Section B. Their definitions are given in a μCRL style, and their standard parts are taken from [GvW94], including the modulo arithmatic of the natural numbers.

In this paper we will refer to some abstract datatype D, with typical variable d, that models the set of customers. For technical reasons we assume that D contains at least the bottom element $\perp$, i.e., a customer that is not allowed in the bakery. A pair d^i models a customer d that holds a label i, and it is called a frame. The datatype of frames is denoted by Frame, with typical variable f, and the datatype of bags of frames is denoted by FBag, with typical variable b and emptybag $\emptyset_{fbag}$.

Nat denotes the standard datatype of natural numbers, $+_n$ denotes addition modulo n.

We have also the standard datatype of the booleans, denoted by Bool, with constants $\mathbf{t}$ (true) and $\mathbf{f}$ (false), and operators $\wedge$ and $\vee$. Over this datatype we have expressions like $i < n$ and $size(b) \leq n$. An arbitrary expression of type Bool is denoted by ϕ. The boolean expression $test(f, b)$ is true if the frame f is in the bag b.

Each datatype S is provided with an equality function $\mathsf{S} \times \mathsf{S} \rightarrow \mathsf{Bool}$, for which holds that $eq(s, s') = \mathbf{t} \leftrightarrow s = s'$. Hence, we may allow ourselves to write the boolean expression $eq(s, s')$ by $s = s'$.

1.2 The Process part

For simplicity we restrict ourselves to *prefix summation*, so if x is a process term (in which the data varable v may occur), then $\sum_{v:\phi} a(v) \cdot x$ is a process term as well. The set of process terms is defined by the following BNF sentence:

$$x \;::=\; \delta \;\mid\; \sum_{\phi} a(v) \cdot x \;\mid\; x + x \;\mid\; \phi :\rightarrow x \;\mid\; x \parallel x \;\mid\; x \mathbin{\|\!\_} x \;\mid\; x | x \;\mid\; \partial_H(x) \;\mid\; \tau_H(x),$$

were v is a data variable of some type, ϕ is an expression of type Bool, and H is a set of unparameterized actions.

The sum construct $\sum_{\phi} a(v) \cdot x$ binds all occurrences of the data variable v in x. For some ground [1] data expression e of the same type, it can execute an action $a(e)$ and evolve into $x[e/v]$, (e substituted for v), for every context in which $\phi[e/v]$ is true. Hence, $\sum_{\phi} a(v) \cdot x$ can be considered as an abbreviation of the μCRL expression $\sum_{v:\mathsf{S}} a(v) \cdot x \triangleleft \phi \triangleright \delta$, were S is the type of v. In some cases, for example if ϕ is a rather large expression, we write $\sum_{v:\phi} a(v) \cdot x$ for $\sum_{\phi} a(v) \cdot x$.

The process $\phi :\rightarrow x$ is enabled only in contexts in which ϕ is true. Again, $\phi :\rightarrow x$ can be considered as an abbreviation of the μCRL expression $x \triangleleft \phi \triangleright \delta$.

This calculus is denoted by $\mathrm{ACP}_{\mathrm{pS}}$, the suffix pS stands for *prefix summation*, and as shown before it can be considered as a subcalculus of μCRL. The axioms for $\mathrm{ACP}_{\mathrm{pS}}$are given in the Appendix, Section C.

[1]i.e., not containing any data variables

2 The Bakery Protocol

The Bakery protocol models the counter/serving process in a bakery; each customer is a provided with a label, or *index*, i and customers are being served by the baker in order of their label. The protocol is parameterized with a natural number, n, that defines its capacity; i.e., the maximal numbers of customers the shop floor can contain.

The description of the Bakery protocol, or B_n, involves three processes: the "in–counter", or $IN_n(i : \mathsf{Nat})$, that provides a new customer with a label i, the "out–counter", or $OUT_n(j : \mathsf{Nat})$, that models the baker serving the customer with label j, and finally we have the clients with their label that are waiting on the shop floor, modelled by the process $Pbag_n(b : \mathsf{FBag})$. Initially both counters are zero and the shop floor is empty.

Definition 2.1 $\gamma(s_1, r_1) = c_1,\ \gamma(s_2, r_2) = c_2$

$$\begin{array}{lcl}
B_n & := & \tau_{\{c_1,c_2\}}(\partial_{\{s_1,s_2,r_1,r_2\}}(IN_n(0) \parallel Pbag_n(\emptyset_{fbag}) \parallel OUT_n(0))) \\
 & & \\
IN_n(i : \mathsf{Nat}) & := & \sum_{d:d\neq\perp} enter(d) \cdot s_1(d^i) \cdot IN_n(i +_n 1) \\
OUT_n(j : \mathsf{Nat}) & := & \sum_{f:index(f)=j} r_2(f) \cdot out(data(f)) \cdot OUT_n(j +_n 1) \\
Pbag_n(b : \mathsf{FBag}) & := & size(b) \neq n :\rightarrow \sum_{f:\mathbf{t}} r_1(f) \cdot Pbag_n(add(f, b)) \\
 & & + \sum_{f:test(f,b)} s_2(f) \cdot Pbag_n(rem(f, b))
\end{array}$$

The goal of introducing such a label/countering mechanism into a bakery is of course that customers are served properly in the order of their entrance. In fact, that is exactly what we are going to prove in this paper. To be able to state this properly, we define DQueue, the datatype of queues of D, see for its definition the Appendix, Section B.6. A process that behaves like a queue with capacity m is denoted by Q_m and is defined below.

Definition 2.2

$$\begin{array}{lcl}
Q_m & := & Q_m(\emptyset_{dqueue}) \\
Q_m(q : \mathsf{DQueue}) & := & \quad size(q) \neq m \ :\rightarrow \sum_{d\neq\perp} enter(d) \cdot Q_m(add(d, q)) \\
 & & + \ size(q) \neq 0 \quad :\rightarrow out(top(q)) \cdot Q_m(untop(q))
\end{array}$$

So, we will prove that the Bakery equals a Queue. As the Bakery has two positions, one at the in–counter, and one at the out–counter, and a shop floor with capacity n, we will prove that B_n equals Q_{n+2}. The Bakery involves certain so called *internal* actions, both communication actions c_1 and c_2 (that model the transfer of customers from the in–counter to the shop floor, resp. the shop floor to the out-counter) are considered internal. A notion of equivalence that considers these internal steps is called *(rooted) Branching Bisimulation*, of van Glabbeek and Weijland for which we refer to [GW91] and [BW90]. This equivalence abstracts from internal actions that do not enforce a choice.

Theorem 2.3 (Correctnes of the Bakery Protocol)
B_n is (rooted) branching bisimilar with Q_{n+2}.

3 A linear specification of the Bakery

The first step in our proof is to characterize the Bakery by four different situations. These are depicted in Figure 1. In this figure d^i $IN_n(i +_n 1)$ denotes the situation at the in–counter were a new customer d has been provided a label i, but has not yet entered the shop floor. Similarly, e $OUT_n(j)$ denotes the situation were customer e has just been served by the baker, but he has not yet left the bakery. The starting state of the Bakery, when it is still empty, corresponds with $\alpha_n(0, \emptyset_{fbag}, 0)$. Note that we have not considered yet the communication actions c_1, c_2 to be internal.

Definition 3.1 (Auxiliary processes $\alpha_n, \beta_n, \gamma_n$ and ϕ_n)

$$\begin{array}{lllll}
\alpha_n(i:\mathsf{Nat}, b:\mathsf{FBag}, j:\mathsf{Nat}) := & \partial_H(& IN_n(i) & \| \, Pbag_n(b) \, \| & OUT_n(j)) \\
\beta_n(d:\mathsf{D}, i:\mathsf{Nat}, b:\mathsf{FBag}, j:\mathsf{Nat}) & & & & \\
\qquad := & \partial_H(s_1(d^i)\cdot & IN_n(i +_n 1) & \| \, Pbag_n(b) \, \| & OUT_n(j)) \\
\gamma_n(i:\mathsf{Nat}, b:\mathsf{FBag}, j:\mathsf{Nat}, e:\mathsf{D}) & & & & \\
\qquad := & \partial_H(& IN_n(i) & \| \, Pbag_n(b) \, \| \, out(e)\cdot & OUT_n(j)) \\
\phi_n(d:\mathsf{D}, i:\mathsf{Nat}, b:\mathsf{FBag}, j:\mathsf{Nat}, e:\mathsf{D}) & & & & \\
\qquad := & \partial_H(s_1(d^i)\cdot & IN_n(i +_n 1) & \| \, Pbag_n(b) \, \| \, out(e)\cdot & OUT_n(j))
\end{array}$$

In Figure 1 it is quite easy to see that the situation α can evolve into a situation β (by the entrance of a new customer), or evolve into a situation γ (when a customer leaves the shop floor and is being served by the baker). The details of these transitions are described by the next Lemma, that gives a linear specification (i.e., without parallel composition).

Lemma 3.2 (A linear specification for the Bakery)

$$\begin{array}{lll}
B'_n & = & \alpha_n(0, \emptyset_{fbag}, 0) \\
 & & \\
\alpha_n(i,b,j) & = & \sum_{d:d\neq\perp} enter(d)\cdot\beta_n(d,i,b,j) \\
 & + & \sum_{f:test(f,b)\wedge index(f)=j} s_2(f)\cdot\gamma_n(i, rem(f,b), j +_n 1, data(f)) \\
 & & \\
\beta_n(d,i,b,j) & = & size(b)\neq n :\rightarrow s_1(d^i)\cdot\alpha_n(i +_n 1, add(d^i,b), j) \\
 & + & \sum_{f:test(f,b)\wedge index(f)=j} s_2(f)\cdot\phi_n(d,i,rem(f,b), j +_n 1, data(f)) \\
 & & \\
\gamma_n(i,b,j,e) & = & \sum_{d:d\neq\perp} enter(d)\cdot\phi_n(d,i,b,j,e) \\
 & + & out(e)\cdot\alpha_n(i,b,j) \\
 & & \\
\phi_n(d,i,b,j,e) & = & size(b)\neq n :\rightarrow c_1(d^i)\cdot\gamma_n(i +_n 1, add(d^i,b), j, e) \\
 & + & out(e)\cdot\beta_n(d,i,b,j)
\end{array}$$

Proof. As the first case is trivial, and the other four cases are similar we only show the case for $\alpha_n(i,b,j)$ in detail.

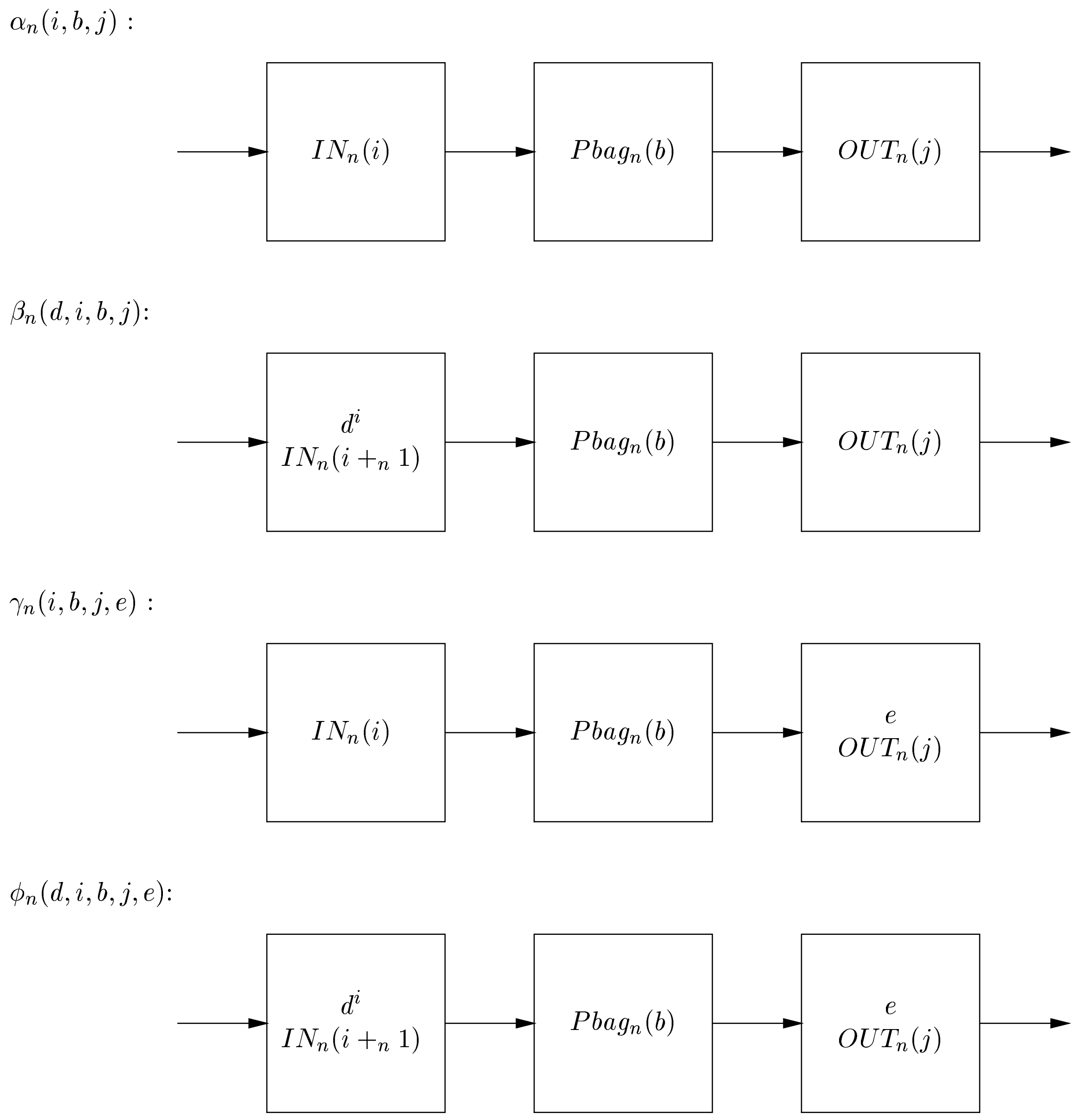

Figure 1: The four different situations of the Bakery

$$
\begin{array}{lll}
\multicolumn{3}{l}{\alpha_n(i,b,j)} \\
= & \partial_H(& IN_n(i) \parallel Pbag_n(b) \parallel OUT_n(j)) \\
= & \partial_H(& (\sum_{d:d\neq\perp} \underline{enter(d)} \cdot s_1(d^i) \cdot IN_n((i +_n 1))) \\
 & \quad\parallel & (size(b) \neq n :\rightarrow \sum_{f:\mathbf{t}} r_1(f) \cdot Pbag_n(add(f,b)) \\
 & & + \sum_{f:test(f,b)} \underline{s_2(f)} \cdot Pbag_n(rem(f,b))) \\
 & \quad\parallel & (\sum_{f:index(f)=j} \underline{r_2(f)} \cdot out(data(f)) \cdot OUT_n((j +_n 1))) \\
= & \multicolumn{2}{l}{\sum_{d:d\neq\perp} enter(d) \cdot \partial_H((s_1(d^i) \cdot IN_n(i +_n 1)) \parallel Pbag_n(b) \parallel OUT_n(j))} \\
+ & \multicolumn{2}{l}{\sum_{f:test(f,b)\wedge index(f)=j} c_2(f) \cdot \partial_H(IN_n(i) \parallel Pbag_n(rem(f,b)) \parallel (out(data(f)) \cdot OUT_n((j +_n 1)))} \\
= & \multicolumn{2}{l}{\sum_{d:d\neq\perp} enter(d) \cdot \beta_n(d,i,b,j)} \\
+ & \multicolumn{2}{l}{\sum_{f:test(f,b)\wedge index(f)=j} c_2(f) \cdot \gamma_n(i, rem(f,b), j +_n 1, data(f))}
\end{array}
$$

In the previous derivation we used the following identities:

$$
\begin{array}{lcl}
\sum_{f:test(f,b)} s_2(f) \cdot x \mid \sum_{f:index(f)=j} r_2(f) \cdot y & = & \sum_{f:test(f,b)\wedge index(f)=j} c_2(f) \cdot (x \parallel y) \\
s_1(d^i) \cdot x | (size(b) \neq n :\rightarrow \sum_{f:\mathbf{t}} r_1(f) \cdot y) & = & size(b) \neq n :\rightarrow c_1(d^i) \cdot (x \parallel (y[d^i/f]))
\end{array}
$$

The first identity is a direct consequence of the axiom $\mathrm{CF1}_{\mathrm{pS}}$. The second one can be derived from axiom $\mathrm{CF1}_{\mathrm{pS}}$ together with COND2. Note that $s_1(d^i) \cdot x$ abbreviates $\sum_{f=d^i} s_1(f) \cdot x$. □

4 The introduction of the Invariant

4.1 The definition of the Invariant

If the baker can serve the next customer, he just calls the value of the out–counter. The mechanism works because there will be exactly one customer with that index. This depends of course on the proper working of the in–counter, if this counter would provide multiple customers with the same index the whole procedure would fall down.

We introduce a so called *invariant*, that is a property of the system that will hold in all possible states, that says that for all indices $j, j +_n 1, \ldots, i -_n 1$ (i.e., all labels of current customers) there will be exactly one customer with that index. As we do not want to deal with some quantification, we formulate this property recursively; if the Bakery is in state (i,b,j) then we say that there is exactly one customer with index j (i.e., $cnt(j,b) = 1$) and if that customer will leave the Bakery and the baker will call the next index then the invariant will hold again. This argument is repeated until the Bakery is empty, i.e., $i = j$ and $b = \emptyset_{fbag}$.

Definition 4.1.1 (The Invariant) I : $\mathsf{Nat} \times \mathsf{Nat} \times \mathsf{FBag} \times \mathsf{Nat} \rightarrow \mathsf{Bool}$

$$
\begin{array}{lll}
I_n(i,b,j) & := & i < n \wedge j < n \\
 & & \wedge \;\; (\quad cnt(j,b) = 1 \wedge I_n(i, rem(j,b), j +_n 1)) \\
 & & \quad\;\; \vee \quad (i = j \wedge b = \emptyset_{fbag}))
\end{array}
$$

Intuitively we do know that the baker does not have to ask for *some* customer with index j, but it can ask for the *unique* customer with index j. We denote this unique customer by $frame(j,b)$. The expression

$\mathit{frame}(j,b)$ results the frame $\perp^j$ if there is no frame with index j or if there is more than one frame with index j in the bag, otherwise it results the properly defined unique frame in bag b with label j.

More formally, in the context of the invariant I the condition $\mathit{test}(f,b) \wedge \mathit{index}(f) = j$ can be reduced to $f = \mathit{frame}(j,b)$, whenever b is not empty. Note that if b is empty, then $\mathit{test}(f,b)$ reduces to **f**.

Lemma 4.1.2

$$\begin{array}{rll} & I_n(i,b,j) & :\to \sum_{f:\mathit{test}(f,b)\wedge \mathit{index}(f)=j} a(f)\cdot x \\ = & (I_n(i,b,j) \wedge b \not\simeq \emptyset_{fbag}) & :\to a(\mathit{frame}(j,b))\cdot x[\mathit{frame}(j,b)/f] \end{array}$$

Proof. From Lemma B.5.2, see Appendix, we know that:

$$\mathit{cnt}(j,b) = 1 \to (\quad \mathit{test}(f,b) \wedge \mathit{index}(f) = j \quad \leftrightarrow \quad \mathit{frame}(j,b) = f \quad)$$

As $I_n(i,b,j)$ implies $\mathit{cnt}(j,b) = 1$ we may replace the condition $\mathit{test}(f,b) \wedge \mathit{index}(f) = j$ by $\mathit{frame}(j,b)$, after which we apply axiom SUM2. $\square$

4.2 Adding the Invariant to the linear specification of the Bakery

In order to simplify the sum expression $\sum_{f:\mathit{test}(f,b)\wedge \mathit{index}(f)=j} a(f)\cdot x$, in the definitions of α_n and β_n, into $a(\mathit{frame}(j,b))\cdot x[\mathit{frame}(j,b)/f]$ we add the invariant to the definitions of $\alpha_n, \beta_n, \gamma_n$ and ϕ_n. Hence, we define the following auxiliary processes:

Definition 4.2.1 (Auxiliary processes with the Invariant)

$$\begin{array}{lll} \alpha_n^I(i:\mathsf{Nat}, b:\mathsf{FBag}, j:\mathsf{Nat}) & := & I_n(i,b,j) :\to \quad \alpha_n(i,b,j) \\ \beta_n^I(d:\mathsf{D}, i:\mathsf{Nat}, b:\mathsf{FBag}, j:\mathsf{Nat}) & := & I_n(i,b,j) :\to \quad \beta_n(d,i,b,j) \\ \gamma_n^I(i:\mathsf{Nat}, b:\mathsf{FBag}, j:\mathsf{Nat}, e:\mathsf{D}) & := & I_n(i,b,j) :\to \quad \gamma_n(i,b,j,e) \\ \phi_n^I(d:\mathsf{D}, i:\mathsf{Nat}, b:\mathsf{FBag}, j:\mathsf{Nat}, e:\mathsf{D}) & := & I_n(i,b,j) :\to \quad \phi_n(d,i,b,j,e) \end{array}$$

And we obtain:

Lemma 4.2.2

$$\begin{array}{lcl}
B'_n & = & \alpha^I_n(0,\emptyset_{fbag},0) \\[1em]
\alpha^I_n(i,b,j) & = & I_n(i,b,j) :\rightarrow \\
& & (\sum_{d:d\neq\bot} enter(d)\cdot\beta^I_n(d,i,b,j) \\
& & +b\not\simeq\emptyset_{fbag} :\rightarrow c_2(\mathit{frame}(j,b))\cdot\gamma^I_n(i,\mathit{rem}(j,b),j+_n 1,\mathit{frame}(j,b)))\ ) \\[1em]
\beta^I_n(d,i,b,j) & = & I_n(i,b,j) :\rightarrow \\
& & (size(b)<n :\rightarrow c_1(d^i)\cdot\alpha^I_n(i+_n 1,add(d^i,b),j) \\
& & +b\not\simeq\emptyset_{fbag} :\rightarrow c_2(\mathit{frame}(j,b))\cdot\phi^I_n(d,i,\mathit{rem}(j,b),j+_n 1,\mathit{frame}(j,b))\ ) \\[1em]
\gamma^I_n(i,b,j,e) & = & I_n(i,b,j) :\rightarrow \\
& & (\sum_{d:d\neq\bot} enter(d)\cdot\phi^I_n(d,i,b,j,e) \\
& & +out(e)\cdot\alpha^I_n(i,b,j)\ ) \\[1em]
\phi^I_n(d,i,b,j,e) & = & I_n(i,b,j) :\rightarrow \\
& & (size(b)<n :\rightarrow c_1(d^i)\cdot\gamma^I_n(i+_n 1,add(d^i,b),j,e) \\
& & +out(e)\cdot\beta^I_n(d,i,b,j)\ )
\end{array}$$

Proof. (Case α^I_n only.)

$$\begin{array}{l}
\alpha^I_n(i,b,j) \\
= \textbf{(}\ \mathit{def}\ \alpha^I_n,\ \mathit{def}\ \alpha_n,\ \text{COND1}\ \textbf{)} \\
\quad I_n(i,b,j) :\rightarrow\ \ \sum_{d:d\neq\bot} enter(d)\cdot\beta_n(d,i,b,j) \\
+I_n(i,b,j) :\rightarrow\ \ \sum_{f:test(f,b)\wedge index(f)=j} c_2(f)\cdot\gamma_n(i,\mathit{rem}(f,b),j+_n 1,data(f)) \\
= \textbf{(}\ \text{Lemma 4.1.2}\ \textbf{)} \\
\quad I_n(i,b,j) :\rightarrow\ \ \sum_{d:d\neq\bot} enter(d)\cdot\beta_n(d,i,b,j) \\
+(I_n(i,b,j)\wedge b\not\simeq\emptyset_{fbag}) :\rightarrow c_2(\mathit{frame}(j,b))\cdot\gamma_n(i,\mathit{rem}(\mathit{frame}(j,b),b),j+_n 1,data(\mathit{frame}(j,b))) \\
= \textbf{(}\ \text{SUM3},\ \mathit{def}\ \mathit{rem}(j,b),\ \mathit{def}\ data(j,b)\ \textbf{)} \\
\quad I_n(i,b,j) :\rightarrow\ \ \sum_{d:d\neq\bot} enter(d)\cdot(I_n(i,b,j) :\rightarrow \beta_n(d,i,b,j)) \\
+(I_n(i,b,j)\wedge b\not\simeq\emptyset_{fbag}) :\rightarrow c_2(\mathit{frame}(j,b))\cdot(I_n(i,\mathit{rem}(j,b),j+_n 1) :\rightarrow \gamma_n(i,\mathit{rem}(j,b),j+_n 1,data(j,b))) \\
= \textbf{(}\ \text{COND1},\ \mathit{def}\ \beta^I_n,\ \mathit{def}\ \gamma^I_n\ \textbf{)} \\
I_n(i,b,j) :\rightarrow\quad (\sum_{d:d\neq\bot} enter(d)\cdot\beta^I_n(d,i,b,j) \\
\qquad\qquad\qquad +b\not\simeq\emptyset_{fbag} :\rightarrow c_2(\mathit{frame}(j,b))\cdot\gamma^I_n(i,\mathit{rem}(j,b),j+_n 1,\mathit{frame}(j,b)))\ )
\end{array}$$

□

4.3 Some properties of the Invariant

For sequel use we state some properties that can be derived from the Invariant, all facts are obvious facts that one indeed may expect in a Bakery.

- (a) Each customer has a unique index.
- (b) If it is not full, then the amount of customers equals the difference between the in– and out–counter (modulo n).

- (c) If it is full, then the in–counter equals the out–counter.
- (d) The amount of customers does not exceeds the capacity.
- (e) If the two counters are equal, then the shop is either empty or full.

These facts are formalized as follows:

Lemma 4.3.1

$$\begin{array}{lllll}
(a) & I_n(i,b,j) \wedge k < size(b) & \rightarrow & cnt(j +_n k, b) = 1 \\
(b) & I_n(i,b,j) \wedge size(b) < n & \rightarrow & size(b) = i -_n j \\
(c) & I_n(i,b,j) \wedge size(b) = n & \rightarrow & i = j \\
(d) & I_n(i,b,j) & \rightarrow & size(b) \leq n \\
(e) & I_n(i,b,j) \wedge i = j & \rightarrow & size(b) = 0 \vee size(b) = n
\end{array}$$

Proof. See Appendix, Section A. □

Furthermore we have the following facts, that consider the Invariant itself. The first one corresponds with the situation were a customer that has just been served by the baker (i.e., the customer with index j), leaves the bakery, after which the baker increases the out–counter to $j +_n 1$. The second one corresponds with the situation in which a new customer arrives, which takes a label i and enters the bakery, after which the in–counter increases to $i +_n 1$.

Lemma 4.3.2

$$\begin{array}{llll}
(a) & I_n(i,b,j) & \rightarrow & I_n(i, rem(j,b), j +_n 1) \\
(b) & (I_n(i,b,j) \wedge size(b) < n) & \rightarrow & I_n(i +_n 1, add(d^i, b), j)
\end{array}$$

Proof. See Appendix, Section A. □

4.4 Adding the invariant to the specification of the Queue

In the previous section we have shown that all customers in the Bakery have a unique label each, such that the customers can be served in order of their label. In other words, the state of the Bakery, that is the triple (i,b,j), defines a queue, namely $frame(j,b), frame(j +_n 1), \ldots, frame(i -_n 1)$. This queue is denoted by $queue_n(i,b,j)$, its detailed definition is given in Section B.6.

In order to relate the Bakery, B_n, with the Queue Q_{n+2} we define two auxiliary processes:

Definition 4.4.1 (Auxiliary Queue–processes with the Invariant)

$$\begin{array}{lll}
\tilde{Q}^I_m(q : \mathsf{DQueue}, i : \mathsf{Nat}, b : \mathsf{FBag}, j : \mathsf{Nat}) & := & I_m(i,b,j) :\rightarrow Q_m(q) \\
Q^I_m(i : \mathsf{Nat}, b : \mathsf{FBag}, j : \mathsf{Nat}) & := & \tilde{Q}^I_m(queue_m(i,b,j), i, b, j)
\end{array}$$

Before we continue we state two obvious properties, that regards the function $queue_n(i,b,j)$ in the context of the invariant. The first one says that a new customer is placed at the back of the queue ($addbck(d, add(d_0, add(d_1, \ldots, add(d_n, \emptyset_{dqueue}) \ldots))) = add(d_0, add(d_1, \ldots, add(d_n, add(d, \emptyset_{dqueue})) \ldots))$, see for details the Appendix).

Lemma 4.4.2

$$\begin{array}{lllll} (a) & (I_n(i,b,j) \wedge size(b) < n) & \rightarrow & addbck(d, queue_n(i,b,j)) & = queue_n(i +_n 1, add(d^i, b), j) \\ (b) & I_n(i,b,j) & \rightarrow & size(b) & = size(queue_n(j,b)) \end{array}$$

Proof. See Appendix, Section A. □

Next, we prove that we can reformulate the definitions of $\tilde{Q}^I_m$ and Q^I_m, such that they are defined in their own terms, like the queue process itself as well.

Note, that the process $Q^I_m(i,b,j)$ is defined completely in terms of the in– and out–counter i,j and the shop floor b. The process $Q^I_m(i,b,j)$ corresponds with a Bakery that does not have separate positions for the customers at the in– and out–counter, these positions are considered to be part of the shop floor as well. As a consequence there are no communications anymore between the in–counter and the shop floor, resp. the shop floor and the out–counter.

Lemma 4.1

$$\begin{array}{llllll} (a) & \tilde{Q}^I_m(q,i,b,j) & = & I_m(i,b,j) :\rightarrow & & \\ & & (& size(q) \neq m & :\rightarrow & \sum_{d \neq \perp} enter(d) \cdot \tilde{Q}^I_m(addbck(d,q), i +_m 1, add(d^i, b), j) \\ & & + & size(q) \neq 0 & :\rightarrow & out(top(q)) \cdot \tilde{Q}^I_m(untop(q), i, rem(j,b), j +_m 1) \\ & &) & & & \end{array}$$

$$\begin{array}{llllll} (b) & Q^I_m(i,b,j) & = & I_m(i,b,j) :\rightarrow & & \\ & & (& size(b) \neq m & :\rightarrow & \sum_{d \neq \perp} enter(d) \cdot Q^I_m(i +_m 1, add(d^i, b), j) \\ & & + & size(b) \neq 0 & :\rightarrow & out(data(j,b) \cdot Q^I_m(i, rem(j,b), j +_m 1)\ ) \end{array}$$

Proof. Part (a)

$$\begin{array}{ll} \multicolumn{2}{l}{\tilde{Q}^I_m(q,i,b,j)} \\ \multicolumn{2}{l}{= \textbf{(}\ def\ \tilde{Q}^I_m,\ \text{SUM3},\ \text{Lemma } 4.3.2(b),(a)\ \textbf{)}} \\ \multicolumn{2}{l}{I_m(i,b,j) :\rightarrow} \\ (size(q) \neq m :\rightarrow & \sum_{d \neq \perp} enter(d) \cdot (I_m(i +_m 1, add(d^i, b), j) :\rightarrow Q_m(addbck(d,q)) \\ +size(q) \neq 0 :\rightarrow & out(top(q)) \cdot (I_m(i, rem(j,b), j +_m 1) :\rightarrow Q_m(untop(q))\ ) \\ \multicolumn{2}{l}{= \textbf{(}\ def\ \tilde{Q}^I_m\ \textbf{)}} \\ \multicolumn{2}{l}{I_m(i,b,j) :\rightarrow} \\ (size(q) \neq m :\rightarrow & \sum_{d \neq \perp} enter(d) \cdot \tilde{Q}^I_m(addbck(d,q), i +_m 1, add(d^i, b), j) \\ +size(q) \neq 0 :\rightarrow & out(top(q)) \cdot \tilde{Q}^I_m(untop(q), i, rem(j,b), j +_m 1)\ ) \end{array}$$

Part (b)

$Q^I_m(i,b,j)$
$= ($ *def* Q^I_m, part (a) $)$
$I_m(i,b,j) :\rightarrow$
$(\ size(queue_m(i,b,j)) \neq m :\rightarrow \sum_{d \neq \perp} enter(d) \cdot \tilde{Q}^I_m(addbck(d, queue_m(i,b,j)), i +_m 1, add(d^i, b), j)$
$+\ size(queue_m(i,b,j)) \neq 0 :\rightarrow out(top(queue_m(i,b,j))) \cdot \tilde{Q}^I_m(untop(queue_m(i,b,j)), i, rem(j,b), j +_m 1)\ )$
$= ($ Lemma $4.4.2(a)$, Proposition $B.5.1(a),(b)$ $)$
$I_m(i,b,j) :\rightarrow$
$(\ size(queue_m(i,b,j)) \neq m :\rightarrow \sum_{d \neq \perp} enter(d) \cdot \tilde{Q}^I_m(queue_m(i +_n 1, add(d^i, b), j), i +_m 1, add(d^i, b), j)$
$+\ size(queue_m(i,b,j)) \neq 0 :\rightarrow out(data(j,b) \cdot \tilde{Q}^I_m(queue_m(i, rem(j,b), j +_m 1), i, rem(j,b), j +_m 1)\ )$
$= ($ Lemma $4.4.2(b), size(b) \neq 0 \leftrightarrow b \not\simeq \emptyset_{fbag}$, *def* Q^I_m $)$
$I_m(i,b,j) :\rightarrow$
$(\ size(b) \neq m :\rightarrow \sum_{d \neq \perp} enter(d) \cdot Q^I_m(i +_m 1, add(d^i, b), j)$
$+\ size(b) \neq 0 :\rightarrow out(data(j,b) \cdot Q^I_m(i, rem(j,b), j +_m 1)\ )$

□

Finally, we use that $I_n(0, \emptyset_{fbag}, 0) = \mathbf{t}$, and we take the definitions of 2.2 and 4.4.1 together:

Corollary 4.4.3 $Q_m \quad = \quad Q_m(\emptyset_{dqueue}) \quad = \quad \tilde{Q}^I_m(\emptyset_{dqueue}, 0, \emptyset_{fbag}, 0) \quad = \quad Q^I_m(0, \emptyset_{fbag}, 0)$

5 Showing the Branching Bisimulation equivalence

From now on we consider the communication action actions c_1, c_2 that occur in the description of the Bakery, as internal actions. Hence, we take $H = \{c_1, c_2\}$ and we consider $\tau_H(\alpha^I_n(0, \emptyset_{fbag}, 0))$ in stead of $\alpha^I_n(0, \emptyset_{fbag}, 0)$.

Until now we have obtained that

$$\begin{array}{lllll} B_n & = & \tau_H(B'_n) & = & \tau_H(\alpha^I_n(0, \emptyset_{fbag}, 0)) \\ Q_m & = & & & Q^I_m(0, \emptyset_{fbag}, 0) \end{array}$$

So, in order to prove that B_n is (rooted) branching bisimilar with Q_{n+2} it is sufficient to prove that $\tau_H(\alpha^I_n(0, \emptyset_{fbag}, 0))$ is (rooted) branching bisimilar with $Q^I_{n+2}(0, \emptyset_{fbag}, 0)$. That is, we have to relate the states of these processes such that the transfer property holds, i.e., each step of one state can be mimicked properly by a state to which it is related.

The state $\tau_H(\alpha^I_n(i,b,j))$ of the Bakery corresponds with the state $Q^I_m(i', b', j')$ under the condition that for both states the invariant holds, $m = n+2$, and the queues that are defined by the triples (i,b,j) and (i', b', j') are equal. Note that in the state $\tau_H(\alpha^I_n(i,b,j))$ the positions at the in– and out–counter are empty.

In case of $\tau_H(\beta^I_n(d,i,b,j))$ there is a "new customer" d at the in–counter. Hence, $\tau_H(\beta^I_n(d,i,b,j))$ is related with $Q^I_m(i', b', j')$ under the condition that, among others, the queue of (i,b,j) with the "new customer" d added to the back corresponds with the queue of (i', b', j'). In other words the condition $R(\tau_H(\beta^I_n(d,i,b,j)), Q^I_m(i', b', j')$ contains the condition that $addbck(d, queue_n(i,b,j)) = queue_m(i', b', j')$.

The rooted branching bisimulation R, that defines when states are related, is defined as follows:

Definition 5.1 (The bisimulation R)

$$\begin{aligned}
R(\tau_H(\alpha^I_n(i,b,j))\ ,\ Q^I_m(i',b',j')) &= I_n(i,b,j) \wedge m=n+2 \wedge I_m(i',b',j') \\
&\quad \wedge\ queue_n(i,b,j) = queue_m(i',b',j') \\[1ex]
R(\tau_H(\beta^I_n(d,i,b,j))\ ,\ Q^I_m(i',b',j')) &= I_n(i,b,j) \wedge m=n+2 \wedge I_m(i',b',j') \\
&\quad \wedge\ addbck(d, queue_n(i,b,j)) = queue_m(i',b',j') \\[1ex]
R(\tau_H(\gamma^I_n(i,b,j,e))\ ,\ Q^I_m(i',b',j')) &= I_n(i,b,j) \wedge m=n+2 \wedge I_m(i',b',j') \\
&\quad \wedge\ add(e, queue_n(i,b,j)) = queue_m(i',b',j') \\[1ex]
R(\tau_H(\phi^I_n(d,i,b,j,e))\ ,\ Q^I_m(i',b',j')) &= I_n(i,b,j) \wedge m=n+2 \wedge I_m(i',b',j') \\
&\quad \wedge\ addbck(d, add(e, queue_n(i,b,j)) = queue_m(i',b',j')
\end{aligned}$$

The “symmetric” part, i.e., the equations for cases like $R(Q^I_m(i',b',j')\ ,\ \alpha^I_n(i,b,j))$ have been omitted.

One can easily check that

$$R(\alpha^I_n(0,\emptyset_{fbag},0)\ ,\ Q^I_{n+2}(0,\emptyset_{fbag},0)) = \mathbf{t},$$

and that R is rooted for $\alpha^I_n(0,\emptyset_{fbag},0)$ and $Q^I_{n+2}(0,\emptyset_{fbag},0))$, as the τ-transition of $\tau_H(\alpha^I_n(i,b,j))$ is not enabled in case b is empty. Moreover, R is symmetric by definition, so, for proving that R is indeed a branching bisimulation we only have to prove that R satifies the “transfer” property.

Lemma 5.2 *R is a rooted branching bisimulation.*

Proof. . First we prove that the $enter(d)$ summand of $\tau_H(\alpha^I_n(i,b,j))$ can be matched with the $enter(d)$ summand of $Q^I_m(i,b,j)$, i.e., we prove that

$$\begin{aligned}
& R(\tau_H(\alpha^I_n(i,b,j))\ ,\ Q^I_m(i',b',j')) \wedge I_n(i,b,j) \wedge d \neq \perp \\
\rightarrow\ & I_m(i',b',j') \wedge size(b') < m \wedge R(\tau_H(\beta^I_n(d,i,b,j))\ ,\ Q^I_m(i' +_m 1, add(d^{i'}, b'), j'))
\end{aligned}$$

Note that the part $I_n(i,b,j)$ in the premisse, and the parts $I_m(i,b,j)$ and $size(b) < m$ in the conclusion are redundant, as the are consequences as well of R.

$$\begin{aligned}
& R(\tau_H(\alpha^I_n(i,b,j))\ ,\ Q^I_m(i',b',j')) \wedge d \neq \perp \\
\leftrightarrow\ & (\ def\, R,\ \text{Lemma } 4.4.2(b),\ \text{Lemma } 4.3.1(d)\ ) \\
& I_n(i,b,j) \wedge m=n+2 \wedge I_m(i',b',j') \\
& \wedge queue_m(i',b',j') = queue_n(i,b,j) \wedge d \neq \perp \\
& \wedge size(b') = size(queue_m(i',b',j')) = size(queue_n(i,b,j)) = size(b) \leq n < n+2 = m \\
\rightarrow\ & (\ \text{Lemma } 4.3.2(b),\ \text{Lemma } 4.4.2(a)\ ) \\
& I_n(i,b,j) \wedge m=n+2 \wedge I_m(i' +_m 1, add(d^{i'}, b'), j') \\
& \wedge addbck(d, queue_n(i,b,j)) = addbck(d, queue_m(i',b',j')) = queue_m(i +_m 1, add(d^i, b'), j') \\
\leftrightarrow\ & (\ def\, R\ ) \\
& R(\tau_H(\beta^I_n(d,i,b,j))\ ,\ Q^I_m(i' +_m 1, add(d^{i'}, b'), j'))
\end{aligned}$$

□

References

[Bae90] J.C.M. Baeten, editor. *Applications of Process Algebra.* Cambridge Tracts in Theoretical Computer Science 17. Cambridge University Press, 1990.

[BG93] M. Bezem and J.F. Groote. Invariants in process algebra with data. Logic Group Preprint Series 98, Dept. of Philosophy, Utrecht University, September 1993.

[BK84] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Computation*, 60(1/3):109–137, 1984.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra.* Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of algebraic specifications I*, volume 6 of *EATCS Monographs on Theoretical Computer Science.* Springer-Verlag, 1985.

[FK95] W.J. Fokkink and A.S. Klusener. An effective axiomatization for real time ACP. *Information and Computation*, 122(2):286–299, 1995.

[GK94] J.F. Groote and H.P. Korver. A correctness proof of the bakery protocol in μCRL. In A. Ponse, C. Verhoef, and S.F.M. Vlijmen, editors, *Proceedings ACP'94*, pages 51–82. Report P9413, Programming Research Group, Univ. of Amsterdam, May 1994.

[GK95] W.O.D. Griffioen and H.P. Korver. Verification of a bakery protocol in the automata model. Unpublished document, 1995.

[GP91] J.F. Groote and A. Ponse. Proof theory for μCRL. Report CS-R9138, CWI, Amsterdam, 1991.

[GP94] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. In D.J. Andrews, J.F. Groote, and C.A. Middelburg, editors, *Proceedings of the International Workshop on Semantics of Specification Languages*, Workshops in Computer Science. Springer Verlag, May 1994. Full version is available as CWI Report CS-R9076, Amsterdam, The Netherlands.

[GvW94] J.F. Groote and J. van Wamel. Algebraic data types and induction in μCRL. Report P9409, University of Amsterdam, 1994.

[GW91] R.J. van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics. Report CS-R9120, CWI, Amsterdam, 1991. An extended abstract of an earlier version has appeared in G.X. Ritter, editor, *Information Processing 89*, North-Holland, 1989.

[KvW95] H.P. Korver and J. van Wamel. Two rules for many sorted constructor induction. 1995. Appeared as chapter 5 in [vW95].

[Mil89] R. Milner. *Communication and Concurrency.* Prentice-Hall International, Englewood Cliffs, 1989.

[vW95] J. van Wamel. *Verification for elemetary data types and retransmission protocols.* PhD thesis, Department of Mathematics and Computing Science, University of Amsterdam, September 1995.

A A more detailed treatment of the Invariant

In the sequel we will prove some lemmas that involve the invariant by induction on the size of the bag b. However, this does not fit directly into the induction scheme of [KvW95]. To remain within the context of [KvW95] we introduce a variant of the Invariant:

Definition A.1 (The Alternative Invariant) $I : \mathsf{Nat} \times \mathsf{Nat} \times \mathsf{FBag} \times \mathsf{Nat} \times \mathsf{Nat} \to \mathsf{Bool}$

$$\begin{array}{lll} I'_n(i,b,j,m) & := & i < n \wedge j < n \\ & & \wedge \quad (\quad cnt(j,b) = 1 \wedge I'_n(i, rem(j,b), j +_n 1, m-1)) \\ & & \quad\;\; \vee \quad (i = j \wedge b = \emptyset_{fbag} \wedge m = 0)) \end{array}$$

Using the *Special Bag Induction 2* from [GvW94], see Definition 7.7 of that paper, we can prove by induction on b that

$$I_n(i,b,j) \leftrightarrow I'_n(i,b,j,size(b))$$

When we say that we prove a property, say θ, that involves $I_n(i,b,j)$, by induction on the size of b we mean formally that we prove the corresponding property, say θ', that involves $I'_n(i,b,j,m)$ by induction on m, after which we apply the above equivalence regarding $I_n(i,b,j)$ and $I'_n(i,b,j,size(b))$ to obtain the proof of the original property θ.

Lemma 4.3.1

$$\begin{array}{llll} (a) & I_n(i,b,j) \wedge k < size(b) & \to & cnt(j +_n k, b) = 1 \\ (b) & I_n(i,b,j) \wedge size(b) < n & \to & size(b) = i -_n j \\ (c) & I_n(i,b,j) \wedge size(b) = n & \to & i = j \\ (d) & I_n(i,b,j) & \to & size(b) \leq n \\ (e) & I_n(i,b,j) \wedge i = j & \to & size(b) = 0 \vee size(b) = n \end{array}$$

Proof. Proof of (a), by induction to the size of b.

$size(b) = 0$ hence $b = \emptyset_{fbag}$:

$$I_n(i,\emptyset_{fbag},j) \wedge k < 0 \;\leftrightarrow\; I_n(i,\emptyset_{fbag},j) \wedge \mathbf{f} \;\leftrightarrow\; \mathbf{f} \;\to\; cnt(j +_n, b) = 1$$

$size(b) > 1$, by case distinction; either$k = 0$ or $k > 0$

$$I_n(i,b,j) \wedge k = 0 \;\to\; cnt(j +_n k, b) = cnt(j,b) = 1$$

$$\begin{array}{ll} & I_n(i,b,j) \wedge 0 < k < size(b) \\ \to & (\text{ Lemma 4.3.2, Proposition } B.5.1 \,) \\ & 0 < k < size(b) \wedge I_n(i, rmv(j,b), j +_n 1) \wedge k - 1 < size(rmv(j,b)) \\ \to & 1 \overset{\text{ind}}{=} cnt((j +_n 1) + (k-1), rmv(j,b)) = cnt(j +_n k, rmv(j,b)) = cnt(j +_n k, b) \end{array}$$

Proof of (b), by induction to the size of b as well.

$size(b) = 0$ hence $b = \emptyset_{fbag}$:

$$I_n(i, \emptyset_{fbag}, j) \quad \rightarrow \quad i = j \quad \leftrightarrow \quad i -_n j = 0 = size(\emptyset_{fbag})$$

$size(b) > 0$:

$$\begin{array}{ll} & I_n(i,b,j) \wedge size(b) < n \\ \rightarrow & I_n(i, rmv(j,b), j +_n 1) \wedge size(rmv(j,b)) < n \\ \rightarrow & size(b) = size(rmv(j,b)) + 1 \overset{\text{ind}}{=} i -_n (j +_n 1) + 1 = i_n j \end{array}$$

For the proof of fact (c) we conclude from part (b) that

$$n -_n 1 = size(rmv(j,b)) = i -_n (j +_n 1)$$

hence $i -_n j = 0$, so i, j are equal modulo n, and as both i and j are smaller than n we obtain $i = j$.

Part (d) is a direct corollary from the previous parts. $\square$

Lemma 4.3.2

$$\begin{array}{lll} (a) & I_n(i,b,j) & \rightarrow \quad I_n(i, rem(j,b), j +_n 1) \\ (b) & (I_n(i,b,j) \wedge size(b) < n) & \rightarrow \quad I_n(i +_n 1, add(d^i, b), j) \end{array}$$

Proof. Part (a) follows directly from the definition of I_n, part (b) is proven by induction on the size of b.

$size(b) = 0$ hence $b = \emptyset_{fbag}$:

$$\begin{array}{ll} & I_n(i, \emptyset_{fbag}, j) \wedge size(\emptyset_{fbag}) < n \\ \leftrightarrow & (\ def\, I_n,\ cnt(j, \emptyset_{fbag}) \neq 1\ ) \\ & i < n \wedge j < n \wedge i = j \\ \leftrightarrow & i < n \wedge j < n \wedge i = j \\ & \wedge(\ \ i +_n 1 < n \wedge j +_n 1 < n \wedge i +_n 1 = j +_n 1 \\ & \quad \wedge rem(j, add(d^i, \emptyset_{fbag})) = rem(i, add(d^i, \emptyset_{fbag})) = \emptyset_{fbag}\ \ ) \\ \leftrightarrow & i < n \wedge j < n \wedge i = j \wedge I_n(i +_n 1, rem(j, add(d^i, \emptyset_{fbag})), j +_n 1) \\ \rightarrow & i +_n 1 < n \wedge j < n \wedge cnt(j, add(d^i, \emptyset_{fbag})) = cnt(i, add(d^i, \emptyset_{fbag})) = 1 \\ & \wedge I_n(i +_n 1, rem(j, add(d^i, \emptyset_{fbag})), j +_n 1) \\ \leftrightarrow & I_n(i +_n 1, add(d^i, \emptyset_{fbag}), j) \end{array}$$

$size(b)n + 1 > 0$:

$$\begin{array}{ll} & I_n(i,b,j) \wedge 0 < size(b) < n \\ \rightarrow & (\ def\, I_n,\ \text{Lemma 4.3.1}\ ) \\ & i < n \wedge j < n \wedge cnt(j,b) = 1 \wedge I_n(i, rem(j,b), j +_n 1) \\ \rightarrow & (\ \text{Induction}\ \ ) \\ & i < n \wedge j < n \wedge cnt(j,b) = 1 \wedge I_n(i +_n 1, add(d^i, rem(j,b)), j +_n 1) \\ \rightarrow & (\ \text{in case } i \neq j \text{ then } cnt(j, add(d^i, b)) = cnt(j,b) \text{ and } add(d^i, rem(j,b)) = rem(j, add(d^i, b))\ ) \\ & i +_n 1 < n \wedge j < n \wedge cnt(j, add(d^i, b)) = 1 \wedge I_n(i +_n 1, rem(j, add(d^i, b)), j +_n 1) \\ \leftrightarrow & (\ def\, I_n\ ) \\ & I_n(i +_n 1, add(d^i, b), j +_n 1) \end{array}$$

$\square$

Lemma 4.4.2

$$\begin{array}{llcrl}
(a) & (I_n(i,b,j) \wedge size(b) < n) & \rightarrow & addbck(d, queue_n(i,b,j)) & = queue_n(i +_n 1, add(d^i, b), j) \\
(b) & I_n(i,b,j) & \rightarrow & size(b) & = size(queue_n(j,b))
\end{array}$$

Proof. The proof of (a) is similar to the proof of part (b) of Lemma 4.3.2.

$size(b) = 0$ hence $b = \emptyset_{fbag}$:

$$\begin{array}{ll}
 & I_n(i, \emptyset_{fbag}, j) \wedge size(\emptyset_{fbag}) = 0 < n \\
\rightarrow & \textbf{(} \ def\, I_n, \ cnt(j, \emptyset_{fbag}) \neq 1 \ \textbf{)} \\
 & i = j \\
\leftrightarrow & i = j \wedge \\
 & \quad addbck(d, queue_n(i, \emptyset_{fbag}, j)) \\
 & = \ addbck(d, \emptyset_{dqueue}) \\
 & = \ add(d, \emptyset_{dqueue}) \\
 & = \ addbck(data(j, add(d^i, \emptyset_{fbag})), queue_n(i +_n 1, \emptyset_{fbag}, j +_n 1)) \\
 & = \ queue_n(i +_n 1, add(d^i, \emptyset_{fbag}), j +_n 1))
\end{array}$$

$size(b) > 0$:

$$\begin{array}{ll}
 & I_n(i,b,j) \wedge 0 < size(b) < n \\
\rightarrow & i \neq j \wedge \\
 & \quad addbck(d, queue_n(i,b,j)) \\
 & = \ addbck(d, add(data(j,b), queue_n(i, rem(j,b), j +_n 1))) \\
 & = \ add(data(j,b), addbck(d, queue_n(i, rem(j,b), j +_n 1))) \\
 & \overset{\text{ind}}{=} \ add(data(j,b), addbck(d, queue_n(i +_n 1, add(d^i, rem(j,b)), j +_n 1))) \\
 & = \ add(data(j, add(d^i, b)), addbck(d, queue_n(i +_n 1, rem(j, add(d^i, b)), j +_n 1))) \\
 & = \ queue_n(i +_n 1, add(d^i, b), j)
\end{array}$$

Part (b) is proven as well by induction on the size of b.

$size(b) = 0$ hence $b = \emptyset_{fbag}$:

$$\begin{array}{ll}
 & I_n(i, \emptyset_{fbag}, j) \\
\rightarrow & i = j \wedge size(\emptyset_{fbag}) = 0 = size(\emptyset_{dqueue}) = size(queue_n(i, \emptyset_{fbag}, j))
\end{array}$$

$size(b) > 0$:

$$\begin{array}{ll}
 & I_n(i,b,j) \\
\leftrightarrow & \textbf{(} \text{ in case } b \neq \emptyset_{fbag} \text{ then } cnt(j,b) = 1 \text{ and thus } size(b) = S(size(rem(j,b))) \ \textbf{)} \\
 & I_n(i,b,j) \wedge I_n(i, rem(j,b), j +_n 1) \wedge \\
 & \quad size(queue_n(i,b,j)) \\
 & = \ size(add(data(j,b), queue_n(i, rem(j,b), j +_n 1))) \\
 & = \ S(size(queue_n(i, rem(j,b), j +_n 1))) \\
 & = \ S(size(rem(j,b)) \\
 & = \ size(b)
\end{array}$$

$\square$

B Datatypes

B.1 The datatype Bool

sort	Bool
cons	$\mathbf{t}, \mathbf{f} :\rightarrow \mathsf{Bool}$
func	$\neg : \mathsf{Bool} \rightarrow \mathsf{Bool}$
	$\wedge, \vee, \Rightarrow, eq : \mathsf{Bool} \times \mathsf{Bool} \rightarrow \mathsf{Bool}$
	$\mathit{if} : \mathsf{Bool} \times \mathsf{Bool} \times \mathsf{Bool} \rightarrow \mathsf{Bool}$
var	$b, b' : \mathsf{Bool}$
note	Infix notation is used for $\wedge, \vee$ and $\Rightarrow$.
rew	$\neg\mathbf{t} = \mathbf{f}$
	$\neg\mathbf{f} = \mathbf{t}$
	$\mathbf{t} \wedge b = b$
	$\mathbf{f} \wedge b = \mathbf{f}$
	$\mathbf{t} \vee b = \mathbf{t}$
	$\mathbf{f} \vee b = b$
	$b \Rightarrow b' = (\neg b) \vee b'$
	$eq(\mathbf{t}, \mathbf{t}) = \mathbf{t}$
	$eq(\mathbf{t}, \mathbf{f}) = \mathbf{f}$
	$eq(\mathbf{f}, \mathbf{t}) = \mathbf{f}$
	$eq(\mathbf{f}, \mathbf{f}) = \mathbf{t}$
	$\mathit{if}(\mathbf{t}, b, b') = b$
	$\mathit{if}(\mathbf{f}, b, b') = b'$

B.2 The datatype Nat, with modulo arithmatic

sort	Nat
cons	$0 :\to \mathsf{Nat}$
	$S : \mathsf{Nat} \to \mathsf{Nat}$
func	$P : \mathsf{Nat} \to \mathsf{Nat}$
	$+, \dot{-} : \mathsf{Nat} \times \mathsf{Nat} \to \mathsf{Nat}$
	$eq, <, \leq : \mathsf{Nat} \times \mathsf{Nat} \to \mathsf{Bool}$
	$mod : \mathsf{Nat} \times \mathsf{Nat} \to \mathsf{Nat}$
	$+, - : \mathsf{Nat} \times \mathsf{Nat} \times \mathsf{Nat} \to \mathsf{Nat}$
	$if : \mathsf{Bool} \times \mathsf{Nat} \times \mathsf{Nat} \to \mathsf{Nat}$
var	$n, m, k, i, j : \mathsf{Nat}$
note	Infix notation is used for $+, \dot{-}, \leq, <$ and $+_n$
rew	$P(0) = 0$
	$P(Sn) = n$
	$n + 0 = n$
	$n + Sn = S(n + m)$
	$n \dot{-} 0 = n$
	$n \dot{-} Sm = P(n \dot{-} m)$
	$eq(0, 0) = \mathbf{t}$
	$eq(0, Sn) = \mathbf{f}$
	$eq(Sn, 0) = \mathbf{f}$
	$eq(Sn, Sm) = eq(n, m)$
	$0 \leq n = \mathbf{t}$
	$Sn \leq 0 = \mathbf{f}$
	$Sn \leq Sm = n \leq m$
	$n < m = Sn \leq m$
	$m \ mod \ 0 = 0$
	$m \ mod \ Sn = if(Sn \leq m, (m \dot{-} Sn) \ mod \ Sn, m)$
	$k +_n m = (k + m) \ mod \ n$
	$k +_n m = if(m \ mod \ n \leq k \ mod \ n, k \ mod \ n \dot{-} m \ mod \ n, n \dot{-} (m \ mod \ n \dot{-} k \ mod \ n))$
	$if(\mathbf{t}, n, m) = n$
	$if(\mathbf{f}, n, m) = m$

B.3 The datatype D

sort	D
cons	$\bot : - > \mathsf{D}$
	$d : \mathsf{Nat} - > \mathsf{D}$
func	$eq : \mathsf{D} \times \mathsf{D} \to \mathsf{Bool}$
	$\mathit{if} : \mathsf{Bool} \times \mathsf{D} \times \mathsf{D} \to \mathsf{D}$
var	$d, d', e, e' : \mathsf{D}$
note	The injection $d : \mathsf{Nat} - > \mathsf{D}$ is arbitrary; it serves only to produce some elements of type D; a finite number of constants may serve as well
rew	$eq(\bot, \bot) = \mathbf{t}$
	$eq(\bot, d(n)) = \mathbf{f}$
	$eq(d(n), \bot) = \mathbf{f}$
	$eq(d(n), d(n')) = eq(n, n')$
	$\mathit{if}(\mathbf{t}, d, d') = d$
	$\mathit{if}(\mathbf{f}, d, d') = d'$

B.4 The datatype Frame

sort	Frame
cons	$frame : \mathsf{D} \times \mathsf{Nat} \to \mathsf{Frame}$
func	$data : \mathsf{Frame} \to \mathsf{D}$
	$index : \mathsf{Frame} \to \mathsf{Nat}$
	$\mathit{if} : \mathsf{Bool} \times \mathsf{Frame} \times \mathsf{Frame} \to \mathsf{Frame}$
var	$f, f' : \mathsf{Frame}$
note	$frame(d, i)$ is denoted by d^i
rew	$data(d^i) = d$
	$index(d^i) = i$
	$\mathit{if}(\mathbf{t}, f, f') = f$
	$\mathit{if}(\mathbf{f}, f, f') = f'$

B.5 The datatype FBag

sort	FBag
cons	$\emptyset_{fbag} :\rightarrow \mathsf{FBag}$
	$add : \mathsf{Frame} \times \mathsf{FBag} \rightarrow \mathsf{FBag}$
func	$rmv : \mathsf{Frame} \times \mathsf{FBag} \rightarrow \mathsf{FBag}$
	$test : \mathsf{Frame} \times \mathsf{FBag} \rightarrow \mathsf{Bool}$
	$eq : \mathsf{FBag} \times \mathsf{FBag} \rightarrow \mathsf{FBag}$
	$size : \mathsf{FBag} \rightarrow \mathsf{Nat}$
	$cnt : \mathsf{Nat} \times \mathsf{FBag} \rightarrow \mathsf{Nat}$
	$frame : \mathsf{Nat} \times \mathsf{FBag} \rightarrow \mathsf{Frame}$
	$data : \mathsf{Nat} \times \mathsf{FBag} \rightarrow \mathsf{D}$
	$rem : \mathsf{Nat} \times \mathsf{FBag} \rightarrow \mathsf{FBag}$
	$if : \mathsf{Bool} \times \times \mathsf{FBag} \times \mathsf{FBag} \rightarrow \mathsf{FBag}$
var	$b, b' : \mathsf{FBag}$
note	The functions *cnt*, *frame*, *data* and *rem* are specific for the bakery protocol; their definitions use the fact that a *frame* is a pair, of which the second element is an index in Nat
rew	$add(f, add(f', b)) = add(f', add(f, b))$
	$rem(f, \emptyset_{fbag}) = \emptyset_{fbag}$
	$rem(f, add(f', b)) = if(f = f', b, add(f', rem(f, b)))$
	$test(f, \emptyset_{fbag}) = \mathbf{f}$
	$test(f, add(f', b)) = if(f = f', \mathbf{t}, test(f, b))$
	$eq(\emptyset_{fbag}, \emptyset_{fbag}) = \mathbf{t}$
	$eq(\emptyset_{fbag}, add(f, b)) = \mathbf{f}$
	$eq(add(f, b), \emptyset_{fbag}) = \mathbf{f}$
	$eq(add(f, b), b') = test(f, b') \wedge eq(b, rem(f, b'))$
	$size(\emptyset_{fbag}) = 0$
	$size(add(f, b)) = S(size(b))$
	$cnt(j, \emptyset_{fbag}) = 0$
	$cnt(j, add(f, b)) = if(index(f) = j, S(cnt(j, b)), cnt(j, b))$
	$frame(j, \emptyset_{fbag}) = \perp^j$
	$frame(j, add(f, b)) = if(index(f) = j, if(cnt(j, b) = 0, f, \perp^j), frame(j, b))$
	$data(j, b) = data(frame(j, b))$
	$rem(j, b) = rem(frame(j, b), b)$
	$if(\mathbf{t}, b, b') = b$
	$if(\mathbf{f}, b, b') = b'$

Proposition B.5.1

$$\begin{array}{lcl} cnt(j, b) = 1 & \rightarrow & size(b) = size(rem(j, b)) + 1 \\ test(j, b) & \leftrightarrow & cnt(index(f), b) > 0 \end{array}$$

Proof. Omitted. □

Lemma B.5.2

$$cnt(j,b)=1 \rightarrow (\quad test(f,b) \wedge index(f)=j \quad \leftrightarrow \quad frame(j,b)=f \quad)$$

Proof. By induction on b. The base case is trivial, as $cnt(j,\emptyset_{fbag})=0 \neq 1$.

For the inductive case, $b=add(f',b')$, we prove the equivalent property

$$\begin{array}{ll}
 & cnt(j,add(f',b'))=1 \wedge test(f,add(f',b')) \wedge index(f)=j \\
\leftrightarrow & cnt(j,add(f',b'))=1 \wedge \\
 & (\ f=f' \wedge cnt(j,b')=0 \wedge index(f)=j \\
 & \vee f \neq f' \wedge index(f')=j=index(f) \wedge cnt(j,b')=0=test(f,b') \\
 & \vee f \neq f' \wedge index(f') \neq j \wedge cnt(j,b')=1 \wedge test(f,b')f)=j\ ) \\
\leftrightarrow & \textbf{(}\ def\, frame(j,b),\ test(f,b) \rightarrow cnt(index(f),b) \neq 0, \text{Induction}\ \ \textbf{)} \\
 & cnt(j,add(f',b'))=1 \wedge \\
 & (\ f=f' \wedge frame(j,add(f,b))=f \\
 & \vee f \neq f' \wedge index(f')=j=index(f) \wedge \mathbf{f} \\
 & \vee f \neq f' \wedge index(f') \neq j \wedge frame(j,b)=f\ ) \\
\leftrightarrow & cnt(j,add(f',b'))=1 \wedge \\
 & (\ f=f' \wedge frame(j,add(f',b))=f \\
 & \vee \mathbf{f} \\
 & \vee f \neq f' \wedge index(f') \neq j \wedge frame(j,add(f',b))=f\ ) \\
\leftrightarrow & cnt(j,b)=1 \wedge frame(j,b)=f
\end{array}$$

□

B.6 The datatype DQueue

sort	DQueue
cons	$\emptyset_{dqueue} :\rightarrow \mathsf{DQueue}$
	$add : \mathsf{D} \times \mathsf{DQueue} \rightarrow \mathsf{DQueue}$
func	$addbck : \mathsf{D} \times \mathsf{DQueue} \rightarrow \mathsf{DQueue}$
	$top : \mathsf{DQueue} \rightarrow \mathsf{D}$
	$untop : \mathsf{DQueue} \rightarrow \mathsf{DQueue}$
	$queue : \mathsf{Nat} \times \mathsf{Nat} \times \mathsf{FBag} \times \mathsf{Nat} \rightarrow \mathsf{DQueue}$
	$if : \mathsf{Bool} \times \mathsf{DQueue} \times \mathsf{DQueue} \rightarrow \mathsf{DQueue}$
var	$q, q' : \mathsf{DQueue}$
note	$queue_n(i, b, j,)$ is specific for the bakery protocol;
	it takes the (unique) data for all indices in between i and j
	out of the bag b, and puts them in order of their indices in a queue
rew	$addbck(d, \emptyset_{dqueue}) = add(d, \emptyset_{dqueue})$
	$addbck(d', add(d, q)) = add(d, addbck(d', q))$
	$top(\emptyset_{dqueue}) = \perp$
	$top(add(d, q)) = if(q = \emptyset_{dqueue}, d, top(q))$
	$untop(\emptyset_{dqueue}) = \emptyset_{dqueue}$
	$untop(add(d, q)) = q$
	$queue_n(i, b, j) = if(i = j \wedge b = \emptyset_{fbag}, \emptyset_{dqueue}, add(data(j, b), queue_n(i, rem(j, b), j +_n 1)))$
	$if(\mathbf{t}, q, q') = q$
	$if(\mathbf{f}, q, q') = q'$

Lemma B.6.1

$$\begin{array}{llcrl}
(a) & b \not\simeq \emptyset_{fbag} & \rightarrow & untop(queue_n(i, b, j)) & = queue_n(i, rem(j, b), j +_n 1) \\
(b) & b \not\simeq \emptyset_{fbag} & \rightarrow & top(queue_n(i, b, j)) & = data(j, b)
\end{array}$$

Proof.

$$\begin{array}{lll}
b \neq \emptyset_{fbag} \rightarrow & untop(queue_n(i, b, j)) & = untop(add(data(j, b), queue_n(i, rem(j, b), j +_n 1)) \\
 & & = queue_n(i, rem(j, b), j +_n 1) \\
b \neq \emptyset_{fbag} \rightarrow & top(queue_n(i, b, j)) & = top(add(data(j, b), queue_n(i, rem(j, b), j +_n 1)) \\
 & & = data(j, b)
\end{array}$$

□

C The axiom system

In Table C the axioms for $\mathrm{ACP}_{\mathrm{pS}}$are given. In that table $var(\phi)$ denotes the set of data variables that occur in the boolean expression ϕ, $fv(x)$ denotes the set of free, unbound, data variables of x. Note that the prefix sum $\sum_{\phi} a(v) \cdot x$ binds all occurrences of the variable v in x.

A1		$x + y$	$=$	$y + x$
A2		$(x + y) + z$	$=$	$x + (y + z)$
$\mathrm{A3_{pS}}$		$\sum_\phi a(v) \cdot x + \sum_\psi a(v) \cdot x$	$=$	$\sum_{\phi \vee \psi} a(v) \cdot x$
A6		$x + \delta$	$=$	x
SUM1		$\sum_{\mathbf{f}} a(v) \cdot x$	$=$	δ
SUM2		$\sum_{v=e} a(v) \cdot x$	$=$	$\sum_{v=e} a(v) \cdot x[e/v]$
SUM3		$\sum_\phi a(v) \cdot x$	$=$	$\sum_\phi a(v) \cdot \phi :\rightarrow x$
COND1		$\phi :\rightarrow (x + y)$	$=$	$\phi :\rightarrow x + \phi :\rightarrow y$
COND2	$v \notin var(\phi)$	$\phi :\rightarrow \sum_\psi a(v) \cdot x$	$=$	$\sum_{\phi \wedge \psi} a(v) \cdot x$
$\mathrm{CF1_{pS}}$	$\gamma(a, b)$ is defined	$\sum_\phi a(v) \cdot x \vert \sum_\psi b(v) \cdot y$	$=$	$\sum_{\phi \wedge \psi} \gamma(a, b)(v) \cdot (x \Vert y)$
$\mathrm{CF2_{pS}}$	otherwise	$\sum_\phi a(v) \cdot x \vert \sum_\psi b(v) \cdot y$	$=$	δ
CM1		$x \Vert y$	$=$	$x \mathbin{⌊\!⌊} y + x \mathbin{⌊\!⌊} y + x \vert y$
$\mathrm{CM3_{pS}}$	$v \notin fv(y)$	$(\sum_\phi a(v) \cdot x) \mathbin{⌊\!⌊} y$	$=$	$\sum_\phi a(v) \cdot (x \Vert y)$
CM4		$(x_1 + x_2) \mathbin{⌊\!⌊} y$	$=$	$x_1 \mathbin{⌊\!⌊} y + x_2 \mathbin{⌊\!⌊} y$
CM8		$(x_1 + x_2) \vert y$	$=$	$x_1 \vert y + x_2 \vert y$
CM9		$x \vert (y_1 + y_2)$	$=$	$x \vert y_1 + x \vert y_2$
$\mathrm{D1_{pS}}$	$a \notin H$	$\partial_H(\sum_\phi a(v) \cdot x)$	$=$	$\sum_\phi a(v) \cdot \partial_H(x)$
$\mathrm{D2_{pS}}$	$a \in H$	$\partial_H(\sum_\phi a(v) \cdot x)$	$=$	δ
D3		$\partial_H(x + y)$	$=$	$\partial_H(x) + \partial_H(y)$
$\mathrm{TI1_{pS}}$	$a \notin H$	$\tau_H(\sum_\phi a(v) \cdot x)$	$=$	$\sum_\phi a(v) \cdot \tau_H(x)$
$\mathrm{TI2_{pS}}$	$a \in H$	$\tau_H(\sum_\phi a(v) \cdot x)$	$=$	$\sum_\phi \tau(v) \cdot \tau_H(x)$
TI3		$\tau_H(x + y)$	$=$	$\tau_H(x) + \tau_H(y)$

Table 1: The axiom system for ACP with prefix summation