# A Functional Programming Approach to the Specification and Verification of Concurrent Systems

Peter Dybjer *  Herbert Sander *

**Abstract**

Networks of communicating processes can be viewed as networks of stream transformers and programmed in a lazy functional language. In this way the correctness of concurrent systems can be reduced to the correctness of functional programs. In this paper such correctness is proved formally in the $\mu$-calculus extended with recursion equations for functional programs. The $\mu$-calculus is chosen since it allows the definition of properties by least fixed points (induction) as well as by greatest fixed point (coinduction) and since greatest fixed points are useful for formalizing properties, such as fairness, of infinitely proceeding programs. Moreover, non-deterministic processes are represented as incompletely specified deterministic processes, that is, as properies of stream transformers. The method is illustrated by proving the correctness of the alternating bit protocol. The $\mu$-calculus has been implemented in Paulson's Isabelle system and the correctness of the protocol has been proved mechanically.

## 1 What is a specification of a concurrent system?

Let us first recall what we mean by a specification of a single program. In 'external' logic a specification is a *property* of a program. (Recently there has been much interest in 'integrated' programming logics, such as Martin-Löf's type theory. See Dybjer [**?**] for a discussion about external versus integrated programming logics.) Usually, such a specification is divided into an input condition $P$ and an input-output relation $R$. The specification then has the form

$$Spec\ f \equiv \forall x \in P.x\, R\, (f\ x).$$

Can this notion of specification be generalised to concurrent systems in a natural way? The idea is to model a system as a Kahn-network or, if there are non-deterministic agents, as an incompletely specified Kahn-network.

Consider a system which satisfies the following criteria:

- The topology of the system is fixed. Input and output channels are determined. Streams of messages are communicated between the agents.
- Agents are separated into those which we wish to program and those which are predetermined. The latter kind may be non-deterministic.

*Preliminary version of a paper which appeared in Formal Aspects of Computing Volume 1 Number 4, 1989, pp 303-319

Authors' address: Programming Methodology Group, Department of Computer Sciences, Chalmers University of Technology and University of Göteborg, S-412 96 Göteborg, Sweden. Electronic mail: peterd@cs.chalmers.se, herbert@cs.chalmers.se.

- The purpose of the system is to realise a certain relation $R$ between inputs and outputs, provided that the inputs satisfy certain conditions $P$, and the predetermined agents satisfy certain other conditions $Q$.

Let $h$ be the stream transformers associated with the agents (the vector notation indicates that there may be several agents), let $is$ be the input streams, and let $js$ be the output streams. Then, since the topology is fixed, there are network transfer functions $trans$ (one for each output channel) such that

$$js = trans\ h\ is.$$

Note that these functions $trans$ are Kahn-network analogues of function application and that they only depend on the topology of the network.

By our second assumption, agents (and thus stream transformers) are separated into those which we wish to program and those which are predetermined. If we use $f$ for the former and $g$ for the latter, we can rewrite the equation:

$$js = trans\ f\ g\ is.$$

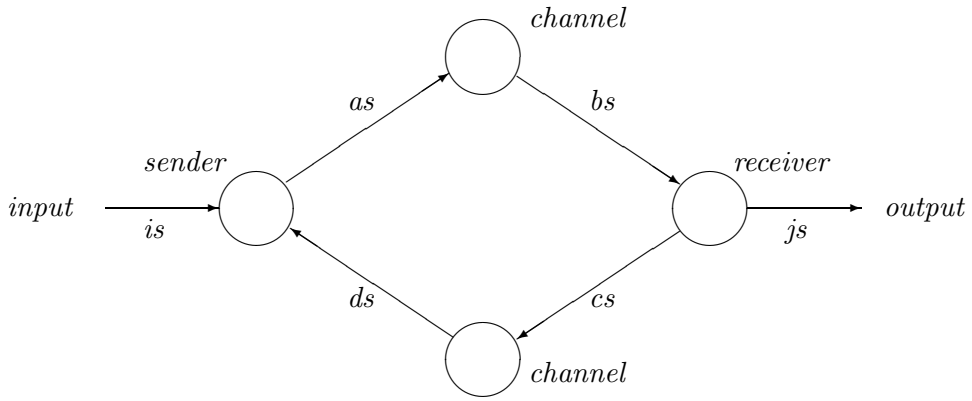
By our third assumption, the intended behaviour of the network is specified by the input conditions $P$, by the conditions on the predetermined agents $Q$, and by the input-output relation $R$. Thus the specification of our task, that is, to program $f$, is

$$Spec\ f \equiv \forall g \in Q. \forall is \in P. is\ R\ (trans\ f\ g\ is).$$

We have thus shown the analogy between the specification of a concurrent system, which satisfies the criteria above, and the specification of a single program.

The specification of a communication protocol satisfies our three criteria:

- It has a fixed topology:




- Two of the four agents (the channels) are predetermined and our task is to program the other two (the sender and the receiver).

- The purpose of the system is to produce output which is extensionally equal to the input, provided the input is a stream of data items. In our proof below we assume that the input stream is infinite. We know that the channels are unreliable transmitters, that is, that each item is either correctly or erroneously transmitted. We also assume that they are fair in the sense that for each 'time' there is a later 'time' when they transmit an item correctly.

Let *trans* be the network transfer function for the protocol topology. It satisfies

$$js = trans\ f_1\ f_2\ f_3\ g_1\ g_2\ is\,,$$

where *is* is the input stream, *js* the output stream, $f_1$ the function associated with the sender, $f_2$ and $f_3$ the functions associated with the receiver, and $g_1$ and $g_2$ the functions associated with the the channels. We can determine *js* from *is* by using the following system of equations:

$$\begin{aligned} as &= f_1\ is\ ds\,,\\ bs &= g_1\ as\,,\\ cs &= f_2\ bs\,,\\ ds &= g_2\ cs\,,\\ js &= f_3\ bs\,. \end{aligned}$$

So the recursion equations for *trans* are the following:(forall ...)

$$\begin{aligned} h_{as}\ f_1\ f_2\ f_3\ g_1\ g_2\ is &= f_1\ is\ (h_{ds}\ f_1\ f_2\ f_3\ g_1\ g_2\ is),\\ h_{bs}\ f_1\ f_2\ f_3\ g_1\ g_2\ is &= g_1\ (h_{as}\ f_1\ f_2\ f_3\ g_1\ g_2\ is),\\ h_{cs}\ f_1\ f_2\ f_3\ g_1\ g_2\ is &= f_2\ (h_{bs}\ f_1\ f_2\ f_3\ g_1\ g_2\ is),\\ h_{ds}\ f_1\ f_2\ f_3\ g_1\ g_2\ is &= g_2\ (h_{cs}\ f_1\ f_2\ f_3\ g_1\ g_2\ is),\\ trans\ f_1\ f_2\ f_3\ g_1\ g_2\ is &= f_3\ (h_{bs}\ f_1\ f_2\ f_3\ g_1\ g_2\ is). \end{aligned}$$

If we use *Inf* for the property of being an infinite stream of data, $\approx$ for the relation of extensional equality of infinite streams, and *Futc* for the property of being a fair unreliable transmission channel, then the specification of our programming task is

$$Protocol\ f_1\ f_2\ f_3 \quad \equiv \quad \forall g_1, g_2 \in Futc.\forall is \in Inf.is \approx trans\ f_1\ f_2\ f_3\ g_1\ g_2\ is\,.$$

We have thus reduced the protocol problem to a pure functional programming problem.

# 2 The $\mu$-calculus

The pure $\mu$-calculus is classical predicate calculus extended with a least fixed point operator $\mu$ which is used for defining predicates (and relations) by induction.

In addition to the pure calculus we shall introduce syntax for the terms of the logic. These terms are functional programs which are built up by application from certain basic combinators. The proper axioms of the theory are rules of conversion (or recursion equations) for the functional terms.

The following presentation is based on Park

## 2.1 Syntax

Let $P, P'$ range over *formulas*, $X$ over *n-ary relation variables*, $R, R', \Phi$ over *n-ary relation terms*, $x$ over *individual variables*, and $a, a'$ over *individual terms*. Vector notation $\overline{x}$ and $\overline{a}$ is used for sequences of $n$ variables and terms.

$\bot, \top, \neg P, P \vee P', P \wedge P', \exists x.P, t = t', R\,\overline{a}$ are formulas.

$X, \overline{x}.P, \mu X.\Phi$ are $n$-ary relation terms, provided all occurrences of $X$ in $\Phi$ are *positive*. An occurrence is positive if it is within the scope of an even number of $\neg$-signs.

$x, c, (a\,a')$ are terms, provided $c$ is a *combinator*. In this paper we use a collection of combinators tailored to the example of the alternating bit protocol. First, we have the *constructors* $O$ and $L$ for bits, $nil'$ and $::$ (infix cons) for lists, $\langle\ ,\ \rangle$ for pairs, and *err* and *ok* for error values. Then, we have *recursively defined functions*, such as *if*, *eq*, *not*, *hd*, *tl*, $++$ (infix append), and *corrupt* and several others introduced later for programming the alternating bit protocol.

*Remark 1.* We do not introduced $\lambda$-abstraction for terms. This is because the standard formulation of the $\mu$-calculus requires that the syntax of terms is first order.

*Remark 2.* We will only make use of higher order functions in the general discussion. The proof of the alternating bit protocol uses only first order function. Thus for this particular purpose we could use a simpler data domain than a whole model of a functional language. Then there would be one function symbol for each combinator above (and noone for application): for example $O$, $L$, $nil'$, *err* would be nullary, *ok* would be unary, and *cons* and $\langle\ ,\ \rangle$ would be binary.

Ordinary syntactic conventions apply regarding parentheses, infix notation, etc. We introduce implication, equivalence, universal quantification, and bounded quantifiers as abbreviations:

$$\begin{array}{lcl} P \supset P' & \equiv & \neg P \vee P', \\ \forall x.P & \equiv & \neg\exists x.\neg P, \\ P \equiv P' & \equiv & P \supset P' \wedge P' \supset P, \\ \forall x \in R.P & \equiv & \forall x.x \in R \supset P, \\ \exists x \in R.P & \equiv & \exists x.x \in R \wedge P, \end{array}$$

We also introduce set notation for relations

$$\begin{array}{rcl} a \in R & \equiv & R\,a\,, \\ \{x \mid P\} & \equiv & x.P\,, \\ \{a_1, \ldots, a_n\} & \equiv & \{x | x = a_1 \vee \cdots \vee x = a_n\}\,, \\ R \cup R' & \equiv & \{x \mid x \in R \vee x \in R'\}\,, \\ \mathcal{C}R & \equiv & \{x | \neg x \in R\}\,, \\ & \text{etc.}, & \end{array}$$

and last, but not least, the greatest fixed point operator

$$\nu X.\Phi \equiv \mathcal{C}(\mu X.\mathcal{C}(\Phi[\mathcal{C}X/X])).$$

## 2.2 Axioms and inference rules

First, we have axioms and inference rules of classical predicate calculus with equality.

Then, we have two axiom schemes for the $\mu$-operator. The first one states that $\mu X.\Phi$ is a prefixed point of the function which maps $X$ to $\Phi$:

$$\Phi[\mu X.\Phi/X] \subseteq \mu X.\Phi.$$

The second one states that it is the least prefixed point:

$$(\Phi[R/X] \subseteq R) \supset (\mu X.\Phi \subseteq R).$$

We also have an axiom scheme

$$(x.P)\,a \equiv P[a/x].$$

This completes the description of the pure $\mu$-calculus.

From these axioms and rules we can derive two axiom schemes for the $\nu$-operator. The first one states that $\nu X.\Phi$ is a postfixed point of the function which maps $X$ to $\Phi$:

$$\nu X.\Phi \subseteq \Phi[\nu X.\Phi/X].$$

The second one states that it is the greatest postfixed point:

$$(R \subseteq \Phi[R/X]) \supset (R \subseteq \nu X.\Phi).$$

This rule is called the rule of *coinduction* or *greatest fixed point induction* or *Park's rule*.

In addition to the pure $\mu$-calculus we have proper axioms, which are *conversion rules* or *recursion equations* for functional terms. These include

$$\begin{aligned}
\forall d, e.\ \mathit{if}\, O\, d\, e &= d, \\
\forall d, e.\ \mathit{if}\, L\, d\, e &= e, \\
\mathit{not}\, O &= L, \\
\mathit{not}\, L &= O, \\
\forall b.\ \mathit{eq}\, O\, b &= \mathit{not}\, b, \\
\forall b.\ \mathit{eq}\, L\, b &= b, \\
\forall a, as.\ \mathit{hd}\, (a :: as) &= a, \\
\forall a, as.\ \mathit{tl}\, (a :: as) &= as, \\
\forall bs.\ \mathit{nil}' + + bs &= bs, \\
\forall a, as, bs.\ (a :: as) + + bs &= a :: (as + + bs)
\end{aligned}$$

as well as recursion equations for the special combinators needed for programming the alternating bit protocol. These recursion equations are given when these combinators are introduced and explained in section 4 and 5.

### 2.3 Semantics

The semantics of the pure $\mu$-calculus is described in Park [**?**]. It is an extension of the semantics of classical predicate calculus with equality. For an interpretation we need a base set, an assignment of elements of that set to individual variables, and an assignment of $n$-ary relations over that set to $n$-ary relation variables. Then a term denotes an element of the set, a formula denotes a truth value, and an $n$-ary relation term denotes an $n$-ary relation over that set.

We refer the reader to Park [**?**] for details of the interpretation. We just recall the main point, namely that $\mu X.\Phi$ denotes the relation which is the least fixed point (is inductively defined by) the operator which maps a relation $Q$ to the relation denoted by $\Phi$ (where the assignment is extended so that $Q$ is assigned to $X$). The existence of this least fixed point is ensured by the *monotonicity* of this operator, which follows from the syntactic restriction that $X$ may only occur positively in $\Phi$.

Any base set will give an interpretation of the pure $\mu$-calculus. But here we need to choose a suitable base set for interpreting our language of functional terms. We need to interpret the combinators as elements of this set and binary application as a binary function over that set, in such a way that the conversion rules are satisfied.

We obtain this from a model of an untyped functional programming language (an untyped enriched $\lambda$-calculus), which gives us a base set, an interpretation of application and $\lambda$-abstraction, an interpretation of a fixed point combinator $Y$, and an interpretation constructors and selectors for bits, lists, pairs, and error values. The recursively defined functions can then be defined in terms of these constructions.

For example, the model would provide us with an interpretation of the constructors $nil'$ and $::$ for lists and the selector *listcases*, such that the conversion rules

$$\begin{aligned} listcases\ nil'\ d\ e &= d \\ , listcases\ (a :: as)\ d\ e &= e\ a\ as \end{aligned}$$

are satisfied. Infix ++ (append) can then be defined by

$$++ \equiv Y(\lambda f.\lambda as, bs.listcases\ as\ nil'\ (\lambda a, as'.a :: (f\ as\ bs))).$$

Then the conversion rules for ++ can be derived.

Note that we have chosen the conversion rules in such a way that it is clear that the recursively defined functions can be defined in terms of the standard combinators in this manner.

*Remark.* We explained the interpretation of our combinator language in terms of an interpretation of the $\lambda$-calculus, which includes the interpretation of $\lambda$. It would thus be natural to allow $\lambda$ in the formal language as well. Our sole reason for not doing so here is our wish to stay inside the standard framework of the $\mu$-calculus.

The construction of such a set is part of the standard modelling of untyped functional programming languages (enriched untyped $\lambda$-calculi). Such a model will provide an interpretation of the binary application operator as a binary function on the base set and of the constructors and selectors (for bits, lists, pairs, and error values, for example) as elements of the base set. It will also provide an interpretation of the fixed point combinator as an element of the base set. To interpret the recursively defined functions

# 3 The specification of a protocol

In section 2 we arrived at the following form for the specification of a communication protocol:

$$Protocol\ f_1\ f_2\ f_3 \equiv \forall g_1, g_2 \in Futc.\forall is \in Inf.is \approx trans\ f_1\ f_2\ f_3\ g_1\ g_2\ is.$$

The predicates *Inf*, $\approx$, and *Futc* can be defined as greatest fixed points in the following way.

$as \in Inf$ is true provided $as$ is an infinite stream of data. Let $D$ be a unary predicate such that $D\,a$ is true provided $a$ is a data item. (We do not specify it further.) Then we define

$$Inf \equiv \nu X.D :: X,$$

where $::$ is used as a map on unary predicates:

$$as \in (P :: Q) \equiv \exists a, as'.a \in P \wedge as' \in Q \wedge as = a :: as'.$$

$as \approx bs$ is true provided $as$ and $bs$ are extensionally equal infinite streams. We define

$$\approx \equiv \nu X. = :: X,$$

where $=$ is the convertibility relation and $::$ is used as a map on binary relations:

$$as\ (R :: S)\ bs \equiv \exists a, as', b, bs'.a\ R\ b \wedge as'\ S\ bs' \wedge as = a :: as' \wedge bs = b :: bs'.$$

Note the similarity between this definition and the definition of bisimulation equivalence in CCS [].

*Futc* $g$ is true provided $g$ is a fair unreliable transmission channel. We express this formally by introducing an auxiliary function *corrupt* of two arguments: an oracle stream (of bits) and

a stream of messages. The bits in the oracle stream determine whether a message is corrupted or not. We have the following recursion equations:

$$\begin{aligned} \forall os, x, xs.\ corrupt\,(O :: os)\,(x :: xs) &= err :: corrupt\ os\ xs, \\ \forall os, x, xs.\ corrupt\,(L :: os)\,(x :: xs) &= (ok\,x) :: corrupt\ os\ xs. \end{aligned}$$

These channels are non-deterministic in the sense that we do not know in advance when messages are corrupted. By using oracles we isolate this non-determinism, and the fairness of a channel is reduced to the fairness (or more properly, the $L$-fairness) of an oracle:

$$Futc\,g \equiv \exists os \in Fair.g = corrupt\ os,$$

where

$$Fair \equiv \nu X.append\,0^*L\,X.$$

As $::$ before *append* is used as a map on unary predicates here. $0^*L$ is a unary predicate such that $0^*L\,al$ is true provided $al$ is a list of zero or more $O$'s followed by a final $L$. It can be defined either as a least fixed point

$$0^*L \equiv \mu X.\{L :: nil'\} \cup O :: X.$$

# 4 The alternating bit protocol and its correctness proof

We shall now program the alternating bit protocol in our lazy functional language by writing down the recursion equations for some new constants. Our programs differ only marginally from the corresponding programs written in Miranda, in appendix A. The sender is programmed by the two mutually recursive functions *abpsend* and *await*. The receiver is programmed by the two functions *abpack*, which sends acknowledgement bits, and *abpout*, which produces output. The functions satisfy the following recursion equations:

$$\begin{aligned} \forall b, i, is, ds.\ abpsend\,b\,(i :: is)\,ds &= \langle i, b\rangle :: await\,b\,i\,is\,ds, \\ \forall b, b_0, i, is, ds.\ await\,b\,i\,is\,((ok\,b_0) :: ds) &= if\ (eq\,b\,b_0)\,(abpsend\,(not\,b)\,is\,ds)\,(\langle i, b\rangle :: await\,b\,i\,is\,ds), \\ \forall b, i, is, ds.\ await\,b\,i\,is\,(err :: ds) &= \langle i, b\rangle :: await\,b\,i\,is\,ds, \\ \forall b, b_0, i, bs.\ abpack\,b\,((ok\,\langle i, b_0\rangle) :: bs) &= if\ (eq\,b\,b_0)\,(b :: abpack\,(not\,b)\,bs)\,((not\,b) :: abpack\,b\,bs), \\ \forall b, bs.\ abpack\,b\,(err :: bs) &= (not\,b) :: abpack\,b\,bs, \\ \forall b, b_0, i, bs.\ abpout\,b\,((ok\langle i, b_0\rangle) :: bs) &= if\ (eq\,b\,b_0)\,(i :: abpout\,(not\,b)\,bs)\,(abpout\,b\,bs), \\ \forall b, bs.\ abpout\,b\,(err :: bs) &= abpout\,b\,bs. \end{aligned}$$

Formally, the alternating bit protocol gives two implementations - one for each possible start bit $b \in Bit \equiv \{O, L\}$. We shall prove the correctness of both implementations simultaneously, that is, the *theorem*

$$\forall b \in Bit.Protocol\,(abpsend\,b)\,(abpack\,b)\,(abpout\,b).$$

If we unfold the definition of *Protocol* and introduce the auxiliary program

$$\begin{aligned} abptrans\,b\,os0\,os1\,is \equiv\ & trans\,(abpsend\,b)\,(abpack\,b) \\ & (abpout\,b)\,(corrupt\ os0)\,(corrupt\ os1)\,is, \end{aligned}$$

then we can rewrite the correctness theorem as

$$\forall b \in Bit.\forall os0, os1 \in Fair.\forall is \in Inf.is \approx abptrans\, b\, os0\, os1\, is.$$

Since $\approx$ is defined as a greatest fixed point $\nu X. = :: X$, we can prove the theorem by coinduction. In CCS jargon: we prove that input and output are in the largest bisimulation $\approx$ by finding another bisimulation $R$ which they are in. We use the relation $R$ defined by

$$is\, R\, js \equiv is \in Inf \wedge \exists b \in Bit.\exists os0, os1 \in Fair.\exists as, bs, cs, ds.\imath Abp\, b\, is\, os0\, os1\, as\, bs\, cs\, ds\, js,$$

where $\imath Abp$ is an abbreviation for the recursion equations of the alternating bit protocol:

$$\begin{aligned} \imath Abp\, b\, is\, os0\, os1\, as\, bs\, cs\, ds\, js \quad \equiv \quad & as = abpsend\, b\, is\, ds \wedge \\ & bs = corrupt\, os0\, as \wedge \\ & cs = abpack\, b\, bs \wedge \\ & ds = corrupt\, os1\, cs \wedge \\ & js = abpout\, b\, bs. \end{aligned}$$

If we assume that $b \in Bit, os0, os1 \in Fair$ and $is \in Inf$, then the rule of coinduction states that we can prove

$$is \approx abptrans\, b\, os0\, os1\, is$$

from the major premise

$$is\, R\, (abptrans\, b\, os0\, os1\, is)$$

and the minor premise

$$\forall is, js.is\, R\, js \supset is\, (= :: R)\, js.$$

The major premise follows from the fact that *abptrans* satisfies the recursion equations for the alternating bit protocol.

The proof of the minor premise uses two lemmas. The first lemma states that given a start state (of the alternating bit protocol) we will arrive at a state, where the message has been received by the receiver, but where the acknowledgement has not been received by the sender. The second lemma states that given a state of the latter kind we will arrive at a new start state, which is identical to the old start state except that the bit has alternated and the first item in the input stream has been removed.

To formalize this argument we need to introduce a new abbreviation $\imath Abp'$ for the recursion equations for the state where the message has been received by the receiver, but where the acknowledgement has not been received by the sender:

$$\begin{aligned} \imath Abp'\, b\, i\, is'\, os0'\, os1'\, as'\, bs'\, cs'\, ds'\, js' \quad \equiv \quad & ds' = corrupt\, os1'\, cs' \wedge \\ & as' = await\, b\, i\, is'\, ds' \wedge \\ & bs' = corrupt\, os0'\, as' \wedge \\ & cs' = abpack\, b\, bs' \wedge \\ & js' = abpout\, b\, bs'. \end{aligned}$$

(Note that $ds'$ and $os1'$ have been evaluated one time less than the other streams.)

To prove that $R$ is a bisimulation, we assume

$$is\, R\, js.$$

Hence, $is \in Inf$ and by unfolding the definition of $Inf$ we conclude that there are $i \in \mathit{1Data}$ and $is' \in Inf$, such that

$$is = i :: is'.$$

Moreover, there are $b \in Bit$, $os0, os1 \in Fair$, and $as, bs, cs, ds$, such that

$$\mathit{1Abp}\, b\, is\; os0\; os1\; as\; bs\; cs\; ds\; js.$$

Hence, by lemma 1, there are $os0', os1' \in Fair$, and $as', bs', cs', ds', js'$, such that

$$\mathit{1Abp'}\, b\, i\, is'\; os0'\; os1'\; as'\; bs'\; cs'\; ds'\; js'$$

and

$$js = i :: js'.$$

Then, by lemma 2, there are $os0'', os1'' \in Fair$, and $as'', bs'', cs'', ds''$, such that

$$\mathit{1Abp}\, (not b)\; is'\; os0''\; os1''\; as''\; bs''\; cs''\; ds''\; js'.$$

Hence

$$is'\, R\, m\, js'.$$

and

$$is\, (=::R)\, js.$$

It remains to prove the two lemmas.

*Lemma 1:* Assume that $b \in Bit$, $i \in \mathit{1Data}$, $is' \in Inf$, $os0, os1 \in Fair$, and $as, bs, cs, ds$, such that

$$\mathit{1Abp}\, b\, (i :: is')\; os0\; os1\; as\; bs\; cs\; ds\; js.$$

Then there are $os0', os1' \in Fair$, $as', bs', cs', ds', js'$, such that

$$\mathit{1Abp'}\, b\, i\, is'\; os0'\; os1'\; as'\; bs'\; cs'\; ds'\; js'$$

and

$$js = i :: js'.$$

To prove this we note that $os0 \in Fair$, and hence there are $ol_0 \in 0^*L$ and $os0' \in Fair$, such that

$$os0 = ol_0 + + os0'.$$

We proceed by induction on $ol_0 \in 0^*L$.

The *base case* is when $ol_0 = L :: nil'$. We deduce successively:

$$\begin{array}{lll}
as = \langle i, b\rangle :: as', & \text{where} & as' = await\, b\, i\, is'\, ds; \\
bs = (ok\, \langle i, b\rangle) :: bs', & \text{where} & bs' = corrupt\; os0'\; as'; \\
cs = b :: cs', & \text{where} & cs' = abpack\, (not\, b)\; bs'; \\
js = i :: js', & \text{where} & js' = abpout\, (not\, b)\; bs'.
\end{array}$$

Let also $ds' = ds$ and $os1' = os1 \in Fair$. Hence

$$ds' = corrupt\; os1\; cs = corrupt\; os1'\, (b :: bs').$$

We have proved

$$\mathit{1Abp'}\, b\, i\, is'\; os0'\; os1'\; as'\; bs'\; cs'\; ds'\; js'$$

and

$$js = i :: js'.$$

The *induction step* is to go from a proof for ${ol_0}^t$ to a proof for $ol_0 = 0 :: {ol_0}^t$.

We deduce successively:

$$\begin{array}{lll}
as = \langle i, b\rangle :: as^t, & \text{where} & as^t = await\, b\, i\, is'\, ds; \\
bs = err :: bs^t, & \text{where} & bs^t = corrupt\, {os0}^t\, as^t \\
 & & \text{and } {os0}^t = ol_0 +\!+ os0'; \\
cs = (not\, b) :: cs^t, & \text{where} & cs^t = abpack\, b\, bs^t; \\
ds = e :: ds^t, & \text{where} & ds^t = corrupt\, {os1}^t\, cs^t \\
 & & \text{and } e = err \text{ or } e = ok\,(not\, b) \\
 & & \text{and } {os1}^t = tl\, os1.
\end{array}$$

It follows that

$$as^t \;=\; abpsend\, b\, (i :: is')\, ds^t$$

and

$$js \;=\; abpout\, b\, bs^t.$$

Hence

$$_1Abp\, b\, (i :: is')\, {os0}^t\, {os1}^t\, as^t\, bs^t\, cs^t\, ds^t\, js.$$

Since ${os0}^t, {os1}^t \in Fair$, we can use the induction hypothesis to conclude that there are $os0', os1' \in Fair$, and $as', bs', cs', ds', js'$, such that

$$_1Abp'\, b\, i\, is'\, os0'\, os1'\, as'\, bs'\, cs'\, ds'\, js'$$

and

$$js = i :: js'.$$

The proof of lemma 1 is thus complete.

*Lemma 2:* Assume that $b \in Bit$, $i \in {}_1Data$, $is' \in Inf$, $os0', os1' \in Fair$, and $as', bs', cs', ds', js'$, such that

$$_1Abp'\, b\, i\, is'\, os0'\, os1'\, as'\, bs'\, cs'\, ds'\, js'.$$

Then there are $os0'', os1'' \in Fair$, $as'', bs'', cs'', ds''$, such that

$$_1Abp\, (not\, b)\, is''\, os0''\, os1''\, as''\, bs''\, cs''\, ds''\, js'.$$

To prove this we note that $os1' \in Fair$, and hence there are $ol_1 \in 0^*L$ and $os1''$, such that

$$os1' = ol_1 +\!+ os1''.$$

We proceed by induction on $ol_1 \in 0^*L$.

The *base case* is when $ol_1 = L :: nil'$. So $os1' = L :: os1''$. By evaluating we deduce:

$$ds' = (ok\, b) :: ds'', \quad \text{where} \quad ds'' = corrupt\, os1''\, cs'.$$

Let also $as'' = as'$, $bs'' = bs'$, $cs'' = cs'$, and $os0'' = os0'$. Hence

$$\begin{array}{l}
as'' = as' = await\, b\, i\, is'\, ds' = abpsend\, (not\, b)\, is'\, ds'', \\
bs'' = bs' = corrupt\, os0'\, as' = corrupt\, os0''\, as'', \\
cs'' = cs' = abpack\, (not\, b)\, bs' = abpack\, (not\, b)\, bs'', \\
ds'' = corrupt\, os1''\, cs'', \\
js' = abpout\, b\, bs''.
\end{array}$$

We have proved

$$\imath Abp\,(not\,b)\;is'\;os0''\;os1''\;as''\;bs''\;cs''\;ds''\;js'.$$

The *induction step* is to go from a proof for ${ol_1}^t$ to a proof for $ol_1 = 0 :: {ol_1}^t$.
By evaluating we deduce successively:

$$\begin{array}{lll}
ds' = err :: ds^t, & \text{where} & ds^t = corrupt\;{os1}^t\;cs' \\
 & & \text{and } {os1}^t = {ol_1}^t + + os1'', \\
as' = \langle i, b\rangle :: as^t, & \text{where} & as^t = await\,b\,i\,is'\;ds^t; \\
bs' = e :: bs^t, & \text{where} & bs^t = corrupt\;{ol_1}^t\;as^t \\
 & & \text{and} e = ok\langle i, b\rangle \text{ or } e = err \\
 & & \text{and} os0^t = tl\;os0'; \\
cs' = (not\,b) :: cs^t, & \text{where} & cs^t = abpack\,(not\,b)\;bs^t.
\end{array}$$

It follows that

$$ds^t \quad = \quad corrupt\;{os1}^t\,((not\,b) :: cs^t).$$

Hence

$$\imath Abp'\,b\,i\;is'\;os0^t\;{os1}^t\;as^t\;bs^t\;cs^t\;ds^t\;js'.$$

Since $os0^t, {os1}^t \in Fair$, we can use the induction hypothesis to conclude that there are $os0'', os1'' \in Fair$, $as'', bs'', cs'', ds''$, such that

$$\imath Abp\,(not\,b)\;is''\;os0''\;os1''\;as''\;bs''\;cs''\;ds''\;js'.$$

The proof of lemma 2 is thus complete.

Obviously, to get a formal proof we have to fill in still more details. We have checked such a formal proof mechanically in Paulson's Isabelle system [**?**].

# References