CWI

Higher Order Implementation of Kahn Networks in Maude: Alternating Bit Protocol

M. Niqui

SEN-1002

Centrum Wiskunde & Informatica

Centrum Wiskunde & Informatica (CWI) is the national research institute for Mathematics and Computer Science. It is sponsored by the Netherlands Organisation for Scientific Research (NWO).
CWI is a founding member of ERCIM, the European Research Consortium for Informatics and Mathematics.

CWI's research has a theme-oriented structure and is grouped into four clusters. Listed below are the names of the clusters and in parentheses their acronyms.

Probability, Networks and Algorithms (PNA)

Software Engineering (SEN)

Modelling, Analysis and Simulation (MAS)

Information Systems (INS)

Copyright © 2010, Centrum Wiskunde & Informatica
P.O. Box 94079, 1090 GB Amsterdam (NL)
Science Park 123, 1098 XG Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

ISSN 1386-369X

Higher Order Implementation of Kahn Networks in *Maude*: Alternating Bit Protocol

Milad Niqui ⋆

Department of Software Engineering
Centrum Wiskunde & Informatica, The Netherlands
M.Niqui@cwi.nl

Abstract. We implement Kahn networks in *Maude* system by using behavioural theory of streams and encoding higher order function types. As an example we implement the alternating bit protocol in our framework.

Keywords: Kahn network, Maude, alternating bit protocol
MSC Classification: 68Q10
ACM Classification: F.1.2, F.3.2

1 Introduction

Kahn networks [8] provide a basic model for dataflow programming. The idea is to model a (possibly parallel) process involving potentially infinite flow of data as a set of equations that capture the dependency of various elements of the system. Such equational description of a system, due to its inherent simplicity and intuitiveness, has proven to be very useful in practice and has led to several tools in dataflow programming [2]. On the other hand *rewriting logic* provides a framework for efficient reasoning in equational theories. A feat that seems to be suitable in combination with the essentially equational description given by Kahn networks.

Maude system is a tool based on rewriting logic and can be used for specification and verification of the equational properties of various types of systems [3]. In this article we show how to use *Maude* to implement Kahn networks in rewriting logic. First, in Section 2 we sketch a general method for transforming arbitrary Kahn Networks to an equational theory in containing higher order functions on streams. Then, in the remainder of the article we apply this method to two illustrative examples. The first example comes from Kahn's original paper [8]. The second example is the alternating bit protocol, a basic communication protocol underlying many more advanced protocols. Because of simplicity of the framework of Kahn networks our implementation of this protocol turns out to be relatively lightweight. Our treatment is very much influenced by [4], where an implementation of Kahn networks in a functional programming setting is discussed. Although our approach to implementing *fairness* is based on the *hidden* logic of *Maude* and is somehow different than in [4].

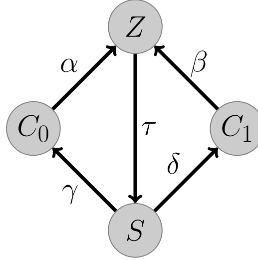
First we briefly introduce the framework of Kahn networks. A *Kahn Network* consists of a *program schema*, i.e., a connected directed graph where the nodes

⋆ Supported by a VENI grant from the Netherlands Organisation for Scientific Research (NWO).

are different *stream transformers* and the connections denote the dependency of the flow of input and output in nodes. The direction on the edges depicts the flow of data. To each edge we assign a stream of data (i.e., an infinite sequence of data) that denotes the flow of data tokens passing through that edge. The relationship between the streams assigned to different edges is governed by a set of *fixed point equations*. Each such equation is assigned to a node together with all of its ingoing and *one* of its ongoing streams. It will contain an stream transformer corresponding with the node, together with the streams assigned to the in and outgoing edges.

There are variants of Kahn networks based on some restrictions on the operation of networks. Here we only mention some assumptions that affect our analysis. First the edges behave like a FIFO buffer, i.e., they have unbounded capacity. In contrast, when a node awaits data on its input it is pended until its read request is fulfilled. Furthermore, the stream transformers in the nodes are assumed to work in a deterministic manner. Following [4], non-determinism can be modelled by incompletely specified networks.

An example of program schema is the following graph [8].



This schema corresponds with a closed Kahn network where there is no input or output. The dependencies given by this graph are listed as the following equations.

$$\begin{aligned} &\tau = Z(\alpha, \beta) \ , && \\ &\alpha = C_0(\gamma) \ , && \beta = C_1(\delta) \ , \\ &\gamma = S_l(\tau) \ , && \delta = S_r(\tau) \ . \end{aligned} \tag{1}$$

Note that since S has two outgoing edges it provides two equations, one for each output with respect to the input τ.

We can obtain a Kahn network by assigning a set of stream transformers to the nodes. Given a set A let A^ω denote the set of streams of elements of A. Given a stream $s \in A^\omega$ we denotes its ith element by $s(i)$, with 0th element denoting the head. By s' we denote the tail of s and by $x\colon s$ we denote the stream whose head is x and whose tail is s. Let $\texttt{zip}\colon A^\omega \times A^\omega \longrightarrow A^\omega$, $\texttt{even}\colon A^\omega \longrightarrow A^\omega$ and $\texttt{odd}\colon A^\omega \longrightarrow A^\omega$ be the functions given by the following Haskell-like specifications.

```
zip (s0,s1) = s0(0) : s1(0) : zip (s0',s1')
```

```
even s = s(0) : even (s'')
odd s = s(1) : odd (s'')
```

Now let $A = \mathbf{2}$, the set of Booleans. Based on this we assign the following stream transformer to each equation in (1).

$$Z(\alpha,\beta) := \texttt{zip}(\alpha,\beta) \ , \tag{2}$$

$$C_0(\gamma) := 0\colon \gamma \ , \qquad C_1(\gamma) := 1\colon \gamma \ , \tag{3}$$

$$S_l(\tau) := \texttt{even}(\tau) \ , \qquad S_r(\tau) := \texttt{odd}(\tau) \ . \tag{4}$$

Hence the behaviour of the Kahn network will be fully captured by the following systems of equations.

$$\begin{aligned} \tau &= \texttt{zip}(\alpha,\beta) \ , \\ \alpha &:= 0\colon \gamma \ , \\ \beta &:= 1\colon \gamma \ , \\ \gamma &= \texttt{even}(\tau) \ , \\ \delta &= \texttt{odd}(\tau) \ . \end{aligned}$$

We implement this 'representation' of Kahn networks in *Maude*. The motivation is that once we have an implementation we are able to test various properties of the network. For example, the above system of equations has a solution, namely,

$$\begin{aligned} \tau &= 0\colon 1\colon \tau && [\text{the stream } \overline{01}\ ] \ , \\ \alpha &:= 0\colon \gamma && [\text{the constant stream } \overline{0}\ ] \ , \\ \beta &:= 1\colon \gamma && [\text{the constant stream } \overline{1}\ ] \ , \\ \gamma &= 0\colon \gamma && [\text{the constant stream } \overline{0}\ ] \ , \\ \delta &= 1\colon \delta && [\text{the constant stream } \overline{1}\ ] \ . \end{aligned} \tag{5}$$

Using our implementation and the reduction mechanism of *Maude* we can observe that indeed the arbitrary portions of the streams satisfying (1) are equal to their counterpart in (5).

For translating this Kahn network to *Maude*, first we implement the equational theory of (1). This step involves higher order functions and an abstract (data-independent) theory of streams. Subsequently, we will implement the transformer at each node (e.g. the functions `zip`, `even` and `odd`) which are ordinary (first order) functions acting on streams. In the next section we will explain this process for a generic program schema.

2 Program Schemata and Higher Order Types

Maude can be easily used for encoding λ-terms and function application [3, § 8]. In our development we would not need a full embedding of λ-terms; the only

thing we need from that theory is a function type constructor together with the function application.

```
(fmod FUNC{X :: TRIV, Y :: TRIV} is
  sort Func{X, Y} .
  op _‘[_‘] : FuncX, Y X$Elt -> Y$Elt [strat (0)] .
endfm)
```

Here we are defining a module of functions between X and Y, introducing the type `Func{X,Y}` for the function space. Further, we introduce the application operation, here denoted by [_], as the sole way of building the elements of the function type. The important thing is that we assume our function application is *lazy*, i.e., it is performed before reducing either of the arguments (note that application is a binary operation). This is enforced by the `strat` keyword that specifies the reduction strategy. We will use the lazy strategy for most functions later on in this work.

Next we need a module for streams. This consists of a (parametric) type with two observers for head and tail.

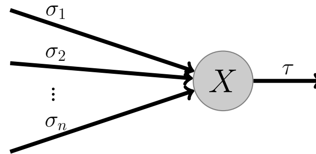
```
(fmod STREAM{X :: TRIV} is
  sort Stream{X} .
  op hd : Stream{X} -> X$Elt [memo] .
  op tl : Stream{X} -> Stream{X} [memo] .
endfm)
```

This is a behavioural theory of streams (also called hidden theory [7]) in that the observers `hd` and `tl` are further unspecified. This is similar to the treatment of streams in the theory of coalgebras [14].

Apart from these we will need some standard *Maude* modules (such as `Bool`). Furthermore we need to define the appropriate *views* for using parametric types. As these are straightforward *Maude* constructs we refrain from explicitly mentioning those in this article but they can be found in [11].

Implementing the equational theory for a Kahn network has a generic part, merely dependent on the shape of the program schema, and a part specific to the semantics of each node. There is a generic method for implementing the first part explained in [4], which we sketch below. We remark that in [4] higher order functions of a functional programming language (e.g. Haskell) are readily usable, while in our adaptation the stream transformer functions in the nodes should be represented as elements of the functions type above.

Consider a node X with n incoming edges and one outgoing edge.



The point is to express that τ is a function of $\sigma_1 \dots \sigma_n$. Hence this configuration would lead to the equation

$$\tau = X(\sigma_1, \dots, \sigma_n) \ .$$

This indicates that τ depends on $\sigma_1, \dots, \sigma_n$. Since this procedure can be repeated for each σ_i, one observes that τ depends on all the streams appearing on the edges of the program schema. Assume there are in total k edges in the program schema.We introduce a functional F_τ that takes k arguments and outputs the stream τ. The k arguments are either stream transformers (hence functions themselves) or in the case of external input nodes they are simply streams (which can be considered as nullary constant functions). This means

$$F_\tau(\bar{X}_1, \dots, \bar{X}_k) = \bar{X}(F_{\sigma_1}, \dots, F_{\sigma_n})$$

where $\bar{X}_i$ is the representation of the stream transformer X_i as an element of the function space. I.e., for instance since $X \colon \underbrace{A^\omega \times \cdots \times A^\omega}_{n} \longrightarrow A^\omega$ we will have

$$\bar{X} : \texttt{Func\{}\underbrace{A^\omega \times \cdots \times A^\omega}_{n},\ A^\omega\texttt{\}} \ .$$

To decrease the clutter, from now on we ignore the over-line and identify $f \colon A \longrightarrow B$ with its representation $\bar{f} \colon$ `Func{A,B}`. Repeating this process for all edges we obtain a set of mutually recursive equations.

3 Kahn Network for Zip

In this section we show the implementation of the Kahn network that was presented in Section 1. In fact we implement the system of equations (1) in *Maude*. First we follow the procedure at the end of Section 2 to implement the program schema. This results in the following system of equations.

$$\begin{aligned}
F_\tau(Z, S_l, S_r, C_0, C_1) &= Z(F_\alpha(Z, S_l, S_r, C_0, C_1), F_\beta(Z, S_l, S_r, C_0, C_1)) \ , \\
F_\alpha(Z, S_l, S_r, C_0, C_1) &= C_0(F_\gamma(Z, S_l, S_r, C_0, C_1)) \ , \\
F_\beta(Z, S_l, S_r, C_0, C_1) &= C_1(F_\delta(Z, S_l, S_r, C_0, C_1)) \ , \\
F_\gamma(Z, S_l, S_r, C_0, C_1) &= S_l(F_\tau(Z, S_l, S_r, C_0, C_1)) \ , \\
F_\delta(Z, S_l, S_r, C_0, C_1) &= S_r(F_\tau(Z, S_l, S_r, C_0, C_1)) \ .
\end{aligned}$$

Note that even though $Z \colon A^\omega \times A^\omega \longrightarrow A^\omega$, in *Maude* we need to pass to F_i the representation of Z as an element of $\texttt{Func\{}A^\omega \times A^\omega,\ A^\omega\texttt{\}}$. Same holds for other arguments. This system can be implemented straightforwardly in *Maude* (see Appendix A.1.)

Now assume we model the ports using the functions in (3). Further assume that we are interested, as in [8], in observing the stream τ. First of all we implement the stream transformers in the parametric stream module `Stream{X}`

(Appendix A.2). Then we have to give the representation of the functions in (1) as function types. The idea is to define the equational behaviour of function application. For instance for function `zip` we have

```
var BS0 BS1 : Stream{Bool} .

op zipbar : -> Func{Tuple{Stream{Bool},Stream{Bool}},Stream{Bool}}
            [strat (0) memo] .
eq zipbar[ ( BS0 , BS1 ) ] = zip(BS0,BS1) .
```

The definition for other functions is given in A.3.

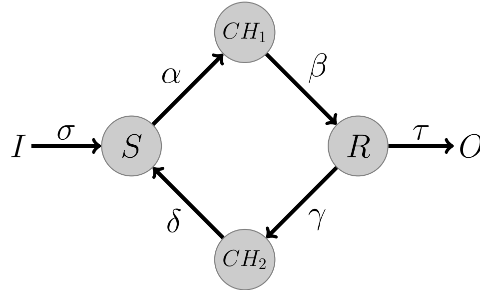
Finally we have to link the program schema with the specific transformers. The following function produces the stream τ.

```
op ziptrans : -> Stream{Bool} [strat (0)] .
eq ziptrans = functau(zipbar, evenbar, oddbar, cons0bar, cons1bar) .
```

4 Alternating Bit Protocol

In this section we present the *Maude* implementation of a more complex Kahn network using the method of the preceding sections. The network corresponds to the alternating bit protocol (ABP). This is a well-known protocol that underlies the principle used in many communication protocols, including the TCP/IP protocol. Yet, due to its simplicity it has been used as a benchmark for several formal verification paradigms and has been implemented and verified in many systems. Among these, related to our work are [13, 5, 1] using the calculus of broadcasting systems (CBS) and process algebra and the implementation in the timed rewriting logic [15]. But our implementation will follow [4].

The program schema for this protocol is depicted below.



This a Kahn network with one input σ and one output τ. There are four modes represent a sender S, a receiver R and two intermediate communication channels CH_1 and CH_2. The sender reads data from the input and tries sending it through the channel CH_1 to the receiver located at R. The receiver in turn sends back acknowledgement bits through CH_2. Both intermediate channels CH_1 and CH_2 may introduce errors in the data. The protocol is designed to ensure the correct

transmission of data through these lossy channels under a *fairness* assumption, namely, that CH_1 and CH_2 eventually pass the correct data. The details of ABP, i.e., how the stream transformers should act, can be found e.g. in [10]; here we only present the implementation in *Maude* and the way in which we tackle the fairness assumption.

First we implement the program schema. The dependencies in the program schema above lead to the following set of equations.

$$\begin{aligned}
&\tau = R_r(\beta), \\
&\gamma = R_l(\beta)\ , \qquad \delta = CH_2(\gamma)\ , \\
&\alpha = S(\sigma,\delta)\ , \qquad \beta = CH_1(\alpha)\ , \\
&\sigma = I()\ .
\end{aligned} \tag{6}$$

Note that I (the external input) is assumed to be a nullary node. Following the procedure in Section 2 we arrive at the set of equations below.

$$\begin{aligned}
&F_\tau(S,R_l,R_r,CH_1,CH_2,I) = R_r(F_\beta(S,R_l,R_r,CH_1,CH_2,I)\ , \\
&F_\gamma(S,R_l,R_r,CH_1,CH_2,I) = R_l(F_\beta(S,R_l,R_r,CH_1,CH_2,I)\ , \\
&F_\delta(S,R_l,R_r,CH_1,CH_2,I) = CH_2(F_\gamma(S,R_l,R_r,CH_1,CH_2,I)\ , \\
&F_\alpha(S,R_l,R_r,CH_1,CH_2,I) = S\big(F_\sigma(S,R_l,R_r,CH_1,CH_2,I), F_\delta(S,R_l,R_r,CH_1,CH_2,I)\big)\ , \\
&F_\beta(S,R_l,R_r,CH_1,CH_2,I) = CH_1(F_\alpha(S,R_l,R_r,CH_1,CH_2,I)\ , \\
&F_\sigma(S,R_l,R_r,CH_1,CH_2,I) = I()\ .
\end{aligned}$$

Conceptually I is an element of `Func{1, A^ω}` which is equivalent to the constant function σ. But for the ease of implementation we replace both I and $I()$ by σ in the *Maude* code. This code can be found in the Appendix B.1.

Implementing the stream transformers follows the alternating bit protocol. There will be functions `abpsend`, `abpack`, `abpout`, `corruptCH1` and `corruptCH2` respectively modelling the transformers S, R_l, R_r, CH_1 and CH_2 in (6). The first three, given in Appendix B.2, are defined following the Haskell-like specifications in [4], while the latter two deal with fairness in a different way. Namely, we do not define any *coinductive predicate* (a recursive infinitary predicate [12]) to capture fairness. Although this is in principle possible but there is no easy way to model that in *Maude*. In the future work we plan to study the general approach of implementing coinductive predicates in *Maude*.

Assume that the lossy behaviour of each of these two channels is given by an oracle stream of Booleans, with 0 denoting the loss of data and 1 denoting the faithful passing of data. For ABP such an oracle stream is fair if it contains infinitely many 1s; and this 'infinite occurrence' is a coinductive predicate. But if we assume that the oracle streams are streams of *natural* numbers then we can bypass this coinductive predicate. Let $\sigma \in \mathbb{N}^\omega$, and let `unpack`: $\mathbb{N}^\omega \longrightarrow \mathbf{2}^\omega$ be the function such that σ is the unary run-length compression of `unpack`(σ). That means `unpack` starts by consuming σ and outputting σ_0 consecutive 0's followed by a 1 (if $\sigma_0 = 0$ then it immediately outputs 1) and continues with the remainder of σ in the same fashion (the precise *Maude* specification can be

found in Appendix B.4). Then it is clear that unpack(σ) is fair. The prove of infinite occurrence which is given by a greatest fixed point in [4] is inherent in the infinitude of σ. So to assume that CH_1 and CH_2 are fair, we assume that their lossy behaviour (whether or not to lose data) is dictated by two oracle streams which are streams of natural numbers. Subsequently in the specification for corruptCH1 and corruptCH2 these oracle streams are unpacked to the Boolean behaviour (see Appendix B.4).

To complete the implementation we have to link the program schema with the specific transformers. The following function produces the stream τ.

```
vars B : Bool .
var OS1 OS2 : Stream{Nat} .

op abptrans : Bool Stream{Nat} Stream{Nat} Stream{Nat}  -> Stream{Nat} .
eq abptrans(B,OS1,OS2,Isigma) =
   ftau(abpsendbar(B),abpackbar(B),abpoutbar(B),
        corruptCH1bar(unpack(OS1)),corruptCH2bar(unpack(OS2)),
        Isigma) .
```

This completes the implementation of ABP as an equational theory in *Maude*. The complete *Maude* code is available for download at [11].

5 Future Work: Verification using CIRC

The motivation for our work is to apply tools based on rewriting logic [3] and hidden algebra [7] to *verify* concurrent systems and dataflow programs that are modelled as Kahn networks. In particular we are interested in behavioural properties of equational theories of streams, as streams are the main ingredients in Kahn networks. For example, for the network in Section 3 we would like to be able to prove:

$$\texttt{ziptrans} = \texttt{zos} \ , \tag{7}$$

where zos is the stream of alternating 0 and 1's, i.e., $\texttt{zos} = 0 : 1 : \texttt{zos}$.

Similarly for the ABP we would like to prove that for any oracle streams $o_1, o_2 \in \mathbb{N}^\omega$, any initial bit b and input stream $\sigma \in A^\omega$

$$\texttt{abptrans}(b, o_1, o_2, \sigma) = \sigma \ , \tag{8}$$

i.e., the Kahn network for ABP acts as a buffer.

The natural way of reasoning about equalities between streams is by using *coinduction proof principle* and finding bisimulations [14]. Coinduction is studied in the context of hidden algebra and rewriting logic [6] and has culminated in the CIRC proof tool. CIRC [9] is a tool for automatically proving behavioural equalities coinduction and induction and is built on top of *Maude*.

Our implementation of Kahn networks paves the way for applying CIRC for the automatic generation of their correctness proofs. Although the current

version of CIRC's decision procedure is unable to prove (7)–(8), this inability is not essential in that it seems not to be a theoretical expressiveness issue and there is evidence that it can be fixed. We are currently working with the developers of CIRC to find a suitable solution for this.

Another direction for future work would be to present a systematic way of formalising coinductive predicate (such as the 'infinite occurrence' in fairness) in *Maude* and CIRC. In a parallel project [12] we are pursuing this track.

Acknowledgements. The author wishes to thank Lacramioara Astefanoaei and Dorel Lucanu for helpful comments on *Maude* implementation.

References

[1] M. Bezem, R. N. Bol, and J. F. Groote. Formalizing process algebraic verifications in the calculus of constructions. *Formal Asp. Comput.*, 9(1):1–48, 1997.

[2] P. Caspi and M. Pouzet. Synchronous kahn networks. In *ICFP '96: Proceedings of the first ACM SIGPLAN international conference on Functional programming*, pages 226–238, New York, NY, USA, 1996. ACM.

[3] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *LNCS*. Springer, 2007.

[4] P. Dybjer and H. P. Sander. A functional programming approach to the specification and verification of concurrent systems. *Formal Asp. Comput.*, 1(4):303–319, 1989.

[5] E. Giménez. An application of co-inductive types in coq: Verification of the alternating bit protocol. In S. Berardi and M. Coppo, editors, *Proc. of TYPES'95*, volume 1158 of *LNCS*, pages 135–152. Springer, 1996.

[6] J. Goguen, K. Lin, and G. Roşu. Circular coinductive rewriting. In *Proc. of ASE'00*, pages 123–132. IEEE, 2000.

[7] J. A. Goguen and G. Malcolm. A hidden agenda. *Theoret. Comput. Sci.*, 245(1):55–101, 2000.

[8] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing 74: Proceedings of IFIP Congress 74, Stockholm*, pages 471–475, Amsterdam, The Netherlands, Aug. 1974. North Holland Publishing Co.

[9] D. Lucanu and G. Roşu. CIRC : A circular coinductive prover. In T. Mossakowski, U. Montanari, and M. Haveraaen, editors, *Proc. of CALCO 2007*, volume 4624 of *LNCS*, pages 372–378. Springer, 2007.

[10] R. Milner. *Communication and concurrency.* Prentice Hall, Upper Saddle River, NJ, USA, 1989.

[11] M. Niqui. `http://www.cwi.nl/~milad/programs/maude/` [cited 18 Jan. 2010], Nov. 2009. Files for Maude v. 2.4.

[12] M. Niqui and J. Rutten. Coinductive predicates as final coalgebras. In R. Matthes and T. Uustalu, editors, *Proc. of FICS 2009*, pages 79–85. Institute of Cybernetics at Tallinn University of Technology, 2009.

[13] K. V. S. Prasad. A calculus of broadcasting systems. *Sci. Comput. Program.*, 25(2–3):285–327, 1995.

[14] J. J. M. M. Rutten. Universal coalgebra: a theory of systems. *Theoret. Comput. Sci.*, 249(1):3–80, Oct. 2000.

[15] L. J. Steggles and P. Kosiuczenko. A timed rewriting logic semantics for sdl: A case study of alternating bit protocol. 15:83–104, 1998.

A Maude specification for Zip network

A.1 Program Schema

```
var Z : Func{Tuple{Stream{Bool},Stream{Bool}},Stream{Bool}} .
vars Sl Sr C0 C1 : Func{Stream{Bool},Stream{Bool}} .

op ftau : Func{Tuple{Stream{Bool},Stream{Bool}},Stream{Bool}}
          Func{Stream{Bool}, Stream{Bool}}
          Func{Stream{Bool}, Stream{Bool}}
          Func{Stream{Bool}, Stream{Bool}}
          Func{Stream{Bool}, Stream{Bool}} -> Stream{Bool} [strat (0) memo] .
eq ftau(Z,Sl,Sr,C0,C1) = Z [ ( falpha(Z,Sl,Sr,C0,C1) , fbeta(Z,Sl,Sr,C0,C1) ) ] .

op falpha : Func{Tuple{Stream{Bool},Stream{Bool}},Stream{Bool}}
            Func{Stream{Bool}, Stream{Bool}}
            Func{Stream{Bool}, Stream{Bool}}
            Func{Stream{Bool}, Stream{Bool}}
            Func{Stream{Bool}, Stream{Bool}} -> Stream{Bool} [strat (0) memo] .
eq falpha(Z,Sl,Sr,C0,C1) = C0 [ fgamma(Z,Sl,Sr,C0,C1) ] .

op fbeta : Func{Tuple{Stream{Bool},Stream{Bool}},Stream{Bool}}
           Func{Stream{Bool}, Stream{Bool}}
           Func{Stream{Bool}, Stream{Bool}}
           Func{Stream{Bool}, Stream{Bool}}
           Func{Stream{Bool}, Stream{Bool}} -> Stream{Bool} [strat (0) memo] .
eq fbeta(Z,Sl,Sr,C0,C1) = C1 [ fdelta(Z,Sl,Sr,C0,C1) ] .

op fgamma : Func{Tuple{Stream{Bool},Stream{Bool}},Stream{Bool}}
            Func{Stream{Bool}, Stream{Bool}}
            Func{Stream{Bool}, Stream{Bool}}
            Func{Stream{Bool}, Stream{Bool}}
            Func{Stream{Bool}, Stream{Bool}} -> Stream{Bool} [strat (0) memo] .
eq fgamma(Z,Sl,Sr,C0,C1) = Sl [ ftau(Z,Sl,Sr,C0,C1) ] .

op fdelta : Func{Tuple{Stream{Bool},Stream{Bool}},Stream{Bool}}
            Func{Stream{Bool}, Stream{Bool}}
            Func{Stream{Bool}, Stream{Bool}}
            Func{Stream{Bool}, Stream{Bool}}
            Func{Stream{Bool}, Stream{Bool}} -> Stream{Bool} [strat (0) memo] .
eq fdelta(Z,Sl,Sr,C0,C1) = Sr [ ftau(Z,Sl,Sr,C0,C1) ] .
```

A.2 Stream Transformers

```
var S1 S2 : Stream{X} .
var E : X$Elt .

op cons : X$Elt Stream{X} -> Stream{X} [strat (0) memo] .
eq hd(cons(E, S1)) = E .
eq tl(cons(E, S1)) = S1 .

op zip : Stream{X} Stream{X} -> Stream{X} [strat (0) memo] .
eq hd(zip(S1, S2)) = hd(S1) .
eq tl(zip(S1, S2)) = zip(S2, tl(S1)) .

op even : Stream{X} -> Stream{X} .
eq hd(even(S1)) = hd(S1) .
eq tl(even(S1)) = even(tl(tl(S1))) .

op odd : Stream{X} -> Stream{X} .
eq hd(odd(S1)) = hd(tl(S1)) .
eq tl(odd(S1)) = odd(tl(tl(S1))) .
```

A.3 Stream Transformers as Function Types

```
var BS0 BS1 : Stream{Bool} .

op zipbar : -> Func{Tuple{Stream{Bool},Stream{Bool}}, Stream{Bool}} [strat (0) memo] .
eq zipbar[ ( BS0 , BS1 )  ] = zip(BS0,BS1) .

op evenbar : -> Func{Stream{Bool}, Stream{Bool}} [strat (0) memo] .
eq evenbar[ BS0 ] = even(BS0) .

op oddbar : -> Func{Stream{Bool}, Stream{Bool}} [strat (0) memo] .
eq oddbar[ BS0 ] = odd(BS0) .

op cons0bar : -> Func{Stream{Bool}, Stream{Bool}} [strat (0) memo] .
eq cons0bar[ BS0 ] = cons(false,BS0) .

op cons1bar : -> Func{Stream{Bool}, Stream{Bool}} [strat (0) memo] .
eq cons1bar[ BS0 ] = cons(true,BS0) .
```

B Maude specification for ABP

B.1 Program Schema

```
var S : Func{Tuple{Stream{Nat},Stream{Maybe{Bool}}},Stream{Tuple{Nat,Bool}}} .
var Rl : Func{Stream{Maybe{Tuple{Nat,Bool}}},Stream{Bool}} .
var Rr : Func{Stream{Maybe{Tuple{Nat,Bool}}},Stream{Nat}} .
var CH1 : Func{Stream{Tuple{Nat,Bool}},Stream{Maybe{Tuple{Nat,Bool}}}} .
var CH2 : Func{Stream{Bool},Stream{Maybe{Bool}}} .
```

```
var Isigma : Stream{Nat} .

op falpha : Func{Tuple{Stream{Nat},Stream{Maybe{Bool}}},Stream{Tuple{Nat,Bool}}}
            Func{Stream{Maybe{Tuple{Nat,Bool}}},Stream{Bool}}
            Func{Stream{Maybe{Tuple{Nat,Bool}}},Stream{Nat}}
            Func{Stream{Tuple{Nat,Bool}},Stream{Maybe{Tuple{Nat,Bool}}}}
            Func{Stream{Bool},Stream{Maybe{Bool}}}
            Stream{Nat} -> Stream{Tuple{Nat,Bool}} [strat (0) memo] .
eq falpha(S,Rl,Rr,CH1,CH2,Isigma) = S[ ( Isigma , fdelta(S,Rl,Rr,CH1,CH2,Isigma) ) ] .


op fbeta : Func{Tuple{Stream{Nat},Stream{Maybe{Bool}}},Stream{Tuple{Nat,Bool}}}
           Func{Stream{Maybe{Tuple{Nat,Bool}}},Stream{Bool}}
           Func{Stream{Maybe{Tuple{Nat,Bool}}},Stream{Nat}}
           Func{Stream{Tuple{Nat,Bool}},Stream{Maybe{Tuple{Nat,Bool}}}}
           Func{Stream{Bool},Stream{Maybe{Bool}}}
           Stream{Nat} -> Stream{Maybe{Tuple{Nat,Bool}}} [strat (0) memo] .
eq fbeta(S,Rl,Rr,CH1,CH2,Isigma) = CH1[ falpha(S,Rl,Rr,CH1,CH2,Isigma) ] .

op fgamma : Func{Tuple{Stream{Nat},Stream{Maybe{Bool}}},Stream{Tuple{Nat,Bool}}}
            Func{Stream{Maybe{Tuple{Nat,Bool}}},Stream{Bool}}
            Func{Stream{Maybe{Tuple{Nat,Bool}}},Stream{Nat}}
            Func{Stream{Tuple{Nat,Bool}},Stream{Maybe{Tuple{Nat,Bool}}}}
            Func{Stream{Bool},Stream{Maybe{Bool}}}
            Stream{Nat} -> Stream{Bool} [strat (0) memo] .
eq fgamma(S,Rl,Rr,CH1,CH2,Isigma) = Rl[ fbeta(S,Rl,Rr,CH1,CH2,Isigma) ] .


op fdelta : Func{Tuple{Stream{Nat},Stream{Maybe{Bool}}},Stream{Tuple{Nat,Bool}}}
            Func{Stream{Maybe{Tuple{Nat,Bool}}},Stream{Bool}}
            Func{Stream{Maybe{Tuple{Nat,Bool}}},Stream{Nat}}
            Func{Stream{Tuple{Nat,Bool}},Stream{Maybe{Tuple{Nat,Bool}}}}
            Func{Stream{Bool},Stream{Maybe{Bool}}}
            Stream{Nat} -> Stream{Maybe{Bool}} [strat (0) memo] .
eq fdelta(S,Rl,Rr,CH1,CH2,Isigma) = CH2[ fgamma(S,Rl,Rr,CH1,CH2,Isigma) ] .

op ftau : Func{Tuple{Stream{Nat},Stream{Maybe{Bool}}},Stream{Tuple{Nat,Bool}}}
          Func{Stream{Maybe{Tuple{Nat,Bool}}},Stream{Bool}}
          Func{Stream{Maybe{Tuple{Nat,Bool}}},Stream{Nat}}
          Func{Stream{Tuple{Nat,Bool}},Stream{Maybe{Tuple{Nat,Bool}}}}
          Func{Stream{Bool},Stream{Maybe{Bool}}}
          Stream{Nat} -> Stream{Nat} [strat (0)] .
eq ftau(S,Rl,Rr,CH1,CH2,Isigma) = Rr[ fbeta(S,Rl,Rr,CH1,CH2,Isigma) ] .
```

B.2 Stream Transformers

```
var B : Bool .
var N : Nat .
var NS : Stream{Nat} .
var MBS : Stream{Maybe{Bool}} .
```

```
var MNBS : Stream{Maybe{Tuple{Nat,Bool}}} .

op abpsend : Bool Stream{Nat} Stream{Maybe{Bool}} ->
             Stream{Tuple{Nat,Bool}} [strat (0) memo] .
eq hd(abpsend(B, NS, MBS)) = ( hd(NS) , B ) .
eq tl(abpsend(B, NS, MBS)) = await(B, hd(NS) , tl(NS) , MBS)

op await : Bool Nat Stream{Nat} Stream{Maybe{Bool}} ->
           Stream{Tuple{Nat,Bool}} [strat (0) memo] .
ceq hd(await(B, N, NS, MBS)) = ( N , B ) if hd(MBS) = maybe .
ceq tl(await(B, N, NS, MBS)) = await(B, N, NS, tl(MBS)) if hd(MBS) = maybe .
ceq hd(await(B, N, NS, MBS)) = hd(abpsend(not B, NS, tl(MBS))) if hd(MBS) = B .
ceq tl(await(B, N, NS, MBS)) = tl(abpsend(not B, NS, tl(MBS))) if hd(MBS) = B .
ceq hd(await(B, N, NS, MBS)) = ( N , B ) if hd(MBS) =/= B .
ceq tl(await(B, N, NS, MBS)) = await(B, N, NS, tl(MBS)) if hd(MBS) =/= B .

op abpack : Bool Stream{Maybe{Tuple{Nat,Bool}}} -> Stream{Bool} [strat (0) memo] .
ceq hd(abpack(B , MNBS)) = not B if hd(MNBS) = maybe .
ceq tl(abpack(B , MNBS)) = abpack(B , tl(MNBS)) if hd(MNBS) = maybe .
ceq hd(abpack(B , MNBS)) = B if p2 hd(MNBS) = B .
ceq tl(abpack(B , MNBS)) = abpack(not B , tl(MNBS)) if p2 hd(MNBS) = B .
ceq hd(abpack(B , MNBS)) = not B if p2 hd(MNBS) =/= B .
ceq tl(abpack(B , MNBS)) = abpack(B , tl(MNBS)) if p2 hd(MNBS) =/= B .

op abpout : Bool Stream{Maybe{Tuple{Nat,Bool}}} -> Stream{Nat} [strat (0) memo] .
ceq hd(abpout(B , MNBS)) = hd(abpout(B , tl(MNBS))) if hd(MNBS) = maybe .
ceq tl(abpout(B , MNBS)) = tl(abpout(B , tl(MNBS))) if hd(MNBS) = maybe .
ceq hd(abpout(B , MNBS)) = p1 hd(MNBS) if p2 hd(MNBS) = B .
ceq tl(abpout(B , MNBS)) = abpout(not B , tl(MNBS)) if p2 hd(MNBS) = B .
ceq hd(abpout(B , MNBS)) = hd(abpout(B , tl(MNBS))) if p2 hd(MNBS) =/= B .
ceq tl(abpout(B , MNBS)) = tl(abpout(B , tl(MNBS))) if p2 hd(MNBS) =/= B .
```

B.3 Stream Transformers as Function Types

```
op abpsendbar  : Bool ->
                 Func{Tuple{Stream{Nat},Stream{Maybe{Bool}}},Stream{Tuple{Nat,Bool}}}
                 [strat (0) memo] .
eq (abpsendbar(B))[ ( NS , MBS ) ] = abpsend(B, NS, MBS) .

op abpackbar  : Bool -> Func{Stream{Maybe{Tuple{Nat,Bool}}},Stream{Bool}} [strat (0) memo] .
eq (abpackbar(B))[ MNBS ] = abpack(B, MNBS) .

op abpoutbar  : Bool -> Func{Stream{Maybe{Tuple{Nat,Bool}}},Stream{Nat}} [strat (0) memo] .
eq (abpoutbar(B))[ MNBS ] = abpout(B, MNBS) .
```

B.4 Transformers for Fairness Oracles

```
var OS BS : Stream{Bool} .
var NBS : Stream{Tuple{Nat,Bool}} .
```

```
var NS : Stream{Nat} .

op decrhd : StreamNat -> StreamNat .
eq hd(decrhd(NS)) = sd(hd(NS),1) .
eq tl(decrhd(NS)) = tl (NS) .

op unpack : Stream{Nat} -> Stream{Bool} .
ceq hd(unpack(NS)) = false if (hd(NS)) > 0 .
ceq hd(unpack(NS)) = true if (hd(NS)) = 0 .
ceq tl(unpack(NS)) = unpack(decrhd(NS)) if (hd(NS)) > 0 .
ceq tl(unpack(NS)) = unpack(tl(NS)) if (hd(NS)) = 0 .

op corruptCH1 : Stream{Bool} Stream{Tuple{Nat,Bool}} -> Stream{Maybe{Tuple{Nat,Bool}}} .
ceq hd(corruptCH1(OS, NBS)) = hd(NBS) if  hd(OS) = true .
ceq hd(corruptCH1(OS, NBS)) = maybe if  hd(OS) = false .
eq tl(corruptCH1(OS, NBS)) = corruptCH1(tl(OS), tl(NBS)) .

op corruptCH2 : Stream{Bool} Stream{Bool} -> Stream{Maybe{Bool}} .
ceq hd(corruptCH2(OS, BS)) = hd(BS) if  hd(OS) = true .
ceq hd(corruptCH2(OS, BS)) = maybe if  hd(OS) = false .
eq tl(corruptCH2(OS, BS)) = corruptCH2(tl(OS), tl(BS)) .

op corruptCH1bar  : Stream{Bool} ->
                    Func{Stream{Tuple{Nat,Bool}},Stream{Maybe{Tuple{Nat,Bool}}}}
                    [strat (0) memo] .
eq (corruptCH1bar(OS))[ NBS ] = corruptCH1(OS, NBS) .

op corruptCH2bar  : Stream{Bool} -> Func{Stream{Bool},Stream{Maybe{Bool}}} [strat (0) memo] .
eq (corruptCH2bar(OS))[ BS ] = corruptCH2(OS, BS) .
```

Centrum Wiskunde & Informatica (CWI) is the national research institute for mathematics and computer science in the Netherlands. The institute's strategy is to concentrate research on four broad, societally relevant themes: earth and life sciences, the data explosion, societal logistics and software as service.

Centrum Wiskunde & Informatica (CWI) is het nationale onderzoeksinstituut op het gebied van wiskunde en informatica. De strategie van het instituut concentreert zich op vier maatschappelijk relevante onderzoeksthema's: aard- en levenswetenschappen, de data-explosie, maatschappelijke logistiek en software als service.

Bezoekadres:
Science Park 123
Amsterdam

Postadres:
Postbus 94079, 1090 GB Amsterdam
Telefoon 020 592 93 33
Fax 020 592 41 99
info@cwi.nl
www.cwi.nl

Centrum Wiskunde & Informatica