

Fixing the semantic

Hampus Weddig

Abstract

This paper examines ways to alter the semantic meaning of recursively defined functions. The idea has its roots in model theoretic semantics and evaluation methods found in logic based languages. The result fits naturally into the purely functional paradigm in the form of fixed point combinators, and can be smoothly embedded into the functional language Haskell. The benefits are the same as in the logic paradigm, improved declarative semantic and termination. Some motivating examples of its applications are given as well.

1 INTRODUCTION

Some examples will be used to give a quick introduction to the main idea. The reader is assumed to be familiar with the functional language Haskell.

1.1 Example

Consider the graph described by figure 1. In Haskell the edges of the graph might be encoded by the following non deterministic function.

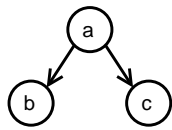


FIGURE 1. acyclic graph

```
edge "a" = return "b" `mplus` return "c"  
edge "b" = mzero  
edge "c" = mzero
```

Given the above function `edge` one could define a function `path` that takes as argument a node and generates the nodes reachable by a path consisting of zero or more edges.

```
path x = return x `mplus` (do y <- edge x; path y)
```

Evaluating `path "a"` for instance would generate the answers "a", "b" and "c" and the evaluation would then terminate.

This will work fine as long as the graph is acyclic. However if the graph contains a cycle problems will arise. Consider the graph given by figure 2.

Evaluating `path "a"` for this graph would create a loop infinitely reporting "a" and "b", and never reporting "c". In other words `path` fails to generate the reachable nodes, contrary to what was claimed above. However by changing the semantic of how recursive function calls are executed it is possible to restore this property. For this purpose the concept of fixed point combinator is introduced.

```
type FPC a b = ((a -> b) -> a -> b) -> a -> b
```

The combinators are simply means of expressing recursion, and though the type might look complicated they are straightforward to use. Consider the following fixed point combinator `rfp`, an abbreviation of recursive fixed point.

```
rfp :: FPC a b
rfp f = f (rfp f)
```

With this combinator every recursive function can be rewritten into an equivalent fixed point computation. For instance `path` can be rewritten as:

```
path = rfp $ \path x ->
      return x `mplus` (do y <- edge x; path x)
```

Note that with this definition the recursive call inside `path` is no longer made to the Haskell definition, but to the function supplied by the fixed point combinator. It is important to understand this because the whole purpose of rewriting the function this way is to give the combinator control over how recursive calls are made. The semantic of the function is now decided by the properties of the fixed point combinator.

To solve the above mentioned problem with cycles the combinator `rfp` is replaced by another fixed point combinator named `ndlfp`. The name is an abbreviation for non deterministic least fixed point. As the name suggest `ndlfp` will, under certain conditions, construct the least fixed point such that for all input `ndlfp f` will terminate and the equality `ndlfp f = f (ndlfp f)` will hold. In this example this means that `path` will always generate all reachable nodes and then terminate, as long as there are not infinitely many nodes or edges in the graph.

Along these lines another type of fixed point combinators will be presented. Consider the following two examples `acyclic` and `cyclic` which takes a node as input and return a boolean value reporting whether or not the graph reachable from the node is acyclic respectively cyclic.

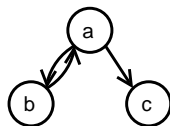


FIGURE 2. cyclic graph

```
acyclic = lfp $ \acyclic x ->
    all (edge x) $ \y -> acyclic y
cyclic = gfp $ \cyclic x ->
    any (edge x) $ \y -> cyclic y
```

A couple of things need to be explained here. First of all the functions `all` and `any` are not the ones provided by the Haskell Prelude, they have for convenience been redefined in this paper. The fixed point combinators are `lfp` and `gfp` creating lowest respectively greatest fixed points.

Both `acyclic` and `cyclic` will terminate regardless of whether the graph contains cycles, and correctly report if cycles are present or not as long as the graph is of finite size. Note the declarative reading of these definitions, the reachable graph is acyclic (cyclic) if all (any) edges leads to nodes where the reachable graph is acyclic (cyclic). Note also that while the combinators makes these definitions well-defined, the plain recursive definition would be meaningless.

2 SEMANTICS

2.1 Origin

The work in this paper is basically wrapped around an attempt to embed the benefits of tabled evaluation into Haskell. Tabled evaluation is an evaluation strategy for logic languages like Prolog. It improves the declarative nature of the language and improves termination. Uses of tabled evaluation include parsing and model checking among others. The main system employing this strategy is the XSB system. For more details see for instance SLG resolution presented in [1], which is the engine behind the XSB system, or see [3].

The aim of this work has been trying out new ideas, so it should be considered experimental. This means that there are definitely things that can be improved or done differently. This goes for the theoretical background presented here as well.

2.2 Monads in Haskell

Monads are used in Haskell as an abstraction of computations, a nice introduction can be found in [10]. They are instances of a type class defined as

```
class Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
```

The functions `return` and `>>=` are required to satisfy certain axioms not stated here. Without getting to technical it could be said that if `m` is a monad then `m a` are computations of values of type `a`. Non determinism can be incorporated in Haskell through special kinds of monads

```
class Monad m => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a
```

In this paper the type constructor `ND` is an instance of `MonadPlus` used to create non deterministic computations. That is if

```
f :: a -> ND b
```

then `f` is a non deterministic function. Deterministic computations on the other hand will be handled with the monad `DT`.

2.3 Some lattice theory

A partially order set (L, \leq) forms a lattice if all finite subsets of L has a least upper bound `lub` as well as a greatest lower bound `glb`. For example $(2^X, \subseteq)$ the power set of any set X forms a lattice under set inclusion. Certain Haskell types can also be considered a lattice, for instance the type `Bool` ordered by $x \leq y \leftrightarrow x \leq y = \text{True}$ forms a lattice. In this paper non deterministic computations will theoretically be considered a lattice, seen as sets of answers ordered by set inclusion. In particular `ND a` is considered a lattice.

Lattices can also be alternatively expressed as an algebraic structure $(L, \vee, \wedge, 0, 1)$ with the operators defined by $x \vee y = \text{lub } \{x, y\}$ and $x \wedge y = \text{glb } \{x, y\}$, and constants $0 = \text{glb } L$ and $1 = \text{lub } L$. Here `lub` denotes least upper bound, and `glb` greatest lower bound. This structure can be captured as a Haskell type class:

```
class JoinLattice a where
  one :: a
  (*) :: a -> a -> a
class MeetLattice a where
  zero :: a
  (+) :: a -> a -> a
class (JoinLattice a, MeetLattice a) =>
  Lattice a
```

Here `one = 1`, `* = \wedge` , `zero = 0` and `+ = \vee` . We can now declare `Bool` instance of `Lattice` like:

```
instance JoinLattice Bool where
  one = True
  x * y = x && y
instance MeetLattice Bool where
  zero = False
  x + y = x || y
instance Lattice Bool
```

The following two definitions will turn out to be useful. They are abstractions of the functions with the same name from the Haskell Prelude, only with the order of the two arguments flipped.

```
all :: Lattice a => [b] -> (b -> a) -> a
all xs f = foldr (*) one (map f xs)
any :: Lattice a => [b] -> (b -> a) -> a
any xs f = foldr (+) zero (map f xs)
```

Important theoretically for this paper is that certain types of functions can be considered lattices. Formally if B is a lattice then functions of type $A \rightarrow B$ forms a lattice ordered by $f \leq g \leftrightarrow \forall x: A f(x) \leq g(x)$. Types of non deterministic functions forms lattices, seen as functions resulting in sets of answers.

2.3.1 The Knaster Tarski theorem

A complete lattice L is a lattice where every subset (finite or infinite) has a least upper bound as well as a greatest lower bound. An order preserving function $f : A \rightarrow B$ is function such that $\forall x, y \in A x \leq y \rightarrow f(x) \leq f(y)$. A fix point of a function $f : A \rightarrow A$ is an $x \in A$ such that $f(x) = x$.

Theorem 1 (Knaster Tarski) *If L is a complete lattice and $f : L \rightarrow L$ is an order preserving function then the set of fix points of f forms a complete lattice.*

In this paper we will only be concerned with the least and greatest fixed points: $\text{lfp}(f)$ and $\text{gfp}(f)$.

And now to the main point: if B is a lattice and $f : (A \rightarrow B) \rightarrow (A \rightarrow B)$ is order preserving then according to Knaster Tarskis theorem f has a least and greatest fixed point: $\text{lfp}(f) : A \rightarrow B$ and $\text{gfp}(f) : A \rightarrow B$. It are these fixed points that the combinators ndlfp , lfp and gfp are concerned with. The type of the combinators are

```
ndlfp :: FPC a (ND b)
lfp :: MeetLattice b => FPC a (DT b)
gfp :: JoinLattice b => FPC a (DT b)
```

Ideally we should have $\text{ndlfp } f = \text{lfp}(f)$, $\text{lfp } f = \text{lfp}(f)$ and $\text{gfp } f = \text{gfp}(f)$, but in general this is not possible.

2.4 Declarative semantic

In this subsection a reasonable condition for correspondence with the theoretical fixed points will be formulated. This condition will serve as the declarative semantic of the fixed point combinators. Internally the combinators use a table to keep track of the calls made and answers generated, and the intuition is that for the computation to terminate the tables must be built in a finite number of steps. More details about the tables are given in the section on the operational semantic. With this intuition in mind we formulate the following criteria

Theorem 2 (Correctness) *If B is a lattice, $f : (A \rightarrow B) \rightarrow (A \rightarrow B)$ is order-preserving and A and B have a finite number of elements then $\text{ndlfp } f \ x = \text{lfp}(f)(x)$.*

with similar conditions for lfp and gfp . The theorem is motivated like this: the fact that A has a finite number of elements enables the table to be of finite size (finite number of entries), the fact that f is order-preserving and B has a finite number of elements enables the fixed point to be reached in a finite number of steps.

To be able to deal with functions that ranges over types that don't have a finite number of elements it is necessary to consider restrictions of the function to finite subsets of the types. By a restriction it is meant that if $g : A \rightarrow B$ and A' and B' are finite subsets of A and B then we can consider a restriction of g to $A' \rightarrow B'$ if for all $x \in A'$ then $g(x) \in B'$. We formulate the following corollary to theorem 2.

Corollary 1 *Suppose B is a lattice and $f : (A \rightarrow B) \rightarrow (A \rightarrow B)$ is order-preserving. If A' and B' are finite subsets of A and B such that f can be restricted to $(A' \rightarrow B') \rightarrow (A' \rightarrow B')$ then the equality $\text{ndlfp } f \ x = \text{lfp}(f)(x)$ holds for all $x \in A'$.*

and similarly for lfp and gfp .

2.4.1 Example

Lets return to the examples in the introduction and verify that they are correct. We take `path` where `x = "a"` and

```
f path x = return x `mplus`
          (do y <- path x; edge y)
```

The fact that `f` is order-preserving means that if `pth` returns more answers than `pth'` for all inputs then `f pth` must return more answers than `f pth'` for all inputs. Given the definition of `f` it is not hard to see that the contrary can not be true. A more formal proof would be to show that `f` is built up by order-preserving subexpressions. Since the number of nodes in the graph is finite we can fulfill the conditions of corollary 1. It is actually possible to tighten this even more by seeing that it is sufficient for the number of nodes reachable from "a" to be finite for the corollary to be used.

The examples `cyclic` and `acyclic` can be verified in a similar way.

2.4.2 Connection to model theory

There exist an analogy between the fixed points and the models of a corresponding logic program. Just as a logic program often has many models most of which are not interesting there are often fixed points which are not interesting. Consider the disconnected graph in figure 3 together with the definition of `path` in the introduction.

The least fixed point will assign both "a" and "b" to {"a", "b"} and "c" to {"c"}. However there exist another fixed point which assigns both "a" and "b" to {"a", "b", "c"} and "c" to {"c"}. Since we intended `path` to generate reachable nodes this is clearly not a fixed point we are interested in.

3 USAGE

3.1 Implementation

The implementation of the fixed point combinators does not involve a lot of code, but is perhaps a little complicated to explain. The embedding does not require any extensions of the language, it is written in ordinary Haskell98. Internally a table is used for memoing. The approach is somewhat similar to the one used for memoing of ordinary functions as described in [7]. In particular low level details for memory management and performance discussed there are relevant here.

3.2 Usage

As said before both ND and DT are monads. Non determinism is introduced by making ND an instance of `MonadPlus`. It probably would have been possible to implement ND and DT as members of monad transformers, but at this point this has been put on the todo list.

```
instance Monad ND where ...
instance Monad DT where ...
instance MonadPlus ND where ..
```

To actually access these monads the following functions are used.

```
toDT :: IO a -> DT a
fromDT :: DT a -> IO a
toND :: IO a -> ND a
solsND :: ND a -> IO [a]
```

Since DT is created to deal with the combinators `lfp` and `gfp` and so is heavily involved with expressions of type `Lattice`, we for convenience define the following instances

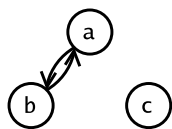


FIGURE 3. disconnected graph

```

instance JoinLattice a => JoinLattice (DT a) where
  one = return one
  x * y = liftM2 (*) x y
instance MeetLattice a => MeetLattice (DT a) where
  zero = return zero
  x + y = liftM2 (+) x y
instance Lattice a => Lattice (DT a)

```

A weakness of `solsND` given above is that it generates all the answers. This prevents from defining a computation that could generate a potential large amount of answers even though we are only interested in one. The function `det` solves this problem by efficiently turning a non deterministic computation to one that computes exactly one answer.

```

class MonadPlus m => MonadDet m where
  det :: a -> m a -> m a
instance MonadDet ND where ...

```

A shortcoming of the combinators are that they are inconvenient to use for defining functions that are mutually recursive. Therefore a function `table` is defined. An example of its use will be given in the section on parsing.

```

class MonadPlus m => MonadTable m where
  table :: (Eq a, Eq b) => (a -> m b) -> a -> m b
instance MonadTable ND where ...

```

To explain how it is used take a look at the following example. With `table` the function `path` could be defined as

```

path = table $ \x ->
  return x `mplus` (do y <- path x; edge y)

```

Internally the function `unsafePerformIO` is used by `table` to create the table used for memoing, so its easy to make mistake about its behavior. In the previous example the table created will be used through out the lifetime of the program, so this could lead to a potential space leak. Also the following wouldn't work, because here a new table will be created each time `path` is invoked.

```

path x = aux x where
  aux = table $ \x ->
    return x `mplus` (do y <- path x; edge y)

```

3.3 Efficiency

The efficiency of the implementation depends upon the lookup time of the data structures making up the table. In the present implementation association lists are

used, which have a lookup time linear to the size of the list. An efficient implementation would use hash tables with a lookup time of $O(1)$. The inconvenience with hash tables are that they require functions calculating good hash keys. A perhaps more subtle point is that the fixed point combinators often would be used to build a large database like structure where all data are alive until the entire structure is released. However the garbage collector typically found in a Haskell implementation wouldn't be the optimal choice for managing such a structure.

4 MOTIVATING EXAMPLES

In the following a couple of examples are given to show some potential applications of the fixed point combinators, as well as to motivate the concept of the combinators.

4.1 Parser

Tabling can extend a parser combinator library with the ability to handle left recursive grammars, as well as taking care of efficiency problems due to backtracking. In fact any context free grammar can be parsed with the addition of tabling. For a detailed exposition of a parser combinator library see for instance [6]. A parser can be defined along the lines described there using ND with

```
newtype Parser a =
  Parser (String -> ND (a,String))
```

The novel extension of the parser would be the addition of tabling. The type `Parser` is made an instance of `MonadTable` like this

```
instance MonadTable Parser where
  table fn x = Parser $ \str ->
    aux (x,str)
  where aux = table $ \(x,str) ->
    applyP (fn x) str
```

Now, here is an example of a left recursive grammar taking advantage of this feature. It captures expressions of ordinary algebra involving floating point numbers. The grammar use `table_` to take care of both the problem of non termination due to left recursion, as well as removing inefficiency due to backtracking.

```
table_ :: (Eq a, MonadTable m) => m a -> m a
table_ m = let aux = table (\() -> m)
  in aux ()
```

```
opr e1 str e2 op =
  do x <- e1; string str; y <- e2; return (op x y)
```

```

exp1 :: Parser Float
exp1 = table_ $ opr exp1 "+" exp2 (+)
      `mplus` opr exp1 "-" exp2 (-)
      `mplus` exp2
exp2 :: Parser Float
exp2 = table_ $ opr exp2 "*" exp3 (*)
      `mplus` opr exp2 "/" exp3 (/)
      `mplus` exp3
exp3 :: Parser Float
exp3 = float `mplus` paren exp1

```

In practice this parser would be inefficient for larger examples if association lists are used internally by the tabling mechanism. But if hash tables were used it should work quite well.

4.2 Model checking and process algebra

Model checking is a technique for verification of finite state concurrent systems, and a process algebra is a formal language for concurrent systems. Both areas are too complex to be described here in a simple way. The reader are directed to other sources for explanation. For a book on model checking see [4], and for books on process algebras see [5] or [8].

One of the applications of tabled logic programming is model checking, for instance the XSB systems has been used for implementing such systems. The reason is that many model checking algorithms are naturally expressed as fixed point computations. It would be an interesting test of the fixed point combinators developed in this article to implement such algorithms.

But the algorithms by them selves are not of much use without a language to express the finite state systems that are to be verified. For this purpose a simple process algebra is embedded in Haskell, represented by the abstract type class `PrAlg`.

4.2.1 Process Algebra

```

class PrAlg a where
  one :: a
  (*) :: a -> a -> a
  zero :: a
  (+) :: a -> a -> a
  (|||) :: a -> a -> a
  silent :: a
  inp :: String -> a
  out :: String -> a
  (\\) :: a -> String -> a
  def :: String -> a -> a

```

```
idn :: String -> a
```

In the class `PrAlg` one represents successful termination and `x * y` sequential composition of `x` and `y`. Non deterministic choice is introduced by `+` with `x + y` being the choice of either `x` or `y`, and `zero` representing a process that has got stuck (will never terminate). The operator `|||` is parallel composition, i.e. `x ||| y` represents `x` and `y` running in parallel.

What makes the algebra capable of expressing interesting systems are the ability of processes to communicate with each other through channels. This communication is accomplished by the functions `inp` and `out`, which allows processes to perform simple synchronous handshake operations. For instance if a process executes `inp a` then it is suspended until another process running in parallel executes a corresponding `out a`. The result of such a communication would be a silent action represented by `silent`.

To restrict processes from communicating with each other the operator `\\` is introduced. Its use is explained by an example.

```
x = (out a \\ a) ||| (inp a \\ a)
y = (out a ||| inp a) \\ a
```

Here `x` and `y` represents two very different processes, with `x` being two deadlocked process running in parallel while `y` results in a silent action caused by communication.

The ability to make recursive definitions are catered by `def` and `idn`. Again an example will clarify how.

```
pause = def "pause" $ one + (silent * idn "pause")
```

The process `pause` will perform zero or more silent actions. Theoretically it might even perform infinitely many such.

Labeled Transition System To actually be able to examine systems expressed in some process algebra they are translated into something called labeled transition systems. The labeled transition system of a process is a graph where the nodes corresponds to the states the process can be in, and the edges are labeled with the action the process perform when moving from one state to another. The type class `LTS` is used to abstract these labeled transition systems. Here `IOS` represent the three action that can be performed; input, output and the silent action.

```
data IOS = Silent
         | Inp String
         | Out String deriving Eq

class Eq a => LTS a where
  arch :: MonadPlus m => a -> m (Maybe (IOS, a))
```

To shed some light on the semantics of `arch` here are some identities.

```
arch zero = mzero
arch (x+y) = arch x `mplus` arch y
arch one = return Nothing
```

4.2.2 Simulation

Working with labeled transition systems the model checking algorithms can be defined. The algorithms that has been chosen to be implemented are simulations, because they are quite simple to express and intuitive to understand. Here is how simulation can be expressed as a fixed point computation.

```
sim :: (LTS a, LTS b) => (a,b) -> DT Bool
sim = gfp $ \sim (x,y) ->
  all (arch x) $ \mx ->
  any (arch y) $ \my ->
  case (mx,my) of
    (Nothing,Nothing) -> return True
    (Just (a,x'), Just (b,y')) ->
      if a == b then sim (x',y')
      else return False
    _ -> return False
```

Bisimulation is easily expressed in terms of simulation.

```
bisim :: (LTS a, LTS b) => (a,b) -> DT Bool
bisim (x,y) = sim (x,y) * sim (y,x)
```

Simulation and bisimulation are often to strict for making interesting comparisons. The reason is that processes must have the same silent actions, and those are the result of hidden internal communications. Usually it is only interesting to compare the way systems interact with the outside world. The concept of weak simulations is used for this purpose. The definition of weak simulation is the same as the one of simulation given above, with the exception that `arch` has been replaced with `warch` where

```
warch :: LTS a => a -> ND (Maybe (IOS, a))
warch = ndlfp $ \warch x ->
  do m <- arch x
  case m of
    (Just (Silent,x'))
      -> warch x'
    _ -> return m
```

```
wsim :: (LTS a, LTS b) => (a,b) -> DT Bool
wsim = gfp $ \wsim (x,y) ->
```

```

do mxs <- toDT (solsND (warch x))
  mys <- toDT (solsND (warch y))
  (all mxs $ \mx ->
   any mys $ \my ->
   ...)

wbisim (x,y) = wsim (x,y) * wsim (y,x)

```

Having defined weak bisimulation it is actually possible to make some interesting verifications. As example, a simplified version of the alternating bit protocol has been verified to be weakly bisimilar to its specification. This protocol is used as example in [2] although for a different algorithm.

The alternating bit protocol is used to transmit messages over an faulty line where the messages might be lost. The protocol ensures that all messages sent from the sender are eventually received by the receiver. Each message sent is tagged with a bit and the receiver will acknowledge each received message by sending that bit. Since the acknowledgment might be lost as well, the sender keeps on sending the same message until it has received the correct acknowledgment, while the receiver keeps on acknowledging.

The version verified here has been simplified so that only a handshake is transmitted, no actual message. The code for the protocol is included in an appendix since it is a little to long to be shown here. The correct behavior to which it is shown to be weakly bisimilar is shown instead.

```

correct :: PrAlg a => String -> String -> a
correct a b = def "correct" $
  inp a * out b * idn "correct"

```

As a final note it can be said that the model checking algorithms presented here are not symbolic. But since binary decision diagrams quite naturally form a lattice it shouldn't be impossible to implement symbolic model checking algorithms using the ideas presented in this article.

5 FUTURE WORK

There is a lot of things that could potentially be improved. From a practical point of view the efficiency of the implementation should be improved. The example with model-checking is an application where more complex examples would place high demands on efficiency, which the current implementation can not live up to.

An other area is proving that the operational semantics of the fixed point operators (operational semantics was not covered in this paper due to lack of space) coincides with the declarative semantic.

Compared to logic based languages there are two major areas not covered by the combinators developed here, negation and unification. In the logic paradigm negation is usually problematic in the sense that it is often only sound under certain

conditions. In the framework presented in this paper the problem with negation is that it is not order-preserving, at least not the way order-preserving was defined in the section on semantics. There exists some interesting solutions however, but due to lack of space it could not be brought up here.

Unification or more generally the ability to handle constraints has not been looked into, but it would definitely be interesting to do so.

6 RELATED WORK

As has been said before this work started out as an attempt to embed tabled evaluation in Haskell, as it seemed both interesting and unlikely it had been done before. The fixed point combinators arose in this process. To what extent the result of the work might coincide with what others have done I don't know. In particular I don't know if tabled evaluation has been translated to the purely functional paradigm before, or if the fixed point combinators has been used to alter the semantic of recursive functions in a similar way.

The work I have found that is most closely related seems to be the one described in[9], called theory of partial-order programming. The goal is similar to the one here, to extend the concept of tabled evaluation to functional programming. One difference is that partial-order programming targets a mixed functional logic language, while the material presented here is set within the purely functional paradigm. It don't appears to be using fixed point combinators either.

One could of course compare the approach with fixed point combinators developed here with tabled evaluation for Prolog. As was brought up in the section on future work negation and unification is not presently covered by the combinators. But the motivating examples brought up shows that there are applications where these features are not needed. By arguing that less is more it one could claim that that in those cases it would be better to do without them.

7 CONCLUSION

The fixed point combinators presented here can provide a way to improve termination and declarative semantic of functional programs. To what extent they will provide a convenient way of solving problems remains to be seen.

A THE ALTERNATING BIT PROTOCOL

```
module AltBit where

import Prelude hiding ((*),(+)
import Alg
import PrAlg
import Bisim
```

```

import DT

faulty a1 a2 b1 b2 b3 = rec
  where aux = inp a1 * (out b1 + out b3)
           + inp a2 * (out b2 + out b3)
        rec = def "faulty" $ aux * idn "faulty"

sender a c1 c2 e1 e2 e3 = rec where
  rec = def "sender" $
    inp a * send1 * inp a *
    send2 * idn "sender"
  send1 = def "send1" $
    out c1 * ((inp e2 + inp e3) *
              idn "send1" + inp e1)
  send2 = def "send2" $
    out c2 * ((inp e1 + inp e3) *
              idn "send2" + inp e2)

receiver b d1 d2 d3 f1 f2 = rec where
  rec = def "receiver" $
    receive1 * receive2 * idn "receiver"
  receive1 = def "receive1" $
    inp d1 * out b * out f1
    + (inp d2 + inp d3) *
    out f2 * idn "receive1"
  receive2 = def "receive2" $
    inp d2 * out b * out f2
    + (inp d1 + inp d3) *
    out f1 * idn "receive2"

altbit a b c1 c2 d1 d2 d3 e1 e2 e3 f1 f2 =

```

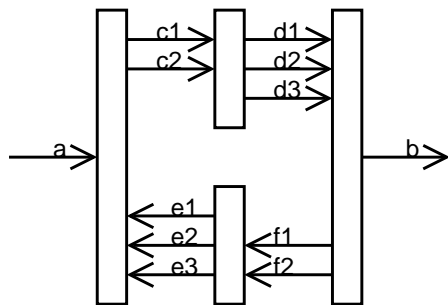


FIGURE 4. schema of altbit

```

sender a c1 c2 e1 e2 e3 |||
receiver b d1 d2 d3 f1 f2 |||
faulty c1 c2 d1 d2 d3 |||
faulty f1 f2 e1 e2 e3

correct :: PrAlg a => String -> String -> a
correct a b = def "correct" $
    inp a * out b * idn "correct"

check = testDT (wbisim (x,y))
  where x :: PA
        x = altbit a b c1 c2 d1 d2 d3
              e1 e2 e3 f1 f2
              \\ c1 \\ c2 \\ d1 \\ d2 \\ d3
              \\ e1 \\ e2 \\ e3 \\ f1 \\ f2
        y = correct a b :: PA
        (a,b) = ("a", "b")
        (c1,c2,d1,d2,d3) =
            ("c1", "c2", "d1", "d2", "d3")
        (e1,e2,e3,f1,f2) =
            ("e1", "e2", "e3", "f1", "f2")

```

REFERENCES

- [1] Weidong Chen and David S. Warren. Towards effective evaluation of general logic programs, 1993.
- [2] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. 1986.
- [3] Lars Degerstedt. *Tabulation-based Logic Programming: A Multi-level View of Query Answering*. PhD thesis, Linköping Studies in Science and Technology, 1996.
- [4] Jr. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. 1999.
- [5] W. J. Fokkink. *Introduction to Process Algebra*. 2000.
- [6] Graham Hutton and Erik Meijer. Monadic parser combinators. Technical report, Department of Computer Science, University of Nottingham, 1996.
- [7] Simon Peyton Jones, Simon Marlow, and Conal Elliott. Stretching the storage manager: weak pointers and stable names in haskell. 1999.
- [8] Robin Milner. *Communicating and mobile systems*. 1999.
- [9] Mauricio Osorio, Bharat Jayaraman, and David A. Plaisted. Theory of partial-order programming. *Sci. Comput. Program.*, 34(3):207–238, 1999.
- [10] Philip Wadler. The essence of functional programming. 1992.