Algebraic specification and simulation of lazy functional programs in a concurrent environment

Eindhoven University of Technology
Department of Mathematics and Computing Science

Algebraic Specification and Simulation of Lazy Functional Programs in a Concurrent Environment

by

Loe Feijs

96/20

ISSN 0926-4515

Computing Science Report 96/20
Eindhoven, October 1996

Algebraic Specification and Simulation of Lazy Functional Programs in a Concurrent Environment

Loe Feijs
Philips Research Laboratories Eindhoven,
Eindhoven University of Technology

Abstract

The mechanism of Landin-style stream input/output (I/O) makes it possible to write functional programs, which behave as reactive systems when executed with lazy evaluation. Functional programming languages like Gofer are attractive for programming the data transformations of a reactive system. But although the I/O behaviour can be programmed in such languages too, the functional paradigm lacks the capabilities for specification and reasoning which are needed to analyse the communication behaviour of the program and its enviroment. We propose to use the Algebra of Communicating Processes (ACP) for that purpose. The present paper attempts to bridge the gap between the functional and the process-oriented worlds. It is shown how a simple generator can produce both Gofer program patterns and ACP equations. The patterns can be completed with data transformation functions and then executed whereas the equations can be used for reasoning and simulation. The term rewriting system (TRS) of the functional language, the structured operational semantics (SOS) of the I/O mechanism, the process equations and the fixed point semantics of a program are described and their relationships are analysed. We abstract from the details of the particular programming language by using an intermediate concept of 'abstract functional program'.

Key Words: ACP, Bisimulation, Communication protocols, Component generation, Denotational semantics, Executable specifications, Fixed point theory, Formal reasoning, Functional programming, Gofer, Labeled transition systems, Lambda calculus, Lazy evaluation, Patterns, PSF, Scripting, Simulation, Streams, Structured operational semantics.

1 Introduction and motivation

In many circumstances it is important to have precise specifications of computer programs in order to analyse their behaviour in the context of a large system. In particular this is important for 'scripted agents', by which we mean programs which are meant for being transfered through a communication network for execution as a reactive program at another site. In this paper we investigate a number of issues arising when a program is written in a programming language which is based on lazy evaluation, like Gofer [1]. Although it is possible to let a functional program perform all kinds of I/O actions, such as reading and writing files, it makes sense to adopt a restriction to so-called Landin-style stream I/O; in that case, a program, viewed as a process in a concurrent environment (a distributed system), will have only one input port and one output port. Of course a distributed system will need multi-port components like routers too, but one can assume that these are realised by other means already and that these can be specified using process-theoretic means, for example ACP [2]. In this way the data processing aspects are separated from the communication aspects. By means of an example it will be shown that this can be done without loss of generality (provided the network contains routers). For our examples we use Gofer and we work out the details to such a level that we can use the axioms of $\mathrm{ACP}_\epsilon^\tau$ [2] for reasoning and ACP-based tools like PSF [3] for simulation.

Related work: in [4] it was already shown that the concept of stream domains makes it possible to relate certain λ-calculus based program descriptions to behavioural descriptions of the program in a concurrent environment. In [5] an operational semantics of a Haskell fragment is given and used to derive process-theoretic properties using CCS. In [6] a powerful I/O mechanism is proposed which is not restricted to Landin-style stream I/O. Whereas we view a functional program as one of many agents in a concurrent world, [6] addresses the problem the other way around, turning the world into a set of objects being manipulated by the functional program.

Survey of the work: in Section 2 a survey of the relevant aspects of lazy functional languages is given. In Section 3 some aspects of ACP are introduced. In Section 4 a first example is studied (ping-pong behaviour). In Section 5 a second example is studied (state-based behaviour). In Section 6 another technicality is added (non ping-pong behaviour). In Section 7 we propose a constructive approach, using a generator to make program patterns together with their ACP equations. In Section 8 we show this constructive approach in action, when we build a simple distributed system consisting of a service provider (an Eliza-like psychiater) and a service manager (filtering Eliza's advice when the user did not pay). In Section 9 we use PSF to simulate the ACP equations of this service manager together with a model of its environment. In Sections 10 and 11 we study the correctness aspects and certain semantic aspects, respectively. In Section 12 some concluding remarks are given.

2 Aspects of lazy functional languages

Functional programming languages have been used for artificial intelligence (AI) applications and for tool construction for many years. Important languages are LISP, ML, Miranda, Haskell, Clean and Gofer. Several of the more recent languages are based on *lazy evaluation*, which amounts to a particular reduction strategy together with certain assumptions about the representation and manipulation of data structures. With respect to the reduction strategy, *lazy evaluation* means that:

- an argument to a function is not earlier evaluated than when its value is needed (so if it is not needed at all, it is not evaluated),
- an argument to a function is evaluated only once, also if its value is needed several times during the function's execution.

The important data type of lists is always built-in to functional languages. With respect to list-processing, lazy evaluation means that:

- if the result of an execution is a list, then this list is delivered in an incremental way, i.e. the head will be delivered first (while arguments only relevant for the tail are not evaluated),
- if the argument of a function is a list, then evaluation of the function can start already before all list elements are available (typically a function requires the list's head first, then the head of its tail and so on).

For a survey of Gofer, see [1]. An interactive Gofer program with top level function `f` is a kind of executable function of type `[Response] -> [Request]` (assuming that Gofer's standard prelude is imported). The program produces requests to its environments, such as requests to read and write strings from standard input (`"stdin"`) and to standard output (`"stdout"`). The environment gives responses, containing success/failure indications and strings, which serve as inputs for the program.

If we refrain from using arbitrary calls to the file system, and instead of that, just read and write from/to standard input and standard output, the type of `f` is much simpler. It is of type `String -> String`. In this case the main program looks as follows:

```
main = interact f
f :: String -> String
```

When this is executed, the characters are read (for example from the keyboard) and then processed by `f`. The function `interact` is a predefined function from Gofer's standard prelude. The lazy evaluation mechanism determines at which points in time there has been enough input in order to produce output. The interaction behaviour can still be complex, in the sense that the the program consumes n_1 inputs before producing m_1 outputs, then n_2 inputs followed by m_2 outputs etc., where the n_i and m_i depend on the contents of the lines read so far. Sometimes this is called Landin-style stream I/O.

Although `f` is declared as `String -> String` it consumes and produces information in certain chunks, normally characters. In order to have a more practical granularity for the I/O, we shall in the sequel assume that each line is treated as a separate chunk of information. Therefore we focus on programs whose '`main`' is as follows:

```
main = interact (unlines . g . lines)
g :: [String] -> [String]
```

Here we used the function `lines` from the standard prelude; it breaks a string into a sequence of strings by recognising the end of line characters. And `unlines` is its inverse. Note the '`.`' operator, which denotes function composition.

Next to the functional behaviour of `g`, we need to understand the behaviour of `g` in a concurrent environment, where synchronisation is relevant. As explained above, the synchronisation between the responses and the requests is regulated by the lazy execution mechanism. This implies that from the environment's point of view, `g` is *eager to deliver results*: it produces as much outputs (requests) as possible, only pauzing to wait for an input (a response) if no other action is possible.

3 Aspects of ACP

The Algebra of Communicating Processes ACP proposed by Bergstra and Klop [7], is a theory about processes and their communication behaviour in the tradition of CCS [8]. For an introduction to ACP, see [2]. We mention some of the most important operators: $+$ for alternative composition, $\cdot$ for sequential composition, τ for silent step and $\|$ for parallel composition. The laws of ACP are always written as equations, such as the following laws, called 'basic process algebra'.

$$x + y = y + x$$

$$\begin{aligned}(x+y)+z &= x+(y+z)\\ x+x &= x\\ (x+y)\cdot z &= x\cdot z+y\cdot z\\ (x\cdot y)\cdot z &= x\cdot(y\cdot z)\end{aligned}$$

To these one has to add additional laws describing $||$, τ etc., for example $x \cdot \tau = x$.

ACP is parameterised over an action-alphabet A, which must be chosen dependent on the application domain. For the purpose of studying interactive functional programs, we assume that A contains the following elements:

- $s(t)$ ($t \in$ `String`),
- $r(t)$ ($t \in$ `String`),
- $c(t)$ ($t \in$ `String`),
- τ.

ACP is also parameterised over a binary communication function $\gamma : A \times A \to A$, which must be chosen dependent on the application domain. We define the partial function γ such that one 'send' and one 'receive' together make one 'communication'. This is expressed by the following equations:

$$\begin{aligned}\gamma(s(t),r(t)) &= c(t)\\ \gamma(r(t),s(t)) &= c(t)\\ \gamma(a_1,a_2) &= \text{undefined} \qquad \text{(otherwise)}\end{aligned}$$

These choices allow us to use ACP for the purpose of studying interactive functional programs provided we may assume that, when viewed as a process, a lazy functional program has a single input port corresponding to actions $r(t)$, and an output port corresponding to actions $s(t)$, as sketched in Figure 1.

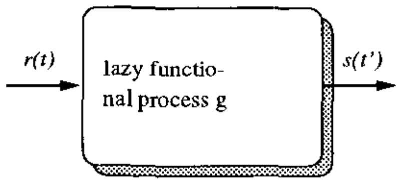

Figure 1: Lazy functional program viewed as a process.

At first sight this model looks too naive, because a simple experiment shows that when the input of the program comes directly from a keyboard, the user can continue typing, even when the program is not ready for consumption of the next line typed. This is explained however, by assuming that there is a buffer between the keyboard and the program. This buffer queues the lines which are typed. Similarly an output buffer is assumed for the results which are to be displayed on the user's screen. The buffers are not considered part of the process of g; they belong to the environment, as sketched in Figure 2 below.

These preparations will enable us to address a central question in the next sections: which ACP equations describe the behaviour of a functional program g, viewed as a process?

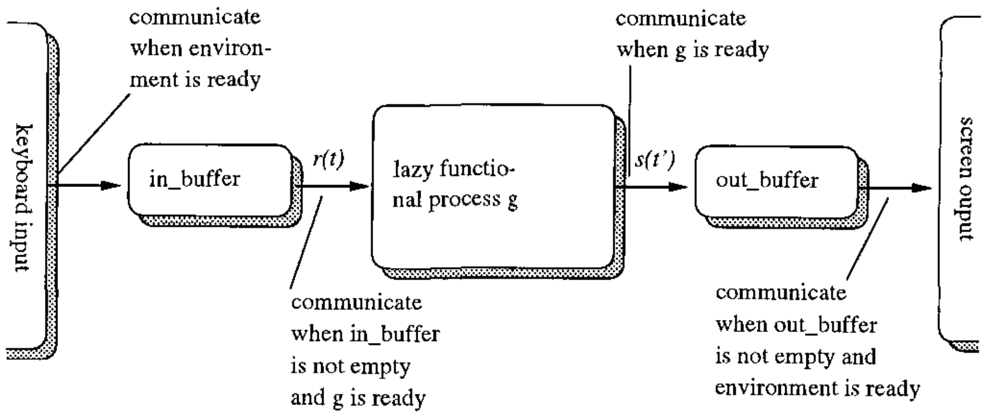
communicate when environ- ment is ready
communicate when g is ready
keyboard input
in_buffer
r(t)
lazy functio- nal process g
s(t')
out_buffer
screen ouput
communicate when in_buffer is not empty and g is ready
communicate when out_buffer is not empty and environment is ready

Figure 2: Environment for teletype I/O.

4 Example (memoryless function)

The first example is a particularly simple kind of process. It produces and consumes its chunks of information in an alternating fashion. The program given below transforms each input line into an output line by applying a memoryless mapping `updline` (for 'update line') from `Line` to `Line`.

```
type Line = String
type Word = String

main = interact (unlines . g . lines)

g :: [Line] -> [Line]
g =  updlines

updlines :: [Line] -> [Line]
updlines []       = []
updlines (l : ls) = (updline l) : (updlines ls)

updline :: Line -> Line
updline l  = unwords (updwords (words l))

updwords :: [Word] -> [Word]
updwords []        = []
updwords (w : ws)  = (map toUpper w) : (updwords ws)
```

Note that `map` is a built-in concept of Gofer and that `toUpper` is a function from the standard prelude. This program can for example perform a dialogue as follows:

```
1. let us explain the compiler           (receive)
2. LET US EXPLAIN THE COMPILER           (send)
3. it has an easy evaluator              (receive)
4. IT HAS AN EASY EVALUATOR              (send)
```

Next we investigate the process behaviour of g. If we want to see a Gofer program g as a process, we denote it as $\mathcal{P}[\![\mathtt{g}]\!]$. If we want to consider g as a mathematical function we denote it as $\mathcal{F}[\![\mathtt{g}]\!]$. From the definition of `updlines` we see that the first element of the result updlines (l : ls) depends only on l and does not require ls. Therefore the program produces an output immediately after each input. Of course the program first performs some internal rewriting, which is modeled by a silent step τ. This tells us that it satifies the ACP equation.

$$\mathcal{P}[\![\mathtt{g}]\!] = \tau \cdot \sum_x r(x) \cdot s(\mathtt{updline}(x)) \cdot \mathcal{P}[\![\mathtt{g}]\!]$$

which is consistent with the following equation concerning the functional behaviour, which is obvious from the program:

$$\mathcal{F}[\![\mathtt{g}]\!](x : xs) = \mathtt{updline}(x) : \mathcal{F}[\![\mathtt{g}]\!](xs)$$

5 Example function with memory

Next we look at an example of a more general case. The process below transforms each input line into an output line, but it is not just a 'memoryless' mapping from Line to Line. It has an internal state, coded as a 'dictionary', whose type is called Dict.

```
type Line = String
type Word = String
type Dict = [Word]

main = interact (unlines . g . lines)

g :: [Line] -> [Line]
g =  updlines iniDict

updlines :: Dict -> [Line] -> [Line]
updlines d []       = []
updlines d (l : ls) = (updline d l) : (updlines (updDict d l) ls)

updDict :: Dict -> Line -> Dict
updDict d l = (add (words l) d)

updline :: Dict -> Line -> Line
updline d = unwords . (map (updword d)) . words

updword d w | isin w d  = w
            | otherwise = map toUpper w

add :: [Word] -> Dict -> Dict
add [] d = d
add (w : ws) d | isin w d  = add ws d
               | otherwise = w : (add ws d)

isin :: Word -> Dict -> Bool
isin w [] = False
isin w (x : xs)
      | (w == x)         = True
      | (w ++ "s" == x)  = True
      | (w == x ++ "s")  = True
      | otherwise        = isin w xs
```

```
iniDict :: Dict
iniDict = ["in","the","and","is","has","it", "from","thing", "easy","have",
          "a","at","of","to","use","will","find","one",  "always","about",
          "an","now","let","us","talk",        "mention","explain","work",
          "so-called","define","separate","treat","way","as","if","clear",
          "next","or","may","be","used","new","some","where", "no","find",
          "should","tell","we",        "however","try","for",     "form",
          "they",                                              "are","not"]
```

This program performs a slightly more interesting task. It maps fresh words (except for frequently used verbs and particles) to uppercase, but only in the line where they occur for the first time. This is a dialogue:

1. `let us explain the compiler`	(receive)
2. `let us explain the COMPILER`	(send)
3. `it has an easy evaluator`	(receive)
4. `it has an easy EVALUATOR`	(send)
5. `evaluators are always easy`	(receive)
6. `evaluators are always easy`	(send)

This process satisfies the ACP equation $\mathcal{P}[\![\mathtt{g}]\!] = U_{\mathrm{iniDict}}$ where for all d of type `Dict` we have that U_d is given by:

$$U_d = \tau \cdot \sum_l r(l) \cdot s(l') \cdot U_{d'}$$

where $l' \equiv$ ((`unwords . (map (updword` d`)) . words)` l) and $d' \equiv$ `(add (words l)` d`)`. This is consistent with the functional behaviour for which we check from the Gofer program text that $\mathcal{F}[\![\mathtt{g}]\!] = u(\mathtt{iniDict})$ where for all strings l and ls,

$$u(d)(l : ls) = l' : u(d')(ls)$$

6 Example which does not ping-pong

The previous examples had alternating send and receive behaviour (ping-pong behaviour). But can we also make a program which does not ping-pong, but which does ping-ping-pong-pong? Such a process G would be described by an ACP equation of the form:

$$G = \tau \cdot \sum_{t_1,t_2} r(t_1) \cdot r(t_2) \cdot s(t'(t_1,t_2)) \cdot s(t''(t_1,t_2)) \cdot G'(G,t_1,t_2)$$

Yes, we can, and then the functional behaviour for g is described by an equation of the form:

$$g\,(t_1 \,:\, t_2 \,:\, ts) = t'(t_1,t_2) \,:\, t''(t_1,t_2) \,:\, g'(g,t_1,t_2)(ts)$$

In fact, the number of pings and pongs can be determined dynamically, as shown by the program given below. This program compares each subsequent input line with the current line, and throws the next line away if it is a subset (with respect to the non-trivial words) of the current line. The set of 'trivial' words is given by `iniDict`.

```
main = interact (unlines . g . lines)

g :: [Line] -> [Line]
```

```
g = mute []

mute :: [Word] -> [Line] -> [Line]
mute c [] = []
mute c (l : ls) | (criterion c l)  =  mute c ls
                | otherwise        = l : (mute (nontrivs (words l)) ls)

criterion :: Dict -> Line -> Bool
criterion c l = subset (nontrivs (words l)) c

subset :: [Word] -> [Word] -> Bool
subset [] c = True
subset (w : ws) c | (isin w c) = subset ws c
                  | otherwise  = False

nontrivs :: [Word] -> [Word]
nontrivs [] = []
nontrivs (w : ws) | (isin w iniDict) = nontrivs ws
                  | otherwise        = w : (nontrivs ws)

-- Line, Word, Dict, isin, iniDict as before
```

This program can for example perform a dialogue of the ping-pong-ping-pong-ping-ping-pong type, as shown below.

```
1. let us explain the compiler        (receive)
2. let us explain the compiler           (send)
3. it has an easy evaluator           (receive)
4. it has an easy evaluator              (send)
5. evaluators are always easy         (receive)
6. thanks heaven                      (receive)
7. thanks heaven                         (send)
```

The line "evaluators are always easy" is viewed as a subset of "it has an easy evaluator" and therefore the latter line is not echoed.

7 A constructive approach

In general it is impossible to automatically analyse arbitrary Gofer programs to find their ACP equations. But we can easily identify a number of useful patterns for which the Gofer program can be analysed. For each such pattern we can give both the ACP equations and an outline of the Gofer program. One could make a library of such pairs consisting of a pattern and a set of equations. We demonstrate this idea for a few interesting pairs and we cast the library into the form of a menu-driven generator. The user (programmer) still has to add definitions of a number of functions called `f0`, `f1`, `f2` etc. We devised a few patterns for which one needs a number of user-defined functions; these have the following characterisation.

- `f0` – initialisation (cf. `iniDict` in Section 5),
- `f1` – reaction (cf. `updline` in Section 5),
- `f2` – reflection (how to update the internal state, cf. `updDict` in Section 5),
- `f3` – filtercriterion (which lines will not pass, cf. `criterion` in Section 6),

- `f4` – stop-criterion,
- `f5` – menu.

We characterise the patterns by a kind of regular expression. The generator given below supports three patterns (although its menu has three more patterns). Of course many extensions and generalisations are possible.

```
--  G = skip . S(f5) . sum( x in Line,
--                        , r(x) . s(f1(c,x))
--                        )
--  S([]) = skip
--  S(x : xs) = s(x) . S(xs)
type Line = String
type Word = String
type Dict = [Word]
main = interact (unlines . g . lines)
g :: [Line] -> [Line]
g xs = f5 ++ g' xs
g' :: [Line] -> [Line]
g' (x : xs) = f1 x
f1 :: Line -> [Line]
f5 :: [Line]

f1 x = generate (head (words x))
f5 = menu

generate :: Word -> [Line]
generate "0"  = acp0 ++ prelu ++ gofer0
generate "1"  = acp1 ++ prelu ++ gofer1
generate "2"  = acp2 ++ prelu ++ gofer2
generate "3"  = acp3 ++ prelu ++ gofer3
generate rest = ["sorry"]

menu = [ "0: s*rs*         ",
         "1: (rs)inf - mem ",
         "2: (rs)*         ",
         "3: (r[s])inf     ",
         "4: (rs)inf       ",
         "5: (rs*)inf      ",
         "6: s(rs)*        "
       ]

prelu = [ "type Line = String                  ",
          "type Word = String                  ",
          "type Dict = [Word]                  ",
          "main = interact (unlines . g . lines) ",
          "g :: [Line] -> [Line]               "
        ]

gofer0 = [ "g xs = f5 ++ g' xs                  ",
           "g' :: [Line] -> [Line]              ",
           "g' (x : xs) = f1 x                  ",
           "f1 :: Line -> [Line]                ",
           "f5 :: [Line]                        "
         ]
```

```
gofer1 = ["g (x : xs) = (f1 x) : (g xs)                          ",
          "f1 :: Line -> Line"
         ]

gofer2 = ["g = g' f0                                              ",
          "g' :: Dict -> [Line] -> [Line]                         ",
          "g' c (x : xs) | (f4 c x)  = []                         ",
          "              | otherwise = (f1 c x) : (g' (f2 c x) xs) ",
          "f0 :: Dict                                             ",
          "f1 :: Dict -> Line -> Line                             ",
          "f2 :: Dict -> Line -> Dict                             ",
          "f4 :: Dict -> Line -> Bool                             "
         ]

gofer3 = ["g = g' f0                                              ",
          "g' :: Dict -> [Line] -> [Line]                         ",
          "g' c (x : xs) | (f3 c x)  = g' (f2 c x) xs             ",
          "             | otherwise = (f1 c x) : (g' (f2' c x) xs) ",
          "f0  :: Dict                                            ",
          "f1  :: Dict -> Line -> Line                            ",
          "f2  :: Dict -> Line -> Dict                            ",
          "f2' :: Dict -> Line -> Dict                            ",
          "f3  :: Dict -> Line -> Bool                            "
       ]

acp0 = ["--  G = skip . S(f5) . sum( x in Line,                          ",
        "--                        , r(x) . s(f1(c,x))                   ",
        "--                        )                                     ",
        "--  S([]) = skip                                                ",
        "--  S(x : xs) = s(x) . S(xs)                                    "
         ]

acp1 = ["--  G = sum( x in Line, r(x) . s(f1(x)) . G )                   "
       ]

acp2 = ["--  G(c) = skip . ( sum( x in Line                              ",
        "--                     , [f4(c,x)=True]->                       ",
        "--                       skip                                   ",
        "--                     )                                        ",
        "--                 + sum( x in Line                             ",
        "--                     , [f4(c,x)=False]->                      ",
        "--                       r(x) . s(f1(c,x)) . G(f2(c,x))         ",
        "--                 )    )                                       "
       ]

acp3 = ["--  G = skip . G(f0)                                            ",
        "--  G(c) = sum( x in Line                                       ",
        "--             , [f3(c,x)=True]-> r(x) . G(f2(c,x))             ",
        "--             )                                                ",
        "--         + sum( x in Line                                     ",
        "--             , [f3(c,x)=False]-> r(x) . s(f1(c,x)) . G(f2'(c,x))",
        "--             )                                                "
       ]
```

This program can generate its own pattern. More precisely, by choosing option 0 it produces its own prelude up to (and including) the typing clauses of the definition of `f1` and `f5`. The ACP equations are generated as comment (this is an example of a component-generator in the sense proposed in [9]). The concrete syntax for ACP chosen is the syntax of PSF. The choice for PSF has two advantages: first, it has an ASCII syntax, and secondly there are simulation and analysis tools for it.

8 The generator in action

In this section we show the generator in action. We generate (part of) a manager for a psychiater. When experimenting with this system we employ M. Jones' Gofer version of Eliza, which comes together with the standard Gofer distribution.

The manager must add addresses to its output. We assume an external router, which is capable of interpreting these addresses. The network structure is as follows: the user's problems (lines beginning with 'p:') and money (lines beginning with 'm:') as well as Eliza's answers (lines beginning with 'a:') are queued and serve as input for the manager. There is a router, whose output '1' is connected to the user's console (address 1) whereas output '2' goes via a buffer to Eliza, which has address 2, (see Figure 3). The manager is made using the generator, by choosing option 3. In order

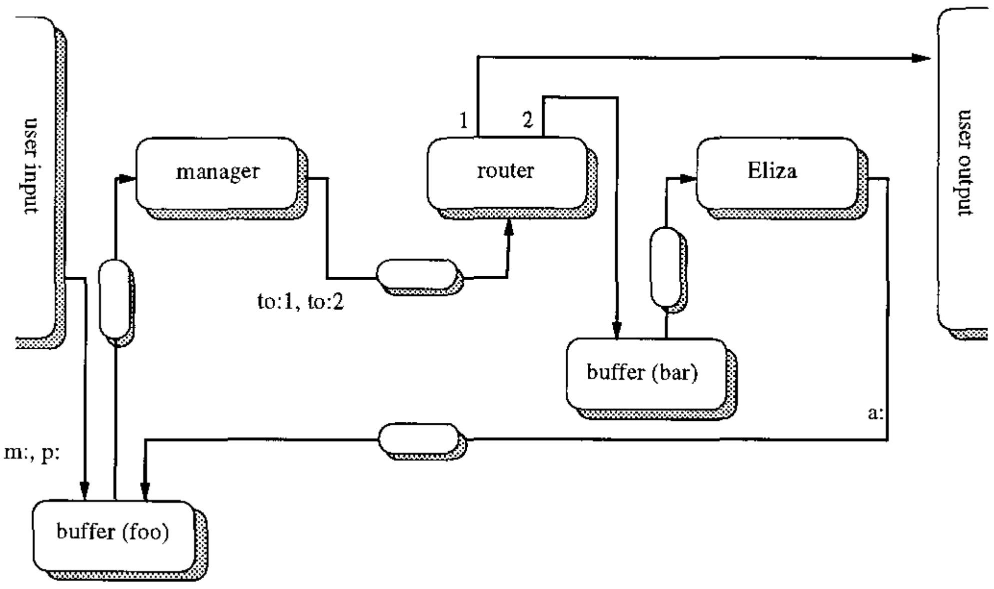

Figure 3: Service network structure.

to support our claim that this approach is at the same time practically executable and amenable to analysis with algebraic means, we did two things:

- we constructed an environment for the Gofer interpreter using buffer-access routines and a router written in C, exploiting the multitasking capabilities of a standard operating system; this is reported in the present section (see below);
- we specified the buffers and the router in PSF, which in combination with an algebraic specification of the data-manipulation functions (`f0`, `f1`, `f2`, `f2'` and `f3`) and the generated equations allowed us to simulate the manager process in

PSF; the details are given in Section 9. At the same time these PSF texts are equational ACP_ϵ^τ specifications which could be used to formally derive properties of the system. We ran the simulator, but we did not really perform any further formal derivations.

We needed a kind of multitasking environment. We realised this in a Unix environment, using files for the two main buffers and using pipes for the four other buffers (the small ones in the figure). We wrote C programs `reader`, `writer` and `router`. The system is started by issuing the following three commands:

1. `( writer foo )`
2. `( reader foo ) | ( gofer manager.gs ) | ( router bar )`
3. `( reader bar ) | ( gofer eliza.gs ) | ( writer foo )`

(each command was issued in a separate shell, which is easy when using e.g. an X window system). The writer takes lines from the users terminal and writes them to a file. The reader reads lines from a file and puts them on the user's screen. The router reads lines from its standard input; lines which begin with `to:1` are routed to its output port 1, lines which begin with `to:2` are routed to its output port 2, and all other lines are thrown away.

We could only use the interactive version of Gofer, probably because the compiled versions of the programs did not force their pipes to be properly flushed. For initialisation purposes we had to put the word 'main' in each of the buffer-files 'foo' and 'bar'. We expect that by modifying Gofer's `runtime.c` these technicalities can be resolved in other ways too.

```
--   G = skip . G(f0)
--   G(c) = sum( x in Line
--             , [f3(c,x)=True]-> r(x) . G(f2(c,x))
--             )
--        + sum( x in Line
--             , [f3(c,x)=False]-> r(x) . s(f1(c,x)) . G(f2'(c,x))
--             )
type Line = String
type Word = String
type Dict = [Word]
main = interact (unlines . g . lines)
g :: [Line] -> [Line]
g = g' f0
g' :: Dict -> [Line] -> [Line]
g' c (x : xs) | (f3 c x)  = g' (f2 c x) xs
              | otherwise = (f1 c x) : (g' (f2' c x) xs)
f0  :: Dict
f1  :: Dict -> Line -> Line
f2  :: Dict -> Line -> Dict
f2' :: Dict -> Line -> Dict
f3  :: Dict -> Line -> Bool

-- m: money
-- p: problems
-- a: advice

f0 = ["$"]
```

```
f1 c l | (words l        == [] )  = l
       | (head (words l) == "m:") = l
       | (head (words l) == "p:") = "to:2 " ++ unwords (tail (words l))
       | (head (words l) == "a:") = "to:1 " ++ unwords (tail (words l))
       | otherwise                = l

f2 []      l | (words l        == []  ) =  []
             | (head (words l) == "m:") =  tendollar
             | otherwise                =  []
f2 (d : c) l | (words l        == [] )  =  (d : c)
             | (head (words l) == "m:") =  tendollar ++ (d : c)
             | (head (words l) == "p:") =                    c
             | otherwise                =                (d : c)

f2' c l = f2 c l

f3 []       l | (words l        == []    ) = True
              | (head (words l) == "a:" ) = False
              | otherwise                  = True
f3 (d : c)  l | (words l        == []    ) = True
              | (head (words l) == "a: ") = False
              | (head (words l) == "p: ") = False
              | otherwise                  = False

tendollar = ["$","$","$","$","$","$","$","$","$","$"]
```

The manager gives an initial credit of one dollar to the user. This suffices for one problem being asked to Eliza. Lines starting with 'm:' are treated as an advance payment of ten dollars. This system can for example perform a dialogue as follows:

1. `Hi! I'm Eliza. Please tell me your problem.` (receive)
2. `p: I do not like my computer any more` (send)
3. `Do computers worry you?` (receive)
4. `p: no, no` (send)
5. `p: no, no` (send)
6. `m: enclosed are ten dollars` (send)
7. `p: no, no` (send)
8. `Are you saying no just to be negative?` (receive)

Note that the user does not get an answer for his problems on lines 4 and 5. This is because the manager's sees the acount is empty and does not pass the problems to Eliza. From the payment of line 6 onwards, the contact with Eliza is re-established.

9 Simulating ACP equations with PSF

We have simulated the generated equations using the PSF tools. The PSF model consists of two parts: data specifications and process specifications. We just survey the sorts and their operations: The sort `Bool` has values `True` and `False`, the sort `Word` has value `D` and the sort `Prefix` has values `'a'`, `'m'`, `'p'`, `'1'` and `'2'`. Next, the sort `Contents` has values like `hi-i-m-eliza-please-tell-me-your-problem` and `do-computers-worry-you`. The sort `Lines` has constructor `_++_ : Prefix # Content`

`-> Line` and operations `head` and `tail`. Finally the sort `Dict`, has constructors `empty-dict` and `cons` and operations `head`, `tail` and finally `_++_` from `Dict` and `Dict` to `Dict`.

```
data module F0122'3
begin
  exports
    begin
      functions
        f0  :             -> Dict
        f1  : Dict # Line -> Line
        f2  : Dict # Line -> Dict
        f2' : Dict # Line -> Dict
        f3  : Dict # Line -> Bool
    end
  imports
    Dicts, Lines, Words, Prefixes, Contents, Bools
  functions
    tendollar : -> Dict
  variables
    c  : -> Dict
    l  : -> Line
    d  : -> Word
  equations
   [01] f0 = cons(D,empty-dict)

   [02] f1(c,l) = l                when head(l) = 'm'
   [03] f1(c,l) = '2' ++ tail(l)   when head(l) = 'p'
   [04] f1(c,l) = '1' ++ tail(l)   when head(l) = 'a'

   [05] f2(empty-dict,l) = tendollar  when head(l) = 'm'
   [06] f2(empty-dict,l) = empty-dict when head(l) = 'p'
   [07] f2(empty-dict,l) = empty-dict when head(l) = 'a'

   [08] f2(cons(d,c),l) = tendollar ++ cons(d,c) when head(l) = 'm'
   [09] f2(cons(d,c),l) = c                      when head(l) = 'p'
   [10] f2(cons(d,c),l) = cons(d,c)              when head(l) = 'a'

   [11] f2'(c,l) = f2(c,l)

   [12] f3(empty-dict,l) = False  when head(l) = 'a'
   [13] f3(empty-dict,l) = True   when head(l) = 'm' -- otherwise
   [14] f3(empty-dict,l) = True   when head(l) = 'p'

   [15] f3(cons(d,c),l)  = False  when head(l) = 'a'
   [16] f3(cons(d,c),l)  = False  when head(l) = 'p'
   [17] f3(cons(d,c),l)  = False  when head(l) = 'm' -- otherwise

   [18] tendollar = cons(D,cons(D,cons(D,cons(D,cons(D,
                    cons(D,cons(D,cons(D,cons(D,cons(D,empty-dict))))))))))
end F0122'3
```

There are some process modules needed to define the enumerated sets needed to make the simulations run efficiently, but these are not shown here. We only show the `communications` section and the `definitions` sections, since it is not hard to guess the essentials of the `exports`, `imports` etc. from these.

```
process module Example2
begin
```

```
communications
  read-foo(x)       | r(x)            = int-read-foo(x)           for x in Line
  s(x)              | s'(x)           = int-manager-to-router(x)  for x in Line
  write-router'(x)  | write-router(x) = int-manager-to-router(x)  for x in Line
  read-router2(c)   | write-bar(c)    = int-router-to-bar(c)      for c in Content
  read-bar(c)       | r(c)            = int-bar-to-eliza(c)       for c in Content
  ss(x)             | ss'(x)          = int-eliza-to-foo(x)       for x in Line
  write-foo'(x)     | write-foo(x)    = int-eliza-to-foo(x)       for x in Line

definitions
  manager = skip . G(f0)
  G(c) = sum( x in Line
            , [f3(c,x)=True]-> r(x) . G(f2(c,x))
            )
       + sum( x in Line
            , [f3(c,x)=False]-> r(x) . s(f1(c,x)) . G(f2'(c,x))
            )

  eliza = ss('a' ++ hi-i-m-eliza-please-tell-me-your-problem) . E

  E     =   r(i-do-not-like-my-computer-any-more)
            . ss('a' ++ do-computers-worry-you)
            . E
          + r(no-no)
            . ss('a' ++ are-you-saying-no-just-to-be-negative)
            . E

  router = sum( x in LINE
              ,   [head(x)='1']-> write-router(x) . read-router1(tail(x)) . router
                + [head(x)='2']-> write-router(x) . read-router2(tail(x)) . router
                + [head(x)='m']-> skip . router
                + [head(x)='a']-> skip . router
                + [head(x)='p']-> skip . router
              )

  foo = sum( x in LINE   , write-foo(x) . read-foo(x)  . foo)
  bar = sum( c in CONTENT, write-bar(c) . read-bar(c) . bar)

  manager-to-router = sum(x in LINE, s'(x)  . write-router'(x)) . manager-to-router
  eliza-to-foo      = sum(x in LINE, ss'(x) . write-foo'   (x)) . eliza-to-foo

  psychosystem = encaps(H, (  manager
                           || eliza
                           || foo
                           || bar
                           || router
                           || manager-to-router
                           || eliza-to-foo
                         ) )
end Example2
```

Here we have modeled the buffers `foo` and `bar` as one-place buffers only. We obtain for example the following execution trace. From line 9 onwards we have summarised the various internal steps by just counting them.

1. com. int-eliza-to-foo(('a' ++ hi-i-m-eliza-please-tell-me-your-problem))
2. com. int-eliza-to-foo(('a' ++ hi-i-m-eliza-please-tell-me-your-problem))
3. skip <0>
4. com. int-read-foo(('a' ++ hi-i-m-eliza-please-tell-me-your-problem))

5. com. int-manager-to-router(('1' ++ hi-i-m-eliza-please-tell-me-your-problem))
6. com. int-manager-to-router(('1' ++ hi-i-m-eliza-please-tell-me-your-problem))
7. atom read-router1(hi-i-m-eliza-please-tell-me-your-problem)
8. atom write-foo(('p' ++ i-do-not-like-my-computer-any-more))
9. (10 $\times$ int)
10. atom read-router1(do-computers-worry-you)
11. atom write-foo(('p' ++ no-no))
12. (1 $\times$ int)
13. atom write-foo(('p' ++ no-no))
14. (1 $\times$ int)
15. atom write-foo(('m' ++ enclosed-are-ten-dollars))
16. (1 $\times$ int)
17. atom write-foo(('p' ++ no-no))
18. (10 $\times$ int)
19. atom read-router1(are-you-saying-no-just-to-be-negative)

Note that the user does not get an answer for his problems on lines 11 and 13. As before, this is because the manager sees that the account is empty and does not pass the problems to Eliza. From the payment of line 15 onwards, the contact with Eliza is re-established.

This concludes our discussion of the practical aspects of relating lazy functional programming to process-algebraic specification and simulation. The next sections are devoted to a more theoretical analysis of the relation between the programs and its process equations.

10 Correctness aspects

Recall that if we want to see a Gofer program g as a process, we denote it as $\mathcal{P}[\![\mathrm{g}]\!]$. It would be nice if we could extract the ACP equations for $\mathcal{P}[\![\mathrm{g}]\!]$ from the Gofer program g. In general there are many process equations possible for the same functional behaviour. Finding the right ones demands that the rules of the lazy evaluation mechanism are taken into account. Our approach is similar to that of [5], but instead of giving a labeled transition system in one step, we separate the internals of Gofer (a TRS) from the external behaviour (an SOS with actions $s(x)$, $r(x)$ and τ). This approach of factoring the definition of the transition system into two steps is not new: it has been presented in [10] with a first set of rules called *operational rewrite rules* (a TRS) and a second set whose elements are called *transition rules.*

For the purpose of our present study we propose a language fragment. We only give the BNF rules but we assume type-correctness as usual.

```
<program>     ::= { <rule> }+
<rule>        ::= <pattern> = <term>
<pattern>     ::= <id>
                | <id> <termlist>
<term>        ::= "string"
                | []
                | ( <term> : <term> )
                | <id>
                | ( <id> <termlist> )
<termlist>    ::= <term> { , <term> }+
```

For a program given as $i_1\,\vec{t_1} = t'_1 \ldots i_m\,\vec{t_m} = t'_m$ we define the set funs $= \{i_1, \ldots, i_m\}$. For terms we define vars("string") $= \emptyset$, vars([]) $= \emptyset$, vars$((t_1 : t_2)) =$ vars$(t_1)\cup$vars(t_2), vars$(i) = \emptyset$ if $i \in$ funs, $\{i\}$, otherwise; vars$((i\,t_1, \ldots, t_n)) =$ vars$(t_1) \cup \ldots \cup$ vars(t_n). For patterns we define vars$(i) = \emptyset$ and vars$(i\,t_1, \ldots, t_n) =$ vars$(t_1) \cup \ldots \cup$ vars(t_n). For each rule $i\,\vec{t} = t'$ we demand vars$(t') \subseteq$ vars$(\vec{t})$ and $i \in$ funs where funs is taken for the whole program. More language features, like conditions etc. can be added easily later.

Now we define a structured operational semantics (SOS) which shall make the operational behaviour of the Gofer interpreter for a given g completely explicit. First we set out to define a term rewriting system (TRS) and a strategy.

We restrict ourselves to such g only which have type `[Line] -> [Line]`. Inside terms we allow for a special artificial subterm: a tail part of a list which is not available yet may be replaced by a placeholder, here resented as © (we prefer to reserve the symbol $\perp$ for a slightly different kind of analysis later). So we change the BNF rule for `term` as follows:

```
term ::= ©
       | "string"
       | []
       | ( <term> :  <term> )
       | <id>
       | ( <id> <termlist> )
```

We write $\rightarrow$ for the functional reduction relation of Gofer (choosing an outermost redex from the reductions β, π_1, π_2)[1]. We assume that the strategy is leftmost outermost (this strategy is well-known to be correct in the sense that a normal form will be found whenever it exists). Also when there is a choice between some redex ocurring in the head of a list and a redex occuring in the tail of that list, the one in the head is to be selected.

There may be several reductions applicable to the same redex (because the left-hand side patterns in the program can overlap). In principle, the first one of these must be chosen, but there is a complication. The binary test for match, say, when checking a concrete term c and the l.h.s. pattern p of a rule of the form $p = t'$ can have three possible outcomes: either match$(c, p) =$ **true**, match$(c, p) =$ **false**, or match$(c, p) =$ **dontknow** ("unknown"). In particular match(©, []) = **dontknow** match(©, "string") = **dontknow** and so is match(©, $(x : xs)$). But © does match a variable. It is understood that the definition of match follows the structure of the 'c' recursively so that a **dontknow** match on an internal subterm will lead to **dontknow** at the top-level unless of course there is a **false**, which overrules the **dontknow**. We resolve the complication as follows: if the current term is c then the i-th rule $p_i = t'_i$ is selected for firing if:

1. match$(c, p_i) =$ **true**, and
2. for all $j < i$ it holds that match$(c, p_j) =$ **false**.

The complication is demonstrated by the following program:

```
g ["foo"]  = ["goodbye"]
g (x : xs) = ["bye"]
```

[1]We define π_1 as the rule head $(x : xs) = x$ and π_2 as the rule tail $(x : xs) = xs$.

After input of `foo` the interpreter will wait for a second line. In our TRS this is explained because match(`(g ("foo" : ©))`,`(g ("foo" : []))`) = **dontknow**. The fact that the second rule has a match is not enough for making it fire: the patterns of all earlier rules for the same redex must yield a definite **false**.

Actually we can abstract away from some of the details of Gofer and in the definition of the semantics $\mathcal{P}[\![\mathrm{g}]\!]$ to be given below, only very few data about the programming language and its reduction mechanism are needed. We collect these data in a six-tuple.

Definition. For an identifier g we say that the six-tuple

$$\langle T, [\,], :, ©, [\,:=\,], \psi\rangle$$

is an *abstract functional program* for g if T is a set of open terms containing $[\,]$ which is closed under the binary operation ':', and where the set of variables must be taken equal to $\{©\}$. We require that $(\mathrm{g}\,©) \in T$. The ternary operation $[\,:=\,]$ takes a term and a variable and returns the result of substituting the third argument for the variable, as usual. Finally ψ must be a partial mapping on T, called the *rewrite function*, and it must satisfy the conditions:

1. $[\,] \notin \mathrm{dom}(\psi)$,
2. if $\psi(x) = x'$ then $\psi(x : xs) = (x' : xs)$ $\qquad (x, x' \in T)$,
3. if $© \notin c$ and $c \not\equiv [\,]$ and $c \not\equiv (x : xs)$ then $c \in \mathrm{dom}(\psi)$.

The second condition expresses that the strategy is leftmost with respect to list construction. The third condition expresses that we exclude programs which get stuck because no more reduction rule applies. □

We thank Jan Bergstra for the suggestion to introduce such notion. We denote equality on T by $\equiv$ and since T is a set of terms we may later use the fact that for no x, xs the equation $[\,] \equiv (x : xs)$ holds. Each correct Gofer program of the form proposed in Section 2 realises an abstract functional program, notably by adopting the ψ derived from the rules $\{p_i \mid i = 1, 2, , \ldots\}$ of the program where $c \in \mathrm{dom}(\psi)$ iff $\exists_i\, \mathrm{match}(c, p_i)$ = **true** and $\forall_{j<i}\, \mathrm{match}(c, p_j)$ = **false**. But of course an abstract functional program could be realised in another lazy functional programming language too.

Usually we write $c \rightarrow_\psi c'$ or even just $c \rightarrow c'$ instead of $\psi(c) = c'$. And we write $x \not\rightarrow_\psi$ or even just $x \not\rightarrow$ if $x \notin \mathrm{dom}(\psi)$. So $\rightarrow$ is a functional reduction relation, i.e. a TRS together with a reduction strategy. We write $c \not\rightarrow$ to mean $\neg\exists x' \cdot x \rightarrow x'$ and we write $c \equiv (x : xs)$ to mean that c is of the form $(x : xs)$ for suitably chosen x and xs.

Now we define a structured operational semantics (SOS), sometimes also called 'action relation'. We shall define a ternary action relation $\xrightarrow{\cdots}$ (note the long arrow) and a unary termination relation $\downarrow$. The 'c' below are the configurations. The $s(x)$ and $r(x)$ are the send- and receive actions, respectively. The rules below are organised as follows: above each line we give the conditions, which are concerned with the rewrite function $\rightarrow$. Below the line we give the axiom schema, which is about the action relation $\xrightarrow{\cdots}$ or the termination relation $\downarrow$.

$$\frac{c \equiv [\,]}{c\downarrow} \qquad \frac{\neg(c \equiv (x : xs) \;\wedge\; x \not\rightarrow) \quad c \rightarrow c'}{c \xrightarrow{\tau} c'}$$

$$\frac{\begin{array}{l} c \equiv (x : xs) \\ \copyright \notin x \\ x \not\rightarrow \end{array}}{c \xrightarrow{s(x)} xs} \qquad \frac{\begin{array}{l} c \equiv (x : xs) \\ \copyright \in x \\ x \not\rightarrow \end{array}}{c \xrightarrow{r(z)} c[\copyright := z : \copyright]} \qquad \frac{\begin{array}{l} c \not\equiv [\,] \\ c \not\equiv (y : ys) \\ c \not\rightarrow \end{array}}{c \xrightarrow{r(x)} c[\copyright := x : \copyright]}$$

The action relation $\xrightarrow{\cdots}$ is defined as the smallest relation satisfying the above rules (axiom-schemas) together with the following, which we call 'standard' rules (check [2] Table 11):

$$\frac{}{\epsilon\downarrow} \qquad \frac{\begin{array}{l} c_1\downarrow \\ c_2\downarrow \end{array}}{(c_1 \cdot c_2)\downarrow} \qquad \frac{\begin{array}{l} c_1\downarrow \\ c_2 \xrightarrow{a} c_2' \end{array}}{c_1 \cdot c_2 \xrightarrow{a} c_2'} \qquad \frac{c_1 \xrightarrow{a} c_1'}{c_1 \cdot c_2 \xrightarrow{a} c_1' \cdot c_2}$$

$$\frac{c_1\downarrow}{(c_1 + c_2)\downarrow} \qquad \frac{c_2\downarrow}{(c_1 + c_2)\downarrow} \qquad \frac{c_1 \xrightarrow{a} c_1'}{c_1 + c_2 \xrightarrow{a} c_1'} \qquad \frac{c_2 \xrightarrow{a} c_2'}{c_1 + c_2 \xrightarrow{a} c_2'}$$

So the actions for a process whose top-level Gofer function is g are found by following the action relation starting with the initial configuration c_0 given by:

$$c_0 = (\mathrm{g}\ \copyright)$$

We write SOS(g) for the triple $(c_0, \downarrow, \xrightarrow{\cdots})$. Following a suggestion of Jan Bergstra, we can make the process-semantics very explicit by means of a single equation. Define $\mathcal{P}[\![\mathrm{g}]\!] := \mathcal{P}[\![(\mathrm{g}\ \copyright)]\!]$, where

$$\begin{array}{rll} \mathcal{P}[\![c]\!] = & [c \equiv [\,]]\text{->} & \epsilon \\ & +\ [\neg(c \equiv (x : xs)\ \wedge\ x \not\rightarrow) \text{ and } c \rightarrow c']\text{->} & \tau \cdot \mathcal{P}[\![c']\!] \\ & +\ [c \equiv (x : xs) \text{ and } \copyright \notin x \text{ and } x \not\rightarrow]\text{->} & s(x) \cdot \mathcal{P}[\![xs]\!] \\ & +\ [c \equiv (x : xs) \text{ and } \copyright \in x \text{ and } x \not\rightarrow]\text{->} & \sum_z r(z) \cdot \mathcal{P}[\![c[\copyright := z : \copyright]]\!] \\ & +\ [c \not\equiv [\,] \text{ and } c \not\equiv (y : ys) \text{ and } c \not\rightarrow]\text{->} & \sum_x r(x) \cdot \mathcal{P}[\![c[\copyright := x : \copyright]]\!] \end{array}$$

We used the notation [condition]-> to denote guards. Please note that all guards describe syntactic conditions on terms, and do not involve any assumptions on the action relation itself. The conditions which occur in the SOS are visualised in Figure 4 below.

Now we set out to use the SOS to define a suitable equivalence on configurations. Suppose that we have a set of configurations C_{g} for a program g together with the relations $\downarrow_{\mathrm{g}}$ and $\xrightarrow{\cdots}_{\mathrm{g}}$. Let c_0 be the initial configuration (the 'root'). Then it may be the case that in a reactive environment we want to consider certain processes as being 'the same'. We define that a relation $R \subseteq C_{\mathrm{g}} \times C_{\mathrm{g}}$ is a rooted branching bisimulation if it satisfies:

1. $c_0 R c_0$
2. if $c \xrightarrow{a} c'$ and cRd, then either
   (a) $a = \tau$ and $c'Rd$, or
   (b) $\exists d_1, d' \cdot d \overset{\tau\cdots\tau}{\twoheadrightarrow} d_1 \xrightarrow{a} d' \wedge cRd_1 \wedge c'Rd'$
3. if $c\downarrow$ and cRd then $\exists d' \cdot d \overset{\tau\cdots\tau}{\twoheadrightarrow} d' \wedge d'\downarrow \wedge cRd'$
4. similarly when the roles of c and d are interchanged

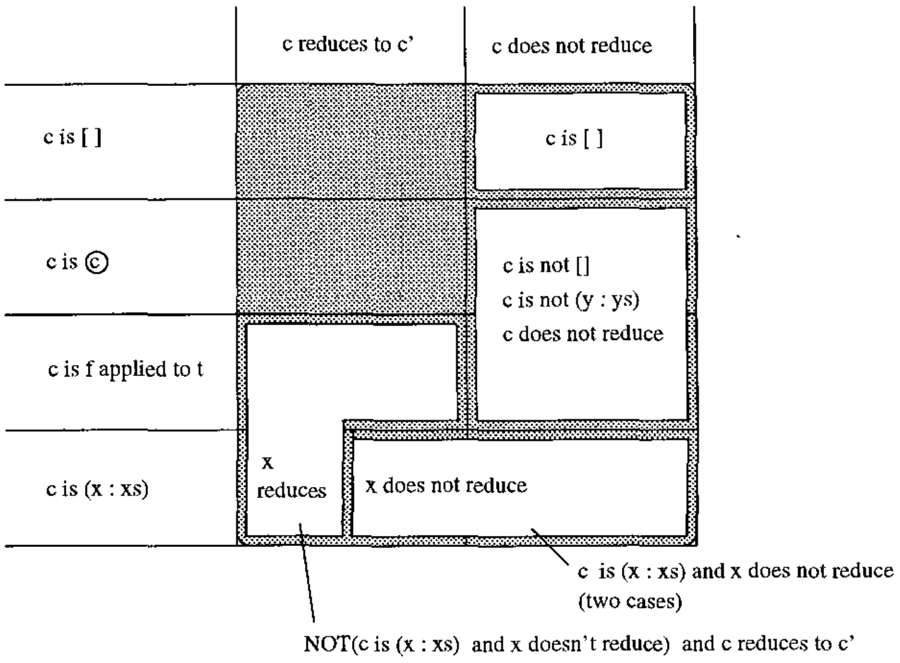

Figure 4: Disjointness of conditions occuring in SOS.

We define $\underline{\leftrightarrow}_{\mathrm{rb}}$ as the unique maximal rooted branching bisimulation relation.

Now we set out to to find the ACP equations which hold for $\underline{\leftrightarrow}_{\mathrm{rb}}$. Next to the usual axioms for $\mathrm{ACP}_\epsilon^\tau$ we introduce five more laws. They are in a one-one correspondence with the SOS axiom schemata. Let us call this set of laws EQ(g).

$$\frac{c \equiv [\,]}{c = \epsilon} \qquad \frac{\begin{array}{l}\neg(c \equiv (x : xs) \ \wedge\ x \not\rightarrow) \\ c \rightarrow c'\end{array}}{c = \tau \cdot c'}$$

$$\frac{\begin{array}{l}c \equiv (x : xs) \\ © \notin x \\ x \not\rightarrow\end{array}}{c = s(x) \cdot xs} \qquad \frac{\begin{array}{l}c \equiv (x : xs) \\ © \in x \\ x \not\rightarrow\end{array}}{c = \sum_z r(z) \cdot c[© := z : ©]} \qquad \frac{\begin{array}{l}c \not\equiv [\,] \\ c \not\equiv (y : ys) \\ c \not\rightarrow\end{array}}{c = \sum_x r(x) \cdot c[© := x : ©]}$$

There is a complication related to possible non-terminating rewriting, which however is easily remedied. If we want to apply these laws to a program which can engage in an infinite rewriting process, we have to use them in a slightly different way: instead of τ, we have to use a special atomic action, say I; if this leads to certain equations describing the configuration c, then the process is specified by $\tau_{\{I\}}(c)$, that is the process in which all I steps are renamed to τ (of course there are certain contexts in which Koomen's fair abstraction rule can be applied and then an infinite sequence of τ steps turns into δ and then disappears).

Theorem (Soundness). Consider an abstract functional program $\langle T, [\,], :, ©, [\,:=\,], \psi\rangle$ for g, then

$$\text{if } \mathrm{ACP}_\epsilon^\tau + \mathrm{EQ(g)} \vdash c = d \quad \text{then} \quad c \underline{\leftrightarrow}_{\mathrm{rb}}\, d$$

Proof. We check the four axiom schemas of EQ(g) first. The key observation is that the conditions for the rules are mutually disjoint. For example consider the first axiom (about ε) whose condition is $c \equiv [\,]$. By definition of the notion of abstract functional program, $c \equiv [\,]$ excludes $c \rightarrow c'$. That it excludes the other conditions is obvious.

In order to check the first axiom, we assume that $c \equiv [\,]$ and we must show that $c \leftrightarrow_{\mathrm{rb}} \varepsilon$. By the standard rules, we know that $\varepsilon\downarrow$. We claim that any (rb) bisimulation R can be extended to $R \cup \{([\,], \varepsilon)\}$, which then still is a (rb) bisimulation. So we must check the four clauses of the definition of $\leftrightarrow_{\mathrm{rb}}$. Clause 1. holds because R is a (rb) bisimulation. Clause 2 does not apply (disjointness of conditions!). Clause 3 must be checked because $[\,]\downarrow$ and because $[\,](R \cup \{([\,], \varepsilon)\})\varepsilon$. We must indicate d' such that $\varepsilon \xrightarrow{\tau \ldots \tau} d' \wedge d'\downarrow \wedge ([\,], d') \in R \cup \{([\,], \varepsilon)\}$. Choose $d' = \varepsilon$, taking zero τ steps. Since $\leftrightarrow_{\mathrm{rb}}$ is the *maximal* bisimulation, we conclude that $c \equiv [\,] \leftrightarrow_{\mathrm{rb}} \varepsilon$.

Checking the other axiom schemas can be done along the same lines. We only add one remark for the last axiom. In general, from $c \xrightarrow{r(x)} c[© := x : ©]$ we can only deduce that $c = r(x) \cdot c[© := x : ©] + \ldots$ (some summand). But because there are no other action triples for such c except those obtained by taking all values of x (which does not occur in the conditions), we know that there are no other summands.

For the soundness of the laws of $\mathrm{ACP}_{\varepsilon}^{\tau}$ we refer to [2], Theorem 5.4.19. □

We can use this to verify the ACP equations for the program patterns generated by the generator of Section 7. We show this for one of the options.

Lemma. For the program g generated as option 1 by the generator of Section 7,

$$\leftrightarrow_{\mathrm{rb}} \;\models\; G = \sum_x r(x) \cdot s(f_1(x)) \cdot G$$

Proof. We start from the Gofer program for g which is `g (x : xs) = (f1 x) : (g xs)` and we assume that the `f1` implements a function f_1 which maps lines to lines.

For the initial configuration, $G = (\mathrm{g}\ ©)$. This term satisfies the condition of the r-law of EQ(g) and therefore $G = \sum_x r(x) \cdot (\mathrm{g}\,©)[© := x : ©] \equiv \sum_x r(x) \cdot (\mathrm{g}\,(x : ©))$. The latter term can be rewritten since $(\mathrm{g}\ (x : ©)) \rightarrow (\mathtt{f1}\ x) : (\mathrm{g}\,©) \equiv (\mathtt{f1}\ x) : G \rightarrow f_1(x) : G$ (where we used the definitions of g and `f1`, respectively). For each rewrite step, the τ-law of EQ(g) applies and thus:

$$G = \sum_x r(x) \cdot \tau \cdot \tau \cdot s(f_1(x)) \cdot G$$

The τ steps are removable because of the usual τ law of $\mathrm{ACP}_{\varepsilon}^{\tau}$ which says that $a \cdot \tau = a$. Finally the soundness theorem can be applied. □

11 Semantic aspects

One of the main advantages of functional programming languages which is often put forward is that all programs denote true mathematical functions. Sometimes this is explained by saying that 'referential transparancy' holds. This means that two subprograms denoting equal mathematical objects can be substituted one for another without affecting the meaning of the program as a whole. In this section we define a denotational semantics for a fragment of Gofer. Then this can be compared with the SOS-based semantics given in the previous section.

Recall that if we want to consider g as a mathematical function, we denote it as $\mathcal{F}[\![\mathrm{g}]\!]$. We must define the meaning function $\mathcal{F}[\![\ \ ]\!]$. We begin with the definition of a suitable semantic domain, following [4]. The idea is as follows: a program transforms a sequence of input strings to a sequence of output strings, so as a first approximation one would expect that the semantic domain is the set String* of all finite sequences of strings. However we have to add two more kinds of strings:

- Strings which are incomplete in the sense that only a finite prefix is given; these are needed since we know that a program sometimes produces already output when only an incomplete input has been consumed. Also the output may or may not be complete.
- Strings which are infinite; these are needed since we know that a program can go on accepting input and producing output forever.

We start with some definitions. For a set V we identify $\underbrace{V \times (\cdots (V \times V) \cdots)}_{n \text{ times}}$ with V^n. And $V^* = \bigcup_{n \in \mathbb{N}} V^n$. So in this section we write String* instead of [String]. Finally V^∞ denotes the function space $\mathbb{N} \to V$. We order the set String $\cup \{\perp\}$ by $\sqsubseteq$ by postulating that for $x, y \in$ String $\cup \{\perp\}$: $x \sqsubseteq y \;:\Leftrightarrow\; (x = \perp) \vee (x = y)$. We write STR(String) for the set:

$$(\text{String}^* \times \{\perp\}) \cup \text{String}^* \cup \text{String}^\infty$$

An element of STR(String) is called a *stream*. Most often we just write STR for STR(String), We order the set STR by $\sqsubseteq$ by postulating that for $s, s' \in$ STR and thus $s_0, s_0', s_1, s_1', \ldots \in$ String $\cup \{\perp\}$ (here s_k denotes the k-th element of s):

$$s \sqsubseteq s' \;:\Leftrightarrow\; (\exists n \in \mathbb{N} \cdot [s_0..s_{n-1}] = [s_0'..s_{n-1}'] \wedge s_n = \perp) \vee (s = s')$$

The intuition is that for example ["a", "b", $\perp$] is an approximation of ["a", "b", "c", "d"]. We write $\bigsqcup D$ for the least upperbound (l.u.b.) of a set $D \subseteq$ STR, if it exists.

In [4] it is shown that (STR, $\sqsubseteq$) forms a *countably algebraic domain* by which it is meant that the following properties hold:

- the set of streams has a least element,
- for every directed set of streams D the l.u.b. $\bigsqcup D$ exists,
- the set of finite approximations of a stream s is directed and every stream s is the l.u.b. of the set of its finite approximations: $s = \bigsqcup_{t \sqsubseteq s \,\wedge\, t \text{ finite}} t$,
- *the set of finite elements (the s such that for all directed D we have $s \sqsubseteq \bigsqcup D \Rightarrow \exists t \in D \cdot s \sqsubseteq t$) is countable.*

The least stream is $[\perp]$. The finite elements are the streams not in String$^\infty$.

A structure $(S, \sqsubseteq)$ for which only the properties of the first two items hold is called a *complete partial order* (CPO). If we have two CPOs it is always possible to construct a product CPO, ordering pairs by $(x, y) \sqsubseteq (x', y') :\Leftrightarrow (x \sqsubseteq x') \wedge (y \sqsubseteq y')$.

We extend the sequence constructor function ':' whose type was String $\times$ String* $\to$ String* to a function ':' of type (String $\cup \{\perp\}) \times$ STR $\to$ STR as follows. Let it be understood that $x \neq \perp$ and also $s_0, s_1, \ldots, s_{n-1} \neq \perp$.

- $x : [s_0, s_1, \ldots, s_{n-1}] = [x, s_0, s_1, \ldots, s_{n-1}]$ (as before),
- $\perp : [s_0, s_1, \ldots, s_{n-1}] = [\perp]$,
- $x : [s_0, s_1, \ldots, s_{n-1}, \perp] = [x, s_0, s_1, \ldots, s_{n-1}, \perp]$,
- $\perp : [s_0, s_1, \ldots, s_{n-1}, \perp] = [\perp]$,

- $x : [s_0, s_1, \ldots(\infty)] = [x, s_0, s_1, \ldots\infty]$,
- $\perp : [s_0, s_1, \ldots(\infty)] = [\perp]$.

Note that ':' is monotonic in both arguments, that is, for $x, y \in (\text{String} \cup \{\perp\})$, $s, s' \in \text{STR}$, $x \sqsubseteq y \Rightarrow (x : s) \sqsubseteq (y : s)$ and $s \sqsubseteq s' \Rightarrow (x : s) \sqsubseteq (x : s')$. It is also strict in its first argument but not in its second.

We extend the equality predicate '==' whose type was String $\times$ String $\to$ Bool to a function '==' of type (String $\cup \{\perp\}$) $\times$ (String $\cup \{\perp\}$) $\to$ (Bool $\cup \{\perp\}$). Let it be understood that $x, y \neq \perp$.

- $(x == y) = \text{True if } x = y$ (as before),
- $(x == y) = \text{False if } x \neq y$ (as before),
- $(\perp == x) = (x == \perp) = (\perp == \perp) = \perp$.

Note that '==' is monotonic in both arguments, that is, for $x, y, z \in (\text{String} \cup \{\perp\})$, $x \sqsubseteq y \Rightarrow (x == z) \sqsubseteq (y == z)$ and $x \sqsubseteq y \Rightarrow (z == x) \sqsubseteq (z == y)$. It is also strict.

We make `head` and `tail` total by putting `head`([ ]) = `head`([$\perp$]) = $\perp$ and `tail`([ ]) = `tail`([$\perp$]) = [$\perp$]. Of course this is not what happens in reality; it is truthful however if we assume that the programmer makes sure that the program will not try to take the `head` or `tail` of empty sequences.

We need special if-then-else operators of type (Bool $\cup \{\perp\}$) $\times$ (String $\cup \{\perp\}$) $\times$ (String $\cup \{\perp\}$) $\to$ (String $\cup \{\perp\}$) and of type (Bool $\cup \{\perp\}$) $\times$ STR $\times$ STR $\to$ STR. We define that:

- (if True then a else b) $= a$,
- (if False then a else b) $= b$,
- (if $\perp$ then a else b) $= \perp$, ([$\perp$], respectively).

It is easy to see that if-then-else is monotonic in all three arguments; it is strict in its condition argument but not in its 2nd and 3d argument.

If we have a CPO $(S, \sqsubseteq)$ then we can order the function space $S \to S$ by letting $f_1 \sqsubseteq f_2 :\Leftrightarrow \forall x \in S \cdot f_1(x) \sqsubseteq f_2(x)$. A function f is *continuous* if for all directed $D \subseteq S$ we have $f(\bigsqcup D) = \bigsqcup f(D)$. In [11] Cor. 1.2.7 it is shown that continuity implies monotonicity. An element $x \in S$ is a *fixed point* of f if $f(x) = x$. By [11] Thm. 1.2.17 each continuous f has a least fixed point $\textbf{Fix}(f) = \bigsqcup_n f^n(\perp)$ where $\perp$ is the least element of the CPO.

Lemma. The following operations are continuous in all argument positions:

(i) ':', `head`, `tail`,
(ii) '==',
(iii) if-then-else,
(iv) λ-abstraction,
(v) function application.

Proof. For (i), (iv) and (v) we can refer to the literature.

(i) See [4], Lemma of Section 2.3.

(ii) We show $(\bigsqcup D == z) = \bigsqcup\{x == z \mid x \in D\}$ for directed D. Note that $|D| \leq 2$. If $z = \bot$ we check $\bot = \bigsqcup\{\bot\}$. Otherwise let $z \neq \bot$. Two cases arise: either (a) $\bigsqcup D = \bot$ so $D = \{\bot\}$ whence $(\bigsqcup D == z) = \bot = \bigsqcup\{\bot\} = \bigsqcup\{\bot == z\}$ (end of a), or (b) $\bigsqcup D \neq \bot$ so for some $x \neq \bot$ we find $D = \{x, \ldots\}$. Now if $x = z$ then $(\bigsqcup D == z) = \text{True} = \bigsqcup\{\text{True}, \ldots\} = \bigsqcup\{x == z, \ldots\}$. And if $x \neq z$ then $(\bigsqcup D == z) = \text{False} = \bigsqcup\{\text{False}, \ldots\} = \bigsqcup\{x == z, \ldots\}$.

(iii) Analogously.

(iv) See [11] 1.2.13.

(v) See [11] 1.2.14. □

Now we turn our attention to Gofer programs again. For a Gofer program g, $\mathcal{F}[\![\text{g}]\!]$ will be a (partial) function $\mathcal{F}[\![\text{g}]\!]$: STR $\to$ STR. For purposes of giving semantics we prefer pure λ-terms. Therefore we show how to eleminate patterns and conditions in favour of `head`, `tail` operations and if-then-else operators, respectively. We eliminate conditions by application of the following rule:

$$\begin{array}{lll} \text{g}\ \vec{x} \mid (p\ \vec{x}) \quad = \text{rhs}_1 & \text{becomes} & \text{g} = \lambda\vec{x}.\ \text{if}\ (p\ \vec{x}) \\ \quad\ \mid \text{otherwise} \quad = \text{rhs}_2 & & \qquad\quad \text{then rhs}_1 \\ & & \qquad\quad \text{else rhs}_2 \end{array}$$

And we eliminate patterns by application of the following rule (if there are more patersn, the if-then-else follows the order in which they are given):

$$\begin{array}{lll} \text{g}\ [\,] \qquad\ = \text{rhs}_1 & \text{becomes} & \text{g} = \lambda z.\ \text{if}\ z == [\,] \\ \text{g}\ (x : xs) = \text{rhs}_2 & & \qquad\ \text{then rhs}_1 \\ & & \qquad\ \text{else if}\ \mathtt{head}(z) == \mathtt{head}(z) \\ & & \qquad\qquad \text{then rhs}_2[x := \mathtt{head}(z), xs := \mathtt{tail}(z)] \\ & & \qquad\qquad \text{else}\ [\bot] \end{array}$$

After these transformations, each function definition, like g has become an equation $\text{g} = \lambda\vec{x}.\,\text{expr}(\text{g}, \vec{x})$. The expression 'expr' in the r.h.s. contains g and thus $\lambda\text{g}.\,\lambda\vec{x}.\,\text{expr}(\text{g}, \vec{x})$ can be viewed as the description of a functional[2]. This functional maps each function g (of the right type) to another function $\lambda\vec{x}.\,\text{expr}(g, \vec{x})$. Let F_{g} be this functional. Define $\mathcal{F}[\![\text{g}]\!] := \mathbf{Fix}(F_{\text{g}})$ (which exists by [11] Thm. 1.2.17, stating that each continuous F has a least fixed point, because F_{g} is continuous as follows from our lemma and the fact that it is defined using the operations ':', `head`, `tail`, '==', if-then-else, λ-abstraction and function application, exclusively).

Now we can compare the denotational semantics with the SOS-based semantics. We give a negative result first.

Fallacy. There is no translation function 'trans' such that for all g:

$$\text{trans} : \mathcal{F}[\![\text{g}]\!] \mapsto \mathcal{P}[\![\text{g}]\!].$$

Counter example. Define two programs g1 and g2 as follows:

```
g1 [] = g1 []                                 g2 [] = g2 []
g1 (x : xs) = g1 (x : xs)                     g2 (x : xs) = g2 xs
```

[2]A functional is a function which maps functions to functions

Then for $i \in \{1,2\}$ we have $\forall \vec{x} \in \mathrm{STR} \cdot \mathcal{F}[\![\mathrm{g}i]\!] = [\bot]$. But g1 performs an infinite rewriting without consuming input (but the first line), whereas g2 consumes all input (still producing nothing). So $\mathcal{P}[\![\mathrm{g1}]\!] = \sum_x r(x) \cdot \tau^\omega$ whereas $\mathcal{P}[\![\mathrm{g2}]\!] = \sum_x r(x) \cdot G_2$. □

The syntactic forms of the definitions of **g1** and **g2** contain clues about the rewriting process and the precise points in time when input is needed. These clues are lost when considering the mathematical semantics alone. The full advantage of referential transparancy is not applicable. Not all is lost however: there is a positive result too.

Proposition. For all g, let $\mathrm{SOS(g)} = (c_0, \downarrow, \overset{\dots}{\longrightarrow})$. Then for all traces $\vec{t} \in \{s(x), r(x), \tau\}^*$, defining $\overset{\vec{t}}{\twoheadrightarrow}$ in the obvious way:

$$\text{if } \exists c \cdot c_0 \overset{\vec{t}}{\twoheadrightarrow} c \text{ then } (\mathcal{F}[\![\mathrm{g}]\!](\vec{t}|_r +\!\!+ [\bot]))|_{\text{non-}\bot} = \vec{t}|_s$$

where ++ denotes sequence concatenation and where $\vec{t}|_r$ is the sequence of values (x) occuring in $r(x)$ values in $\vec{t}$ and where $|_s$ is defined similarly. The operator $|_{\text{non-}\bot}$ removes $\bot$ from streams.

Proof. We show the essential details for an example. Consider the program g and the dialogue of Section 4. We write 'l' for '`let us explain the compiler`', 'L' for '`LET US EXPLAIN THE COMPILER`', and so on. We shall consider the trace $\vec{t} = [r(\text{'l'}), \tau, \tau, s(\text{'L'}), r(\text{'i'}), \tau, \tau, s(\text{'I'})]$ for which $\vec{t}|_r = [\text{'l'}, \text{'i'}]$ and $\vec{t}|_s = [\text{'L'}, \text{'I'}]$.

First we analyse the operational behaviour, for which we rewrite according to the program's TRS and apply the SOS rules. (g ©) $\xrightarrow{r(\text{'l'})}$ (g ('l' : ©)) $\xrightarrow{\tau}$ (`updline` 'l') : (g ©)) $\xrightarrow{\tau}$ 'L' : (g ©)) $\xrightarrow{s(\text{'L'})}$ (g ©) $\xrightarrow{r(\text{'i'})}\xrightarrow{\tau}\xrightarrow{\tau}\xrightarrow{s(\text{'I'})}$ (g ©). This shows that the premiss of our proposition holds, that is, $\exists c \cdot c_0 \overset{\vec{t}}{\twoheadrightarrow} c$.

Now we turn to the denotational semantics. By elimination of the auxiliary function `updlines` and by elimination of all patterns from the left-hand side of its definition, g can be transformed and then we find that $\mathcal{F}[\![\mathrm{g}]\!]$ is the least function g satisfying the following fixed point equation:

```
$g = \lambda z.$ if $z == [\,]$
          then $[\,]$
          else if head($z$) == head($z$)
                then (updline head($z$)) : ($g$ tail($z$))
                else $[\bot]$
```

We claim that $\mathcal{F}[\![\mathrm{g}]\!]([\text{'l'}, \text{'i'}, \bot]) = [\text{'L'}, \text{'I'}, \bot]$. The claim follows by noting first that for any g satisfying the above fixed point equation we can perform the following calculation:

```
$g$ ([ 'l', 'i', $\bot$]) = if $z == [\,]$
                   then $[\,]$
                   else if 'l' == 'l'
                         then 'L' : $g$(['i', $\bot$])
                         else $[\bot]$
                = 'L' : ($g$ ['i', $\bot$])
                = 'L' : ('i' : $g$([$\bot$]))
                = ...
```

Moreover if g is the *least* solution, then $g([\bot]) = [\bot]$, so we continue:

```
... = 'L' : ('I' : [$\bot$])
    = ['L', 'I', $\bot$]
```

From the claim, $(\mathcal{F}[\![g]\!](\vec{t}\,|_r{+}{+}[\bot]))|_{\text{non-}\bot} = g([\text{`}\texttt{l}\text{'},\text{`}\texttt{i}\text{'},\bot])|_{\text{non-}\bot} = [\text{`}\texttt{L}\text{'},\text{`}\texttt{I}\text{'},\bot]|_{\text{non-}\bot} = [\text{`}\texttt{L}\text{'},\text{`}\texttt{I}\text{'}] = [r(\text{`}\texttt{l}\text{'}),\tau,\tau,s(\text{`}\texttt{L}\text{'}),r(\text{`}\texttt{i}\text{'}),\tau,\tau,s(\text{`}\texttt{I}\text{'})]|_s = \vec{t}\,|_s$, as was to be shown. □

This is the consistency property we mentioned for several of our introductory example programs. A survey of the theory developed so far is given in Figure 5 below.

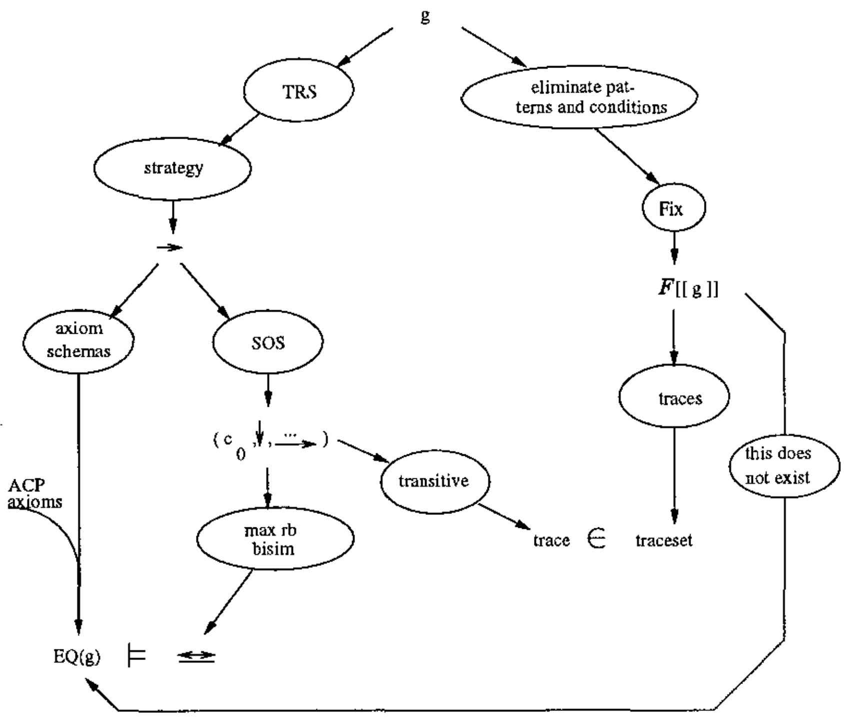

Figure 5: Survey of the theory.

12 Concluding remarks

Before we can conclude we must explain one technical point. We decided to consider a 'line' as the smallest unit of data which is consumed or produced in one step. In reality, a Gofer interpreter can do the lazy evaluation on a character-basis. For most of our example program this makes no difference, since the program needs the entire line anyhow before it is able to proceed. However in general this is not true. But one can always enforce our assumptions by means of a simple trick: replace the standard function `lines` by a local version `lines'` which is defined as `(map (reverse . reverse)) . lines`. This works because the `reverse` function demands its argument to be completed before it can reverse it (the lazy execution mechanism is is not smart enough to 'see' that the double reversal commutes with `lines`).

Looking back, we have covered both the practical aspects for which we proposed a generator-based approach for important program patterns, and certain theoretical aspects when we analysed the correctness conditions and some important semantical issues. We showed that the strong points of modern functional programming and algebraic concurrency analysis can be combined, but that there are a number of subtle points which have to be addressed. More precisely we are able to provide correct ACP equations for typical program patterns and we showed that the operational behaviour is only partially compatible with a functional interpretation. Although the generator, the multitasking experiment and the PSF simulation as developed so far are very small, this is not because of any fundamental limitation. On the contrary, there is evidence that the approach can be made operational for much complexer systems too.

Acknowledgements: The author wishes to thank Lex Augusteijn, Jan Bergstra, Gert Geurts and Michel Reniers, for the discussions and the help that contributed to the work presented in this paper.

References

[1] M.P. Jones. *An introduction to Gofer*, Draft report, available at http://lal.cs.byu.edu/cs532/gofer/docs/goferdoc/goferdoc.html (1991).

[2] J.C.M. Baeten, W.P. Weijland. *Process algebra*, Cambridge Tracts in Theoretical Computer Science 18, Cambridge University Press (1990).

[3] S. Mauw, G.J. Veltink (Eds.). *Algebraic specification of communication protocols*, Cambridge Tracts in Theoretical Computer Science 36, Cambridge University Press (1993).

[4] M. Broy. *Extensional behaviour of concurrent, nondeterministic, communicating systems*, in: M. Broy (Ed.), Control flow and data flow: concepts of distributed programming, NATO ASI Series Vol. F14, Springer Verlag (1985).

[5] A.D. Gordon. *An operational semantics for I/O in a lazy functional language*, in: Conference on functional programming languages and computer architecture, Copenhagen 1993, pp. 136–145, ACM Press, (1993).

[6] P. Achten, J. van Groningen, R. Plasmeijer. *High level specification of I/O in functional languages*, in: Functional programming Glasgow 1992, J. Launchbury, P. Sansom (Eds.), pp. 1–17, Springer Verlag (1992).

[7] J.A. Bergstra, J.W. Klop. Process algebra for synchronous communication, Inf. & Control 60, pp. 109–137 (1984).

[8] R. Milner. *A calculus of communicating systems*, LNCS 92, Springer Verlag (1980).

[9] H.B.M. Jonkers. *An Overview of the SPRINT Method*, Proceedings of Formal Method Europe (FME) '93, J. Woodcock et. al. (Eds.) (1993).

[10] M.W. Mislove, F.J. Oles. *A simple language supporting angelic nondeterminism and parallel composition*, in: Mathematical foundations of programming semantics, Brookes et. al. (Eds.), LNCS 598.

[11] H.P. Barendregt. *The lambda calculus, its syntax and semantics*, revised edition, North-Holland (1984).

93/31	W. Körver	Derivation of delay insensitive and speed independent CMOS circuits, using directed commands and production rule sets, p. 40.
93/32	H. ten Eikelder and H. van Geldrop	On the Correctness of some Algorithms to generate Finite Automata for Regular Expressions, p. 17.
93/33	L. Loyens and J. Moonen	ILIAS, a sequential language for parallel matrix computations, p. 20.
93/34	J.C.M. Baeten and J.A. Bergstra	Real Time Process Algebra with Infinitesimals, p.39.
93/35	W. Ferrer and P. Severi	Abstract Reduction and Topology, p. 28.
93/36	J.C.M. Baeten and J.A. Bergstra	Non Interleaving Process Algebra, p. 17.
93/37	J. Brunekreef J-P. Katoen R. Koymans S. Mauw	Design and Analysis of Dynamic Leader Election Protocols in Broadcast Networks, p. 73.
93/38	C. Verhoef	A general conservative extension theorem in process algebra, p. 17.
93/39	W.P.M. Nuijten E.H.L. Aarts D.A.A. van Erp Taalman Kip K.M. van Hee	Job Shop Scheduling by Constraint Satisfaction, p. 22.
93/40	P.D.V. van der Stok M.M.M.P.J. Claessen D. Alstein	A Hierarchical Membership Protocol for Synchronous Distributed Systems, p. 43.
93/41	A. Bijlsma	Temporal operators viewed as predicate transformers, p. 11.
93/42	P.M.P. Rambags	Automatic Verification of Regular Protocols in P/T Nets, p. 23.
93/43	B.W. Watson	A taxomomy of finite automata construction algorithms, p. 87.
93/44	B.W. Watson	A taxonomy of finite automata minimization algorithms, p. 23.
93/45	E.J. Luit J.M.M. Martin	A precise clock synchronization protocol,p.
93/46	T. Kloks D. Kratsch J. Spinrad	Treewidth and Patwidth of Cocomparability graphs of Bounded Dimension, p. 14.
93/47	W. v.d. Aalst P. De Bra G.J. Houben Y. Kornatzky	Browsing Semantics in the "Tower" Model, p. 19.
93/48	R. Gerth	Verifying Sequentially Consistent Memory using Interface Refinement, p. 20.
94/01	P. America M. van der Kammen R.P. Nederpelt O.S. van Roosmalen H.C.M. de Swart	The object-oriented paradigm, p. 28.
94/02	F. Kamareddine R.P. Nederpelt	Canonical typing and Π-conversion, p. 51.
94/03	L.B. Hartman K.M. van Hee	Application of Marcov Decision Processe to Search Problems, p. 21.
94/04	J.C.M. Baeten J.A. Bergstra	Graph Isomorphism Models for Non Interleaving Process Algebra, p. 18.
94/05	P. Zhou J. Hooman	Formal Specification and Compositional Verification of an Atomic Broadcast Protocol, p. 22.
94/06	T. Basten T. Kunz J. Black M. Coffin D. Taylor	Time and the Order of Abstract Events in Distributed Computations, p. 29.
94/07	K.R. Apt R. Bol	Logic Programming and Negation: A Survey, p. 62.
94/08	O.S. van Roosmalen	A Hierarchical Diagrammatic Representation of Class Structure, p. 22.
94/09	J.C.M. Baeten J.A. Bergstra	Process Algebra with Partial Choice, p. 16.

94/10	T. verhoeff	The testing Paradigm Applied to Network Structure. p. 31.
94/11	J. Peleska C. Huizing C. Petersohn	A Comparison of Ward & Mellor's Transformation Schema with State- & Activitycharts, p. 30.
94/12	T. Kloks D. Kratsch H. Müller	Dominoes, p. 14.
94/13	R. Seljée	A New Method for Integrity Constraint checking in Deductive Databases, p. 34.
94/14	W. Peremans	Ups and Downs of Type Theory, p. 9.
94/15	R.J.M. Vaessens E.H.L. Aarts J.K. Lenstra	Job Shop Scheduling by Local Search, p. 21.
94/16	R.C. Backhouse H. Doornbos	Mathematical Induction Made Calculational, p. 36.
94/17	S. Mauw M.A. Reniers	An Algebraic Semantics of Basic Message Sequence Charts, p. 9.
94/18	F. Kamareddine R. Nederpelt	Refining Reduction in the Lambda Calculus, p. 15.
94/19	B.W. Watson	The performance of single-keyword and multiple-keyword pattern matching algorithms, p. 46.
94/20	R. Bloo F. Kamareddine R. Nederpelt	Beyond β-Reduction in Church's $\lambda\!\rightarrow$, p. 22.
94/21	B.W. Watson	An introduction to the Fire engine: A C++ toolkit for Finite automata and Regular Expressions.
94/22	B.W. Watson	The design and implementation of the FIRE engine: A C++ toolkit for Finite automata and regular Expressions.
94/23	S. Mauw and M.A. Reniers	An algebraic semantics of Message Sequence Charts, p. 43.
94/24	D. Dams O. Grumberg R. Gerth	Abstract Interpretation of Reactive Systems: Abstractions Preserving $\forall$CTL*, $\exists$CTL* and CTL*, p. 28.
94/25	T. Kloks	$K_{1,3}$-free and W_4-free graphs, p. 10.
94/26	R.R. Hoogerwoord	On the foundations of functional programming: a programmer's point of view, p. 54.
94/27	S. Mauw and H. Mulder	Regularity of BPA-Systems is Decidable, p. 14.
94/28	C.W.A.M. van Overveld M. Verhoeven	Stars or Stripes: a comparative study of finite and transfinite techniques for surface modelling, p. 20.
94/29	J. Hooman	Correctness of Real Time Systems by Construction, p. 22.
94/30	J.C.M. Baeten J.A. Bergstra Gh. Ştefanescu	Process Algebra with Feedback, p. 22.
94/31	B.W. Watson R.E. Watson	A Boyer-Moore type algorithm for regular expression pattern matching, p. 22.
94/32	J.J. Vereijken	Fischer's Protocol in Timed Process Algebra, p. 38.
94/33	T. Laan	A formalization of the Ramified Type Theory, p.40.
94/34	R. Bloo F. Kamareddine R. Nederpelt	The Barendregt Cube with Definitions and Generalised Reduction, p. 37.
94/35	J.C.M. Baeten S. Mauw	Delayed choice: an operator for joining Message Sequence Charts, p. 15.
94/36	F. Kamareddine R. Nederpelt	Canonical typing and Π-conversion in the Barendregt Cube, p. 19.
94/37	T. Basten R. Bol M. Voorhoeve	Simulating and Analyzing Railway Interlockings in ExSpect, p. 30.
94/38	A. Bijlsma C.S. Scholten	Point-free substitution, p. 10.

94/39	A. Blokhuis, T. Kloks	On the equivalence covering number of splitgraphs, p. 4.
94/40	D. Alstein	Distributed Consensus and Hard Real-Time Systems, p. 34.
94/41	T. Kloks, D. Kratsch	Computing a perfect edge without vertex elimination ordering of a chordal bipartite graph, p. 6.
94/42	J. Engelfriet, J.J. Vereijken	Concatenation of Graphs, p. 7.
94/43	R.C. Backhouse, M. Bijsterveld	Category Theory as Coherently Constructive Lattice Theory: An Illustration, p. 35.
94/44	E. Brinksma, J. Davies, R. Gerth, S. Graf, W. Janssen, B. Jonsson, S. Katz, G. Lowe, M. Poel, A. Pnueli, C. Rump, J. Zwiers	Verifying Sequentially Consistent Memory, p. 160
94/45	G.J. Houben	Tutorial voor de ExSpect-bibliotheek voor "Administratieve Logistiek", p. 43.
94/46	R. Bloo, F. Kamareddine, R. Nederpelt	The λ-cube with classes of terms modulo conversion, p. 16.
94/47	R. Bloo, F. Kamareddine, R. Nederpelt	On Π-conversion in Type Theory, p. 12.
94/48	Mathematics of Program Construction Group	Fixed-Point Calculus, p. 11.
94/49	J.C.M. Baeten, J.A. Bergstra	Process Algebra with Propositional Signals, p. 25.
94/50	H. Geuvers	A short and flexible proof of Strong Normalazation for the Calculus of Constructions, p. 27.
94/51	T. Kloks, D. Kratsch, H. Müller	Listing simplicial vertices and recognizing diamond-free graphs, p. 4.
94/52	W. Penczek, R. Kuiper	Traces and Logic, p. 81
94/53	R. Gerth, R. Kuiper, D. Peled, W. Penczek	A Partial Order Approach to Branching Time Logic Model Checking, p. 20.
95/01	J.J. Lukkien	The Construction of a small CommunicationLibrary, p.16.
95/02	M. Bezem, R. Bol, J.F. Groote	Formalizing Process Algebraic Verifications in the Calculus of Constructions, p.49.
95/03	J.C.M. Baeten, C. Verhoef	Concrete process algebra, p. 134.
95/04	J. Hidders	An Isotopic Invariant for Planar Drawings of Connected Planar Graphs, p. 9.
95/05	P. Severi	A Type Inference Algorithm for Pure Type Systems, p.20.
95/06	T.W.M. Vossen, M.G.A. Verhoeven, H.M.M. ten Eikelder, E.H.L. Aarts	A Quantitative Analysis of Iterated Local Search, p.23.
95/07	G.A.M. de Bruyn, O.S. van Roosmalen	Drawing Execution Graphs by Parsing, p. 10.
95/08	R. Bloo	Preservation of Strong Normalisation for Explicit Substitution, p. 12.
95/09	J.C.M. Baeten, J.A. Bergstra	Discrete Time Process Algebra, p. 20
95/10	R.C. Backhouse, R. Verhoeven, O. Weber	Math∫pad: A System for On-Line Prepararation of Mathematical Documents, p. 15

95/11	R. Seljée	Deductive Database Systems and integrity constraint checking, p. 36.
95/12	S. Mauw and M. Reniers	Empty Interworkings and Refinement Semantics of Interworkings Revised, p. 19.
95/13	B.W. Watson and G. Zwaan	A taxonomy of sublinear multiple keyword pattern matching algorithms, p. 26.
95/14	A. Ponse, C. Verhoef, S.F.M. Vlijmen (eds.)	De proceedings: ACP'95, p.
95/15	P. Niebert and W. Penczek	On the Connection of Partial Order Logics and Partial Order Reduction Methods, p. 12.
95/16	D. Dams, O. Grumberg, R. Gerth	Abstract Interpretation of Reactive Systems: Preservation of CTL*, p. 27.
95/17	S. Mauw and E.A. van der Meulen	Specification of tools for Message Sequence Charts, p. 36.
95/18	F. Kamareddine and T. Laan	A Reflection on Russell's Ramified Types and Kripke's Hierarchy of Truths, p. 14.
95/19	J.C.M. Baeten and J.A. Bergstra	Discrete Time Process Algebra with Abstraction, p. 15.
95/20	F. van Raamsdonk and P. Severi	On Normalisation, p. 33.
95/21	A. van Deursen	Axiomatizing Early and Late Input by Variable Elimination, p. 44.
95/22	B. Arnold, A. v. Deursen, M. Res	An Algebraic Specification of a Language for Describing Financial Products, p. 11.
95/23	W.M.P. van der Aalst	Petri net based scheduling, p. 20.
95/24	F.P.M. Dignum, W.P.M. Nuijten, L.M.A. Janssen	Solving a Time Tabling Problem by Constraint Satisfaction, p. 14.
95/25	L. Feijs	Synchronous Sequence Charts In Action, p. 36.
95/26	W.M.P. van der Aalst	A Class of Petri nets for modeling and analyzing business processes, p. 24.
95/27	P.D.V. van der Stok, J. van der Wal	Proceedings of the Real-Time Database Workshop, p. 106.
95/28	W. Fokkink, C. Verhoef	A Conservative Look at term Deduction Systems with Variable Binding, p. 29.
95/29	H. Jurjus	On Nesting of a Nonmonotonic Conditional, p. 14
95/30	J. Hidders, C. Hoskens, J. Paredaens	The Formal Model of a Pattern Browsing Technique, p.24.
95/31	P. Kelb, D. Dams and R. Gerth	Practical Symbolic Model Checking of the full μ-calculus using Compositional Abstractions, p. 17.
95/32	W.M.P. van der Aalst	Handboek simulatie, p. 51.
95/33	J. Engelfriet and JJ. Vereijken	Context-Free Graph Grammars and Concatenation of Graphs, p. 35.
95/34	J. Zwanenburg	Record concatenation with intersection types, p. 46.
95/35	T. Basten and M. Voorhoeve	An algebraic semantics for hierarchical P/T Nets, p. 32.
96/01	M. Voorhoeve and T. Basten	Process Algebra with Autonomous Actions, p. 12.
96/02	P. de Bra and A. Aerts	Multi-User Publishing in the Web: DreSS, A Document Repository Service Station, p. 12
96/03	W.M.P. van der Aalst	Parallel Computation of Reachable Dead States in a Free-choice Petri Net, p. 26.
96/04	S. Mauw	Example specifications in phi-SDL.
96/05	T. Basten and W.M.P. v.d. Aalst	A Process-Algebraic Approach to Life-Cycle Inheritance Inheritance = Encapsulation + Abstraction, p. 15.
96/06	W.M.P. van der Aalst and T. Basten	Life-Cycle Inheritance A Petri-Net-Based Approach, p. 18.
96/07	M. Voorhoeve	Structural Petri Net Equivalence, p. 16.
96/08	A.T.M. Aerts, P.M.E. De Bra, J.T. de Munk	OODB Support for WWW Applications: Disclosing the internal structure of Hyperdocuments, p. 14.
96/09	F. Dignum, H. Weigand, E. Verharen	A Formal Specification of Deadlines using Dynamic Deontic Logic, p. 18.
96/10	R. Bloo, H. Geuvers	Explicit Substitution: on the Edge of Strong Normalisation, p. 13.
96/11	T. Laan	AUTOMATH and Pure Type Systems, p. 30.
96/12	F. Kamareddine and T. Laan	A Correspondence between Nuprl and the Ramified Theory of Types, p. 12.
96/13	T. Borghuis	Priorean Tense Logics in Modal Pure Type Systems, p. 61