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

Citation for published version (APA):

Document status and date:
Published: 01/01/1996

Document Version:
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:
• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher’s website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:
www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:
openaccess@tue.nl
providing details and we will investigate your claim.

Download date: 02. Oct. 2020
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 environment. 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 transferred 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 ACP\textsuperscript{c} [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 psychiatrist) 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 \([\text{Response}] \rightarrow [\text{Request}]\) (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 \( \text{String} \rightarrow \text{String} \). In this case the main program looks as follows:

```haskell
main = interact f
f :: String \rightarrow String
```

When this is executed, the characters are read (for example from the keyboard) and then processed by \( f \). The function \( \text{interact} \) 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 \( \text{String} \rightarrow \text{String} \) 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:

```haskell
main = interact (unlines . g . lines)
g :: [String] \rightarrow [String]
```

Here we used the function \( \text{lines} \) from the standard prelude; it breaks a string into a sequence of strings by recognising the end of line characters. And \( \text{unlines} \) is its inverse. Note the \( \cdot \) 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 pausing 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, \( \tau \) for silent step and \( \parallel \) for parallel composition. The laws of ACP are always written as equations, such as the following laws, called 'basic process algebra'.

\[
x + y = y + x
\]
\[(x + y) + z = x + (y + z)\]
\[x + x = x\]
\[(x + y) \cdot z = x \cdot z + y \cdot z\]
\[(x \cdot y) \cdot z = x \cdot (y \cdot z)\]

To these one has to add additional laws describing ||, \(\tau\) 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:

- \(s(t)\) \((t \in \text{String})\)
- \(r(t)\) \((t \in \text{String})\)
- \(c(t)\) \((t \in \text{String})\)
- \(\tau\).

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

\[\gamma(s(t), r(t)) = c(t)\]
\[\gamma(r(t), s(t)) = c(t)\]
\[\gamma(a_1, a_2) = \text{undefined}\] (otherwise)

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?
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 $P[g]$. If we want to consider g as a mathematical function we denote it as $F[g]$. 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 \( \tau \). This tells
us that it satisfies the ACP equation.

\[
P[G] = \tau \cdot \sum_x \tau(x) \cdot s(updline(x)) \cdot P[G]
\]

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

\[
F[G](x : xs) = updline(x) : F[G](xs)
\]

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.

```haskell
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 (1 : ls) = (updline d 1) : (updlines (updDict d 1) ls)

 updDict :: Dict -> Line -> Dict
 updDict d 1 = (add (words 1) 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
```
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 $P[g] = U_{\text{iniDict}}$ where for all $d$ of type Dict we have that $U_d$ is given by:

$$U_d = \tau \cdot \sum_i r(i) \cdot s(I') \cdot U_{d'}$$

where $I' \equiv ((\text{unwords} \cdot \langle \text{map} \cdot \text{updword} \cdot d \rangle) \cdot \text{words}) \cdot I$ and $d' \equiv (\text{add} \cdot \text{words} \cdot 1) \cdot d$. This is consistent with the functional behaviour for which we check from the Gofer program text that $F[g] = u(\text{iniDict})$ where for all strings $I$ and $ls$,

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

### 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-pong-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 $\text{iniDict}$.

```haskell
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 = 1 : (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-
pong type, as shown below.

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

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 \( f_0, f_1, f_2 \)
etc. We devised a few patterns for which one needs a number of user-defined functions;
these have the following characterisation.

- \( f_0 \) – initialisation (cf. iniDict in Section 5),
- \( f_1 \) – reaction (cf. updline in Section 5),
- \( f_2 \) – reflection (how to update the internal state, cf. updDict in Section 5),
- \( f_3 \) – 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.

```haskell
-- G = skip . S(f5) . sum( x in Line,
--          x(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*x*r*        ",
        "1: (r*x)*inf - mem ",
        "2: (r*x)*        ",
        "3: (r[x])*inf    ",
        "4: (r*x)inf      ",
        "5: (r*x)*inf     ",
        "6: s(r*x)*       "]

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 psychiatrist. 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 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 

![Figure 3: Service network structure.](image-url)
PSF; the details are given in Section 9. At the same time these PSF texts are equational $ACP^2$ 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 user's 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.

```plaintext
-- 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 = ["$"]
```

12
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.  
2. p: I do not like my computer any more  
3. Do computers worry you?  
4. p: no, no  
5. p: no, no  
6. m: enclosed are ten dollars  
7. p: no, no  
8. Are you saying no just to be negative?

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 account 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, Boals
  functions
  tendollar : -> Dict
  variables
  c : -> Dict
  l : -> Line
  d : -> Word
  equations
  [01] f0 = cons(D,empty-dict)
  [02] f1(c,l) = 1 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,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
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({'l' ++ hi-i-m-eliza-please-tell-me-your-problem})`
6. `com. int-manager-to-router({'l' ++ 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 × int)`
10. `atom read-router1(do-computers-worry-you)`
11. `atom write-foo({'p' ++ no-no})`
12. `(1 × int)`
13. `atom write-foo({'p' ++ no-no})`
14. `(1 × int)`
15. `atom write-foo({'m' ++ enclosed-are-ten-dollars})`
16. `(1 × int)`
17. `atom write-foo({'p' ++ no-no})`
18. `(10 × 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 \( \mathcal{P}[g] \). It would be nice if we could extract the ACP equations for \( \mathcal{P}[g] \) 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 \( r \)). 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 \( i_1 \bar{t}_1 = t'_1 \ldots i_m \bar{t}_m = t'_m \) we define the set \( \text{funs} = \{ i_1, \ldots, i_m \} \).

For terms we define \( \text{vars}("string") = \emptyset \), \( \text{vars}([]) = \emptyset \), \( \text{vars}((t_1 : t_2)) = \text{vars}(t_1) \cup \text{vars}(t_2) \), \( \text{vars}(i) = \emptyset \) if \( i \in \text{funs} \), \( \{ i \} \), otherwise; \( \text{vars}((i t_1, \ldots, t_n)) = \text{vars}(t_1) \cup \ldots \cup \text{vars}(t_n) \). For patterns we define \( \text{vars}(i) = \emptyset \) and \( \text{vars}(i t_1, \ldots, t_n) = \text{vars}(t_1) \cup \ldots \cup \text{vars}(t_n) \).

For each rule \( i \bar{t} = t' \) we demand \( \text{vars}(t') \subseteq \text{vars}(\bar{t}) \) and \( i \in \text{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 \([\text{Line}] \to [\text{Line}]\). 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 \( \bot \) for a slightly different kind of analysis later). So we change the BNF rule for term as follows:

\[
\text{term} ::= @ \\
| "string" \\
| [] \\
| ( <term> : <term> ) \\
| <id> \\
| ( <id> <termlist> )
\]

We write \( \to \) for the functional reduction relation of Gofer (choosing an outermost redex from the reductions \( \beta_1, \pi_1, \pi_2 \))\(^1\). 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 occuring 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 \( \text{match}(c, p) = \text{true} \), \( \text{match}(c, p) = \text{false} \), or \( \text{match}(c, p) = \text{don't know} \) ("unknown"). In particular \( \text{match}(\emptyset, []) = \text{don't know} \) match(\emptyset, "string") = \text{don't know} and so is match(\emptyset, \((x : xs))\). But @ does match a variable. It is understood that the definition of match follows the structure of the 'c' recursively so that a \text{don't know} match on an internal subterm will lead to \text{don't know} at the top-level unless of course there is a \text{false}, which overrules the \text{don't know}. 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. \( \text{match}(c, p_i) = \text{true} \), and
2. for all \( j < i \) it holds that \( \text{match}(c, p_j) = \text{false} \).

The complication is demonstrated by the following program:

\[
g \[\text{"foo"} \] = \[\text{"goodbye"}\] \\
g (x : xs) = \[\text{"bye"}\]
\]

\(^1\)We define \( \pi_1 \) as the rule head \((x : xs) = x \) and \( \pi_2 \) 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 $\mathcal{P}[g]$ 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

$$(T, [], \odot, [], :=, \psi)$$

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

1. $[] \not\in \text{dom}(\psi)$,
2. if $\psi(x) = x'$ then $\psi(x : xs) = (x' : xs)$ $(x, x' \in T)$,
3. if $\odot \not\in c$ and $c \not\in []$ and $c \not\in (x : xs)$ then $c \in \text{dom}(\psi)$.

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 $\psi$ derived from the rules $\{p_i | i = 1, 2, \ldots \}$ of the program where $c \in \text{dom}(\psi)$ iff $\exists$ match($c, p_i$) $= \text{true}$ and $\forall j < i$, match($c, p_j$) $= \text{false}$. 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 $x \not\in \text{dom}(\psi)$. So $\rightarrow$ is a functional reduction relation, i.e. a TRS together with a reduction strategy. We write $c \not\rightarrow$ to mean $\exists x', x \rightarrow x'$ 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 $\rightarrow$ (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 $\rightarrow$ or the termination relation $\downarrow$.

$$
\begin{align*}
\frac{c \equiv []}{c \downarrow} & \quad \frac{\neg(c \equiv (x : xs) \land x \not\rightarrow)}{c \rightarrow c'}
\end{align*}
$$
The action relation $\rightsquigarrow$ 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):

\[
\begin{align*}
\begin{array}{llll}
  & c \equiv (x : xs) & c \equiv (x : xs) & c \neq [] \\
\odot \notin x & \odot \in x & c \neq (y : ys) & c \neq \tau \\
\vdash s(x) & \vdash r(x) & c \vdash \begin{array}{c}
  \circ \leftarrow z \\
  \vdash \begin{array}{c}
  c \circ := x : \circ \\
  \end{array}
\end{array} & c \vdash \begin{array}{c}
  c \circ := x : \circ \\
  \end{array}
\end{array}
\end{align*}
\]

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:

\[
c_0 = (g \odot)
\]

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

\[
\mathcal{P}[c] =
\begin{align*}
& [c \equiv []] \rightarrow c \\
& + [\neg (c \equiv (x : xs) \land x \vdash) \text{ and } c \vdash c'] \rightarrow \tau \cdot \mathcal{P}[c'] \\
& + [c \equiv (x : xs) \text{ and } \odot \notin x \text{ and } x \vdash] \rightarrow s(x) \cdot \mathcal{P}[xs] \\
& + [c \equiv (x : xs) \text{ and } \odot \in x \text{ and } x \vdash] \rightarrow \sum_z r(z) \cdot \mathcal{P}[c[\odot := z : \circ]] \\
& + [c \neq [] \text{ and } c \neq (y : ys) \text{ and } x \vdash] \rightarrow \sum_z r(x) \cdot \mathcal{P}[c[\odot := x : \circ]]
\end{align*}
\]

We used the notation $[\text{condition}] \rightarrow$ 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 $\vdash_g$ and $\rightsquigarrow_g$. 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 $R \subseteq C_g \times C_g$ is a rooted branching bisimulation if it satisfies:

1. $c_0 Rc_0$
2. if $c \vdash c'$ and $cRd$, then either
   \begin{enumerate}
   \item $a = \tau$ and $c' Rd$, or
   \item $\exists d_1, d' \cdot d \xrightarrow{\tau} d_1 \xrightarrow{a} d' \land cRd_1 \land c' Rd'$
   \end{enumerate}
3. if $c \vdash cRd$ then $\exists d' \cdot d \xrightarrow{\tau} d' \land d' \vdash cRd'$
4. similarly when the roles of $c$ and $d$ are interchanged.
We define $\leftrightarrow_{rb}$ as the unique maximal rooted branching bisimulation relation.

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

$$
\begin{align*}
\text{if } c = \epsilon & \text{ then } c \equiv \epsilon \\
\text{if } c = \tau \cdot c' & \text{ then } c \equiv \tau \\
\text{if } c = x : xs & \text{ then } c \equiv (x : xs) \\
\text{if } c = s(x) \cdot xs & \text{ then } c \equiv s(x) \cdot xs
\end{align*}
$$

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 $\tau$, 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 $\tau$ (of course there are certain contexts in which Koomen's fair abstraction rule can be applied and then an infinite sequence of $\tau$ steps turns into $\delta$ and then disappears).

**Theorem (Soundness).** Consider an abstract functional program $(T, [], ;, \otimes, :=, \psi)$ for $g$, then

$$
\text{if } \text{ACP}_\tau + \text{EQ}(g) \vdash c = d \text{ then } c \leftrightarrow_{rb} d
$$

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

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

Checking the other axiom schemas can be done along the same lines. We only add one remark for the last axiom. In general, from $c \xrightarrow{r(x)} c[\cdot] : = x : \cdot$ we can only deduce that $c = r(x) \cdot c[\cdot] : = x : \cdot + ...$ (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 ACP 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,

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

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

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

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

The $\tau$ steps are removable because of the usual $\tau$ law of ACP 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 transparency' 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 $\mathcal{F}[g]$. We must define the meaning function $\mathcal{F}[\cdot]$. 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 $V \times \cdots (V \times V) \cdots$ with $V^n$. And $V^* = \bigcup_{n \in \mathbb{N}} V^n$. So in this section we write String* instead of [String]. Finally $V^\infty$ denotes the function space $\mathbb{N} \to V$. We order the set String $\cup \{\bot\}$ by $\sqsubseteq$ by postulating that for $x, y \in \text{String} \cup \{\bot\}$: $x \sqsubseteq y :\Leftrightarrow (x = \bot) \lor (x = y)$. We write $\text{STR}$(String) for the set:

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

An element of $\text{STR}$(String) is called a stream. Most often we just write STR for $\text{STR}$(String). We order the set STR by $\sqsubseteq$ by postulating that for $s, s' \in \text{STR}$ and thus $s_0, s'_0, s_1, s'_1, \ldots \in \text{String} \cup \{\bot\}$ (here $s_k$ denotes the $k$-th element of $s$):

$s \sqsubseteq s' :\Leftrightarrow (\exists n \in \mathbb{N} \cdot [s_0..s_{n-1}] = [s'_0..s'_{n-1}] \land s_n = \bot) \lor (s = s')$

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

In [4] it is shown that $(\text{STR}, \sqsubseteq)$ 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. $\bigcup D$ 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: $s = \bigcup \{t \subseteq s \mid t \text{ finite}\}$,
- the set of finite elements (the $s$ such that for all directed $D$ we have $s \sqsubseteq \bigcup D \Rightarrow \exists t \in D \cdot s \sqsubseteq t$) is countable.

The least stream is $[\bot]$. 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 $(x, y) \sqsubseteq (x', y') :\Leftrightarrow (x \sqsubseteq x') \land (y \sqsubseteq y')$.

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

- $x : [s_0, s_1, \ldots, s_{n-1}] = [x, s_0, s_1, \ldots, s_{n-1}]$ (as before),
- $\bot : [s_0, s_1, \ldots, s_{n-1}] = [\bot],$
- $x : [s_0, s_1, \ldots, s_{n-1}, \bot] = [x, s_0, s_1, \ldots, s_{n-1}, \bot],$
- $\bot : [s_0, s_1, \ldots, s_{n-1}, \bot] = [\bot].$
Note that `';` is monotonic in both arguments, that is, for \( x, y \in (\text{String} \cup \bot) \), \( s, s' \in \text{STR} \), \( x \sqsubseteq y \Rightarrow (x : s) \sqsubseteq (y : s) \) 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 \( \text{String} \times \text{String} \rightarrow \text{Bool} \) to a function `==` of type \( (\text{String} \cup \bot) \times (\text{String} \cup \bot) \rightarrow (\text{Bool} \cup \bot) \). Let it be understood that \( x, y \neq \bot \).

- \((x == y) = \text{True} \) if \( x = y \)
- \((x == y) = \text{False} \) if \( x \neq y \)
- \((\bot == x) = (x == \bot) = \bot\).

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

We make head and tail total by putting head([]) = head([\bot]) = \bot and tail([]) = tail([\bot]) = [\bot]. 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 \( (\text{Bool} \cup \bot) \times (\text{String} \cup \bot) \times (\text{String} \cup \bot) \rightarrow (\text{String} \cup \bot) \) and of type \( (\text{Bool} \cup \bot) \times \text{STR} \times \text{STR} \rightarrow \text{STR} \). We define that:

- \((\text{if True then } a \text{ else } b) = a\),
- \((\text{if False then } a \text{ else } b) = b\),
- \((\text{if } \bot \text{ then } a \text{ else } b) = \bot\),

\( ([\bot], \text{respectively})\).

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 3rd argument.

If we have a CPO \((S, \sqsubseteq)\) then we can order the function space \( S \rightarrow S \) by letting \( f_1 \sqsubseteq f_2 \iff \forall x \in S : f_1(x) \sqsubseteq f_2(x) \). A function \( f \) is continuous if for all directed \( D \subseteq S \) we have \( f(\bigcup D) = \bigcup f(D) \). 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 \( \text{Fix}(f) = \bigcup_n f^n(\bot) \) where \( \bot \) is the least element of the CPO.

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

- (i) `';
- (ii) head, tail,
- (iii) `==`,
- (iv) if-then-else,
- (v) \( \lambda \)-abstraction,
- (vi) 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 $\bigcup D = z = \bigcup \{ x \mid x \in D \}$ for directed $D$. Note that $|D| \leq 2$. If $z = \bot$ we check $\bot = \bigcup \{ \bot \}$. Otherwise let $z \neq \bot$. Two cases arise: either (a) $\bigcup D = \bot$ so $D = \{ \bot \}$ whence $(\bigcup D = z) = \bot = \bigcup \{ \bot \} = \bigcup \{ \bot = z \}$ (end of a), or (b) $\bigcup D \neq \bot$ so for some $x \neq \bot$ we find $D = \{ z, \ldots \}$. Now if $z = z$ then $(\bigcup D = z) = \text{True} = \bigcup \{ \text{True}, \ldots \} = \bigcup \{ z = z, \ldots \}$. And if $z \neq z$ then $(\bigcup D = z) = \text{False} = \bigcup \{ \text{False}, \ldots \} = \bigcup \{ z = z, \ldots \}$.

(iii) Analogously.


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

$$g \bar{x} (p \bar{x}) = \text{rhs}_1 \quad \text{becomes} \quad g = \lambda \bar{x}. \text{if} \ (p \bar{x})\ \text{then} \ \text{rhs}_1 \ \text{else} \ \text{rhs}_2$$

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

$$g [\ ] = \text{rhs}_1 \quad \text{becomes} \quad g = \lambda x. \text{if} \ x = [\ ] \ \text{then} \ \text{rhs}_1 \ \text{else if head}(z) = \text{head}(z) \ \text{then} \ \text{rhs}_2[x := \text{head}(z), z := \text{tail}(z)] \ \text{else} \ [\ ]$$

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

$$\text{trans} : F[g] \to P[g].$$

**Counter example.** Define two programs $g_1$ and $g_2$ as follows:

$$g_1 [\ ] = g_1 [\ ] \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \ quad
Then for $i \in \{1, 2\}$ we have $\forall \vec{x} \in \text{STR} \cdot \mathcal{F}[g_i] = [\bot]$. But $g_1$ performs an infinite rewriting without consuming input (but the first line), whereas $g_2$ consumes all input (still producing nothing). So $P[g_1] = \sum_\tau \tau(x) \cdot r(x)$ whereas $P[g_2] = \sum_\tau r(x) \cdot G_2$. 

The syntactic forms of the definitions of $g_1$ and $g_2$ 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 transparency is not applicable. Not all is lost however; there is a positive result too.

**Proposition.** For all $g$, let $\text{SOS}(g) = (c_0, 1, \ldots, 1)$. Then for all traces $\vec{f} \in \{s(x), r(x), \tau\}^*$, defining $\mapsto$ in the obvious way:

$$\text{if } \exists c \cdot c_0 \mapsto c \text{ then } (\mathcal{F}[g](\vec{f}), ++[\downarrow])|_{\text{non-}\downarrow} = \vec{f}_s$$

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

**Proof.** We show the essential details for an example. Consider the program $g$ and the dialogue of Section 4. We write ‘$1$’ for ‘let us explain the compiler’, ‘$L$’ for ‘LET US EXPLAIN THE COMPILER’, and so on. We shall consider the trace $\vec{f} = [r('l'), \tau, s('L'), r('i'), \tau, \tau, s('I')]$ for which $\vec{f}_s = ['l', 'i']$ and $\vec{f}_s = ['L', 'I']$.

First we analyse the operational behaviour, for which we rewrite according to the program’s TRS and apply the SOS rules. $(g \odot) \xrightarrow{r('1')} (g (('1': \odot)) \xrightarrow{} (\text{updline '1'}) : (g \odot)) \xrightarrow{} ('L' : (g \odot)) \xrightarrow{s('L')} (g \odot) \xrightarrow{r('i')} \tau : s('i') \xrightarrow{\tau} s('i') (g \odot)$. This shows that the premiss of our proposition holds, that is, $\exists c : c_0 \mapsto c$.

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

$$g = \lambda z. \text{if } z = \epsilon \text{ then } \epsilon \text{ else if } \text{head}(z) = \epsilon \text{ then } (\text{updline head}(z)) : (g \text{ tail}(z)) \text{ else } [\downarrow]$$

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

$$g ('l', 'i', \downarrow) = \text{if } z = \epsilon \text{ then } \epsilon \text{ else if } 'l' = 'L' \text{ then } 'L': g('i', \downarrow) \text{ else } [\downarrow]$$

$$= ['L': (g ['i', \downarrow]) \text{ else } 'L': (g ['i', \downarrow])]$$

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

$$\ldots = ['L': (g ['i', \downarrow]) \text{ else } 'L': (g ['i', \downarrow])]$$

$$= ['L', 'I', \downarrow]$$

25
From the claim, $(F[g][\ell,+\perp[\perp]])_{\text{non-}\ell} = g([1',1',\perp])_{\text{non-}\ell} = [1',1',\perp]_{\text{non-}\ell} = [1',1'] = [r(1'),\tau,\tau,s(1'),\tau,\tau,s(1')]_{s} = \ell_{s}$, 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.

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 $(\text{map (reverse . reverse)} . \text{lines})$. 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


In Computing Science Reports

In this series appeared:

93/01  R. van Geldrop
93/02  T. Verhoeff
93/03  T. Verhoeff
93/04  E.H.L. Aarts  J.H.M. Kusst  P.J. Zwietenig
93/05  J.C.M. Baeten  C. Verhoef
93/06  J.P. Velkkamp
93/07  P.D. Moerland
93/08  J. Verhoosel
93/09  K.M. van Hee
93/10  K.M. van Hee
93/11  K.M. van Hee
93/12  K.M. van Hee
93/13  K.M. van Hee
93/14  J.C.M. Baeten  J.A. Bergstra
93/15  J.C.M. Baeten  J.A. Bergstra  R.N. Bol
93/16  H. Schepers  J. Hoorman
93/17  D. Alstein  P. van der Stok
93/18  C. Verhoeff
93/19  G.J. Houben
93/20  F.S. de Boer
93/21  M. Codish  D. Dams  G. Fles  M. Bruynooghe
93/22  E. Poll
93/23  E. de Kogel
93/24  E. Poll  and Paula Severi
93/25  H. Schepers and R. Gerth
93/26  W.M.P. van der Aalst
93/27  T. KIoks and D. Kratsch
93/28  F. Kamarecddine and  R. Nederpel
93/29  R. Post and P. De Bra
93/30  J. Deogun  T. KIoks  D. Kratsch  H. Müller

Department of Mathematics and Computing Science
Eindhoven University of Technology

Deriving the Aho-Corasick algorithms: a case study into the synergy of programming methods, p. 36.
A continuous version of the Prisoner's Dilemma, p. 17
Quick sort for linked lists, p. 8.
Deterministic and randomized local search, p. 78.
A congruence theorem for structured operational semantics with predicates, p. 18.
On the unavoidability of metastable behaviour, p. 229
Exercises in Multiprogramming, p. 97
A Formal Deterministic Scheduling Model for Hard Real-Time Executions in DEDOS, p. 32.
Systems Engineering: a Formal Approach Part II: Frameworks, p. 44.
A Trace-Based Compositional Proof Theory for Fault Tolerant Distributed Systems, p. 27
Hard Real-Time Reliable Multicast in the DEDOS system, p. 19.
A congruence theorem for structured operational semantics with predicates and negative premises, p. 22.
The Design of an Online Help Facility for ExSpect, p.21.
Freeness Analysis for Logic Programs - And Correctness, p. 24
A Typechecker for Bijective Pure Type Systems, p. 28.
Relational Algebra and Equational Proofs, p. 23.
Pure Type Systems with Definitions, p. 38.
Multi-dimensional Petri nets, p. 25.
Finding all minimal separators of a graph, p. 11.
A Semantics for a fine λ-calculus with de Bruijn indices, p. 49.
GOLD, a Graph Oriented Language for Databases, p. 42.
On Vertex Ranking for Permutation and Other Graphs, p. 11.
Derivation of delay insensitive and speed independent CMOS circuits, using directed commands and production rule sets, p. 40.


ILIAS, a sequential language for parallel matrix computations, p. 20.

Real Time Process Algebra with Infinitesimals, p. 39.

Abstract Reduction and Topology, p. 28.

Noa Interleaving Process Algebra, p. 17.

Design and Analysis of Dynamic Leader Election Protocols in Broadcast Networks, p. 73.

A general conservative extension theorem in process algebra, p. 17.

Job Shop Scheduling by Constraint Satisfaction, p. 22.


Temporal operators viewed as predicate transformers, p. 11.

Automatic Verification of Regular Protocols in P/T Nets, p. 23.

A taxonomy of finite automata construction algorithms, p. 87.

A taxonomy of finite automata minimization algorithms, p. 23.

A precise clock synchronization protocol, p. 16.


Verifying Sequentially Consistent Memory using Interface Refinement, p. 20.

The object-oriented paradigm, p. 28.

Canonical typing and ω-conversion, p. 51.


Graph Isomorphism Models for Non Interleaving Process Algebra, p. 18.


Time and the Order of Abstract Events in Distributed Computations, p. 29.


A Hierarchical Diagrammatic Representation of Class Structure, p. 22.

Process Algebra with Partial Choice, p. 16.
94/10  T. verhoeff  The testing Paradigm Applied to Network Structure. p. 31.
94/13  R. Seljæs  A New Method for Integrity Constraint checking in Deductive Databases. p. 34.
94/14  W. Peremans  Ups and Downs of Type Theory, p. 9.
94/16  R.C. Buckhouse  H. Doornbos  Mathematical Induction Made Calculational, p. 36.
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 λ→, p. 22.
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 ∀CTL*, 3CTL* and CTL*, p. 28.
94/25  T. Kloks  K_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.
94/29  J. Hooman  Correctness of Real Time Systems by Construction, p. 22.
94/33  T. Laan  A formalization of the Ramified Type Theory, p.40.
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 II-conversion in the Barendregt Cube, p. 19.
94/38  A. Bijlsma  C.S. Scholten  Point-free substitution, p. 10.
On the equivalence covering number of splitgraphs, p. 4.

Distributed Consensus and Hard Real-Time Systems, p. 34.

Computing a perfect edge without vertex elimination ordering of a chordal bipartite graph, p. 6.

Concatenation of Graphs, p. 7.

Category Theory as Coherently Constructive Lattice Theory: An Illustration, p. 35.

Verifying Sequentially Consistent Memory, p. 160

The \( \lambda \)-cube with classes of terms modulo conversion, p. 16.

On \( \Pi \)-conversion in Type Theory, p. 12.

Fixed-Point Calculus, p. 11.

Process Algebra with Propositional Signals, p. 25.

A short and flexible proof of Strong Normalization for the Calculus of Constructions, p. 27.

Listing simplicial vertices and recognizing diamond-free graphs, p. 4.

Traces and Logic, p. 81

A Partial Order Approach to Branching Time Logic Model Checking, p. 20.

The Construction of a small Communication Library, p. 16.

Formalizing Process Algebraic Verifications in the Calculus of Constructions, p. 49.

Concrete process algebra, p. 134.

An Isotopic Invariant for Planar Drawings of Connected Planar Graphs, p. 9.

A Type Inference Algorithm for Pure Type Systems, p. 20.

A Quantitative Analysis of Iterated Local Search, p. 23.

Drawing Execution Graphs by Parsing, p. 10.

Preservation of Strong Normalisation for Explicit Substitution, p. 12.

Discrete Time Process Algebra, p. 20

MathPad: A System for On-Line Preparation of Mathematical Documents, p. 15
<table>
<thead>
<tr>
<th>Page</th>
<th>Title</th>
<th>Authors</th>
</tr>
</thead>
<tbody>
<tr>
<td>95/11</td>
<td>Deductive Database Systems and Integrity Constraint Checking</td>
<td>R. Seljee</td>
</tr>
<tr>
<td>95/12</td>
<td>Empty Interworkings and Refinement Semantics of Interworkings Revised</td>
<td>S. Mauw and M. Reniers</td>
</tr>
<tr>
<td>95/13</td>
<td>A Taxonomy of Sublinear Multiple Keyword Pattern Matching Algorithms</td>
<td>B.W. Watson and G. Zwaan</td>
</tr>
<tr>
<td>95/14</td>
<td>De Proceedings: ACP'95</td>
<td>A. Peute, C. Verheof, S.F.M. Vlijmen</td>
</tr>
<tr>
<td>95/15</td>
<td>On the Connection of Partial Order Logics and Partial Order Reduction Methods</td>
<td>P. Niebert and W. Penczek</td>
</tr>
<tr>
<td>95/16</td>
<td>Abstract Interpretation of Reactive Systems: Preservation of CTL*</td>
<td>D. Dams, O. Grumberg, R. Gerth</td>
</tr>
<tr>
<td>95/17</td>
<td>Specification of Tools for Message Sequence Charts</td>
<td>S. Mauw and E.A. van der Meulen</td>
</tr>
<tr>
<td>95/18</td>
<td>A Reflection on Russell’s Ramified Types and Kripke’s Hierarchy of Truths</td>
<td>F. Kamareddine and T. Laan</td>
</tr>
<tr>
<td>95/19</td>
<td>Discrete Time Process Algebra with Abstraction</td>
<td>J.C.M. Baeten and J.A. Bergstra</td>
</tr>
<tr>
<td>95/20</td>
<td>On Normalisation</td>
<td>F. van Raamsdonk and P. Severi</td>
</tr>
<tr>
<td>95/21</td>
<td>Axiomatizing Early and Late Input by Variable Elimination</td>
<td>A. van Deursen</td>
</tr>
<tr>
<td>95/23</td>
<td>Petri Net Based Scheduling</td>
<td>W.M.P. van der Aalst</td>
</tr>
<tr>
<td>95/24</td>
<td>Solving a Time Tabling Problem by Constraint Satisfaction</td>
<td>F.P.M. Dignum, W.P.M. Nijten, L.M.A. Janssen</td>
</tr>
<tr>
<td>95/25</td>
<td>Synchronous Sequence Charts in Action</td>
<td>L. Feijs</td>
</tr>
<tr>
<td>95/26</td>
<td>A Class of Petri Nets for Modeling and Analyzing Business Processes</td>
<td>W.M.P. van der Aalst</td>
</tr>
<tr>
<td>95/27</td>
<td>Proceedings of the Real-Time Database Workshop</td>
<td>P.D.V. van der Stok, J. van der Wal</td>
</tr>
<tr>
<td>95/28</td>
<td>A Conservative Look at Term Deduction Systems with Variable Binding</td>
<td>W. Fokkink, C. Verheof</td>
</tr>
<tr>
<td>95/29</td>
<td>On Nesting of a Nonmonotonic Conditional</td>
<td>H. Jurjus</td>
</tr>
<tr>
<td>95/30</td>
<td>The Formal Model of a Pattern Browsing Technique</td>
<td>J. Hidders, C. Hoskens, J. Paredaens</td>
</tr>
<tr>
<td>95/31</td>
<td>Practical Symbolic Model Checking of the Full μ-Calculus Using Compositional Abstractions</td>
<td>P. Kelb, D. Dams and R. Gerth</td>
</tr>
<tr>
<td>95/32</td>
<td>Handbook Simulation</td>
<td>W.M.P. van der Aalst</td>
</tr>
<tr>
<td>95/33</td>
<td>Context-Free Graph Grammars and Concatenation of Graphs</td>
<td>J. Engelfriet and J.J. Vereijken</td>
</tr>
<tr>
<td>95/34</td>
<td>Record Concatenation with Intersection Types</td>
<td>J. Zwienen</td>
</tr>
<tr>
<td>95/35</td>
<td>An Algebraic Semantics for Hierarchical P/T Nets</td>
<td>T. Basten and M. Voorhoeve</td>
</tr>
<tr>
<td>96/01</td>
<td>Process Algebra with Autonomous Actions</td>
<td>M. Voorhoeve and T. Basten</td>
</tr>
<tr>
<td>96/02</td>
<td>Multi-User Publishing in the Web: DreSS, A Document Repository Service Station</td>
<td>P. de Bra and A. Aerts</td>
</tr>
<tr>
<td>96/03</td>
<td>Parallel Computation of reachable Dead States in a Free-choice Petri Net</td>
<td>W.M.P. van der Aalst</td>
</tr>
<tr>
<td>96/04</td>
<td>Example Specifications of phi-SDL</td>
<td>S. Mauw</td>
</tr>
<tr>
<td>96/05</td>
<td>A Process-Algebraic Approach to Life-Cycle Inheritance</td>
<td>T. Basten and W.M.P. v.d. Aalst</td>
</tr>
<tr>
<td>96/06</td>
<td>Life-Cycle Inheritance</td>
<td>W.M.P. van der Aalst and T. Basten</td>
</tr>
<tr>
<td>96/07</td>
<td>Structural Petri Net Equivalence</td>
<td>M. Voorhoeve</td>
</tr>
<tr>
<td>96/08</td>
<td>OODB Support for WWW Applications: Disclosing the Internal Structure of Hyperdocuments</td>
<td>A.T.M. Aerts, P.M.E. De Bra, J.T. de Munk</td>
</tr>
<tr>
<td>96/10</td>
<td>Explicit Substitution: on the Edge of Strong Normalisation</td>
<td>R. Bloo, H. Geuvers</td>
</tr>
<tr>
<td>96/11</td>
<td>AUTOMATH and Pure Type Systems</td>
<td>T. Laan</td>
</tr>
<tr>
<td>96/12</td>
<td>A Correspondence between Nuprl and the Ramified Theory of Types</td>
<td>F. Kamareddine and T. Laan</td>
</tr>
<tr>
<td>96/13</td>
<td>Priorean Tense Logics in Modal Pure Type Systems</td>
<td>T. Borghuis</td>
</tr>
<tr>
<td>Year</td>
<td>Authors</td>
<td>Title</td>
</tr>
<tr>
<td>------</td>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>96/14</td>
<td>S.H.J. Bos and M.A. Reniers</td>
<td>The $I^2$ C-bus in Discrete-Time Process Algebra</td>
</tr>
<tr>
<td>96/15</td>
<td>M.A. Reniers and J.J. Vereijken</td>
<td>Completeness in Discrete-Time Process Algebra</td>
</tr>
<tr>
<td>96/16</td>
<td>P. Hoogendijk and O. de Moor</td>
<td>What is a data type?</td>
</tr>
<tr>
<td>96/17</td>
<td>E. Boiten and P. Hoogendijk</td>
<td>Nested collections and polytypism</td>
</tr>
<tr>
<td>96/18</td>
<td>P.D.V. van der Stok</td>
<td>Real-Time Distributed Concurrency Control Algorithms with mixed time constraints</td>
</tr>
<tr>
<td>96/19</td>
<td>M.A. Reniers</td>
<td>Static Semantics of Message Sequence Charts</td>
</tr>
</tbody>
</table>