
Algebraic Specification of Lazy Functional Programs in a Concurrent Environment

L.M.G. FEIJS^{1*} AND M.A. RENIERS²

¹*Information & Software Technology, Philips Research Laboratories Eindhoven, Prof.
Holstlaan 4, P.O. Box NL-5656 AA Eindhoven, The Netherlands.*

²*Department of Mathematics and Computing Science, Eindhoven University of Technology,
P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands.*

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 analyze the communication behaviour of the program and its environment. We propose to use the Algebra of Communicating Processes (ACP_{ϵ}^{τ}) for that purpose. The present paper attempts to bridge the gap between the functional and the process-oriented worlds. The term rewriting system of the functional language, the operational semantics of the I/O mechanism and the process equations of a program are described and their relationships are analyzed. We abstract from the details of the particular programming language by using an intermediate concept of ‘abstract functional program’.

Keywords: ACP_{ϵ}^{τ} , Communication protocols, Formal reasoning, Functional programming, Lazy evaluation, Operational semantics, Patterns, Rooted branching bisimulation, Simulation, Term deduction systems.

1. INTRODUCTION AND MOTIVATION

It is important to have precise specifications of reactive computer programs in order to analyze their behaviour in the context of a large system. In particular, this applies to programs which are meant for execution in a complex communication network. In this paper we establish a rigorous relationship between the reactive I/O behaviour of lazy functional programs and the process theoretic equations that describe this behaviour. Therefore the term rewriting system of a lazy functional language, the operational semantics of the I/O mechanism, and the process equations are described and analyzed. For the lazy functional programs, details are taken from Gofer [1]. The process equations are given in ACP_{ϵ}^{τ} [2], which enables us to simulate them in PSF (Process Specification Formalism) [3].

Although it is possible to let a functional program perform all kinds of I/O actions, such as reading and writing files, we have adopted a restriction to so-called Landin-style stream I/O [4]; 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 most often

these are realized by other means already, and their process behaviour can be specified using process-theoretic means too. In this way the data processing aspects are separated from the communication aspects.

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_{ϵ}^{τ} are introduced. In Section 4 we give an introductory example. In Section 5 we study the process semantics. In Section 6 we give two examples of programs and their process semantics. In Section 7 we discuss options for exploiting the results. In Section 8 we discuss related work. In Section 9 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 representa-

tion and manipulation of data structures. With respect to the reduction strategy, *lazy evaluation* means that:

- an argument to a function is not evaluated before 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 head of the list 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 `String -> String`. In this case the main program has the following form:

```
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 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 recognizing the end of line characters. The function `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 synchronization is relevant. As explained above, the synchronization between the responses and the requests is regulated by the lazy evaluation 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.

The interaction of Gofer programs with a user via a teletype IO mechanism may seem not a challenging subject, the point is that functional programs can participate as components in networked or distributed applications. Then it can be important to have a rigorous description of the functional program viewed as a process. Although we restrict ourselves to Landin-style stream I/O, we expect that some of the techniques we studied can be extended to more complicated I/O as well.

In our present study we propose a language fragment. We only give the BNF rules but we assume type-correctness as usual. Function names are productions from `<id-f>` and variable names from `<id-v>`.

```
<program> ::= <rule>+
<rule> ::= <pattern> = <term>
<pattern> ::= <id-f>
| <id-f> <term>+
<term> ::= "string"
| []
| ( <term> : <term> )
| <id-f>
| <id-v>
| ( <id-f> <term>+ )
```

A functional program consists of a number of rules each of which is of the form `pattern equals term`. The constant `[]` represents the empty list, the operation `:` represents the 'cons' operator which can be used for adding a single element in front of a list.

The following example program accepts strings as inputs and generates two copies of each input string as output.

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

For a program given as

$$p \equiv \begin{cases} f_1 \vec{t}_1 & = t'_1 \\ f_2 \vec{t}_2 & = t'_2 \\ \vdots & \vdots \\ f_m \vec{t}_m & = t'_m \end{cases}$$

with, for $m \geq 1$, f_1, \dots, f_m function names, $\vec{t}_1, \dots, \vec{t}_m$ lists of terms (which can be empty), and t'_1, \dots, t'_m terms, we define the mapping *funcs* which associates to such a program p a set of *function names* as follows:

$$\text{funcs}(p) = \{f_1, \dots, f_m\}.$$

For a given program p , the mapping vars_p associates to a term t the variables occurring in t in the context of program p ². Note that the identifier preceding a list of terms can not be a variable, it has to be a function. For each rule $i\vec{t} = t'$ of a program p we demand $\text{vars}_p(t') \subseteq \text{vars}_p(\vec{t})$.

3. ASPECTS OF ACP_ϵ^τ

The Algebra of Communicating Processes ACP_ϵ^τ , proposed by Baeten and Weijland [2], is a theory about processes and their communication behaviour in the tradition of CCS [5]. For an introduction to ACP_ϵ^τ , see [2]. We mention some of the most important operators: constants from a set A , $+$ for alternative composition, \cdot for sequential composition, τ for silent step and \parallel for parallel composition. Expressions that can be built using these constants and operators are called terms or processes and these are denoted by \mathcal{P} . The laws of ACP_ϵ^τ are always written as equations, such as the laws given in Table 1, called Basic Process Algebra ($\text{BPA}_{\delta\epsilon}^\tau$). To

TABLE 1. Axioms of $\text{BPA}_{\delta\epsilon}^\tau$

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

²Formally, for a function name f , a variable name v , and terms t_1, \dots, t_n ($n \geq 1$), this mapping is defined inductively as follows: $\text{vars}_p(\text{"string"}) = \emptyset$, $\text{vars}_p(\square) = \emptyset$, $\text{vars}_p((t_1:t_2)) = \text{vars}_p(t_1) \cup \text{vars}_p(t_2)$, $\text{vars}_p(f) = \emptyset$, $\text{vars}_p(v) = \{v\}$, $\text{vars}_p((ft_1 \dots t_n)) = \bigcup_{1 \leq i \leq n} \text{vars}_p(t_i)$ and $\text{vars}_p(t_1 \dots t_n) = \bigcup_{1 \leq i \leq n} \text{vars}_p(t_i)$.

these one has to add additional laws describing additional operators such as \parallel , ∂_H , τ_I , etc. For these axioms we refer to Table 52 of [2].

We also allow for the use of recursion for specifying processes. The notation $\langle X \mid E \rangle$ describes the solution(s) for recursion variable X in the recursive specification E . Also, for an arbitrary term $s_X(V)$, $\langle s_X(V) \mid E \rangle$ denotes the term(s) that are obtained by replacing the recursion variables by their solution(s).

The standard semantics for process algebras in the tradition of ACP_ϵ^τ are the term deduction systems, also called action relations or structured operational semantics. For these we introduce a ternary action relation $\text{action} \subseteq \mathcal{P} \times A_\tau \times \mathcal{P}$ and a termination predicate $\text{terminates} \subseteq \mathcal{P}$. Usually a triple $(p, a, q) \in \text{action}$ is denoted by $p \xrightarrow{a} q$ and the fact that $p \in \text{terminates}$ by $p \downarrow$. The action relation and termination predicate are defined by so-called deduction rules. These are of the form $\frac{H}{C}$ where H is a set of hypotheses and C is the conclusion. If all of the hypotheses hold, then we can conclude that the conclusion holds as well. For $\text{BPA}_{\delta\epsilon}^\tau$, these are the smallest relations satisfying the deduction rules given in Table 2 (see [2], Table 11 and Table 12). The term deduction system for $\text{BPA}_{\delta\epsilon}^\tau$ is denoted by $T(\text{BPA}_{\delta\epsilon}^\tau)$ and for ACP_ϵ^τ by $T(\text{ACP}_\epsilon^\tau)$.

TABLE 2. Deduction rules: for $a \in A$ and $X = t_X \in E$

$$\begin{array}{c} \frac{}{\epsilon \downarrow} \quad \frac{}{a \xrightarrow{a} \epsilon} \\ \frac{x \xrightarrow{a} x'}{x + y \xrightarrow{a} x'} \quad \frac{y \xrightarrow{a} y'}{x + y \xrightarrow{a} y'} \\ \frac{x \downarrow}{x + y \downarrow} \quad \frac{y \downarrow}{x + y \downarrow} \\ \frac{x \xrightarrow{a} x'}{x \cdot y \xrightarrow{a} x' \cdot y} \quad \frac{x \downarrow, y \xrightarrow{a} y'}{x \cdot y \xrightarrow{a} y'} \quad \frac{x \downarrow, y \downarrow}{x \cdot y \downarrow} \\ \frac{\langle t_X \mid E \rangle \downarrow}{\langle X \mid E \rangle \downarrow} \quad \frac{\langle t_X \mid E \rangle \xrightarrow{a} y}{\langle X \mid E \rangle \xrightarrow{a} y} \end{array}$$

As a notion of equivalence on closed terms we use rooted branching bisimulation. We will not present the definition here, however it closely resembles the definition of rooted branching bisimulation we give in Section 5.2 for configurations.

ACP_ϵ^τ is parameterized over an action-alphabet A , which can be chosen dependent on the application domain. For the purpose of studying interactive functional programs, we assume that A contains, for all

$t \in \mathbf{String}$, $s(t)$, $r(t)$, and $c(t)$. The actions $s(t)$ and $r(t)$ describe the sending and receiving of t respectively. The action $c(t)$ is the result of the successful communication (synchronization) of the actions $s(t)$ and $r(t)$.

ACP_τ^r is also parameterized over an associative and commutative, binary communication function $\gamma : A \times A \rightarrow A$, which can be chosen dependent on the application domain. We define the partial function γ such that one ‘send’ and one ‘receive’ with identical arguments together make one ‘communication’ with that argument. This is expressed by the following equations: for all $t \in \mathbf{String}$

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

These choices allow us to use ACP_τ^r 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 $s(t)$.

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 a functional program; they belong to the environment.

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

4. EXAMPLE OF A CONCURRENT SYSTEM

We present a small example concerning four processes: a dispatcher, two filters, and a one-place buffer. The two filters F_1 and F_2 are realised as functional programs $\mathbf{f1}$, $\mathbf{f2}$. The dispatcher D has two output ports, the two filters have an input port and an output port each, and the one-place buffer B has two input ports and one output port. The processes are connected by ports (channels), numbered 1 to 5, as shown in Figure 1.

We assume $\gamma(s_p(x), r_p(x)) = c_p(x)$ for the ports $p = 1, 2, 3, 4$ and $x \in \mathbf{String}$, and $\gamma(a_1, a_2) = \delta$, otherwise, as before. The function $\mathbf{f1}$ adds 1 to the number represented by its input string, whereas $\mathbf{f2}$ subtracts 1. We give the program of F_1 .

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

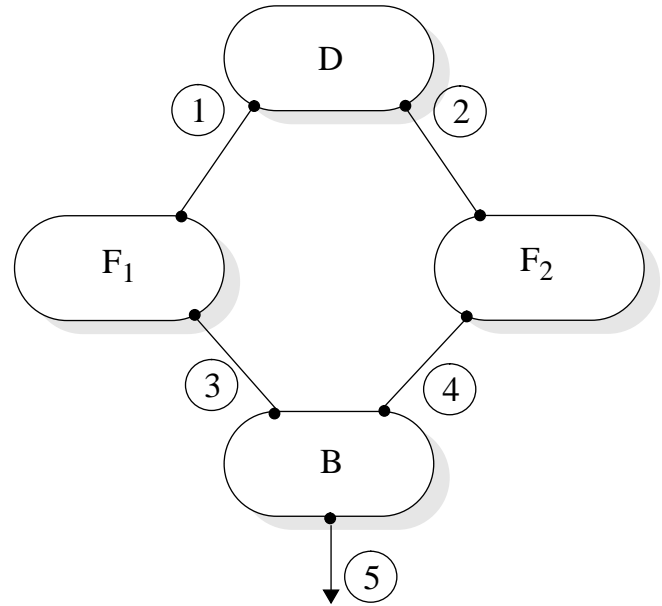


FIGURE 1. Configuration for example protocol

```
g (x : xs) = ((f1 x) : (g xs))
f1 :: String -> String
f1 "1" = "2"
f1 "2" = "3"
...
```

The program for F_2 is similar, but now $\mathbf{f2}$ "2" = "1", $\mathbf{f2}$ "3" = "2", etc. Using the results of Section 5.3 we find the process equations

$$\begin{aligned} F_1 &= \left(\sum_{x \in \mathbf{String}} r_1(x) \cdot s_3(\mathbf{f1}(x)) \right) \cdot F_1, \\ F_2 &= \left(\sum_{x \in \mathbf{String}} r_2(x) \cdot s_4(\mathbf{f2}(x)) \right) \cdot F_2, \end{aligned}$$

and we postulate that the dispatcher and the buffer satisfy

$$\begin{aligned} D &= s_1("1") \cdot s_2("2"), \\ B &= \left(\sum_{x \in \mathbf{String}} (r_3(x) + r_4(x)) \cdot s_5(x) \right) \cdot B, \end{aligned}$$

which means that D issues "1" over port 1 and next issues "2" over port 2. The buffer is willing to accept its input from either port 3 or port 4, and then offers its contents to port 5 (becoming a fresh B again).

Now we can analyse the system built as the parallel composition of all components,

$$S = \tau_I(\partial_H(D \parallel F_1 \parallel F_2 \parallel B)),$$

where

$$I = \{c_p(x) \mid 1 \leq p \leq 4, x \in \mathbf{String}\}$$

indicates the communication actions to be turned into the silent step τ by means of the operator τ_I and

$$H = \{s_p(x), r_p(x) \mid 1 \leq p \leq 4, x \in \mathbf{String}\}$$

indicates the set of ‘loose’ send and receive actions (s_p and r_p) that are blocked by means of the operator ∂_H . This is the ACP way of stating that a send matches a receive and that outside the system this match is viewed as a τ step. Now **f1** gets "1" and yields "2", whereas **f2** gets "2"+ and yields "1". These ‘yields’ may arrive in either order at B . Using the laws of ACP we find that $S = \tau \cdot (\tau \cdot s_5("2") \cdot s_5("1")) + \tau \cdot s_5("1") \cdot s_5("2") \cdot \delta$. So in this tiny protocol, we see how the choice introduced by the parallel composition leads to a non-deterministic choice observable at port 5. If we replace D by $s_1("1") \cdot s_2("3")$ we find that the composed system equals $\tau \cdot s_5("2") \cdot s_5("2") \cdot \delta$, which means that outside the system, it is impossible to tell according to which scenario the two "2" values were obtained.

5. INPUT/OUTPUT BEHAVIOUR

If we want to view a Gofer program \mathbf{g} as a process, we denote it as $\mathcal{P}[\mathbf{g}]$. It would be nice if we could extract the ACP_ϵ^τ equations for $\mathcal{P}[\mathbf{g}]$ from the Gofer program \mathbf{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 [6], but instead of giving a labeled transition system in one step, we separate the internals of Gofer (a term rewriting system) from the external behaviour (an operational semantics 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 [7] with a first set of rules called *operational rewrite rules* (a term rewriting system) and a second set whose elements are called *transition rules*.

5.1. Functional rewrite relation

Now we define an operational semantics which shall make the operational behaviour of the Gofer interpreter for a given program \mathbf{g} completely explicit. First, we set out to define a term rewriting system and a strategy.

We restrict ourselves to such \mathbf{g} only which have type $[\mathbf{String}] \rightarrow [\mathbf{String}]$. Then the state of a functional program is represented by a configuration, i.e., a term in which only the artificial variable \textcircled{c} is allowed. The reason for including such a variable is that it represents the tail part of a list that is not available yet. Typically, the initial configuration of a functional program \mathbf{g} is then given by $(\mathbf{g}\textcircled{c})$. The configurations of functional programs are defined by the following BNF rule:

$$\begin{aligned} \langle \text{config} \rangle ::= & \textcircled{c} \\ & \mid \text{"string"} \\ & \mid [] \end{aligned}$$

$$\begin{aligned} & \mid (\langle \text{config} \rangle : \langle \text{config} \rangle) \\ & \mid \langle \text{id-f} \rangle \\ & \mid (\langle \text{id-f} \rangle \langle \text{config} \rangle +) \end{aligned}$$

We write \rightarrow for the functional reduction relation of Gofer (choosing an outermost redex from the reductions β , π_1 , π_2)³. We assume that the strategy is leftmost outermost.

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 complication is demonstrated by the following program:

$$\begin{aligned} \mathbf{g} (\text{"foo"} : []) &= (\text{"goodbye"} : []) \\ \mathbf{g} (\mathbf{x} : \mathbf{x}\mathbf{s}) &= (\text{"bye"} : []) \end{aligned}$$

After input of **foo** the interpreter will wait for a second line. 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*.

The binary test for match, say, when checking a configuration c and the left-hand side 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 $\text{match}(c, p) = \text{dontknow}$ (‘unknown’). For function names f , configurations c_1, \dots, c_n , terms t_1, \dots, t_n , and pattern p it is defined inductively as follows:

$$\begin{aligned} \text{match}(f, f) &= \text{true} \\ \text{match}((f c_1 \dots c_n), f t_1 \dots t_n) &= \bigwedge_{1 \leq i \leq n} \text{match}(c_i, t_i) \\ \text{match}(c, p) &= \text{false} \quad (\text{otherwise}) \end{aligned}$$

where \bigwedge on $\{\text{true}, \text{dontknow}, \text{false}\}$ is given by the following truth table:

\bigwedge	<i>true</i>	<i>dontknow</i>	<i>false</i>
<i>true</i>	<i>true</i>	<i>dontknow</i>	<i>false</i>
<i>dontknow</i>	<i>dontknow</i>	<i>dontknow</i>	<i>false</i>
<i>false</i>	<i>false</i>	<i>false</i>	<i>false</i>

Thus we use a three-valued logic in the spirit of Kleene [8].

The above definition makes use of a binary test for checking a configuration c and a term t : $\text{match}(c, t)$. For configurations c, c_i , variable name v , function name f , and terms t_i and term t not a variable name, this

³We define π_1 as the rule $\text{head}(x:xs) = x$ and π_2 as the rule $\text{tail}(x:xs) = xs$.

binary test is defined as follows:

$$\begin{aligned}
\text{match}(c, v) &= \text{true} \\
\text{match}(\odot, \text{"string"}) &= \text{dontknow} \\
\text{match}(\odot, \square) &= \text{dontknow} \\
\text{match}(\odot, (t_1:t_2)) &= \text{dontknow} \\
\text{match}(\odot, f) &= \text{false} \\
\text{match}(\odot, (ft_1 \cdots t_n)) &= \text{false} \\
\text{match}(\text{"string"}, t) &= t \equiv \text{"string"} \\
\text{match}(\square, t) &= t \equiv \square \\
\text{match}((c_1:c_2), t) &= t \equiv (t_1:t_2) \wedge \text{match}(c_1, t_1) \\
&\quad \wedge \text{match}(c_2, t_2) \\
\text{match}(f, t) &= t \equiv f \\
\text{match}((fc_1 \cdots c_n), t) &= t \equiv (ft_1 \cdots t_n) \\
&\quad \wedge \bigwedge_{1 \leq i \leq n} \text{match}(c_i, t_i).
\end{aligned}$$

If the configuration 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 $\text{match}(c, p_j) = \text{false}$.

In our term rewriting system the earlier mentioned complication, demonstrated by the program \mathbf{g} , is explained because

$$\text{match}(\mathbf{g}(\text{"foo":}\odot), \mathbf{g}(\text{"foo":}\square)) = \text{dontknow}.$$

Actually we can abstract from some of the details of Gofer and in the definition of the semantics $\mathcal{P}[\mathbf{g}]$ 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, called an abstract functional program [9].

DEFINITION 5.1. For an identifier \mathbf{g} we say that the six-tuple

$$\langle T, \square, :, \odot, [:=], \psi \rangle$$

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

1. $\square \notin \text{dom}(\psi)$,
2. if $\psi(x) = x'$ then $\psi(x:xs) = (x':xs)$,
3. if $\odot \notin c$ and $c \neq \square$ and $c \neq (x:xs)$ then $c \in \text{dom}(\psi)$.

The notation $\odot \notin c$ is indicating that the variable \odot does not occur in configuration c . \blacksquare

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 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 $\square \equiv (x:xs)$ holds. Each correct Gofer program of the form proposed in Section 2 realizes an abstract functional program, notably by adopting the ψ derived from the rules $\{p_i | i = 1, 2, \dots\}$ of the program where $c \in \text{dom}(\psi)$ iff $\text{match}(c, p_i) = \text{true}$ for some i , and $\text{match}(c, p_j) = \text{false}$ for all $j < i$. But of course an abstract functional program could be realized in another lazy functional programming language too.

Usually we write $c \rightarrow_\psi c'$ or even just $c \rightarrow c'$ instead of $\psi(c) = c'$. We write $x \not\rightarrow_\psi$ or even just $x \not\rightarrow$ if $x \notin \text{dom}(\psi)$. So \rightarrow is a functional reduction relation, i.e., a term rewriting system together with a reduction strategy.

5.2. Operational semantics for input/output behaviour

Now we define an operational semantics, sometimes also called ‘action relation’. We shall define a binary action relation \xrightarrow{a} on configurations for each atomic action $a \in A$ and a unary termination predicate \Downarrow on configurations. Intuitively $c \xrightarrow{r(x)} c'$ describes the transformation of configuration c into configuration c' due to the input of x . Similarly, $c \xrightarrow{s(x)} c'$ denotes the transformation of c into c' due to the output of x . Finally, $c \xrightarrow{\tau} c'$ denotes the internal rewriting of configuration c into c' . The predicate $c \Downarrow$ denotes that no interaction with the environment is to take place any more, i.e., no more inputs are consumed and outputs are produced, i.e., the functional program has the option to terminate immediately and successfully.

TABLE 3. Operational semantics for I/O behaviour

$\frac{c \equiv \square}{c \Downarrow}$	$\frac{\neg(c \equiv (x:xs) \wedge x \not\rightarrow) \quad c \rightarrow c'}{c \xrightarrow{\tau} c'}$
$\frac{c \equiv (x:xs) \quad \odot \notin x \quad x \not\rightarrow}{c \xrightarrow{s(x)} xs}$	$\frac{c \equiv (x:xs) \quad \odot \in x \quad x \not\rightarrow}{c \xrightarrow{r(z)} c[\odot := (z:\odot)]}$
$\frac{c \neq \square \quad c \neq (y:ys) \quad c \not\rightarrow}{c \xrightarrow{r(x)} c[\odot := (x:\odot)]}$	

The rules in Table 3 are organized 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 \Longrightarrow or the termination relation \Downarrow . The first rule expresses that the functional program is ready. The second rule expresses that at this point no output can be produced and that some internal rewriting is performed. The third rule expresses that an output can be produced since a value x is available which does not need any more inputs and which does not invoke internal rewriting. The fourth rule expresses that there is more input needed before an output can be produced since the head of the configuration list contains a \odot and no internal rewriting is possible. The fifth rule expresses that no output can be produced (since the configuration is a function application) nor internal rewriting can be performed and therefore input is needed.

The action relation \Longrightarrow and the termination predicate \Downarrow are defined as the smallest relations satisfying these rules.

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 $OS(g)$ for the triple $(c_0, \Downarrow, \Longrightarrow)$.

Now we set out to use the operational semantics 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_g and \Longrightarrow_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’. The notion of equivalence that we propose is rooted branching bisimulation. We introduce some convenient abbreviations: $c \xrightarrow{a} c'$ abbreviates $c \xrightarrow{a} c' \vee (a \equiv \tau \wedge c \equiv c')$, i.e., if $a \equiv \tau$ it means zero or one τ -step, and otherwise it simply means an a -step, and $c \Rightarrow c'$ denotes the reflexive, transitive closure of the relation $\xrightarrow{\tau}$, i.e., a sequence of zero or more τ -steps. We define that a relation $R \subseteq C_g \times C_g$ is a branching bisimulation if it satisfies: for all configurations c, d such that cRd

1. if $c \xrightarrow{a} c'$ (for $a \in A_\tau$), then there exist configurations d' and d'' such that $d \Rightarrow d'' \xrightarrow{a} d'$ and cRd'' and $c'Rd'$,
2. if $c \Downarrow$, then there exists a configuration d' such that $d \Rightarrow d'$, $d' \Downarrow$, and cRd' , and
3. similarly when the roles of c and d are interchanged.

In fact this is the optimized version of branching bisimulation [10]. Two configurations c_1 and c_2 are called rooted branching bisimilar, notation $c_1 \xrightarrow{rb} c_2$, if there exists a branching bisimulation R such that $c_1 R c_2$ and moreover, if $c_1 \xrightarrow{a} d_1$, then there exists a configuration d_2 such that $c_2 \xrightarrow{a} d_2$ and $d_1 R d_2$ (the ‘root condition’), and if $c_1 \Downarrow$, then $c_2 \Downarrow$, and vice versa.

5.3. Process semantics

We can make the process-semantics very explicit by means of a single equation [9].

DEFINITION 5.2. Define $\mathcal{P}[g] := \mathcal{P}[(g \odot)]$, where

$$\begin{aligned} \mathcal{P}[c] &= [c \equiv \square] \rightarrow c \\ &+ [\neg(c \equiv (x:xs) \wedge x \not\rightarrow) \text{ and } c \rightarrow c'] \\ &\quad \rightarrow \tau \cdot \mathcal{P}[c'] \\ &+ [c \equiv (x:xs) \text{ and } \odot \notin x \text{ and } x \not\rightarrow] \\ &\quad \rightarrow s(x) \cdot \mathcal{P}[xs] \\ &+ [c \equiv (x:xs) \text{ and } \odot \in x \text{ and } x \not\rightarrow] \\ &\quad \rightarrow \sum_z r(z) \cdot \mathcal{P}[c[\odot := (z:\odot)]] \\ &+ [c \not\equiv \square \text{ and } c \not\equiv (y:ys) \text{ and } c \not\rightarrow] \\ &\quad \rightarrow \sum_x r(x) \cdot \mathcal{P}[c[\odot := (x:\odot)]]. \quad \blacksquare \end{aligned}$$

We used the notation $[...] \rightarrow$ to denote guards [3]. Please note that all guards describe syntactic conditions on terms, and do not involve any assumptions on the action relation itself.

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 $\tau_{\{i\}}(\mathcal{P}[c])$, 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 [2] can be applied and then an infinite sequence of τ steps turns into δ and then disappears).

There is a very strong relation between the operational semantics for configurations of abstract functional programs and the operational semantics of the corresponding ACP_ϵ^τ term. A configuration can perform a transition to another configuration if and only if this transition is also possible between the corresponding ACP_ϵ^τ terms. A similar result holds for the termination predicate: a configuration can terminate if and only if its corresponding ACP_ϵ^τ term can do so.

THEOREM 5.1 (**Compliance**). Consider an arbitrary abstract functional program g and two configurations c and c' and $a \in A_\tau$. Then

$$OS(g) \models c \Downarrow \quad \text{iff} \quad T(ACP_\epsilon^\tau) \models \mathcal{P}[c] \Downarrow$$

and

$$OS(g) \models c \xrightarrow{a} c' \quad \text{iff} \quad T(ACP_\epsilon^\tau) \models \mathcal{P}[c] \xrightarrow{a} \mathcal{P}[c'].$$

Proof. Suppose that $c \Downarrow$. From the operational rules one easily establishes that this must be due to $c \equiv \square$. Then, clearly, $\mathcal{P}[c] = \epsilon$, and hence $\mathcal{P}[c] \Downarrow$. Next, suppose that $\mathcal{P}[c] \Downarrow$. Since the conditions occurring in $\mathcal{P}[c]$ are disjoint this must be due to $c \equiv \square$, hence $c \Downarrow$. Thus we have proven $c \Downarrow$ iff $\mathcal{P}[c] \Downarrow$.

Suppose that $c \xrightarrow{a} c'$. This must be due to either one of the following cases:

1. $\neg(c \equiv (x:xs) \wedge x \not\rightarrow) \wedge c \rightarrow c'$, with $a \equiv \tau$. Then, $\mathcal{P}[c] = \tau \cdot \mathcal{P}[c']$, and hence $\mathcal{P}[c] \xrightarrow{\tau} \mathcal{P}[c']$.
2. $c \equiv (x:xs) \wedge \textcircled{c} \notin x \wedge x \not\rightarrow$, with $a \equiv s(x)$ and $c' \equiv xs$. Then, $\mathcal{P}[c] = s(x) \cdot \mathcal{P}[xs]$, and hence $\mathcal{P}[c] \xrightarrow{s(x)} \mathcal{P}[xs]$.
3. $c \equiv (x:xs) \wedge \textcircled{c} \in x \wedge x \not\rightarrow$, with $a \equiv r(z)$ for some z and $c' \equiv c[\textcircled{c} := (z:\textcircled{c})]$. Then, $\mathcal{P}[c] = \sum_z r(z) \cdot \mathcal{P}[c[\textcircled{c} := (z:\textcircled{c})]]$, and hence $\mathcal{P}[c] \xrightarrow{r(z)} \mathcal{P}[c[\textcircled{c} := (z:\textcircled{c})]]$.
4. $c \not\equiv \square \wedge c \not\equiv (y:ys) \wedge c \not\rightarrow$, with $a \equiv r(x)$ for some x and $c' \equiv c[\textcircled{c} := (x:\textcircled{c})]$. Then, $\mathcal{P}[c] = \sum_x r(x) \cdot \mathcal{P}[c[\textcircled{c} := (x:\textcircled{c})]]$, and hence $\mathcal{P}[c] \xrightarrow{r(x)} \mathcal{P}[c[\textcircled{c} := (x:\textcircled{c})]]$.

Clearly, in all cases $\mathcal{P}[c] \xrightarrow{a} \mathcal{P}[c']$. The proof in the other direction is similar. ■

As a direct consequence of the previous theorem, we find that the notions rooted branching bisimulation on configurations and on ACP_ϵ^τ terms correspond.

THEOREM 5.2 (Soundness). Consider an abstract functional program g and two configurations c and d . Then,

$$\text{if } T(\text{ACP}_\epsilon^\tau) \models \mathcal{P}[c] \xleftrightarrow{\text{rb}} \mathcal{P}[d], \text{ then } \text{OS}(g) \models c \xleftrightarrow{\text{rb}} d.$$

THEOREM 5.3 (Completeness). Consider an abstract functional program g and two configurations c and d . Then,

$$\text{if } \text{OS}(g) \models c \xleftrightarrow{\text{rb}} d, \text{ then } T(\text{ACP}_\epsilon^\tau) \models \mathcal{P}[c] \xleftrightarrow{\text{rb}} \mathcal{P}[d].$$

Since the axioms of ACP_ϵ^τ are sound with respect to rooted branching bisimilarity [2], we have that these may be used in deriving the equality of the configurations.

THEOREM 5.4. Consider an abstract functional program g and two configurations c and d . Then

$$\text{if } \text{ACP}_\epsilon^\tau \vdash \mathcal{P}[c] = \mathcal{P}[d], \text{ then } \text{OS}(g) \models c \xleftrightarrow{\text{rb}} d.$$

Proof. Suppose that $\text{ACP}_\epsilon^\tau \vdash \mathcal{P}[c] = \mathcal{P}[d]$. Then, by the soundness of the axioms of ACP_ϵ^τ , we have $T(\text{ACP}_\epsilon^\tau) \models \mathcal{P}[c] \xleftrightarrow{\text{rb}} \mathcal{P}[d]$. Then by Theorem 5.2, we have $\text{OS}(g) \models c \xleftrightarrow{\text{rb}} d$. ■

5.4. Some relations with denotational semantics

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.

Semantically, functional programs are viewed as functions from streams to streams, where, roughly speaking, streams are sequences, extended with an undefined element \perp [11]. The question whether ‘referential transparency’ holds for all I/O aspects too, is a subtle one. We expect that it is no problem to define a denotational semantics $\mathcal{F}[\mathbf{g}]$ which faithfully reflects the I/O behaviour of the functional program \mathbf{g} . This corresponds to the investigations of Thompson [12], who found a stream-based denotational semantics of Miranda adequate for reasoning about trace-based I/O behaviour in the context of a special set of operators, including sequential composition.

When we take non-termination into account, we shall find that there exist functional programs $\mathbf{g1}$ and $\mathbf{g2}$ which denote mathematically equivalent functions, but which have different I/O behaviour. Define two programs $\mathbf{g1}$ and $\mathbf{g2}$ as follows:

$$\begin{aligned} \mathbf{g1} \ [] &= \mathbf{g1} \ [] \\ \mathbf{g1} \ (\mathbf{x} : \mathbf{xs}) &= \mathbf{g1} \ (\mathbf{x} : \mathbf{xs}) \\ \mathbf{g2} \ [] &= \mathbf{g2} \ [] \\ \mathbf{g2} \ (\mathbf{x} : \mathbf{xs}) &= \mathbf{g2} \ \mathbf{xs} \end{aligned}$$

In a typical denotational semantics of the programs we obtain, for stream s , that

$$\mathcal{F}[\mathbf{g1}](s) = \mathcal{F}[\mathbf{g2}](s) = [\perp],$$

that is, no output is ever generated. Thus, $\mathcal{F}[\mathbf{g1}] = \mathcal{F}[\mathbf{g2}]$. But $\mathbf{g1}$ performs an infinite rewriting without consuming input (but the first line), whereas $\mathbf{g2}$ consumes all input (still producing nothing). So $\mathcal{P}[\mathbf{g1}] = \sum_x r(x) \cdot \tau^\omega$ whereas $\mathcal{P}[\mathbf{g2}] = \sum_x r(x) \cdot \mathcal{P}[\mathbf{g2}]$, and clearly those are not rooted branching bisimilar (they are not even trace equivalent).

The syntactic forms of the definitions of $\mathbf{g1}$ and $\mathbf{g2}$ 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 mismatch between the operational semantics and the denotational semantics is well-known, see [6], footnote 2, which refers to an argument of Hughes in 1988). The full advantage of referential transparency is not applicable.

The fact that functional denotational semantics (stream semantics) is inadequate to describe concurrent behaviour is known since long. In particular Brock and Ackerman [13] have shown in 1981 that a functional characterisation of non-deterministic behaviour is problematic (this is known as the Brock-Ackerman anomaly [14]). Their example involves a feedback loop over the process whose behaviour is to be characterised.

6. EXAMPLE PROGRAMS

In this section we give two example programs and the process equations that describe their I/O behaviour. Besides the syntax used in Section 2 we also use guarded equations. Furthermore, the notation $[\mathbf{e1}, \mathbf{e2}, \dots, \mathbf{en}]$ is used for the list $(\mathbf{e1} : (\mathbf{e2} : \dots (\mathbf{en} : []) \dots))$.

6.1. Example of a 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

main = interact (unlines . g . lines)

g :: [Line] -> [Line]
g = map updline

updline :: Line -> Line
updline = map toUpper
```

Note that `map` and `toUpper` are functions from the standard prelude of Gofer: `map f xs` applies the function `f` to each element of the list `xs` returning the corresponding list of results and `toUpper` replaces a lower case letter by the corresponding upper case letter.

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 $\mathcal{P}[\mathbf{g}]$. If we want to consider `g` as a mathematical function we denote it as $\mathcal{F}[\mathbf{g}]$. From the definition of `g` we see that the first element of the result $\mathbf{g}(l:ls)$ 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 satisfies the ACP equation:

$$\mathcal{P}[\mathbf{g}] = \tau \cdot \sum_{x \in \text{Line}} r(x) \cdot s(\mathcal{F}[\text{updline}](x)) \cdot \mathcal{P}[\mathbf{g}],$$

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

$$\mathcal{F}[\mathbf{g}](x:xs) = \mathcal{F}[\text{updline}](x) : \mathcal{F}[\mathbf{g}](xs).$$

6.2. Example of a 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 :: Dict -> Word -> Word
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",
          "for","form","they","are","not"]
```

The functions `words` and `unwords` are taken from the standard Gofer prelude. 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 $\mathcal{P}[\mathbf{g}] = U_{iniDict}$ where for all `d` of type `Dict` we have that U_d is given by

$$U_d = \tau \cdot \sum_{l \in \text{Line}} r(l) \cdot s(l') \cdot U_d'$$

where

$$l' \equiv \mathcal{F}[\text{((unwords.(map (updword d)).words) l)}]$$

and

$$d' \equiv \mathcal{F}[\text{(add (words l) d)}].$$

This is consistent with the functional behaviour for which we check from the Gofer program text that $\mathcal{F}[\mathbf{g}] = u(\text{iniDict})$ where for all strings l and ls ,

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

7. EXPLOITATION OF THE PROCESS SEMANTICS

The theoretical sections of this paper allow us to use the process theoretic equations for reasoning about the process behaviour of (certain kinds of) functional programs. We propose two practical techniques for exploiting this connection between the functional programming world and the process-oriented world.

1. template generation,
2. process-oriented simulation.

We describe these techniques next.

7.1. Template generation

Template generation separates the development of a functional program from the concern of analysing its concurrent behaviour. A designer can indicate the desired pattern of communication behaviour and use it as the input for a generator, which produces two outputs, see Figure 2. The generator is an example of a component-generator in the sense proposed in [15].

The first output is a template of a functional program. The user must still fill in functions, e.g. called **f0**, **f1**, **f2**, **f3**, **f4**, and **f5** (some patterns do not need all functions). Function **f0** describes the initial state, **f1** is the reaction telling how a single line of input is mapped onto zero or more lines of output, **f2** describes a kind of reflection, that is, how to update the internal state, **f3** is the filter criterion, determining which lines will get an answer and which ones will not get an answer, **f4** is the stop criterion and **f5** is the sequence of lines that form the menu.

The second output describes the process behaviour of the first output as ACP_τ⁷ equations. This is useful for analysis and simulation. The designer must add other processes: those not developed in a functional program, or describing a network context.

Choosing software-engineering solutions for the formalisms involved, we experimented with Gofer and PSF [3]. The latter is a practical form of ACP, with an ASCII syntax and simulation tools. The approach could be instantiated with other languages as well.

A 'pattern of communication behaviour' is a regular expression showing the alternation between send and receive actions (more sophisticated notions of pattern could be developed). The generator is based on the adopted restriction to Landin-style stream I/O.

As an example, let the pattern be $send^* receive send^*$. This indicates a program which begins with producing zero or more lines of output, then waits for one line of input and then sends zero or more lines of output. An

example of such a program is a menu-based generator: the first lines of output give the menu, the input reflects the user-choice, and the subsequent lines are the generator's output. This yields the Gofer template shown in Figure 2. The functions **f1** and **f5** are given by their type declaration only. In this example, function **f5** is the menu. Function **f1** is the reaction of the functional program on the choice from the menu. The essential PSF equations that are the (second) output of the generator are shown in Figure 2 too.

7.2. Process-oriented simulation

Process oriented simulation applies results and tools from the area of process theory to the equations describing the communication behaviour of a program. Equations can be obtained from a generator as above, or by other means (manual or theoretical work). We did a small case study to explore this idea, which we summarize now (more details are in [16]). It is about a service network with a manager and a psychiatrist, see Figure 3. Two processes are implemented in Gofer: a manager (interpreting messages about money, its internal state keeping track of payments), and a psychiatrist (based on M. Jones' version of Eliza, standard Gofer distribution). The manager is based on the state-based pattern $(receive [send])^\infty$.

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. The router's output '1' is connected to the user's console whereas output '2' goes via a buffer to Eliza.

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. Firstly, we made 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 (Sun-OS Unix). We used files for the two main buffers and 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 commands like (**writer foo**), and (**reader foo**) | (**gofer manager.gs**) | (**router bar**), each in a separate X window shell. 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, etc. and all other lines are thrown away.

Secondly, we specified the buffers and the router in PSF, which in combination with an algebraic specification of the data-manipulation functions and the generated equations allowed us to simulate the manager process in PSF. We ran the simulator, and played around

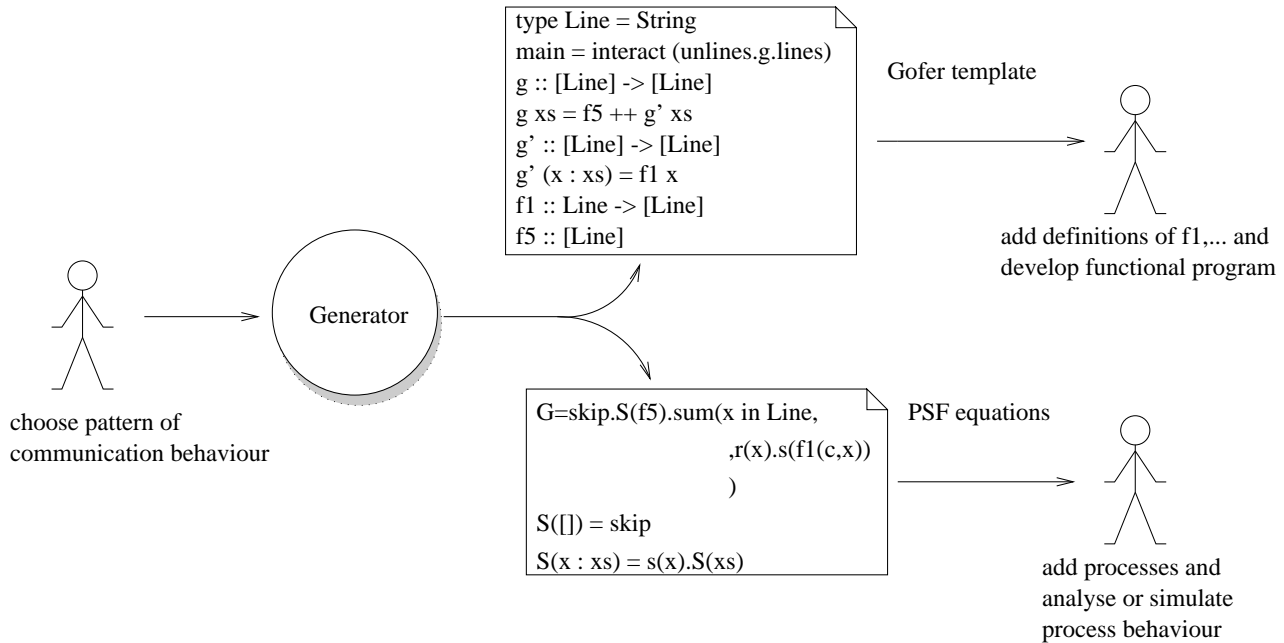


FIGURE 2. Template generation

with it for some time, noting that the behaviour was consistent with the ‘real’ system.

Although some aspects of the experimental set-up are clumsy, the experiments show the feasibility of the approach. They show that non-trivial networks of functional and non-functional programs can be put into operation: the full power of functional programming can be used for the data transformation aspects in the functional processes, and the full power of process-theoretic reasoning and simulation can be used to analyse the concurrency aspects. The clumsiness is due to the fact that we retrofitted an existing functional language and an existing concurrency workbench to our approach. But when a functional language’s I/O system were to be designed using the principle that the I/O behaviour must be governed by known process-theoretic equations, there are better opportunities to have the tools well-integrated into a common environment.

8. RELATED WORK

Sangiorgi [17], following Milner [18] examines the encoding of the lazy λ -calculus into the π -calculus. Focus is on constructing λ -models and using special forms of bisimulation to analyse their properties.

Broy showed already in [11] 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.

Gordon [6] gives an operational semantics of a Haskell fragment and uses it to derive process-theoretic properties using methods from Milner’s CCS. See also the remarks in the beginning of section 5.

Plasmeijer et al. [19] proposes a powerful I/O mechanism 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, [19] addresses the problem the other way around, turning the world into a set of objects being manipulated by the functional program.

Thompson [12] analyses the interactive behaviour of functional programs in the lazy Miranda system. The aim is to avoid multiparadigm programming by providing combinators for `get_char`, `put_char`, `apply` (state assignment), `sq` (sequential composition), `alt` (a kind of conditional), `skip`, and `while`. In a way this is opposite to the work of the present paper: whereas Thompson moves the operators of other paradigms into the functional programs, the present paper lets the functional programs appear as components in a non-functional environment. The subtleties concerning I/O and lazy evaluation are analysed in great detail (for example the fact that pattern matching can delay output). In part II of his work, Thompson provides a theoretical justification of the proposed operators (but leaving out systems with an internal state). A notion called *weak trace of* (a functional program) is defined in terms of the function’s denotational semantics, applied to \perp -terminated lists. It is found that this identifies functional programs whose I/O behaviour differs. Then the improved notion called *trace of* (a functional program) is defined, a trace being a sequence of read actions, write actions, and a special tick \checkmark . The tick marks the position where the result is completed, in the sense that no more input is needed. A similar notion called *terminal trace of* is defined, implying that if some trace is a terminal trace of f : $\text{input} \rightarrow (\text{input}, \text{output})$, then f passes its superflu-

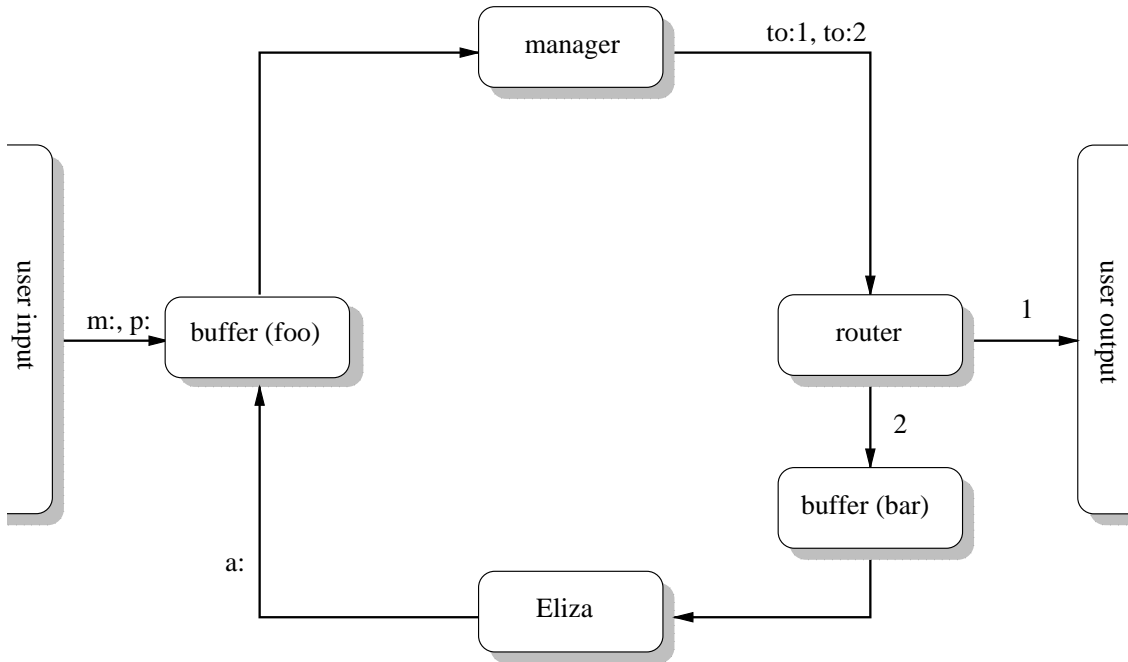


FIGURE 3. Service network structure

ous input as a kind of continuation to the first element in its result-pair (with the effect that it is to be used by a subsequent function when combined with `sq`). These notions of ‘trace’ and ‘terminal trace’ (both of which are compatible with the denotational semantics), capture all relevant information needed to reason about the process behaviour in a manner which is compositional with respect to the proposed combinators (`get_char`, `put_char`, `sq` etc.). When combined, the notions of ‘trace’ and ‘terminal trace’ in [12] can be compared to the operational semantics proposed in the present paper. The tick \checkmark plays a role similar to the \downarrow predicate. An important difference is that the operational semantics captures the behaviour in a manner which is compositional with respect to more process-theoretic operators than sequential composition, notably \parallel . Also, the operational semantics distinguishes deadlock from divergence.

Thomsen et al. [20] investigate a hybrid language called Facile, where functional programs are integrated with distribution concepts, including channels, exceptions, `spawn` instructions, and time-outs. Processes use functions. The semantics is defined by two labeled transition systems, one for expression evaluation, and the other for process execution. The core language, which is formally studied, only deals with arithmetic and Boolean operators (no lists, unlike the present paper). The evaluation strategy is leftmost innermost. The transition system attaches a τ label to each β -reduction step (as in the present paper). In order to deal with the language’s timing aspects, a timed version of process algebra is used.

Niehren [21] investigates an embedding of call-by-need and call-by-value λ -calculi into a subset of Milner’s π -calculus, called δ_0 . Focus is on proving confluence and on comparing the complexity of calculations in the call-by-need and call-by-value calculi.

Dybjer and Sander [22, 23] consider networks of stream transformers, programmed in the lazy functional language Miranda. Nondeterminism is introduced for the purpose of modeling an unreliable channel, using an oracle stream of bits. Following Kahn, the network is described by a transfer function *trans*, whose recursion equations reflect the network’s topology. The alternating bit protocol is described and proven correct, using the μ -calculus of Scott and de Bakker. The conversion rules for functional terms are added as axioms to the μ -calculus.

9. CONCLUDING REMARKS

We defined an operational semantics for the input/output behaviour of a functional program. This operational semantics is based on the lazy evaluation mechanism and the leftmost outermost reduction strategy. With little effort we can interpret a functional program g as a recursive specification in the process algebra ACP_τ^τ . It is shown that the standard operational semantics of ACP_τ^τ and the operational semantics for input/output behaviour of functional programs coincide with this interpretation. As a consequence we can use the axioms of ACP_τ^τ for reasoning about the equality (rooted branching bisimulation) of functional programs. Instead of taking rooted branching bisimulation as the notion of equivalence on configurations,

one can also consider other notions of equivalence, for example strong bisimulation or trace semantics. This does not affect the compliance between the operational semantics for configurations and the operational semantics for the corresponding ACP_{ϵ}^{τ} terms. However, considering a different notion of equivalence does affect the use of the axioms of ACP_{ϵ}^{τ} in reasoning about equality; these axioms possibly have to be replaced by others (much research has been done already on the axiomatisation of different equivalences, see e.g. van Glabbeek's thesis [24]).

The restriction to Landin-style stream I/O can be relaxed by considering a placeholder \odot_i for every input port i of the reactive program. The theory can easily be extended to incorporate guarded equations as well.

A survey of the theory developed so far is given in Figure 4.

A generator-based approach is discussed in [16]. A simulation of the input/output behaviour of functional programs based on the ACP_{ϵ}^{τ} equations is performed using PSF [3] (Process Specification Formalism).

In our own view, the main contribution of the present paper is to make a synthesis of a number of techniques, most of which are not new as such. Mauw developed PSF for simulation of concurrent systems [3] (many other concurrency workbenches exist). Jonkers proposed component-generators in [15]. Thompson has already explored the tricky examples concerning the interaction of pattern matching and lazy evaluation (see Section 8). Broy, Dybjer and Sander, and Thompson considered functional programs as stream transformers (see Section 8). Milner, Gordon, and Sangiorgi investigated certain connections between lazy functional programs and process theory (see Section 8). The idea of using a placeholder ('mock argument' \odot) for unconsumed input has been used in the context of Haskell and by Gordon. Mislove factorised a transition system's definition into two steps (see section 5). The idea of associating a τ -step with a β -reduction can be found in Niehren (see Section 8). In the present paper, these techniques are turned into a synthesis (theory and applications) which allows rigorous ACP-based reasoning (sufficiently fine-grained to deal with bisimulation and τ steps) about certain lazy functional programs.

ACKNOWLEDGEMENTS

The authors wish to thank Lex Augusteijn, Jos Baeten, Jan Bergstra, Herman Geuvers, Sjouke Mauw, and the anonymous referees for the discussions and the help that contributed to the work presented in this paper.

REFERENCES

- [1] M.P. Jones. An introduction to Gofer 2.20, 1991. Draft report, available electronically at <http://lal.cs.byu.edu/cs532/gofer/docs/goferdoc/goferdoc.html>.
- [2] J.C.M. Baeten and W.P. Weijland. *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1990.
- [3] S. Mauw and G.J. Veltink. *Algebraic Specification of Communication Protocols*, volume 36 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1993.
- [4] P.J. Landin. A correspondence between ALGOL 60 and Church's lambda-notation: Parts I and II. *Communications of the ACM*, 8(2,3):89–101, 158–165, 1965.
- [5] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.
- [6] A.D. Gordon. An operational semantics for I/O in a lazy functional language. In *Conference on functional programming languages and computer architecture (FPCA'93)*, pages 136–145. ACM Press, 1993.
- [7] M.W. Mislove and F.J. Oles. A simple language supporting angelic nondeterminism and parallel composition. In S. Brookes, M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Mathematical Foundations of Programming Semantics, 7th International Conference*, volume 598 of *Lecture Notes in Computer Science*, pages 77–101. Springer-Verlag, 1991.
- [8] S.C. Kleene. *Introduction to metamathematics*. Van Nostrand, Princeton, New Jersey, 1952.
- [9] J.A. Bergstra, 1995. Personal communication.
- [10] T. Basten. Branching bisimilarity is an equivalence indeed! *Information Processing Letters*, 58(3):141–147, 1996.
- [11] M. Broy. Extensional behaviour of concurrent, nondeterministic, communicating systems. In M. Broy, editor, *Control flow and data flow: concepts of distributed programming*, volume F14 of *NATO ASI series*, pages 229–276. Springer-Verlag, 1985.
- [12] S. Thompson. Interactive functional programs: A method and a formal semantics. In D.A. Turner, editor, *Research topics in functional programming*, chapter 10, pages 249–285. Addison-Wesley, 1990.
- [13] J.D. Brock and W.B. Ackerman. Scenarios: A model of non-determinate computation. In J. Díaz and I. Ramos, editors, *Formalization of Programming Concepts*, volume 107 of *Lecture Notes in Computer Science*, pages 252–259. Springer-Verlag, 1981.
- [14] J.R. Russell. Full abstraction for nondeterministic dataflow networks. In *Foundations of Computer Science*, pages 170–175. IEEE, 1989.
- [15] H.B.M. Jonkers. An overview of the sprint method. In J. Woodcock, editor, *Proceedings of Formal Methods Europe (FME '93)*. Springer Verlag, 1993.
- [16] L.M.G. Feijs. Algebraic specification and simulation of lazy functional programs in a concurrent environment. Technical Report CSR 96-20, Eindhoven University of Technology, Department of Computing Science, 1996.
- [17] D. Sangiorgi. An investigation into functions as processes. In *Proceedings of the IX Conference on the Mathematical Foundations of Programming Semantics*, volume 802 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.
- [18] R. Milner. Functions as processes. In *Automata, Languages and Programming 17th Int. Coll.*, volume 443

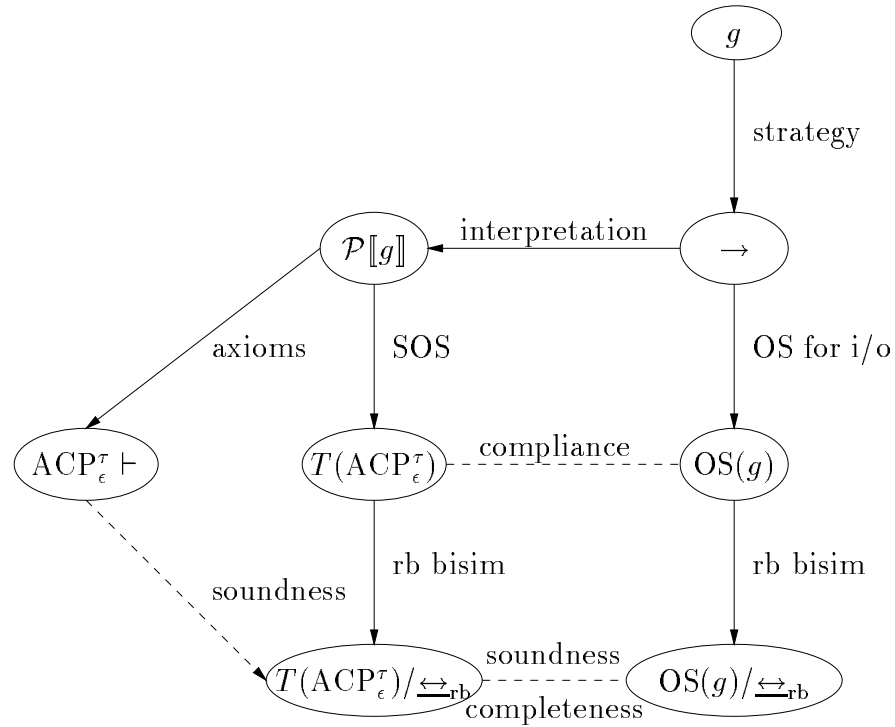


FIGURE 4. Survey of the theory

of *Lecture Notes in Computer Science*, pages 167–180. Springer-Verlag, 1990.

- [19] P.M. Achten, J.H.G. van Groningen, and R.M.J. Plasmeijer. High level specification of I/O in functional languages. In J. Launchbury and P. Sansom, editors, *Functional Programming Glasgow 1992, Workshops in Computing*, pages 1–17. Springer-Verlag, 1993.
- [20] B. Thomsen, L. Leth, and A. Giacalone. Some issues in the semantics of facile distributed programming. Technical Report ECRC-92-32, European Computer-Industry Research Centre, 1993.
- [21] J. Niehren. Functional computation as concurrent computation. In *Conference Record of POPL '96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 333–343. ACM Press, 1996.
- [22] P. Dybjer and H.P. Sander. A functional programming approach to the specification and verification of concurrent systems. *Formal aspects of computing*, 1:303–319, 1989.
- [23] H.P. Sander. *A logic of functional programs with an application to concurrency*. PhD thesis, Chalmers University, Gotenborg, 1992.
- [24] R. J. van Glabbeek. *Comparative Concurrency Semantics and Refinement of Actions*. PhD thesis, Vrije Universiteit Amsterdam, 1990.