Distributed Programming in a Multi-Paradigm Declarative Language*

Michael Hanus

Informatik II, RWTH Aachen, D-52056 Aachen, Germany
hanus@informatik.rwth-aachen.de

Abstract. Curry is a multi-paradigm declarative language covering functional, logic, and concurrent programming paradigms. Curry's operational semantics is based on lazy reduction of expressions extended by a possibly non-deterministic binding of free variables occurring in expressions. Moreover, constraints can be executed concurrently which provides for concurrent computation threads that are synchronized on logical variables. In this paper, we extend Curry's basic computational model by a few primitives to support distributed applications where a dynamically changing number of different program units must be coordinated. We develop these primitives as a special case of the existing basic model so that the new primitives interact smoothly with the existing features for search and concurrent computations. Moreover, programs with local concurrency can be easily transformed into distributed applications. This supports a simple development of distributed systems that are executable on local networks as well as on the Internet. In particular, sending partially instantiated messages containing logical variables is quite useful to implement reply messages. We demonstrate the power of these primitives by various programming examples.

1 Introduction

Curry [9,13] is a multi-paradigm declarative language which integrates functional, logic, and concurrent programming paradigms. Curry combines in a seamless way features from functional programming (nested expressions, lazy evaluation, higher-order functions), logic programming (logical variables, partial data structures, built-in search), and concurrent programming (concurrent evaluation of expressions with synchronization on logical variables). Moreover, Curry provides additional features in comparison to the pure paradigms (compared to functional programming: search, computing with partial information; compared to logic programming: more efficient evaluation due to the deterministic and demand-driven evaluation of functions) and amalgamates the most important operational principles developed in the area of integrated functional logic languages: "residuation" and "narrowing" (see [7] for a survey on functional logic programming).

* This research has been partially supported by the German Research Council (DFG) under grant Ha 2457/1-1 and by the DAAD under the PROCOPE programme.

Curry's operational semantics is based on a single computation model, firstly described in [9], which combines lazy reduction of expressions with a possibly non-deterministic binding of free variables occurring in expressions. Thus, purely functional programming and purely logic programming are obtained as particular restrictions of this model. Moreover, impure features of Prolog (e.g., arithmetic, cut, I/O) are avoided and don't know non-deterministic computations can be encapsulated and controlled by the programmer [12]. For concurrent computations, the evaluation of functions can be suspended depending on the instantiation of arguments, and constraints can be executed concurrently. This provides an easy modeling of concurrent objects as functions synchronizing on a stream of messages. Based on this computation model, we propose to add a new kind of constraint to relate a multiset of incoming messages with a list containing these messages. Such port constraints have been proposed in the context of concurrent logic programming [16] for the local communication between objects. We generalize and embed them into the functional logic language Curry to obtain a simple but powerful mechanism to implement distributed applications that are executable on a network with an unknown number of communication partners.

The paper is structured as follows. In the next section, we review the basics of the operational model of Curry. We introduce and discuss the necessary extensions of this model to support distributed applications in Section 3. We demonstrate the use of these features by several examples in Section 4. Section 5 discusses some implementation issues and Section 6 relates our approach to other existing proposals before we conclude in Section 7.

2 Operational Semantics of Curry

In this section, we sketch the basic computation model of Curry. More details and a formal definition can be found in [9,13].

From a syntactic point of view, a Curry program is a functional program[1] extended by the possible inclusion of free (logical) variables in conditions and right-hand sides of defining rules. Thus, the basic computational domain of Curry consists of *data terms*, constructed from constants and data constructors, whose structure is specified by a set of *data type declarations* like

```
data Bool = True | False
data List a = [] | a : List a
```

`True` and `False` are the Boolean constants and `[]` (empty list) and `:` (non-empty list) are the constructors for polymorphic lists (`a` is a type variable and the type `List a` is usually written as `[a]` for conformity with Haskell). Then, a *data term* is a well-formed expression containing variables, constants, and data constructors, e.g., `True:[]` or `[x,y]` (the latter stands for `x:(y:[])`).

[1] Curry has a Haskell-like syntax [20], i.e., (type) variables and function names start with lowercase letters and the names of type and data constructors start with an uppercase letter. Moreover, the application of f to e is denoted by juxtaposition ("f e").

Functions are operations on data terms whose meaning is specified by (*conditional*) *rules* of the general form "l `|` c `=` r `where` vs `free`" where l has the form $f\, t_1 \ldots t_n$ with f being a function, $t_1, \ldots, t_n$ data terms and each variable occurs only once, the *condition* c is a constraint, r is a well-formed *expression* which may also contain function calls, and vs is the list of *free variables* that occur in c and r but not in l (the condition and the `where` parts can be omitted if c and vs are empty, respectively). A *constraint* is any expression of the built-in type `Constraint` where primitive constraints are equations of the form e_1 `=:=` e_2. A conditional rule can be applied if its condition is satisfiable. A *Curry program* is a set of data type declarations and rules.

Example 1. Assume that the above data type declarations are given. Then the following rules define the concatenation of lists, the last element of a list, and a constraint which is satisfied if the first list argument is a prefix of the second list argument:

```
conc [] ys = ys
conc (x:xs) ys = x : conc xs ys

last xs | conc ys [x] =:= xs = x where x,ys free

prefix ps xs = let ys free in conc ps ys =:= xs
```

If the equation "`conc ys [x] =:= xs`" is solvable, then `x` is the last element of the list `xs`. Similarly, `ps` is a prefix of `xs` if the equation "`conc ps ys =:= xs`" is solvable for some value `ys` (note that existentially quantified variables vs can be introduced in a constraint c by `let` vs `free in` c).

Functional programming: In functional languages, the interest is in computing *values* of expressions, where a value does not contain function symbols (i.e., it is a data term) and should be equivalent (w.r.t. the program rules) to the initial expression. The value can be computed by applying rules from left to right. For instance, we compute the value of "`conc [1] [2]`" by applying the rules for concatenation to this expression:

```
conc [1] [2] → 1 : (conc [] [2]) → [1,2]
```

To support computations with infinite data structures and a modular programming style by separating control aspects [14], Curry is based on a lazy (outermost) strategy, i.e., the selected function call in each reduction step is an outermost one among all reducible function calls. This strategy yields an optimal evaluation strategy [1] and a demand-driven search method [10] for the logic programming part that will be discussed next.

Logic programming: In logic languages, expressions (or constraints) may contain free variables. A logic programming system should compute solutions, i.e., find values for these variables such that the expression (or constraint) is reducible to some value (or satisfiable). Fortunately, it requires only a slight extension of the lazy reduction strategy to cover non-ground expressions and variable instantiation: if the value of a free variable is demanded by the left-hand sides of program rules in order to proceed the computation (i.e., no program rule is applicable if the variable remains unbound), the variable is non-deterministically bound to the different demanded values. For instance, if the function `f` is defined by the rules

```
f 0 = 2
f 1 = 3
```

(the integer numbers are considered as an infinite set of constants), then the expression "`f x`" with the free variable `x` is evaluated to `2` by binding `x` to `0`, or it is evaluated to `3` by binding `x` to `1`. Thus, a single computation step may yield a single new expression (*deterministic step*) or a disjunction of new expressions together with the corresponding bindings (*non-deterministic step*). For inductively sequential programs (these are, roughly speaking, function definitions without overlapping left-hand sides), this strategy, called *needed narrowing* [1], computes the shortest possible successful derivations (if common subterms are shared, as usual in implementations of lazy languages) and a minimal set of solutions, and it is fully deterministic if free variables do not occur.

Encapsulated search: Since functions in Curry have no side effects, the strategy to handle non-deterministic computations is not fixed in Curry (in contrast to Prolog which fixes a backtracking strategy). To provide flexible application-oriented search strategies and to avoid global backtracking like in Prolog which causes problems when integrated with I/O and concurrent computations, don't know non-deterministic computations can be encapsulated and controlled by the programmer [12]. For this purpose, a *search goal* is a lambda abstraction `\`x`->`c where c is the constraint to be solved and x is the search variable occurring in c for which solutions should be computed. Based on a single language primitive to control non-deterministic computation steps, various search strategies can be defined (see [12] for details). For instance, `findall` computes the list of all solutions for a search goal with a depth-first strategy, i.e., the expression "`findall \ps->prefix ps [1,2]`" reduces to the list `[[],[1],[1,2]]` (w.r.t. the program in Example 1).

An important point in the treatment of encapsulated search is that (i) the search has only local effects and (ii) non-deterministic steps are only performed if they are unavoidable. To satisfy requirement (i), "global" variables (i.e., variables that are visible outside the search goal) are never bound in local search steps. To satisfy requirement (ii), a possible non-deterministic step in a search goal is suspended if the search goal contains a global variable (since binding this variable outside the search goal might make this step deterministic) or another deterministic step is possible. This corresponds to the *stability* requirement in AKL [15]. In the context of this paper, the important point is that non-deterministic steps are not performed if the search goal has a reference to some global variable. Since we shall model the coordination of distributed activities by partially instantiated global variables, non-deterministic steps are automatically avoided if they refer to global communication channels.

Constraints: In functional logic programs, it is necessary to solve equations between expressions containing defined functions (see Example 1). In general, an *equation* or *equational constraint* e_1`=:=`e_2 is satisfied if both sides e_1 and e_2 are reducible to the same value (data term). As a consequence, if both sides are undefined (non-terminating), then the equality does not hold (*strict equality* [5]). Operationally, an equational constraint e_1`=:=`e_2 is solved by evaluating e_1 and e_2 to unifiable data terms where the lazy evaluation of the expressions is interleaved with the binding of variables to constructor terms. Thus, an equational constraint e_1`=:=`e_2 without occurrences of defined functions has the same meaning (unification) as in Prolog. The basic kernel of Curry only provides equational constraints. Since it is conceptually fairly easy to add other constraint structures, extensions of Curry can provide richer constraint systems to support constraint logic programming applications. In this paper, we add one special kind of constraint ("port constraint", see Section 3) to enable the efficient sending of messages from different clients to a server.

Concurrent computations: To support flexible computation rules and avoid an uncontrolled instantiation of free argument variables, Curry provides the *suspension of a function call* if a demanded argument is not instantiated. Such functions are called *rigid* in contrast to *flexible* functions which instantiate their arguments if it is necessary to proceed their evaluation. As a default in Curry (which can be easily changed), constraints (i.e., functions with result type `Constraint`) are flexible and non-constraint functions are rigid. Thus, purely logic programs (where predicates correspond to constraints) behave as in Prolog, and purely functional programs are executed as in lazy functional languages like Haskell.

To continue computations in the presence of suspended function calls, constraints can be combined with the *concurrent conjunction* operator `&`, i.e., c_1 `&` c_2 is a constraint which is evaluated by solving c_1 and c_2 concurrently. There is also a *sequential conjunction* operator `&>`, i.e., the expression c_1 `&>` c_2 is evaluated by first evaluating c_1 and then c_2.

A design principle of Curry is the clear separation of sequential and concurrent activities. Sequential computations, which form the basic units of a program, can be expressed as usual functional (logic) programs, and they are composed to concurrent computation units via concurrent conjunctions of constraints. This separation supports the use of efficient and optimal evaluation strategies for the sequential parts, where similar techniques for the concurrent parts are not available. This is in contrast to other, more fine-grained concurrent computation models like AKL [15], CCP [22], or Oz [25]. In this paper, we extend the basic concurrent computation model to support distributed applications where different (external) clients interact.

Monadic I/O: Since the communication with external programs require some knowledge about performing I/O declaratively, we assume familiarity with the monadic I/O concept of Haskell [20,27] which is also used in Curry. Due to lack of space, we cannot describe it here in detail but it is sufficient to remember that I/O actions are sequentially composed by the operators `>>=` and `>>`, `putStrLn` is an action that prints its string argument to the output stream, and `done` is the empty action. Since disjunctive I/O actions as a result of a program are not reasonable, all possible search must be encapsulated between I/O operations, otherwise the entire program suspends.

3 From Concurrent to Distributed Computations

This section motivates the primitives which we add to Curry to support distributed applications. Since these primitives should smoothly interact with the basic computation model, in particular encapsulated search and local concurrent computations, we introduce them as a specialization of the existing features for concurrent object-oriented programming.

It is well known from concurrent logic programming [24] that (concurrent) objects can be easily implemented as predicates processing a stream of incoming messages. The internal state of the object is a parameter which may change in recursive calls when a message is processed. For instance, a counter object which understands the messages `Set` v, `Inc`, and `Get` v can be implemented in Curry as follows (the predefined type `Int` denotes the type of all integer values and `success` denotes the always satisfiable constraint):

```
data CounterMessage = Set Int | Inc | Get Int

counter eval rigid
counter _ (Set v : ms) = counter v ms
counter n (Inc : ms) = counter (n+1) ms
counter n (Get v : ms) = v=:=n & counter n ms
counter _ [] = success
```

The *evaluation annotation* "`counter eval rigid`" marks `counter` as a rigid function, i.e., an expression "`counter` n s" can reduce only if s is a bound variable. The first argument of `counter` is the current value of the counter and the second argument is the stream of messages. Thus, the evaluation of the constraint "`counter 0 s`" creates a new counter object with initial value `0` where messages are sent by instantiating the variable `s`. The final rule terminates the object if the stream of incoming messages is finished. For instance, the constraint

```
let s free in counter 0 s & s=:=[Set 41, Inc, Get x]
```

is successfully evaluated by binding `x` to the value `42`. Although the stream variable `s` is instantiated at once to all messages in this simple example, it should be clear that messages can be individually sent by incrementally instantiating `s`.

If there is more than one process sending messages to the same counter object, it is necessary to merge the message streams from the different processes into a single message stream (otherwise, the processes must coordinate themselves for message sending). Since the processes work concurrently, the stream merger must be fair. A fair merger can be implemented in Curry as follows:

```
merge eval choice
merge (x:xs) ys = x : merge xs ys
merge xs (y:ys) = y : merge xs ys
merge [] ys = ys
merge xs [] = xs
```

The evaluation annotation choice has the effect that at most one rule is applied to a call to merge even if there is another applicable rule (where all alternatives are evaluated in a fair manner), i.e., this corresponds to a committed choice in concurrent logic languages. Although a committed choice restricts the declarative reading of programs and destroys the completeness results for the basic operational semantics [9], such or a similar construct is usually introduced to program reactive systems. Using the indeterministic merge function, we can create a counter that accepts messages from different clients:

```
counter 0 (merge s1 s2) & client1 s1 & client2 s2
```

If we want to access the counter object from n different clients, it is immediate to use $n-1$ mergers to combine the different message streams into a single one. It has been argued [16] that this causes a significant overhead due to the forwarding of incoming messages through the mergers. Moreover, this solution causes difficulties if the number of clients can change dynamically as in many distributed applications. Therefore, Janson et al. [16] proposed the use of ports to solve these problems. Ports provide a constant time message merging w.r.t. an arbitrary number of senders and a convenient way to dynamically extend the number of senders. Therefore, we also propose an extension of the base language by ports but embed this concept into concurrent functional logic programming (where Janson et al. proposed ports for the concurrent logic language AKL) and extend it to communication with external partners.

In principle, a *port* is a constraint between a multiset and a list that is satisfied if all elements in the multiset occur in the list and vice versa. A port is created by evaluating the constraint "openPort p s" where p and s are uninstantiated free variables. p and s will later be constrained to the multiset and list of elements, respectively. Since sending messages is done through p, p is often identified with the port and s is the stream of incoming messages. "Port a" denotes the type of a port to which messages of type a can be sent, i.e., openPort has the type definition

```
openPort :: Port a -> [a] -> Constraint
```

A message is sent to the port by evaluating the constraint "send m p" which constrains (in constant time) p and the corresponding stream s to hold the element m. From a logic programming point of view, the stream s has always an uninstantiated variable s_tail at the end and evaluating the send constraint means evaluating the constraint

```
let s_tail1 free in s_tail =:= (m : s_tail1)
```

Thus, the new message is appended at the end of the stream by instantiating the current open end of the stream. Since the instantiation is done by solving

a strict equation (compare Section 2), it is also evident that the message `m` is evaluated before sending it ("strict communication", like in Eden [3]). If the communication were lazy, the lazy evaluation of messages at the receiver's side would cause a communication overhead.
Using ports, we can rewrite our counter example with two clients as

```
openPort p s &> counter 0 s & client1 p & client2 p
```

Thus, the code for the object remains unchanged but we have to replace the instantiation of the streams in the clients by calls to the `send` constraint.

This approach to communication between different processes has remarkable consequences:

– It has a logical reading, i.e., communication is not done by predicates or functions with side effects (like, e.g., the socket library of Sicstus-Prolog) but can be described as instantiation of logical variables and constraint solving. Thus, the operational semantics of our communication primitives is a simple extension of the operational semantics of the base language.
– It interacts smoothly with the operational principles of the base language. For instance, local search and non-deterministic computations are only possible if the search goal contains no reference to global variables (compare Section 2). Thus, it is impossible to send messages to global ports inside local search computations or to split a server object into two non-deterministic computation threads. This is perfectly intended, since backtracking on network-oriented applications or copying server processes to interact with non-deterministic clients is difficult to implement.
– It provides an efficient implementation since message sending can be implemented without forwarding through several mergers and the senders have no reference to old messages, i.e., the multiset of the port must not be explicitly stored.
– Partially instantiated messages containing free variables (e.g., message "`Get x`") provide an elegant approach to return values to the sender without explicitly creating reply channels.
– The number of senders can be dynamically extended—every process which gets access to the port reference (the multiset variable) can send messages to the port. This property can be exploited in many distributed applications (see below).

Up to now, we can use ports only inside one program (similarly to [16]) but for many distributed applications (like Internet servers) it is necessary to communicate between different programs. Therefore, we introduce two operations to create and connect to *external ports*, i.e., ports that are accessible from outside. Since the connections of ports to the outside world changes the environment of the program, these operations are introduced as I/O actions (see Section 2).

The I/O action "`openNamedPort` n" creates a new external port with name n and returns the stream of incoming messages. If this action is executed on machine m (where m is a symbolic Internet name), the port (but not the stream of incoming messages) is now globally accessible as "n@m" by other applications.

On the client side, the I/O action "`connectPort` *pn*" returns the external port which has the symbolic name *pn* so that clients can send messages to this port. For instance, to create a globally accessible "counter server", we add the following definitions to our counter:

```
main = openNamedPort "counter" >>= counter_server
counter_server s | counter 0 s = done
```

If we execute `main` on the machine `medoc.cs.rwth.de`, we can implement a client by

```
client port_name msg = connectPort port_name >>= sendPort msg

sendPort msg p | send msg p = done
```

and increment the global counter by evaluating

```
client "counter@medoc.cs.rwth.de" Inc
```

Before we present some more interesting examples, we introduce a final primitive which has no declarative meaning but is useful in real distributed applications. Since the communication over networks is unsafe and a selected server could be down or may not respond in a given period of time, one want to take another action (for instance, choosing a different server or inform the user) if this happens. Therefore, we introduce a temporal constraint "`after` *t*" which is satisfied *t* milliseconds after the constraint has been checked for the first time, i.e., "`after 0`" is immediately satisfied. Typically, this temporal constraint is used as an alternative in a committed choice like in

```
getAnswer eval choice
getAnswer (msg:_) = msg
getAnswer _ | after 5000 = <take an alternative action>
```

For instance, if `getAnswer` is called with a stream of a port as an argument, it returns the first message if it is received within five seconds, otherwise an alternative action is taken.

The following type definitions summarizes the proposed new primitives to support the development of distributed applications:

```
-- open an internal port for messages of type "a":
openPort :: Port a -> [a] -> Constraint

send :: a -> Port a -> Constraint -- send message to port

-- open a new external port, return stream of messages:
openNamedPort :: String -> IO [a]

-- connect to external port, return port for sending messages:
connectPort :: String -> IO (Port a)

after :: Int -> Constraint -- timeout
```

4 Examples

In this section, we demonstrate the use of the primitives for distributed applications introduced in the previous section. In order to avoid presenting all the tedious details of such applications, we have simplified the examples so that we concentrate on the communication structures.

4.1 A Name Server

The first example represents a class of client/server applications where the server holds some database which is requested by the clients. For the sake of simplicity, we consider a simple name server which stores an assignment from symbolic names to numbers. It understands the messages "`PutName` *n i*" to store the name *n* with number *i* and "`GetName` *n i*" to retrieve the number *i* associated to the name *n*. The name server is implemented as a function which has the assignment from names to numbers as the first argument (function `n2i` below) and the incoming messages as the second argument (initially, `0` is assigned to all names by the lambda abstraction `\_->0`):

```
nameserver = openNamedPort "nameserver" >>= ns_loop \_->0

ns_loop n2i (GetName n i : ms) | i=:=(n2i n) = ns_loop n2i ms
ns_loop n2i (PutName n i : ms) = ns_loop new_n2i ms
where new_n2i m = if m==n then i else n2i m
```

In the first rule of `ns_loop`, the (usually uninstantiated) variable `i` is instantiated with the number assigned to the name `n` by solving the equational constraint in the condition. In the second rule, a modified assignment map `new_n2i` is passed to the recursive call. If we evaluate `nameserver` on the machine `medoc.cs.rwth.de`, then we can add the assignment of the name `talk` to the number `42` by evaluating

```
client "nameserver@medoc.cs.rwth.de" (PutName "talk" 42)
```

on some machine connected to the Internet (where `client` was defined in Section 3). After this assignment, the evaluation of

```
client "nameserver@medoc.cs.rwth.de" (GetName "talk" x)
```

binds the free variable `x` to the value `42`.

Note that the sending of messages containing free variables is an elegant way to return values to the sender. Here we exploit the fact that the base language is an integrated functional logic language which can deal with logical variables. Functional languages extended for distributed programming like Eden [3], Erlang [2], or Goffin [4] require the explicit creation or sending of reply channels.

An extension of our name server should demonstrate the advantages of using logical variables in messages. Consider a hierarchical name server organization: if the local name server has no entry for the requested name (i.e., the assigned number is the initial value `0`), it forwards this request to another name server. This can be easily expressed by changing the first rule of `ns_loop` to

```
ns_loop n2i (GetName n i : ms)
| if (n2i n)==0 then send (GetName n i) master else i=:=(n2i n)
= ns_loop n2i ms
```

It is assumed that `master` is the port of the other name server to which the request is forwarded. Note that the local name server can immediately proceed its service after forwarding the request to the master server and need not to wait for the answer from the master since the master becomes responsible for binding the free variable in the `GetName` message.

If the requested name server is down so that no answer is returned, one would like to inform the user about this fact instead of an infinite waiting. This can be easily implemented with a temporal constraint by the following function:

```
showAnswer eval choice
showAnswer ans | ans==ans = show ans
showAnswer _  | after 10000 = "No answer from name server"
```

"$t_1 == t_2$" denotes strict equality on ground data terms like in Haskell, i.e., if t_1 or t_2 reduces to a data term containing an uninstantiated variable, the evaluation of this equality is suspended until the variable has been bound to some ground data term. Thus, "`showAnswer` x" yields a string representation of the value of x if it evaluates to a ground data term or it yields the string `"No answer from name server"` if x has not been bound to a ground term within ten seconds. Thus, the evaluation of

```
client "nameserver@medoc.cs.rwth.de" (GetName "talk" x)
>> putStrLn (showAnswer x)
```

prints the value assigned to `talk` or the required timeout message.

4.2 Talk

The next example shows a distributed application between two partners where both of them act as a server as well as a client. The application is a simplification of the well known Unix "talk" program. Here we consider only the main talk activity (and not the calling of the partner via the talk daemon) where each partner program must do the following (we assume that each partner has an external *talk port* with symbolic name `talk` to receive the messages from the partner):

– If the user inputs a line on the keyboard (which is transmitted through the port with symbolic name `stdin`), this line is sent to the talk port of the partner.
– If the program receives a line from the partner through its own talk port, this line (preceded by '`*`') is shown at the screen by the I/O action `putStrLn`.

Since the sequence of both events is not known in advance, the standard input port as well as the talk port must be examined in parallel. For this purpose, we use a committed choice. Thus, the talk program consists of a loop function `tloop`

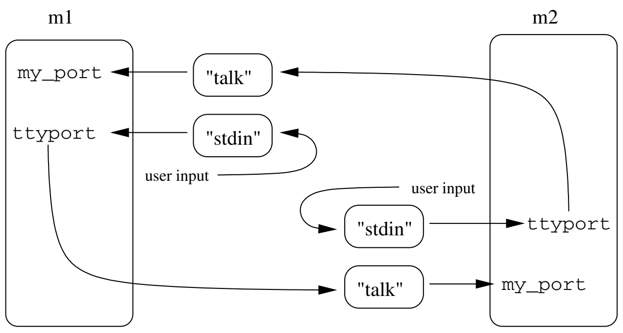

Fig. 1. Communication structure of the talk program

which has three arguments: the talk port of the partner, the stream connected to the own standard input, and the stream connected to the own talk port:

```
tloop eval choice
tloop your tty (m:ms) = putStrLn ('*':m) >> tloop your tty ms
tloop your (m:ms) my = sendPort m your >> tloop your ms my
```

The `tloop` is activated by the following main program:[2]

```
talk your_portname = do my_port <- openNamedPort "talk"
tty_port <- openNamedPort "stdin"
your_port <- connectPort your_portname
tloop your_port tty_port my_port
```

If a user on machine `m1` wants to talk with the user on machine `m2`, they must evaluate

on machine `m1`: `talk "talk@m2"`
on machine `m2`: `talk "talk@m1"`

The communication structure created by these calls is shown in Fig. 1.

4.3 A Computation Server

Since our communication through ports is strict, i.e., messages are evaluated before sending them (cf. Section 3), there is no direct way to distribute computational work like remote procedure calls (RPCs) where procedures are evaluated at some other node in the network. Although port communication corresponds to message passing, we can easily implement RPCs using the higher-order features of the base language. For instance, a computation server, i.e., a process running on some node in the network offering to execute some work by evaluating functions, can be implemented as a function accepting messages containing triples `(`f`,`x`,`y`)` where f is a function to be applied to the actual argument x

[2] Here we make use of Haskell's `do` notation [20] where "`do` p_1`<-`e_1`;`$\ldots$`,`p_n`<-`e_n`;`e" is syntactic sugar for "e_1 `>>= \`p_1`->`$\ldots e_n$ `>>= \`p_n`->`e".

and y is a free variable which is instantiated with the result of $f\,x$. Thus, the entire computation server can be implemented as follows:

```
start_compserver = openNamedPort "compserver" >>= compserver
compserver ((f,x,y) : ms) | y=:=(f x) = compserver ms
```

If `prime` is a function to compute the n-th prime number, we can use this computation server to compute prime numbers, e.g., the execution of

```
client "compserver@cs" (prime,1000,p)
```

binds the free variable `p` to the 1000th prime number where the computation is performed on the node `cs` where the server has been started. This remarkable simple implementation needs some comments.

1. In Section 2, we introduced the constraint `=:=` as equality on data terms and, thus, it might be unclear how we can send functional objects in messages. For this purpose, we consider partially applied functions, i.e., functions where the number of actual arguments is less than their arity, as data terms since they are not evaluable. This is conform with standard methods to add higher-order features to logic programming [28] and theoretically justified for lazy functional logic languages in [6]. As a consequence, an equation like "`x=:=prime`" is solved by binding the variable `x` to the function name `prime`. Since partially applied function calls are considered as data terms, the code implementing the function is not immediately sent in the above message but it will be transferred from the client to the server when the server evaluates it (dynamic code fetching).
2. The RPC is asynchronously performed since the client sends its request without explicitly waiting for the answer. The client can proceed with other computations as long as it does not need the result of this call which is passed back through the third argument of the message. Thus, the free result variable is similar to a "promise" which has been proposed by Liskov and Shrira [17] to overcome the disadvantages of synchronous RPCs. A promise is a special place holder for a future return value from an RPC. Since we can use the logic part of the base language for this purpose, no linguistic extension is necessary to implement asynchronous RPCs.
3. The attentive reader might raise the question what happens if the execution of the transmitted function causes a non-deterministic computation step. Does the server split into two disjunctive branches? This does not happen since, as mentioned at the end of Section 2, non-deterministic steps between I/O actions are suspended. One method to avoid this suspension is to return only the first solution to the sender. This can be done by encapsulating the search, i.e., we could replace the constraint "`y=:=(f x)`" by the expression

   ```
   y =:= head (findall \z -> z=:=(f x))
   ```

A disadvantage of the above computation server is the fact that the complete server is blocked if the evaluation of a single RPC is suspended or takes a long time. Fortunately, it is very simple to provide a concurrent version of this server using the concurrency features of the base language. For this purpose, we turn the server function into a constraint and evaluate the RPC in parallel to the main server process:

```
start_compserver = openNamedPort "compserver" >>= serve
where serve ms | compserver ms = done

compserver eval rigid[3]
compserver ((f,x,y) : ms)  =  y=:=(f x) & compserver ms
```

4.4 Encrypting Messages

To support more security during message sending, messages should be encrypted before sending. For this purpose, public key methods are often used. The idea of public key methods is to encode a message with a key before sending and to decode the message with another key after receiving. Both keys must be chosen in a way so that decoding the encoded message gives the original message back. Since the coding algorithm as well as one key are publicly known, it is essential for the security of the method to choose keys that are large enough.

In the following, we use a similar idea but functions instead of keys, i.e., the encoding algorithm as well as the key is put into a single function. Thus, one has to choose a public encrypt function e and a private decrypt function d so that $d(e(m)) = m$ for all messages m (the additional property $e(d(m)) = m$ would be necessary for authentication).

As a simple example, we show a server which processes requests and returns the answers encrypted. The public encrypt function is sent together with the message. This has the advantage that for each message and client, another encryption can be used. Since there are a huge number of encrypt/decrypt function pairs, the functions could be relatively simple without sacrificing security. Similarly to the computation server, this server receives triples (e,rq,rs) where e is the public encrypt function, rq is the request to the server and rs will be instantiated to the encrypted result (the unspecified function `computeanswer` determines the main activity of the server):

```
start_crypticserver = openNamedPort "cryptserver" >>= cserver

cserver ((encode,rq,rs) : ms) | rs =:= encode(computeanswer rq)
   = cserver ms
```

For strings, i.e., lists of characters, the pair `rev`/`rev` (list reversing) is a simple encrypt/decrypt pair. Thus, we can send a request to the server and decode the answer by

```
client "cryptserver@cs" (rev,"Question...",y) >> show (rev y)
```

Although this example is simplified, it should be obvious that further features like authentication can be easily added.

[3] The rigid annotation is necessary since constraints are flexible by default in Curry.

5 Implementation

The full implementation of the presented concepts is ongoing. We have tested the examples in this paper with a prototypical implementation of Curry based on Sicstus-Prolog. In this implementation, we used the socket library of Sicstus-Prolog to implement the port communication via sockets. Free variables sent in messages are implemented by dynamic reply channels on which the receiver sends the value back if the variable is instantiated.

Currently, we are working on a more efficient implementation based on the compiler from Curry into Java described in [11]. In this implementation, we use the distribution facilities of Java to implement our communication model. In particular, we use Java's RMI model to implement ports. Sending a message amounts to binding a free variable (the stream connected to the port) by a method call on the remote machine. Free variables sent in messages reside in the sender's computation space and if the receiver binds this variable, he calls a remote method on the sender's machine to bind this variable. The implementation of functional objects sent in messages is more advanced. It could be implemented by sending a reference to the code that implements this function. If the function is applied and evaluated by the receiver, the function code is dynamically loaded from the sender to the receiver (dynamic code fetching).

6 Related Work

Since features for concurrent and distributed programming become important for many applications, there are a various approaches to extend functional or logic programming languages with such features. In the following, we relate our proposal to some of the existing ones.

Initiated by Japan's fifth generation project, various approaches to add concurrency features to logic programming [23] have been proposed culminating in Saraswat's framework for concurrent constraint programming [22]. Usually, these approaches consider only concurrency inside an application but provide no features for connecting different programs to a distributed system. The concurrent logic language AKL [15] also supports only concurrency inside a program but proposed ports [16] for the efficient communication between objects. Ports have been also adapted to Oz [25] where it has been also embedded into a framework for distributing the computational activities over a network [26]. In contrast to our approach, ports are not a primitive constraint but are implemented by the stateful features of Oz. All these languages are strict (and untyped) while our proposal combines optimal lazy reduction for the sequential computation parts with strict communication between the distributed and concurrent entities.

Concurrent Haskell [21] extends the lazy functional language Haskell by methods to start processes inside an application and synchronize them with mutable variables, but facilities for distribution are not provided. Closest to our approach w.r.t. the communication features are Erlang [2] and an extension of Goffin [4]. Erlang is a concurrent functional language developed for telecommunication applications. Processes in Erlang can communicate over a network via symbolic names which provides for communication between different applications. In contrast to our proposal, Erlang is a strict and untyped language and provides no features for logic programming. Thus, partial messages can not be sent so that explicit reply channels (process identifiers) must be included in messages where answers should be sent back. The extension of Goffin described in [4] extends a lazy typed concurrent functional language by a port model for internal and external communication. Although it uses logical variables for synchronization, it does not provide typical logic programming features like search for solutions. Differently to our proposal for communication, partial messages including logical variables are not supported, the creation of connections to external ports is not integrated in the I/O monad (and, hence, I/O operations like reading/writing files can not be used in a distributed program) and, once a port is made public on the network, every node can not only send messages to this port but can also read all messages incoming at this port. The latter property may cause security problems for many distributed applications. This is avoided in our proposal by allowing only one server process to read the incoming messages at an external port.

7 Conclusions

We have proposed an extension of the concurrent functional logic language Curry that supports a simple implementation of distributed applications. This extension is based on communication via ports. The important point is that the meaning of port communication can be described in terms of computation with constraints. This has the consequence that (i) the communication mechanism interacts smoothly with the existing language features for search and concurrency so that all these features can be used to program server applications, and (ii) existing programs can be fairly easy integrated into a distributed environment. Moreover, the use of logical variables in partially instantiated messages is quite useful to avoid complicated communication structures with reply channels. Nevertheless, external communication ports can be given a symbolic name so that they can be passed in messages as in the π-calculus [18]. We have demonstrated the appropriateness and feasibility of our language extensions by implementing several distributed applications. As far as we know, this is the first approach which combines functional logic programming based on a lazy (optimal) evaluation strategy with features for concurrent and distributed programming.

For future work, we will investigate the application of program analysis techniques to ensure the safe execution of distributed applications. For instance, deadlock exclusion can be approximated by checking groundness of relevant variables [8] or the non-conflicting use of free variables transmitted in messages could be ensured by proving that they are instantiated by at most one receiver.

Acknowledgements.

The author is grateful to Frank Steiner and Philipp Niederau for many discussions and comments on this paper and for providing the implementation of the talk program.

References

1. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. In *Proc. 21st ACM Symp. on Principles of Programming Languages*, pp. 268–279, 1994.
2. J. Armstrong, M. Williams, C. Wikstrom, and R. Virding. *Concurrent Programming in Erlang.* Prentice Hall, 1996.
3. S. Breitinger, R. Loogen, and Y. Ortega-Mallen. Concurrency in Functional and Logic Programming. In *Fuji International Workshop on Functional and Logic Programming.* World Scientific Publ., 1995.
4. M.M.T. Chakravarty, Y. Guo, and M. Köhler. Distributed Haskell: Goffin on the Internet. In *Proc. of the Third Fuji International Symposium on Functional and Logic Programming.* World Scientific, 1998.
5. E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel LEAF: A Logic plus Functional Language. *Journal of Computer and System Sciences*, Vol. 42, No. 2, pp. 139–185, 1991.
6. J.C. González-Moreno, M.T. Hortalá-González, and M. Rodríguez-Artalejo. A Higher Order Rewriting Logic for Functional Logic Programming. In *Proc. Int. Conference on Logic Programming (ICLP'97)*, pp. 153–167. MIT Press, 1997.
7. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, Vol. 19&20, pp. 583–628, 1994.
8. M. Hanus. Analysis of Residuating Logic Programs. *Journal of Logic Programming*, Vol. 24, No. 3, pp. 161–199, 1995.
9. M. Hanus. A Unified Computation Model for Functional and Logic Programming. In *Proc. 24th ACM Symp. Principles of Programming Languages*, pp. 80–93, 1997.
10. M. Hanus and P. Réty. Demand-driven Search in Functional Logic Programs. Research Report RR-LIFO-98-08, Univ. Orléans, 1998.
11. M. Hanus and R. Sadre. An Abstract Machine for Curry and its Concurrent Implementation in Java. *Journal of Functional and Logic Programming*, 1999(6).
12. M. Hanus and F. Steiner. Controlling Search in Declarative Programs. In *Principles of Declarative Programming*, pp. 374–390. Springer LNCS 1490, 1998.
13. M. Hanus (ed.). Curry: An Integrated Functional Logic Language (Vers. 0.5). Available at `http://www-i2.informatik.rwth-aachen.de/~hanus/curry`, 1999.
14. J. Hughes. Why Functional Programming Matters. In D.A. Turner, editor, *Research Topics in Functional Programming*, pp. 17–42. Addison Wesley, 1990.
15. S. Janson and S. Haridi. Programming Paradigms of the Andorra Kernel Language. In *Proc. 1991 Int. Logic Programming Symposium*, pp. 167–183. MIT Press, 1991.
16. S. Janson, J. Montelius, and S. Haridi. Ports for Objects in Concurrent Logic Programs. In *Research Directions in Concurrent Object-Oriented Programming.* MIT Press, 1993.
17. B. Liskov and L. Shrira. Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. In *Proc. SIGPLAN'88 Conference on Programming Language Design and Implementation*, pp. 260–267, 1988.

18. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, Parts I and II. *Information and Computation*, Vol. 100, pp. 1–77, 1992.
19. J.J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic Programming with Functions and Predicates: The Language BABEL. *Journal of Logic Programming*, Vol. 12, pp. 191–223, 1992.
20. J. Peterson et al. Haskell: A Non-strict, Purely Functional Language (Version 1.4). Technical Report, Yale University, 1997.
21. S.L. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proc. 23rd ACM Symp. on Principles of Programming Languages*, pp. 295–308, 1996.
22. V.A. Saraswat. *Concurrent Constraint Programming.* MIT Press, 1993.
23. E. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, Vol. 21, No. 3, pp. 412–510, 1989.
24. E. Shapiro and A. Takeuchi. Object Oriented Programming in Concurrent Prolog. In *Concurrent Prolog: Collected Papers*, volume 2, pp. 251–273. MIT Press, 1987.
25. G. Smolka. The Oz Programming Model. In *Computer Science Today: Recent Trends and Developments*, pp. 324–343. Springer LNCS 1000, 1995.
26. P. Van Roy, S. Haridi, P. Brand, G. Smolka, M. Mehl, and R. Scheidhauer. Mobile Objects in Distributed Oz. *ACM TOPLAS*, 19(5), pp. 804–851, 1997.
27. P. Wadler. How to Declare an Imperative. In *Proc. of the 1995 International Logic Programming Symposium*, pp. 18–32. MIT Press, 1995.
28. D.H.D. Warren. Higher-order extensions to PROLOG: are they needed? In *Machine Intelligence 10*, pp. 441–454, 1982.