LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TM-386

COMMUNICATION EFFECTS FOR MESSAGE-BASED CONCURRENCY

Pierre Jouvelot
David K. Gifford

February 1989

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

MIT/LCS/TM-386

Communication Effects for Message-Based Concurrency

Pierre Jouvelot
David K. Gifford

February 1989

Abstract

We describe a new framework for explicit concurrency that uses an *effect system* to describe the communication behavior of expressions in a typed polymorphic programming language. Concurrency occurs between *processes* connected by *channels* on which *messages* are transmitted. Communication operations are characterized by two *communication effect* constructors, `out` and `in`, depending on whether a message has been sent or received. Synchronization is only allowed by message-passing along shared channels; communication via mutation of global variables is statically prohibited by our communication effect system, thus restricting the amount of non-determinacy in user programs. Unobservable communication effects can be masked by the effect system.

We show that this system is powerful enough to express many other parallel paradigms, like systolic arrays or pipes, in a typed framework. The programmer can thus express concurrency in a rather flexible way while preserving the correctness of implicit detection of parallelism and optimization by the compiler. This new concurrency framework has been implemented in the *FX-87* programming language.

Categories and Subject Descriptions: D.1.3 [**Programming Techniques**] – Concurrent Programming: *Effect systems*; D.1.m [**Programming Techniques**] – Miscellaneous: *First-class channels*; D.3.1 [**Programming Languages**] – Formal Definitions and Theory; D.3.3 [**Programming Languages**]–Language Constructs: *Control structures, effect systems*; D.3.4 [**Programming Languages**] – Processors: *Compilers, optimization.*

General Terms: Languages, Theory, Verification.

Additional Key Words and Phrases: effect systems, type systems, communication effects, effect masking, explicit parallelism, *FX-87*.

1 Introduction

Although quite a fair amount of parallelism can be automatically extracted from sequential programs by smart compilers, there are some problems for which an explicitly parallel algorithm is both more natural to express and easier to efficiently implement. There are numerous parallel paradigms that can be added to an otherwise sequential language to fulfill that goal like message passing, systolic programming, data parallelism or fork/join. We propose in this paper to show how a message-based communication framework based on *communication effects* can be used to explicitly express parallelism in a typed polymorphic language which uses an effect system [LG88].

An effect system is based on a *kinded* type system for the second-order lambda calculus [M79]. Kinds are the types of descriptions. Our type and effect system has three base kinds: *types*, which describe the value that an expression may return; *effects*, which describe the side-effects that an expression may have; and *regions*, which are used to describe where side-effects may occur. An expression that does not have an effect is said to be `pure`. Expressions that are `pure` are referentially transparent.

Types, effects and regions are closely interrelated. In particular, a function type incorporates a *latent effect*, which describes the side-effects that the function may have when it is applied, and a reference type incorporates a region, which describes where the reference is allocated. The kind system is used to verify the well-formedness of descriptions; the type and effect system is used to verify the well-formedness of expressions.

We can use an effect system for communication analysis by introducing two types of communication effects, `out` and `in`, that describe the communication properties of expressions. An expression that does not have an `out` effect never sends messages to an outside process. An expression that does not have an `in` effect never receives messages from an outside process. Unobservable communication effects can be *masked* by the effect system. Our masking rule applies to expressions that don't communicate externally, even though they might use communication internally.

Communication effects are useful to the programmer, the language designer and the compiler writer:

- Communication effects let the programmer specify, in machine-verifiable form, the expected run-time communication behavior of a given pro-

gram, thus increasing documentation, modularity and maintainability of programs. Communication effects also provide a programmer with a new framework in which to reason about languages with explicit parallelism. Moreover, when unobservable communication effects are masked, a programmer knows that an expression will not interact with other processes.

- Communication effects let the language designer limit the use of concurrency to simplify the semantics of the language. For instance, by saying that concurrent processes are only allowed to communicate by messages and not via global mutable variables, the amount of non-deterministic behavior of parallel programs is controlled. In consequence, programs that do not use explicit concurrency constructs are sure to behave in a deterministic sequential way.

- Communication effects let the compiler writer perform any usual optimization like common subexpression elimination or memoization in the presence of explicit parallelism.

Communication effects can be introduced in any programming language which uses an effect system [LG88]. For the sake of homogeneity in presentation, we will use the *FX-87* language throughout this paper [GJLS87].

In the remainder of this paper, we sketch the main characteristics of the effect system of the *FX-87* programming language (Section 2), introduce our concurrency framework by defining channels, processes and communication effects (Section 3), show how other paradigms for parallelism can be adapted to ours (Section 4), describe the innards of our experimental parallel implementation (Section 5), stress the major benefits of our approach (Section 6), survey related work (Section 7) and summarize our results (Section 8).

2 The *FX-87* Programming Language

FX-87 [GJLS87] is a programming language with a second-order polymorphic type and effect system [LG88] and in which higher-order functionals are first-class values. The concrete syntax and standard operations of *FX-87* are reminiscent of the ones of Scheme [R86]. The static analysis of an *FX-87* program consists in the determination of a set of *descriptions*:

- Every ordinary variable is described by a *type* and a *region*. The region of a variable, optionally provided by a programmer, represents an abstract set of memory locations into which the variable is located.

- Every *FX-87* expression is described at compile-time by the *type* of its result and the *effect* of its evaluation. The effect of an expression is a static description of the observable store (as defined by regions) operations like `read`, `write` or `alloc` (for allocation) that may be performed during its evaluation.

- Every *FX-87* procedure is characterized by a type that includes a *latent effect*, the types of the arguments and the type of the result. A latent effect describes the effect that may occur when the procedure is called.

FX-87 gives the programmer the ability to manipulate all these descriptions (types, regions, effects), and the flexibility to build description functions. An example of such a description function is `listof` which maps a type *texp* and a region *rexp* to the type (`listof` *texp rexp*) of lists, located in *rexp*, of values of type *texp*. Ordinary expressions can be abstracted over any kind of description.

As an example, consider the definition of a polymorphic `map` procedure that applies a function `f` to a list `l` of values. The function `map` can be defined as follows in *FX-87*:

```
(define map
  (plambda ((r region))
    (plambda ((e effect) (t1 type) (t2 type))
      (lambda ((f (subr e (t1) t2)) (l (listof t1 r)))
        (the (maxeff e (alloc r) (read r)) (listof t2 r)
             (if (null? l)
                 ()
                 ((proj cons r) (f (car l)) (map f (cdr l))))))))))
```

Note that `map` is abstracted (by `plambda` forms) over a region `r`, and an effect `e` and two types `t1` and `t2`; thus, `map` must be projected onto a region, an effect, and two types before it can be used. For example:

```
((proj (proj map @=) pure int bool) odd? (list 1 2 3))
```

yields a list allocated in the immutable region (noted `@=`) and whose value is the list `(#t #f #t)` where `#t` and `#f` are the *FX-87* literals for the boolean values true and false. *FX-87* uses *implicit projection* to allow users to omit some of these projections; we used this in the body of `map`. The fact that the function `map` is polymorphic over a region, an effect and two types can be seen in its `poly` type:

```
map : (poly ((r region))
        (poly ((e effect) (t1 type) (t2 type))
          (subr (maxeff e (alloc r) (read r))
                ((subr e (t1) t2) (listof t1 r))
                (listof t2 r))))
```

Focusing on `map`'s effect parameter `e` for a moment, `map` must be provided with the effect of the procedure `f` it is going to map over a list. Note that the latent effect `e` of the procedure `f` passed to `map` will also be part of the latent effect of `map` (given as the first argument of the declarative `the` form, the second argument being the return type); (`maxeff` $eexp_1$ $eexp_2$) denotes the cumulative effect of the effects $eexp_1$ and $eexp_2$. Thus, if `map` is given a pure procedure to map over an immutable list (i.e., `r` is `@=`), `map` will be pure. However, if `map` is given a procedure with the latent effect `(write @sample)`, then `map` will have `(write @sample)` in its latent effect. Note that in any case, `map`'s latent effect will include both the effect `(alloc r)` for allocating the result list in the region `r` and `(read r)` for reading values inside the argument list `l`.

When regions or effects are abstract (as in the case of `r` and `e` in the body of `map`), the effect system ensures that they do not alias. Namely, region and effect variables are mutually independent, so that polymorphic expressions can be efficiently compiled on parallel architectures. This property is enforced at compile-time by the effect system of *FX-87* whenever an expression is projected on regions and effects.

The *FX-87* effect system masks *unobservable* effects. An effect of an expression is unobservable if the effect is performed on a region that is not accessible outside of the scope of the expression. This can be detected statically by the *FX-87* system; for instance, a `write` effect on *rexp* can be masked if there are no free variables that have *rexp* in their type. Thus it is possible for an expression to perform arbitrary side-effects in the heap and still have effect `pure`. *Effect masking* is useful because it allows mutable objects

to be used for efficiency where necessary without forcing the serialization of expressions.

3 Message-Based Concurrency

The use of an effect system allows a compile-time detection of parallelism in a language with first-class functions. Expressions that don't have interfering effects can be scheduled in parallel. We show how an effect system can be extended with *communication effects* in order to deal with explicit parallelism. In this paper, concurrency occurs between *processes* connected by *channels* on which *messages* are transmitted.

Before going into a precise definition of the few new concepts that are needed to deal with explicit concurrency, we first begin by a simple example, a logical system, which gives a flavor of the constructs we are about to introduce:

```
(pdefine wire (channelof bool @wire))

(define (and-gate (x wire) (y wire) (xy wire))
    ((to xy) (and? (from x) (from y))))
(define (not-gate (x wire) (-x wire))
    ((to -x) (not? (from x))))

(define (nand-gate (x wire) (y wire) (-xy wire))
    (let ((xy (proj (proj channel @wire) bool)))
        (cobegin (and-gate x y xy)
                 (not-gate xy -xy))))
```

Logical gates like `and-gate` are implemented as functions that receive as arguments the wires (i.e. channels) used to connect them to the outside world. On these wires, boolean values are transmitted by the primitives `to`, for output, and `from`, for input; each of these constructs has a latent communication effect, respectively `(out @wire)` and `(in @wire)`. The `nand-gate` function shows how new channels can be created with the `channel` polymorphic function and how parallel function invocations (i.e. processes) can be created by the `cobegin` construct.

3.1 Channels

A *channel* is a typed asynchronous unbounded error-free FIFO communication medium. It is introduced in *FX-87* by the type constructor `channelof` of kind:

```
channelof ::  (dfunc (type region) type)
```

For any type *texp* and region *rexp*, (`channelof` *texp rexp*) is the type of channels located in region *rexp* on which values of type *texp* are transmitted. For instance, a value of type (`channelof int @request`) denotes a channel on which values of type `int` can be transmitted. Any *FX-87* type can be used within the `channelof` constructor (even channels).

Whenever some expression performs a communication over a channel (in a way which is described in the next subsection), then a *communication side-effect* has to be reported. If not reported, an only-communicating expression could be dead-code eliminated, which could create a starvation situation for the processes with which it was supposed to communicate. These effects are defined by the following effect constructors:

```
in  ::  (dfunc (region) effect)
out ::  (dfunc (region) effect)
```

For a region *rexp*, an (`in` *rexp*) (resp. (`out` *rexp*)) effect will be charged to an expression that gets (resp. puts) a value from (resp. into) a channel located in the region *rexp*. Note that the usual `read` and `write` effects cannot be used here since the communication side-effects (called hereinafter *communication effects*) are not subjected to the same effect-masking and interfering rules as the one used for store side-effects. Communication effects have the following properties:

- An `out` effect is a static error on channels that are located in the immutable region.
- A communication effect of an expression `E` can be masked if the region it is referring to doesn't appear free in the type of any free variable of `E`.
- An `in` (resp. `out`) effect is only a subeffect of another `in` (resp. `out`) effect; the subeffect rule is that its region has to be a (possibly not proper) subregion of the other one.

- The communication effects distribute over `maxeff` and `runion`[1] in the same way the other effects do, e.g. `(in (runion` $rexp_1$ $rexp_2$`))` = `(maxeff (in` $rexp_1$`) (in` $rexp_2$`))`.

- The interference relation between effects can be extended in the following way to deal with communication effects: *Communication effects interfere only with other communication effects. The effects* `(in` $rexp_1$`)` *and* `(out` $rexp_1$`)` *both interfere with* `(in` $rexp_2$`)` *and* `(out` $rexp_2$`)` *iff* $rexp_1$ *and* $rexp_2$ *aliase.*

3.2 Communication

Three functions are provided to manipulate channels: `channel` creates a new channel, `to` puts a value into a given channel and `from` gets a value from a given channel.

The creation of a channel is performed by the function `channel`:

```
channel : (poly ((r region))
            (poly ((t type))
                  (vsubr (alloc r) t (channelof t r))))
```

The `channel` function accepts an optional argument (it is a dynamic error if more than one argument is provided) and returns a new channel possibly initialized with the optional value. The `alloc` effect reports the creation of a new data structure and is moreover necessary to avoid common subexpression elimination on newly created (and mutable) channel values.

The `to` function is used to put a value onto a given channel:

```
to : (poly ((r region))
       (poly ((t type))
             (subr pure
                   ((channelof t r))
                   (subr (out r) (t) unit))))
```

Performing a `to` operation on an immutable channel is a static error. A `to` operation is never blocking since channels are unbounded. The message is

[1] `(runion` $rexp_1$ $rexp_2$`)` denotes the region that includes $rexp_1$ and $rexp_2$

queued in a FIFO manner onto the given channel except if it is the first message sent on an initialized channel; in that case, the initial value is discarded and replaced by the incoming message.

The `from` function is used to get a value from a given channel:

```
from : (poly ((r region))
         (poly ((t type))
               (subr (in r) ((channelof t r)) t)))
```

If the channel is in its initial state, then its initial value is returned and the state of the channel remains unaltered. If the channel has at least one message available, the first one is returned by the `from` function and the channel keeps only the rest of the queue. If the channel is empty, then the process that performs the `from` function is blocked and will be awaken as soon as a value is input into the channel.

3.3 Processes

A *process* is a single thread of control[2]. They are created by the `cobegin` special form:

(`cobegin` exp_1 exp_2)

A `cobegin` construct creates two new *child* processes that will concurrently (in a fair way [F86]) evaluate their unevaluated arguments. The *parent* process is blocked up to the completion of its two children. The value returned (if it exists) is an immutable pair made with the result values of the child processes.

The type of a `cobegin` expression is `void` if the type of one of its children has type `void`, else (`pairof` $texp_1$ $texp_2$ `@=`) where $texp_i$ is the type of exp_i. In *FX-87*, `void` is the type of certain non-terminating expressions. The effect of a `cobegin` expression is the `maxeff` of the effects of its children.

The two child processes are not allowed to communicate except via shared channels. This property can be enforced at compile-time by the *FX-87* effect system which is extended with the following rule:

> In (`cobegin` exp_1 exp_2), the effects of exp_1 and exp_2 must not interfere, except via communication effects.

[2]We don't take into account here the presence of other processes implicitly introduced by the *FX-87* compiler on the basis of memory effect constraints.

4 Examples of Other Parallel Paradigms

We will give here four examples of programs written with the previous constructs: Fibonacci, systolic sorting, pipeline and fair merge. They intend to show how different styles of parallel programming can be used within our framework. All these examples have been run on the experimental implementation described in the next section.

4.1 Fibonacci

Our first example is an explicitly parallel version of the Fibonacci function. Note that this code could very well be what an *FX-87* compiler would generate after effect analysis of a sequential version of the Fibonacci function:

```
(define (fibo (n int))
  (the pure int
       (if (<= n 1)
           n
           (let ((fn-1/fn-2 (cobegin (fibo (- n 1))
                                     (fibo (- n 2)))))
             (+ (car fn-1/fn-2) (cdr fn-1/fn-2))))))
```

It is also possible to use the continuation-passing style popularized in [S78] to code a parallel version of Fibonacci. The idea is to pass as continuation a function that will put the return value in the appropriate channel:

```
(define fibo-c
  (plambda ((e effect))
    (lambda ((cont (subr e (int) unit)) (n int))
        (the e unit
             (if (<= n 1)
                 (cont n)
                 (let ((local ((proj (proj channel @local) int))))
                     (cont
                      (car
                       (cobegin
                        (+ (from local) (from local))
                        (cobegin
```

```
                    (fibo-c (to local) (- n 1))
                    (fibo-c (to local) (- n 2)))))))))))
```

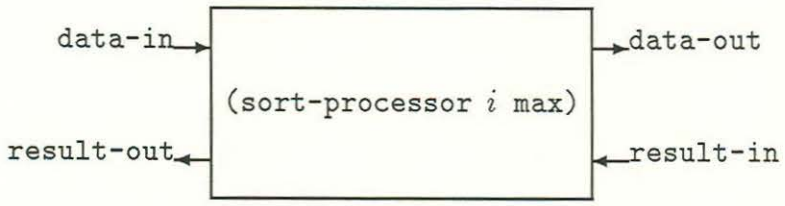
Here the `fibo-c` function has to be provided with both the continuation and the number n whose Fibonacci's value f_n is to be computed. The recursive `fibo-c` either calls the continuation with `n` in the basis case or launches the parallel evaluation of f_{n-1}, f_{n-2} and the expression that will compute f_n from these partial results.

The interesting aspect of this definition is that the continuation passed to the child processes will send the result value to a local channel that will be read by the first child process. This internal communication is *not* visible outside of `fibo-c` because of communication effect masking.

4.2 Systolic sorting

Using an asynchronous systolic array of N processes, we describe here in our new typed framework an algorithm to sort N elements in time $O(N)$ [MC80].

The processing elements are arranged in a linear array. Each processor, implemented as a function, is connected to its two neighbors by four channels. The first two (`data-in` to the left and `data-out` on the right) are used in the first phase of the algorithm. The last two (`result-out` to the left and `result-in` on the right) are only used to output sorted values during the second phase of the algorithm.



The type of a processor element is defined by:

```
(pdefine sort-processor
 (subr (maxeff (in @sort) (out @sort) (alloc @sort))
       ((channelof int @sort)                    ; data-in
        (channelof int @sort)                    ; data-out
        (channelof int @sort)                    ; result-out
```

```
        (channelof int @sort))                    ; result-in
    unit))
```

Each processor is responsible for one element of the input data. In the first phase, each processor, defined by its `position` in the systolic array of `max-position` elements, inputs value from `data-in`, compares it to its current value, keeps the smaller one and output the greater one to its right via `data-out`. At the end of phase one, processor i will have in its store the i-th element of the sorted vector. The second phase will transfer this value on the left `result-out` and forward to the same channel all the superior ones coming from `result-in`.

```
(define (sort-processor (position int) (max-position int))
  (lambda ((data-in (channelof int @sort))
           (data-out (channelof int @sort))
           (result-out (channelof int @sort))
           (result-in (channelof int @sort)))
      (let ((final (do ((i 0 (+ i 1))
                        (v (from data-in)
                           (let* ((new (from data-in))
                                  (min (if (< new v) new v))
                                  (max (if (< v new) new v)))
                              ((to data-out) max)
                              min))
                        ((= (+ i position) max-position) v)))))
           ((to result-out) final)
           (do ((i 0 (+ i 1)))
               ((= (+ i position) max-position) #u)
               ((to result-out) (from result-in))))))
```

The N processors will be glued together by using a reduction-like operation with the `glue-sort-processor`. Note that all the intermediate channels will be created at the time of the setting of the systolic array, and not during the sorting phase.

```
(define (glue-sort-processor (f1 sort-processor)
                             (f2 sort-processor))
  (let ((inter-data ((proj (proj channel @sort) int)))
```

```
      (inter-result ((proj (proj channel @sort) int))))
    (lambda ((data-in (channelof int @sort))
             (data-out (channelof int @sort))
             (result-out (channelof int @sort))
             (result-in (channelof int @sort)))
      (cobegin (f1 data-in inter-data result-out inter-result)
               (f2 inter-data data-out inter-result result-in))
      #u)))
```

The `sort` function receives as input a list of values to sort. It creates an appropriate systolic array `proc` and two channels on which to input and output values. Then, the main program is forked into two processes:

- A driver expression that inputs the values to be sorted in the `input` channel and outputs them to create a new sorted list,
- The systolic array connected to `input` and `output`. (Note that two dummy channels are used as the right channels `result-in` and `data-out`.)

```
(define (sort (to-sort (listof int @=)))
  (let* ((max-position (- (length to-sort) 1))
         (proc (do ((i 0 (+ i 1))
                    (proc (the sort-processor
                               (sort-processor 0 max-position))
                          (glue-sort-processor
                            proc
                            (sort-processor (+ i 1) max-position))))
                   ((= i max-position) proc)))
         (dummy ((proj (proj channel @sort) int)))
         (input ((proj (proj channel @sort) int)))
         (output ((proj (proj channel @sort) int))))
    (car
      (cobegin
        (do ((s to-sort (cdr s)))
            ((null? s) (do ((i 0 (+ i 1))
                            (s (the (listof int @=) ())
                               (cons (from output) s)))
                           ((> i max-position) ((reverse s))))))
```

```
      ((to input) (car s)))
    (proc input dummy output dummy)))))
```

The evaluation of `sort` terminates because every loop used in each process is bounded. Note that generating an efficient code for this kind of communication pattern should not be too difficult because of its regularity.

4.3 Pipeline

We show here how to use the message-based approach to implement a pipeline-style parallel programming in *FX-87*.

A *pipe* is a process that repetitively takes a single input and returns a processed value. If t is the type of the processed value (to simplify the presentation, we will suppose that the incoming value has also this type), then a pipe has type `(pipeof` t`)`:

```
(pdefine (pipeof (t type))
  (subr (maxeff (in @pipe) (out @pipe))
        ((channelof t @pipe)                    ; input
         (channelof t @pipe))                   ; output
    unit))
```

Pipes with mutually disjoint store effects (to simplify, we will suppose here that they are `pure`) can be connected together to create larger pipes. Each elementary pipe operates in parallel with the others.

In our framework, a pipe is constructed with the `pipe` constructor. It takes a processing function `f` as argument and returns a value of type `(pipeof t)`. To terminate a pipe process, we provide the `pipe` constructor with a second argument `eop?` that is a function that tests whether a received value denotes the "end-of-pipe" value. On reception of such a value, the pipe sends this distinguished value to its follower in the pipeline and terminates. Another approach could have been to rely on some sort of automatic *process* garbage collection; we preferred an explicit termination system in this example.

```
(define pipe
  (plambda ((t type))
    (lambda ((f (subr pure (t) t)) (eop? (subr pure (t) bool)))
```

```
    (lambda ((c-in (channelof t @pipe))
             (c-out (channelof t @pipe)))
      (do ((c (from c-in) (from c-in)))
          ((eop? c) ((to c-out) c))
        ((to c-out) (f c))))))
```

A pipeline can be built from these pipe basic blocs. This can be done with the `pipeline` function that takes a variable number of pipes and connect their channels together:

```
(define pipeline
  (plambda ((t type))
    (vlambda (pipes (pipeof t))
      (if (null? pipes)
          (error "PIPELINE expects at least one arg"))
          (let ((rpipes ((reverse pipes))))
              (reduce
               (lambda ((p (pipeof t))
                        (pipes (pipeof t)))
                   (let ((inter ((proj (proj channel @pipe) t))))
                       (lambda ((c-in (channelof t @pipe))
                                (c-out (channelof t @pipe)))
                           (cobegin (p c-in inter)
                                    (pipes inter c-out))
                           #u)))
               ((reverse (cdr rpipes)))
               (car rpipes))))))
```

Note that the intermediate channels are created when the whole pipeline is set, and not when the values are processed. The `pipeline` function returns, when called, a value of type `(pipeof t)`.

To connect a pipe to an I/O stream, we provide two `O-connect` and `I-connect` functions. For instance, the function `O-connect` expects a pipe `p` that computes values to be output, an `IO-out` function to perform the output operation and an `eop?` function to test when the pipe has become useless:

```
(define O-connect
```

```
(plambda ((t type))
  (lambda ((p (pipeof t))
           (IO-out (subr
                    (maxeff (read @IO) (alloc @IO) (write @IO))
                    (t)
                    unit))
           (eop? (subr pure (t) bool)))
    (let ((inter ((proj (proj channel @pipe) t))))
      (lambda ((c-in (channelof t @pipe)))
        (cobegin (p c-in inter)
                 (do ((c (from inter) (from inter)))
                     ((eop? c) #u)
                   (IO-out c)))))))
```

We now have all the necessary facilities to create a toy processing pipe:

```
(define (by-2-plus-1 (l (listof int @=)))
  (let* ((eop -1)
         (eop? (lambda ((x int)) (= x eop)))
         (add-1 (pipe (lambda ((x int)) (+ x 1)) eop?))
         (mul-2 (pipe (lambda ((x int)) (* x 2)) eop?))
         (a-pipe (pipeline mul-2 add-1))
         (input ((proj (proj channel @pipe) int))))
    (cobegin
      (do ((l l (cdr l)))
          ((null? l) ((to input) eop))
        ((to input) (car l)))
      (((proj 0-connect int) a-pipe write-int eop?) input))))
```

The function `by-2-plus-1` expects a list `l` of integers i and outputs the list of processed integers $2*i+1$. The multiplication by 2 and the increment are performed in two parallel pipes.

4.4 Merge

The current proposal doesn't allow explicit non-deterministic constructs. However, non-determinacy comes implicitly from scheduling race conditions

on shared channels. It is then easy to define a non-deterministic `merge` operator which will take five arguments: two pairs of (input channel, end of channel function) and the merged channel. This function will non-deterministically shuffle, in a fair way, the data coming from the two input channels and output a value of `merged` type to the output channel:

```
(pdefine (merged (t1 type) (t2 type))
  (oneof ((left t1) (right t2)) @=))
```

A value of `merged` type is a variant value whose tag can be either `left` or `right` indicating from which channel the value came from. The merge is performed via the `cobegin` construct in a straightforward way:

```
(define merge
  (plambda ((r1 region) (r2 region) (r region))
    (plambda ((t1 type) (t2 type))
      (lambda ((c1 (channelof t1 r1)) (c2 (channelof t2 r2))
               (c (channelof (merged t1 t2) r)))
        (cobegin
          (do ((v (from c1) (from c1)))
              (#f)
            ((to c) (one (merged t1 t2) left v)))
          (do ((v (from c2) (from c2)))
              (#f)
            ((to c) (one (merged t1 t2) right v))))))))
```

Note that the `merge` function (not a real mathematical function because of the non-determinism) doesn't terminate. A terminating version of `merge` could be easily designed by providing supplementary arguments which would test for "end-of-channel" marker values.

5 Implementation

The four previous examples were tested on an experimental implementation of this message-based facility included in the current *FX-87* Interpreter [JG88]. We will briefly describe the Symbolics Lisp Machine implementation.

The basic type-checker was augmented to know about `in` and `out` effects, in particular to deal with effect masking and algebraic properties of these

effect constructors. The `cobegin` special form was included in the type-checking of standard types, which was an easy task to do, except for testing the anti-aliasing property.

But the most interesting part is the extension of the run-time library to support channels and processes.

A channel is implemented as a CommonLISP [S84] structure that includes a lock (managed by Symbolics `locf` construct), a list of pending messages and a flag `initial-p` to memorize the state of the channel (initial or not). The structure is allocated by the `fx-channel` function.

Every process that performs a `to` or `from` function call on a given channel tries to acquire the lock of the channel and, as soon as it succeeds (this may induce the blocking of the process), updates the channel information and unlocks. These two functions are respectively implemented by `fx-to` and `fx-from`.

A process is implicitly created by the `cobegin` special form. The Scheme program generated by the *FX-87* Interpreter during the intermediate code generation phase looks like this:

```
(code-gen cobegin) =
  (let ((ch1 (gensym "ch1"))
        (ch2 (gensym "ch2")))
    `(let ((,ch1 (fx-channel))
           (,ch2 (fx-channel)))
       (fx-process-run-function
         (lambda ()
           ((fx-to ,ch1) ,(code-gen (cobegin-left cobegin)))))
       (fx-process-run-function
         (lambda ()
           ((fx-to ,ch2) ,(code-gen (cobegin-right cobegin)))))
       (cons (fx-from ,ch1) (fx-from ,ch2))))
```

Note that channels are used to implement the semantics of `cobegin` return value. The `fx-process-run-function` uses the Symbolics function `process-run-function` to launch processes; this function performs a fork to evaluate its argument concurrently. No join operation is possible.

6 Benefits of Communication Effects

We saw that channels are powerful enough to express many other parallel paradigms; we showed above how other typical paradigms can easily be interpreted within the framework of channels. Communication effects are useful in many other different ways:

- Communication effects allow explicit parallelism to be expressed by the programmer in a typed language that permits higher-order functions and mutations while preserving the possibility of implicit parallelism detection by the compiler. More specifically, if two expressions only interfere via communication effects, they can be implicitly scheduled in parallel if they operate on channels located in different regions. Moreover, by using regions inside of communication effects, this implicit parallelism can be detected at a finer grain than would have been possible with less sophisticated frameworks.

- By only allowing processes to communicate via channels (i.e., not via shared data structures), communication effects permit efficient implementations of *FX-87* on distributed computer architectures that don't provide a large global shared memory. Moreover, since these restrictions are compile-time checkable, a compiler can enforce such a policy of use of explicit parallelism.

- By eliminating race-conditions on global data structures between processes, communication effects help the programmer write parallel programs. One of the major problems in the design of parallel or concurrent algorithms is the mastering of non-determinacy. Restrictions on communication between processes enforced at compile-time by the *FX-87* effect system limit non-determinacy to shared access to channels, thus limiting the amount and spread of its usage. Moreover, with communication effect masking, subroutines that use internal parallelism will have no communication effects in their latent effect.

- Via communication effect masking, communication effects help reason about programs. Subroutines that use internal and masked parallelism can be understood and manipulated as if they were sequential, thus allowing standard reasoning techniques to be used in these cases. In case

of non-determinacy induced programming errors, the programmer can limit his attention to subroutines that exhibit communication effects.

- By appearing in the types of values, communication effects improve the documentation of programs. For instance, by using region within channel types, the programmer can expose the intended semantics of a channel (e.g., exchanging request messages if use the region is `@request`).

- Communication effects can be used to enforce other correctness conditions. For instance, no top-level expression should be allowed to have a (`in` *rexp*) and no (`out` *rexp*) effect, or vice-versa. If this condition were not verified, then an inside process could be indefinitely waiting for an impossible communication on the "orphan" channel.

- By sequentializing accesses to channels located in the same region within the same process, communication effects can be used to enforce semantic constraints between communications. For instance, if a programmer used two different channels located in the same region to monitor the same device, the *FX-87* system would correctly prohibit these commands to be reordered.

- Communication effects are easy to implement in a language that uses a type and effect systems. Simple modifications to the type and effect check phases can accommodate these new type and effect constructors. We showed that a simulation of explicit parallelism on a sequential computer is easy to implement.

7 Related Work

Message-based concurrency systems, based on CSP [H78], have been used in a large variety of systems ranging from transputers to VLSI [M85]. These systems are generally limited to messages of basic types, like characters or bytes. We allow any value (even channels and functions) to be transmitted on channels; we gave an example of this in the second Fibonacci parallel implementation. Moreover, these systems do not provide the rich expressive power of an effect system.

Lucassen's thesis [L87] introduces a system for explicit concurrency in *MFX*, a minimal subset of *FX-87*, based on monitors [H74]. This approach

is well suited to a parallel architecture with a global shared memory. To deal with massive parallelism, such a framework may not be realistic. We believe that using a distributed memory model may lead to a more efficient use of a large number of processors. In such a model, processes have only an immediate access to local memories and communicate via messages sent on channels.

8 Conclusion

We described a new framework to express concurrency. It uses an *effect system* to describe the communication behavior of expressions in a typed polymorphic programming language. Concurrency occurs between *processes* connected by *polymorphic channels* on which *messages* are transmitted. Communication behaviors are characterized by two *communication effect* constructors, `out` and `in`, depending on whether a message has been sent or received. Data synchronization is only allowed by message-passing on shared channels; communication via mutation of shared global variables is automatically ruled out at compile-time by the effect system. Unobservable communication effects are masked.

We used this system to express other parallel paradigms and gave examples for systolic and pipelined programming. The programmer can thus express concurrency in its own way while preserving the correctness of detection of implicit parallelism and optimizations by the compiler.

This new concurrency framework was added to the *FX-87* programming language and implemented on Symbolics Lisp Machine, by extending the *FX-87* Interpreter with a few definitions for the typechecking phase and providing a simple run-time support for parallel execution.

References

[F86] Francez, N. *Fairness.* Springer Verlag, 1986.

[GJLS87] Gifford, D. K., Jouvelot, P., Lucassen, J. M., and Sheldon, M. A. *The FX-87 Reference Manual.* MIT/LCS/TR-407, 1987.

[**H74**] Hoare, C. A. R. Monitors: An Operating System Structuring Concept. *Communications of the ACM 10*, 17 (1974).

[**H78**] Hoare, C. A. R. Communicating Sequential Processes. *Communications of the ACM 8*, 21 (1978).

[**JG88**] Jouvelot, P. and Gifford, D. K. The *FX-87* Interpreter. In *Proceedings of the 2nd IEEE International Conference on Computer Languages*. IEEE, New York, 1988, pp. 65-72.

[**L87**] Lucassen, J. M. *Types and Effects: Towards the Integration of Functional and Imperative Programming*. PhD Dissertation, MIT/LCS/TR-408, 1987.

[**LG88**] Lucassen, J. M., and Gifford, D. K. Polymorphic Effect Systems. In *Proceedings of the 15th Annual ACM Conference on Principles of Programming Languages*. ACM, New York, 1988, pp. 47-57.

[**M79**] McCracken, N. J. *An Investigation of a Programming Language with a Polymorphic Type Structure*. PhD Dissertation, Syracuse University, 1979.

[**M85**] Milne, G. J. CIRCAL and the Representation of Communication, Concurrency and Time. *ACM Trans. on Prog. Lang. and Syst. 2*, 7 (1985).

[**MC80**] Mead, C., and Conway, L, *Introduction to VLSI Design*, Addison Wesley, 1980.

[**R86**] Rees, J. A., and Clinger, W. Eds. *The Revised^3 Report on the Algorithmic Language Scheme*. MIT/AI Memo 848a, 1986.

[**S78**] Steele, G. L., and Sussman, G. J. *The Art of the Interpreter or, The Modularity Complex (Parts Zero, One and Two)*. MIT/AI Lab Memo 453, 1978.

[**S84**] Steele, G. L. *CommonLISP*. Digital Press, 1984.