Towards a Strongly Typed Functional Operating System

Arjen van Weelden and Rinus Plasmeijer

Computer Science Institute
University of Nijmegen
Toernooiveld 1, 6525 ED Nijmegen, The Netherlands
arjenw@cs.kun.nl, rinus@cs.kun.nl

Abstract. In this paper, we present Famke. It is a library for Clean that enables the creation and management of independent distributed processes (written in Clean) on a network of computers. The main feature of Famke is that values of any type, i.e. data and code, can be communicated between independent processes in a type safe way. Famke uses Clean's dynamic types and its dynamic linker to extend running applications with new code (plug-ins) that, if its type matches the types used in the application, are guaranteed to fit. Clean no longer offers any support for concurrent evaluation, but fortunately, we can realize threads, signalling and exception handling by using first class continuations without the need for additional run-time support. We have made an interactive shell on top of Famke with which the user can manipulate processes interactively. The shell uses a functional-style command language and it type checks the command line before performing it. Preemptive scheduling is done by the underlying operating system (currently Microsoft Windows) but cooperative scheduling is done by Famke. Famke has been made in such a way that it could very well serve as the kernel of a stand-alone operating system entirely written in a pure functional language.

1 Introduction

Functional programming languages like Haskell [1] and Clean [2] offer a very flexible and powerful static type system. Compact, reusable, and readable programs can be written in these languages while the static type system is able to detect many programming errors at compile time. But this works only for a single program.

Independently developed applications often need to communicate with each other. One would like the communication of objects of any type to take place in a type safe manner as well. In practice, this is not so easy to realize: the compile time type information is generally not kept inside a compiled executable, and therefore cannot be used at run-time. In real life therefore, applications often only communicate simple data types like streams of characters, ASCII text, or use some ad-hoc defined (binary) format. Although more and more applications use XML to communicate data together with the definitions of the data types used,

most programs do not support run-time type unification, cannot use previously unknown data types and cannot exchange functions or code.

It would be great if we could communicate expressions of any of the types modern functional programming languages offer. In particular, we want to be able to communicate functions, i.e. code! If we can communicate a (possibly polymorphic) function in a type safe way, we are in principle able to extend a running application with a new plug-in that can be type checked after which it is guaranteed to fit. Clearly, we need a powerful dynamic type system for this purpose and a way to dynamically extend a running application with new code.

Fortunately, the new Clean system offers some of the required basic facilities: it offers a hybrid type system with static as well dynamic typing (dynamics), including run-time support for dynamic linking. However, to realize process management and type safe communication Clean is lacking some important facilities as well.

Clean offers only very limited library support for process creation and process communication. Besides support for heavyweight (distributed) processes it would also be nice to have support for lightweight threads. Old versions of Concurrent Clean [3] did offer sophisticated support for parallel evaluation and lightweight processes. An implementation existed on a network of Apple Macintosh computers and multiprocessor Transputer systems. Unfortunately, Intel compatible Personal Computers turned out to be less suited for an efficient implementation of the required run-time support and, over the years, Microsoft Windows has become the main platform for Clean. Consequently, current versions of Clean no longer support the old Concurrent Clean annotations for parallel and interleaved evaluation.

In the context of processes, exception handling is a very important facility. Many strict functional programming languages offer an exception handling mechanism. However, a semantically sound and nice solution for exception handling in a pure and lazy functional language has not yet been proposed. Haskell is an example of pure and lazy languages that does provide exception handling, but only in strict contexts because exceptions can only be caught in the `IO` monad. Clean does not have an exception handling mechanism at all.

In this paper, we present Famke. Famke enables easy creation and management of (distributed) processes and type safe communication between these processes. It makes use of the new facilities offered by Clean (dynamic typing and dynamic linking) and offers threads, processes and exception handling without requiring new language constructs or run-time support. Famke can be seen as a tiny functional operating system written (in Clean) on top of an existing operating system (Windows). This leaning on Windows avoids duplication of work and enables a better integration with existing software (e.g. other processes written in other programming languages, resources and the file system).

With Famke, we want to accomplish the following objectives without being forced to make any changes to the Clean compiler or run-time system.

– Present an interface for the Clean programmer with which it is easy to create (distributed) processes that can communicate any expression of any type in a type safe way;
– Present a typed interactive shell for the user with which it is easy to manage, apply and combine (distributed) processes interactively and in a type safe way;
– Achieve a modular design using an extendable micro kernel approach;
– Achieve a reliable system by using static types where possible and (early) dynamic type checks only when static types are insufficient;
– Achieve a system that is easy to port to another operating system.

The outline of this paper is as follows: Sect. 2 introduces the static/dynamic hybrid type system of Clean. In Sect. 3, we will show a way to build cooperative threads and exceptions in Clean, without changing the compiler or the run-time system. The implementation of exception handling will show that dynamics are very useful as containers for handling values of any type. In Sect. 4 we presents an interface for communication between threads and an interface for creating preemptive processes. Again, the dynamics play an important role in the implementation of run-time checked type safe communication. Sending and receiving of dynamics that contain functions, which requires dynamic linking, will be used to implement process management. The last thing we will present in Sect. 5 is the first application that uses our concurrency primitives: an interactive shell that type checks the command line. Related work is discussed in Sect. 6 and we conclude and mention further research in Sect. 7.

2 Dynamics

Clean has recently been extended with a dynamic type system [4] in addition to its static type system. A dynamic is a value of type Dynamic which contains a value as well as a representation of the type of that value.

```
dynamic 42 :: Int
```[1]

Dynamics can be formed using the keyword `dynamic` in combination with the value and an optional type (otherwise the compiler will infer the type), separated by a double colon.

```
matchInt :: Dynamic -> Int
matchInt (x :: Int) = x
matchInt else       = abort "Not an integer"
```

Values of type `Dynamic` can be matched in function alternatives and case patterns. Such pattern matches consist of an optional value pattern and a type pattern. In the example above, `matchInt` returns the value `x` inside the dynamic if it has type `Int`; it aborts if it has any other type.

[1] Numeric literals are not overloaded in Clean, hence the type `Int` instead of `(Num a) => a`.

```
dynamicApply :: Dynamic Dynamic -> Dynamic[2]
dynamicApply (f :: a -> b) (x :: a) = dynamic (f x) :: b
dynamicApply _             _        = dynamic "Cannot apply"
```

A type pattern can contain type variables which, if the run-time unification is successful, are bound to the offered type. In the example above, `dynamicApply` tests if the type of the function `f` inside its first argument can be unified with the type of the value `x` inside the second argument. If this is the case then `dynamicApply` can safely apply `f` to `x`. The result of the application has type `b`. At compile time it is generally unknown what this type `b` will be, but, since `b` will be instantiated by the unification, the result can be wrapped into a dynamic again.

```
:: Maybe a = Nothing | Just a[3]

matchDynamic :: Dynamic -> Maybe t | TC t[4]
matchDynamic dyn = case dyn of
                     (x :: t^) -> Just x
                     _         -> Nothing
```

Type variables in dynamic patterns can also relate to a type variable in the static type of a function. A carrot behind a variable in a pattern tells the compiler that the type variable has to corresponds with the static type variable with the same name in the static type of the function. The static type variable then becomes overloaded in the predefined `TC` (type code) class. In the example above, the static type `t` will be determined by the context in which `matchDynamic` is used, and will impose a restriction on the actual dynamic type that is accepted by `matchDynamic`. `matchDynamic` is therefore an example of a type dependent function. It yields `Just` the value inside the dynamic (if the dynamic contains a value of the required, context dependant, type) or `Nothing` (if it does not).

```
writeDynamic :: String Dynamic *World -> (Bool, *World)[5]
readDynamic :: String *World -> (Bool, Dynamic, *World)
```

The dynamic run-time system supports writing dynamics to disk and reading them in again, possibly in another program or during another run of the same program. When a dynamic has been read in, a run-time unification (caused by a pattern match on the dynamic) has succeeded and the value contained in the dynamic is actually used, the dynamic run-time system will instruct the dynamic

[2] Function types in Clean separate arguments by white space instead of `->`.

[3] A `::`, instead of the `data` keyword of Haskell, precedes a type definition in Clean. This `Maybe` type is exactly the same as the Haskell type `Maybe`.

[4] Clean denotes overloading in a class `K` as: `a | K a`, whereas Haskell uses `(K a) => a`.

[5] The `*` in front of the `World` is a uniqueness attribute indicating that the World environment will be passed around in a single-threaded way. Clean's type checker will reject sharing of unique objects. Unique objects allow safe destructive updates and are therefore also used to do I/O in Clean. The value of type `World` in Clean corresponds to the hidden state of the `IO` monad in Haskell.

linker to automatically link in the required data and code. Later on, we will use Clean's ability to read in typed data and to link in code at run-time to implement type safe communication of values of any type.

```
dynamicToString :: Dynamic -> (Bool, String)
stringToDynamic :: String -> (Bool, Dynamic)
```

The run-time system also provides functions to convert dynamics to its string representation and back, which are also internally used by `writeDynamic` and `readDynamic`. Because we do not want to read and write files each time we want to send a message to someone, we will use `dynamicToString` and `stringToDynamic` when we introduce our communication interface.

3 Threads

Although, Windows offers threads to enable multi-tasking within a single process, there is no run-time support for making use of these preemptive threads in Clean. We can use the preemptive processes that Windows provides by starting multiple Clean programs, but this is not practical. In order to give the programmer the choice between processes and threads we decided to implement threads ourselves in Clean, and to provide an interface to the Windows processes later on. Unfortunately, because Clean generates sequential code, we can only provide cooperative threads. The advantage of this approach is that these threads are very lightweight, and implementing them in Clean does not require any special run-time support. As a result, threads can be used independently of any underlying operating system. This makes it easier to eventually build in an operating system written entirely in Clean.

In order to implement threads we need a way to save running computations and to resume them later. Wand [5] shows us that this can be done using continuations and the call/CC construct offered by Scheme and other functional programming languages. We implement threads by using first class continuations in Clean, without the usage of call/CC (with, according to Thielecke [6] can be dangerous).

```
:: Thread a :== (a -> KernelOp) -> KernelOp[6]
:: KernelOp :== Kernel -> Kernel

threadExample :: Thread a
threadExample = \cont kernel -> cont x kernel‘
where
  x = ...                // calculate argument for cont
  kernel‘ = ...kernel... // operate on the kernel state
```

A function of the type Thread, such as the example function above, gets the tail of a computation (named `cont`; of type `a -> KernelOp`) as its argument and combines that with a new computation step, which calculates the argument

[6] Clean uses `:==` to indicate a type synonym. Haskell uses the `type` keyword.

(named `x`) for the tail computation, to form a new function (of type `KernelOp`). This function yields when evaluated on a kernel state (named `kernel`; of type `Kernel`) a new kernel state.

```
:: ThreadId  // abstract thread id

:: *Kernel[7] = {currentId  :: ThreadId, newId :: ThreadId, world :: *World,
                 ready :: [ThreadInfo], paused :: [ThreadInfo]}

:: ThreadInfo = {threadId :: ThreadId, threadCont :: KernelOp}

:: Void = Void  // written more elegantly as () in Haskell
```

The kernel state is a record that contains the information required to do the scheduling of the threads. It contains information like the current running thread (named `currentId`), the threads that are ready to be scheduled (in the `ready` list), paused or blocked threads (in the `paused` list), and the `world` state which is provided by the Clean run-time system.

```
newThread :: (Thread any) -> Thread ThreadId
newThread thread = \c k=:{newId, ready}[8] ->
  let threadInfo = {threadCont = thread ignoreResult, threadId = newId}[9] in
  c newId {k & newId = inc newId, ready = ready ++ [threadInfo]}
where
  ignoreResult x kernel = kernel

threadId :: Thread ThreadId
threadId = \c k=:{currentId} -> c currentId k

pauseThread :: Thread Void
pauseThread = \c k=:{currentId, paused} ->
  let threadInfo = {threadCont = c Void, threadId = currentId} in
  {k & paused = [threadInfo:paused]}

resumeThread :: ThreadId -> Thread Void
resumeThread id = \c k=:{ready, paused} ->
  let (threadInfo, paused) = extract id paused in
  c Void {k & ready = ready ++ [threadInfo]}
where
  extract id list = ...  // return and remove from list

liftIO :: (*World -> (a, *World)) -> Thread a
liftIO f = \c k=:{world} ->
  let (x, world') = f world in c x {k & world = world'}
```

[7] Record types in Clean are surrounded by { and }. The `*` before `Kernel` indicates that the record must always be unique. In the rest of the code, therefore, the `*` can then be left out in front of `Kernel`.

[8] In Clean, `r=:{f}` denotes the selection of the field `f` in the record `r`.

[9] `{r & f = v}` denotes a new record value that is equal to `r` except for the field `f`, which is equal to `v`.

The `newThread` function starts the given thread concurrently with the other running and paused threads within the current process. Threads are primarily used to have an effect on the kernel and world state. They therefore do not yield a result, hence the polymorphically parameterized `Thread any` type. It relieves our system from the additional complexity of returning a result to the parent thread. The communication primitives that will be introduced later, enable programmers to extend the `newThread` primitive to deliver a result to the parent.

Threads can obtain their thread identification with `threadId`. A thread can pause or block itself with `pauseThread`. Another thread can resume the paused thread with the help of `resumeThread`. In order to use functions defined in the Clean run-time system that operate on the world state, `liftIO` is available. It evaluates the given function on the current world state, stored in the kernel state. Ideally, `liftIO` should only be used in device drivers and other low level I/O code.

```
yieldThread :: Thread Void
yieldThread = \c k -> yield (c Void) k

yield :: KernelOp Kernel -> Kernel
yield c k=:{currentId, ready} = {k & ready = ready ++ [threadInfo]}
where
  threadInfo = {threadId = currentId, threadCont = c}

schedule :: Kernel -> Kernel
schedule k=:{ready = []} = k
schedule k=:{ready = [threadInfo:tail]}
  let threadId, threadCont = threadInfo
      k‘ = {k & ready = tail, currentId = threadId}
      k‘‘ = threadCont k‘  /* evaluate the thread */  in
  = schedule k‘‘

StartFamke :: (Thread any) World -> World
StartFamke mainThread world = (schedule kernel).world
where
  firstId = ...  // first thread id
  kernel = {currentId = firstId, newId = inc firstId,
            ready = [threadInfo], paused = [], world = world}
  threadInfo = threadId = firstId, threadCont = mainThread ignoreResult
  ignoreResult _ kernel = kernel
```

Scheduling of the threads is done cooperatively. This means that threads must force rescheduling themselves occasionally using the yieldThread function. The schedule function then evaluates the next ready thread. `StartFamke` can be used much like the standard Clean `Start` function to start the evaluation of the main thread.

A thread that is currently being evaluated returns directly to the scheduler whenever it performs a `yield`(`Thread`) action, because `yield` does not evaluate the tail of the computation. Instead, it stores the continuation at the back of the ready queue (to achieve round-robin scheduling) and yields the current kernel

state. The scheduler then uses this new kernel state to evaluate the next ready thread.

Programming threads using a continuation style is cumbersome, in particular because one often has to perform an explicit yield. Therefore, we added thread-combinators resembling a more common monadic programming style. Our `return`, `>>=` and `>>` functions resemble the monadic `return`, `>>=` and `>>` functions of Haskell.

Whenever a running thread performs a `bind` or `return`, control is voluntarily given to the scheduler, using `yield`.

```
return :: a -> Thread a
return x = \c k -> yield (c x) k

(>>=) infixl 1[10] :: (Thread a) (a -> Thread b) -> Thread b
(>>=) l r = \c k -> l (\x -> yield (r x c)) k

(>>) infixl 1 :: (Thread a) (Thread b) -> Thread b
(>>) l r = l >>= \_ -> r

combinatorExample =
  calculateX >>= \x ->
  doSomething >>
  return x
```

Unfortunately, Clean does not support the Haskell do-notation for monads, which would make the code even more readable.

Our continuation implementation of concurrent threads allows us, for example, to construct the UNIX fork operation on top of the `newThread` primitive.

```
fork :: Thread (Maybe ThreadId)
fork = \cont kernel ->
  let child = \_ -> cont Nothing
      parent = \childId -> cont (Just childId) in
  newThread child parent kernel

forkExample =
  fork >>= \maybe ->
  case maybe of
    Nothing -> ...       // child branch
    Just childId -> ...  // parent branch
```

The fork function returns `Nothing` to the child and returns `Just` the child's thread id to the parent. Both the child and the parent concurrently execute the same tail of the computation. `fork` accomplishes this by catching the given continuation (named `cont`) that represents the tail of the computation and creating a new thread using `newThread`.

[10] `(`*op*`) infix`*a n* defines an infix operator called *op* in Clean. *a* indicates left, right or no associativity and *n* indicates its priority.

3.1 Exceptions

Most of the primitive thread operations may fail because of external conditions such as errors returned by the underlying operating system or by other threads. Programs can therefore easily become cluttered with lots of error checking code. An elegant solution for this kind of problem is the use of exception handling.

There is no exception handling mechanism in Clean, but our thread continuations can easily be extended to handle exceptions. Therefore, exceptions can only be thrown or caught by a thread, just as exceptions can only be caught in the `IO` monad in Haskell (using their `ioError` and `catch` functions).

To provide exception handling, the enhanced thread continuations do not only have a continuation argument for success, but they also have a continuation argument for the case that an exception was thrown.

```
:: Thread a  :== (SuccCnt a) ExcCnt -> KernelOp
:: SuccCnt a :== a ExcCnt -> KernelOp
:: ExcCnt    :== Exception -> KernelOp
```

Exceptions are implemented using dynamics, which make it possible to store any value and to easily extend the set of exceptions.

```
:: Exception = Exception Dynamic

toException :: e -> Exception | TC e
toException e = Exception (dynamic e)

throw :: e -> Thread any | TC e
throw x = \sc ec k -> ec (toException x) k

try :: (Thread a) (Exception -> Thread a) -> Thread a
try thread catcher =
  \sc ec k -> thread (\x _ -> sc x ec) (\e -> catcher e sc ec) k

rethrow :: Exception -> Thread any
rethrow exception = \sc ec k -> ec exception k

(>>|) infixr 1 :: (e -> a) (Exception -> a) -> (Exception -> a) | TC e
(>>|) catchThis catchOther =
  \(Exception exception) -> case exception of
      (this :: e^) -> catchThis this
      other -> catchOther other
```

The `throw` function wraps values in an exception and throws them to the enclosing try clause. The `try` function catches any exceptions that occur during the evaluation of its first argument (`thread`) and feeds it to its second argument (`catcher`). Because any value can be thrown, exception handlers must be matched against the exceptions. For this purpose, we introduced the `>>|` operator. It is a type dependant function, which uses a dynamic pattern match to check whether the type of the thrown exception is of the expected static type.

If so, it applies the corresponding handler: `catchThis`. Otherwise, the other handler, `catchOther`, is used.

The kernel provides an outermost exception handler that aborts the thread when an uncaught exception remains. This exception handler informs the programmer that an exception was not caught by any of the handlers and shows the type of occurred exception.

The addition of an exception continuation to the thread type also requires small changes in the implementation of the return and bind functions. We also show part of the exception type that is used by thread primitives that may throw exceptions.

```
return :: a -> Thread a
return x = \sc ec k -> yield (sc x ec) k

(>>=) infixl 1:: (Thread a) (a -> Thread b) -> Thread b
(>>=) l r = \sc ec k -> l (\x ec‘ -> yield (r x sc ec‘)) ec k

:: ThreadExceptions = InvalidThreadId | ...
```

Note how the `return` and (`re`)`throw` functions complement each other: `return` evaluates the success continuation while `throw` evaluates the exception continuation. This gives a lean implementation of exception handling because there is no need to test if an exception occurred at every bind or return. The only overhead caused by our exception handling mechanism is the need to carrying the exception continuation along.

Below we present a very simple (and very forced) example of the use of exceptions:

```
:: ArithErrors = DivByZero | Overflow

exceptionExample =
  try (
    divide 42 0
  ) (catchDiv >>| rethrow)
where
  divide x 0 = throw DivByZero
  divide x y = return (x / y)
  catchDiv DivByZero = abort "Division by zero"
  catchDiv Overflow = abort "Arithmetic overflow"
```

The `divide` function in the example throws the value `DivByZero` as an exception when the programmer tries to divide by zero. Exceptions caught in the try body that contain values of the type `ArithErrors` are handled by `catchDiv`. Caught exceptions of any other type are handled by `rethrow` and are therefore thrown outside the try body.

The try/catch construction looks nice, thanks to the `>>|` operator and the `rethrow` function. Unfortunately, there is no support for identifying the exceptions that a function may throw automatically. This is partly because exception

handling is written in Clean and not built in the language/compiler, partly because exceptions are wrapped in dynamics and can therefore not be expressed in the type of a function. Exceptions of any type can be thrown by any thread, which makes it hard to be sure that all relevant exceptions are caught by the programmer.

3.2 Signals

In a distributed or concurrent setting, there is also a need for throwing and catching exceptions between different threads. We call this kind of exceptions signals (in Haskell they are called asynchronous exceptions). Signals allow threads to detect things like a kill request from other threads.

```
throwTo :: ThreadId e -> Thread Void | TC e
rethrowTo :: ThreadId Exception -> Thread Void

signalsOn :: (Thread a) -> Thread a
signalsOff :: (Thread a) -> Thread a
```

Signals are transferred from one thread to the other by the scheduler. A signal becomes an exception when it arrives that the designated thread, and can therefore be caught in the same way as other exceptions. To prevent interruption by signals, threads can enclose operations in a `signalsOff` clause. Regardless of any nesting, `signalsOn` means interruptible and `signalsOff` always means non-interruptible. It is, therefore, always clear whether program code can or cannot be interrupted. It also allows easy composition and nesting of program fragments that use these functions. When a signal is thrown, control goes to the exception handler. The interruptible state is then restored back to the state it was in before entering the try.

If a thread pauses or blocks inside a `signalsOff` clause, it could be suspended indefinitely. One way to prevent this is to make functions that contain a busy waiting loop interruptible by placing a `signalsOn (return Void)` inside the loop. This creates a small window for signals to occur while waiting for a resource to become available.

```
interrupableUseOfResourceExample =
  signalsOff (
    acquireResource >>= \r ->
    useResource r >>
    releaseResource r
  )
where
  acquireResource =
    attemptAcquire >>= \maybe ->
    case maybe of
      Just resource -> return resource
      Nothing       -> signalsOn (return Void) >>
                       acquireResource
```

The `acquireResource` function in the above example uses the (fictional) non-blocking `attemptAcquire` function to allocate some resource, without allowing interruption by signals. If the resource cannot be allocated, it goes into a busy-waiting loop until it can be allocated. This makes `acquireResource` appear blocking. In order to prevent deadlock and allow interruption, it allows interruption only at moments where it is still safe to interrupt: while the resource cannot be allocated.

4 Processes

Our system uses Windows processes to provide preemptive task switching between groups of threads running inside different processes. Once processes have been created on one or more computers, threads can be started in any one of them. The dynamic linker plays an essential role in getting the code of a thread from one process to another.

4.1 Ports

Elegant ways for type-safe communication between threads are Haskell's M-vars and Clean's lazy graph copying, upon which easy-to-use first class channels can be built. Unfortunately, both solutions do not scale very well to a network of workstations because they require distributed garbage collection.

A common solution that does not need a distributed garbage collector is the use of ports. In contrast with M-vars and channels that can have multiple readers, ports have only one reader: the thread that created the port. Therefore, it is always clear where a port resides: at the location of the reader. This restriction allows a less complex administration to implement ports than to implement distributed M-vars. It also makes reasoning about them easier, especially in combination with failing remote computers. If the port does not reside on the failing computer, it will continue to work after the failure.

```
:: Port msg  // abstract port id

:: PortExceptions = UnregisteredPort | PortInvalidMessage | ...

newPort :: Thread (Port msg) | TC msg
closePort :: (Port msg) -> Thread Void | TC msg

writePort :: (Port msg) msg -> Thread Void | TC msg
writePort port m = writeMailslot port (dynamicToString (dynamic m))

readPort :: (Port msg) -> Thread msg | TC msg
readPort port = readMailslot port >>= \maybe ->
  case maybe of
    Just s -> case (stringToDynamic s) of
                (m :: msg^) -> return m
                _           -> throw PortInvalidMessage
```

```
    _       -> readPort port

registerPort :: (Port msg) String -> Thread Void | TC msg
lookupPort :: String -> Thread (Port msg) | TC msg
```

The `newPort` function creates a new port and `closePort` removes a port. All primitives on ports operate on typed messages. The dynamics run-time system is used to convert the messages to and from a dynamic. The `readPort` and `writePort` function use `dynamicToString` and `stringToDynamic` from the dynamics library to convert a dynamic to and from a string.

The actual sending/receiving of these strings is done via Windows using the mail slot interface (the `readMailslot` and `writeMailslot` functions). The `registerPort` function associates a unique name with a port, by which the port can be looked up using `lookupPort`. This implementation of ports gives us an asynchronous message passing system. Such a system allows the programmer to build other communication and synchronization methods such as: remote procedure calls, semaphores and channels.

Here is a skeleton example of a database server that uses a port to receive functions from clients and applies them to the database.

```
:: DBase = ...  // list of records, or something

server :: DBase -> Thread Void
server db = openPort >>= \port ->
            registerPort port "MyDBase" >>
            handleRequests db
where
  handleRequests db = readPort port >>= \f ->
                      let db‘ = f db /* apply to database */ in
                      handleRequests db‘

client :: Thread Void
client = lookupPort "MyDBase" >>= \port ->
         writePort port mutateDatabase
where
  mutateDatabase :: DBase -> DBase
  mutateDatabase db = ...  // change the database
```

The server creates, and registers, a port that receives functions of the type `DBase -> DBase`. Clients send the functions that perform mutations of the database to the registered port. The registered port has the type `Port (DBase -> DBase)` and therefore accepts only messages of this type. It then waits for messages (i.e. functions) to arrive and applies them to the database `db`. These functions can be safely applied to the database because the dynamic run-time system guarantees that both the server and the client have the same notion of the type of the database: `DBase`.

4.2 Process Management

Since Windows does the preemptive scheduling of processes, our thread scheduler does not need any knowledge about multiple processes. Instead of changing the scheduler, we make our system automatically add a management thread to each process when it is created. This management thread handles signals from other processes and routes them to the designated threads. On request from threads on other processes, it also handles the creation of new threads on its own process.

```
:: Proc                 // abstract process id
:: Location :== String  // Windows PC name

newProc :: Location -> Thread Proc
procLocation :: Proc -> Location

newThreadAt :: Proc (Thread a) -> Thread ThreadId | TC a
threadProc :: ThreadId -> Proc
```

The `newProc` function creates a new process at a given location and returns its process id. The location of a process can be found out using `procLocation`. The `newThreadAt` function starts a new thread in another process, `threadProc` returns the process id where the given thread resides. The creation of a new process is implemented by starting a pre-compiled Windows executable, the loader, which becomes the new process. The loader is a simple Clean program that runs the management thread. The thread is then started inside the new process by sending it to the remote management thread via a typed port. When the management receives the thread, it starts the threads using the local `newThread` function. The dynamic linker on the remote computer then links in the corresponding code of the new thread automatically.

The extension of our system with this kind of heavyweight process enables the programmer to build distributed concurrent applications. A port of our system to operating systems other than Windows would enable distributed programs over a heterogeneous network of computers. If one wants to make Clean programs, that contain parallel algorithms, run on a farm of computers, this is a first step. However, it does require non-trivial changes to the original program. These changes include splitting the program code into separate threads and making communication between the threads explicit.

Here is an example of starting a thread at a remote computer, in order to evaluate the thread in a preemptive concurrent way, and getting the result back to the parent.

```
:: *Remote a = Remote (Port a)

remote :: Proc (Thread a) -> Thread (Remote a) | TC a
remote pid thread = newPort >>= \port ->
                    newThreadAt pid (thread‘ port) >>
                    return (Remote port)
where
  thread‘ port = thread >>= \result ->
```

```
                writePort port result

join :: (Remote a) -> Thread a | TC a
join (Remote port) = waitPort port >>= \result ->
                     closePort port >>
                     return result
```

The remote function creates a port where the result of the given thread must be sent. It then starts a child thread that calculates the result and writes it to the port, and returns the port enclosed in a remote handle to the parent. When the parent decides that it wants the result, it can use the join function to get it and to close the port.

So far, we have discussed our library that adds support for threads (with exceptions and signals), processes and type-safe communication of values of any type between them. Now it is time to present the first application that makes use of these strongly typed concurrency primitives.

5 The Shell

A shell provides a way to interact with an operating system, usually via a textual command line/console interface. Normally, a shell does not provide a complete programming language, but it does enable users to start pre-compiled programs.

Although most shells provide simple ways to combine multiple programs, e.g. pipelining and parallel execution, and support execution flow controls, e.g. if-then-else constructs, they do not provide a way to construct new programs. Furthermore, they provide very limited error checking before executing the given command line. This is mainly because the programs mentioned at the command line are practically untyped since they work on, and produce, streams of characters. The intended meaning of these streams of characters varies from one program to the other.

Our view on pre-compiled programs differs from the common operating system in that they are dynamics that contain a typed function, and not untyped executables. Programs are therefore typed and our shell puts this information to good use by actually type checking the command line before performing the specified actions. The shell also understands function application and a subset of Clean's constant denotations. The shell syntax closely resembles Haskell's do-notation. It has been extended with operations to read and write files.

Here follow some examples of command lines.

```
map (add 1) [1..10]
```

The `map` and `add` are unbound names in this example and our shell therefore assumes that they are names of files. All files are supposed to contain dynamics, which combined represent a typed file system. The shell reads them in from disk and inspects the types of the dynamics. It uses the types of map (let us assume: `(Int -> Int) [Int] -> [Int]`), add (let us assume: `Int Int -> Int`) and the list comprehension (which has type: `[Int]`) to type-check the command line. If

this succeeds, which it should given the types above, the shell applies the partial application of `add` with the integer one to the list of integers from one to ten, using the `map` function.

```
inc <- add 1; map inc [2,4..10]
```

Defines a variable with the name `inc` as the partial application of the `add` function to the integer one. Then it applies the `map` function using the variable `inc` to the list of even integers from two to ten.

```
inc <- add 1; map inc ['a'..'z']
```

Defines the variable `inc` as in the previous example, but applies it, using the `map` function, to the list of all the characters in the alphabet. This obviously fails with a type error.

```
write "result" (add 1 2); x <- read "result"; x

add 1 2 > result; x < result; x
```

Both the above examples do the same thing, because the `<` (read file) and `>` (write file) shell operators can be expressed using the predefined read and write functions. The sum of one and two is written to the file with the name `result`. The variable `x` is defined as the contents of the file with the name `result`, and the result of the command line is the contents of the variable `x`. In contrast with the `add` and `map` functions that are read from disk by the shell before type checking and executing the command line, `result` is read in during the execution of the command line.

```
pid <- newProc "RemotePC"; newThreadAt pid (3.14 > pi)
```

In this example, the shell is used to start a new process on a remote computer. There must already be at least one other Famke-process (e.g. the shell) running at the computer, named `RemotePC`, to service the request. It then creates a thread inside the new remote process that writes the floating-point number 3.14 to a file named `pi` on the remote computer.

6 Related Work

There are concurrent versions of both Haskell and Clean. Concurrent Haskell [7] offers lightweight threads in a single UNIX process and provides M-vars as the means of communication between threads. Concurrent Clean was only available on multiprocessor Transputers and on a network of single-processor Apple Macintosh computers. Concurrent Clean provided support for native threads on the Transputer systems. On a network of Apple computers, it ran the same Clean program on each processor, providing a virtual multiprocessor system. Concurrent Clean provided lazy graph copying as the primary communication mechanism. Both concurrent systems provided cannot easily provide type safety

between different programs or between multiple incarnations of a single program. The same goes for values written to disk.

Another difference between Famke and the concurrent versions of Haskell and Clean is the choice of communication primitives. Both lazy graph copying and M-vars do not scale very well to a distributed setting because they require distributed garbage collection. This issue has led to a distributed version of Concurrent Haskell [8] that also uses ports, but it implementation does not allow functions to be sent over ports.

Both Cooper [9] and Lin [10] have extended Standard ML with threads (implemented as continuations using call/CC) to form a small functional operating system. Both systems implement the basics needed for a stand-alone operating system. However, both of them do not support type-safe communication of any value between different computers.

The university of Utah [11] built two prototypes of a Java operating system. Although they show that Java's extensibility, portable byte code and static/dynamic type system provides a way to build an operating system where multiple Java programs can safely run concurrently, Java (currently) lacks the power of polymorphic and higher-order functions and closures (to allow laziness) that our functional approach offers.

Haskell is an example of a pure and lazy functional programming language that provides exception handling. In [12] support for asynchronous exceptions has been added to Concurrent Haskell. Our implementation of signals follows their approach closely.

The Scheme Shell [13] integrates a shell into the programming language in order to enable the user to use the full expressiveness of Scheme. Es [14] is a shell that supports higher-order functions and allows the user to construct new functions at the command line. Neither shell provides a way to read and write typed objects from and to disk, and they cannot provide our type safety because they operate on untyped executables.

7 Conclusions and Future Work

In this paper, we presented a functional operating system interface, which could be used as the kernel of a stand-alone strongly typed functional operating system. We extended the lazy and pure functional programming language Clean with lightweight threads, exceptions and heavyweight processes, provided a type safe communication mechanism and we have built a typed interactive shell. With the help of these mechanisms it becomes feasible to build distributed concurrent Clean programs running on a heterogeneous network like the Internet. Nevertheless, there remain issues that need further research.

The current implementation of ports does not check if the name is truly unique (when registering) or even exists (when looking up), entrusting this responsibility upon the programmer. The inability of Windows to detect mail slots on other computers makes it hard to implement a more robust version of the lookup function. Since we cannot be sure that the port exists, and is not too busy

to respond, we do not want to communicate with it to perform the dynamic type check. This makes it possible to send a message of the wrong type. Fortunately, this situation will be detected at run-time because it causes an exception at the receiving end. However, we agree that it ought to be detected at the time of the lookup. These problems might be solved by a smarter implementation of our message passing system using some sort of handshake protocol.

The remote/join example in Sect. 4.2 contains a potential memory leak. The port is not closed when the remote handle becomes garbage without the use of the join function. This problem could be solved using finalizers, which unfortunately are not supported by Clean.

We have done some preliminary research on adding lambda expressions to the shell. In order to automatically derive the type of a lambda expression with the help of the type representation contained within dynamics, we need support for polymorphic types in the dynamic type system. There is work on progress on adding support for polymorphic types to the dynamics implementation, but we were not able to use it just yet. The focus of further research of the Famke project will be increasing the power and usability of the shell.

References

[1] Simon Peyton Jones and John Hughes et al. *Report on the programming language Haskell 98*. University of Yale, 1999. http://www.haskell.org/definition/.

[2] Marko van Eekelen and Rinus Plasmeijer. *Concurrent CLEAN Language Report (version 2.0)*. University of Nijmegen, December 2001. http://www.cs.kun.nl/~clean.

[3] E. G. J. M. H. Nocker, J. E. W. Smetsers, Marko C. J. D. van Eekelen, and Marinus J. Plasmeijer. Concurrent clean. In *PARLE (2)*, pages 202–219, 1991.

[4] M. Pil. Dynamic types and type dependent functions. In T. Davie K. Hammond and C. Clack, editors, *Proceedings of the 10th International Workshop on the Implementation of Functional Languages*, volume LNCS 1595 of *Lecture Notes in Computer Science*, pages 171–188. Springer-Verlag, 1998.

[5] Mitchell Wand. Continuation-based multiprocessing. In J. Allen, editor, *Conference Record of the 1980 LISP Conference*, pages 19–28, Palo Alto, CA, 1980. The Lisp Company.

[6] Hayo Thielecke. Using a continuation twice and its implications for the expressive power of call/CC. *Higher-Order and Symbolic Computation*, 12(1):47–73, 1999.

[7] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *Conference Record of POPL '96: The* 23rd *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–308, St. Petersburg Beach, Florida, 21–24 1996.

[8] Frank Huch and Ulrich Norbisrath. Distributed programming in Haskell with Ports. *Lecture Notes in Computer Science*, 2011:107–??, 2001.

[9] Eric C. Cooper and J. Gregory Morrisett. Adding threads to standard ML. Technical Report CMU-CS-90-186, Pittsburgh, PA, 1990.

[10] Albert C. Lin. Implementing concurrency for an ml-based operating system.

[11] Godmar Back, Patrick Tullmann, Leigh Stoller, Wilson C. Hsieh, and Jay Lepreau. Java operating systems: Design and implementation. Technical Report UUCS-98-015, 6, 1998.

[12] Simon Marlow, Simon L. Peyton Jones, Andrew Moran, and John H. Reppy. Asynchronous exceptions in haskell. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 274–285, 2001.
[13] O. Shivers. A scheme shell. Technical Report MIT/LCS/TR-635, 1994.
[14] Paul Haahr and Byron Rakitzis. Es: A shell with higher-order functions. In *USENIX Winter*, pages 51–60, 1993.