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,rinus}@cs.kun.nl

Abstract. In this paper, we present Famke. It is a prototype implementation of a strongly typed operating system written in Clean. Famke enables the creation and management of independent distributed Clean processes on a network of workstations. It uses Clean's dynamic type system and its dynamic linker to communicate values of any type, e.g. data, closures, and functions (i.e. compiled code), between running applications in a type safe way. Mobile processes can be implemented using Famke's ability to communicate functions. We have built an interactive shell on top of Famke that enables user interaction. The shell uses a functional-style command language that allows construction of new processes, and it type checks the command line before executing it. Famke's type safe run-time extensibility makes it a strongly typed operating system that can be tailored to a given situation.

1 Introduction

Functional programming languages like Haskell [1] and Clean [2,3] 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 within a single application.

Independently developed applications often need to communicate with each other. One would like the communication of objects to take place in a type safe manner as well. And not only simple objects, but objects of any type, including functions. In practice, this is not 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 or cannot exchange functions (i.e. code) between different programs in a

[⋆] This work was supported by STW as part of project NWI.4411.

type safe way. This is mainly because the used programming language has no support for such things.

In this paper, we present a prototype implementation of a micro kernel, called *Famke* (emphfunction*a*l *m*icro *k*ernel *e*xperiment). It provides explicit non-deterministic concurrency and type safe message passing for all types to processes written in Clean. By adding servers that provide common operating system services, an entire strongly typed, distributed operating system can be built on top of Famke.

Clearly, we need a powerful dynamic type system [4] 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 as dynamic typing (*dynamics*) [5], including run-time support for *dynamic linking* [6] (currently on Microsoft Windows only). To achieve type safe communication, Famke uses the above mentioned facilities offered by Clean to implement *lightweight threads*, *processes*, *exception handling* and *type safe message passing* without requiring additional language constructs or run-time support.

It also makes use of an underlying operating system to avoid some low-level implementation work and to integrate better with existing software (e.g. resources such as the console and the file system). With Famke, we want to accomplish the following objectives without changing the Clean compiler or run-time system.

- Present an interface (API) for Clean programmers with which it is easy to create (distributed) processes that can communicate expressions of any type in a type safe way;
- Present an interactive shell with which it is easy to manage, apply and combine (distributed) processes, and even construct new processes interactively. The shell should type check the command line before executing it in order to catch errors early;
- Achieve a modular design using an extensible micro kernel approach;
- Achieve a reliable system by using static types where possible and, if static checking cannot be done (e.g. between different programs), dynamic type checks;
- Achieve a system that is easy to port to another operating system (if the Clean system supports it).

We will introduce the static/dynamic hybrid type system of Clean in section 2. Sections 3 and 4 present the micro kernel of Famke, which provides cooperative thread scheduling, exception handling, and type safe communication. It also provides an interface to the preemptively scheduled processes of the underlying operating system. These sections are very technical, but necessary to understand the interesting sections that follow. On top of this micro kernel an interactive shell has been implemented, which we describe in section 5. During these sections the crucial role of dynamics will become apparent. Related work is discussed in section 6 and we conclude and mention future research in section 7.

2 Dynamics in Clean

Clean has recently been extended with a polymorphic dynamic type system [4,5,6] in addition to its static type system. Here, we will give a small introduction to dynamics in Clean. 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 (i.e. lifted from the static to the dynamic type system) 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.

```
:: Maybe a = Nothing | Just a
```
[2]

```
matchInt :: Dynamic -> Maybe Int
matchInt (x :: Int) = Just x
matchInt  other     = Nothing
```

Values of type `Dynamic` can be matched in function alternatives and case patterns to bring them from the dynamic back into the static type system. Such pattern matches consist of an optional value pattern and a type pattern. In the example above, `matchInt` returns `Just` the value contained inside the dynamic if it has type `Int`; and `Nothing` if it has any other type. The compiler translates type pattern matches into run-time type unifications. If the unification fails, the next function alternative is tried, as in a common pattern match.

```
dynamicApply :: Dynamic Dynamic -> Dynamic
dynamicApply (f :: a -> b) (x :: a) = dynamic f x :: b
dynamicApply  _              _      = dynamic "Error: cannot apply"
```
[3]

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 this application has type `b`. At compile time it is generally unknown what this type `b` will be. The result can be wrapped into a dynamic (and only a dynamic) again, because the type variable `b` will be instantiated by the run-time unification.

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

Type variables in dynamic patterns can also relate to a type variable in the static type of a function. Such functions are called type dependent functions. A

[1] Numeric literals are not overloaded in Clean, hence 42 has type `Int` instead of Haskell's `(Num a) => a`.

[2] A `::`, instead of the `data` keyword of Haskell, precedes a type definition in Clean.

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

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

carrot (^) behind a variable in a pattern associates it with the 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 [5]. In the example above, the static type `t` will be determined by the static context in which it is used, and will impose a restriction on the actual type that is accepted at run-time by `matchDynamic`. It yields `Just` the value inside the dynamic (if the dynamic contains a value of the required context dependent type) or `Nothing` (if it does not).

The new dynamic run-time system of Clean [6] supports writing dynamics to disk and reading them in again, possibly in another program or during another execution of the same program.

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

The dynamic will be read in lazily after a successful run-time unification (triggered by a pattern match on the dynamic). The amount of data and code that the dynamic linker will link in, is therefore determined by the amount of evaluation of the value inside the dynamic. Dynamics written by a program can be safely read by any other program, providing a form of persistence and a rudimentary means of communication.

The ability of Clean, as well as other functional languages, to construct new functions (e.g. currying and higher-order functions) in combination with Clean's new support for run-time linking, enables us to extend a running application with new code that can be type checked after which it is guaranteed to fit.

3 Threads in Famke

Here we show how a programmer can construct concurrent programs in Clean, using Famke's thread management and exception handling primitives.

Currently, Clean offers only very limited library support for process management and communication.

Old versions of Concurrent Clean [7] did offer sophisticated support for parallel evaluation and lightweight processes, but no support for exception handling. Concurrent Clean was targeted at deterministic, implicit concurrency, but we want to build a system for distributed, non-deterministic, explicit concurrency.

Porting Concurrent Clean to Microsoft Windows is a lot of work and still would not give us exactly what we want. Although Microsoft 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 could emulate threads using the preemptive processes that Microsoft Windows provides by multiple incarnations of the same Clean program, but this would make the

[5] The `*` in front of `World` is a uniqueness attribute. It indicates that the (state of the) world will be passed around in a unique/single-threaded way. Clean's type checker allows destructive updates, but reject sharing, of such unique objects. Clean's `World` type corresponds to the hidden state of Haskell's `IO` monad.

threads unacceptably heavyweight, and it would prevent them from sharing the Clean heap, and we still would not have exception handling.

Therefore, Famke does her own scheduling of threads in order to keep them lightweight and to provide exception handling.

3.1 Thread Implementation

In order to implement cooperative threads we need a way to suspend running computations and to resume them later. Wand [8] shows that this can be done using continuations and the call/CC construct offered by Scheme and other functional programming languages. We copy this approach using first class continuations in Clean. Because Clean has no call/CC construction, we have to write the continuation passing explicitly. Our approach closely resembles Claessen's concurrency monad [9], but our primitives operate directly on the kernel state using Clean's uniqueness typing, and we have extended the implementation with easily extendable exception handling (see section 3.2).

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

threadExample :: Thread a
threadExample = \cont kernel -> cont x kernel‘
where
  x = ...                //[7] 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 (named `x`) for the tail computation, to form a new function (of type `KernelOp`). This function returns, when evaluated on a kernel state (named `kernel`; of type `Kernel`), a new kernel state.

```
:: ThreadId  // abstract thread id

:: *Kernel[8] = {currentId  :: ThreadId, newId :: ThreadId,
               ready :: [ThreadState], world :: *World}

:: ThreadState = {thrId :: ThreadId, thrCont :: KernelOp}

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

The kernel state (of type `Kernel`) 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

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

[7] This is a single line comment in Clean, Haskell uses `--`

[8] Record types in Clean are surrounded by { and }. The `*` before `Kernel` indicates that the record must always be unique. Therefore, the `*` can then be omitted in the rest of the code.

scheduled (in the `ready` list), and the `world` state which is provided by the Clean run-time system. Clean's uniqueness type system makes these types a little more complicated, but we will not show this in the examples in order to keep them readable.

```
newThread :: (Thread a) -> Thread ThreadId
newThread thread = \cont k=:{newId, ready}[9] ->
  cont newId {k & newId = inc newId, ready = [threadState:ready]}[10]
where
  threadState = {thrId = newId, thrCont = thread (\_ k -> k)}

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

The `newThread` function starts the given thread concurrently with the other threads. Threads are evaluated for their effect on the kernel and the world state. They therefore do not return a result, hence the polymorphically parameterized `Thread a` type. It relieves our system from the additional complexity of returning the 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`.

Scheduling of the threads is done cooperatively. This means that threads must occasionally allow rescheduling using `yield`, and should not run endless tight loops. The `schedule` function then evaluates the next ready thread. `StartFamke` can be used like the standard Clean `Start` function to start the evaluation of the main thread.

```
yield :: KernelOp Kernel -> Kernel
yield cont k=:{currentId, ready} = {k & ready = ready ++ [threadState]}
where
  threadState = {thrId = currentId, thrCont = cont}

schedule :: Kernel -> Kernel
schedule k=:{ready = []} = k  // nothing to schedule
schedule k=:{ready = [{thrId, thrCont}:tail]} =
  let k` = {k & ready = tail, currentId = thrId}
      k`` = thrCont k`  // evaluate the thread until it yields
  in schedule k``

StartFamke :: (Thread a) *World -> *World
StartFamke mainThread world = (schedule kernel).world
where
  firstId = ...  // first thread id
  kernel = {currentId = firstId, newId = inc firstId,
```

[9] `r=:{f}` denotes the (lazy) selection of the field `f` in the record `r`. `r=:{f = v}` denotes the pattern match of the field `f` on the value `v`.

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

```
            ready = [threadState], world = world}
  threadState = {thrId = firstId, thrCont = mainThread (\_ k -> k)}
```

The thread that is currently being evaluated returns directly to the scheduler whenever it performs a `yield` 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 returns 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, because one has to carry the continuation along and one has to perform an explicit yield often. Therefore, we added thread-combinators resembling a more common monadic programming style. Our `return`, `>>=` and `>>` functions resemble the monadic `return`, `>>=` and `>>` functions of Haskell[11]. Whenever a running thread performs an atomic action, such as a return, control is voluntarily given to the scheduler using `yield`.

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

(>>=) :: (Thread a) (a -> Thread b) -> Thread b
(>>=) l r = \cont k -> l (\x -> r x cont) k

(>>) l r = l >>= \_ -> r

combinatorExample = newThread (print ['h', 'e', 'l', 'l', 'o']) >>
                    print ['w', 'o', 'r', 'l', 'd']
where
    print []     = return Void
    print [c:cs] = printChar c >> print cs
```

The `combinatorExample` above starts a thread that prints "`hello`" concurrent with the main thread that prints "`world`". It assumes a low-level print routine `printChar` that prints a single character. The output of both threads is interleaved by the scheduler, and is printed as "`hweolrllod`".

3.2 Exceptions and Signals

Thread operations (e.g. `newThread`) may fail because of external conditions such as the behavior of other threads or operating system errors. Robust programs quickly 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. Because of this, exceptions can only be thrown or caught by a thread. This is analogous to Haskell's `ioError` and `catch` functions, with which exceptions can only be caught in the `IO` monad.

In contrast to Haskell exceptions, we do not want to limit the set of exceptions to system defined exceptions and strings, but instead allow any value. Exceptions

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

are therefore implemented using dynamics. This makes it possible to store any value in an exception and to easily extend the set of exceptions at compile-time or even at run-time. To provide this kind of exception handling, we extend the `Thread` type with a continuation argument for the case that an exception is thrown.

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

:: Exception :== Dynamic

throw :: e -> Thread a | TC e
throw e = \sc ec k -> ec (dynamic e :: e^) k

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

The `throw` function wraps a value in a dynamic (hence the `TC` context restriction) and throws it to the enclosing `try` clause by evaluating the exception continuation (`ec`). `rethrow` can be used to throw an exception without wrapping it in a dynamic again. The `try` function catches 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 match against the type of the exception using dynamic type pattern matching.

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

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

(>>=) :: (Thread a) (a -> Thread b) -> Thread b
(>>=) l r = \sc ec k -> l (\x -> r x sc) ec k
```

The addition of an exception continuation to the thread type also requires small changes in the implementation of the `return` and `bind` functions. Note how the `return` and `throw` functions complement each other: `return` evaluates the success continuation while `throw` evaluates the exception continuation. This implementation of exception handling is relatively cheap, 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 carry the exception continuation along.

```
:: ArithErrors = DivByZero | Overflow
```

```
exceptionExample = try (divide 42 0) handler

divide x 0 = throw DivByZero
divide x y = return (x / y)

handler (DivByZero :: ArithErrors) = return 0  // or any other value
handler  other                     = rethrow other
```

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 body of the `try` clause are handled by `handler`, which returns zero on a `DivByZero` exception. Caught exceptions of any other type are thrown again outside the try, using `rethrow`.

In a distributed or concurrent setting, there is also a need for throwing and catching exceptions between different threads. We call this kind of inter-thread exceptions *signals*. Signals allow threads to throw kill requests to other threads. Our approach to signals, or *asynchronous exceptions* as they are also called, follows the semantics described by Marlow et. al. in an extension of Concurrent Haskell [11]. We summarize our interface for signals below.

```
throwTo :: ThreadId e -> Thread Void | TC e
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 again when it arrives at 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, during which signals are queued until they can interrupt. Regardless of any nesting, `signalsOn` always means interruptible and `signalsOff` always means non-interruptible. It is, therefore, always clear whether program code can or cannot be interrupted. This allows easy composition and nesting of program fragments that use these functions. When a signal is caught, control goes to the exception handler and the interruptible state will be restored to the state before entering the try.

The try construction allows elegant error handling. Unfortunately, there is no automated support for identifying the exceptions that a function may throw. This is partly because exception handling is written in Clean and not built in the language/compiler, and partly because exceptions are wrapped in dynamics and can therefore not be expressed in the type of a function. Furthermore, 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. But the same can be said for an implementation that uses user defined strings, in which non-matching strings are also not detected at compile-time.

4 Processes in Famke

In this section we will show how a programmer can execute groups of threads using processes on multiple workstations, to construct distributed programs in Clean.

Famke uses Microsoft 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. First we introduce Famke's message passing primitives for communication between threads and processes. The dynamic linker plays an essential role in getting the code of a thread from one process to another.

4.1 Process and Thread Communication

Elegant ways for type-safe communication between threads are Concurrent Haskell's M-Vars [10] and Concurrent Clean's lazy graph copying [7].

Unfortunately, M-Vars do not scale very well to a distributed setting because of two problems, described by Stolz and Huch in [12]. The first problem is that M-Vars require distributed garbage collection because they are first class objects, which is hard in a distributed or mobile setting. The second problem is that the location of the M-Var is generally unknown, which complicates reasoning about them in the context of failing or moving processes. Automatic lazy graph copying allows processes to work on objects that are distributed over multiple (remote) heaps, and suffers from the same two problems.

Distributed Haskell [13,12] solves the problem by implementing an asynchronous message passing system using ports. Famke uses the same kind of ports. Ports in Famke are channels that vanish as soon as they are closed by a thread, or when the process containing the creating thread dies. Accessing a closed port results in an exception. Using ports as the means of communication, it is always clear where a port resides (at the process of the creating thread) and when it is closed (explicitly or because the process died). In contrast with Distributed Haskell, we do not limit ports to a single reader (which could be checked at compile-time using Clean's uniqueness typing). The single reader restriction also implies that the port vanishes when the reader vanishes but we find it too restrictive in practice.

```
:: PortId msg  // abstract port id
:: PortExceptions = UnregisteredPort | InvalidMessageAtPort | ...

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

writePort :: (PortId msg) msg -> Thread Void | TC msg
writePort port m = windowsSend port (dynamicToString (dynamic m :: msg^))

readPort :: (PortId msg) -> Thread msg | TC msg
readPort port = windowsReceive port >>= \maybe ->
                case maybe of
```

```
                  Just s  -> case stringToDynamic s of
                               (True, (m :: msg^)) -> return m
                               other -> throw InvalidMessageAtPort
                  Nothing -> readPort port  // make it appear blocking

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

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

All primitives on ports operate on typed messages. The `newPort` function creates a new port and `closePort` removes a port. `writePort` and `readPort` can be used to send and receive messages. The dynamic run-time system is used to convert the messages to and from a dynamic. Because we do not want to read and write files each time we want to send a message to someone, we will use the low-level `dynamicToString` and `stringToDynamic` functions from the dynamic run-time system library. These functions are similar to Haskell's `show` and `read`, except that they can (de)serialize functions and closures. They should be handled with care, because they allow you to distinguish between objects that should be indistinguishable (e.g. between a closure and its value). The actual sending and receiving of these strings is done via simple message (string) passing primitives of the underlying operating system. The `registerPort` function associates a unique name with a port, by which the port can be looked up using `lookupPort`.

Although Distributed Haskell and Famke both use ports, our system is capable of sending and receiving functions (and therefore also closures) using Clean's dynamic linker. The dynamic type system also allows programs to receive, through ports of type `(PortId Dynamic)`, previously unknown data structures, which can be used by polymorphic functions or functions that work on dynamics such as the `dynamicApply` functions in section 2. An asynchronous message passing system, such as presented here, allows programmers to build other communication and synchronization methods (e.g. 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 like that

server :: Thread Void
server = openPort >>= \port ->
         registerPort port "MyDBase" >>
         handleRequests emptyDBase
where
  emptyDBase = ... // create new data base
  handleRequests db = readPort port >>= \f ->
                      let db‘ = f db in  // apply function to data base
                      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 type `DBase -> DBase`. Clients send functions that perform changes to the database to the registered port. The server then waits for 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`), even if they reside in different programs. This is also an example of a running program that is dynamically extended with new code.

4.2 Process Management

Since Microsoft Windows does the preemptive scheduling of processes, our scheduler does not need any knowledge about multiple processes. Instead of changing the scheduler, we let our system automatically add an additional thread, called the *management thread*, to each process when it is created. This management thread is used to handle signals from other processes and to route them to the designated threads. On request from threads running at other processes, it also handles the creation of new threads inside its own process. This management thread, in combination with the scheduler and the port implementation, form the micro kernel that is included in each process.

```
:: ProcId               // abstract process id
:: Location :== String

newProc :: Location -> Thread ProcId
newThreadAt :: ProcId (Thread a) -> Thread ThreadId
```

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

Here is an example of starting a thread at a remote process and getting the result back to the parent.

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

remote :: ProcId (Thread a) -> Thread (Remote a) | TC a
remote pid thread = newPort >>= \port ->
```

```
                     newThreadAt pid (thread >>= writePort port) >>
                     return (Remote port)

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

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

The extension of our system with this kind of heavyweight process enables the programmer to build distributed concurrent applications. If one wants to run Clean programs that contain parallel algorithms on a farm of workstations, this is a first step. However, non-trivial changes are required to the original program to fully accomplish this. These changes include splitting the program code into separate threads and making communication between the threads explicit. The need for these changes is unfortunate, but our system was primarily designed for explicit distributed programs (and eventually mobile programs), not to speedup existing programs by running them on multiple processors.

This concludes our discussion of the micro kernel and its interface that provides 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 Interacting with Famke: The Shell

In this section we introduce our shell that enables programmers to construct new (concurrent) programs interactively.

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 concurrent 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 because 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 common operating systems 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. For example, it could test if a printing program (`:: WordDocument -> PostScript`) matches a document (`:: WordDocument`).

The shell supports function application, variables, and a subset of Clean's constant denotations. The shell syntax closely resembles Haskell's do-notation, extended with operations to read and write files.

Here follow some command line examples with an explanation of how they are handled by the shell.

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

The names `map` and `add` are unbound (do not appear in the left hand side of a let of lambda expression) in this example and our shell therefore assumes that they are names of files (dynamics on disk). All files are supposed to contain dynamics, which together represent a typed file system. The shell reads them in from disk, practically extending its functionality with these functions, and inspects the types of the dynamics. It uses the types of `map` (let us assume that the file `map` contains the type that we expect: `(a -> b) [a] -> [b]`), `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. The application of one dynamic to another is done using the `dynamicApply` function from Section 2, extended with better error reporting. With the help of the `dynamicApply` function, the shell constructs a new function that performs the computation `map (add 1) [1..10]`. This function uses the compiled code of `map`, `add`, and the list comprehension. Our shell is a hybrid interpreter/compiler, where the command line is interpreted/compiled to a function that is almost as efficient as the same function written directly in Clean and compiled to native code. Dynamics are read in before executing the command line, so it is not possible to change the meaning of a part of the command line by overwriting a dynamic.

```
> 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. The dynamic linker detects that `map` and `add` are already linked in, and reuses their code.

```
> 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 the usual type error: `Cannot unify [Int] with [Char]`.

```
> 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 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 final result of the command line is the contents of the variable `x`. In contrast to the `add` and `map` functions, which 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.

```
> newThread server;
> p <- lookupPort "MyDBase"; writePort p (insertDBase MyRecord)
```

The first line in the example above creates a new thread that executes the `server` from section 4.1. Let us assume that we have two dynamics on disk: one with the name `insertDBase` containing a function that can insert a record into a database, and one with the name `MyRecord` containing a record for the database. In the second line, we get the port of the server by looking it up using the name MyDBase. We send the function `insertDBase` applied to `MyRecord` to the server by writing the closure to the port. This example shows how we can interactively communicate with threads in a type safe way.

6 Related Work

There are concurrent versions of both Haskell and Clean. Concurrent Haskell [10] offers lightweight threads in a single UNIX process and provides M-Vars as the means of communication between threads. Concurrent Clean [7] is only available on multiprocessor Transputers and on a network of single-processor Apple Macintosh computers. Concurrent Clean provides support for native threads on 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 cannot easily provide type safety between different programs or between multiple incarnations of a single program.

Another difference between Famke and the concurrent versions of Haskell and Clean is the choice of communication primitives. Neither lazy graph copying nor M-Vars scale very well to a distributed setting because they require distributed garbage collection. This issue has led to a distributed version of Concurrent Haskell [13] that also uses ports. However, its implementation does not allow functions or closures to be sent over ports, because it cannot serialize functions. Support for this could be provided by a dynamic linker for Concurrent Haskell.

Both Cooper [14] and Lin [15] 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, none of them support the type-safe communication of any value between different computers.

Erlang [16] is a functional language specifically designed for the development of concurrent processes. It is completely dynamically typed and primarily uses interpreted byte-code, while Famke is mostly statically typed and executes native code generated by the Clean compiler. A simple spelling error in a token used during communication between two processes is often not detected by Erlang's dynamic type system, sometimes causing deadlock.

Back et al. [17] 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 lacks the power of polymorphic and higher-order functions and closures (to allow laziness) that our functional approach offers.

Haskell provides exception handling, while remaining pure and lazy. In [11] support for asynchronous exceptions has been added to Concurrent Haskell. Our implementation of signals closely follows their approach.

The Scheme Shell [18] integrates a shell into the programming language in order to enable the user to use the full expressiveness of Scheme. Es [19] 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 type safety because they operate on untyped executables.

7 Conclusions and Future Work

In this paper, we presented the basics of our prototype functional operating system called Famke. Famke is written entirely in Clean and provides lightweight threads, exceptions and heavyweight processes, and a type safe communication mechanism, using Clean's dynamic type system and dynamic linking support. Furthermore, we have built an interactive shell that type checks the command line before executing it. With the help of these mechanisms it becomes feasible to build distributed concurrent Clean programs running on a network. Programs can easily be extended with new code at run-time using the dynamic run-time system of Clean.

We can extend our kernel in a modular way by putting all extensions in separate dynamics, which would allow us to tailor our system (at run-time) to a given situation. Nevertheless, there remain issues that need further research.

We would like to give the programmer more information about what exceptions a function may throw. Unfortunately, we have not yet found a way to do this without compromising the flexibility of our approach.

The implementation of ports given in this paper does not check if the name is unique (when registering) or even exists (when looking up), entrusting this responsibility upon the programmer. Fortunately, this situation will be detected at run-time because it causes an exception at the receiving end. We intend to repair it in a more mature implementation.

The current focus of further research on Famke is to increase the power and usability of the shell.

References

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

2. M. J. Plasmeijer and M. C. J. D. van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison Wesley, 1993.
3. M. van Eekelen and R. Plasmeijer. *Concurrent CLEAN Language Report (version 2.0, draft)*. University of Nijmegen, December 2001. http://www.cs.kun.nl/~clean.
4. M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic Typing in a Statically Typed Language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.
5. 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 1595 of *Lecture Notes in Computer Science*, pages 171–188. Springer-Verlag, 1998.
6. M. Vervoort and R. Plasmeijer. Lazy Dynamic Input/Output in the Lazy Functional Language Clean. In R. Peña and T. Arts, editors, *Proceedings of the 14th International Workshop on the Implementation of Functional Languages*, Lecture Notes in Computer Science. Springer-Verlag, 2002.
7. E.G.J.M.H. Nocker, J.E.W. Smetsers, M.C.J.D. van Eekelen, and M.J. Plasmeijer. Concurrent Clean. In E.H.L. Aarts, J. van Leeuwen, and M. Rem, editors, *PARLE '91: Parallel Architectures and Languages Europe, Volume II*, volume 506 of *Lecture Notes in Computer Science*, pages 202–219. Springer, 1991.
8. M. 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.
9. K. Claessen. A Poor Man's Concurrency Monad. *Journal of Functional Programming*, 9, May 1999.
10. S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Conference Record of POPL '96: The* 23$^{\text{rd}}$ *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–308, St. Petersburg Beach, Florida, 21–24 1996.
11. S. Marlow, S.L. Peyton Jones, A. Moran, and J.H. Reppy. Asynchronous Exceptions in Haskell. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 274–285, 2001.
12. V. Stolz and F. Huch. Implementation of Port-based Distributed Haskell, 2001. http://www-i2.informatik.rwth-aachen.de/Research/distributedHaskell/ifl2001.ps.gz.
13. F. Huch and U. Norbisrath. Distributed Programming in Haskell with Ports. In M. Mohnen and P.W.M. Koopman, editors, *Implementation of Functional Languages, 12th International Workshop, IFL 2000*, volume 2011 of *Lecture Notes in Computer Science*, pages 107–121. Springer, September 2000.
14. E.C. Cooper and J.G. Morrisett. Adding Threads to Standard ML. Technical Report CMU-CS-90-186, Pittsburgh, PA, 1990.
15. A.C. Lin. *Implementing Concurrency For An ML-based Operating System*. PhD thesis, Massachusetts Institute of Technology, February 1998.
16. J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall, second edition, 1996.
17. G. Back, P. Wullmann, L. Stoller, W. C. Hsieh, and J. Lepreau. Java Operating Systems: Design and Implementation. Technical Report UUCS-98-015, 6, 1998.
18. O. Shivers. A Scheme Shell. Technical Report MIT/LCS/TR-635, 1994.
19. P. Haahr and B. Rakitzis. Es: A shell with higher-order functions. In *USENIX Winter*, pages 51–60, 1993.