owards a Strongly Typed nctional Operating System*

rjen van Weelden and Rinus Plasmeijer

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

this paper, we present Famke. It is a prototype imple-
strongly typed operating system written in Clean. Famke
ation and management of independent distributed Clean
network of workstations. It uses Clean's dynamic type
dynamic linker to communicate values of any type, e.g.
and functions (i.e. compiled code), between running ap-
type safe way. Mobile processes can be implemented using
to communicate functions. We have built an interactive
f Famke that enables user interaction. The shell uses a
e command language that allows construction of new pro-
ype checks the command line before executing it. Famke's
ime extensibility makes it a strongly typed operating sys-
e tailored to a given situation.

n

ming languages like Haskell [1] and Clean [2,3] offer a very
l static type system. Compact, reusable, and readable pro-
n in these languages while the static type system is able to
mming errors at compile time. But this works only within a

eveloped applications often need to communicate with each
e the communication of objects to take place in a type safe
not only simple objects, but objects of any type, including
e, this is not easy to realize: the compile time type informa-
kept inside a compiled executable, and therefore cannot be

is mainly because the used programming language has no
ngs.
e present a prototype implementation of a micro kernel,
nfunctional *m*icro *k*ernel *e*xperiment). It provides explicit
ncurrency and type safe message passing for all types to
Clean. By adding servers that provide common operating
entire strongly typed, distributed operating system can be
e.
a powerful dynamic type system [4] for this purpose and a
extend a running application with new code. Fortunately,
n offers some of the required basic facilities: it offers a hybrid
tic as well as dynamic typing (*dynamics*) [5], including run-
namic linking [6] (currently on Microsoft Windows only).
communication, Famke uses the above mentioned facilities
mplement *lightweight threads*, *processes*, *exception handling*
ge passing without requiring additional language constructs
.
e of an underlying operating system to avoid some low-
work and to integrate better with existing software (e.g.
e console and the file system). With Famke, we want to
ving objectives without changing the Clean compiler or run-

face (API) for Clean programmers with which it is easy to
ed) processes that can communicate expressions of any type
ay;
ctive shell with which it is easy to manage, apply and com-
) processes, and even construct new processes interactively.
type check the command line before executing it in order
arly;
lar design using an extensible micro kernel approach;
e system by using static types where possible and, if static
be done (e.g. between different programs), dynamic type

n that is easy to port to another operating system (if the
pports it).

n Clean

been extended with a polymorphic dynamic type system its static type system. Here, we will give a small introduction n. A dynamic is a value of type `Dynamic` which contains a presentation of the type of that value.

formed (i.e. lifted from the static to the dynamic type sys- ord `dynamic` in combination with the value and an optional compiler will infer the type), separated by a double colon.

```
g | Just a
```
[2]

```
: -> Maybe Int
  = Just x
  = Nothing
```

`ynamic` can be matched in function alternatives and case em from the dynamic back into the static type system. Such sist of an optional value pattern and a type pattern. In the `chInt` returns `Just` the value contained inside the dynamic nd `Nothing` if it has any other type. The compiler translates s into run-time type unifications. If the unification fails, the ative is tried, as in a common pattern match.

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

an contain type variables which, if the run-time unification nd to the offered type. In the example above, `dynamicApply` the function `f` inside its first argument can be unified with ue `x` inside the second argument. If this is the case then safely apply `f` to `x`. The result of this application has type t is generally unknown what this type `b` will be. The result a dynamic (and only a dynamic) again, because the type stantiated by the run-time unification.

```
namic -> Maybe t | TC t
```
[4]

variable in a pattern associates it with the type variable with
he static type of the function. The static type variable then
in the predefined `TC` (type code) class [5]. In the example
pe `t` will be determined by the static context in which it
npose a restriction on the actual type that is accepted at
ynamic`. It yields `Just` the value inside the dynamic (if the
value of the required context dependent type) or `Nothing`

c run-time system of Clean [6] supports writing dynamics to
em in again, possibly in another program or during another
he program.

```
ring Dynamic *World -> (Bool, *World)
ng *World -> (Bool, Dynamic, *World)
```
[5]

ll be read in lazily after a successful run-time unification
ern match on the dynamic). The amount of data and code
nker will link in, is therefore determined by the amount of
ue inside the dynamic. Dynamics written by a program can
ny other program, providing a form of persistence and a
of communication.

ean, as well as other functional languages, to construct new
ing and higher-order functions) in combination with Clean's
n-time linking, enables us to extend a running application
can be type checked after which it is guaranteed to fit.

Famke

a programmer can construct concurrent programs in Clean,
d management and exception handling primitives.
offers only very limited library support for process manage-
cation.
oncurrent Clean [7] did offer sophisticated support for paral-
ghtweight processes, but no support for exception handling.
was targeted at deterministic, implicit concurrency, but we
em for distributed, non-deterministic, explicit concurrency.

y heavyweight, and it would prevent them from sharing the still would not have exception handling.

e does her own scheduling of threads in order to keep them rovide exception handling.

lementation

ent cooperative threads we need a way to suspend running o resume them later. Wand [8] shows that this can be done and the call/CC construct offered by Scheme and other func- languages. We copy this approach using first class continu- cause Clean has no call/CC construction, we have to write ssing explicitly. Our approach closely resembles Claessen's [9], but our primitives operate directly on the kernel state eness typing, and we have extended the implementation with ception handling (see section 3.2).

```
 -> KernelOp) -> KernelOp[6]
rnel -> Kernel

hread a
ont kernel -> cont x kernel`

         //[7] calculate argument for cont
nel... // operate on the kernel state
```

e type `Thread`, such as the example function above, gets the on (named `cont`; of type `a -> KernelOp`) as its argument with a new computation step, which calculates the argument il computation, to form a new function (of type `KernelOp`). s, when evaluated on a kernel state (named `kernel`; of type el state.

```
stract thread id

rentId  :: ThreadId, newId :: ThreadId,
dy :: [ThreadState], world :: *World}

thrId :: ThreadId, thrCont :: KernelOp}
```

ady list), and the world state which is provided by the Clean
ean's uniqueness type system makes these types a little more
will not show this in the examples in order to keep them

```
d a) -> Thread ThreadId
 \cont k=:{newId, ready}^9 ->
ewId = inc newId, ready = [threadState:ready]}^10

rId = newId, thrCont = thread (\_ k -> k)}

 ThreadId
:{currentId} -> cont currentId k
```

function starts the given thread concurrently with the other
e evaluated for their effect on the kernel and the world state.
ot return a result, hence the polymorphically parameterized
elieves our system from the additional complexity of return-
 parent thread. The communication primitives that will be
ble programmers to extend the newThread primitive to de-
parent. Threads can obtain their thread identification with

 threads is done cooperatively. This means that threads must
escheduling using yield, and should not run endless tight
 function then evaluates the next ready thread. StartFamke
 standard Clean Start function to start the evaluation of

```
ernel -> Kernel
entId, ready} = {k & ready = ready ++ [threadState]}

rId = currentId, thrCont = cont}

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

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

is currently being evaluated returns directly to the scheduler
s a `yield` action, because `yield` does not evaluate the tail
Instead, it stores the continuation at the back of the ready
und-robin scheduling) and returns the current kernel state.
uses this new kernel state to evaluate the next ready thread.
reads using a continuation style is cumbersome, because one
inuation along and one has to perform an explicit yield often.
l thread-combinators resembling a more common monadic
Our `return`, `>>=` and `>>` functions resemble the monadic
functions of Haskell[11]. Whenever a running thread performs
ch as a return, control is voluntarily given to the scheduler

```
ead a
 -> yield (cont x) k

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

_ -> r

= newThread (print ['h', 'e', 'l', 'l', 'o']) >>
  print ['w', 'o', 'r', 'l', 'd']

 return Void
 printChar c >> print cs
```

`Example` above starts a thread that prints "`hello`" concur-
thread that prints "`world`". It assumes a low-level print
hat prints a single character. The output of both threads is
cheduler, and is printed as "`hweolrllod`".

and Signals

e.g. `newThread`) may fail because of external conditions such
other threads or operating system errors. Robust programs

nented using dynamics. This makes it possible to store any n and to easily extend the set of exceptions at compile-time To provide this kind of exception handling, we extend the continuation argument for the case that an exception is

```
ucCnt a) -> ExcCnt -> KernelOp
-> ExcCnt -> KernelOp
ception -> KernelOp

ynamic

ad a | TC e
 -> ec (dynamic e :: e^) k

on -> Thread a
= \sc ec k -> ec exception k

(Exception -> Thread a) -> Thread a
 =
ad (\x _ -> sc x ec) (\e -> catcher e sc ec) k
```

sion wraps a value in a dynamic (hence the TC context re-s it to the enclosing try clause by evaluating the exception ethrow can be used to throw an exception without wrapping in. The try function catches exceptions that occur during first argument (thread) and feeds it to its second argument any value can be thrown, exception handlers must match the exception using dynamic type pattern matching.

des an outermost exception handler (not shown here) that hen an exception remains uncaught. This exception handler mer that an exception was not caught by any of the handlers of the occurring exception.

```
ead a
x -> yield (sc x ec) k

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

```
try (divide 42 0) handler

DivByZero
n (x / y)

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

tion in the example throws the value DivByZero as an excep-
ammer tries to divide by zero. Exceptions caught in the body
e handled by handler, which returns zero on a DivByZero
xceptions of any other type are thrown again outside the try,

or concurrent setting, there is also a need for throwing and
between different threads. We call this kind of inter-thread
signals allow threads to throw kill requests to other threads.
gnals, or *asynchronous exceptions* as they are also called,
s described by Marlow et. al. in an extension of Concurrent
nmarize our interface for signals below.

```
l e -> Thread Void | TC e
ad a) -> Thread a
ead a) -> Thread a
```

ferred from one thread to the other by the scheduler. A signal
on again when it arrives at the designated thread, and can
n the same way as other exceptions. To prevent interruption
an enclose operations in a signalsOff clause, during which
ntil they can interrupt. Regardless of any nesting, signalsOn
uptible and signalsOff always means non-interruptible. It
clear whether program code can or cannot be interrupted.
nposition and nesting of program fragments that use these
signal is caught, control goes to the exception handler and

n Famke

vill show how a programmer can execute groups of threads
nultiple workstations, to construct distributed programs in

osoft Windows processes to provide preemptive task switch-
of threads running inside different processes. Once processes
n one or more computers, threads can be started in any one
troduce Famke's message passing primitives for communica-
s and processes. The dynamic linker plays an essential role
of a thread from one process to another.

Thread Communication

pe-safe communication between threads are Concurrent Has-
nd Concurrent Clean's lazy graph copying [7].
I-Vars do not scale very well to a distributed setting because
scribed by Stolz and Huch in [12]. The first problem is that
ibuted garbage collection because they are first class objects,
stributed or mobile setting. The second problem is that the
ar is generally unknown, which complicates reasoning about
of failing or moving processes. Automatic lazy graph copying
work on objects that are distributed over multiple (remote)
om the same two problems.
kell [13,12] solves the problem by implementing an asyn-
passing system using ports. Famke uses the same kind of
ke are channels that vanish as soon as they are closed by
he process containing the creating thread dies. Accessing a
n an exception. Using ports as the means of communication,
here a port resides (at the process of the creating thread)
d (explicitly or because the process died). In contrast with
we do not limit ports to a single reader (which could be
ime using Clean's uniqueness typing). The single reader re-
s that the port vanishes when the reader vanishes but we
e in practice.

```
abstract port id
```

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

rtId msg) String -> Thread Void | TC msg
ring -> Thread (PortId msg)      | TC msg

 Dynamic -> String
 String -> (Bool, Dynamic)
```

n ports operate on typed messages. The `newPort` function and `closePort` removes a port. `writePort` and `readPort` and receive messages. The dynamic run-time system is used ges to and from a dynamic. Because we do not want to read time we want to send a message to someone, we will use the `String` and `stringToDynamic` functions from the dynamic ary. These functions are similar to Haskell's `show` and `read`, (de)serialize functions and closures. They should be handled hey allow you to distinguish between objects that should be g. between a closure and its value). The actual sending and rings is done via simple message (string) passing primitives perating system. The `registerPort` function associates a port, by which the port can be looked up using `lookupPort`. buted Haskell and Famke both use ports, our system is cand receiving functions (and therefore also closures) using ker. The dynamic type system also allows programs to re- of type `(PortId Dynamic)`, previously unknown data struc- used by polymorphic functions or functions that work on he `dynamicApply` functions in section 2. An asynchronous tem, such as presented here, allows programmers to build n and synchronization methods (e.g. remote procedure calls, nnels).

n example of a database server that uses a port to receive ts and applies them to the database.

```
 list of records or something like that
```

```
: "MyDBase" >>= \port ->
 port mutateDatabase

 DBase -> DBase
) = ...  // change the database
```

tes, and registers, a port that receives functions of type lients send functions that perform changes to the database rt. The server then waits for functions to arrive and applies e db. These functions can be safely applied to the database c run-time system guarantees that both the server and the e notion of the type of the database (DBase), even if they ograms. This is also an example of a running program that nded with new code.

nagement

dows does the preemptive scheduling of processes, our schedny knowledge about multiple processes. Instead of changing our system automatically add an additional thread, called *read*, to each process when it is created. This management ndle signals from other processes and to route them to the On request from threads running at other processes, it also of new threads inside its own process. This management ion with the scheduler and the port implementation, form t is included in each process.

```
      // abstract process id
ring

n -> Thread ProcId
cId (Thread a) -> Thread ThreadId
```

nction creates a new process at a given location and re- The creation of a new process is implemented by starting n executable, the *loader*, which becomes the new process. ple Clean program that starts a management thread. The on starts a new thread in another process. The thread is

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

-> Thread a | TC a
= readPort port >>= \result ->
  closePort port >>
  return result
```

ction creates a port to which the result of the given thread
en starts a child thread at the remote location `pid` that
and writes it to the port, and returns the port enclosed in
e parent. When the parent decides that it wants the result,
et it and to close the port.

f our system with this kind of heavyweight process enables
build distributed concurrent applications. If one wants to run
t contain parallel algorithms on a farm of workstations, this
ver, non-trivial changes are required to the original program
this. These changes include splitting the program code into
d making communication between the threads explicit. The
es is unfortunate, but our system was primarily designed for
programs (and eventually mobile programs), not to speedup
y running them on multiple processors.

ur discussion of the micro kernel and its interface that pro-
reads (with exceptions and signals), processes and type-safe
lues of any type between them. Now it is time to present the
t makes use of these strongly typed concurrency primitives.

with Famke: The Shell

ntroduce our shell that enables programmers to construct
ograms interactively.

a way to interact with an operating system, usually via a
ne/console interface. Normally, a shell does not provide a
ing language, but it does enable users to start pre-compiled
most shells provide simple ways to combine multiple pro-
g and concurrent execution, and support execution-flow con-

mple, it could test if a printing program (`:: WordDocument`
tches a document (`:: WordDocument`).
ts function application, variables, and a subset of Clean's
s. The shell syntax closely resembles Haskell's do-notation,
tions to read and write files.
command line examples with an explanation of how they
hell.

```
0]
```

and `add` are unbound (do not appear in the left hand side
xpression) in this example and our shell therefore assumes
of files (dynamics on disk). All files are supposed to contain
gether represent a typed file system. The shell reads them
cally extending its functionality with these functions, and
the dynamics. It uses the types of `map` (let us assume that
the type that we expect: `(a -> b) [a] -> [b]`), `add` (let us
`Int`) and the list comprehension (which has type: `[Int]`)
mmand line. If this succeeds, which it should given the types
lies the partial application of `add` with the integer one to
rom one to ten, using the `map` function. The application of
ther is done using the `dynamicApply` function from Section
etter error reporting. With the help of the `dynamicApply`
nstructs a new function that performs the computation `map`
This function uses the compiled code of `map`, `add`, and the list
shell is a hybrid interpreter/compiler, where the command
ompiled to a function that is almost as efficient as the same
ectly in Clean and compiled to native code. Dynamics are
ting the command line, so it is not possible to change the
the command line by overwriting a dynamic.

```
inc [2,4..10]
```

e with the name `inc` as the partial application of the `add`
er one. Then it applies the `map` function using the variable
n integers from two to ten. The dynamic linker detects that
ady linked in, and reuses their code.

```
inc ['a'..'z']
```

ble `inc` as in the previous example, but applies it, using the functions, which are read from disk by the shell before type ng the command line, `result` is read in during the execution e.

```
"MyDBase"; writePort p (insertDBase MyRecord)
```

the example above creates a new thread that executes the n 4.1. Let us assume that we have two dynamics on disk: insertDBase containing a function that can insert a record d one with the name MyRecord containing a record for the ond line, we get the port of the server by looking it up using We send the function insertDBase applied to MyRecord to g the closure to the port. This example shows how we can nicate with threads in a type safe way.

rk

nt versions of both Haskell and Clean. Concurrent Haskell nt threads in a single UNIX process and provides M-Vars as nication between threads. Concurrent Clean [7] is only available sor Transputers and on a network of single-processor Apple rs. Concurrent Clean provides support for native threads ems. On a network of Apple computers, it ran the same each processor, providing a virtual multiprocessor system. rovided lazy graph copying as the primary communication ncurrent systems cannot easily provide type safety between r between multiple incarnations of a single program.

ce between Famke and the concurrent versions of Haskell and f communication primitives. Neither lazy graph copying nor ell to a distributed setting because they require distributed This issue has led to a distributed version of Concurrent so uses ports. However, its implementation does not allow s to be sent over ports, because it cannot serialize functions. ld be provided by a dynamic linker for Concurrent Haskell. and Lin [15] have extended Standard ML with threads (imbuilt two prototypes of a Java operating system. Although 's extensibility, portable byte code and static/dynamic type ay to build an operating system where multiple Java pro- n concurrently, Java lacks the power of polymorphic and ns and closures (to allow laziness) that our functional ap-

exception handling, while remaining pure and lazy. In [11] onous exceptions has been added to Concurrent Haskell. Our ignals closely follows their approach.

ll [18] integrates a shell into the programming language in user to use the full expressiveness of Scheme. Es [19] is a higher-order functions and allows the user to construct new mmand line. Neither shell provides a way to read and write and to disk, and they cannot provide type safety because yped executables.

s and Future Work

resented the basics of our prototype functional operating . Famke is written entirely in Clean and provides lightweight and heavyweight processes, and a type safe communication lean's dynamic type system and dynamic linking support. ve built an interactive shell that type checks the command g it. With the help of these mechanisms it becomes feasible concurrent Clean programs running on a network. Programs led with new code at run-time using the dynamic run-time

our kernel in a modular way by putting all extensions in which would allow us to tailor our system (at run-time) to evertheless, there remain issues that need further research.

give the programmer more information about what excep- y throw. Unfortunately, we have not yet found a way to do omising the flexibility of our approach.

tion of ports given in this paper does not check if the name istering) or even exists (when looking up), entrusting this he programmer. Fortunately, this situation will be detected

nd M. C. J. D. van Eekelen. *Functional Programming and Parallel* Addison Wesley, 1993.
nd R. Plasmeijer. *Concurrent CLEAN Language Report (version* sity of Nijmegen, December 2001. http://www.cs.kun.nl/˜clean.
delli, B. Pierce, and G. Plotkin. Dynamic Typing in a Statically *ACM Transactions on Programming Languages and Systems*, ril 1991.
Types and Type Dependent Functions. In T. Davie K. Hammond tors, *Proceedings of the 10th International Workshop on the Im-unctional Languages*, volume 1595 of *Lecture Notes in Computer* 1–188. Springer-Verlag, 1998.
R. Plasmeijer. Lazy Dynamic Input/Output in the Lazy Func-Clean. In R. Peña and T. Arts, editors, *Proceedings of the 14th rkshop on the Implementation of Functional Languages*, Lecture er Science. Springer-Verlag, 2002.
er, J.E.W. Smetsers, M.C.J.D. van Eekelen, and M.J. Plasmeijer. In E.H.L. Aarts, J. van Leeuwen, and M. Rem, editors, *PARLE itectures and Languages Europe, Volume II*, volume 506 of *Lecture er Science*, pages 202–219. Springer, 1991.
uation-Based Multiprocessing. In J. Allen, editor, *Conference 0 LISP Conference*, pages 19–28, Palo Alto, CA, 1980. The Lisp
oor Man's Concurrency Monad. *Journal of Functional Program-*9.
A. Gordon, and S. Finne. Concurrent Haskell. In *Conference 96: The* 23rd *ACM SIGPLAN-SIGACT Symposium on Principles Languages*, pages 295–308, St. Petersburg Beach, Florida, 21–24
Peyton Jones, A. Moran, and J.H. Reppy. Asynchronous Excep-n *SIGPLAN Conference on Programming Language Design and* pages 274–285, 2001.
uch. Implementation of Port-based Distributed Haskell, 2001. formatik.rwth-aachen.de/Research/distributedHaskell/
orbisrath. Distributed Programming in Haskell with Ports. In M. M. Koopman, editors, *Implementation of Functional Languages, Workshop, IFL 2000*, volume 2011 of *Lecture Notes in Computer* 7–121. Springer, September 2000.