Introduction to Concurrent ML

CML consists of a sequential core language—Standard ML—extended with concurrency primitives.

Introduction to Concurrent ML

CML consists of a sequential core language—Standard ML—extended with concurrency primitives.

Although CML is thought of as a *language*, it's actually a collection of SML/NJ *structures*, the main one being `CML`.

Introduction to Concurrent ML

CML consists of a sequential core language—Standard ML—extended with concurrency primitives.

Although CML is thought of as a *language*, it's actually a collection of SML/NJ *structures*, the main one being `CML`.

CML is a *message-passing* language, even though threads are capable of *sharing storage*. As a result, CML programming seems to be mostly functional.

Threads

CML processes are very lightweight, and are called *threads* to emphasize this.

Initially, a single thread is created using the function

```
doit : (unit -> unit) * Time.time option -> OS.Process.status
```

of the `RunCML` structure.

The main thread evaluates the expression `f()`, where f is the first argument to `doit`. The second parameter to `doit` specifies the scheduling time slice; `NONE` means the default of 20 milliseconds.

Threads

CML processes are very lightweight, and are called *threads* to emphasize this.

Initially, a single thread is created using the function

```
doit : (unit -> unit) * Time.time option -> OS.Process.status
```

of the `RunCML` structure.

The main thread evaluates the expression `f()`, where f is the first argument to `doit`. The second parameter to `doit` specifies the scheduling time slice; `NONE` means the default of 20 milliseconds.

A CML program is shut down, causing a return of the call to `RunCML.doit` with a specified termination status, using the function

```
shutdown : OS.Process.status -> 'a
```

of the `RunCML` structure. If all threads terminate, or become permanently blocked (so that global deadlock has occurred), then the call to `RunCML.doit` will return with failure status.

Threads (Cont.)

Further threads may be created, and the running threads may be caused to terminate, using the functions

```
val spawn : (unit -> unit) -> thread_id
val exit  : unit -> 'a
```

of the `CML` structure. Threads also terminate when their functions return or when they raise uncaught exceptions. (Exceptions don't propagate from children to parents.)

Threads (Cont.)

Further threads may be created, and the running threads may be caused to terminate, using the functions

```
val spawn : (unit -> unit) -> thread_id
val exit  : unit -> 'a
```

of the CML structure. Threads also terminate when their functions return or when they raise uncaught exceptions. (Exceptions don't propagate from children to parents.)

Because CML threads are implemented using first-class continuations, they will be garbage collected when they are permanently blocked, e.g., waiting to receive or send on a channel that no other thread has access to. As a result, it's typically not necessary to develop a protocol for asking a thread to exit.

Channels

Communication and synchronization in CML is performed by synchronous message passing on typed channels.

The `CML` structure defines a type constructor

```
type 'a chan
```

along with the functions

```
val channel : unit -> 'a chan
val recv    : 'a chan -> 'a
val send    : 'a chan * 'a -> unit
```

for creating typed channels, receiving values from channels, and sending values on channels.

Channels

Communication and synchronization in CML is performed by synchronous message passing on typed channels.

The `CML` structure defines a type constructor

```
type 'a chan
```

along with the functions

```
val channel : unit -> 'a chan
val recv    : 'a chan -> 'a
val send    : 'a chan * 'a -> unit
```

for creating typed channels, receiving values from channels, and sending values on channels.

When a receive and a send are being offered by two threads on the same channel, CML may match these two offerings, transferring the value of the sender to the receiver. This involves both communication of data and synchronization.

More on Message Passing

Asynchronous communication isn't built-in, but can be programmed out of synchronous communication:

- by implementing buffers; or
- by spawning threads that wait to send messages (this doesn't deliver messages in order).

More on Message Passing

Asynchronous communication isn't built-in, but can be programmed out of synchronous communication:

- by implementing buffers; or
- by spawning threads that wait to send messages (this doesn't deliver messages in order).

Part of CML's power comes from the fact that channels may be passed over channels. Threads are not data, and so can't be passed on channels. But the same effect can be obtained by communicating the channels needed to interact with threads.

More on Message Passing

Asynchronous communication isn't built-in, but can be programmed out of synchronous communication:

- by implementing buffers; or
- by spawning threads that wait to send messages (this doesn't deliver messages in order).

Part of CML's power comes from the fact that channels may be passed over channels. Threads are not data, and so can't be passed on channels. But the same effect can be obtained by communicating the channels needed to interact with threads.

When a resource needs to be shared among multiple processes, a server thread is created to manage it. Client threads communicate with this server thread by message passing. When desired, the communication between clients and servers can be made to look like ordinary function calls.