ESL Reference Guide

Contents

1. Introduction

This document describes the language ESL and its implementation. ESL is an actor-based language that has been designed to offer a convenient way to build applications that benefit from a large number of independent concurrent processes that communicate in terms of structured data. ESL compiles directly to Java source code and can therefore easily integrate with Java applications. The benefit of using ESL compared to Java is the abstraction that it provides over the construction and management of processes and the data that is passed between them. In addition, ESL enforces a separation of concerns since communication is by asynchronous message which reduces the risks associated with large scale concurrency due to race conditions and shared data. Having said that, whilst ESL actors are encapsulated, it is unreasonable to build a large scale system without some degree of shared data, so ESL provides mechanisms for data sharing where needed. ESL aims to support system verification through an expressive statically checked type system.

The Actor Model of Computation is characterised in figure 1 which shows three actors. Each actor has a local thread of control which selects a message from the head of the mailbox and processes it. When the current message is processed, the thread inspects the mailbox for the next message; if no message

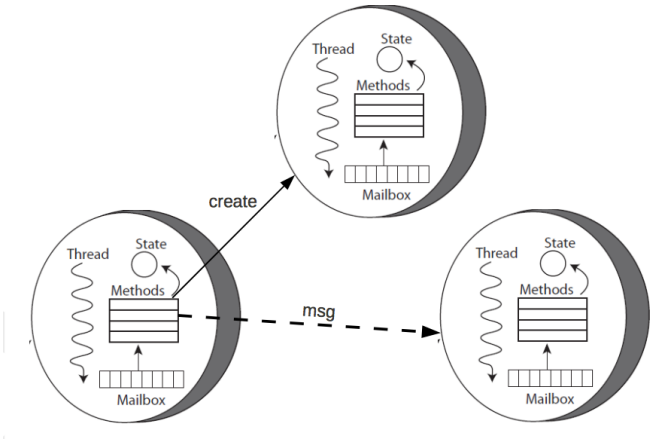


Figure 1: Actor Model of Computation [1]

is available then the actor becomes idle until a message is received. Each actor has a local state that can be inspected and updated by the methods that handle the messages.

An actor can send messages to any actor that it knows about. An actor will know about an actor if it created the actor or it received the actor as part of a message. A message may contain basic data items, collections of data item, actors, and functions. When a source actor passes a function to a target actor, the source may provide access to its internal state when the target calls the function on its own thread.

2. Data

ESL provides basic data types: integers, floats, booleans and strings. For example the following listing declares identifiers along with their values. Comments follow the Java format:

```
1 /* The parameters of the program are defined below.
2    Change the starting position to get a different
3    configuration. */
4
5 x::Int          = 100;      // Starting x position.
6 y::Int          = 200;      // Starting y position.
7 pi::Float       = 3.14;     // Used for position calculation.
8 isActive::Bool  = true;     // True when moving.
9 name::Str       = 'Wilma'; // Used as a unique identifier.
```

ESL supports homogeneous arrays, for example the following is a one- dimensional array of strings with 100 elements indexed from 0:

```
1 names::Array[Str] = new Array[Str](100);
```

```
names[0] := 'Wilma';
print[Str]('value at 0 = ' + names[0]);
```

Arrays may be nested, for example to create a two-dimensional array of booleans:

```
points::Array[Array[Bool]] = new Array[Array[Bool]](100);
for x::Int in 0..100 do {
  points[x] := new Array[Bool](200);
  for y::Int in 0..200 do {
    points[x][y] := true;
  }
}
```

Note that array elements can be updated and passed between actors, therefore they are a basis for race conditions and other concurrency problems. The intention is that arrays are used locally within an actor as efficient indexable storage. If arrays are shared then access should be protected using locks.

ESL provides homogeneous lists whose elements cannot be modified by side effect and are therefore safe to pass between actors. A list is constructed using `::` and `[ ... ]` and can be decomposed using the operations `head` and `tail` in addition to pattern matching (see below). For example:

```
nums::[Int] = [1,2,3,4,5,6,7,8,9,0];
print[Int](head[Int](nums));
print[[Int](tail[Int](nums))
print[[Int]](head[Int](nums) : tail[Int](tail[Int](nums));
```

which produces `1`, `[2,3,4,5,6,7,8,9,0]` and `[1,3,4,5,6,7,8,9,0]` respectively. Note that the operations `head` and `tail` are supplied with the type of the list element; this is because there operations are *polymorphic* as described in section 13. The definition of `nums` above is equivalent to:

```
nums::[Int] = 1:2:3:4:5:6:7:8:9:0:[];
```

Sets and bags are like lists except the elements are not ordered. Adding an element to a set or bag produces a new set or bag respectively. Adding an element to a set has no effect if the set already contains the element, otherwise both the set and bag are extended:

```
setOfNames::Set{Str} = Set{'Fred','Wilma'};
bagOfNames::Bag{Str} = Bag{'Fred','Wilma'};
print[Set{Str}](Set{'Fred' | setOfNames});
print[Bag{Str}](Bag{'Fred' | setOfNames});
```

produces `Set{'Fred','Wilma'}` and `Bag{'Fred','Wilma','Fred'}` respectively. Note that extraction from sets and bags occurs via pattern matching is described below. Sets require a definition of equality for ESL values which is provided in figure 2.

ESL hash tables map keys to values. A hash table provides operations for adding a mapping, accessing the value for a key, testing whether a mapping exists. A hash table uses field reference to get the list of keys and the list of values:

```
ages::Hash[Str,Int] = new Hash[Str,Int];
ages.put('Fred',35);
ages.put('Wilma',32);
print[Bool](ages.hasKey('Fred'));
print[Int](ages.get('Wilma'));
print[[Str]](ages.keys);
print[[Int]](ages.vals);
```

prints `true`, `32`, `[Fred,Wilma]`, and `[35,32]`. Note that because hash tables are updated by side effect, they are not thread safe and should be used locally within an actor, or access should be protected using locks.

ESL functions are first-class values and may be passed between actors within data structures. The rules for identifier binding within functions are defined in section 8. A simple example of a function is:

```
add1::(Int)→Int = fun(x::Int)::Int x + x;
print[Int](add1(100));
```

which prints `101`.

Records map names to values. Unlike hash tables, the domain of a record is fixed. New type names can be defined that refer to existing types, so the following code implements an two-dimensional point object:

```
type Point = { x::Int; y::Int }
mkPoint(n::Int,m::Int)::Point = {x→n; y→m}
print[Point](mkPoint(100,200));
print[Int](mkPoint(100,200).x);
```

prints `{x=100,y=200}` and `100`. Records cannot be updated by side effect.

The value `null` can be used as an undefined value for any data type. It is polymorphic and is followed by the type for which it acts as undefined. For example:

```
x::Int = null[Int];
p::Point = null[Point];
```

The following operations are builtin to ESL:

```
_ + _      :: (Int,Int)        → Int
_ + _      :: (Float,Float)    → Float
_ + _      :: (Str,T)          → Str
_ + _      :: (T,Str)          → Str
_ + _      :: ([T],[T])        → [T]
_ + _      :: (Set{T},Set{T})  → Set{T}
_ - _      :: (Int,Int)        → Int
_ - _      :: (Float,Float)    → Float
_ * _      :: (Int,Int)        → Int
_ * _      :: (Float,Float)    → Float
_ / _      :: (Int,Int)        → Int
_ / _      :: (Float,Float)    → Float
_ % _      :: (Int,Int)        → Int
_ = _      :: (T,T)            → Bool
_ <> _     :: (T,T)            → Bool
_ and _    :: (Bool,Bool)      → Bool
_ or _     :: (Bool,Bool)      → Bool
not        :: (Bool)           → Bool
print[T]   :: (T)              → Void
random     :: (Int)            → Int
wait       :: (Int)            → Void
isqrt      :: (Int,Int)        → Float
```

3. Blocks and Commands

The type `Void`, like Java, represents an evaluation that does not return a result. The operation `print[T](t)` takes a value `t::T` and 'returns' `Void`. A *block* is a sequence of *commands* enclosed in curly brackets. A command is just any expression, but syntactically, if the expression does not end with a closing `}`, then

Type	Equality for Values of Type
i1,i2::Int	i1 and i2 are the same number
f1,f2::Float	f1 and f2 are the same number
s1,s2::Str	s1 and s2 have the same characters in order
b1,b2::Bool	b1 and b2 are the same boolean values
h1:l1,h2:l2::[T]	h1 is equal to h2, and l1 is equal to l2
[]::[T]	equal to an empty list of the same type.
set{e1::T | s1::Set{T}} set{e2::T | s2::Set{T}}	e1 and e2 are equal, and s1 and s2 are equal
set{}::Set{T}	is only equal to an empty set of the same type.
bag{e1::T | b1::Bag{T}} bag{e2::T | b2::Bag{T}}	e1 and e2 are equal, and b1 and b2 are equal
bag{}::Bag{T}	is only equal to an empty bag of the same type.
a1,a2::Array[T]	a1 and a2 are the same array
t1,t2::Hash[K,V]	t1 and t2 are the same table
a1,a2::Act {...}	a1 and a2 are the same actor
f1,f2::(T ...) → T	f1 and f2 are the same function
r1,r2::{ n→T ... }	r1.n is equal to r2.n for all names
C(n1,...), C(m2,...)	equal when corresponding ni and mi are equal
null[T]	only equal to the value null

Figure 2: Data Equality

the command must be terminated with a semicolon. The return value of a block is the return value of the last command. For example:

```
print3(a::Int,b::Int,c::Int)::Void = {
  print[Int]('first = ' + a);
  print[Int]('second = ' + b);
  print[Int]('third = ' + b);
}
```

The special case of the block {} is of type Void.

4. List Comprehensions and List Operations

Lists are a very useful way of organising collections of elements. They can easily be processed using recursion because of their structure: a list is either empty [] or is a list l with a head element e 'consed' to the tail: e:l.

A *list comprehension* is an expression that transforms lists. It has the form: [e | q ...] where e is an expression and q is a *qualifier*. The idea is that a list is constructed whose elements are formed by evaluating e after performing each of the qualifiers in turn. There are two types of qualifier: *binding* and *predicate*. A binding qualifier has the form p ← e where p is a pattern and e is an expression whose value is a list. When it is evaluated, a binding qualifier repeatedly matches the elements from the list against the pattern. A predicate qualifier has the form ?e where e is a boolean valued expression. When it is evaluated, if the expression in a predicate qualifier produces true then execution continues, otherwise it ignores the element most recently selected by a binding qualifier and selects the next element. For example:

```
[ n | n ← 0..100, ?(n % 2 = 0) ]
```

produces a list of even numbers between 0 and 99 inclusive. Multiple binding qualifiers may be used:

```
[ mkPoint(x,y) | x ← 0..100, y ← 0..200, ?(x <> y) ]
```

Lists can be processed using recursive functions and pattern matching. Typically a case-expression is used to match against a base case (often []) and a recursion case (often h:t). For example the following operation adds up the distances of all points to the origin:

```
addDists(ps::[Point])::Float =
  case ps {
    [] → 0.0;
    h::t → distanceToOrigin(h) + addDists(t);
  }
```

The following defines Quicksort in ESL:

```
qsort(l::[Int])::[Int] =
  case l {
    [] →[];
    x:l →
      qsort([ n | n ← l, ?(n<x) ])
      + [x] +
      qsort([ n | n ← l, ?(n>x) ]);
  }
```

The collection of list operations that are supported by ESL are shown in B.

5. Algebraic Data Types and Pattern Matching

ESL supports algebraic data types, also known as *union types*. For example, the directions of the compass can be represented using different integers or strings, but it is better to define a new type Direction and to have 4 unique values of that type. This can be defined as follows:

```
data Direction =
  North
| South
| East
```

```
5 | West;
```

Now any identifier defined to have type `Direction` can hold one of the 4 values and the special value `null[Direction]`:

```
1 d::Direction = North;
```

Pattern matching can be used with such a data type:

```
1 moveRight(dir::Direction)::Direction =
2   case dir {
3     North → East;
4     East  → South;
5     South → West;
6     West  → North;
7   }
```

Given a list of such directions, they can be mapped:

```
1 dirs::[Direction] = [ North, South, East, West ];
2 [ moveRight(d) | d ← dirs ]
```

The element types of a union can have components. For example a tree of integers:

```
1 data TreeOfInt =
2   Branch(TreeOfInt,TreeOfInt)
3 | Leaf(Int);
```

Note that a branch term contains two integer trees and a leaf contains an integer. The integers in the tree can be added together:

```
1 addTreeOfInt(t::TreeOfInt)::Int =
2   case t {
3     Branch(l,r) → addTreeOfInt(l) + addTreeOfInt(r);
4     Leaf(n)     → n;
5   }
6 print[Int](addTreeOfInt(Branch(Leaf(100),Leaf(200))));
```

Prints `300`. When pattern matching, the case arms are tried in turn, so we can define a short-cut version:

```
1 addTreeOfInt(t::TreeOfInt)::Int =
2   case t {
3     Branch(Leaf(0),t) → addTreeOfInt(t);
4     Branch(t,Leaf(0)) → addTreeOfInt(t);
5     Branch(l,r)       → addTreeOfInt(l) + addTreeOfInt(r);
6     Leaf(n)           → n;
7   }
```

Case arms may contain conditions. For example, adding up all those elements of a tree that satisfy a predicate:

```
1 addIf(p::(Int)→Bool,t::TreeOfInt)::Int =
2   case t {
3     Branch(l,r)       → addIf(p,l) + addIf(p,r);
4     Leaf(n) when p(n) → n;
5     Leaf(n)           → 0;
6   }
7 addIf(fun(n::Int)::Bool n > 100,Branch(Leaf(100),Leaf(200)));
```

Prints `200`.

6. Sets and Collection Matching

ESL supports homogenous sets: the empty set `set{}`; a set constructed from an element `e` and a set `s`, `set{e | s}`; a set union `s1 + s2`. Both sets and lists can be processed using pattern matching. For example:

```
1 setContainsInt(s::Set{Int},x::Int)::Bool =
2   case s {
3     set{y | z} when x = y → true;
4     s                     → false;
5   }
```

The operation `setContainsInt` uses pattern matching to select the element `y` which is equal to the argument `x`. Note that sets are unordered, so the pattern `set{y | z}` is *non-deterministic*. When the supplied value `s` matches the pattern, one of the elements in the set is matched against `y`; if the pattern fails to match before the corresponding → then a different element `y` will be chosen. In the example, the condition `x = y` is within the scope fo the choice, this causes elements from the set `s` to be tried un turn until one that matches `x` is found or the set is exhausted. If all the values have been tried from the set `s` then the case-arm fails and the next case-arm is tried.

Non-deterministic choice may occur more than once in the same pattern. For example, the following returns a duplicate entry in two sets:

```
1 duplicate(s1::Set{Int},s2::Set{Int},notFound::Int)::Int =
2   case s1,s2 {
3     set{x | p}, set{y | q} when x = y → x;
4     s1,s2                             → notFound;
5   }
```

The first case-arm in `duplicate` selects an element `x` from `s1`, then selects `y` from `s2` such that they are equal elements. If there is no element in common then the `notFound` element is returned.

Non-deterministic pattern-based choice can be used over lists using the pattern `p1 + [p2] + p3` where `p1` matches a prefix of a list, `p2` matches some element, and `p3` matches a suffix. The pattern fails in the case that the list if empty. For example, testing whether a given element occurs in a list:

```
1 intMember(n::Int,l::[Int])::Bool =
2   case l {
3     l1 + [x] + l2 when x = n → true;
4     l                        → false;
5   }
```

7. Actors

Actors are implemented in terms of *behaviour types* and *behaviours*. A behaviour type provides an interface definition for a set of behaviours. An interface consists of message types and exported identifiers and their types. Consider a simple behaviour type called `Sink` that accepts a single type of message `M(Int)`:

```
1 Act Sink {
2   M(Int);
3 }
```

Any behaviour of type `Sink` must implement the message `M`:

```
1 act sink::Sink {
2   M(n::Int) → print[Str]('got: ' + n);
3 }
```

There can be any number of behaviours with the same type:

```
act sink2::Sink {
  M(n::Int) → {} // Ignore the message.
}
```

An actor is created with a specific behaviour:

```
s::Sink = new sink;
```

Once created, the actor can receive messages. A message is asynchronous: `s ← M(100)` and is placed on the input queue of the target actor. Each actor is processing at most one message at any time, and is idle if there are no messages on its queue and it is not currently handling a message. Since all actors run concurrently, there is no guarantee that of message ordering other than messages sent from the same actor will retain the ordering in which they are sent.

Any number of behaviours can be defined to conform to a behaviour type. The following shows three different implementations of the type `A`. The first, `a`, just loops indefinitely which an actor can do because all message handling occurs concurrently with other actors. The second, `forward` is supplied with a second actor of type `A` to which it forwards messages. Finally, `broadcast` is supplied with a collection of actors of type `A` and will send any message to all of them:

```
Act A {
  M(Int,Bool);
}

act a::A {
  M(n::Int,b::Bool) →
   self ← M(n,not(b));
}

actor::A = new a;

act forward(other::A)::A {
  M(n::Int,b::Bool) →
    other ← M(n,b);
}

proxy::A = new forward(actor);

act broadcast(as::[A])::A {
  toAll(l::[A],n::Int,b::Bool)::Void =
    case l {
      [] → {};
      h::A:t::[A] → {
        h ← M(n,b);
        toAll(t,n,b);
      }
    }
  M(n::Int,b::Bool) →
    toAll(as,n,b);
  }

spray::A = new broadcast([new forward(new a) | i ← 0.1000])
    ;
```

Often we want to do some initialisation when an actor is created. The special message handler → is run each time a new actor is created before any messages are processed:

```
act sink3::Sink {
  → print[Str]('I am initialising');
  M(n::Int) → print[Str]('got: ' + n);
}
```

Actors can refer to other actors. The following pair of actors will bounce messages back and forth until the counter runs down:

```
Act Ping {
  Ping(Int,Pong);
}

Act Pong {
  Pong(Int,Ping);
}

act ping::Ping {
  Ping(0,p::Pong)       → print[Str]('stop');
  Ping(n::Int,p::Pong)  →
    p ← Pong(n-1,self);
}

act pong::Pong {
  Pong(0,p::Ping)       → print[Str]('stop');
  Pong(n::Int,p::Ping)  →
    p ← Ping(n-1,self);
}

p1::Ping = new ping;
p2::Pong = new pong;
p1 ← ping(100,p2);
```

Behaviour definitions can be parameterised so that new actors can initialise the behaviour. For example, in order to provide an identifier to each actor. In the following example, we create a collection of machines and then broadcast jobs. A machine can process a job if the identifier matches:

```
Act Machine {
  Process(Int);
}

act machine(id::Int)::Machine {
  Process(m::Int) when m = id → {
    print[Str]('handling job');
  }
  Process(m::Int) → {} // Ignore job if not target machine.
}

noOfMs::Int          = 100;
noOfJs::Int          = 10000;
machines::[Machine] = [ new machine(n) | n ← 0..noOfMs ];

for job::Int in 0..noOfJs do {
  let targetMachine::Int = random(noOfMs);
  in {
    for m::Machine in machines do {
      m ← Perform(targetMachine);
    }
  }
}
```

An actor may perform `become` to change behaviour. The replacement behaviour must implement the same behaviour type. Subsequent messages are handled by the new behaviour. For example, a resource manager allocates resource on request and becomes pending resource manager for the period of time when resources are unavialable:

```
type Resource = ...;
```

```
Act Consumer {
  Receive(Resource);
}

Act ResourceManager {
  Allocate(Consumer);
  Free(Resource);
}

act manager(rs::[Resource])::ResourceManager {
  Allocate(c::Consumer) when rs = [] → {
    become pending;
    self ← Allocate(c);
  }
  Allocate(c::Consumer) → {
    become manager(tail[Resource](rs));
    c ← Receive(head[Resource](rs);
  }
  Free(r::Resource) → become manager(r:rs);
}

act pending::ResourceManager {
  Allocate(c::Customer) → self ← Allocate(c);
  Free(r::Resource) → become manager([r]);
}
```

8. Binding and Scope

ESL is a statically scoped language meaning that identifiers can be referenced within the textual binding block in which they are defined. Identifiers can be introduced as: function arguments; let-bindings; letrec-bindings; behaviour definitions; behaviour arguments (equivalent to function arguments); top-level definitions; pattern variables; for-loop controls. This section addresses each of these categories in turn.

Let-expressions introduce local bindings for the scope of the body of the let. A let-expression may have multiple bindings in which case they are performed in parallel:

```
let dx::Int = x1-x2;
    dy::Int = y1-y2;
in isqrt((dx*dx)+(dy*dy));
```

A let-binding can be used to define a local function. Special syntax allows a function to be defined without using the keyword `fun`:

```
let
  distance(x1::Int,y1::Int,x2::Int,y2::Int)::Float =
    let dx::Int = x1-x2;
        dy::Int = y1-y2;
    in isqrt((dx*dx)+(dy*dy));
in distance(100,200,50,70);
```

A letrec-expression can be used to create local recursive functions:

```
letrec
  contains(l::[Int],n::Int)::Bool =
    case l {
      [] → false;
      h:t when h=n → true;
      h:t → contains(t,n);
    }
in contains([1,2,3,4,5,6,7,8,9],5);
```

A letrec-expression can be used to create mutually recursive function definitions:

```
letrec
  isEvan(n::Int)::Bool = if n = 0 then true else isOdd(n-1);
  isOdd(n::Int)::Bool = if n = 0 then false else isEven(n-1);
in isEven(101);
```

Letrec-bindings that do not establish functions are performed in sequence.

Behaviour definitions contain a collection of bindings that are established as a letrec and scoped over the message handlers of the behaviour. In addition, the names of local bindings can be exported and referenced using the `_._` operation. An an example, support we have a post office actor that delivers letters to people by matching up the address in the message:

```
Act Person {
  export getAddress::()→Str;
}

act person(address::Str)::Person {
  export getAddress;
  getAddress()::Str = address;
  Open(Letter);
}

data Letter = Letter(Str,Str);

Act PostOffice {
  Deliver(Str);
}

act postOffice(people::[Person])::PostOffice {
  findPerson(address::Str,people::[Person])::Person =
    case people {
      [] → throw[Person]('cannot deliver to ' + address);
      p:ps when p.getAddress() = address → p;
      p:ps → findPerson(address,people);
    }
  Deliver(Letter(name,address)) →
    findPerson(address) ← Open(Letter(name,address));
}
```

9. Modules

labelsec:modules

ESL is a compiled language. The units of compilation are called modules and are usually contained in text files. A module contains a collection of mutually recursive bindings for types, behaviours, functions and values. A module `m` may export some of its bindings so that they can be used by any other module that imports `m`. For example a module defined in a file `a/b/c.esl` defines two names and exports one of them:

```
export f;

f(n::Int)::Int = g(n) + 1;
g(n::Int)::Int = n + 100;
```

A second module can then import `c` and use the exported name:

```
export main;

import 'a/b/c.esl';
```

```
4
5 g(n::Int)::Int = n = 200;
6
7 act Main::Act{} {
8   → print[Int](f(10));
9 }
```

prints 111. Note that the second module above defines a behaviour names `main`. When building an application there should be a root module that defines `main` which is the entry point for the application. When ESL starts to run a module it creates a single actor with the behaviour `main`. There is no restriction on the behaviour type of `main`.

10. Time

ESL applications can be driven by a system clock. This can be useful when the application is driven by click ticks or when the application should run for a specific length of time. In order to receive clock ticks, a behaviour must implement at least one message handler of type `Time(Int)`. Any actor with such a behaviour will receive clock ticks *when the actor is idle*. If an actor does not define a handler for `ntTimeI` then there is no overhead for handling time.

The value of `n` in `Time(n)` is the time in milliseconds since the start of the application. The frequency of the ticks is undefined and therefore there is no guarantee that an actor will receive a tick with a specific value of `n`, and it may be the case the an actor receives multiple messages with the same value of `n`.

An actor may call `wait(n)` which causes the actor's thread to wait for `n` milliseconds. Since all actors have concurrent behaviours, a call of `wait` will not affect any other actor.

The following definition shows a typical pattern involving click ticks: two message handlers are defined, the first to detect whether a limit has been achieved and stops the application, the second just ignores the click:

```
1 Time(n::Int) when n > limit → stopAll();
2 Time(n::Int)                → {}
```

11. State

A value binding establishes an association between an identifier and a value. The association has a scope that defines the ESL code where the identifier can be referenced. ESL uses *lexical scoping* where the association is established in a construct (module, behaviour, let, letrec, function, case-arm, message handler) whose textual definition is the scope.

The original definition of the actor model of computation does not support side-effects. This restriction provides a rather austere application development platform where certain obvious implementation approaches become more unwieldy than they might otherwise be. Therefore, ESL provides side-effects on value bindings. These must be used with care since the actor model was originally designed to avoid issues such as race conditions that occur because of shared state.

Side-effects may be used to provide actors with mutable state:

```
1 Act BankAccount {
2   Deposit(Int);
3 }
4
5 act account::BankAccount {
6   funds::Int = 0;
7   Deposit(n::Int) → funds := funds + n;
8 }
```

An actor may encapsulate its state. Since an actor is singly-threaded, this means that there can be no doubt regarding whether an update has occurred or not when the value of a state variable is used, *i.e.* the state is not used by multiple threads. However, it is often useful for an actor to provide access to its internal state via its interface. This can be achieved in one of two key ways: message passing or an interface operation.

If access is provided by message passing then reference to the state remains singly threaded, however the message must contain the requesting actor who receives the state value via a return message:

```
1  Act BankAccount {
2    Deposit(Int);
3    Withdraw(Int,A);
4  }
5
6  act account::BankAccount {
7    funds::Int = 0;
8    Deposit(n::Int) → funds := funds + n;
9    Withdraw(n::Int,a::A) when n >= funds → {
10     funds := funds - n;
11     A ← Withdrawn;
12   }
13   Withdraw(n::Int,a::A) →
14     a ← WithdrawFailed;
15 }
```

Alternatively, access may be provided via an interface operation. In this case the state can be accessed using multiple threads raising the possibility of race conditions. Consider the following version of the bank account:

```
1  Act BankAccount {
2    export
3      getFunds::() → Int;
4      widthdraw::(Int) → Void;
5      deposit::(Int) → Void;
6  }
7
8  act account::BankAccount {
9    export getFunds,withdraw,deposit;
10   funds::Int = 0;
11   getFunds()::Int = funds;
12   deposit(n::Int)::Void = funds := funds + n;
13   withdraw(n::Int)::Void =
14     if n >= funds
15     then funds := funds - n;
16     else {}
17 }
```

Two different actors may share access to a bank account. The first actor performs the following:

```
1 a.deposit(100);
2 a.withdraw(10);
```

```
3 print[Str]('funds = ' + a.getFunds());
```

what value will be printed? We might expect the funds to be increased by 90. However, the answer depends on whether the second actor has performed a deposit or withdrawal in between line 1 and 2 or between line 2 and 3.

In order to be sure, such transactional blocks must be protected. All access to shared state must use locks to gain exclusive access. Each actor that uses the exported operations must do so within a `grab(lock)` block. Any value may be used as the lock providing the same value is used by all threads that share the state. In this case it makes sense to use the account as the lock:

```
1 grab(a) {
2   a.deposit(100);
3   a.withdraw(10);
4   print[Str]('funds = ' + a.getFunds());
5 }
```

Now providing that all actors that share the account use a similar `grab(a)` to wrap any transactions, the actor above can be sure that the funds will have been increased by 90.

The scheme above relies on the client actors being well behaved in their use of the bank account by wrapping transactions in a `grab(a)`. A better way is for the bank account to enforce the protection of the state whilst still offering an interface. This can be achieved by passing the account a transaction function as follows:

```
1  Act BankAccount {
2    export
3      transaction::(()→Int,(Int)→Void,(Int)→Void)→Void
4  }
5
6  act account::BankAccount {
7    export transaction;
8    funds::Int = 0;
9    getFunds()::Int = funds;
10   deposit(n::Int)::Void = funds := funds + n;
11   withdraw(n::Int)::Void =
12     if n >= funds
13     then funds := funds - n;
14     else {}
15   transaction(action::(()     → Int,
16                        (Int) → Void,
17                        (Int) → Void) → Void)::Void =
18     grab(self) {
19       action(getFunds,deposit,withdraw);
20     }
21 }
```

Any client actor must provide the account with an action which is performed on the client's thread and is guaranteed to occur within the scope of the lock:

```
1 a.transaction(fun(getFunds::()     → Int,
2                   deposit::(Int)   → Void,
3                   withdraw::(Int)  → Int)::Void {
4   deposit(100);
5   withdraw(10);
6   print[Str]('funds = ' + getFunds());
7 });
```

12. Concurrency

ESL actors run concurrently. An actor is created using `new b` or `new b(x,...)` which immediately returns a handle to the newly created actor and schedules the thread of the newly created actor. The behaviour `b` may have an initialisation clause (a message handler starting with →) that is the first action performed on the new thread. After initialisation, the new thread enters a loop that inspects the actor's message queue and dispatches to a message handler in `b` if a message exists, otherwise the thread waits until a message is received. If `b` defines a handler for `Time(Int)` then the ESL system sends the actor a message providing the queue is empty.

ESL is designed to make the creation and interaction of concurrent actors very lightweight. As described in `sec:state`, actors can share state and, if so, must make appropriate use of locks. However, in general the state of an actor is encapsulated and is modified by third party actors using message passing.

The use of large scale concurrency and asynchronous message passing changes the conventional approach to system design and implementation which is based on task decomposition and sequential ordering. This section provides a number of examples that show how actors can be used to implement tasks that take advantage of the concurrent computational model.

12.1. Search

Actors make the implementation of applications involving brute-force search simple because a number of actors can be created that concurrently navigate the search space. The amount of search space explored at any given time is proportional to the number of actors created (and the amount of processing power available).

Consider a square two-dimensional string array that contains a single occurrence of `'*'` at an unknown location. A sequential program might start at `(0,0)` and move through the locations until it finds `'*'`. The worst case execution will inspect all the locations in the array.

A simple actor-based solution creates several actors that all search for the `'*'` concurrently. Each actor can start at a random location and move around at random. This provides a way of comparing the benefits of scaling up the number of actors in a brute-force search.

Figure 3 shows the implementation of the simple search application. The worst-case results of several runs are as follows:

numOfSearchers	count
1	2155079
10	40255
100	30331
1000	656

12.2. Quicksort

Sequential quicksort was defined in section 4. The algorithm clearly lends itself to concurrency since it splits on an element such that the lists of elements below and above the chosen number can be sorted independently. Figure 4 shows the definition of concurrent quicksort in ESL. The actor `qsorter` selects

```
Act Main       {                   }
Act Searcher   { Time(Int);        }
Act Controller { Time(Int); Found; }

size::Int                = 1000;
numOfSearchers::Int      = 10;
board::Array[Array[Str]] = new Array[Array[Str]](size);
max(a::Int,b::Int)::Int  = if a > b then a; else b;

act controller::Controller {
  count::Int = 0;
  Time(n::Int) → count := count + 1;
  Found → {
    print[Str]('Found in ' + count + ' steps');
    stopAll();
  }
}

act searcher(control::Controller)::Searcher {
  x::Int = random(size);
  y::Int = random(size);
  delta(n::Int)::Int = (n + max(random(3) - 1,0)) % size;
  Time(n::Int) →
    if board[x][y] = '*'
    then control ← Found;
    else { x := delta(x); y := delta(y); }
}

act main::Main {
  → {
    for x::Int in 0..size do {
      board[x] := new Array[Str](size);
      for y::Int in 0..size do
        board[x][y] := '';
    }
    board[random(size)][random(size)] := '*';
    let control::Controller = new controller;
    in {
      for i::Int in 0..numOfSearchers do
        new searcher(control);
    }
  }
}
```

Figure 3: Searching an Array

```
data Direction = Left | Right | Final;

Act QSort { Sorted([Int],Direction); }

nums::[Int] = [ random(50) | n::Int ← 0..25 ];

act qmain(l::[Int])::QSort {
  → new qsorter(self,l,Final);
  Sorted(l::[Int],Final) → {
    print[[Int]](l);
    stopAll();
  }
}

act qwaiter(parent::QSort,n::Int,dir::Direction)::QSort {
  left::[Int] = null[[Int]];
  right::[Int] = null[[Int]];
  check()::Void =
    if left <> null[[Int]] and right <> null[[Int]]
    then parent ← Sorted(left+[n]+right,dir);
    else {}
  Sorted(l::[Int],Left)  → { left := l;  check(); }
  Sorted(l::[Int],Right) → { right := l; check(); }
}

act qsorter(parent::QSort,l::[Int],dir::Direction)::QSort {
  → case l {
       [] → parent ← Sorted(l,dir);
       x::Int:l::[Int] → {
         new qsorter(self,[ n | n::Int ← l, ?(n<x) ],Left);
         new qsorter(self,[ n | n::Int ← l, ?(n>x) ],Right)
    ;
         become qwaiter(parent,x,dir);
         {}
       }
    }
  Sorted(l::[Int],d::Direction) → throw[Void] 'error!';
}

act main::Act{} { → new qmain(nums); }
```

Figure 4: Concurrent Quicksort

and element `x` and creates two independent child `qsorter` actors to sort the list of elements below `x` and above `x` respectively before changing behaviour to a `qwaiter`.

The `qwaiter` actor implements a typical scenario: waiting for two independent calculations to terminate before continuing. It achieves this using `Left` and `Right` tokens in the two children and waiting until it receives both tokens before sending the sorted list to its own parent.

Figure 5 shows a diagram representation of a simple quicksort in terms of the actors, their links and the messages that are sent. Note the dashed arrow showing `qsorter` actors becoming `qwaiter` actors.

12.3. Termites

Actor based systems can be used to exhibit *emergent behaviour* where system-level behaviour can be observed as a result of many simple individual behaviours even though each individual behaviour has no knowledge of the whole. An example is *termites* shown in figure 6 where a collection of twigs (black squares) and moved by a collection of termites (red squares) to form piles even though the individual termites do not have any knowledge of pile-formation.

The behaviour of a termite is simple: walk around at random until a twig is found that is not next to other twigs. The termite then picks up the lone-twig and walks at random until it finds another twig that is next to other twigs. The termite then drops the twig and then starts over again.

The implementation of termites in ESL is shown in figure 7. The implementation shows a typical scenario in actor-based systems where there is a single world-state and many different actors that share the state. Each individual needs the state in order to determine its next move which involves an update to the world state. Given that all the actors are operating concurrently, this dependency on a single world state can cause problems if access and updates are not protected. Conventional programming languages provide *data locks* to deal with shared state.

ESL provides locks, but often they are not required because actors operate in terms of asynchronous messages and queues.

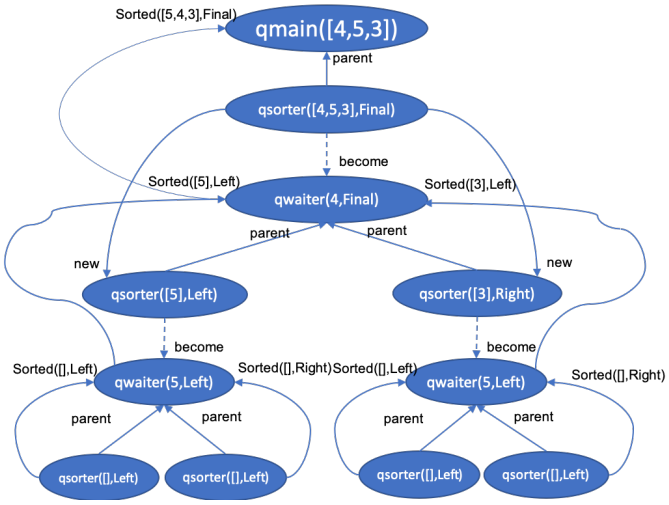


Figure 5: Example Quicksort

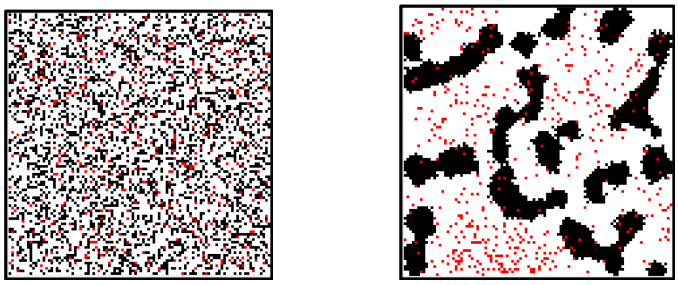
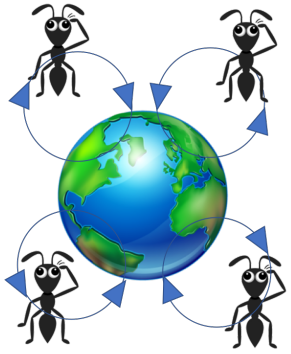


Figure 6: Termites Before and After

The diagram below shows how this works:



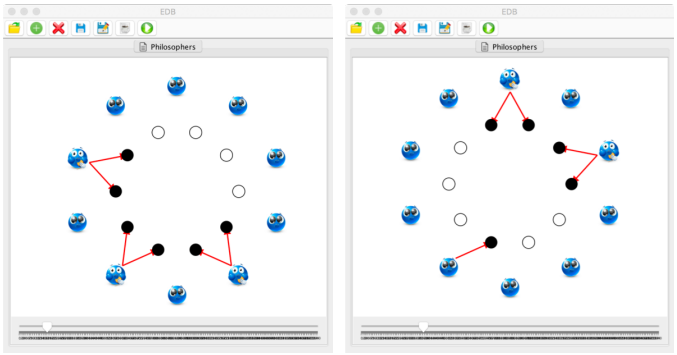
The world at the centre of the figure represents the shared state. Four independent worker actors are shown, although any number can be involved. The shared state offers an interface that supports the operations required by the worker actors which, in turn, provide an interface that represents the different outcomes from performing the world operations.

From the perspective of a single worker, it looks like they get dedicated access to the shared world state because a message sent to the world is queued until the world becomes free. Each worker message must include the worker actor so that the shared state can reply to it. When a worker sends a message to the shared state the worker becomes idle and will be woken up by the reply from the shared state. When the shared state processes a worker message, the worker has exclusive access to the data, the data can be changed and then the message is sent back to the worker which is woken up.

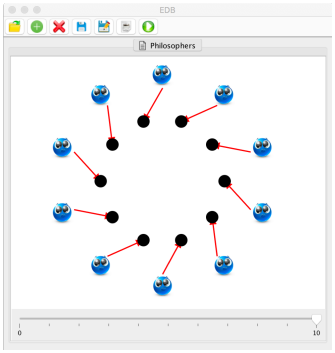
12.4. Dining Philosophers

Dining Philosophers is a standard scenario that is used to exemplify the issues of shared access to resources. Several concurrent processes (philosophers) are vying for shared resources (chopsticks) in order to eat. Each process needs exactly two resources to each and each resource is shared by exactly two processes. When a process acquires a pair of resources, it performs activity (eats) and then releases the resources. Access to resources is assumed to be fair. If a process acquires resources sequentially, then a situation can arise where all resources have acquired one resource and is awaiting on a resource that has been acquired by another process (starvation). In order to avoid starvation, the processes need to wait until they can individually acquire a pair of resources as an atomic action.

A situation where the philosophers are happily eating is shown below:



In the case where chopsticks are grabbed one at a time, the situation rapidly deadlocks:



The implementation of deadlock-free dining philosophers is shown below:

```
1 numberOfPhilosophers::Int = 10;
2 eatTime::Int               = 50;
3 thinkTime::Int             = 50;
4
5 Act Main { Time(Int); }
6 Act Chopstick { }
7 Act Philosopher { Time(Int); }
8
```

```
Act Main { Time(Int); }
Act Grid { SetColour(Int,Int,Str); TermiteAt(Int,Int,Int);}

size::Int           = 120;
limit::Int          = 40000;
numOfTermites::Int = 600;
twig::Str           = 'black';
background::Str     = 'white';
grid::Grid          = new 'esl.grid.Grid'[Grid](size,size,6);

isLegal(x::Int,y::Int)::Bool =
  (x >= 0 and x < size) and (y >= 0 and y < size);

Act Termite {  Search; Drop; FindSpace; GetAway(Int); }

act termite(id::Int,w::World)::Termite {

  x::Int  = random(size);
  y::Int  = random(size);
  dx::Int = random(3) - 1;
  dy::Int = random(3) - 1;

  randomDir()::Void = {
    dx := random(3)-1;
    dy := random(3)-1;
  }
  move()::Void = {
    x := (x + dx); y := (y + dy);
    if x < 0
    then { dx := 1; x := 0; move();
    } else if x > (size - 1)
    then { dx := (0-1); x := (size - 1); move();
    } else if y < 0
    then { dy := 1; y := 0; move();
    } else if y > (size - 1)
    then { dy := 0 -1; y := (size - 1); move();
    } else grid ← TermiteAt(id,x,y);
  }
  moveRandom()::Void = {
    randomDir();
    move();
  }

  → self ← Search;

  Search → {
    moveRandom();
    w ← TryPickup(x,y,self);
  }
  Drop → {
    moveRandom();
    w ← FindPile(x,y,self);
  }
  FindSpace → {
    moveRandom();
    w ← TryDrop(x,y,self);
  }
  GetAway(0) → {
    self ← Search;
  }
  GetAway(n::Int) → {
    move();
    self ← GetAway(n-1);
  }
}


Act World {
  TryPickup(Int,Int,Termite);
  FindPile(Int,Int,Termite);
  TryDrop(Int,Int,Termite);
}

act world::World {
  locations::Array[Array[Str]] =
    let a::Array[Array[Str]] = new Array[Array[Str]](size);
    in {
      for x::Int in 0..size do {
        a[x] := new Array[Str](size);
        for y::Int in 0..size do {
          a[x][y] := if random(100) < 30
                     then twig
                     else background;
          grid ← SetColour(x,y,a[x][y]);
        }
      }
      edb.display[Grid]('Termites',grid);
      a;
    }
  termites::[Termite] =
    [ new termite(n,self) | n::Int ← 0..numOfTermites ];
  foundSingleton(x::Int,y::Int)::Bool =
    locations[x][y] = twig and twigCount(x,y) < 5;
  foundPile(x::Int,y::Int)::Bool =
    locations[x][y] = twig and twigCount(x,y) > 4;
  isTwig(x::Int,y::Int)::Bool =
    if isLegal(x,y)
    then locations[x][y] = twig;
    else false;
  twigCount(x::Int,y::Int)::Int =
    sum([ if isTwig(x+dx,y+dy) then 1 else 0 |
          dx ← [-1,0,1],
          dy ← [-1,0,1],
          ?(x<>0 or y<>0) ]);

  TryPickup(x::Int,y::Int,t::Termite) → {
    if foundSingleton(x,y)
    then {
      locations[x][y] := background;
      grid ← SetColour(x,y,background);
      t ← Drop;
    } else t ← Search;
  }
  FindPile(x::Int,y::Int,t::Termite) → {
    if foundPile(x,y)
    then t ← FindSpace;
    else t ← Drop;
  }
  TryDrop(x::Int,y::Int,t::Termite) → {
    if locations[x][y] = background
    then {
      locations[x][y] := twig;
      grid ← SetColour(x,y,twig);
      t ← GetAway(20);
    } else t ← FindSpace;
  }
}

w::World = new world;

act main::Main {
  Time(n::Int) when n > limit → stopAll();
  Time(n::Int) → { }
}
```

Figure 7: Termites

```
eat()::Void   = wait(eatTime);
think()::Void = wait(thinkTime);

act philosopher(i::Int,
                left::Chopstick,
                right::Chopstick)::Philosopher {
  Time(n::Int) → {
    think();
    grab(left,right) {
      eat();
    }
  }
}

act chopstick::Chopstick {}

chopsticks::[Chopstick] =
  [ new chopstick | i::Int ← 0..numberOfPhilosophers ];
chop(i::Int)::Chopstick =
  nth[Chopstick](chopsticks,i%numberOfPhilosophers);

philosophers::[Philosopher] =
  [ new philosopher(i,chop(i),chop(i+1))
  | i::Int ← 0..numberOfPhilosophers ];
```

Deadlock is avoided because the `grab` block acquires the locks on `left` and `right` at the same time. If the locks are not available then the `grab` block will wait until they are both free and acquire them simultaneously. Changing the definition of `philosopher` to the following leads to deadlock:

```
act philosopher(i::Int,
                left::Chopstick,
                right::Chopstick)::Philosopher {
  Time(n::Int) → {
    think();
    grab(left) {
      grab(right) {
        eat();
      }
    }
  }
}
```

12.5. Segregation

13. Polymorphism

14. Simulation

15. Object Orientation

16. EDB

17. Java Interface

18. Type Checking

19. Compilation

20. Implementation

[1] Rajesh K Karmani, Amin Shali, and Gul Agha. Actor frameworks for the jvm platform: a comparative analysis. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, pages 11–20. ACM, 2009.

A. Syntax and Type Checking

B. List Operations

```
adjoin[T](x::T,l::[T])::[T] =
  if member[T](x,l)
  then l;
  else x:l;

select1[T](l::[T],d::T,p::(T)→Bool)::T =
  case l {
    [] → d;
    h:t when p(h) → h;
    h:t → select1[T](t,d,p);
  }

map[M,N](f::(M)→N,l::[M])::[N] =
  case l {
    [] → [];
    m:ms → (f(m)):map[M,N](f,ms);
  }

remove[T](v::T,l::[T])::[T] =
  case l {
    h:t when (h=v) → remove[T](v,t);
    h:t          → h:remove[T](v,t);
    []           → [];
  }

remove1[T](v::T,l::[T])::[T] =
  case l {
    h:t when (h=v) → t;
    h:t          → h:remove1[T](v,t);
    []           → [];
  }

length[T](l::[T])::Int =
  case l {
    h:t → 1 + length[T](t);
    [] → 0;
  }

flatten[T](lists::[[T]])::[T] =
  case lists {
    h:t → h+flatten[T](t);
    [] → [];
  }

count[T](x::T,l::[T])::Int =
  case l {
    h:t → if h=x then 1+count[T](x,t); else count[T](x,t);
    []  → 0;
  }

hasPrefix[T](list::[T],prefix::[T])::Bool =
  case list,prefix {
    l1,[] → true;
    x:list,y:prefix when x=y → hasPrefix[T](list,prefix);
    l1,l2 → false;
  }

nth[T](l::[T],n::Int)::T =
  case l {
    h:t    → if n = 0 then h; else nth[T](t,n-1);
    []  → throw[T]('cannot take nth element.');
  }

take[T](l::[T],n::Int)::[T] =
  if n = 0
  then [];
  else
    case l {
      h:t → h:(take[T](t,n-1));
      [] → throw[[T]]('cannot take element ' + n);
    }

drop[T](l::[T],n::Int)::[T] =
  if n = 0
  then l;
  else
    case l {
      h:t → drop[T](t,n-1);
      [] → throw[[T]]('cannot drop element ' + n);
    }

subst[T](n::T,o::T,l::[T])::[T] =
  case l {
    [] → [];
    h:t →
      if h = o
      then n:(subst[T](n,o,t));
      else h:(subst[T](n,o,t));
  }

head::Forall[T]([T])→T = fun(l::[T])::T
  case l {
    h:t → h;
    [] → throw[T]('cannot take the head of []');
  }

tail::Forall[T]([T])→[T] = fun(l::[T])::[T]
  case l {
    h:t → t;
    [] → throw[[T]]('cannot take the tail of []');
  }

isNil[T](l::[T])::Bool =
  case l {
    [] → true;
    l → false;
  }

member[T](e::T,l::[T])::Bool =
  case l {
    [] → false;
    x:xs when x = e → true;
    x:xs→ member[T](e,xs);
  }

reverse[T](l::[T])::[T] =
  case l {
    [] → [];
    x:xs → reverse[T](xs) + [x];
  }

exists[T](pred::(T)→Bool,l::[T])::Bool =
  case l {
    [] → false;
    x:xs when pred(x) → true;
    x:xs → exists[T](pred,xs);
  }

forall::[T](pred::(T)→Bool,l::[T])::Bool =
  case l {
    [] → true;
    x:xs when pred(x) → forall[T](pred,xs);
    x:xs → false;
  }

replaceNth[T](l::[T],n::Int,x::T)::[T] =
  case l {
    [] → throw[[T]]('cannot replace nth of []');
    h:t when n=0 → x:t;
    h:t → h:replaceNth[T](t,n-1,x);
  }

indexOf[T](t::T,l::[T])::Int =
  case l {
    [] → 0-1;
    h:l when h=t → 0;
    h:l → 1 + indexOf[T](t,l);
  }

select[T](p::(T)→Bool,l::[T])::[T] =
  case l {
    [] → [];
    h:t when p(h) → h:select[T](p,t);
    h:t → select[T](p,t);
  }

reject[T](p::(T)→Bool,l::[T])::[T] =
  case l {
    [] → [][T];
    h:t when p(h) → reject[T](p,t);
    h:t → h:reject[T](p,t);
  }

last[T](l::[T])::T =
  case l {
    [] → throw [T]('no last element of empty list');
    x:[] → x;
    h:l → last[T](l);
  }

butlast[T](l::[T])::[T] =
  case l {
    [] → [];
    [x] → [];
    h:l → h:butlast[T](l);
  }

occurrences[T](x::T,l::[T])::Int =
  case l {
    [][T] → 0;
    h:t when h=x → 1 + occurrences[T](x,t);
    h:t → occurrences[T](x,t);
  }

filter[T](pred::(T)→Bool,l::[T])::[T] =
  case l {
    [] → [];
    h:t →
```

Syntax:

t ::=		*terms*
	x	*variable*
	new(t)	*creation*
	$\lambda(\bar{d})$ t	*function*
	act { $\bar{d}$ $\bar{h}$ }	*behaviour*
	n	*number*
	s	*string*
	b	*boolean*
	grab(x) t	*lock*
	if t **then** t **else** t	*conditional*
	t ← t	*send*
	case t { $\bar{h}$ }	*projection*
	[$\bar{t}$]	*list*
	t($\bar{t}$)	*application*
	t[$\bar{T}$]	*type application*
	C($\bar{t}$)	*injection*
i ::=	d=t	*initialisation*
d ::=	x[$\bar{X}$]::T	*declarations*
h ::=	C($\bar{x}$) → t	*handlers*
T ::=		*types*
	Int	*integer type*
	Bool	*boolean type*
	Str	*string type*
	∀ $\bar{X}$. T	*universal*
	X	*variable*
	Union { $\bar{m}$ }	*union*
	Act { $\bar{d}$ $\bar{m}$ }	*behaviour*
	($\bar{T}$) → T	*function type*
m ::=	C($\bar{T}$)	*message type*

Type Checking:

$$\frac{x :: T \in \Gamma}{\Gamma \vdash x :: T}\ \text{T-VAR}$$

$$\frac{\Gamma \vdash t :: \textbf{Act}\ \{\ \bar{d}\ \bar{m}\ \}}{\Gamma \vdash \textbf{new}(t) :: \textbf{Act}\ \{\ \bar{d}\ \bar{m}\ \}}\ \text{T-NEW}$$

$$\frac{\Gamma, \bar{x}::\bar{T} \vdash t::T}{\Gamma \vdash \lambda(\bar{d})\ t::(\bar{T})\to T}\ \text{T-FUN}$$

$$\Gamma \vdash \textbf{act}\ \{\ \bar{d}\ \bar{m}\ \}::\textbf{Act}\ \{\ \bar{d}\ \bar{m}\ \}\ \text{T-ACT}$$

$$\frac{\Gamma \vdash t_1::\textbf{Act}\ \{\ \bar{d}\ \bar{m}\ \} \quad \Gamma \vdash t_2::\textbf{Union}\ \{\ \bar{m}\ \}}{\Gamma \vdash t_1 \leftarrow t_2::\textbf{Union}\ \{\ \bar{m}\ \}}\ \text{T-SEND}$$

$$\Gamma \vdash n::\textbf{Int}\ \text{T-INT}$$

$$\Gamma \vdash s::\textbf{Str}\ \text{T-STR}$$

$$\Gamma \vdash b::\textbf{Int}\ \text{T-BOOL}$$

$$\frac{\Gamma \vdash t::T}{\Gamma \vdash \textbf{grab}(x)\ t\ ::\ T}\ \text{T-LOCK}$$

$$\frac{\Gamma \vdash t_1::\texttt{Bool} \quad \Gamma \vdash t_2, t_3::T}{\Gamma \vdash \textbf{if}\ t_1\ \textbf{then}\ t_2\ \textbf{else}\ t_3\ ::\ T}\ \text{T-IF}$$

$$\frac{\Gamma \vdash \textbf{case}\ t\ \{\bar{m}_1\} :: T \quad \Gamma \vdash \textbf{case}\ t\ \{\bar{m}_2\} :: T}{\Gamma \vdash \textbf{case}\ t\ \{\bar{m}_1, \bar{m}_2\} :: T}\ \text{T-CASE1}$$

$$\frac{\Gamma \vdash t_1 :: \textbf{Union}\{\bar{m}, C(\bar{T})\} \quad \Gamma, \bar{x} :: \bar{T} \vdash t_2 :: T}{\Gamma \vdash \textbf{case}\ t_1\ \{C(\bar{x}) \to t_2\} :: T}\ \text{T-CASE2}$$

$$\Gamma \vdash [] :: \forall X.[X]\ \text{T-NIL}$$

$$\frac{\Gamma \vdash [\bar{t}_1] :: [T] \quad \Gamma \vdash [\bar{t}_2] :: [T]}{\Gamma \vdash [\bar{t}_1, \bar{t}_2] :: [T]}\ \text{T-LIST1}$$

$$\frac{\Gamma \vdash t :: T}{\Gamma \vdash [t] :: [T]}\ \text{T-LIST2}$$

$$\frac{\Gamma \vdash \bar{t} :: \bar{T} \quad \Gamma \vdash t :: (\bar{T}) \to T}{\Gamma \vdash t(\bar{t}) :: T}\ \text{T-APP}$$

$$\frac{\Gamma \vdash t :: \forall \bar{X}.T}{\Gamma \vdash t[\bar{T}] :: T[\bar{T}/\bar{X}]}\ \text{T-TAPP}$$

$$\frac{\Gamma, C(\bar{T}) \mapsto \textbf{Union}\{\bar{m}\} \quad \Gamma \vdash \bar{t} :: \bar{T}}{\Gamma \vdash C(\bar{t}) :: \textbf{Union}\{\bar{m}\}}\ \texttt{T-INJ}$$

Figure A.8: ESL Syntax and Type Checking

```
189       if pred(h)
190       then h:filter[T](pred,t);
191       else filter[T](pred,t);
192   }
193
194 foldr[M,N](map::(M)→N,op::(N,N)→N,empty::N,list::[M])::N =
195   case list {
196     [] → empty;
197     h:t → op(map(h),foldr[M,N](map,op,empty,t));
198   }
199
200 id[T]::(T)→T = fun(x::T)::T x;
201
202 sum(list::[Int])::Int =
203   foldr[Int,Int](id[Int],fun(x::Int,y::Int)::Int x+y,0,list);
204
205 removeDups[T](list::[T])::[T] =
206   case list {
207     [] → [];
208     x:l → x:removeDups[T](remove[T](x,l));
209   }
210
211 prefixes[T](l::[T])::[[T]] =
212   letrec pre(prefixes::[[T]],rest::[T])::[[T]] =
213     if rest = []
214     then prefixes;
215     else pre(prefixes + [
216            last[[T]](prefixes)+[head[T](rest)]
217          ],tail[T](rest));
218   in pre([[]],l);
219
220 takeWhile[T](p::(T)→Bool,l::[T])::[T] =
221   case l {
222     [] → [];
223     t:l when p(t) → t:takeWhile[T](p,l);
224     t:l → [];
225   }
226
227 dropWhile[T](p::(T)→Bool,l::[T])::[T] =
228   case l {
229     [] → [];
230     t:l when p(t) → dropWhile[T](p,l);
231     l → l;
232   }
233
234 removeAll[T](removables::[T],l::[T])::[T] =
235   case removables {
236     [] → l;
237     t:r → removeAll[T](r,remove[T](t,l));
238   }
```