Addendum to
The Limbo Programming Language

Vita Nuova
30 March 2005

1. Introduction

This addendum provides a brief summary of several language changes to Limbo since *The Limbo Programming Language* was last revised:

- buffered channels
- unrestricted `alt`
- function references
- exceptions
- exponentiation
- fixed-point types

2. Buffered channels

A buffered channel can now be declared:

```
c := chan[1] of int;
```

Here the buffer size is 1. A send on this channel will succeed immediately if there is a receiver waiting or if the buffer is empty. A receive on this channel will succeed immediately if there is a data item in the buffer. This allows us to write a very simple locking mechanism:

```
acquire(c: chan of int)
{
        c <-= 0;
}

release(c: chan of int)
{
        <-c;
}

new(): chan of int
{
        return chan[1] of int;
}
```

The declaration

```
c := chan[0] of int;
```

is equivalent to

```
        c := chan of int;
```

An attempt to create a channel with a negative buffer size will raise an exception. An attempt to create a channel with a very large buffer may result in an immediate memory exception if there is not enough room for the buffer.

3. Unrestricted alt

The implementation has changed to remove the restriction that only one process can be waiting in an `alt` to send or receive on a particular channel. The busy exception never occurs now. Thus you can do

```
i()
{
        c := chan of int;
        c1 := chan of int;
        spawn p(c, c1);
        <-c;
        spawn p(c, c1);
        <-c;
        for(i := 0; i < 20; i++)
                c1 <-= i;
}

p(c: chan of int, c1: chan of int)
{
        c <-= 0;
        for(;;)
                alt{
                        i := <-c1 =>
                                ;
                }
}
```

The two spawned processes can both wait on `c1` without fuss. Processes are queued on a strict FIFO basis so, in the example above, the two processes receive on `c1` alternately.

4. Function references

Function references may be declared as follows:

```
fp: ref fn(s1: string, s2: string): int;
```

Given the function

```
cmp(s1: string, s2: string): int
{
        if(s1 < s2)
                return -1;
        if(s1 > s2)
                return 1;
        return 0;
}
```

a reference to it can be created by assignment:

```
fp = cmp;
```

where the name can be qualified by an explicit module reference as usual:

```
fp = mod->cmp;
```

or it can be returned from a function:

```
Cmp: type ref fn(s1: string, s2: string): int;

rcmp(s1: string, s2: string): int
{
        return -cmp(s1, s2);
}

choose(i: int): Cmp
{
        if(i)
                return rcmp;
        return cmp;
}
```

(the declaration of the synonym `Cmp` was done only for clarity). They may be declared and passed as parameters:

```
sort(a: array of string, f: ref fn(s1, s2: string): int): array of string
{
        # ...
}
        # ...
b := sort(a, cmp);
c := sort(a, rcmp);
```

The function is called via the reference by

```
        r := fp("fred", "bloggs");
```

Otherwise function references behave just like any other reference type.

5. Exceptions

Both string exceptions and user defined exceptions are now provided. The `Sys` module interface to exceptions has been removed and replaced by new language constructs in limbo.

5.1. String exceptions

Simple string exceptions can be raised as follows

```
raise s;
```

where *s* is any value of type string (it need not be constant).

Exception handlers may be attached to a block (or sequence of statements) :-

```
{
        foo();
        bar();
} exception e {
"a" or "b" =>
        sys->print("caught %s\n", e);
        raise;
"ab*" =>
        sys->print("caught %s\n", e);
        exit;
"abcd*" =>
        sys->print("caught %s\n", e);
        raise e;
"a*" =>
        sys->print("caught %s\n", e);
        raise "c";
"*" =>
        sys->print("caught %s\n", e);
}
LL:
```

Any exception occurring within the block (and in nested function calls within the block) can potentially be caught by the exception handler. An exception is caught by a guard exactly maching the exception string or by a guard `"`*s*`*"` where *s* is a prefix of the exception string. The most specific match is used. Thus a raise of "a" will be caught by the first guard and not by the fourth guard. A raise of "abcde" is caught by the third and not the second or fourth. If a match is found, the sequence of statements following the guard are executed. If not, the system searches for a handler at a higher level.

As shown above, the exception is available through the exception identifier (e in this case) if given following the exception keyword.

The exception is reraised using

```
raise;
```

or

```
raise e;
```

Both the block and the exception code will fall through to the statement labelled LL unless, of course, they do an explicit exit, return or raise first.

5.2. User-defined exceptions

You can declare your own exceptions:

```
implement Fibonacci;

include "sys.m";
include "draw.m";

Fibonacci: module
{
        init: fn(nil: ref Draw->Context, argv: list of string);
};

init(nil: ref Draw->Context, nil: list of string)
{
        sys := load Sys Sys->PATH;
        for(i := 0; ; i++){
                f := fibonacci(i);
                if(f < 0)
                        break;
                sys->print("F(%d) = %d\n", i, f);
        }
}

FIB: exception(int, int);

fibonacci(n: int): int
{
        {
                fib(1, n, 1, 1);
        }exception e{
        FIB =>
                (x, nil) := e;
                return x;
        "*" =>
                sys->print("unexpected string exception %s raised\n", e);
        * =>
                sys->print("unexpected exception raised\n");
        }
        return 0;
}

fib(n: int, m: int, x: int, y: int) raises (FIB)
{
        if(n >= m)
                raise FIB(x, y);

        {
                fib(n+1, m, x, y);
        }exception e{
        FIB =>
                (x, y) = e;
                x = x+y;
                y = x-y;
                raise FIB(x, y);
        }
}
```

`FIB` is a declared exception that returns two integers. The values are supplied when raising the exception:

```
raise FIB(3, 4);
```

When caught the values can be recovered by treating the declared exception identifier as if it were a tuple of 2 integers:

```
(x, y) = e;
```

In general each exception alternative treats the exception identifier appropriately : as a string when the exception qualifier is a string, as the relevant tuple when the exception is declared.

If you do

```
"abcde" or FIB =>
        (x, y) = e;
        sys->print("%s\n", e);
```

you will get a compiler error as `e`'s type is indeterminate within this alternative.

Reraising is the same as in the case of string exceptions.

Note also the difference between the string guard `"*"` and the guard `*` in the function fibonacci. The former will match any string exception, the latter any exception. If a string exception does occur it matches the former as it is the most specific. If an unexpected user defined exception occurs it matches the latter.

The main difference between declared exceptions and string exceptions is that the former must be caught by the immediate caller of a function that raises them, otherwise they turn into a string exception whose name is derived from that of the exception declaration.

5.3. The `raises` clause

The definition of the function fib in the above example also lists the user defined exceptions it can raise via the use of a `raises` clause. In this case there is just the one exception (`FIB`). These clauses (if given) must be compatible between any declaration and definition of the function.

The compiler reports instances of functions which either raise some exception which is not mentioned in their raises clause or does not raise some exception which is mentioned in their raises clause. Currently the report is a warning.

6. Exponentiation

The exponentiation operator (written as `**`) is now part of the Limbo language. Its precedence is above that of multiplication, division and modulus but below that of the unary operators. It is right associative. Thus

```
3**4*2 = (3**4)*2 = 81*2 = 162
-3**4 = (-3)**4 = 81
2**3**2 = 2**(3**2) = 2**9 = 512
```

The type of the left operand must be `int`, `big` or `real`. The type of the right operand must be `int`. The type of the result is the type of the left operand.

7. Fixed point types

A declaration of the form

```
x: fixed(0.2, 12345.0);
```

declares `x` to be a variable of a fixed point type. The scale of the type is 1/5 and the maximum absolute value of the type is 12345.0.

Similarly

```
x: fixed(0.125, 4096.0)
```

specifies a scale of 0.125 and a maximum absolute value of 4096. This requires only 17 bits so the underlying type will be `int` and the compiler is free to allocate the remaining 15 bits to greater range or greater accuracy. In fact the compiler always chooses the latter.

The maximum absolute value is optional :-

```
x: fixed(0.125);
```

is equivalent to

```
x: fixed(0.125, 2147483647.0 * 0.125);
```

and ensures the underlying type is exactly an int ie the compiler has no scope to add any extra bits for more accuracy.

A binary fixed point type with 8 bits before the binary point and 24 after might therefore be declared as

```
x: fixed(2.0**-24);
```

The scale must be static: its value known at compile time and it must be positive and real; similarly for the maximum absolute value when specified.

Currently the only underlying base type supported is `int`.

A shorthand for fixed point types is available through the use of `type` declarations:

```
fpt: type fixed(2.0**-16);
```

We can then do

```
x, y, z: fpt;
zero: con fpt(0);

x = fpt(3.21);
y = fpt(4.678);
z = fpt(16r1234.5678);
z = -x;
z = x+y;
z = x-y;
z = x*y;
z = x/y;
sys->print("z=%f", real z);
```

There is no implicit numerical casting in Limbo so we have to use explicit casts to initialize fixed point variables. Note the use of a base to initialize `z` using a new literal representation.

Given

```
fpt1: type fixed(0.12345);
x: fpt1;
fpt2: type fixed(0.1234);
y: fpt2;
fpt3: type fixed(0.123);
z: fpt3;
```

then

```
z = x*y;
```

is illegal. We must add casts and do

```
z = fpt3(x)*fpt3(y);
```

ie type equivalence between fixed point types requires equivalence of scale (and of maximum absolute value when specified).

Fixed point types may be used where any other numerical type (byte, int, big, real) can be used. So you can compare them, have a list of them, have a channel of them, cast them to or from string and so on.

You cannot use complement(~), not(!), and(&), or(|), xor(^) or modulus(%) on them as fixed point types are basically a form of real type.

7.1. Accuracy

A fixed point value is a multiple of its scale. Given fixed point values X, Y and Z of scale s, t and u respectively, we can write

```
X = sx
Y = ty
Z = uz
```

where x, y and z are integers.

For the multiplication Z = X*Y the accuracy achieved is given by

```
| z - (st/u)xy | < 1
```

and for the division Z = X/Y

```
| z - (s/(tu))x/y | < 1
```

That is, the result is always within the result scale of the correct real value.

This also applies when casting a fixed point type to another, casting an integer to a fixed point type and casting a fixed point type to an integer. These are all examples of the multiplication law with t = y = 1 since an integer may be thought of as a fixed point type with a scale of 1.