Miranda – Pure Functional Languages

Miranda

Characteristics of Miranda:

- Purely Functional
- Lazy Evaluation
- Statically typed with Type Inference.

Haskell is similar to Miranda. It is more complicated and more widely used.

Drawbacks:

- Efficiency
- Acceptance
- Unnatural for some things: I/O and user interaction.

Running Miranda

On educational Linux machines:

```
% mira
```

or

```
% mira myfile.m
```

This is the Miranda read-eval-print loop.

Using `mira`

Miranda compiles functions to an intermediate representation.

While running `mira`:

`/edit` – Opens `emacs` with source file. When saved and quit, code gets recompiled and reloaded.
`/quit` – Quits `mira`.
`/help` – Prints `mira` command list.
`/aux` – Prints a different `mira` command list.
`?` – Prints list of all identifiers in global scope.
`/man` – Starts Miranda Manual (try item 28).

`<expr>` – Evaluates `<expr>` and returns its value.
`<expr> ::` – Evaluates `<expr>` and returns its type.

Miranda Expressions

function calls: <f-expr> <arg-expr-1> ... <arg-expr-k>
For Example:

```
sqrt 81
reverse "hello"
map sqrt [1,4,9
```

infix operators: <expr-1> <op> <expr-2>
For Example:

```
5 + 10
5 > 10
1 : [5,6,7]
[1,2,3] ++ [4,5,6]
[0,1,2,3] ! 2
```

infix operators as expressions: (<op>) <expr-1> <expr-2>
Turn on infix operator into a 2 argument function with (<op>).
For Example:

```
(+) 5 10
(:) 1 [5,6,7]
```

Use parentheses for grouping.

Miranda Types

num: Either real or integer. E.g. `5` and `3.14`.

char: E.g. `'c'` and `'\n'`.

bool: E.g. `True` and `False`.

lists: Can be list of any single type.
E.g. `[True,True,False]` and `[1,2,3.14]` and `[[1],[2,3],[]]`.

string: Just a list of `char`. E.g. `['h','i']` or `"hello"`.

tuples: groupings of values.
E.g. `(1,"hi",True)` is of type `(num,[char],bool)`

functions: Functions are first class and have types.

Type Exercise

What are the types of (or is it a type error?):

1. `(1,'a',3.4)`
2. `[1,'a',3.4]`
3. `[(1,'a',3.4),(2,'b',6)]`
4. `[[1,2,3],[]]`
5. `[[1,2,3],['a','b']]`
6. `["hi",'a','b']]`
7. `([1,2,3],['a','b'])`

Lists

Like Scheme, Lists are building block for Miranda.
Creating lists:

```
[1,2,3]
[1..10]      || list of 1 to 10
[1..]        || infinite data structure
```

List Functions and operators:

`(:)` – the cons operator: `1 : [2,3,4]`.
`(++)` – the append operator: `[1,2] ++ [3,4]`.
`(!)` – the index operator: `[0,1,2,3] ! 2`.
`take k list` – first `k` elements of `list`.
`drop k list` – `list` with first `k` elements dropped.
`member list obj` – true if `obj` is in `list`.

Functions

Define functions like:

```
add x y = x + y

add3 x y z = x + y + z
```

These can be used as:

```
add 5 10

add3 5 10 15
```


Functions with *Guards*

(*Whitespace in function definitions is significant!*)

Instead of if-statements, use functions with *guards*:

```
my_abs x = x, if x >= 0
         = -x, otherwise          || note '=' are aligned.
```

Define `is_positive x` and `factorial n`

Functions with *Patterns*

```
is_zero 0 = True
is_zero x = False                 || 'x' will match anything.
```

Notes:

- First pattern that matches is used.
- Patterns can contain variables.

```
my_and True x = x
my_and False x = False
```

Define `my_xor`

More Patterns

Patterns on lists:

```
car (x:xs) = x
cdr (x:xs) = xs

length [] = 0                     || match empty list
length (x:xs) = 1 + length xs     || match non-empty list
```

Patterns on tuples:

```
sum_3tuple (x,y,z) = x+y+z

check_length (len,str) = len = length str
```

Patterns on numbers:

```
factorial 0 = 1
factorial (n+1) = (n+1) * factorial n
```

Patterns on Lists

These are *very* useful.

Miranda provides `hd` and `tl` (like `car` and `cdr`). Compare:

```
length [] = 0                   || match empty list
length (x:xs) = 1 + length xs   || match non-empty list

length2 list = 0, if list = []
             = 1 + length2 (tl list), otherwise
```

Can pattern match on any constant number of items in list:

```
is_length_one [x] = True
is_length_one xs = False        || this is a "match all" pattern

is_one_two_three [1,2,3] = True
is_one_two_three xs = False

cadr (y:x:xs) = x
```


Local Variables & Scoping

Miranda is lexically scoped like Scheme:

```
twice_reversed list = rev ++ rev
                      where rev = reverse list

|| make forward-forward-reverse-reverse... list
ffrrffrr list = ffrr ++ ffrr
                where ffrr = ff ++ rev ++ rev
                             where
                                rev = reverse list
                                ff = list ++ list
```


Currying

Miranda functions are automatically curried:

```
add x y = x + y
add3 x y z = x + y + z
```

Consider:

1. `add 5 10`
2. `(add 5) 10`
3. `((add3 5) 10) 15`

Here `(add 5)` is the function that takes an number an returns than number plus 5.

Consider `plus5 x = add 5 x` or equivalently `plus5 = add 5` or even `plus5 = add3 1 4`

Can you redefine `add` using currying?

Function Types

```
add x y = x + y
plus5 = add 5
```

What is the type of `plus5`?

It takes a number and returns a number. Type is: `num->num`

What is type of `add`?

Curried Function Types

```
add x y = x + y
plus5 = add 5
```

Recall type of `plus5` is `num->num`.

What is type of `add`?

When `add` is passed `5` it returns `plus5`.

Type of `add` is `num->(plus5 ::)` which is `num->(num->num)`

Define `->` to be right associative.

```
add ::= num->num->num
```


Curried Types (cont)

In general for :

```
f x1 x2 ... xN = y
```

We have:

```
f ::= (x1 ::)->(x2 ::)->...->(xN ::)->(y ::)
```

(Note: `y` may be an function)

Curried Type Examples

Given:

```
foo 5 True 'a' = [1,2,3]
bar (2,'a') = add
baz 1 = foo
```

What are:

1. `foo ::`
2. `bar ::`
3. `baz ::`
4. `foo 5 True ::`

Polymorphic Functions

Recall definition of `car`:

```
car (x:xs) = x
```

What is type of `car`?

Type of `car`

```
car (x:xs) = x
```

We can use `car` as `car [1,2,3]` or `car [True,False]`.

Is `car` type `[num]->num` or `[bool]->bool`?

Actually `car` is type `[*]->*`.

Notice that `car` is a *polymorphic* function: it takes parameters of different types. E.g. `[num]` or `[bool]`.

Higher-Order Functions

Miranda has `map`, `map2`, `foldr`, `foldl`, and `const` builtin.

For Example:

```
const x y = x

map2 f [] [] = []
map2 f (x:xs) (y:ys) = f x y : map2 f xs ys
```

Evaluate the following:

1. `map2 (+) [1,2] [3,4]`
2. `foldl (*) 1 [1,2,3]`
3. `foldr (:)  [4,5,6] [1,2,3]`
4. `const 5 False`
5. `foldr (+) 0 (map (const 1) [1..100])`

We can define our own:

```
twice f x = f (f x)
rev2 f x y = f y x
```


Higher-Order Function Types

Recall:

```
const x y = x

map2 f [] [] = []
map2 f (x:xs) (y:ys) = f x y : map2 f xs ys
```

What are the values of:

1. `map2 ::`
2. `const ::`
3. `const 5 ::`
4. `map2 (+) ::`
5. `map ::`
6. `map (const True) ::`
7. `map2 (<) ::`

Foldr and Foldl

```
foldr op id = f
              where
                 f [] = id
                 f (x:xs) = op x (f xs)

foldl op id = f id
              where
                 f id [] = id
                 f id (x:xs) = f (op id x) xs

|| these are standard
sum = foldl (+) 0
product = foldl (*) 1
or = foldr (\/) False
and = foldr (&) True
```

Lazy Evaluation

Consider definitions:

```
my_if True x y = x
my_if False x y = y

ones = 1 : ones
```

Now run:

```
my_if (x>y) 3 4

my_if (x>y) 3 (1/0)

ones ! 1000
```


Prime Number Generators

```
factors n = filter is_factor [1..n]
            where
               is_factor k = n mod k = 0

dullprimes = filter isprime [2..]
             where
                isprime p = (factors p = [1,p])

interestingprimes
   = sieve [2..]
     where
        sieve (p:x) = p : sieve filtered
                      where
                         filtered = filter not_multiple x
                         not_multiple n = n mod p > 0
```


Hamming Numbers

Hamming numbers are $2^i 3^j 5^k$ for some i, j, k.

Lets generate an infinite list of Hamming numbers in order.

```
my_merge (x:a) (y:b) = x : my_merge a (y:b) , if x<y
                     = y : my_merge a b , if x=y
                     = y : my_merge (x:a) b , otherwise

ham = 1: my_merge ham2 (my_merge ham3 ham5)

ham2 = map (* 2) ham
ham3 = map (* 3) ham
ham5 = map (* 5) ham
```


Type Inference

Do not need to provide miranda with types, but it is a good idea.

```
map::(*->**)->[*]->[**]
map f [] = []
map f (x:xs) = f x : map f xs
```

Whether you do or not, Miranda will infer the function type:
From line 1: `f::*`, `[]::[**]`, and `[]::[***]`
From line 2: `x::**` and `(f x)::***`

Thus: `f::**->***` so `map::(**->***)->[**]->[***]`

But we can rewrite using less `*`s:

`map::(*->**)->[*]->[**]`

Type Inference (cont)

Use type inference to find type of: `twice f x = f (f x)`

Currying & Type Inference

Consider: `map::(*->**)->[*]->[**]` and `neg::num->num`

What is `map neg ::`

Currying & Type Inference (cont)

Consider: `map::(*->**)->[*]->[**]` and `(+)::num->num->num`

What is `map (+) ::`

Type Aliases

Can create alias types:

```
string == [char]

arith == num->num->num
```

Tagged Types

Also called *algebraic types*:

```
weather ::= Rainy | Sunny | Cloudy | Snowy
month ::= Jan | Feb | Mar | Apr | May | Jun | Jul | Aug | ...

forecast :: month->weather

forecast Aug = Sunny
forecast Jul = Sunny
forecast xxx = Rainy
```

Tag names must begin with a capital letter (variables must begin with lowercase letter).

`Rainy`, `Sunny`, etc. are constructors for type `weather`.

Tagged Types (cont)

Tagged values can contain variables (Product Types):

```
shape ::= Circle num | Square num |
          Rectangle num num | Triangle num num

pi = 3.14159

area :: shape->num
area (Circle r) = pi * r * r
area (Square width) = width * width
area (Rectangle width height) = width * height
area (Triangle base height) = base * height / 2.0

simplify :: shape->shape
simplify (Rectangle w w) = Square w
simplify s = s
```

These constructors are like functions.
For example, `Triangle ::= num->num->shape`.

Polymorphic Tagged Types

We can also have polymorphic and recursive tagged types:

```
tree * ::= EmptyTree | Node * (tree *) (tree *)

t = Node 7 EmptyTree (Node 4 EmptyTree EmptyTree)

flatten EmptyTree = []
flatten (Node n left right) = flatten left ++ [n] ++ flatten right
```

What is `t ::`

What does `flatten t` return?

What is `Node ::`? How about `Node 5 ::`?

List Comprehensions

Provide handy notation for constructing lists from other lists.

Recall set notation: $\{x \mid x \in A \text{ and } x > 0\}$

Miranda has a very similar notation: `[ x | x <- list; x > 0 ]`

This is read: "`x` such that `x` comes from `list` and `x > 0`."

```
[x | x <- [1..10]; x mod 2 = 0]    || evens between 1 and 10
[x*2 | x <- [1..5]]                || evens between 1 and 10

[x^3 | x <- [1..]]                 || all cubes.
```

List Comprehensions (cont)

Redefine `filter` and `map`

```
filter' predicate list = [ x | x <- list; predicate x ]

map' f list = [ f x | x <- list ]
```

Can use patterns! If patterns do not match, item is discarded.

```
shapes = [Circle 5, Square 10, Rectangle 1 2, Circle 2]

radiuses list = [r | (Circle r) <- list]
```


List Comprehensions (cont)

Lets write quicksort with list comprehensions:

More examples:

```
|| produce all pairs from [1..5] and "abcd"
[ (a,b) | a <- [1..5]; b <- "abcd"]

|| same but diagonalized, for infinite lists
[ (a,b) // a <- [1..5]; b <- "abcd"]
```


Abstract Data Types

```
abstype queue *
    with emptyqueue :: queue *
         isempty :: queue * -> bool
         head :: queue * -> *
         enqueue :: queue * -> * -> queue *
         dequeue :: queue * -> queue *

queue * == [*]
emptyqueue = []
isempty [] = True
isempty (x:xs) = False
head (x:xs) = x
enqueue q x = q ++ [x]
dequeue (x:xs) = xs
```


Using the `queue`

Can use `queue` as follows:

```
q1 = enqueue (enqueue emptyqueue 1) 2

q2 = dequeue q1

x = head q1
```

But not:

```
q3 = dequeue [1,2,3]
```

Persistent Queues Revisited

This is *Queue 2*. Recall, this queue implementation is inefficient. Why?

We can instead implement queue like *Queue 3*:

```
queue * == ([*],[*])                    || (front,rear)
emptyqueue = ([],[])
isempty ([],[]) = True
isempty q = False
head ([x:front],rear) = x
enqueue (front,rear) x = make_proper_queue (front,x:rear)
dequeue (x:front,rear) = make_proper_queue (front,rear)

make_proper_queue ([],x:rear) = (reverse (x:rear),[])
make_proper_queue q = q
```


Persistent Usage

```
|| create queue with 1..100 in it
q = foldl enqueue emptyqueue [1..100]

|| access second element.
head (dequeue q)
head (dequeue q)
head (dequeue q)
head (dequeue q)
```

What is cost?

Recall: Get \$2 for `make_proper_queue` and pay \$$n$ for `reverse`.

Memoization

Consider:

```
<some-code>
   where
      x = <some-expression>
```

Because Miranda is lazy, `x` will only be evaluated if needed. When `x` is first evaluated, its value will be *memoized*.

Subsequent references to `x` will not re-evaluate `<some-expression>`. Instead, they will return `x`'s memoized value.

Lazyness & Memoization

What is the runtimes of the following?

1. `reverse [1..n] ! 0`
2. `([1,2,3] ++ reverse [1..n]) ! 0`
3. `([1,2,3] ++ reverse [1..n]) ! 3`

Here `reverse` is expensive.

We can use this idea to improve our queue implementation.

Queue 4

Instead of copying `reverse rear` to `front` when `front` is empty, copy it when `length rear > length front`.

Since length is expensive to compute, store lengths in queue tuple.

```
queue * == (num,[*],num,[*])            || (fl,front,rl,rear)
emptyqueue = (0,[],0,[])
isempty (0,[],0,[]) = True
isempty q = False
head (fl,(x:front),rl,rear) = x

enqueue (fl,front,rl,rear) x
       = make_proper_queue (fl,front,rl+1,x:rear)

dequeue (fl,x:front,rl,rear)
       = make_proper_queue (fl-1,front,rl,rear)

make_proper_queue (fl,front,rl,rear)
       = (fl+rl, front ++ reverse rear,0,[]), if rl > fl
make_proper_queue q = q
```


Amortized Analysis

What part of `make_proper_queue` is expensive?

It is `reverse`.

For queue length n, must call `dequeue` $n/2$ times before reverse happens because it is lazy!

Charge \$2 for `make_proper_queue` call.
Pay \$1 for no-op call.
Pay \$$k$ for call to reverse of k item list.

We must have $n/2$ no-op calls before we call `reverse`.

A persistent queue with amortized $O(1)$ cost per operation!!