Lecture 12

Turner's algorithm

- **Define:**

$$\texttt{LET}\ \texttt{S}' = \lambda c\ f\ g\ x.\ c\ (f\ x)\ (g\ x)$$

- $\texttt{S}'$ **has the reduction rule:**

$$\texttt{S}'\ C\ E_1\ E_2\ E_3 \underset{\text{c}}{\longrightarrow} C\ (E_1\ E_3)\ (E_2\ E_3)$$

(C, E_1, E_2, E_3 are arbitrary combinatory expressions)

- **If C is a combinator (i.e. contains no variables), then for any E_1 and E_2:**

$$\lambda^* x.\ C\ E_1\ E_2 = \texttt{S}'\ C\ (\lambda^* x.\ E_1)\ (\lambda^* x.\ E_2)$$

  - **this is shown using extensionality**

$$\boxed{\lambda^* x.\ C\ E_1\ E_2 = \mathtt{S}'\ C\ (\lambda^* x.\ E_1)\ (\lambda^* x.\ E_2)}$$

- x **not in** $\lambda^* x.\ C\ E_1\ E_2$ **or** $\mathtt{S}'\ C\ (\lambda^* x.\ E_1)\ (\lambda^* x.\ E_2)$
- **so is sufficient to show:**

$$(\lambda^* x.\ C\ E_1\ E_2)\ x = (\mathtt{S}'\ C\ (\lambda^* x.\ E_1)\ (\lambda^* x.\ E_2))\ x$$

- **from definition of** $\lambda^* x$ **it easily follows that:**

$$\lambda^* x.\ C\ E_1\ E_2 = \mathtt{S}\ (\mathtt{S}\ (\mathtt{K}\ C)\ (\lambda^* x.\ E_1))\ (\lambda^* x.\ E_2)$$

- **hence**

$$\begin{aligned}(\lambda^* x.\ C\ E_1\ E_2)\ x &= (\mathtt{S}\ (\mathtt{S}\ (\mathtt{K}\ C)\ (\lambda^* x.\ E_1))\ (\lambda^* x.\ E_2)\ x \\ &= \mathtt{S}\ (\mathtt{K}\ C)\ (\lambda^* x.\ E_1)\ x\ ((\lambda^* x.\ E_2))\ x) \\ &= \mathtt{K}\ C\ x\ ((\lambda^* x.\ E_1)\ x)\ ((\lambda^* x.\ E_2)\ x) \\ &= C\ ((\lambda^* x.\ E_1)\ x)\ ((\lambda^* x.\ E_2))\ x)\end{aligned}$$

- **but**

$$\mathtt{S}'\ C\ (\lambda^* x.\ E_1)\ (\lambda^* x.\ E_2)\ x = C\ ((\lambda^* x.\ E_1)\ x)\ ((\lambda^* x.\ E_2))\ x$$

- **so:** $(\lambda^* x.\ C\ E_1\ E_2)\ x = (\mathtt{S}'\ C\ (\lambda^* x.\ E_1)\ (\lambda^* x.\ E_2))\ x$

Description of Turner's algorithm

- $\mathtt{S}'$ **can improve translation of** $\lambda V_n \ \cdots \ V_2\ V_1.\ E_1\ E_2$

- **Define:**

$$\begin{array}{lll} E' & \textbf{to mean} & \lambda^* V_1.\ E \\ E'' & \textbf{to mean} & \lambda^* V_2.\ (\lambda^* V_1.\ E) \\ E''' & \textbf{to mean} & \lambda^* V_3.\ (\lambda^* V_2.\ (\lambda^* V_1.\ E)) \\ \quad\vdots & & \end{array}$$

- **Recall that:**

$$\begin{array}{l} (\lambda V_n\ \cdots\ V_2\ V_1.\ E_1\ E_2)_{\mathtt{C}} = \\ \quad \lambda^* V_n.\ (\ \cdots\ (\lambda^* V_2.\ (\lambda^* V_1.\ (E_1\ E_2)_{\mathtt{C}})))\ \cdots\ ) \end{array}$$

- **Size of** $\lambda^* V_n.\ \ldots \lambda^* V_2.\ \lambda^* V_1.\ (E_1\ E_2)$ **quadratic in** n

  (i) $\lambda^* x_1.\ E_1\ E_2 = \mathtt{S}\ E_1'\ E_2'$

  (ii) $\lambda^* x_2.\ (\lambda^* x_1.\ E_1\ E_2) = \mathtt{S}\ (\mathtt{B}\ \mathtt{S}\ E_1'')\ E_2''$

  (iii) $\lambda^* x_3.\ (\lambda^* x_2.\ (\lambda^* x_1.\ E_1\ E_2)) = \mathtt{S}\ (\mathtt{B}\ \mathtt{S}\ (\mathtt{B}\ (\mathtt{B}\ \mathtt{S})\ E_1'''))\ E_2'''$

  (iv) $\lambda^* x_4.\ (\lambda^* x_3.\ (\lambda^* x_2.\ (\lambda^* x_1.\ E_1\ E_2))) =$
  $\mathtt{S}\ (\mathtt{B}\ \mathtt{S}\ (\mathtt{B}\ (\mathtt{B}\ \mathtt{S})\ (\mathtt{B}\ (\mathtt{B}\ (\mathtt{B}\ \mathtt{S})))\ E_1''''))\ E_2''''$

Quadratic to linear

- **Size of $\lambda^* V_n. \ \ldots \lambda^* V_2. \ \lambda^* V_1. \ (E_1 \ \ E_2)$ is proportional to the** *square* **of** n

  - **using $\mathtt{S}'$, the size can be made to grow** *linearly*

$$
\begin{aligned}
\lambda^* x_2.\ (\lambda^* x_1.\ E_1\ E_2) &= \lambda^* x_2.\ \mathtt{S}\ E_1'\ E_2' \\
&= \mathtt{S}'\ \mathtt{S}\ (\lambda^* x_2.\ E_1')\ (\lambda^* x_2.\ E_2') \\
&= \mathtt{S}'\ \mathtt{S}\ E_1''\ E_2''
\end{aligned}
$$

$$
\begin{aligned}
\lambda^* x_3.\ (\lambda^* x_2.\ (\lambda^* x_1.\ E_1\ E_2)) &= \lambda^* x_3.\ \mathtt{S}'\ \mathtt{S}\ E_1''\ E_2'' \\
&= \mathtt{S}'\ (\mathtt{S}'\ \mathtt{S})\ (\lambda^* x_3.\ E_1'')\ (\lambda^* x_3.\ E_2'') \\
&= \mathtt{S}'\ (\mathtt{S}'\ \mathtt{S})\ E_1'''\ E_2'''
\end{aligned}
$$

$$
\begin{aligned}
\lambda^* x_4.\ (\lambda^* x_3.\ (\lambda^* x_2.\ (\lambda^* x_1.\ E_1\ E_2))) &= \lambda^* x_4.\ \mathtt{S}'\ (\mathtt{S}'\ \mathtt{S})\ E_1'''\ E_2''' \\
&= \mathtt{S}'\ (\mathtt{S}'\ (\mathtt{S}'\ \mathtt{S}))\ (\lambda^* x_4.\ E_1''')\ (\lambda^* x_4.\ E_2''') \\
&= \mathtt{S}'\ (\mathtt{S}'\ (\mathtt{S}'\ \mathtt{S}))\ E_1''''\ E_2''''
\end{aligned}
$$

$\mathtt{B}'$ and $\mathtt{C}'$

- $\mathtt{B}$ and $\mathtt{C}$ simplify expressions $\mathtt{S}\ (\mathtt{K}\ E_1)\ E_2$ and $\mathtt{S}\ E_1\ (\mathtt{K}\ E_2)$

- $\mathtt{B}'$ and $\mathtt{C}'$ have analogous role for $\mathtt{S}'$

- Required properties:

$$\begin{aligned} \mathtt{S}'\ C\ (\mathtt{K}\ E_1)\ E_2 &= \mathtt{B}'\ C\ E_1\ E_2 \\ \mathtt{S}'\ C\ E_1\ (\mathtt{K}\ E_2) &= \mathtt{C}'\ C\ E_1\ E_2 \end{aligned}$$

  (C any combinator, E_1, E_2 arbitrary

- Achieved by defining:

$$\mathtt{LET}\ \mathtt{B}' = \lambda c\ f\ g\ x.\ c\ f\ (g\ x)$$

$$\mathtt{LET}\ \mathtt{C}' = \lambda c\ f\ g\ x.\ c\ (f\ x)\ g$$

Properties of $\mathtt{B}'$ and $\mathtt{C}'$

$$\mathtt{LET}\ \mathtt{B}' = \lambda c\ f\ g\ x.\ c\ f\ (g\ x)$$

$$\mathtt{LET}\ \mathtt{C}' = \lambda c\ f\ g\ x.\ c\ (f\ x)\ g$$

- **For arbitrary λ-expressions C, E_1, E_2 and E_3:**

$$\mathtt{B}'\ C\ E_1\ E_2\ E_3 \underset{\mathrm{c}}{\longrightarrow} C\ E_1\ (E_2\ E_3)$$
$$\mathtt{C}'\ C\ E_1\ E_2\ E_3 \underset{\mathrm{c}}{\longrightarrow} C\ (E_1\ E_3)\ E_2$$

- **For arbitrary λ-expressions E_1, E_2 and E_3:**

(i) $\mathtt{S}'\ E_1\ (\mathtt{K}\ E_2)\ E_3 = \mathtt{B}'\ E_1\ E_2\ E_3$

(ii) $\mathtt{S}'\ E_1\ E_2\ (\mathtt{K}\ E_3) = \mathtt{C}'\ E_1\ E_2\ E_3$

(iii) $\mathtt{S}\ (\mathtt{B}\ E_1\ E_2)\ E_3 = \mathtt{S}'\ E_1\ E_2\ E_3$

(iv) $\mathtt{B}\ (E_1\ E_2)\ E_3 = \mathtt{B}'\ E_1\ E_2\ E_3$

(v) $\mathtt{C}\ (\mathtt{B}\ E_1\ E_2)\ E_3 = \mathtt{C}'\ E_1\ E_2\ E_3$

Turner's description of his algorithm

- **Turner describes has algorithm as follows:**

  Use the algorithm of Curry but whenever a term beginning in $\mathtt{S}$, $\mathtt{B}$ or $\mathtt{C}$ is formed use one of the following transformations if it is possible to do so

  $$\mathtt{S}\ (\mathtt{B}\ K\ A)\ B \longrightarrow \mathtt{S}'\ K\ A\ B,$$

  $$\mathtt{B}\ (K\ A)\ B \longrightarrow \mathtt{B}'\ K\ A\ B,$$

  $$\mathtt{C}\ (\mathtt{B}\ K\ A)\ B \longrightarrow \mathtt{C}'\ K\ A\ B.$$

  Here A and B stand for arbitrary terms as usual and K is any term composed entirely of constants. The correctness of the new algorithm can be inferred from the correctness of the Curry algorithm by demonstrating that in each of the above transformations the left- and right-hand sides are extensionally equal. In each case this follows directly from the definitions of the combinators involved.

- **EXERCISE: justify this algorithm**

Towards $\mathtt{Y}$ in ML: call-by-value $\mathtt{Y}$

- **Recall $\mathtt{Y}$:**

$$\mathtt{LET}\ \mathtt{Y} = \lambda f.\ (\lambda x.\ f(x\ x))\ (\lambda x.\ f(x\ x))$$

- **$\mathtt{Y}$ doesn't work with call-by-value**

- **It goes into a loop:**

$$\begin{aligned}\mathtt{Y}\ f &\longrightarrow f(\mathtt{Y}\ f)\\ &\longrightarrow f(f(\mathtt{Y}\ f))\\ &\longrightarrow f(f(f(\mathtt{Y}\ f)))\\ &\ \vdots\end{aligned}$$

- **Define:**

$$\mathtt{LET}\ \hat{\mathtt{Y}} = \lambda f.\ (\lambda x.\ f(\lambda y.\ x\ x\ y))\ (\lambda x.\ f(\lambda y.\ x\ x\ y))$$

  - **$\hat{\mathtt{Y}}$ is $\mathtt{Y}$ with "$x\ x$" η-converted to "$\lambda y.\ x\ x\ y$"**

- **$\hat{\mathtt{Y}}$ doesn't goes into a loop with call-by-value:**

$$\hat{\mathtt{Y}}\ f \longrightarrow f(\lambda y.\ \hat{\mathtt{Y}}\ f\ y)$$

  - **call-by-value doesn't evaluate λs**

$\hat{\text{Y}}$ in ML

```
val Y = fn f => (fn x => (f (fn y => x x y)))
                (fn x => (f (fn y => x x y)));
> Type clash  in:  (x x)
> Looking  for a:  'a
> I have found a:  'a -> 'b
```

- **To avoid type clash need to solve equation:**

```
'a = 'a -> 'b
```

```
datatype 'a t = T of 'a t -> 'a;
> con T = Fn : (('a t) -> 'a) -> ('a t)
```

- **Now write "`x (T x)`" instead of "`x x`"**

```
val Y =
 fn f =>    (fn (T x) => (f (fn a => x (T x) a)))
         (T (fn (T x) => (f (fn a => x (T x) a))))
> val Y =
>  Fn : (('a -> 'b) -> ('a -> 'b)) -> ('a -> 'b)

val Fact =
 Y(fn f => fn n => if n=0 then 1 else n*(f(n-1)));
> val Fact = Fn : int -> in

Fact 6;
> 720 : int
```

Object oriented programming in ML

- **Object oriented (OO) programming is very trendy**

- **There is a debate about whether functional languages like ML can provide benefits of OO**

  - **lots of research on translating between OO calculi and λ-calculi**
  - **many of the issues involve types**
  - **types for OO are hard to get right**

- **See the paper**

  - *Object-Oriented Programming and Standard ML*
  - **by Lars Thorup and Mads Tofte**
  - **available from the course web page**

Objects

- **OO programming is based on objects**
- **Objects can be created**
  - **from** *classes*
  - **directly**
  - **depends on flavour of OO**
- **Objects are self-contained units packaging together**
  - **state – e.g. values of local variables**
  - **operations (methods) for**
    - **accessing state**
    - **modifying state**

Classes and subtyping

- **Objects are organised hierarchically**
  - **one object may extend another object by adding new methods**
  - **a** *coloured point* **extends a** *point* **with extra colour state**
  - *coloured point* **has all the methods of** *point* **plus more**
    - **subtyping**
- **Objects can easily be represented in ML**
  - **but harder to represent subtyping and classes**

A point object

- **A point object has**
  - **local state variables giving x and y coordinates**
  - **methods:**
    - `move` **to move the point – changes position**
    - **getx – gets x coordinate**
    - **gety – gets y coordinate**
- **Represent a point by a record of its methods**

```
type Point = {move : int * int -> unit,
              getx : unit -> int,
              gety : unit -> int}
```

- **Methods represented by**
  - **local functions:** `move_fn`, `getx_fn`, `gety_fn`
  - **acting on shared local variables:** `x`, `y`

NewPoint in ML

```
fun NewPoint (newx,newy) : Point =
 let val (x,y) = (ref newx, ref newy);
     fun move_fn(newx,newy) = (x:= newx; y := newy);
     fun getx_fn () = !x;
     fun gety_fn () = !y;
 in
  {move = move_fn, getx = getx_fn, gety = gety_fn}
 end;
> val NewPoint = fn : int * int -> Point

val p = NewPoint(2,3);
> val p = {getx=fn,gety=fn,move=fn} : Point

#getx p ();
> val it = 2 : int

#gety p ();
> val it = 3 : int

#move p (5,9);
> val it = () : unit

(#getx p (), #gety p ());
> val it = (5,9) : int * int
```

Conclusions

- **I'll conclude with a summary of the course**
- **The examinable material is what was covered in the lectures**
  - **stuff in notes that's not in lectures is not examinable**
  - **e.g. SECD machine**
- **Don't forget to return your evaluation form to Eileen Murray!**

Overview of ML

- Expressions
- Declarations
- Comments
- Functions
- Type abbreviations
- Operators
- Lists
- Strings
- Records
- Polymorphism
- `fn`-expressions
- Conditionals
- Recursion
- Equality types
- Pattern matching
- The `case` construct
- Exceptions
- Datatype declarations
- Abstract types
- Type constructors
- References and assignment
- Iteration

Introduction to the λ-calculus

- **Syntax and semantics of the λ-calculus**
- **Notational conventions**
- **Free and bound variables**
- **Conversion rules**
  - α**-conversion**
  - β**-conversion**
  - η**-conversion**
  - **Generalized conversions**
- **Equality of λ-expressions**
- **The $\longrightarrow$ relation**
- **Extensionality**
- **Substitution**

Representing Things in the λ-calculus

- **Truth-values and the conditional**
- **Pairs and tuples**
- **Numbers**
- **Definition by recursion**
- **Functions with several arguments**
- **Mutual recursion**
- **Representing the recursive functions**
    - **The primitive recursive functions**
        - **Substitution**
        - **Primitive recursion**
    - **The recursive functions**
        - **Minimization**
        - **Higher-order primitive recursion**
    - **The partial recursive functions**
- **Extending the λ-calculus**
- **Theorems about the λ-calculus**
- **Call-by-value and Y**

Combinators

- Combinator reduction
- Functional completeness
- Reduction machines
- Improved translation to combinators
- More combinators
- Curry's algorithm
- Turner's algorithm