Programming Languages

Lecture 4: Functional Programming Languages (SML)

Benjamin J. Keller
Department of Computer Science, Virginia Tech
Lecture Outline

- Overview
- Primitive Data Types
- (Built-in) Structured Data Types
- Pattern Matching
- Type Inference
- Polymorphism
- Declarations
- Examples
Lecture Outline

- Exceptions
- Lazy vs. Eager Evaluation
- Higher Order Functions
- Program Correctness
- Imperative Language Features
- Implementation
- Efficiency
- Concurrency
- Summary
Overview of ML

- Developed in Edinburgh in late 1970’s
- Meta-Language for automated theorem proving system
- Designed by Robin Milner, Mike Gordon, Chris Wadsworth
- Found useful and extended to programming language
Functional Programming in ML

- Functional programs are made up of functions applied to data
- We write expressions rather than commands
- Pure functional languages have no side effects
- ML is not a pure language
  - reference variables
  - commands
  - I/O
ML Characteristics

- Functions as first class values
- Statically scoped
- Static typing via type inference
- Polymorphic types
- Type system includes support for ADTs
- Exception handling
- Garbage collection
Using ML Interpreter

- Type `sml`

  Standard ML of New Jersey, Version 110.0.3, January 30, 1998

- Hyphen (-) is prompt

- Can load definitions from file named `myfile.sml`

  use "myfile.sml";

- End session by typing `ctrl-d`
Expressions

- Expression evaluation
  - 3;
  val it = 3 : int
  - 23 - 6;
  val it = 17 : int

- Name it refers to last value computed
Constants

- In ML we name values rather than have variables:
  
  - val pi = 3.14159;
  val pi = 3.14159 : real
  - val r = 2.0;
  val r = 2.0 : real
  - val area = pi * r * r;
  val area = 12.56636 : real

- A name can be rebound

  - val area = "pi r squared";
  val area = "pi r squared" : string
Functions

- Syntax: `fun name arg = expression`

- Example
  
  - `fun area(r) = pi*r*r;`
  
  `val area = fn : real -> real`

- Parenthesis optional for single argument

- Can also write function as a value

  - `val area = fn r => pi * r * r;`
  
  `val area = fn : real -> real`
Function applications

- area 2.0;
val it = 12.56636 : real
- area(2.0);
val it = 12.56636 : real
Environment

- pi defined outside of area
  
  ```sml
  val pi = 3.14159;
  fun area(r) = pi*r*r;
  ```

- What happens if change pi?
  
  ```sml
  - area 1.1;
  val it = 3.8013239 : real
  - val pi = 2000;
  val pi = 2000 : int
  - area 1.1;
  val it = 3.8013239 : real
  ```

- Environment of function determines value
Primitive Data Types

- **unit** — has one value: ()

- **bool**
  - values: true, false
  - operators: not, andalso, orelse

- **int**
  - values: positive and negative integers (\ldots \sim 2, \sim 1, 0, 1, 2, \ldots).
  - operators: +, -, *, div, mod, <, <=, >, >=, <>
Primitive Data Types (cont)

- real
  - values: real numbers 3.1, 2.4E100
  - operators: +, -, *, /, <, <=, >, >=, <>, log, exp, sin, arctan

- string
  - values: ”a string”, uses special characters \t, \n
- operators: ^ (concatenation), length, substring
## Type Inference and Overloading

- ML attempts to infer type from values of expressions
- Some operators overloaded (+, *, -)
- Inferred type may not be what you want
  - `fun double x = x + x;`
  - `val double = fn : int -> int`
- Sometimes ML can’t determine type
- Force type with type constraints
  - `fun double x:real = x + x;`
  - `fun double (x):real = x + x;`
  - `fun double (x:real):real = x + x;`
  - `has type fn : real -> real`
Structured Data Types

- Tuples — ordered collection of values
- Records — collection of named values
- Lists — list of values of homogeneous type
• Syntax: ( exp-list )
  - (1, 2, 3);
  val it = (1, 2, 3) : int * int * int
  - (pi, r, area);
  val it = (3.14159, 2.0, fn) : real * real * (real -> real)

• Access by pattern matching or by label
  - val (a, b) = (2.3, "zippy");
  val a = 2.3 : real
  val b = "zippy" : string
  - #3 (a, b, pi);
  val it = 3.14159 : real
Multi-Argument Functions

- Argument of a function can be a tuple
  
  - fun mult (x,y) = x*y;
  
  val mult = fn : int * int -> int
  
  - fun mult (t : int*int) = #1 t * #2 t; (* ugly! *)
  
  val mult = fn : int * int -> int
Curried Functions

- Function with two arguments
  
  ```ml
  fun power(m,n) : int =
  = if n = 0 then 1
  = else m * power(m,n-1);
  val power = fn : int * int -> int
  ```

- Equivalent function
  
  ```ml
  fun cpower m n : int =
  = if n = 0 then 1
  = else m * cpower m (n-1);
  val cpower = fn : int -> int -> int
  ```
• power and cpower different functions, but
  - power(2,3);
    val it = 8 : int
  - cpower 2 3;
    val it = 8 : int

• Function cpower is “Curried” (Haskell Curry)

• Can define new functions by partial evaluation
  - val power_of_two = cpower 2;
    val power_of_two = fn : int -> int
  - power_of_two 3;
    val it = 8 : int
Records

- A collection of labeled data items
  - val ex = { name = "george", userid = 12 };  
  val ex = {name="george",userid=12} :  
  {name:string, userid:int}

- Access elements by pattern matching or label
  - #name ex;  
  val it = "george" : string  
  - val {name=username, ...} = ex;  
  val username = "george" : string

- Tuples shorthand for records with labels 1, 2, ....
Lists

• All elements must be of same type
  - [ 2, 6, 4, 9];
  val it = [2,6,4,9] : int list
  - [ "a", "b", "c" ];
  val it = ["a","b","c"] : string list
  - [ 1, "a" ];
  ... Error: operator and operand don’t agree
  operator domain: int * int list
  operand: int * string list
  in expression:
  1 :: "a" :: nil
Lists Constructors

- `[]`, `nil` — empty list (all types)
- `::` — `cons` operator
  - `1 :: []`;
  - `val it = [1] : int list`
  - `1 :: (2 :: [2])`;
  - `val it = [1,2,2] : int list`
Functions on Lists

- **length**
- **Head and tail**
  - `hd [ 3, 4];`
  - `val it = 3 : int`
- `tl [3, 4];`
  - `val it = [4] : int list`
- **Concatenation**
  - `[ 1, 2] @ [3, 4];`
  - `val it = [1,2,3,4] : int list`
- **rev** — reverse list
Map Function

- `map` applies another function to all elements of a list
  - `fun sqr x = x * x;
  val sqr = fn : int -> int
  - `map` `sqr` [2,3,4,5];
  val `it` = [4,9,16,25] : int list

- Example of *polymorphic* and *higher order* function
  - `map`;
  val `it` = fn : (’a -> ’b) -> ’a list -> ’b list
Pattern Matching

- Pattern matching important in ML
- Used to bind variables
  - \[ \text{val } (x,y) = (5 \text{ div } 2, 5 \text{ mod } 2); \]
  - \[ \text{val } x = 2 : \text{int} \]
  - \[ \text{val } y = 1 : \text{int} \]
  - \[ \text{val } \{a = x, b = y\} = \{b = 3, a = "one"\}; \]
  - \[ \text{val } x = "one" : \text{string} \]
  - \[ \text{val } y = 3 : \text{int} \]
Pattern Matching

- Pattern matching on lists
  
  - val head::tail = [1,2,3];
  stdin:67.1-67.25 Warning: binding not exhaustive
    head :: tail = ...
  val head = 1 : int
  val tail = [2,3] : int list

  - val head::_ = [4,5,6]; (* "_" wildcard *)
  stdin:69.1-69.22 Warning: binding not exhaustive
    head :: _ = ...
  val head = 4 : int
Pattern Matching in Functions

- Can do pattern matching in functions

  fun product [] : int = 1
  | product (h::t) = h * product t;

- May use different types like integers

  - fun oneTo 0 = []
  = | oneTo n = n::(oneTo(n-1));
  val oneTo = fn : int -> int list
  - oneTo 5;
  val it = [5,4,3,2,1] : int list

- Example (definition of reverse)

  fun reverse [] = []
  | reverse (h::t) = reverse(t) @ [h];
Aside: Function Composition

- Can define factorial as
  
  ```
  fun fact n = product (oneTo n);
  ```

- Equivalent to writing
  
  ```
  val fact = product \ o \ oneTo;
  ```

- The operator \( \circ \) is function composition
Type Inference

- ML determines types of expressions or functions
- Don’t have to declare types except to disambiguate types
  - val x = 3.2;
  val x = 3.2 : real
  - fun addx y = x + y;
  val addx = fn : real -> real
- Language strongly typed
Polymorphic Functions

- **Polymorphism** — many “forms” (types)

- A function
  
  ```
  fun last [x] = x
  | last (h::t) = last t;
  ```

  has type \( \text{fn} : \ 'a \ \text{list} \rightarrow \ 'a \)

- Symbol \( 'a \) is a type variable

- Type variables for types with equality have form \( ’’a \)
  
  ```
  fun search item [] = false
  | search item (fst::rest) =
    if item = fst then true else search item rest;
  ```

  has type \( \text{fn} : \ ’’a \rightarrow \ ’’a \ \text{list} \rightarrow \ \text{bool} \)
Function and value declarations at the top level stay visible until a new definition of same identifier

- `val x = 3 * 3;`
- `val x = 9 : int`
- `val it = 18 : int`
Local Declarations

- Declarations within functions
- Syntax: `let decl in exp end`

```sml
fun fact n =
  let
    fun facti(n,p) =
      if n = 0 then p
      else facti(n-1,n*p);
  in
    facti (n,1)
  end;
```
- Allows naming intermediate values
Hiding Declarations

- Declarations can be hidden with `local`
- Syntax: `local decl in decl-list end`

```sml
local
    fun facti(n,p) =
        if n = 0 then p
        else facti(n-1,n*p);
    in
    fun fact n = facti(n,1);
end;
```

- Can declare several functions
Order of Evaluation

- Evaluate operand, substitute operand value for formal parameter, and evaluate
- Inside record, evaluate fields from left to right
- Inside let expression `let decl in exp end`
  1. evaluate decl producing new environment
  2. evaluate exp in new environment
  3. restore old environment
  4. return computed value of exp
Declarations

- **Sequential Declarations**
  - `val x = 12;
  val x = 12 : int`  
  - `val y = x + 2;
  val y = 14 : int`  

- **Parallel (Simultaneous) Declarations**
  - `val x = 2 and y = x + 3;
  val x = 2 : int`  
  - `val y = 15 : int`
Mutual Recursion

- Example: take alternate elements

\[
\text{fun take } [] = [] \\
\quad | \text{take } (h::t) = h::(\text{skip } t) \\
\text{and skip } [] = [] \\
\quad | \text{skip } (h::t) = \text{take } t;
\]

- Output

- \text{take } [1,2,3,4,5,6];
  \text{val it = } [1,3,5] : \text{int list}
- \text{skip } [1,2,3,4,5,6];
  \text{val it = } [2,4,6] : \text{int list}
Recursive Functions

- Recursion is the norm in ML
  
  - fun fact n =
    = if n=0 then 1 else n * fact(n-1);
  
  val fact = fn : int -> int
  
  - fact 7;
  
  val it = 5040 : int

- Tail recursive functions more efficient
  
  - fun facti(n,p) =
    = if n=0 then p else facti(n-1,n*p);
  
  val facti = fn : int * int -> int

- But not necessarily practical
local
  fun partition (pivot, nil) = (nil, nil)
  | partition (pivot, h :: t) =
    let val (smalls, bigs) = partition(pivot, t)
    in
      if h < pivot then (h :: smalls, bigs)
      else (smalls, h :: bigs)
    end;
  end;

in
  fun qsort nil = nil
  | qsort [singleton] = [singleton]
  | qsort (h :: t) =
    let val (smalls, bigs) = partition(h, t)
    in qsort(smalls) @ [h] @ qsort(bigs)
    end;
  end;
Polymorphic Quicksort

local
   fun partition (pivot, nil) (lessThan) = (nil,nil)
   | partition (pivot, first :: others) (lessThan) =
      let val (smalls, bigs) = partition(pivot,others) (lessThan)
      in
         if (lessThan first pivot) then (first::smalls,bigs)
         else (smalls,first::bigs)
      end;
   in
      fun qsort nil lessThan = nil
      | qsort [singleton] lessThan = [singleton]
      | qsort (first::rest) lessThan =
         let
            val (smalls, bigs) = partition(first,rest) lessThan
         in
            (qsort smalls lessThan) @ [first] @ (qsort bigs lessThan)
         end;
      end;
end;
Using Polymorphic QuickSort

- Define comparison function
  
  ```sml
  fun intLt (x:int) y = x < y;
  ```

- Must be curried: (why?)
  
  ```sml
  val intLt = fn : int -> int -> bool
  ```

- Application
  
  ```sml
  - qsort [9,1,6,3,4,7,5,8,2,10] intLt;
  val it = [1,2,3,4,5,6,7,8,9,10] : int list
  ```
Fibonacci

- Obvious Fibonacci function slow
- Iterative solution faster

```c
int fastfib(int n) {
    int a = 1, b = 1;
    while (n > 0) {
        a = b; b = a + b; n--; /* could be parallel */
    }
    return a;
}
```

- Equivalent ML

```ml
fun fastfib n : int = 
    let
        fun fibLoop a b 0 = a
         | fibLoop a b n:int = fibLoop b (a+b) (n-1)
    in fibLoop 1 1 n
end;
```
Declaring Types

- **type** defines a new name for a type
  
  - type username = {name:string, userid:int};
  
  type username = {name:string, userid:int}

- May be needed to constrain function types
  
  - fun nme user = #name user;
  stdIn:1.1-35.5 Error: unresolved flex record
    (can’t tell what fields there are besides #name)
  
  - fun nme(user:username) = #name user;
  val nme = fn : username -> string

- A polymorphic type
  
  type ’a pair = ’a * ’a
Concrete Data Types

- Ways of declaring types of data structures
- Enumerated types
  
  \[
  \text{datatype ulevel} = \\
  \text{Freshman} | \text{Soph} | \text{Junior} | \text{Senior};
  \]
  
  \[
  \text{datatype glevel} = \text{MS} | \text{PhD};
  \]

- More general types
  
  \[
  \text{datatype student} = \text{Undergrad of ulevel};
  \]
  
  \[
  | \text{Grad of int} * \text{glevel};
  \]

- Undergrad and Grad are constructors
Pattern Matching

- Functions

  fun level Undergrad(_) = "An undergrad"
  | level Grad(_,MS) = "An MS student"
  | level Grad(_,PhD) = "A PhD student"

- Case Expressions

  (case s of
   Undergrad(_) = "An undergrad"
   | Grad(_,MS) = "An MS student"
   | Grad(_,PhD) = "A PhD student"
Recursive Types

- Can define types that use each other
  - datatype s = a of t
  = and t = b of s | c;
  datatype s = a of t
datatype t = b of s | c
  - a(b(a c));
  val it = a (b (a c)) : s

- Useful when have two types that can contain the other
Polymorphic Types

- Name of type preceded by a type variable

  datatype 'a notmuch = Nothing
  | Something of 'a;
  datatype ('a,'b)sum = In1 of 'a | In2 of 'b;

- To use just use constructors and some value

  - In1 1;
  val it = In1 1 : (int,'a) sum
  - Something "me";
  val it = Something "me" : string notmuch
Aside: Structure Sharing

- Updating of data structures uses sharing
  - fun updatehd nh [] = [nh]
    | updatehd nh (h::t) = nh :: t;
  = val updatehd = fn : 'a -> 'a list -> 'a list
  - val l = [1,2,3];
  val l = [1,2,3] : int list
  - val l2 = updatehd 2 l;
  val l2 = [2,2,3] : int list
  - l;
  val it = [1,2,3] : int list

- Sharing safe because of update policy
Exceptions

- Changes order of execution (used if error detected)
- Declaration like datatype
  
  ```
  exception FailedMiserably;
  exception BadBadMan of string;
  ```
- Raising/throwing exceptions
  
  ```
  raise FailedMiserably;
  ```
- Catching/handling exceptions
  
  ```
  badcall("jimmy")
  handle FailedMiserably => 0
  | BadBadMan(s) => 1;
  ```
Lazy vs Eager Evaluation

- Order of Operations:
  - Eager — Evaluate operand, substitute value for formal parameter, and evaluate expression.
  - Lazy — Substitute operand for formal parameter, evaluate expression, evaluate parameter only when value is needed.

- Lazy evaluation also called *call-by-need* or *normal order* evaluation

- In lazy evaluation each actual parameter either never evaluated or only once.
Lazy vs Eager Example

• Function

```sml
fun test (x:{a:int,b:unit}) =
    if (#a{a=2, b=print("A")} = 2)
        then (#a x)
        else (#a x);
```

• Evaluation

```sml
test {a = 7, b = print("B")};
```

• Eager evaluation:

```sml
BA val it = 7 : int
```

• Lazy evaluation:

```sml
AB val it = 7 : int
```
Infinite Lists

- Function generates rest of list
  
  ```sml
  fun from n = n :: from (n+1)
  val nats = from 1
  ```

- Rest of list computed as needed (in lazy dialect of ML)
  
  ```sml
  fun nth (1, fst::rest) = fst
  | nth (n, fst::rest) = nth(n-1,rest)
  ```

- `nth 10 nats` builds list up to 10
Why Not?

- Why not use lazy evaluation?
- Eager language easier and more efficient to implement (with current technology)
- If language has side-effects, difficult to know when they will occur
- Many optimizations introduce side-effects
- For concurrent execution often better to evaluate as soon as possible.
Simulating Lazy Evaluation

- Make expression into parameterless function
  
  ```sml
  val x = 3 and y = 5;
  val e = fn () => x*y;
  ```

- Force evaluation by expression `e()`

- Example: eager version
  
  ```sml
  fun f x y = if x = [] then [] else x @ y;
  ```

- Implement parameter with lazy evaluation
  
  ```sml
  fun f' x y' = if x = [] then [] else x @ (y' ());
  ```

- Instead of `f e1 e2` write `f' e1 (fn () => e2)`

- `e2` evaluated only if `x<>[]`
Suspended Lists in Eager Language

datatype 'a susplist =
    Mksl of (unit -> 'a * 'a susplist) | Endsl;

(* add head to front of list *)
fun slCons( newhd, slist) =
    let fun f () = (newhd,slist) in Mksl f end;
exception empty_list;

(* extract head *)
fun slHd Endsl = raise empty_list
    | slHd (Mksl f) = let val (a,s) = f () in a end;

(* extract tail *)
fun slTl Endsl = raise empty_list
    | slTl (Mksl f) = let val (a,s) = f () in s end;
Using Lazy Lists

- From function
  
  ```sml
  fun from n = 
    let fun f() = (n, from(n+1)) in Mksl f end;
  ```

- Infinite list
  
  ```sml
  val nat = from 1;
  val nat = Mksl fn : int susplist 
  - slHd(nat);
  val it = 1 : int
  - slHd(slTl(nat));
  val it = 2 : int
  ```
Higher Order Functions As Glue

- Can construct ‘glue’ with higher order functions
- Example functions
  
  ```sml
  fun prod [] = 1
  | prod (h::t) = h * prod t
  
  fun sum [] = 0
  | sum (h::t) = h + sum t
  ```
- Functions follow same pattern
### Building Higher Order Function

- Function encodes same approach

  ```sml
  fun listify (oper, identity:'a) ([]:'a list) = identity
  | listify (oper, identity) (h::t) =
    oper(h,listify(oper,identity) t);
  ```

- Can be used to build new functions

  ```sml
  val listsum = let fun sum(x,y) = x+y:int
    in listify(sum,0) end;
  val listmult = let fun mult(x,y) = x*y:int
    in listify(mult,1) end;
  val length = let fun add1(x,y) = 1 + y
    in listify(add1,0) end;
  ```
Program Correctness

- Referential transparency makes verification easier
- If have let val I = E in E’ end;
- Then get same value by substituting for I by E in E’ before evaluating
- Can reason that
  
  \[
  \text{let val} \ x = 2 \ \text{in} \ x + x \ \text{end} \\
  = 2 + 2 \\
  = 4
  \]

- Only works if no side effects or lazy evaluation
  
  \[
  \text{let val} \ x = m \ \text{div} \ n \ \text{in} \ 3 \ \text{end};
  \]

- Raises exception if n = 0
Proof Rule

Theorem: Let $E$ be a functional expression (with no side effects). If $E$ converges to a value under eager evaluation, then $E$ converges to the same value with lazy evaluation.
Program Verification

- **Specification**: for every natural number $n$, $facti(n, 1) = n!$
- **Program**:
  
  \[
  \text{fun facti}(n, p) = \begin{cases} 
  p & \text{if } n = 0 \\
  \text{facti}(n-1, n*p) & \text{else}
  \end{cases}
  \]
- **Verification**: show that program meets specification
Proof

- Induction on $n$

- **Base Case:** $\forall p. facti(0, p) = 0! \times p$
  Holds because for arbitrary $p$, $facti(0, p) = p = 1 \times p = 0! \times p$

- **Inductive step:** assume $\forall p. facti(n, p) = n! \times p$
  Show $\forall p. facti(n + 1, p) = (n + 1)! \times p$
  For arbitrary $p$,

  $facti(n + 1, p) = facti(n, (n + 1) \times p)$ [def of facti] 
  
  $= n! \times ((n + 1) \times p)$ [inductive hyp] 
  
  $= (n! \times (n + 1)) \times p$ [associativity] 
  
  $= (n + 1)! \times p$ [def of factorial]
Imperative Features

- Input and Output
- Reference variables
- Assignment operator
- Command sequence
- While loop
Input and Output

- **print** takes string argument
- Structures for builtin types have toString functions
  - `print(Int.toString(1)^"\n");`  
    1
    ```  
    val it = () : unit  
    ```
- Other i/o done with TextIO structure
- Two streams **instream** and **outstream**
- Provides stdIn and stdOut streams
  - `TextIO.inputLine(TextIO.stdIn);`  
    gotta love nested structure references  
    ```  
    val it = "gotta love nested structure references\n" : string  
    ```
- Functions for opening, reading from and writing to text files.
Commands

- Commands are treated differently than other expressions
- Have a return type of `unit` (value is `()`)  
- Command list – has value of last expression
  - `(print("a\n"); 2);`
  a
  val it = 2 : int
  - `(print("a "); print("b\n"));`
  a b
  val it = () : unit
- Can also put command list inside expression part of `let`
Reference Variables

- A reference is basically an address

- `ref` is a built-in constructor for references
  
  ```
  - val p = ref 17;
  val p = ref 17 : int ref
  - p;
  val it = ref 17 : int ref
  ```

- Dereference with `!`
  
  ```
  - !p;
  val it = 17 : int
  ```
Assignment Operator

- Allows value referenced to be changed
  
  ```
  - p := !p + 1;
  val it = () : unit
  - !p;
  val it = 18 : int
  ```
While Loop

- Syntax: while E1 do E2
- Repeat: Evaluate E1, if true then evaluate E2
- Example:

  counter := 1;
  while !counter < 10 do (  
    counter := !counter + 1;
    print(Int.toString(!counter)" ")
  );
## Efficiency

Functional languages historically slower than imperative

- Use of lists instead of arrays — complexity of access time?
- Passing functions as arguments can be expensive. Local variables must be retained — allocate from heap instead of stack.
- Recursion takes more space than iterative. However, new compilers can detect tail recursion and convert to iteration.
- Nondestructive updating results in copying (minimized by structure sharing). Generates more garbage and requires background garbage collection.
- Easy to write programs that pass lists when a single element would suffice.
Program compiled with SML of NJ estimated to be 2 to 5 times slower than equivalent C programs. (SML/NJ uses optimizations like continuations.)

- Difficult to properly compare.
- Lazy evaluation languages slower.
- What about designing alternative computer architectures to support functional languages?
Concurrency

- Motivation for functional languages
- Idea: same program runs on single and multiple processor machines
- Functional results not dependent on order of evaluation
- Explicit synchronization constructs unnecessary
- Can make distributed copies without copies becoming inconsistent
- Can simultaneously evaluate $g(x)$ and $f(x)$ in $h(g(x), f(x))$.
- Architectures
  - Demand driven – request for value fires execution
  - Data driven – presence of operands fires execution
Functional Language Summary

- Functional programming forces different way of thinking about algorithms
- Referential transparency supports reasoning about programs and parallel execution
- Trade-off between loss of imperative control structures and ability to write higher-order control structures
- Trade-off between loss of efficiency and higher-level features that make programming and reasoning about programs easier
- Support for polymorphism improves code reuse
ML Summary

- ML features not discussed
  - Modules, separate compilation
  - Automatic storage management
- ML used in large system projects. (Carnegie Mellon University)
- Current research into extensions