

Lecture 26: Continuations

CSC 131
Fall, 2014

Kim Bruce

Continuations

- Continuation of expression is remaining work to be done after evaluating expression
 - the future
 - Represented as a function, applied to value of exp, which is value computed so far.
- Capture continuation
 - use it later to return to execution.
- Explicitly represented in Scheme, ML
- Have been important in compilers for functional languages, concurrency, web programming

Example 0: Roots of Quadratic

```
exception notQuadratic;  
exception imaginaryRoots;  
fun equal(x:real, y:real) = x <= y andalso x >= y;  
fun roots(a, b, c) = let  
  val discBase = (b*b) - (4.0*a*c)  
  val denom = 2.0*a  
in  
  if equal(denom, 0.0) then raise notQuadratic  
  else if discBase < 0.0 then raise imaginaryRoots  
  else let  
    val disc = Math.sqrt(discBase)  
  in  
    if disc > 0.0 then [(-b + disc)/denom, (-b - disc)/denom]  
    else [-b/denom]  
  end  
end;
```

Example 1: Roots of Quadratic

```
fun checkQuad (x, n_continuation, e_continuation) =  
  if equal(x, 0.0) then e_continuation() else n_continuation(x);  
fun roots1(a, b, c) =  
  let fun econt () = raise notQuadratic  
      fun ncont (x) =  
        let val discBase = (b*b) - (4.0*a*c)  
        in  
          if discBase < 0.0 then raise imaginaryRoots  
          else let val disc = Math.sqrt(discBase)  
              in  
                if disc > 0.0 then [(-b + disc)/x, (-b - disc)/x]  
                else [-b/x]  
              end  
            end  
        in  
          checkQuad(2.0*a, ncont, econt)  
        end;
```

Example 2: Roots of Quadratic

```
fun checkImaginary(x, n_continuation, e_continuation) =
  if x < 0.0 then e_continuation() else n_continuation(Math.sqrt(x));

fun roots2(a, b, c) =
  let
    fun econt () = raise notQuadratic
    fun ncont(x) =
      let
        fun econt () = raise imaginaryRoots
        fun ncont(disc) =
          if disc > 0.0 then [(-b + disc)/x, (-b - disc)/x]
          else [-b/x]
      in
        checkImaginary(b*b - 4.0*a*c, ncont, econt)
      end
  in
    checkQuad(2.0*a, ncont, econt)
  end;
```

Example 3: Roots of Quadratic

```
fun checkNumRoots(disc, continuation1, continuation2) =
  if disc > 0.0 then continuation1(disc) else continuation2();

fun roots3(a, b, c) =
  let fun econt () = raise notQuadratic
    fun ncont(x) =
      let fun econt () = raise imaginaryRoots
        fun ncont(disc) =
          let fun con1(disc) = [(-b + disc)/x, (-b - disc)/x]
            fun con2 () = [-b/x]
          in
            checkNumRoots(disc, con1, con2)
          end
        in
          checkImaginary(b*b - 4.0*a*c, ncont, econt)
        end
      in
        checkQuad(2.0*a, ncont, econt)
      end;
```

*Final version is
tail call!!*

Continuation-Passing Form

- Continuation-passing form of fcn f of n values is function w/one more -- continuation
- $f_{\text{cpf}}(x_1, \dots, x_n, k) =_{\text{def}} k(f(x_1, \dots, x_n))$
- So $f_{\text{cpf}}(x_1, \dots, x_n, \lambda y. y) = \lambda y. y(f(x_1, \dots, x_n)) = f(x_1, \dots, x_n)$
- Use idea to derive cpf form

Using Continuations

- Used w/multiple threads w/separate stack
 - Blocked thread is represented as ptr to continuation
- CPS transform allows to rewrite programs so no need to ever return!
- Useful in web programming where no state.

Changing Execution Order

Manipulating Evaluation Order

- What if actual params are expensive to evaluate, but aren't always used (and you are in an eager language)?
- Can suspend evaluation of e by replacing it by $\text{Delay}(e) = \text{fn } () \Rightarrow e$.
- Evaluate by $\text{Force}(d) = d()$
- Important to have as macros, not functions!

Summary of Statements

- Progression from goto to higher-level abstractions:
 - Expression \Rightarrow function
 - Statement \Rightarrow procedure
 - control structure \Rightarrow iterator
- Modern control: iterators, exceptions, continuations, delay-force

Programming in the Large

How can we understand large programs we write so that we can get them right and then make modifications to them?

Problems w/Large Programs

- Wulf & Shaw: *Global variables considered harmful*
- Problems:
 - Side effects: executing procedure can change global
 - Indiscriminate access - can't prevent access - may be difficult to make changes later if others use details
 - Screening - may lose access via new declaration of vble (hole in scope!)
 - Aliasing - control shared access to prevent more than one name for reference variables.

Characteristics of Solution

- No implicit inheritance of variables
- Right to access by mutual consent
- Access to structure not imply access to substructure
- Provide different types of access (e.g. read-only)
- Decouple declaration, name access, and allocation of space.
 - Scope independent of where declared,
 - similarly w/allocation of space - like "new"

Abstract Data Types (1970's)

- Already did procedural -- now data!
- Encapsulation:
 - Package data structure and its operations in same unit
 - Data type consists of set of elements + operations
 - constructors, observers, operators.
- Representation hidden
 - representation independence
- Look like built-in types.

Specification

- Definitions should not depend on implementation details.
- Constants, types, variables, and operations
 - Behavior must be specified abstractly
 - pre- and postconditions
 - Axioms and rules: $\text{pop}(\text{push}(S,x)) = S$,
if not empty(S) then $\text{push}(\text{pop}(S), \text{top}(S)) = S$
 - Details of implementation provided elsewhere
 - Data + Operations (+ equations) = Algebra

Implementation

- Details of representation and implementation of operations.
- Details not accessible outside unit.

Design Methodologies

- Top-down design
 - Start w/ high-level procedural specification and successively refine.
- Abstract data types (more bottom-up):
 - Identify abstract types and specify operations.
 - Use high-level types and ops to solve problem.
 - Implement ADT with concrete data type.

Design Methodologies

- Combine:
 - Partition first into modules via ADT's
 - Use top-down w/in ADT's to refine

Language Design Concerns

- Simplicity
- Application of formal techniques to specification and verification
- Minimize lifetime costs

Modules

- Reusable modules:
 - Separate, but not independent compilation
 - Maintain type checking
 - Control over export and import of names

Simula 67

```
class vehicle(weight,maxload);
  real weight, maxload;
begin
  integer licenseno;
  real load;
  Boolean procedure tooheavy;
  tooheavy := weight + load > maxload;
  load := 0;      (* initialization code *)
end

ref(vehicle) rv, pickup;
rv:- new vehicle(2000,2500);
pickup:- rv;      (* assignment via sharing *)
pickup.licenseno := 3747;
pickup.load := pickup.load +150;
if pickup.tooheavy then ...
```

Simula 67

- Derived from Algol 60 for discrete simulations.
- Nygaard and Dahl: Turing award 2001
- Introduced classes and objects
 - No information hiding

Representation Independence & Information Hiding

- Choice of representation doesn't affect computation. E.g., rationals.
- If represent new type in terms of old:
 - Rep may have values not corresponding to new type. E.g., (3,0)
 - Rep may have several values corresponding to same abstract value. E.g., (1,2) and (2,4).
 - Values of new type can be confused w/values of rep type.

ADT values

- Only constructible values count.
- Specified abstractly
 - pop(push(fst,rest)) = rest,
 - top(push(fst,rest)) = fst,
 - empty(EmptyStack) = true,
 - empty(push(fst,rest)) = false
- Avoid previous problems because rep hidden

Clu (1974)

- Cluster is used for ADT's
 - Cluster is a type -- not hold one.
 - Can create numerous objects from one cluster
 - Held as implicit references (like Java)
 - cvt used to go back and forth to representation
- ```
sorted_bag = cluster [t : type] is create, insert, ...
 where t has
 lt, equal : proctype (t,t) returns (bool);
```

## Ada (1980)

```
generic
 length : Natural := 100; -- generic parameters
 type element is private;
package stack is
 type stack is private;
 procedure make_empty (S : out stack);
 procedure push (S : in out stack; X : in element);
 procedure pop (S : in out stack; X : out element);
 function empty (S : stack) return boolean;
 stack_error : exception;
private
 type stack is record
 space : array(1..length) of element;
 top : integer range 0..length := 0;
 end record;
end stack;
```

*Why does specification have "private" part?*

## Ada (1980)

```
package body stack is
 procedure make_empty (S : out stack);
 begin
 S.top := 0;
 end make_empty ;

 procedure push (S : in out stack; X : element) is
 begin
 if full(S) then
 raise stack_error;
 else
 S.top := S.top + 1;
 S.space(S.top) := X;
 end if;
 end push;

 ...
end stack;
```

# Ada (1980)

```
s: stack(100,int);
begin
 make_empty(s);
 push(s, 47);
 if (empty(s)) then ...
end;
```