CS 312

Spring 2002

Continuations

Continuations

- SML/NJ has ability to capture a *control context* as a value: a *continuation*
- Continuation = "the rest of the program"
- Example: `fn(z:bool) => if z then ~x else x`

  `if x<0 then ~x else x`

  `fn(a: int)=>a`

  `fn(a: int)=>~a`
- Open SMLofNJ.Cont:

  `'a cont` is a continuation expecting a value of type **`'a`**

  `throw: 'a cont -> 'a -> 'b` throws control to continuation, never comes back

Handling errors

- Can be used in place of exceptions to send control to an arbitrary place

```
errors: string cont
…
fun compute(x: real, errors: string cont) =
  …
  let val z = if y >= 0.0 then sqrt(y)
              else throw errors "negative"
  in
    …
  end
```


Creating continuations

`callcc : ('a cont->'a)->'a`

`callcc f` invokes **`f`** passing it the *current continuation* (which expects an **`'a`**!)

What happened?

- Design and specification of programs
  - modules and interfaces
  - documenting functions and ADTs
  - programming in functional style
  - testing
- Data structures and algorithms
  - collections
  - graphs
  - showing correctness and complexity
- Programming languages
  - Features and methodologies
  - models of evaluation
  - implementation

Life after SML

- 312 is not about SML or even about functional programming
- Lessons apply to Java, C, C++, etc.

Design

- Break up your program into modules with clearly defined interfaces (signatures)
- Use abstract data types (data abstractions)
- Good interfaces are *narrow, implementable,* but *adequate*
- Avoid stateful abstractions, imperative operations unless compelling justification
- Testing strategy and test cases: *coverage*

7

Specification

- Good specifications: clear, simple, concise, accurate
- Think about your audience
- Avoid overspecification
- Abstraction barrier: user should not need to know implementation/representation
- Convince someone that every spec is met
- Specify representation invariants and abstraction functions

8

Data structures and algorithms

- Collections (ordered and unordered)
  - Lists, arrays
  - Hash tables
  - Tries
  - Binary search trees (red-black, splay, treaps, B-trees)
  - Priority queues/heaps
- Graphs
  - BFS, DFS, Dijkstra
  - Game search
- String and regular-expression matching
- Mutable vs immutable data structures
- Locality

9

Correctness and complexity

- Using specifications, invariants to reason about correctness
- Constructing, solving recurrence relations
- Worst-case run time, average case run time, amortized run time
- Proofs by induction

10

Programming languages

- Features
  - Higher-order functions
  - Explicit refs
  - Recursive types and functions
  - Lazy vs. eager evaluation, thunks and streams
  - Concurrency
- Evaluation models (semantics)
  - Substitution
  - Environments and closures
- Implementation
  - Type checking and type inference
  - Objects
  - Memory management, garbage collection

11