Lecture 10: Evaluating Functional Languages

CSCI 131
Spring, 2011

Kim Bruce

Program Correctness

- Verification easier if language satisfies declarative language test -- evaluate once and reuse as necessary.
- let val I = E in E' end is equivalent to [E/I]E'
- Not true if imperative features.
- Only true if lazy evaluation.
- Let E be a functional expression (no side effects). If E converges to a value v with eager evaluation then it converges to the same value with lazy eval.

Why not be lazy?

- Eager languages easier to implement w/ conventional techniques
- If side-effects then when will they occur?
- If computing in parallel, want to start as soon as feasible.
  - Even if result may be wasted!
- Can simulate in eager languages by adding extra dummy parameter to delay evaluation.

Haskell Later ...

- Type inference algorithm
- Support for ADT's and modules
- Support for exception handling
- Garbage collection

Implementation Issues

- Slower than imperative
  - Lists instead of arrays
  - Passing around functions is expensive
    - See why later!
  - Recursion can use more space
  - Lack of destructive updating (but sharing helps!)
  - Listful style
  - Lazy has its own extra overhead

Summary of Haskell

- Successful language for designing large systems
- Lots of experimentation with language design
  - Type classes, software transactional memory, parser combinators
  -

Summary of Functional Langs

- Use requires alternative approaches
- Declarative languages support reasoning
- Higher-order functions powerful
- Some loss of efficiency balanced by programmer efficiency
- Implicit Polymorphism

Types

Why Types?

- Increase readability
- Hide representation
- Detection of errors.
- Help disambiguate operators
- Compiler optimization. E.g. know where fields of record are.
- Help ensure different components in separately compiled units will interoperate properly

Types & Constructors

- Built-in types - *primitive types (incl. size)*
- Aggregate types
- Mapping types
- Recursive types
- Sequence types - *files and strings (primitive?)*
- User-defined types

Aggregate Types

- Cartesian products (tuples)
- Records / Structures
- Union Types
  - C: typedef union {int i; float r;} utype
  - unsafe
  - Discriminated union safer
  - Haskell type defs safe

Discriminated Union: Ada

```
type geometric (Kind: (Triangle, Square) := Square) is
   record
     color : ColorType := Red ;
     case Kind of
       when Triangle =>
            pt1,pt2,pt3:Point;
       when Square =>
            upperleft : Point;
            length : INTEGER range 1..100;
     end case;
   end record;

ob1 : geometric -- default is Square
ob2 : geometric(Triangle) -- frozen, can't be changed
```

Mappings

- Arrays
  - Static - *location & size frozen at compile time* (FORTRAN)
  - Semi-static - *size bound at compile time, location at invocation* (Pascal, C)
  - Dynamic - *size and location bound at creation* (ALGOL 60, Ada, Java)
  - Flex - *size and location can be changed any time* (Java vectors)
- Function Types - *update less efficient*
  - update f arg nuVal = fn x => if x = arg then nuVal else f x

Recursive Types

- In Haskell: data List = Nil | Cons (Integer, List)
- In C: struct list { int x; list *next; };
- Solutions to: list = { Nil } ∪ (int × list)
  - finite seqs of ints followed by Nil: e.g., (2,(5,Nil))
  - finite or infinite seqs: if finite then end w/ Nil
- Recursive eqn's always have a least solution
  - least fixed point!

Least Recursive Solutions

$$
\begin{aligned}
list_0 &= \{Nil\} \\
list_1 &= \{Nil\} \cup (int \times list_0) \\
&= \{Nil\} \cup \{(n, Nil) | n \in int\} \\
list_2 &= \{Nil\} \cup (int \times list_1) \\
&= \{Nil\} \cup \{(n, Nil) | n \in int\} \cup \{(m, (n, Nil)) | m, n \in int\} \\
&\ldots \\
list &= \bigcup_n list_n
\end{aligned}
$$

Some solutions inconsistent w/classical math!

User-Defined Types

- Named types
  - More readable
  - Easy to modify if localized
  - Factorization (why repeat same def?)
  - Added consistency checking if generative
- Enumeration types added to Java 5