Functional Languages

Functional Languages (Applicative, value-oriented)

Importance?

- In their pure form they dispense with notion of assignment
  - claim is: it's easier to program in them
  - also: easier to reason about programs written in them
- FPL's encourage thinking at higher levels of abstraction
  - support modifying and combining existing programs
  - thus, FPL's encourage programmers to work in units larger than statements of conventional languages: "programming in the large"
- FPL's provide a paradigm for parallel computing
  - absence of assignment (single assignment) } provide basis
  - independence of evaluation order } for parallel
  - ability to operate on entire data structures } functional programming

Importance of Functional Languages…

- Valuable in developing executable specifications and prototype implementations
  - Simple underlying semantics
    - rigorous mathematical foundations
    - ability to operate on entire data structures
      => ideal vehicle for capturing specifications
- Utility to AI
  - Most AI done in func langs (extensibility. symbolic manipulation)
- Functional Programming is tied to CS theory
  - provides framework for viewing decidability questions
    - (both programming and computers)
  - Good introduction to Denotational Semantics
    - functional in form

Expressions

- Key purpose of functional programming:
  - to extend the advantages of expressions (over statements) to an entire programming language
- Backus ('78 FP paper) has said that expressions and statements come from two different worlds.
  - expressions: (a + b) * c arithmetic
    (a + b) = 0 relational
    $\neg(a \vee b)$ boolean
  - statements: the usual assortment with assignment singled out
  - *assignments alter the state of a computation* (ordering is important)
    e.g. a:= a * i; i:= i + 1
- In contrast, ordering of expressions is not side-effecting and therefore not order dependent (Church-Rosser property /Church Diamond)

More Expressions

- With Church-Rosser
  - reasoning about expressions is easier
  - order independence supports fine-grained parallelism
  - Diamond property is quite useful
- Referential transparency
  - In a fixed context, the replacement of a subexpression by its value is completely independent of the surrounding expression
    - having once evaluated an expression in a given context, shouldn't have to do it again.

    Alternative: referential transparency is the universal ability to substitute equals for equals (useful in common subexpression optimizations and mathematical reasoning)

Hoare's Principles of Structuring

(1973, "Hints on Programming Language Design," Stanford Tech Rep)

1) Transparency of meaning
   - Meaning of whole expression can be understood in terms of meanings of its subexpressions.

2) Transparency of Purpose
   - Purpose of each part consists solely of its contribution to the purpose of the whole. ➡ No side effects.

3) Independence of Parts
   - Meaning of independent parts can be understood completely independently.
     - In E + F, E can be understood independently of F.

Hoare's Principles of Structuring

4) Recursive Application
   – Both construction and analysis of structure (e.g. expressions) can be accomplished through recursive application of uniform rules.

5) Narrow Interfaces
   – Interface between parts is clear, narrow (minimal number of inputs and outputs) and well controlled.

6) Manifestness of Structure
   – Structural relationships among parts are obvious. e.g. one expression is subexpression of another if the first is textually embedded in the second. Expressions are unrelated if they are not structurally related.

Properties of Pure Expressions

- Value is independent of evaluation order
- Expressions can be evaluated in parallel
- Referential transparency
- No side-effects (Church Rosser)
- Inputs to an expression are obvious from written form
- Effects of operation are obvious from written form

→ Meet Hoare's principles well

→ Good attributes to extend to all programming (?)

A Scheme Continuation

```
(set init-rand (lambda (seed)
  (lambda () (set seed (mod (+ (* seed 9) 5) 1025)))))

(set rand (init-rand 1))
```

- Sequence of calls to (rand) produces a changing set of values
- So much for referential transparency…

Basic Data Types in Application Languages

- Atomic data types

  - Integers I } $+$, $-$, x, $\div$, $/$, $=$,
  - Reals R } $\neq$, $<$, $\leq$, $>$, $\geq$

  - Strings $=$, $\neq$

  - Boolean B $=$, $\neq$, $<$, $\leq$, $>$, $\geq$

Composite Data Types

- Sequences (a generic data type)

e.g. <'CS655', 'CS654', 'CS851'>

< <'CS655', 'CS851'>, < 'CS654'> >

-- can have subsequences to any depth as long as subsequences are matching type.

i.e. <'CS655'> $\neq$ 'CS655'

- "sequence" by itself is not a type
  - type of a sequence determined by its components

Sequence(I) = sequence of integer

Sequence(R) = sequence of real, etc.

(continue) Data Types

- Notational convenience concerning sequences:

τ^* means sequence of objects of type τ

e.g. I^* is sequence of integers.

- Typical operations on sequences -

1) *constructors*: build sequences from components

2) *selectors*: select elements from a sequence

3) *discriminators*: distinguish among different *classes* of sequences. e.g. empty vs. non-empty.

$\rightarrow$ Would like to:

a) minimize overall set

b) have set be complete (in sense that functional composition can create any other operation)

(continue) Data Types

- Traditional FPL constructors, selectors and discriminators:
  constructors: NIL, prefix (Cons)
  selectors: first (car), rest (cdr)
  discriminators: Null, ¬ Null

- Comments about domains:
  1) first, rest: not defined on null sequences or atoms
  2) Null: not defined on atoms

Partial functions

Archetypes

- (sequence example, next slide)

- Archetype: ideal characterization
  - as contrasted with prototypes
    archetype: what you want
    prototype: shows feasibility (implementability)
- Archetypes tend to be written using algebraic specification
  - for syntax and semantics
  - also, often include pragmatics: comments on efficiency

Infinite Sequences Archetype

- Syntax:

  nil $\in \tau^*$

  null: $\tau^* \to B$

  first: $\tau^* \to \tau$

  rest: $\tau^* \to \tau^*$

  prefix: $\tau \times \tau^* \to \tau^*$

  Primitive function signatures; domain $\to$ range

  - generic (polymorphic)
  - partial

  x:S $\Rightarrow$ prefix (x, S)

  < > $\Rightarrow$ nil

  <x1, x2, …, xn > $\Rightarrow$ x1 : <x2, …, xn>

  Rewrite rules

Infinite Sequences Archetype (cont)

- Semantics:

  nil $\in \tau^*$ x : S $\in \tau^*$

  z $\notin \tau^*$, otherwise

  Existence axioms

null nil = **true**	null (x:S) = **false**
first nil $\neq$ x	first (x:S) = x
rest nil $\neq$ S	rest (x:S) = S

  Equations

- Pragmatics:
  - The first, rest, prefix and null operations all take constant time. The prefix operation is significantly slower than the others.

Notes on Sequence Archetype

a) $< x_1, x_2, \ldots x_n > \;\rightarrow\; x_1 : < x_2, \ldots x_n >$
$\qquad\qquad\qquad\qquad \rightarrow\; x_1 : x_2 : x_3 \ldots : x_n : NIL$

b) *domains*, *ranges* and *signatures* are important concepts

c) defined functions are *generic*

d) defined functions are *partial*

- First, rest don't work on null sequences

More on Sequence Archetype

$\texttt{nil} \in \tau^*$
$\texttt{x : S} \in \tau^*$
$\texttt{z} \notin \tau^* \;\; \texttt{otherwise}$

- tells us that members of sequence type τ^* include NIL, and every result of prefix operation: x : S
- $z \notin \tau^*$ otherwise -- means nothing else is included in type τ^*

rest(x : S) = S } define equivalence relations
first(x : S) = x } on well-formed formulae

formula

equation

Completeness and Consistency

- Semantic equations are ***complete*** if they are sufficient to either prove or disprove every well-formed equation between formulae
  - e.g. if Null NIL = TRUE had been omitted from sequences specification then we would have not been able to prove from other equations whether

    Null NIL = TRUE

    or Null NIL ≠ TRUE

- *Consistency*: No two equations contradict
  - In short, don't want to be able to prove both S and ¬S for any S
  - e.g. if we had both Null NIL = TRUE

    Null NIL = FALSE

    - could prove TRUE = FALSE from Null NIL = Null NIL
    - contradicts Boolean archetype

Boolean Archetype (partial)

- Syntax

  . . .

- Semantics:

  x ∈ B if and only if x = **true** or x = **false**

  true ≠ **false**

  ¬ **true** = **false**

  ¬ **false** = **true**

  . . .

Handout...

Completeness and Consistency (cont)

- Consistency of a specification can be shown by implementing it.
  - Existence of a model is proof of consistency
  - Show that every primitive operation defines a unique output for every allowable input.

- Completeness is much more difficult. To establish completeness, one must specify behavior of opertions on all inputs.
  - e.g. in case of first and rest, for all x's and all S's:

    first NIL ≠ x -- first NIL is not an element

    & rest NIL ≠ S -- rest NIL is not a sequence

  - use ≠ since saying "first NIL is undefined" allows us to write first NIL = 1/0 -- not something we want to allow.
  - Thus, ≠ is used to mean "undefined."

Infinite Sequences

- What if we want to specify the sequence:

  5, 10, 13, 1, 7, 1, 7, 1, 7, ... ?

- Do the specifications given earlier allow this?

- Consider the sequence: C = <0, 0, 0, ... >
  - Is C a legal sequence?

- Recall we have:

  $\texttt{nil} \in \tau^*$

  $\texttt{x : S} \in \tau^*$

  $\texttt{z} \notin \tau^*$ otherwise

Infinite Sequences Archetype

- Syntax:

nil $\in \tau^*$
null: $\tau^* \rightarrow B$
first: $\tau^* \rightarrow \tau$
rest: $\tau^* \rightarrow \tau^*$
prefix: $\tau \times \tau^* \rightarrow \tau^*$

Primitive function signatures;
domain $\rightarrow$ range

- generic (polymorphic)
- partial

x:S $\Rightarrow$ prefix (x, S)
< > $\Rightarrow$ nil
<x1, x2, …, xn > $\Rightarrow$ x1 : <x2, …, xn>

Rewrite rules

Infinite Sequences Archetype (cont)

- Semantics:

nil $\in \tau^*$ x : S $\in \tau^*$
z $\notin \tau^*$, otherwise

Existence axioms

null nil = **true** null (x:S) = **false**
first nil $\neq$ x first (x:S) = x
rest nil $\neq$ S rest (x:S) = S

Equations

- Pragmatics:
  - The first, rest, prefix and null operations all take constant time. The prefix operation is significantly slower than the others.

Infinite Sequences (cont)

- Considering C = 0:C yields

$C \in I^*$ if $0 \in I$ and $C \in I^*$
$C \notin I^*$ otherwise

well... $0 \in I$ and $C \in I^*$ iff $C \in I^*$

We can consistently assume it either *is* or *is not.*
$\Rightarrow$ sequence archetype is incomplete

- Aside: assuming $C \in I^*$ works okay in other axioms:
  first C = 0, rest C = C, 0:C = C
  null C = null (0:C) = false, etc.

Finite Sequences

- Syntax:

$$\left.\begin{array}{l} nil \in \tau^* \\ null: \tau^* \rightarrow B \\ first: \tau^* \rightarrow \tau \\ rest: \tau^* \rightarrow \tau^* \\ prefix: \tau \times \tau^* \rightarrow \tau^* \\ length: \tau^* \rightarrow N \end{array}\right\} \text{Primitive function signatures; domain} \rightarrow \text{range}$$

length: $\tau^* \rightarrow N$ ← Finite, natural numbers

$$\left.\begin{array}{l} x{:}S \Rightarrow prefix\ (x, S) \\ \langle\ \rangle \Rightarrow nil \\ \langle x1, x2, \ldots, xn\rangle \Rightarrow x1 : \langle x2, \ldots, xn\rangle \end{array}\right\} \text{Rewrite rules}$$

Finite Sequences (cont)

- Semantics:

  nil $\in \tau^*$ x : S $\in \tau^*$

  z $\notin \tau^*$, otherwise

  Existence axioms

  null nil = **true** null (x:S) = **false**

  first nil $\neq$ x first (x:S) = x

  rest nil $\neq$ S rest (x:S) = S

  Equations

  length nil = 0 length (x:S) = 1 + length(S)

  length is a total function on τ^*

- Pragmatics:
  - The first, rest, prefix and null operations all take constant time. The prefix operation is significantly slower than the others. Length takes time at most proportional to length of argument

Operations on Sequences

- Concatenation

  cat: τ^* x $\tau^* \rightarrow \tau^*$

  e.g. cat (<1,2> <3,4,5>) = <1,2,3,4,5>

- Reductions

  sum: R* $\rightarrow$ R

  e.g. sum(<1,2,3,4,5>) = 15

  max: R* $\rightarrow$ R

  e.g. max(<1,2,3,4,5>) = 5

  . . .

- Mappings...

Finite Sets

- Handout...

Higher Order Functions

- Defs:
  - *zero-order functions*: data in the traditional sense.
  - *first-order functions*: functions that operate on zero-order functions.

    e.g. FIRST: $\tau^* \rightarrow \tau$

    REST: $\tau^* \rightarrow \tau^*$
  - *second-order functions*: operate on first order

    e.g. map: $(D \rightarrow R) \rightarrow (D^* \rightarrow R^*)$ $\forall$ D, R $\in$ type

    uncurried: $((D \rightarrow R) \times D^*) \rightarrow R^*$

Higher Order Functions (cont)

- In general, higher-order functions are those that can operate on functions of any order as long as types match.
  - HOF's are usually polymorphic

- Higher-order functions can take other functions as arguments and produce functions as values.

- More defs:
  - *Applicative programming* has often been considered the application of first-order functions.
  - *Functional programming* has been considered to include higher-order functions: *functionals*.

Functional Programming

- Functional programming allows *functional abstraction* that is not supported in imperative languages, namely the definition and use of functions that take functions as arguments and return functions as values.

  - supports higher level reasoning

  - simplifies correctness proofs

Functional Abstraction

- For an arbitrary function, f, and sequence, S, we can define:

$$\text{map f S} \equiv \begin{cases} \text{NIL, if null S} \\ \text{else f(first S) : map f (rest S)} \end{cases}$$

  - map is a two-argument function
    - map is applied to f
    - resulting function is applied to S
- map f S signature:

$$\text{map: } [(D \to R) \times D^*] \to R^* \qquad \text{-- uncurried}$$
$$\text{map: } (D \to R) \to (D^* \to R^*) \quad \forall\, D, R \in \text{type} \qquad \text{-- curried}$$

- map takes a function that maps from D to R and yields a function that maps from D^* to R^*. (Note, independent of S)

Functional Abstraction (cont)

- *Functional abstraction* includes giving meaning to 'map f ' independent of S.
- In general, for any functional equation:

$$Fx = E$$

we can, through functional abstraction, modify to:

$$F = x \mapsto E$$

- where x is arbitrary and does not occur in F

- ($x \mapsto E$ means "taking x into E")

e.g. x taken into x^2 - 3a means x's are the same - recall λ calculus

Functional Abstraction (cont)

- We can rewrite map as:

$$\text{map f} \equiv \text{S} \mapsto \begin{cases} \text{NIL, if null S} \\ \text{else f(first S) : map f(rest S)} \end{cases}$$

the high order function we seek

* In Scheme:

```
(define (map f)
   (lambda (S)
    (IF (null? S) nil
       (cons (f (car S))((map f) (cdr S)) )) ))
```

could call map with: ((map sin) (interval 0 90))

Map Archetype

Syntax:

map: $(T \rightarrow U) \rightarrow (T^* \rightarrow U^*)$, for all $T, U \in$ **type**

Semantics:

map f nil = nil

map f (x : S) = f x : map f S

Pragmatics:

with sequential implementations map f S takes linear time; on some parallel implementations it takes constant time.

Prototype:

$$\text{map } f \equiv \text{ S } \mapsto \begin{cases} \text{nil, if null S} \\ \text{else } f \text{ (first S) : map } f \text{ (rest S)} \end{cases}$$

Filtering

- Consider:

positives <3, -2, 6, -1, -5, 8, 9> = <3, 6, 8, 9>

$$\text{positives } S \equiv \begin{cases} \text{nil, if null } S \\ \text{first } S : \text{positives (rest } S) \text{ if first } S > 0 \\ \text{else positives (rest } S) \end{cases}$$

- Let's generalize and consider a general filtering function:

$$\text{fil } P \equiv \quad S \mapsto \begin{cases} \text{nil, if null } S \\ \text{first } S : \text{fil } P \text{ (rest } S) \text{ if } P \text{ (first } S) \\ \text{else fil } P \text{ (rest } S) \end{cases}$$

A Scheme Filter

```
(define (fil P)
        (lambda (S)
           (cond ((null S) NIL)
                  ((P (car S)) (cons (car S) ((fil P) (cdr S) )))
                  (else ((fil P) (cdr S))   ))   ))
```

Filter Archetype

Syntax:

fil: $(T \rightarrow B) \rightarrow (T^* \rightarrow T^*)$, for all $T \in$ **type**

Semantics:

fil P nil = nil

fil P (x : S) = x : fil P S, if Px = **true**

fil P (x : S) = fil P S, if Px = **false**

Pragmatics:

with a sequential implementation fil P S takes linear time; with some parallel implementations it takes constant time.

Prototype:

$$\text{fil P} \equiv \ \text{S} \mapsto \begin{cases} \text{nil, if null S} \\ \text{first S : fil P (rest S) if P (first S)} \\ \text{else fil P (rest S)} \end{cases}$$

Sequences and Sets

- A typical prototype for the finset archetype axiom:

$$x \in (S \cap T) = x \in S \wedge x \in T$$

is:

$$S \cap T = \begin{cases} \phi \text{ if empty S} \\ \text{else adjoin (first S, rest S } \cap \text{T) if first S} \in \text{T} \\ \text{else rest S } \cap \text{T} \end{cases}$$

$\Rightarrow$ <u>if we use sequences to represent sets then we must prove that we have true set operators.</u>

- Proof can be lengthy, but HO functions can help.

A Note on Correctness

- Proving properties about functionals simplifies subsequent proofs where functionals are used:

  e.g. to prove: $x \in$ fil P S = Px $\wedge$ $x \in$ S (A)

  – x a member of the *sequence* produced by applying fil P to S is equivalent to the truth of Px and x a member of the sequence S.

- Three lemmata to prove:

  1) $x \in$ fil P S $\rightarrow$ $x \in$ S -- fil doesn't add any elements
  2) $x \in$ fil P S $\rightarrow$ Px -- x only added to seq if Px true
  3) $x \in$ S $\wedge$ Px $\rightarrow$ $x \in$ fil P S -- fil captures all members of S that satisfy P

  Prove once

  (note: 1 & 2 give $\rightarrow$ and 3 gives $\leftarrow$, proving (A))

More on Correctness

- Consider

  P $\equiv \in$ S (test for membership)

  then:

  fil [$\in$ S] T produces the *sequence* of elements that are members of both S & T (which are sequences)

  that is:

  S $\cap$ T $\equiv$ fil [$\in$ S] T -- by definition (C)

Continue Correctness of Set Intersection

- So correctness of set intersection:

  $x \in (S \cap T) = x \in S \wedge x \in T$ -- from finite set archetype

  can be demonstrated by:

  $x \in (S \cap T) = x \in \text{fil } [\in S]\ T$ -- from (C)

  $= [\in S]\ x \wedge x \in T$ -- from (A)

  $= x \in S \wedge x \in T$ □

- (This proof shows that a sequence implementation of a set can satisfy a set archetype)

Composition Archetype

Syntax:

$\circ$: $[(S \to T) \text{ x } (R \to S)] \to (R \to T)$, for all $R, S, T \in$ **type**

that is, $(f \circ g)$: $R \to T$ for f: $S \to T$ and g: $R \to S$

Semantics:

$(f \circ g)\ x = f\ (g\ x)$

Pragmatics:

Composition takes the same time as the composed functions.

Prototype:

$f \circ g \equiv x \mapsto f\ (g\ x)$

Construction Archetype

Syntax:

[;]: $[(S \rightarrow T) \times (S \rightarrow U)] \rightarrow [(S \rightarrow (T \times U)]$, for all $S, T, U \in$ **type**

that is, $(f\ ;\ g): S \rightarrow (T \times U)$, for $f: S \rightarrow T$ and $g: S \rightarrow U$.

$(f_1; f_2; \ldots ; f_n) \qquad (f_1; (f_2; \ldots ; f_n))$

Semantics:

$(f\ ;\ g)\ x = (f\ x, g\ x)$

Pragmatics:

With sequential implementations, n-ary construction takes the sum of the times of the constructed functions. With some parallel implementations it takes the time of the slowest function.

Prototype:

$f\ ;\ g \equiv\ x \mapsto (f\ x, g\ x)$

Haskell & ML: Interesting Features

- Type inferencing
- Freedom from side effects
- Pattern matching
- Polymorphism
- Support for higher order functions
- Lazy patterns / lazy evaluation
- Support for object-oriented programming

Type Inferencing

- Def: ability of the language to infer types without having programmer provide type signatures.
  - SML e.g.:

```
fun min (a: real,  b)
  =  if  a  >  b
  then  b
  else  a
```

  - type of a has to be given, but then that's sufficient to figure out
    - type of b
    - type of min
  - What if type of a is not specified?
    - could be ints
    - could be bools...

Type Inferencing (cont)

- Haskell (as with ML) guarantees type safety
  - Haskell example:

```
eq  =  (a = b)
```

  - a polymorphic function that has a return type of bool,
    - assumes only that its two arguments are of the same type and can have the equality operator applied to them.
  - ML has similar assumption, for what it calls *equality types*.
- Overuse of type inferencing in both languages is discouraged
  - declarations are a design aid
  - declarations are a documentation aid
  - declarations are a debugging aid

Polymorphism

- ML:

```
fun factorial (0)  =  1
= | factorial (n)  =  n * factorial (n - 1)
```

  – ML infers factorial is an integer function: int -> int

- Haskell:

```
factorial (0)  =  1
factorial (n) = n * factorial (n - 1)
```

  – Haskell infers factorial is a (numerical) function: Num a => a -> a

Polymorphism (cont)

- ML:

```
fun mymax(x,y) = if x > y then x else y
```

  – SML infers mymax is an integer function: int -> int

```
fun mymax(x: real ,y) = if x > y then x else y
```

  – SML infers mymax is real

- Haskell:

```
mymax(x,y) = if x > y then x else y
```

  – Haskell infers factorial is an Ord function