CMPUT 325: Abstract Programming

Dr. B. Price and Dr. G. Greiner

19th October 2004

Abstract Programming

- λCalculus has precise semantics, simple syntax, simple evaluation
- Its also extremely tedious
- Standard idioms for many high-level control constructs
- Use abstract idioms in place of λ-calculus
  - Easy to read
  - Guaranteed semantics and simple evaluation
- Simple parser converts abstractions to idioms
- λcalculus solves problem

Abstract Programming: Datatypes

- Numbers: use Church's 2 arg function representation
  - Integers: n $\equiv$ `(λ s z | s`^{k} `z)`
    where `s`^{k} is a string of k `s`'s
- Boolean values: T $\equiv$(λc d| c) and F$\equiv$(λc d | d)
- List
  - Cons cell (M . N) $\equiv$ (λz | z m n)
  - List (a b c 0): (λz | z a (λz | z b (λz | z c 0)))
- String: treat chars as an integer in base 256
  - Each char replaced by ASCII value
  - HELLO $\equiv$ H*256^4+E*256^3+L*256^2+L*256^1+O*256^0

Abstract Programming: Functions

- Assume primitive operators on datatypes defined
  - Mathematical ops: add, sub, mul, div, zerop
  - List ops: cons, car, cdr
  - Boolean operators: and, or, not
- Allow standard mathematical notations
  - Infix notation:

```
1+2 ≡(+ 1 2)
≡(λ x y | (λs z | x s (y s z) ) ) 1 2
```

  - Functional notation:

```
f(x) ≡ (λy | ... ) x

square(2) ≡ (λy | (* y y)) 2
          ≡ (λy | ⟨multiplication-idiom⟩) 2
```

Conditionals

```
IF x<0 THEN -x ELSE x
```

- λ-calculus translation?

  `(λxyz|xyz) x<0 -x x`

- NOTE: must have both THEN and ELSE clauses. Why?
- λ-calculus predicates resolve to T or F
  - T chooses first argument
  - F chooses second argument
- Must have an argument for each case or program will behave strangely

Special Forms: LET by Examples

- In abstract programming we define " LET AND IN" special form

```
LET x=5 IN x+1   → 6

LET x=2 IN LET y=2 IN x+y   → 4

LET x=2 AND y=2 IN x+y   →  4

LET f(x)=x*x AND y=3 IN
   LET x=f(y) IN
     x
→ 9
```

Special Forms: LET Semantics I

```
LET x = ⟨E⟩ IN ⟨BODY⟩
```

- λ calculus translation? `(λx | ⟨BODY⟩ ) ⟨E⟩`

```
LET x = ⟨E⟩ IN
   LET y = ⟨F⟩ IN ⟨BODY⟩
```

- λ calculus translation?
  `(λx| (λy| ⟨BODY⟩) ⟨E⟩) ⟨F⟩`

```
LET x = ⟨E⟩ AND y = ⟨F⟩ IN ⟨BODY⟩
```

- λ calculus translation? Parallel substitution
  `(λxy| ⟨BODY⟩) ⟨E⟩⟨F⟩`

Special Forms: LET Semantics II

```
LET x = ⟨E⟩ IN LET x = ⟨F⟩ IN ⟨BODY⟩
```

- λ calculus translation?
  `(λx| (λx| ⟨BODY⟩) ⟨E⟩) ⟨F⟩`

```
LET x =⟨E⟩
   LET x = ⟨F⟩ AND y = x IN ⟨BODY⟩
```

- λ calculus translation?
  `(λx| (λxy| ⟨BODY⟩) ⟨F⟩ x) ⟨E⟩`

```
LET f(x) = ⟨E⟩ IN ⟨BODY⟩
```

- λ calculus translation?
- Closer: `LET f = (λx| ⟨E⟩) IN ⟨BODY⟩`

  `(λf| ⟨BODY⟩) (λx |⟨E⟩)`

- λ-calculus gives precise meaning to each case of LET

Special Forms: LET and Self-reference

```
LET f(n) =
   IF zerop(n)   THEN 1   ELSE n*f(n-1)
IN ⟨BODY⟩
```

- λ calculus translation? Approximately:

```
(λf | ⟨BODY⟩)
   ( (λxyz|xyz) zerop(n) 1 n*f(n-1) )
```

$$(\lambda f \mid \langle BODY\rangle)\ \ (\ (\lambda xyz|xyz)\ zerop(n)\ 1\ n*\underbrace{f}_{\uparrow}(n-1)\ )$$

- What does the recursive call to f point to? It is a free variable!
- Is this correct? Yes.
  Otherwise `LET x=2 IN LET x=2*x IN ⟨BODY⟩` would fail:
  $(\lambda x|\ (\lambda x|\langle BODY\rangle)\ 2{*}x)\ 2$

Special Forms: LETREC

```
LETREC f(n) =
   IF zerop(n)
   THEN 1
   ELSE n*f(n-1)
IN ⟨BODY⟩
```

- Sometimes we want vars in definition to refer to their labels
- Different semantics than LET — needs different name
- λ-calculus translation?Use combinator operator Y

```
(λf|⟨BODY⟩) (Y (λf| (λn| zerop(n) 1 n*f(n-1)) ))
```

$$(\lambda \underbrace{f}_{\uparrow} |\langle BODY\rangle)\ (Y\ (\lambda f|\ (\lambda n|\ zerop(n)\ 1\ n*\underbrace{f}_{\uparrow}(n-1))\ ))$$

- Are 2 f's the same? No. f in function def is not free!

Special Forms: Nested LETREC

```
LETREC
  f(n) = IF zerop(n) THEN 1 ELSE n*f(n-1) IN
  LETREC
     g(n) = IF zerop(n) THEN 0 ELSE f(n)+g(n-1) IN
       ⟨BODY⟩
```

- What does this do? Sums first n factorials. Translation?

```
(λf|
   (λg|
      ⟨BODY⟩
   ) (Y (λg| (λn| zerop(n) 0 f(n)+g(n-1)) ))
) (Y (λf| (λn| zerop(n) 1 n*f(n-1)) ))
```

- What does each f in this definition refer to?
- Functions can refer to themselves and to earlier definitions

Special Forms: Parallel LETREC

```
LETREC
    even(n) IF zerop n THEN T ELSE odd( n-1 )
AND odd(n)  IF zerop n THEN F ELSE even( n-1 ) IN
   ⟨BODY⟩
```

- In mutually recursive functions, earlier functions also refer to later functions
- Translation? Need pair of combinators that generate either function

```
Y1=(λfg|RRS)  Y2=(λfg|SRS)
Where R=(λrs|f(rrs)(srs)), S=(λrs|g(rrs)(srs))
```

- Combinator Properties

```
Y1 F G = F (Y1 F G) (Y2 F G)
Y2 F G = G (Y1 F G) (Y2 F G)
```

Special Forms: Parallel LETREC

- Given the following definitions for F and G

```
F≡even(n) IF zerop n THEN T ELSE odd( n-1 )
G≡odd(n)  IF zerop n THEN F ELSE even( n-1 )
```

- LETREC Expansion using pair of combinators

```
(λfg|⟨BODY⟩)
   (Y1 (λfg | F)(λfg|G))
   (Y2 (λfg | F)(λfg|G))
```

- Why can't I use 2 independent combinators?
  - Each copy of the function F has to also be able to reference G

Abstract Programming: BNF

```
⟨identifier⟩:= ⟨alpha-char⟩{⟨alpha-char⟩|⟨number⟩}
⟨constant⟩:=⟨number⟩|⟨boolean⟩|⟨char-string⟩
⟨expression⟩:=⟨constant⟩|⟨identifier⟩
  | (λ⟨identifier⟩ "|" ⟨expression⟩ )
  | (⟨expression⟩⁺ )
  | ⟨identifier⟩(⟨expression⟩{,⟨expression⟩}*)
  | let ⟨definition⟩ in ⟨expression⟩
  | letrec ⟨definition⟩ in ⟨expression⟩
  | if ⟨expression⟩ then ⟨expression⟩ else ⟨expression⟩
  | ⟨arithmetic expression⟩
⟨definition⟩:=⟨header⟩=⟨expression⟩
   | ⟨definition⟩ {and ⟨definition⟩}*
⟨header⟩:=⟨identifier⟩
   | ⟨identifier⟩ ( ⟨identifier⟩ {, ⟨identifier⟩}*)
⟨abstract-program⟩:=⟨expression⟩
```

Convenience: WHERE and WHEREREC

- Sometimes convenient to put definitions after usage

  ⟨BODY⟩ WHERE ⟨DEFINITION⟩

- Example

```
LET a(r) = pi * r IN
   a(10)
WHERE pi = 3.1415
```

- Do we need brackets? No.
  - LET's ⟨BODY⟩ is a single term
  - Abstract Programming is Left-associative
- WHEREREC is analogous to LETREC
- WHERE and WHEREREC do not add expressive power, just convenience

Performance Considerations

- LET and LETREC mean different things

  LET x=x+2 IN ⟨BODY⟩ $\not\equiv$ LETREC x=x+2 IN ⟨BODY⟩

- Meaning overlaps when there is no self-reference

  LET x=2 IN ⟨BODY⟩ $\equiv$ LETREC x=2 IN ⟨BODY⟩

```
LET y=2 IN                ≡ LETREC y=2 IN
   LET x=y IN ⟨BODY⟩           LETREC x=y IN ⟨BODY⟩
```

- Depending on compiler, may be more efficient to use LET when possible

Higher-order Functions

- Abstract language looks like traditional languages
- Underlying semantics does not distinguish data and functions
- Higher-order function has at least one of these properties
  - Accepts a function as an argument
  - Returns a function as its value
- Can treat functions as arguments or return values

```
LET map(f,L) =
   IF null(L)
   THEN nil
   ELSE cons( f(car(L)) , map(cdr(L)) )  IN

   LET square(x)=x*x IN

      map(square, [1 2 3 4])
```


Other Traditional Higher-order Functions

- Filter: apply a predicate to each item and return those items that satisfy

   (filter 'even [1 2 3 4])$\longrightarrow$[2 4]

- Reduce: combine elements of list with given function left associatively
  (common Lisp: reduce)

```
(reduce #'- '(1 2 3 4))
≡(((1 - 2) - 3) - 4)
≡((-1 - 3) - 4)
≡(-4 -4)
≡-8
```

Global Definitions

- In principle, there aren't any: no `DEFUN` or `SETF`
- There are only nested `LET` statements
- In principle, integers and primitives defined by `LET`

  LET T = $(\lambda$xy|x)
  AND F = $(\lambda$xy|y)
  AND + = ...
  $\vdots$
  IN $\langle$BODY$\rangle$

Abstract Programming

- Can be used to implement any functional language
- Is equivalent in power to a Turing machine
- Abstract programming language approximately equivalent to Pure Lisp
  - Parallel `LET` $\approx$ Lisp `LET`
  - Nested `LET`'s $\approx$Lisp `LET*`
  - `Parallel LETREC`'s $\approx$Lisp `LABELS`

Partial application / Currying

- In principle, λ's can be used anywhere in abstract programming

  `map( (λx| 2+x), [1 2 3])` $\rightarrow$ `[3 4 5]`
- A more elegant method:
- Let pa be the partial application operator

  `LET pa = (λf x| (λy| f x y)) IN ⟨BODY⟩`
- Allows us to write:

```
LET inc = pa '+ 1 IN   ;; i.e. inc = (λy| (+ 1 y))
   inc(1) →2
```

- Or more impressively:

  `map( pa + 2, [1 2 3] ) →[3 4 5]`
- Partial application $\equiv$ currying

Combinators as a Calculus

- The central operation in λ-calculus is the β-substitution
- It requires
  - scanning expressions for variables
  - analyzing free vs. bound variables
  - renaming when conflicts are discovered
  - rebuilding substituted copies of expressions repeatedly
- λ-parameters just "steer" copies of expressions to places in code
- Define "combinators" which move, copy and delete arguments

Combinators as Special Functions

- Suppose we had a library of useful combinators: X,Y,Z
- Intuitive example:
  - Program ≡string of combinators: `ZXYZYY...`
  - Suppose 2 argument combinator Z reverses its arguments `ZXYZYY...` → `YXZYY...`
  - Suppose 1 argument combinator Y duplicates its arguments `YXZYY...` →`XXZYY...`
  - Suppose 1 argument combinator X deletes its second argument `XXZYY...` →`XZYY...`
- Combinators can be defined using λ-calculus: X≡(λxy|x)
- Given combinators, no λ's, formal parameters or substitution required

Combinators as a Calculus

- Left-associative like λ-caculus: ABCD...≡(((AB)C)D)
- Proved that two combinators can generate all others

Symbol	Name	A λCalculus Def	Semantics
s	distribute	(λxyz|xz(yz))	S A B C → AC (BC)
k	constant	(λxy|x)	K A B →A

- Identity function:I≡S K K A ≡K A (K A) ≡ A

Common Combinators

- Common combinators can be defined using S and K

Symbol	Name	A λCalculus Def	Semantics
B	compose	$(\lambda xyz \vert x(yz))$	B A B C $\rightarrow$ A (BC)
C	reversal	$(\lambda xyz \vert xzy)$	C A B C $\rightarrow$A C B
W	duplicate	$(\lambda xy \vert xyy)$	W M N $\rightarrow$M N N

Common Combinators

- There is a mechanical mapping between λ-calculus and the minimal SKI combinator language consisting of only S,K and I combinators
- cons $\equiv$B C (C I)
- car $\equiv$C I K
- cdr $\equiv$C I (K I)
- integers: zero$\equiv$ KI, Z_i= (s (s ...(s z)...)) with i copies of s
- successor(n) $\equiv$ (S B Z_i) s z
- Factorial: f(n)=(if (= n 0) 1 ($\times$n (f (- n 1))))
  $\equiv$(S(C(B if (C=0)) 1) (S x (B f (C - 1))))

Combinator Notes

- Common subsequences can be compiled into super-combinators
- VLSI chips have been fabricated to directly implement combinator logic
- Haskell and Miranda define many high-level combinators
- Another example
- Divide every number in L by 2
  `MAP (/ SWAP 2 ) L`
- SWAP reverses arguments to / so we get
  - each number divided by 2
  - instead of 2 divided by each number