Abstract Programming

- λ-calculus has precise semantics, simple syntax, simple evaluation
- It’s 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
- λ-calculus solves problem
Abstract Programming: Datatypes

- Numbers: use Church’s 2 arg function representation
  - Integers: \( n \equiv (\lambda s z | s^k z) \)
    where \( s^k \) is a string of \( k \) s’s

- Boolean values: \( T \equiv (\lambda c d | c) \) and \( F \equiv (\lambda c d | d) \)

- List
  - Cons cell \( (M . N) \equiv (\lambda z | z m n) \)
  - List \( (a b c 0) \): \( (\lambda z | z a (\lambda z | z b (\lambda z | z c 0))) \)

- String: treat chars as an integer in base 256
  - Each char replaced by ASCII value
  - \( \text{HELLO} \equiv H \times 256^4 + E \times 256^3 + L \times 256^2 + L \times 256^1 + O \times 256^0 \)

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:
    \( 1 + 2 \equiv (+ \ 1 \ 2) \)
    \( \equiv (\lambda x y | (\lambda s z | x s (y s z)) \ ) \ 1 \ 2 \)
  - Functional notation:
    \( f(x) \equiv (\lambda y | \ldots ) \ x \)
    \( \text{square}(2) \equiv (\lambda y | (* \ y \ y)) \ 2 \)
    \( \equiv (\lambda y | (\text{multiplication-idiom})) \ 2 \)
Conditionals

IF \( x < 0 \) THEN \(-x\) ELSE \(x\)

- \(\lambda\)-calculus translation?

\((\lambda xyz \mid xyz) \ x < 0 \ -x \ x\)

- NOTE: must have both THEN and ELSE clauses. Why?

- \(\lambda\)-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 \(f(x) = x^2\) 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⟩

LET x = ⟨E⟩ AND y = x IN ⟨BODY⟩
▶ λ calculus translation?
(λx | (λx | ⟨BODY⟩) ⟨E⟩) ⟨F⟩

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 \( f(n) = \)
\[
\begin{align*}
&\text{IF zerop}(n) \ \text{THEN} \ 1 \\
&\text{ELSE} \ n*f(n-1)
\end{align*}
\]
IN \langle BODY \rangle

▶ λ calculus translation? Approximately:
\[
(\lambda f \mid \langle BODY \rangle)
\]
\[
( (\lambda xyz|xyz) \text{zerop}(n) \ 1 \ n*f(n-1) )
\]
\[
(\lambda f \mid \langle BODY \rangle)
\]
\[
( (\lambda xyz|xyz) \text{zerop}(n) \ 1 \ n*f(n-1) )
\]

▶ 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 \langle BODY \rangle would fail:
\[
(\lambda x \mid (\lambda x \mid \langle BODY \rangle) \ 2*x) \ 2
\]

Special Forms: LETREC

LETREC \( f(n) = \)
\[
\begin{align*}
&\text{IF zerop}(n) \\
&\text{THEN} \ 1 \\
&\text{ELSE} \ n*f(n-1)
\end{align*}
\]
IN \langle BODY \rangle

▶ Sometimes we want vars in definition to refer to their labels

▶ Different semantics than LET — needs different name

▶ λ-calculus translation? Use combinator operator \( Y \)

\[
(\lambda f \mid \langle BODY \rangle) \ ( Y (\lambda f \mid (\lambda n \mid \text{zerop}(n) \ 1 \ n*f(n-1)) ) )
\]
\[
(\lambda f \mid \langle BODY \rangle) \ ( Y (\lambda f \mid (\lambda n \mid \text{zerop}(n) \ 1 \ n*f(n-1)) ) )
\]

▶ 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

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

(\lambda f |
 (\lambda g |
  \langle \text{BODY} \rangle 
  (Y (\lambda g | (\lambda n | zerop(n) 0 f(n)+g(n-1))) ))
 (Y (\lambda f | (\lambda n | 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

▶ In mutually recursive functions, earlier functions also refer to later functions

▶ Translation? Need pair of combinators that generate either function

Y_1=(\lambda fg|R_R) \quad Y_2=(\lambda fg|S_S)
Where \, R=(\lambda rs | f(rss)(srs)) , \, S=(\lambda rs | g(rss)(srs))

▶ Combinator Properties

Y_1 \, F \, G = F \, (Y_1 \, F \, G) \, (Y_2 \, F \, G)
Y_2 \, F \, G = G \, (Y_1 \, F \, G) \, (Y_2 \, F \, G)
Special Forms: Parallel LETREC

- Given the following definitions for $F$ and $G$
  
  \[
  F \equiv \text{even}(n) \ IF \ \text{zerop} \ n \ \text{THEN} \ T \ \text{ELSE} \ \text{odd}( \ n-1 )
  \]
  
  \[
  G \equiv \text{odd}(n) \ IF \ \text{zerop} \ n \ \text{THEN} \ F \ \text{ELSE} \ \text{even}( \ n-1 )
  \]

- LETREC Expansion using pair of combinators

  \[
  (\lambda fg | \langle \text{BODY} \rangle)
  \]
  
  \[
  (Y_1 (\lambda fg \ | \ F)(\lambda fg \ | \ G))
  \]
  
  \[
  (Y_2 (\lambda fg \ | \ F)(\lambda fg \ | \ G))
  \]

- 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

\[
\langle \text{identifier} \rangle := \langle \text{alpha-char} \rangle \{(\text{alpha-char})|\langle \text{number} \rangle\}
\]

\[
\langle \text{constant} \rangle := \langle \text{number} \rangle | \langle \text{boolean} \rangle | \langle \text{char-string} \rangle
\]

\[
\langle \text{expression} \rangle := \langle \text{constant} \rangle | \langle \text{identifier} \rangle
\]

| (\lambda \langle \text{identifier} \rangle "|" \langle \text{expression} \rangle )
| (\langle \text{expression} \rangle^+)
| \langle \text{identifier} \rangle (\langle \text{expression} \rangle , \langle \text{expression} \rangle[^])
| \text{let} \langle \text{definition} \rangle \text{in} \langle \text{expression} \rangle
| \text{letrec} \langle \text{definition} \rangle \text{in} \langle \text{expression} \rangle
| \text{if} \langle \text{expression} \rangle \text{then} \langle \text{expression} \rangle \text{else} \langle \text{expression} \rangle
| \langle \text{arithmetic expression} \rangle

\[
\langle \text{definition} \rangle := \langle \text{header} \rangle = \langle \text{expression} \rangle
\]

| \langle \text{definition} \rangle \{\text{and} \ \langle \text{definition} \rangle\}[^]
\[
\langle \text{header} \rangle := \langle \text{identifier} \rangle
\]

| \langle \text{identifier} \rangle (\langle \text{identifier} \rangle \{\langle \text{identifier} \rangle \}, \langle \text{identifier} \rangle[^])

\[
\langle \text{abstract-program} \rangle := \langle \text{expression} \rangle
\]
Convenience: WHERE and WHHEREC

- Sometimes convenient to put definitions after usage
  \( \langle \text{BODY} \rangle \text{ WHERE } \langle \text{DEFINITION} \rangle \)

- Example
  
  \[
  \text{LET } a(r) = \pi \times r \text{ IN } \\
  a(10) \text{ WHERE } \pi = 3.1415
  \]

- Do we need brackets? No.
  - LET's \( \langle \text{BODY} \rangle \) 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
  
  \[
  \text{LET } x=x+2 \text{ IN } \langle \text{BODY} \rangle \not\equiv \text{LETREC } x=x+2 \text{ IN } \langle \text{BODY} \rangle
  \]

- Meaning overlaps when there is no self-reference
  
  \[
  \text{LET } x=2 \text{ IN } \langle \text{BODY} \rangle \equiv \text{LETREC } x=2 \text{ IN } \langle \text{BODY} \rangle
  \]

  \[
  \text{LET } y=2 \text{ IN } \equiv \text{LETREC } y=2 \text{ IN } \\
  \text{LET } x=y \text{ IN } \langle \text{BODY} \rangle \quad \text{LETREC } x=y \text{ IN } \langle \text{BODY} \rangle
  \]

- 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]) → [1 4 9 16]

Other Traditional Higher-order Functions

- Filter: apply a predicate to each item and return those items that satisfy
  (filter 'even [1 2 3 4]) → [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

\[ \text{LET } T = (λxy|x) \]
\[ \text{AND } F = (λxy|y) \]
\[ \text{AND } + = \ldots \]
\[ \text{IN } ⟨\text{BODY}⟩ \]

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, \( \lambda \)'s can be used anywhere in abstract programming
  
  \[
  \text{map}( (\lambda x| 2+x), [1 \ 2 \ 3]) \rightarrow [3 \ 4 \ 5]
  \]

- A more elegant method:
  
  Let \( pa \) be the partial application operator
  
  \[
  \text{LET } pa = (\lambda f \ x | (\lambda y | f \ x \ y)) \text{ IN } \langle \text{BODY} \rangle
  \]

  Allows us to write:

  \[
  \text{LET inc = } pa + 1 \text{ IN }; \text{ i.e. inc = } (\lambda y | (+ 1 \ y))
  \]

  \[
  \text{inc}(1) \rightarrow 2
  \]

- Or more impressively:
  
  \[
  \text{map( pa + 2, [1 \ 2 \ 3] )} \rightarrow [3 \ 4 \ 5]
  \]

- Partial application \( \equiv \) currying

Combinators as a Calculus

- The central operation in \( \lambda \)-calculus is the \( \beta \)-substitution

- It requires
  - scanning expressions for variables
  - analyzing free vs. bound variables
  - renaming when conflicts are discovered
  - rebuilding substituted copies of expressions repeatedly

- \( \lambda \)-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 \( \equiv \) string of combinators: \( ZXYZYY... \)
  - Suppose 2 argument combinator \( Z \) reverses its arguments \( ZXYZYY... \rightarrow YXZYY... \)
  - Suppose 1 argument combinator \( Y \) duplicates its arguments \( YXZYY... \rightarrow XXZYY... \)
  - Suppose 1 argument combinator \( X \) deletes its second argument \( XXZYY... \rightarrow XZYY... \)

- Combinators can be defined using \( \lambda \)-calculus: \( X \equiv (\lambda xy|x) \)

- Given combinators, no \( \lambda \)'s, formal parameters or substitution required

Dr. B. Price and Dr. G. Greiner

Combinators as a Calculus

- Left-associative like \( \lambda \)-calculus: \( ABCD... \equiv (((AB)C)D) \)

- Proved that two combinators can generate all others

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Name</th>
<th>A ( \lambda )-calculus Def</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s )</td>
<td>distribute</td>
<td>((\lambda xyz</td>
<td>xz(yz)))</td>
</tr>
<tr>
<td>( k )</td>
<td>constant</td>
<td>((\lambda xy</td>
<td>x))</td>
</tr>
</tbody>
</table>

- Identity function: \( I \equiv S \ K \ K \ A \equiv K \ A \ (K \ A) \equiv A \)

Dr. B. Price and Dr. G. Greiner

CMPUT 325: Abstract Programming
Common Combinators

- Common combinators can be defined using S and K

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Name</th>
<th>A $\lambda$-Calculus Def</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>compose</td>
<td>$(\lambda xyz</td>
<td>x(yz))$</td>
</tr>
<tr>
<td>C</td>
<td>reversal</td>
<td>$(\lambda xyz</td>
<td>xzy)$</td>
</tr>
<tr>
<td>W</td>
<td>duplicate</td>
<td>$(\lambda xy</td>
<td>xyy)$</td>
</tr>
</tbody>
</table>

- There is a mechanical mapping between $\lambda$-calculus and the minimal SKI combinator language consisting of only S,K and I combinators

- $\text{cons} \equiv B \ C \ (C \ I)$
- $\text{car} \equiv C \ I \ K$
- $\text{cdr} \equiv C \ I \ (K \ I)$
- $\text{integers: zero} \equiv K I$, $Z_i \equiv (s \ (s \ . \ . \ (s \ z) \ . \ .))$ with i copies of s
- $\text{successor}(n) \equiv (S \ B \ Z_i) \ s \ z$
- $\text{Factorial: } f(n) = (\text{if } (= n 0) \ 1 \ (\times n (f \ (- n 1))))$
  $\equiv (S(C(B \ \text{if } (C=0))) \ 1) \ (S \ x \ (B \ f \ (C - 1))))$
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
  \[ \text{MAP} (\div \space \text{SWAP} \space 2 \space ) \space L \]

- SWAP reverses arguments to \( \div \) so we get
  - each number divided by 2
  - instead of 2 divided by each number