Mar 13 00

Programming languages

- **Importance of good languages**

  – All **communication with computers** via some programming language, general purpose or application–specific

  – **User accessibility** to computers beyond application programs & their limitations

  – **Impact on thinking & on planning** software

  – **Impact on cost** of software, IBM & users

- **Current situation**

  – **Lack of interest** due to past failures to make major improvements in cost of programming

  – **What's wrong:** programs are complex, low level, restrictive, unstructured; program parts can't be reused

Our ambitious goals for the FL language

No language meets these goals yet

- Power

  - Programming an order of magnitude faster, cheaper & more reliable

  - Good for reasoning about programs

- Broad scope

  - Unified, universal language for

    - interactive, number-crunching, graphics

    - operating systems, database systems

- Speed

  - Programs run as fast as best Fortran or Lisp programs

What is function level programming?

- Referentially transparent

  $P = Q \Rightarrow F(P) = F(Q)$

  ↑ function equality

- Function level (vs object level)

  Object level definition ← application

  For all x, $f:x = g:(h:x)$

  ↑ object equality

  Function level definition

  ← function equality

  $f = g \cdot h = comp:\langle g,h \rangle$

- Build new functions with combining forms:

Example: Inner Product

IP:<<1,2,3>,<4,5,6>> =

+ ∘ α:x ∘ trans:<<1,2,3>,<4,5,6>>

+ ∘ α:x:<<1,4>,<2,5>,<3,6>>

+:<x:<1,4>, x:<2,5>, x:<3,6>>

+:<4, 10, 18> = 32

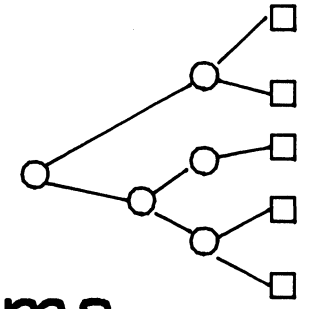
Why function level?

Engineering discipline:

combining forms = ○

- Structured programs

programs = □



- Off-the-shelf programs

- Put programs together so that understand parts
  $\Rightarrow$ understand whole
- General equational theorems
  substitute for progr vbls
- Correct progs $\Rightarrow$ useable progs

Combining forms

Composition

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

where $f \circ g = comp:\langle f,g \rangle$

Construction

tuple of all values

$[f_1,\ldots,f_n]:x = \langle f_1:x,\ldots,f_n:x \rangle$ ⟵

where $[f_1,\ldots,f_n] = cons:\langle f_1,\ldots,f_n \rangle$

T 4/11

Combining forms, cont'd

Condition

$$(p \rightarrow f;g):x = \begin{cases} f:x & \text{if } p:x = T \\ g:x & \text{if } p:x = F \\ ? \text{ (error)} & \text{ow} \end{cases}$$

Predicate construction

$$[\![p_1,\ldots,p_n]\!]:\langle x_1,\ldots,x_n\rangle = T \quad \text{if } p_i:x_i = T$$

$$[\![[\![\text{isint},\text{isint}]\!], \text{ischar}]\!]:\langle\langle 3,4\rangle,\text{"a}\rangle = T$$

T5

Example: factorial

- **In Pascal:**

```
function fact(n : integer) : integer ;
var i, result : integer ;
begin
  result := 1 ;                    (housekeeping = ■ )
  for i := 1 to n  do
     result := result * i ;
  fact := result
end;
```

- **Recursive, Pascal:**

```
function fact(n : integer) : integer ;
begin
  if n=0 then fact := 1
         else fact := n * fact(n−1)
end ;
```

- **In FL:**

```
Def fact ≡  * ∘ intsto
Sig fact :: isint ⟹  isint        (optional)
```

fact:4 = (* ∘ intsto):4 = *:<1,2,3,4> = 24

Example: Inner Product

- **In Pascal (without its restriction)**

```
function  IP(a,b : array[1..n]) : real ;
var i:integer, result:real ;
begin                    efficient, unclear, non-reusable
  result := 0.0 ;
  for  i := 1 to n  do
     result := result + a[i]*b[i] ;
  IP := result
end
```

(housekeeping = ■)

- **In FL**

Def IP ≡ + ∘ α :* ∘ **trans**

inefficient, clear, reusable

algebraic optimization yields efficient program like the Pascal program

Functional programming

- **Functional vs conventional programs**

  – Conventional program: changes complex state

  – Functional program: transforms arguments into result

- **Good properties of functional languages**

  - referential transparency
  - powerful, clean, high level
  - easier to reason about

  Lots of activity & interest in field

- **Languages: Lisp, ML, SASL, FP, others**

  General problems:

  slowness
  difficulties with I/O
  lack of type facilities
  some not entirely functional
  complexity
  lack of expressive power

Strategy to reach goals

- **Power**
  - radically new programming style
  - combining forms for building programs
  - reusable housekeeping operations
  - sacrifice efficiency at language level
    but get it back by optimization
  - interactive I/O
  - expressive features
- **Broad scope**
  - data type facilities, higher order functions
  - provision for variety of I/O devices
  - file system
- **Speed**
  - algebraic type analysis
  - algebraic program optimization

Algebra, combining forms

composition ↔ construction

$$[f_1,\ldots,f_n]\circ h = [f_1\circ h,\ldots,f_n\circ h]$$

composition ↔ condition

$$(p \to f;g)\circ h = p\circ h \to f\circ h;\ g\circ h$$

$$h\circ(p \to f;g) = p \to h\circ f;\ h\circ g$$

construction ↔ condition

$$[..(p \to f;g)..] = p \to [..f..];[..g..]$$

α & composition ↔ construction

$$\alpha{:}h \circ [f_1,\ldots,f_n] = [h\circ f_1,\ldots,h\circ f_n]$$

Non-recursive defn style

$$\text{len} = + \circ \alpha:\tilde{}1$$

$$\text{len}:\langle 3,4,5\rangle = (+ \circ \alpha:\tilde{}1):\langle 3,4,5\rangle$$

$$= + : \langle \tilde{}1:3, \tilde{}1:4, \tilde{}1:5\rangle$$

$$= + : \langle 1,1,1\rangle$$

$$= 3$$

Algebraic reasoning about programs

Prop: $len \circ al \circ [f,g] = + \circ [\tilde{}1, len \circ g]$ when f is defined.

Proof:

$$len \circ al \circ [f,g] = + \circ (\alpha : \tilde{}1) \circ al \circ [f,g]$$

← def of len

$$= + \circ al \circ [\tilde{}1 \circ f, (\alpha : \tilde{}1) \circ g]$$

← since $\alpha h \circ al \circ [f,g] = al \circ [h \circ f, \alpha h \circ g]$

$$= + \circ al \circ [\tilde{}1, (\alpha : \tilde{}1) \circ g]$$

← since $\tilde{}k \circ f = \tilde{}k$, f defined

$$= + \circ [\tilde{}1, + \circ (\alpha : \tilde{}1) \circ g]$$

← since $+ \circ al \circ [f,g] = + \circ [f, + \circ g]$

$$= + \circ [\tilde{}1, len \circ g]$$

← by defn of len

□

Dilemma about Housekeeping Operations

- FL resolves tension between programming power and execution speed in favor of programming power.
  - ♦ Simple and clear (but inefficient) housekeeping operations
- FL relies on optimization to turn simple and clear (but inefficient) housekeeping operations into efficient (complex) ones.

FL vs Fortran

- **Programs in FL do housekeeping by rearranging data**
  - ♦ **Reusable housekeeping operations (+)**
  - ♦ **Inefficient since much data copying (-)**
- **Programs in Fortran do housekeeping by explicit repetition and control structures**
  - ♦ **Housekeeping operations are not reusable (-)**
  - ♦ **Efficient (+)**
    1. **little data copying**
    2. **few intermediate results**
- **Goal: How to make FL programs run efficiently without data copying?**

General Optimization Technique

- Add new constructs to FL
  - Straightforward, efficient implementations
  - Used (extensively) by compiler
  - Not used by the programmer
- Develop algebraic laws for the new constructs
- Given an FL function F, the compiler uses syntactic information to find a form C (built using the new constructs) that is the identity on the domain of F.
  - $F \circ C = F$
- Using algebraic laws, the compiler transforms $F \circ C$ to F'.
  - Goal: to have F' expressed in terms of the new efficient constructs.

Optimization: factorial

- **What needs optimizing?**

  Recall fact = * · intsto

  In * ∘ intsto the formation of the sequence of integers from 1 to n is waste motion

- **What does our strategy call for?**

  The rightmost primitive is intsto:
  find construct C where intsto = intsto ∘ C
  OR find C where intsto = C
  a table provides constructs for each primitive

- **Optimization**

  choose C = {~i i=id} = intsto
  since {~i i=id}:n = [~1,...,~n]:n =<1,...,n>

  Program * ∘ intsto = * ∘ C is equivalent to the Fortran program:

```
result := 1
for i = 1 to n
    result := result * ~i:n (= result*i)
```

Optimization: Inner product

- **The identity construct for IP**

$$\{\{S{:}i \circ S{:}j \;\; i{=}len \circ s1\} \;\; j{=}\sim 2\} \;=\; C$$

is the identity for pairs of equal length seqs:

a subset of the args of trans

$$\{\{S{:}i \circ S{:}j \;\; i{=}len \circ s1\} \;\; j{=}\sim 2\} : \langle x,y\rangle =$$

$$\langle \{S{:}i \circ S{:}1 \;\; i{=}len \circ s1\}{:}\langle x,y\rangle,\; \{S{:}i \circ S{:}2 \;\; i{=}len \circ s1\}{:}\langle x,y\rangle\rangle =$$

$$\langle\langle S{:}1{:}x,\ldots,S{:}(len{:}x){:}x\rangle,\langle S{:}1{:}y,\ldots,S{:}(len{:}x){:}y\rangle\rangle$$

$$= \langle x,y\rangle \quad \text{if } len{:}x = len{:}y$$

- **Optimization of IP**

$$IP \circ C \;=\; +\circ \alpha{:}* \circ trans \circ C$$

$$= +\circ \alpha{:}* \circ \{\{S{:}i \circ S{:}j \;\; j{=}\sim 2\} \;\; i{=}len \circ s1\}$$

since $trans \cdot \{\{E(i,j) \;\; i{=}r\} \;\; j{=}s\} = \{\{E(i,j) \;\; j{=}s\} \;\; i{=}r\}$

$$= +\circ \{*\circ \{S{:}i \circ S{:}j \;\; j{=}\sim 2\} \;\; i{=}len \circ s1\}$$

since $\alpha{:}h \cdot \{E(i) \;\; i{=}r\} = \{h \cdot E(i) \;\; i{=}r\}$

$$= +\circ \{*\circ [S{:}i \circ s1,\; S{:}i \circ s2] \;\; i{=}len \circ s1\}$$

by Lem 11 Prf, since $s1 = s1$, $s2 = s2$

Optimization of IP cont'd

- **Result of optimization**

IP = + ∘ {*∘[S:i ∘ s1, S:i ∘ s2] i=len∘ s1}

Thus

IP:<x,y> = + : (*:<S:i:x, S:i:y> i=1 to len:x)

Optimized IP equivalent to efficient Fortran program

IP:<x,y> equivalent to Fortran

```
result := 0.0
for i = 1 to len:x
    result := result + x[i]*y[i]
```

optimized FL program does none of the data rearranging or formation of intermediate results specified by programmer in his FL program

Status of FL, 1988

- **Language complete**

  new manual & paper — May 1, 1988

- **Bootstrap compiler**

  Lisp compiler complete for 87 language
  updated Lisp compiler for FL: soon

- **Techniques for optimization, type analysis**

  examples (to follow)

- **Personnel**

  John Backus (since 1970)
  John Williams (since 7/78)
  Ed Wimmers (since 9/83)
  Peter Lucas (since 2/88)

Comment: much larger effort on other languages: less on design, more on implementation, fewer hard language problems (eg, keeping strong algebra)

Scientific Contributions

Areas

- **Language design and exposition - 6**
- **Algebra of programs - 4**
- **Mathematical properties of languages - 6**

Distribution

- **JACM - 1**
- **CACM - 1**
- **TOPLAS - 1**
- **LNCS - 3**
- **Book chapters - 3**
- **POPL - 3**
- **LICS - 2**
- **RJ's - 2**

External Activities

Program Committees

J. Backus

1. FPCA 1981
2. Lisp and FP 1982

J.H. Williams

1. POPL 1984
2. Lisp and FP 1984
3. FPCA 1985
4. Lisp and FP 1986 (prog. com. chairman)
5. FPCA 1987 (general chairman)

Standing Committees

J. Backus

- National Academy of Sciences
- National Academy of Engineering
- American Academy of Arts and Sciences
- IFIP WG2.2 member
- IFIP WG2.8 member

P. Lucas

- **IFIP WG2.2 member**
- **Editorial board: Acta Informatica, Computer Languages**
- **Honorary Professor: Johannes Kepler University, Austria**

J.H. Williams

- **IFIP WG2.2 member**
- **IFIP WG2.8 chairman**

Summary

- Goals
  - power
  - scope
  - speed
- Basic Philosophy
  - function level style
  - mathematics of programs
  - power and simplicity
  - structured programs
- Issues
  - optimization
  - I/O
  - data types
  - higher order functions

Potential impact

- **Big savings in cost of programming**
  - to IBM
  - to customers
- **Big reduction in time to produce software**
- **Wider accessibility of computers & programming**
  - greatly increased use & sales of computers
- **Programs more reliable, maintainable**
- **IBM regains leadership in languages**
- **Contributions to computer science**

Future Plans

■ First stage

- **checkpoint: 6/91**
- **Research: type analysis & optimization**
- **Implementation: User–friendly system for RT**
  - not highly optimized
  - but design data structures, etc = basis for final compiler prototype
- **Manpower: current 4 + 1 more (perhaps + 1 in 3rd year) = total of 5 (maybe 6 in 3rd yr)**
- **Objectives:**
  - test power of language with other users
  - produce basic optimization & type analysis strategies
  - test power of language by writing compiler in FL

Future Plans, cont'd

- Second stage
  - If first stage judged successful
  - Write highly optimizing compiler
    - use techniques developed in 1st stage
  - Test system with other users
- Manpower & scheduling:
  - another 5–6 people (tot = 10 – 12) over 3 years (1991 to 94)
  - completion: 1994–95