Item # 188

# FL Project Status

April 8, 1991 McGrodd's

## ■ Personnel

RSMs (4): Alex Aiken, Peter Lucas, Ed Wimmers, John Williams

Systems (1): Thom Linden

Manager (1): John Backus

## ■ FL Language

Language complete, with manual

## ■ FL Optimizing Compiler

Implements complete language

Precise type analysis for small–med programs

Optimizes many small programs

Produces Common Lisp code

## ■ FL Runtime System

Common Lisp runtime system

# Primary Goal of FL Project

**Solve problem of writing clear, high level but slow specifications and transforming them into fast product programs**

**New FL contributrions to a solution**

- Paradigm for expressing clear but slow specifications/programs
- Mathematical properties of paradigm = basis for optimization
- Non–restrictive type safety technology
- Optimization technology using rewriting

**Why hasn't this problem been solved?**

- No general programming paradigm
- Paradigms lack mathematical basis for optimization
- It is a very hard problem

# Prototyping factorial

## Specification

"factorial of a positive integer n is the product of the integers from 1 to n; factorial of 0 is 1"

## Prototype FL program

```
Def fact ← isnonnegint ≡ * ∘ intsto

    fact:4 = (* ∘ intsto):4
           = *:<1,2,3,4> = 24
    fact:0 = (*∘intsto):0 = *:<> = 1
```

## C program

```
fact(arg)  int arg;
  {int ans, count;
ans = 1 ;
for (count = 1; count <= arg; count++)
  ans *= count ;
return(ans) ; }
```

# The FL Compiler

## Parser

De-sugars program; program in tree form

## Simplifier

Program in very simple form, flat name space

## Type analysis

Determines argument & result types

Marks functions "safe" that can't make errors

## Rewriter-optimizer

Transforms program by applying rewrite rules

Depends on information from type analysis

Result: extended FL with "Fortran" constructs

## Code generator

Transforms optimized EFL code into Lisp (ultimately into C)

# Optimizing factorial

```
def fact ← isnonnegint ≡  * ∘ intsto
```

**Parser & simplifier:**

```
isnonnegint → * ∘ intsto; errmsg

        errmsg = signal∘[~""fact",~""arg 1",id]
```

**Type analysis:**

```
isnonnegint_sf → *_sf ∘ intsto_sf; errmsg
```

**Rewriter:**

```
insto_sf ⇒ for:<s1_sf, id>

isnonnegint_sf → *_sf ∘ for:<s1_sf, id>;
                                    errmsg

*_sf ∘ for:<$f_sf,$g> ⇒
            reduce:<(*∘[s1,$f∘s2])_sf, ~1, $g>
with $f = s1  and $g = id

isnonnegint→reduce:<*∘[s1,s1∘s2],~1,id>;
                                    errmsg
```

**Result program:**

```
if nonnegint:arg then
   acc = 1
   for i = 1 to arg
      acc = acc * i
   return acc
else errmsg:arg
```

# Protyping matrix multiply

## Specification

"The product of matrices A and B is a matrix C such that $c_{i,j}$ = inner product of ith row of A and jth column of B"

## Approach

Build structure with row—column pairs in the right place, then apply inner product to each pair.

whose ith row comprises the inner products of the ith row of A with each of the columns of B.

# Prototyping matrix multiply

Def MMPY←⟦a.,b.⟧ ≡ α:(α:IP)∘(a rcpairs trans∘b)

where {Def rcpairs ≡ lift:(α:distl ∘ distr) }

MMPY : <(▬▬▬)(▬▬▬)> =

α:(α:IP)((▬▬▬) rcpairs trans :(▬▬▬)) =

α:(α:IP)((▬▬▬) rcpairs (1 2 3)) =

< <<▬ 1><▬ 2><▬ 3>>

α:(α:IP) : <<▬ 1><▬ 2><▬ 3>>

<<═ 1><═ 2><═ 3>> >

# Matrix multiply optimized

## Original program:

MMPY • Build

## Output of the optimizer:

```
E_0 ==
 isseq -> lenis:3 -> isposint @ s:1 ->
   isposint @ s:2 -> isposint @ s:3; ff; ff; ff; ff
   -> for:<iszero@s:2@s:2 -> ~<> ;
         for:<add @
               for:<mul@[mul@[~1, s:1], s:1@s:2],
                                    s:2 @ s:2 @ s:2>,
                s:3 @ s:2>,
            s:1> ;
    signal @ [~"build", ~"arg1", id]
```

## A really optimal program

Compiler must "understand" MMPY & Build

# Summary, status

## FL high-level paradigm:

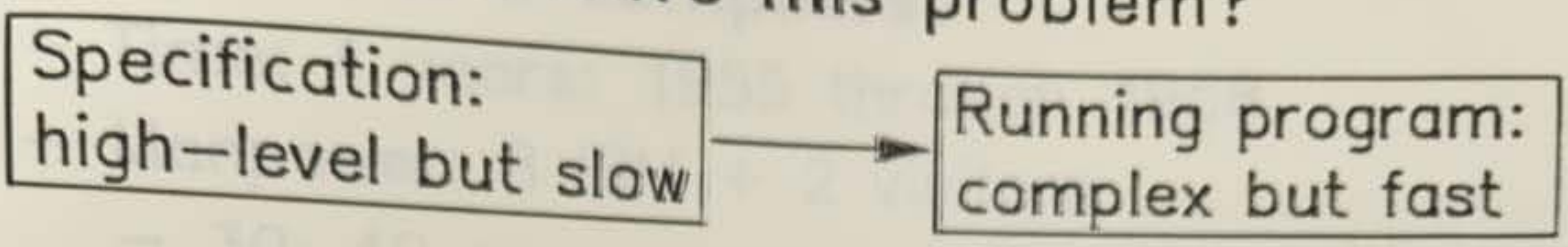
- Build programs with combining forms that preserve understanding
- High-level data rearrangements replace loops & indexing
- "The right way to think about a problem is the wrong way to do it"
- Mathematical properties of combining forms = basis for optimization
- Rewrite rules = large knowledge base about transforming specs into fast low-level code

## Turning slow high-level specs into fast product programs. Can we do it?

- Type analysis: precise info for small-med programs
- Optimization: turns small-med programs into fast code

# *FL*'s future

Why strive to solve this problem?




- Order of magnitude decrease in time and cost of software development cycle
- Improve competitiveness & profitability of software business – most rapidly growing
- Customers get valuable productivity tool
- Have some evidence that it is soluble:
  FI paradigm & technology work for examples
- IBM leadership in software area
- Ad hoc methods won't solve general problem

**Challenges:**

- Complete technical work
- Acceptance: learning hurdle, new paradigm
- Very hard problem; can we represent enough programming knowledge as rules/strategies?

# Manpower: Fortran vs FL

## Fortran I & II compilers

- Time: 4 years: 1955 through 1958
- Manpower: 8 IBM + 2 visitors
  = 30–40 man–years

## FL compiler

- Time: 3 years: 4/88 to 4/91
- Manpower(now): 4 RSM (+1 systems, 1 mgr)
  = 14.75 man–years (to date)
- Difficulty ratio, FL/Fortran = 10
- Potential economic impact of FL:
  x10 on universe 50–1,000x larger than
  the universe Fortran impacted
- Probable manpower to complete prototype:
  10–15 man–years
- Time to complete prototype:
  4 people = 2.5 – 4 years
  6 people = 1.7 – 2.5 years