15-150 Fall 2013

Stephen Brookes

LECTURE 1
Tuesday, August 27

Functional programming

What is SML?

- A *functional* programming language

  computation = evaluation

- A *typed* functional language

  only *well-typed* expressions are evaluated

- A *polymorphic* typed functional language

  well-typed expressions have a *most general type*

- A *call-by-value* language

  function calls evaluate their argument

Features

- Functional programs are *referentially transparent*

  safe substitution for *equivalent* code

- Functional programs are *mathematical objects*

  use math techniques to prove correctness

  use *induction* to analyze *recursive* code

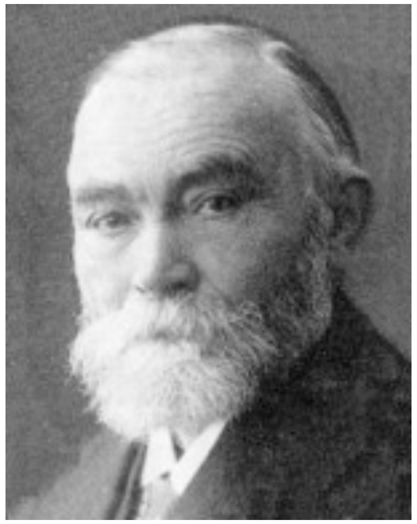
- Functions are *values*

  can be used as arguments or results

  can be used in lists, tuples, ...

Referential transparency

- The *value* of an expression depends only on the *values* of its sub-expressions
- The *type* of an expression depends only on the *types* of its sub-expressions



Equivalence

- Expressions of type **int** are *equivalent* if they evaluate to the same integer
- Expressions of type **int list** are *equivalent* if they evaluate to the same list of integers
- Functions of type **int -> int** are *equivalent* if they map *equivalent arguments* to *equivalent results*

Equivalence is a form of *semantic equality*

Equivalence

- 21 + 21 is *equivalent* to 42
- [2,4,6] is *equivalent* to [1+1, 2+2, 3+3]
- fn x => x+x is *equivalent* to fn y => 2*y

21 + 21 = 42

fn x => x+x = fn y => 2*y

(fn x => x+x) (21 + 21) = (fn y => 2*y) 42

We use = for *equivalence*

Don't confuse with = in ML

Equivalence

- For every type t there is a notion of *equivalence* for expressions of that type
  - We usually just use $=$
  - When necessary we use $=_t$

So far we talked about

$$=_{\text{int}}$$

$$=_{\text{int list}}$$

$$=_{\text{int -> int}}$$

Compositionality

- In any functional program, replacing an expression by an *equivalent* expression produces an *equivalent* program

The key to
compositional reasoning
about programs

Parallelism

- Expression evaluation has ***no side-effects***
  - evaluation order typically has *no effect* on the *value* of an expression
  - can evaluate *independent* code *in parallel*
- Parallel evaluation may be *faster* than sequential

Learn to *exploit* parallelism!

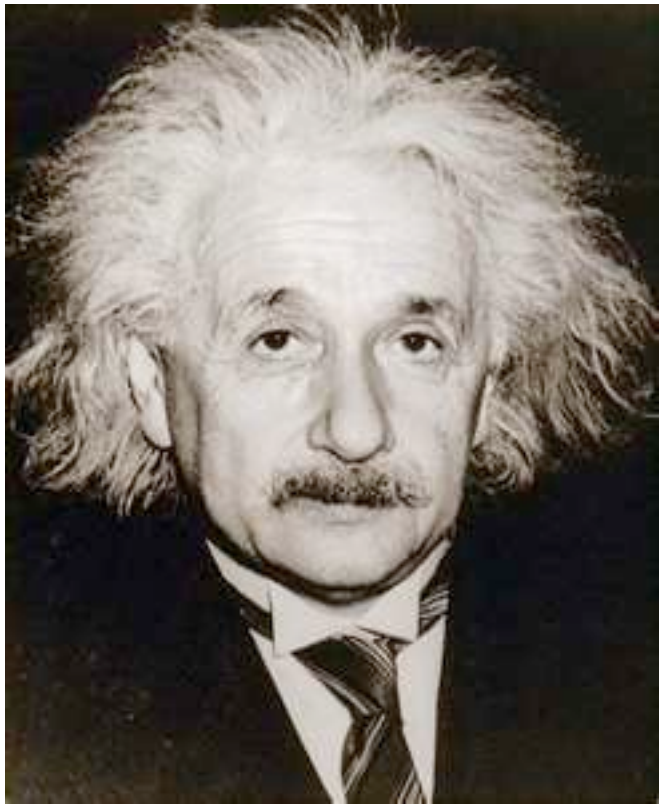
Principles

- Expressions must be well-typed.
  Well-typed expressions don't go wrong.
- Every function needs a specification.
  Well-specified programs are easy to understand.
- Every specification needs a proof.
  Well-proven programs do the right thing.

**Those are my principles,
and if you don't like them...
well, I have others.**

Principles

- Large programs should be designed as modules.
  Well-interfaced code is easier to maintain.
- Data structures algorithms.
  Choice of data structure can lead to better code.
- Think parallel, when feasible.
  Parallel programs may go faster.
- Strive for simplicity
  As simple as possible, but no simpler



sum

```
fun sum [ ] = 0
 |    sum (x::L) = x + sum(L);
```

- sum : int list -> int
- sum [1,2,3] = 6
- For all L:int list,
  sum(L) = the sum of the integers in L

sum

```
fun sum [ ] = 0
 |   sum (x::L) = x + sum(L);
```

sum [1,2,3]
  = 1 + sum [2,3]
  = 1 + (2 + sum [3])
  = 1 + (2 + (3 + sum [ ]))
  = 1 + (2 + (3 + 0))
  = 6.

equational reasoning

count

```
fun count [ ] = 0
  |   count (r::R) = (sum r) + (count R);
```

- count : (int list) list -> int
- count [[1,2,3], [1,2,3]] = 12
- For all R : (int list) list,
  count(R) = the sum of the ints in the lists of R.

count

Since

sum [1,2,3] = 6

and

count [[1,2,3], [1,2,3]]

= sum[1,2,3] + sum [1,2,3]

equational reasoning

it follows that

count [[1,2,3], [1,2,3]]

= 6+6

= 12

sum’

```
fun sum’ ([ ], a) = a
 | sum’ (x::L, a) = sum’ (L, x+a);
```

- sum’ : int list * int -> int
- sum’ ([1,2,3], 4) = 10
- For all L:int list and a:int,

  sum’ (L, a) = sum(L)+a

Sum

```
fun sum' ([ ], a) = a
 |  sum' (x::L, a) = sum' (L, x+a);

fun Sum L = sum' (L, 0);
```

- Sum : int list -> int
- Sum and sum are *extensionally equivalent*

  For all L:int list, Sum L = sum L.

Hence...

```
fun count [ ] = 0
  |   count (r::R) = (sum r) + (count R);

fun Count [ ] = 0
  |   Count (r::R) = (Sum r) + (Count R);
```

- Count and count are *extensionally equivalent*

  For all R: (int list) list, Count R = count R.

Evaluation

fun sum [ ] = 0

| sum (x::L) = x + sum(L);

sum [1,2,3] =>* $_{[x:1,\ L:[2,3]]}$ (x + sum(L))

=>* 1 + sum [2,3]

=>* 1 + (2 + sum [3])

=>* 1 + (2 + (3 + sum [ ]))

=>* 1 + (2 + (3 + 0))

=>* 1 + (2 + 3)

=>* 1 + 5

=>* 6

Evaluation

count [[1,2,3], [1,2,3]]

=>* sum [1,2,3] + count [[1,2,3]]

=>* 6 + count [[1,2,3]]

=>* 6 + (sum [1,2,3] + count [ ])

=>* 6 + (6 + count [ ])

=>* 6 + (6 + 0)

=>* 6 + 6

=>* 12

Analysis

- sum(L) takes time proportional to the length of L
- count(R) takes time proportional to the sum of the lengths of the lists in R

(sequential evaluation)

Addition

- + is *associative*
- + is *commutative*
- The order in which we combine additions doesn't affect the result

Using parallelism

```
fun parcount R = reduce (op +) (map sum R)
```

parcount [[1,2,3], [4,5], [6,7,8]]

=>* reduce (op +) [sum [1,2,3], sum [4,5], sum [6,7,8]]

=>* reduce (op +) [6, 9, 21]

=>* 36

Analysis

- Let R be a *row* list of length k,
  each row an integer list of length n/k
- *If we have enough parallel processors*,
  parcount(R) takes time proportional to k + n/k

work and **spam**

We will introduce techniques for analysing

- *work* (sequential runtime)
- *span* (= optimal parallel runtime)

of functional code...

Themes

- **functional programming**
- correctness, termination, and performance
- types, specifications and proofs
- evaluation, equivalence and referential transparency
- compositional reasoning
- exploiting parallelism

Objectives

- Write functional programs
- Write specifications, and use rigorous techniques to prove correctness
- Learn techniques for analyzing sequential and parallel running time
- Choose data structures and exploit parallelism to improve efficiency
- Structure code using abstract types and modules, with clear interfaces

Next

- Lab tomorrow
- Homework 1 out tomorrow
- Homework 1 due Tuesday, 3 Sept

Why ML?

More difficult to shoot your foot off

It's too easy in C

Harder in C++ but blows your whole leg off

Can only shoot your foot in ML if

(a) the foot and gun are *well-typed*

and (b) the gun uses the *right type* of bullets

Moreover, if your foot is *polymorphic*
there's a *most general* way to shoot it

Why Functional Programm Will No e Over The World

Functional programming is really cool. Defin ctions and passing round is a very powerful paradigm. However, functional programming is about expressing ram as a set of func . . This is effectively a static mathematical definition ... . The computer will then convert that static de into dynamic runti e which does the computation ... .

My concern with functional progra do, we also have to convert this static definition into an internal re y 'run' that program in our heads.

```
fun fact 0 = 1
  | fact n = n * fact
```

The above piece of ML is a ve
To understand it one has to sa

1. A number n is taken an
2. If n is zero, then 1 is re
3. If n is non zero, the ev
4. Eventually, the numbe ers between n and 1.

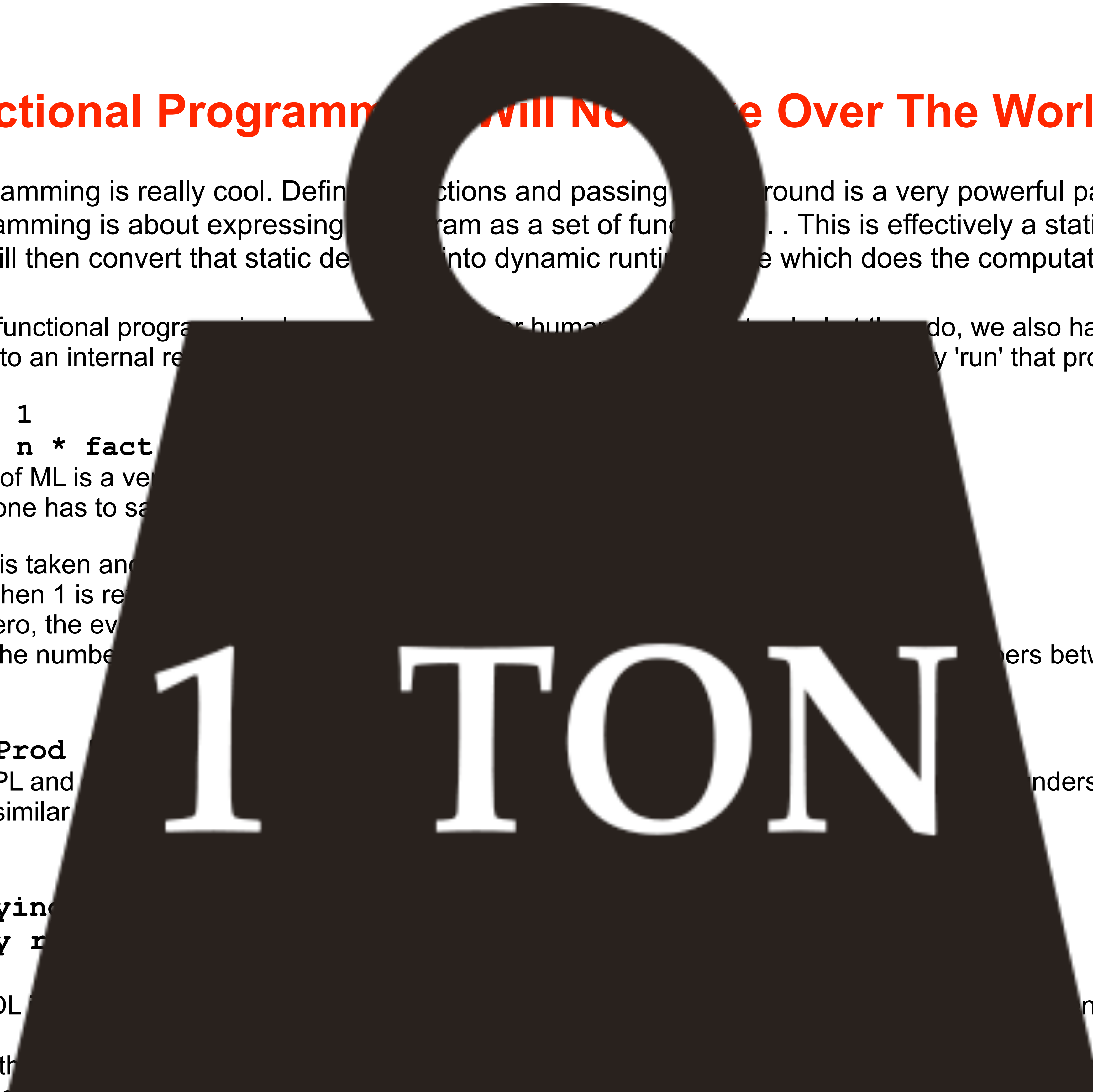
```
1 !r
2,?n,{ ,?r:Prod
```

The above is VSPL and nderstand and the code looks somewhat more similar

```
move 1 to r
perform varyin
    multiply r
end-perform
```

The above COBOL ntal translation.

My conclusion is th to dominate coding - because humans think that way.



nerds-central.blogspot.com

Why Functional Programming Will Take Over The World

Functional programming is really cool. Defining functions and passing them around is a very powerful paradigm. Functional programming is about expressing a program as a set of functions ... . This is effectively a static mathematical definition ... . The computer will then convert that static definition into dynamic runtime code which does the computation ... .

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

The above piece of ML is a very simple and clear definition of how to compute a factorial.