Topic A
Dataflow Model of Computation – Part 2

CPEG 852 - Spring 2014
Advanced Topics in Computing Systems

Guang R. Gao
ACM Fellow and IEEE Fellow
Endowed Distinguished Professor
Electrical & Computer Engineering
University of Delaware
ggao.capsl@gmail.com
Outline

• Parallel Program Execution Models
• Dataflow Models of Computation
• Dataflow Graphs and Properties
• Three Dataflow Models
  – Static
  – Recursive Program Graph
  – Dynamic
• Dataflow Architectures
Dataflow Models

• Static Dataflow Model
• Recursive Program Graphs
• Tagged Token Dataflow Model
  – Also known as *Dynamic*
Static Dataflow Model

• “...for any actor to be enabled, there must be no tokens on any of its output arcs...”

• So-called: at most “one-token-per-arc” rule
Conditional Expression

if \( p(y) \)  
{  
    f(x,y);
}  
else  
{  
    g(y);
}
Example
Power Function

\[
Y = X^N
\]

```c
long power(int x, int n){
    int y = 1;
    for(int i = n; i > 0; --i)
        y *= x;
    return y;
}
```
Example

Power Function

\[ Y = x^n \]

Example Power Function

\[ \text{return} \]

\[ Y = x^n \]
Example Power Function

\[ Y = 2^3 \]
**Example Power Function**

\[ Y = 2^3 \]
Example
Power Function

$Y = 2^3$
Example
Power Function

\[ Y = 2^3 \]
Example Power Function

\[ Y = 2^3 \]
Example Power Function

\[ Y = 2^3 \]

\[
\begin{align*}
\text{if } i > 0 \\
\text{return } 2^n - 1
\end{align*}
\]
Example
Power Function

\[ Y = 2^3 \]
Example Power Function

$Y = 2^3$
Example Power Function

\[ Y = 2^3 \]
Example Power Function

\[ Y = 2^3 \]
Example Power Function

\[ Y = 2^3 \]
Example
Power Function

\[ Y = 2^3 \]
Example Power Function

\[ Y = 2^3 \]
Example Power Function

\[ Y = 2^3 \]
Example Power Function

\[ Y = 2^3 \]
Example Power Function

\[ Y = 2^3 \]
Homework: Vector Addition

for(i = 0; i < N; ++i)
    c[i] = a[i] + b[i];
Static Dataflow Model

Features

• One-token-per-arc
• Deterministic merge
• Conditional/iteration construction
• Consecutive iterations of a loop appear to be only subjected to sequential execution (?)
• A dataflow graph ➔ activity templates
  – Opcode of the represented instruction
  – Operand slots for holding operand values
  – Destination address fields
• Token ➔ value + destination
Initial State

Static Dataflow

\[ x^2 \]

\[ -1 \]

\[ 0.5x \]

\[ + \]

\[ x \]

\[ \text{Data Arc} \]

\[ \text{ACK Arc} \]

\[ \text{Data Token} \]

\[ \text{ACK Token} \]
Static Dataflow

\[ x^2 + 0.5x - 1 \]
Actors can be fired since ACK tokens are present in their outputs.
Static Dataflow

\[ x^2 - 1 + 3 = 9 \]
\[ 0.5x \]

\[ x \]
Static Dataflow

\[ x^2 + (-1) + 0.5x \]
Actors must wait until having ACK tokens in their outputs to be fired.
Static Dataflow

\[ x^2 + 0.5x - 1 \]
Static Dataflow

\[ x^2 + 0.5x - 1 \]
Static Dataflow

This actor is waiting for ACK token
Static Dataflow

Which also delays the firing of this actor
Static Dataflow

The diagram illustrates a dataflow network with nodes labeled as follows:

- \( x^2 \)
- \(-1\)
- \(+\)
- \(0.5x\)
- \(X\)

Connections and values:

- \( x^2 \) connects to \(+\) and \(-1\) with values 25 and 1, respectively.
- \(-1\) connects to \(+\) with a value of 6.
- \(+\) connects to \(0.5x\) and \(X\) with values 12 and 6, respectively.
- \(0.5x\) connects to \(X\) with a value of 6.
- \(X\) connects to the output.
Static Dataflow

\[ x^2 - 1 + 0.5x \]

Now it can be fired
Static Dataflow

\[ x^2 - 1 + 0.5x \times 72 \]
This actor can now be fired too since it has ACK tokens on each output
Static Dataflow

\[ x^2 + (0.5x - 1) = 26 \]
Static Dataflow

\[
\begin{align*}
\mathcal{X}^2 + 0.5\mathcal{X} + 13 & \\
\mathcal{X} & \\
-1 & \\
\end{align*}
\]
Static Dataflow

\[ x^2 - 1 + 0.5x \]
Static Dataflow

\[
\begin{align*}
&x^2 \\
&-1 \\
&x + 0.5x \\
&x
\end{align*}
\]
Initial Configuration is reestablished
Static Dataflow Model Features

• **Deficiencies:**
  – Due to acknowledgment tokens, the token traffic is doubled.
  – Lack of support for programming constructs that are essential to modern programming language.
  – No procedure calls.
  – No recursion.

• **Advantage:**
  – Simple model.
Update unit

Operation units

Fetch unit

queue

Activity store

Z = (a + b) * (c - d)

Gao and Theobald: The Static Dataflow Enable Memory Chip (fall, 1984)
Instructions in a Static Dataflow Architecture

CPEG852-Spring14: Topic A - Dataflow - 2
Dataflow Software Pipelining

\[ \text{for } i \text{ in } [1, n] \]
\[ H[i] = a * X[i] + Y[i] \]
\[ G[i] = b * U[i] + V[i] \]
\text{end for}
Recursive Program Graphs

- Outlaw iterations:
  - Graph must be acyclic
- One-token-per-arc-per-invocation
- Iteration is expressed in terms of a tail recursion
- [Jim Rambaugh 1977, MIT PhD Thesis]
Tail Function Application

• Tail-procedure application
  – a procedure application that occurs as the last statement in another procedure;

• Tail-function application is a function application (appears in the body expression) whose result value is also returned as the value of the entire functions

• Consider the role of stack
Factorial

```c
long fact(n){
    if(n == 0) return 1;
    else return n * fact(n-1);
}
```

```c
long fact_1(n, p){
    if(n == 0) return p;
    else return fact_1(n-1, n*p);
}
```

- **Normal Recursive**
- **Tail Recursive**
Factorial
The Normal Version

long fact(n){
    if(n == 0) return 1;
    else return n * fact(n-1);
}

Hand Simulate fact(3)
Apply fact*

```java
long fact(n)
{
    if(n == 0) return 1;
    else return n * fact(n-1);
}
```

Hand Simulate fact(3)
Factorial

The Normal Version

long fact(n){
    if(n == 0) return 1;
    else return n * fact(n-1);
}

Hand Simulate fact(3)
Factorial
The Normal Version

long fact(n){
    if(n == 0) return 1;
    else return n * fact(n-1);
}
Factorial
The Normal Version

Hand Simulate fact(3)

long fact(n){
    if(n == 0) return 1;
    else return n * fact(n-1);
}

Apply fact
long fact(n) {
    if (n == 0) return 1;
    else return n * fact(n - 1);
}

Hand Simulate fact(3)
Factorial
The Normal Version

```java
long fact(n) {
    if (n == 0) return 1;
    else return n * fact(n - 1);
}
```

Hand Simulate fact(3)

Apply fact

long fact(n) {
    if (n == 0) return 1;
    else return n * fact(n - 1);
}
Factorial
The Normal Version

long fact(n){
    if(n == 0) return 1;
    else return n * fact(n-1);
}

Hand Simulate fact(3)
The Normal Version

```c
long fact(n){
    if(n == 0) return 1;
    else return n * fact(n-1);
}
```

3 * fact(2)
Factorial
The Normal Version

long fact(n) {
    if (n == 0) return 1;
    else return n * fact(n-1);
}

3 * fact(2)
Factorial
The Normal Version

long fact(n){
    if(n == 0) return 1;
    else return n * fact(n-1);
}

3 * fact(2)
Factorial
The Normal Version

long fact(n){
    if(n == 0) return 1;
    else return n * fact(n-1);
}

3 * fact(2)
Factorial
The Normal Version

```c
long fact(n){
    if(n == 0) return 1;
    else return n * fact(n-1);
}
```

3 * fact(2)

Diagram of the factorial function with a flowchart.
Factorial
The Normal Version

```c
long fact(n){
if(n == 0) return 1;
else return n * fact(n-1);
}
```

3 * fact(2)

Apply fact

n

n==0

F

T

1

-1

2

*
Factorial
The Normal Version

```java
long fact(long n)
{
    if (n == 0) return 1;
    else return n * fact(n - 1);
}
```

3 * fact(2)

long fact(n){
    if(n == 0) return 1;
    else return n * fact(n-1);
}
Factorial
The Normal Version

long fact(n){
    if(n == 0) return 1;
    else return n * fact(n-1);
}

3 * fact(2 * fact(1))
Factorial
The Normal Version

long fact(n){
    if(n == 0) return 1;
    else return n * fact(n-1);
}

3 * fact(2 * fact(1))
Factorial
The Normal Version

long fact(n){
    if(n == 0) return 1;
    else return n * fact(n-1);
}

3 * fact(2 * fact(1))
Factorial
The Normal Version

```c
long fact(n){
    if(n == 0) return 1;
    else return n * fact(n-1);
}
```

3 * fact(2 * fact(1))
Factorial
The Normal Version

long fact(n) {
    if(n == 0) return 1;
    else return n * fact(n-1);
}

3 * fact(2 * fact(1))
Factorial
The Normal Version

```java
long fact(n){
    if(n == 0) return 1;
    else return n * fact(n - 1);
}
```

3 * fact(2 * fact(1))
Factorial
The Normal Version

long fact(n){
    if(n == 0) return 1;
    else return n * fact(n-1);
}

3 * fact(2 * fact(1))
Factorial
The Normal Version

```
long fact(n){
    if(n == 0) return 1;
    else return n * fact(n-1);
}
```

```
3 * fact(2 * fact(1 * fact(0)))
```

Flowchart:
- Start with `n == 0` condition.
- If `n == 0`, return 1.
- Otherwise, return `n * fact(n-1)`.
Factorial
The Normal Version

```cpp
long fact(n) {
    if (n == 0) return 1;
    else return n * fact(n - 1);
}
```

3 * fact(2 * fact(1 * fact(0)))

Apply fact
Factorial
The Normal Version

```c
long fact(n){
    if(n == 0) return 1;
    else return n * fact(n-1);
}
```

3 * fact(2 * fact(1 * fact(0)))
Factorial
The Normal Version

long fact(n){
    if(n == 0) return 1;
    else return n * fact(n-1);
}

3 * fact(2 * fact(1 * fact(0)))
Factorial
The Normal Version

long fact(n){
    if(n == 0) return 1;
    else return n * fact(n-1);
}

3 * fact(2 * fact(1 * fact(0)))
Factorial
The Normal Version

long fact(n){
    if(n == 0) return 1;
    else return n * fact(n-1);
}

3 * fact(2 * fact(1 * 1))
long fact(n) {
    if (n == 0) return 1;
    else return n * fact(n - 1);
}

3 * fact(2 * fact(1 * 1))
Factorial
The Normal Version

long fact(n){
    if(n == 0) return 1;
    else return n * fact(n-1);
}

3 * fact(2 * 1)
Factorial
The Normal Version

long fact(n){
    if(n == 0) return 1;
    else return n * fact(n-1);
}

3 * fact(2 * 1)
Factorial
The Normal Version

long fact(n){
    if(n == 0) return 1;
    else return n * fact(n-1);
}
Factorial
The Normal Version

long fact(n){
    if(n == 0) return 1;
    else return n * fact(n-1);
}

3 * 2
Factorial
The Normal Version

long fact(n){
    if(n == 0) return 1;
    else return n * fact(n-1);
}

Apply fact
Can you see where is the problem ??
Factorial

The Tail Recursion Version

long fact_1(n, p){
    if(n == 0) return p;
    else return fact_1(n-1, n*p);
}

Call fact_1(3,1)
long fact_1(n, p) {
    if (n == 0) return p;
    else return fact_1(n - 1, n * p);
}

fact_1(3, 1)
Factorial
The Tail Recursion Version

long fact_1(n, p) {
    if(n == 0) return p;
    else return fact_1(n - 1, n*p);
}

apply fact_1

fact_1(3, 1)
long fact_1(n, p){
    if(n == 0) return p;
    else return fact_1(n-1, n*p);
}

fact_1(3,1)
Factorial
The Tail Recursion Version

```c
long fact_1(n, p){
    if(n == 0) return p;
    else return fact_1(n-1, n*p);
}

fact_1(3,1)
```
Factorial
The Tail Recursion Version

long fact_1(n, p) {
    if (n == 0) return p;
    else return fact_1(n-1, n*p);
}

fact_1(3, 1)
Factorial
The Tail Recursion Version

```java
long fact_1(n, p){
    if(n == 0) return p;
    else return fact_1(n-1, n*p);
}
```

Apply fact_1

```
fact_1(3,1)
```
Factorial
The Tail Recursion Version

long fact_1(n, p) {
    if (n == 0) return p;
    else return fact_1(n - 1, n*p);
}

Call fact_1(2,3)
Factorial
The Tail Recursion Version

long fact_1(n, p){
    if(n == 0) return p;
    else return fact_1(n-1, n*p);
}

fact_1(2,3)
Factorial
The Tail Recursion Version

long fact_1(n, p){
    if(n == 0) return p;
    else return fact_1(n-1, n*p);
}

fact_1(2,3)
Factorial
The Tail Recursion Version

long fact_1(n, p) {
    if (n == 0) return p;
    else return fact_1(n-1, n*p);
}

fact_1(2, 3)
Factorial
The Tail Recursion Version

long fact_1(n, p) {
    if (n == 0) return p;
    else return fact_1(n - 1, n * p);
}

fact_1(2, 3)
Factorial
The Tail Recursion Version

long fact_1(n, p){
    if(n == 0) return p;
    else return fact_1(n-1, n*p);
}

Call fact_1 (1,6)
Factorial
The Tail Recursion Version

long fact_1(n, p) {
    if (n == 0) return p;
    else return fact_1(n-1, n*p);
}

fact_1(1, 6)
_factorial

The Tail Recursion Version

long fact_1(n, p){
    if(n == 0) return p;
    else return fact_1(n-1, n*p);
}

CPEG852-Spring14: Topic A - Dataflow - 2
Factorial
The Tail Recursion Version

long fact_1(n, p) {
    if (n == 0) return p;
    else return fact_1(n - 1, n*p);
}

fact_1(1, 6)
Factorial
The Tail Recursion Version

long fact_1(n, p) {
    if (n == 0) return p;
    else return fact_1(n - 1, n * p);
}

fact_1(1, 6)
long fact_1(n, p) {
    if (n == 0) return p;
    else return fact_1(n-1, n*p);
}

fact_1(1,6)
Factorial
The Tail Recursion Version

```
long fact_1(n, p){
    if(n == 0) return p;
    else return fact_1(n-1, n*p);
}
```

```
 fact_1(1,6)
```
Factorial
The Tail Recursion Version

```
long fact_1(n, p){
    if(n == 0) return p;
    else return fact_1(n-1, n*p);
}
```

Call fact(0,6)
long fact_1(n, p) {
    if (n == 0) return p;
    else return fact_1(n - 1, n*p);
}

fact_1(0,6)
Factorial
The Tail Recursion Version

long fact_1(n, p){
    if(n == 0) return p;
    else return fact_1(n-1, n*p);
}

fact_1(0,6)
Factorial
The Tail Recursion Version

long fact_1(n, p){
    if(n == 0) return p;
    else return fact_1(n-1, n*p);
}

fact_1(0,6) returns 6
Factorial
The Tail Recursion Version

long fact_1(n, p) {
    if(n == 0) return p;
    else return fact_1(n-1, n*p);
}

fact_1(1,6) returns 6
Factorial
The Tail Recursion Version

```cpp
long fact_1(n, p){
    if(n == 0) return p;
    else return fact_1(n-1, n*p);
}
```

```
fact_1(3,1) returns 6
```
Factorial
The Tail Recursion Version

long fact_1(n, p){
    if(n == 0) return p;
    else return fact_1(n-1, n*p);
}

Apply fact_1
Recursive Program Graph

Features

• Acyclic.

• One-token-per-link-in-lifetime.

• Tags.

• No deterministic merge needed.

• Recursion is expressed by runtime copying.

• No matching is needed (why?).
Homework:
Implement $y = x^n$ using tail recursion.
A Recursive Version of Power  
(not tail recursive)

Function  Rec-Power (x, n : integer  
  »  returns integer)

Returns
If  n = 0  then 1
else
  x * rec-power (x, n - 1)
endif
endfun

Note: tail-recursive = iterations:  
i.e. the states of the computation are captured  
explicitly by the set of iteration variables.
Dynamic Dataflow

- **Static Dataflow**
  - Only one token per arc
  - Problems with Function calls, nested loops and data structures
  - A signal is needed to allow the parent’s operator to fire

- **Dynamic Dataflow**
  - No limitations on number of tokens per arc
  - Tokens have tags – new firing rules and tag matching
  - The MIT tagged token dataflow model
The Token in Dynamic Dataflow

Token ➔ Tag + Value

\[ [v, <u, s>, d] \]

- \( v \) : Value
- \( u \) : activation instance
- \( s \) : destination actor
- \( d \) : operand slot

Different from Static Dataflow that it needs the tag \(<s,d,v>\)
Dynamic Dataflow

• Loops and function calls
  – Should be executed in parallel as instances of the same graph.

• Arc ➔ a container with different tokens that have different tags.

• A node can fire as soon that all tokens with identical tags are presented in its input arcs.
Tags and Colors

• Concept of colors.

• In a tag <u,s> - which one is the color?

• When a new color is generated?

• When an old color is recovered?
Factorial

The Normal Version – Dynamic Dataflow

long fact(n){
    if(n == 0) return 1;
    else return n * fact(n-1);
}

fact(3)
Factorial

The Normal Version – Dynamic Dataflow

long fact(n){
    if(n == 0) return 1;
    else return n * fact(n-1);
}
Example of Dynamic Dataflow

• Example A
• Example B
The Apply Operator

\[
\begin{align*}
\text{Apply} & \quad a \\
A^{-1} & \quad A \\
\equiv & \quad \text{Dataflow graph of Procedure } q \\
\text{BEGIN} & \\
\text{END}
\end{align*}
\]
The Apply Operator

Apply

BEGIN

END

Dataflow graph of Procedure q

A

A^{-1}

BEGIN

Dataflow graph of Procedure q

END

equiv

equiv

equiv
The Apply Operator

Apply

A

A^{-1}

BEGIN

END

≡

Dataflow graph of Procedure q

BEGIN

END

Dataflow graph of Procedure q

BEGIN

END

CPEG852-Spring14: Topic A - Dataflow - 2
The Apply Operator
The Apply Operator
The Apply Operator

Apply

\( q \)

\( a \)

\( A \)

\( A^{-1} \)

BEGIN

Dataflow graph of Procedure q

END

\( \equiv \)
The Apply Operator

The Apply Operator

The dataflow graph of Procedure q:

\[ \begin{align*}
& \text{BEGIN} \\
& A \\
& A^{-1} \\
& \text{END} \\
\end{align*} \]

The Apply operator: \( aqA \)
Dynamic Dataflow

Advantages

• More Parallelism.
• Handle arbitrary recursion.

Disadvantages

• Implementation of the token tag matching unit.
• Associative Memory would be ideal.
• Not cost effective:
  • Hashing is used
Features of Data Flow Computation Model

- Not history-sensitive.
- Semantics (determinate).
- Parallelism.
Interpreting function invocation as module substitution

(a) Function F

(b) Function G

(c) Function G’
The Apply Actor

(a) Notation for apply

Firing Rule

(b) Firing Rule

F is an (m,n) schema
Concept of Strictness

**Intuitive:**

A mechanism that requires all arguments to be evaluated before evaluation of the body of a function may begin.

**Formal:**

A function \( f \) is strict if

\[
\begin{align*}
\downarrow & = \\
\downarrow & \\
\text{nonstrict} & = \text{not strict}
\end{align*}
\]
Comment on Strict ‘Apply’ Actor

**Advantage:**
- Parallelism.
- Call-by-value semantics.

**Disadvantage:**
- Lose “substitution” property or “referential transparency” property
  - to avoid it, need strictness analysis
Referential Transparency

“...The only thing that matters about an expression is its value, and any sub-expression can be replaced by any other equal in value...”

“...Moreover, the value of an expression is, within certain limits, the same when ever it occurs...”
“Referential Transparency”
or
“Property of Substitution”

Let \( f, g \) be two procedures/functions such that \( f \) appears as an application inside of \( g \); let \( g' \) be the procedure obtained from \( g \) by substituting the text of \( f \) in place of the application;

In the languages which are “referential transparent”, the specification of the function for \( g \) will not depend on any terminating property of \( f \), or \( g = g' \).
Concept of Non-Strictness

- Lenient evaluation model
- Lazy evaluation model
Outline

• Parallel Program Execution Models
• Dataflow Models of Computation
• Dataflow Graphs and Properties
• Three Dataflow Models
  – Static
  – Recursive Program Graph
  – Dynamic
• Dataflow Architectures: Evolution of Multithreading
• **Wait-Match Unit:**
  • Tokens for unary ops go straight through
  • Tokens for binary ops: try to match incoming token and a waiting token with same instruction address and context id
    • Success: Both tokens forwarded
    • Fail: Incoming token → Waiting Token Mem, Bubble (no-op) forwarded
Memory Model and Dataflow

- What is “memory” in dataflow model?

- What is “memory model” under dataflow model?

- Concept of functional programming

- Concept of *single-assignment* and single-assignment programming languages
The I Structures

• Single Assignment Rule and Complex Data structures
  – Consume the entire data structure after each access.

• The Concept of the I Structure
  – Only consume the entry on a write
  – A data repository that obeys the Single Assignment Rule
  – Written only once, read many times

• Elements are associated with status bits and a queue of deferred reads
I-Structure Memory

• Extending dataflow abstract Machine model with I-Structure Memory.

• Use a data structure before it is completely defined.

• Incremental creating or reading of data structures.
The I Structures

• An element of a I-structure becomes defined on a write and it only happens only once.

• At this moment all deferred reads will be satisfied.

• Use a I-structure before it is completely defined.

• Incremental creating or reading of data structures.

• Lenient evaluation model – revisited.
The I Structures: state transitions

- **States**
  - **Present**: The element of an I-structure (e.g. A[i]) can be read but not written
  - **Absent**: The element has been attempted to be read but the element has not been written yet (initial state)
  - **Waiting**: At least one read request has been deferred
The I Structures

- **Elementary**
  - Allocate: reserves space for the new I-Structure
  - I-fetch: Get the value of the new I-structure (deferred)
  - I-store: Writes a value into the specified I structure element

- **Used to create construct nodes:**
  - SELECT
  - ASSIGN