Topic B (Cont’d)
Dataflow Model of Computation

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

- Parallel Program Execution Models
- Dataflow Models of Computation
- Dataflow Graphs and Properties
- Three Dataflow Models
  - Static
  - Recursive Program Graph
  - Dynamic
- Dataflow Architectures and Evolution of Multithreading
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
if (p(y)) {
  f(x, y);
} else {
  g(y);
}
long power(int x, int n) {
    int y = 1;
    for (int i = n; i > 0; --i) {
        y *= x;
    }
    return y;
}
Power Function

\[ Y = X^N \]
Power Function

\[ Y = 2^3 \]
Power Function

\[ Y = 2^3 \]
Power Function

\[ Y = 2^3 \]
Power Function

\[ Y = 2^3 \]
Power Function

\[ Y = 2^3 \]
Power Function

\[ Y = 2^3 \]
Power Function

\[ Y = 2^3 \]
Power Function

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

\[ Y = 2^3 \]
Power Function

\[ Y = 2^3 \]
Power Function

\[ Y = 2^3 \]
Power Function

\[ Y = 2^3 \]
Power Function

\[ Y = 2^3 \]
Power Function

\[
Y = 2^3
\]
Power Function

$Y = 2^3$
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
• 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
MIT Static Dataflow Processor Architecture and Program Activity Templates

\[ Z = (a + b) \times (c - d) \]

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

<table>
<thead>
<tr>
<th>opcode</th>
<th>EC</th>
<th>RC</th>
<th>Destination list</th>
</tr>
</thead>
<tbody>
<tr>
<td>mult2</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

- result arc
- signal arc

ac-bd
ad+bc

\[ \text{sub} 2 \quad 3 \]
\[ \text{plus} 2 \quad 3 \]
for \( i \) in \([1, n]\)
\[
\begin{align*}
H[i] &= a \times X[i] + Y[i] \\
G[i] &= b \times U[i] + V[i] \\
Z[i] &= T[i] \times H[i] + S[i] \times G[i]
\end{align*}
\]
Recursive Program Graphs

- Outlaw iterations:
  Graph must be acyclic
- One-token-per-arc-per-invocation
- Iteration is expressed in terms of a tail recursion
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);
}
```

```
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)
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)
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);
}

Hand Simulate fact(3)
Factorial
The Normal Version

Hand Simulate fact(3)

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

n
1

n==0

F
T

3

-1

Apply fact

*
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

```c
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);
}
```

3 * fact(2)

Diagram:

- Start with `n == 0`.
- If `n == 0`, return 1.
- Otherwise, return `n * fact(n-1)`.
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 * 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

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

3 * fact(2)

n
1

F

T

-1

Apply fact

*
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

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

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

```
Apply fact
```

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);
}
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);
}
```

$$3 \times \text{fact}(2 \times \text{fact}(1 \times \text{fact}(0)))$$
```java
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

```java
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))
Factorial
The Normal Version

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)
The Normal Version

```c
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
Factorial
The Normal Version

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

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

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

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

```
fact_1(3, 1)
```

```
n == 0
F
-F

n-1
T

* 

2

Apply fact_1
```

```
p
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(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)
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);
}

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);
}

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);
}

Call fact(0,6)
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(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);
}

dataflow diagram:

```
Apply fact_1

6

n

p

F

n==0

T

-1

*

fact_1(1,6) 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(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

F T

n p

n==0

-1 *

6

6
Features

- Acyclic
- One-token-per-link-in-lifetime
- Tags
- No deterministic merge needed
- Recursion is expressed by runtime copying
- No matching is needed (why?)
A Quiz: \( y = x^n \) ?
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.
The power function as a recursive program graph

function Rec (x, y, n)
if n = 0 then y
else
    Rec (x, x*y, n-1)
endif
end function
Rec (x, 1, n)
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 $\Rightarrow$ 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 $<u,s>$
Dynamic Dataflow

- Loops and function calls
  Should be executed in parallel as instances of the same graph
- Arc ➔ a container with different token 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 is the color?
- When a new color is generated?
- When an old color is recovered?
The Normal Version – Dynamic Dataflow

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

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

Fact(3)

Apply fact

F

T

-1

3

1

n==0

*
Example of Dynamic Dataflow

- Example A
- Example B
The Apply Operator

\[
\text{Dataflow graph of Procedure } q
\]

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

Apply

Dataflow graph of Procedure q
BEGIN
END
≡
The Apply Operator
The Apply Operator

\[ \text{Apply} \quad a \quad A^{-1} \quad A \quad \text{Dataflow graph of Procedure q} \quad \text{BEGIN} \quad \text{END} \]
The Apply Operator

Dataflow graph of Procedure q

BEGIN

END
The Apply Operator

The dataflow graph of Procedure q:

\[ \begin{align*}
\text{BEGIN} & \rightarrow A & \rightarrow A^{-1} & \rightarrow \text{END} \\
q & \rightarrow a & \rightarrow q \\
\text{Apply} & & & \\
\end{align*} \]
The Apply Operator

\begin{align*}
\text{Apply} & \quad \equiv \\
\text{BEGIN} & \\
\text{Dataflow graph of Procedure } q \\
\text{END} & \\
\end{align*}
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
Function F

Function G

Function G′

Interpreting function invocation as module substitution
The Apply Actor

(a) Notation for apply

(b) Firing Rule
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

\[ f \downarrow = \downarrow \]

nonstrict = not strict
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 subexpression 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 $\rightarrow$ 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
Dataflow
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
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