Topic 4b – Program Execution Models
Dataflow Model of Computation:
A Fine-Grain Kind of PXM

CPEG421/621: Compiler Design

Most slides taken from Prof. Gao’s and J.Manzano’s previous courses, with additional material from J.Landwehr.
Dataflow Model of Computation

\[ a + b + c + d + e \]

1
\[ \blacklozenge \]
3
\[ \blacklozenge \]
4
\[ \blacklozenge \]
3
\[ \blacklozenge \]
\[ + \]
\[ + \]
\[ \times \]
Dataflow Model of Computation

a b c d e

\[
\begin{align*}
&\quad \quad + \\
&\quad \\n&\quad \quad + \\
&\quad 4 \quad 3 \\
&\quad \quad * \\
&\quad 4 \\
&\quad \quad * \\
\end{align*}
\]
Dataflow Model of Computation
Dataflow Model of Computation

a b c d e

\[ a + b + c + d + e = 28 \]
Dataflow Model of Computation

Dataflow Software Pipelining
A Base-Language

~ Data Flow Graphs ~

To serve as an intermediate-level language for high-level languages
To serve as a machine language for parallel machines

- J.B. Dennis
Data Flow Years at MIT
1974 – 1975

• August 1975: 1975 Sagamore Computer Conference on Parallel Processing:
  • Rumbaugh: “Data Flow Languages”
  • Rumbaugh: “A Data Flow Multiprocessor”
  • Dennis: “Packet Communication Architecture”
  • Misunas: “Structure Processing in a Data-Flow Computer”
Early Roots on Dataflow Work at MIT in 70s

- Asynchronous Digital Logic [Muller, Bartky]
- Control Structures for Parallel Programming: [Conway, McIlroy, Dijkstra]
- Abstract Models for Concurrent Systems: [Petri, Holt]
- Theory of Program Schemes [Ianov, Paterson]
- Structured Programming [Dijkstra, Hoare]
- Functional Programming [McCarthy, Landin]
Dataflow Operators

- A small set of dataflow operators can be used to define a general programming language.
Values in dataflow graphs are represented as tokens of the form:

\[<ip, p, v>\]

Where \(ip\) is the instruction pointer, \(p\) is the port, and \(v\) represents the data.

An operator executes when all its input tokens are present; copies of the result token are distributed to the destination operators.
Operational Semantics
(Firing Rule)

- Values represented by tokens
- Placing tokens on the arcs (assignment)
  - snapshot/configuration: state
- Computation

configuration → configuration
Operational Semantics

Firing Rule

- Tokens $\rightarrow$ Data
- Assignment $\rightarrow$ Placing a token in the output arc
- Snapshot / configuration: state
- Computation
  The intermediate step between snapshots / configurations
- An actor of a dataflow graph is enabled if there is a token on each of its input arcs
Any enabled actor may be fired to define the “next state” of the computation.

An actor is fired by removing a token from each of its input arcs and placing tokens on each of its output arcs.

Computation ➔ A Sequence of Snapshots

Many possible sequences as long as firing rules are obeyed.

Determinacy

“Locality of effect”
General Firing Rules

• A switch actor is enabled if a token is available on its control input arc, as well as the corresponding data input arc.

  The firing of a switch actor will remove the input tokens and deliver the input data value as an output token on the corresponding output arc.

• A (unconditional) merge actor is enabled if there is a token available on any of its input arcs.

  An enabled (unconditional) merge actor may be fired and will (non-deterministically) put one of the input tokens on the output arc.
if (p(y)) {
    f(x, y);
} else {
    g(y);
}
A Conditional Schema
A Loop Schema

Initial Loop value

Loop op

COND

T F F

T F
Properties of Well-Behaved Dataflow Schemata

An \((m, n)\) Schema with no enabled actors

(a) Initial Snapshot

An \((m, n)\) Schema with no enabled actors

(a) Final Snapshot
Well-behaved Data Flow Graphs

- Data flow graphs that produce exactly one set of result values at each output arcs for each set of values presented at the input arcs
- Implies the initial configuration is re-established
- Also implies determinacy
Well Behaved Schemas

one-in-one-out & self cleaning

Before

After

Conditional

Loop
Well-formed Dataflow Schema
(Dennis & Fossen 1973)

- An operator is a WFDS
- A conditional schema is a WFDS
- A iterative (loop) schema is a WFDS
- An acyclic composition of component WFDS is a WFDS
Theorem

“A well-formed data flow graph is well-behaved”

proof by induction
Arbitrary connections of data flow operators can result in pathological programs, such as the following:

1. Deadlock
2. Hangup
3. Conflict
4. Unclean
Well-behaved Program

• Always determinate in the sense that a unique set of output values is determined by a set of input values

• References:

  Rodrίquez, J.E. 1966, “A Graph Model of Parallel Computation”, MIT, TR-64]
  Denning, P.J. “On the Determinacy of Schemata” pp 143-147
Remarks on Dataflow Models

• A fundamentally sound and simple parallel model of computation (features very few other parallel models can claim)
• Few dataflow architecture projects survived passing early 1990s. But the ideas and models live on..
• In the new multi-core age: we have many reasons to re-examine and explore the original dataflow models and learn from the past
A Short Story

- **Karp and Miller** analyzed Computation Graphs w/o branches or merges.
- **Dennis** proposes a dataflow language. Pure Dataflow is born.
- **Chamberlain** proposes Single Assignment language for dataflow.
- **Rodriguez** proposes Dataflow Graphs.
- **Kahn** proposes a simple parallel processing language with vertices as queues. Static Dataflow is born.
- **Dennis** designs a dataflow arch.
- **Estrin and Turn** proposed an early dataflow model.
- **Arvind, Nikkel, et al** designed the Monsoon dataflow machine.
- **Arvind and Gostelow, & separately Gurd and Watson** created a tagged token dataflow model. Dynamic Dataflow is born.
Evolution of Multithreaded Execution and Architecture Models

Non-dataflow based

CDC 6600 1964
Flynn’s Processor 1969

HEP B. Smith 1978
Cosmic Cube Seitz 1985

MASA Halstead 1986
J-Machine Dally 1988-93

Alwife Agarwal 1989-96
Tera B. Smith 1990-

Eldorado
CASCADE

The technical contents

Dataflow model inspired

MIT TTDA Arvind 1980
LAU Syre 1976
Manchester Gurd & Watson 1982

Monsoon Papadopoulos & Culler 1988
Iannuci’s 1988-92

P-RISC Nikhil & Arvind 1989
TAM Culler 1990
SIGMA-I Shimada 1988

Cilk Leiserson

*T/Start-NG MIT/Motorola 1991-

CARE Marquez04

Static Dataflow Dennis 1972 MIT

Arg-Fetching Dataflow DennisGao 1987-88
MDFA Gao 1989-93
MTA HumTheobald Gao 94

EARTH PACT95, ISCA96, Theobald99
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

\( Y = X^N \)

\[
x \quad y \quad i
\]

\[
T \quad F
\]

\[
T \quad F
\]

\[
T \quad F
\]

\[
T
\]

\[
1
\]

\[
x
\]

\[
X
\]

\[
Y = x^n
\]

\[
\text{return}
\]

\[
i > 0
\]

\[
f
\]

\[
f
\]

\[
f
\]
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

\[ Y = 2^3 \]

Return value of the function.
Power Function

```
i > 0
*  
2  

T  F

4

T  F

1

T

-1

return

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

\[ Y = 2^3 \]
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
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
for i in [1, n]
    \( H[i] = a \times X[i] + Y[i] \)
    \( G[i] = b \times U[i] + V[i] \)
    \( Z[i] = T[i] \times H[i] + S[i] \times G[i] \)
end 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
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
An Example with Fibonacci
(Thanks to J. Landwehr)

Dataflow is inherently parallel, but if it is recursively called it can lead to deadlocks if the parent is not clean.
An Example with Fibonacci
(Thanks to J.Land)

Let’s begin: we have 1 token inserted for Fib(2)
An Example with Fibonacci
(Thanks to J.Land)

\[
\begin{align*}
T &\quad F \\
2 & \quad -1 \\
<2 & \\
T & \quad F \\
2 & \quad -2 \\
Fib() & \\
Fib() & \quad +
\end{align*}
\]

It splits into two tokens here at a split node
An Example with Fibonacci
(Thanks to J.Landwehr)
An Example with Fibonacci
(Thanks to J. Landwehr)

2 < 2 is false
An Example with Fibonacci
(Thanks to J.Landwehr)
An Example with Fibonacci
(Thanks to J.Landwehr)
An Example with Fibonacci
(Thanks to J.Landwehr)
An Example with Fibonacci
(Thanks to J.Landwehr)
An Example with Fibonacci
(Thanks to J.Landwehr)
An Example with Fibonacci
(Thanks to J.Landwehr)
An Example with Fibonacci
(Thanks to J.Landwehr)

1 is recursed
An Example with Fibonacci (Thanks to J.Landwehr)
An Example with Fibonacci
(Thanks to J.Landwehr)
An Example with Fibonacci (Thanks to J.Landwehr)
An Example with Fibonacci
(Thanks to J.Landwehr)

Split node cannot be applied since there is a False token on 1 of its outputs.
An Example with Fibonacci
(Thanks to J.Landwehr)

That means the program deadlocks
An Example with Fibonacci
(Thanks to J.Landwehr)

1 cannot move forward!
An Example with Fibonacci
(Thanks to J.Landwehr)
Analysis: static dataflow cannot handle arbitrary recursion unless it is tailor recursive
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*
The Normal Version

Hand Simulate fact(3)

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

Apply fact

-1

n==0

n

1

F

T

*
Factorial
The Normal Version

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

Hand Simulate fact(3)

3

1

n==0

F

T

-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)

Apply fact

*  

-1

F  

T  

n==0

3

1

n

3
Factorial
The Normal Version

Hand Simulate fact(3)

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

Apply fact
Factorial
The Normal Version

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

```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

Hand Simulate fact(3)

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

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

Diagram:
- A decision node with the condition `n==0`
- Flow from node `F` to node `T` on a path labeled `-1`
- Flow from node `T` to node `T` on a path labeled `3 * fact(2)`
- Flow from node `F` to node `F` on a path labeled `Apply fact`
Factorial
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 * 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)

Apply fact
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

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

3 * fact(2)

![Diagram of the factorial function with a flowchart and code snippet.](image)
Factorial
The Normal Version

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

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

Apply fact

1

1

n

n==0

F

T

-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))

n
F
n==0
T

Apply fact

1

1

1

*
The Normal Version

Factorial

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

3 * fact(2 * fact(1))

n == 0

Apply fact

* 1

1

-1

0

n

F

T

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)))

Apply fact

-1

n==0

F

T

0

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)))
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

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

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

Apply fact

0

F

1

T

T

T

-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)))
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))

n
1

n==0

F

T

Apply fact

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

n
1

n==0

F

T

Apply fact

* 6

-1

3 * 2

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);
}

n
T
F

n==0
-1

Apply fact

n
1
Can you see where is the problem??
**Factorial**

The Tail Recursion Version

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

Call `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)
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);
}

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)
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

```c
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);
}

Call fact(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)
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

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

6
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?)
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
Dynamic Dataflow

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

• Arc $\rightarrow$ 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
Dynamic Dataflow: Colored Tokens
(Thanks to J.Landwehr)

• On recursion a new color is assigned via a special node called the apply function
• On return the original color is restored
Dynamic Dataflow: Colored Tokens
(Thanks to J.Landwehr)

Let’s begin: we have 1 token inserted for Fib(2)
Dynamic Dataflow: Colored Tokens
(Thanks to J.Landwehr)

It splits into two token here at a split node

\[
\begin{align*}
2 & \quad \text{< 2} \\
T & \quad \text{F} \\
-1 & \quad \text{Fib()} \\
-2 & \quad \text{Fib()} \\
+ & \quad \text{TF} \\
\end{align*}
\]
Dynamic Dataflow: Colored Tokens
(Thanks to J. Landwehr)
Dynamic Dataflow: Colored Tokens (Thanks to J. Landwehr)

2 < 2 is false
Dynamic Dataflow: Colored Tokens
(Thanks to J. Landwehr)
Dynamic Dataflow: Colored Tokens
(Thanks to J. Landwehr)
Dynamic Dataflow: Colored Tokens
(Thanks to J.Landwehr)
Dynamic Dataflow: Colored Tokens
(Thanks to J. Landwehr)
Dynamic Dataflow: Colored Tokens
(Thanks to J.Landwehr)
Dynamic Dataflow: Colored Tokens
(Thanks to J.Landwehr)

Recursively call the Fib graph
Dynamic Dataflow: Colored Tokens
(Thanks to J.Landwehr)

1 is recursed
Dynamic Dataflow: Colored Tokens
(Thanks to J. Landwehr)

New color chosen on recursion
Dynamic Dataflow: Colored Tokens
(Thanks to J. Landwehr)
Dynamic Dataflow: Colored Tokens
(Thanks to J.Landwehr)
Dynamic Dataflow: Colored Tokens (Thanks to J.Landwehr)
Split node can be applied since multiple tokens per arc are allowed.

Dynamic Dataflow: Colored Tokens
(Thanks to J. Landwehr)
Dynamic Dataflow: Colored Tokens (Thanks to J. Landwehr)
Dynamic Dataflow: Colored Tokens (Thanks to J.Landwehr)

Another new color chosen on recursion
Dynamic Dataflow: Colored Tokens
(Thanks to J. Landwehr)
Dynamic Dataflow: Colored Tokens
(Thanks to J.Landwehr)
Dynamic Dataflow: Colored Tokens
(Thanks to J. Landwehr)
Dynamic Dataflow: Colored Tokens
(Thanks to J. Landwehr)
Dynamic Dataflow: Colored Tokens
(Thanks to J. Landwehr)
Dynamic Dataflow: Colored Tokens
(Thanks to J.Landwehr)
Dynamic Dataflow: Colored Tokens
(Thanks to J. Landwehr)
Dynamic Dataflow: Colored Tokens
(Thanks to J. Landwehr)
Dynamic Dataflow: Colored Tokens
(Thanks to J. Landwehr)
Dynamic Dataflow: Colored Tokens
(Thanks to J.Landwehr)
Dynamic Dataflow: Colored Tokens
(Thanks to J.Landwehr)
Dynamic Dataflow: Colored Tokens
(Thanks to J.Landwehr)
Dynamic Dataflow: Colored Tokens
(Thanks to J.Landwehr)
Dynamic Dataflow: Colored Tokens (Thanks to J.Landwehr)
Dynamic Dataflow: Colored Tokens
(Thanks to J. Landwehr)
Dynamic Dataflow: Colored Tokens
(Thanks to J. Landwehr)
Dynamic Dataflow: Colored Tokens
(Thanks to J.Landwehr)
Dynamic Dataflow: Colored Tokens (Thanks to J. Landwehr)
Dynamic Dataflow: Colored Tokens
(Thanks to J.Landwehr)
Dynamic Dataflow: Colored Tokens (Thanks to J. Landwehr)
Dynamic Dataflow: Colored Tokens
(Thanks to J.Landwehr)
Dynamic Dataflow: Colored Tokens
(Thanks to J.Landwehr)
Dynamic Dataflow: Colored Tokens
(Thanks to J.Landwehr)
Dynamic Dataflow: Colored Tokens (Thanks to J.Landwehr)

Analysis: dynamic dataflow can handle arbitrary recursion, but what are its weaknesses? Token matching and color choosing on recursion…
The Normal Version – Dynamic Dataflow

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

**factorial**

```
fn==0
```

```
1
```

```
F
```

```
T
```

```
apply fact
```

```
-1
```

```
fact(3)
```
The Normal Version – Dynamic Dataflow

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

fact(3)
Example of Dynamic Dataflow

- Example A
- Example B
The Apply Operator

Dataflow graph of Procedure q

BEGIN
Dataflow graph of Procedure q

\[ \equiv \]

Apply

\[ \equiv \]

A

\[ A^{-1} \]

\[ q \]

\[ a \]
The Apply Operator

Dataflow graph of Procedure q
BEGIN
END
≡

Apply

A

A⁻¹

BEGIN

Dataflow graph of Procedure q

END

a

q
The Apply Operator

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

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

Apply

Dataflow graph of Procedure q

BEGIN

END

≡
The Apply Operator

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

\[ \text{BEGIN} \]

\[ A \]

\[ A^{-1} \]

\[ \text{END} \]

\[ \equiv \]

\[ \text{Apply} \]

\[ q \rightarrow a \]
The Apply Operator

Apply

Dataflow graph of Procedure q

BEGIN

END

A

A^-1

q

a

≡
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 ➔ no real concept of time!
- Semantics (determinate)
- Parallelism
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
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