CS 476 – Programming Language Design

William Mansky
Input and Output

• What does this code do?

```c
int main()
{
    printf("Hello world");
}
```

\[(\text{printf}(s), \sigma, k) \rightarrow ?\]
I/O as State

• What does this code do?

int main(){
    printf("Hello world");
}

(printf(s), σ, k, o) → (skip, σ, k, o "Hello world")
I/O as State: Output

\[ E := \langle \text{ident} \rangle \mid \langle \# \rangle \mid E + E \mid \ldots \]

\[ C := \text{skip} \mid C; C \mid \ldots \mid \text{print}(E) \]

\[
\begin{align*}
(e, \sigma) \Downarrow v \\
(\text{print}(e), \sigma, k, o) &\rightarrow (\text{skip}, \sigma, k, o \ v)
\end{align*}
\]

- \( o \) builds up a list of the output produced by the program
I/O as State: Typed Output

\[ E := <\text{ident}> \mid <\#> \mid E + E \mid \ldots \]
\[ C := \text{skip} \mid C; C \mid \ldots \mid \text{print\_int}(E) \mid \text{print\_string}(E) \mid \ldots \]

\[ \Gamma \vdash e : \text{int} \]
\[ \Gamma \vdash \text{print\_int}(e) : \text{ok} \]

\[ (e, \sigma) \Downarrow i \]
\[ (\text{print\_int}(e), \sigma, k, o) \rightarrow (\text{skip}, \sigma, k, o, i) \]
I/O as State: Input

\[ E := \langle \text{ident} \rangle \mid \langle \# \rangle \mid E + E \mid \ldots \]

\[ C := \text{skip} \mid C; C \mid \ldots \mid \text{print}(E) \mid x = \text{scan}() \]

\[
(x = \text{scan}(), \sigma, k, v_{i, o}) \rightarrow (\text{skip}, \sigma[x \mapsto v], k, i, o)
\]

- The program consumes the list \( i \) of input
I/O as State: Example

```
print ("Enter two inputs:")
input1 = scan();
input2 = scan();
print("You entered:");
print(input1);
print(input2);
```

\[
i: \quad o:\quad \frac{(e, \sigma) \downarrow v}{(\text{print}(e), \sigma, k, o) \rightarrow (\text{skip}, \sigma, k, o \; v)}
\]
I/O as State: Example

print ("Enter two inputs:"), 𝑖: 𝑜: "Enter"
input1 = scan();
input2 = scan();
print("You entered:");
print(input1);
print(input2);
I/O as State: Example

print ("Enter two inputs:"), i: o:
input1 = scan();
input2 = scan();
print("You entered:");
print(input1);
print(input2);

\[ (x = \text{scan}(), \sigma, k, v \ i, o) \rightarrow \\
(\text{skip}, \sigma[x \mapsto v], k, i, o) \]
I/O as State: Example

\[
i: \text{“a” “b”} \quad o:\]

print ("Enter two inputs:"")
input1 = scan();
input2 = scan();
print("You entered:"");
print(input1);
print(input2);
I/O as State: Example

i: “a” “b”  o: “Enter”

print (“Enter two inputs:”)
input1 = scan();
input2 = scan();
print (“You entered:”);
print(input1);
print(input2);
I/O as State: Example

```
print ("Enter two inputs:")
input1 = scan();
input2 = scan();
print("You entered:");
print(input1);
print(input2);
```

```
i: "a  b"  o: "Enter"
i: "b"  o: "Enter"
```
I/O as State: Example

i: “a” “b” o:

print (“Enter two inputs:”)

input1 = scan();

input2 = scan();

print(“You entered:”);

print(input1);

print(input2);
I/O as State: Example

```
i: “a” “b”    o: “Enter...”
print ("Enter two inputs:"),
input1 = scan();    i: “b”    o: “Enter...”
input2 = scan();    i:    o: “Enter...”
print("You entered:");
print(input1);
print(input2);
```

```
i: “a” “b”    o: “Enter” “You”
i:    o: “Enter” “You” “a”
i:    o: “Enter” “You” “a” “b”
```
I/O as State: Example

\[ i: v_1 \quad v_2 \quad o: \]

print ("Enter two inputs:"") \[ i: v_1 \quad v_2 \quad o: \text{"Enter..."} \]
input1 = scan(); \[ i: v_2 \quad o: \text{"Enter..."} \]
input2 = scan(); \[ i: \quad o: \text{"Enter..."} \]
print("You entered:"); \[ i: \quad o: \text{"Enter" "You"} \]
print(input1); \[ i: \quad o: \text{"Enter" "You" } v_1 \]
print(input2); \[ i: \quad o: \text{"Enter" "You" } v_1 \quad v_2 \]

• Initial state is \((c, \{\}, \text{nil, i, nil})\), where \(i\) is all the input to be read
I/O as State

• State contains list of input to receive and output produced
• Program builds up output list, consumes input list
• Initial state is \((c, \{\}, \text{nil}, i, \text{nil})\), where \(i\) is all the input to be read

• Treats I/O as internal to the program
• What if other programs/threads are also doing I/O?
I/O as State

• What does this code do?

```c
int main(){
    printf(“Hello world”);
}
```

$(\text{printf}(s), \sigma, k, o) \rightarrow (\text{skip}, \sigma, k, o \ "Hello world")$
I/O as Labels

• What does this code do?

```c
int main()
{
    printf("Hello world");
}

out("Hello world")
```

$(\text{printf}(s, \sigma, k)) \rightarrow (\text{skip}, \sigma, k)$
I/O as Labels: Syntax and Semantics

\[ E := \langle \text{id} \rangle \mid \langle \# \rangle \mid E + E \mid \ldots \]

\[ C := \text{skip} \mid C; C \mid \ldots \mid \text{print}(E) \mid x = \text{scan()} \]

\[
\begin{align*}
(e, \sigma) & \Downarrow v \\
\text{out}(v) & \quad \frac{(\text{print}(e), \sigma, k) \xrightarrow{\text{out}(v)} (\text{skip}, \sigma, k)} \\
\text{in}(v) & \quad \frac{(x = \text{scan}(), \sigma, k) \xrightarrow{\text{in}(v)} (\text{skip}, \sigma[x \mapsto v], k)}
\end{align*}
\]
I/O as Labels: Semantics

• The behavior of a program is now a sequence of steps

\[(c_0, \sigma_0, k_0) \xrightarrow{a_1} (c_1, \sigma_1, k_1) \xrightarrow{a_2} \cdots \xrightarrow{a_n} (c_n, \sigma_n, k_n)\]

• The sequence \(a_1, a_2, \ldots, a_n\) is the external behavior of the program (sometimes called the trace)

• Divides I/O semantics into two separate questions:
  — What I/O events happen in a program?
  — What do those events mean?
I/O as Labels: Example

\[ \sigma_0 \]

print ("Enter two inputs:"");
input1 = scan();
input2 = scan();
print("You entered:");
print(input1);
print(input2);
I/O as Labels: Example

print ("Enter two inputs:")
input1 = scan();
input2 = scan();
print("You entered:");
print(input1);
print(input2);
I/O as Labels: Example

print ("Enter two inputs:"")
input1 = scan();
input2 = scan();
print("You entered:");
print(input1);
print(input2);

• Trace: out("Enter") in(v₁) in(v₂) out("You") out(v₁) out(v₂)
I/O as Labels: Syntax and Semantics

\[ E := \langle \text{ident} \rangle \mid \langle \# \rangle \mid E + E \mid \ldots \]
\[ C := \text{skip} \mid C ; C \mid \ldots \mid \text{print}(E) \mid x = \text{scan}() \]

\[
\begin{array}{c}
\frac{(e, \sigma) \Downarrow v \quad \text{out}(v)}{(\text{print}(e), \sigma, k) \xrightarrow{\text{out}(v)} (\text{skip}, \sigma, k)}
\frac{(x = \text{scan}(), \sigma, k) \xrightarrow{\text{in}(v)} (\text{skip}, \sigma[x \mapsto v], k)}
\end{array}
\]
I/O as Labels: From Labels to State

\[
\begin{align*}
(x = \text{scan}(), \sigma, k) & \xrightarrow{\text{in}(v)} (\text{skip}, \sigma[x \mapsto v], k) \\
(c, \sigma, k) & \xrightarrow{\text{in}(v)} (c', \sigma', k') \\
(c, \sigma, k, v i, o) & \rightarrow (c', \sigma', k', i, o) \\
(c, \sigma, k) & \xrightarrow{\text{out}(v)} (c', \sigma', k') \\
(c, \sigma, k, i, o) & \rightarrow (c', \sigma', k', i, o \, v)
\end{align*}
\]
I/O as Labels: Interaction

Program:

\[ \sigma_0 \]
\[ \text{out}(v_1) \rightarrow \sigma_1 \]
\[ \text{in}(v_2) \rightarrow \sigma_2 \]

User:

sees \( v_1 \)

inputs \( v_2 \)
I/O as Labels: Interaction

Program:

\[
\begin{align*}
\sigma_0 & \xrightarrow{\text{out}(v_1)} \sigma_1 \\
\text{in}(v_2) & \xrightarrow{} \sigma_2
\end{align*}
\]

User:

\[
\begin{align*}
\sigma_0' & \xrightarrow{\text{in}(v_1)} \sigma_1' \\
\text{out}(v_2) & \xrightarrow{} \sigma_2'
\end{align*}
\]

\[
\begin{align*}
(c_1, \sigma_1) & \xrightarrow{\text{in}(v)} (c_1', \sigma_1') \\
(c_2, \sigma_2) & \xrightarrow{\text{out}(v)} (c_2', \sigma_2')
\end{align*}
\]

\[
(c_1, \sigma_1) \parallel (c_2, \sigma_2) \rightarrow (c_1', \sigma_1') \parallel (c_2', \sigma_2')
\]
I/O as Labels: Interaction

• Each component produces a trace of I/O behavior
• When running in parallel, one component’s input can be another’s output
• The two components synchronize, stepping simultaneously and communicating a value at the same time
• This is synchronous message-passing concurrency, as opposed to the actor model’s asynchronous message-passing
  — Still no shared state between, e.g., program and user
• Examples: console I/O, network sockets, Go, CSP, pi-calculus
Interaction and Concurrency

\[ E ::= \langle \text{ident} \rangle \mid \langle \# \rangle \mid E + E \mid \ldots \]

\[ C ::= \text{skip} \mid C; C \mid \ldots \mid \text{print}(E) \mid x = \text{scan()} \]

\[
\begin{array}{c}
(e, \sigma) \Downarrow v \\
\hline
\text{out}(v)
\end{array}
\]

\[
\begin{array}{c}
\text{(print}(e), \sigma, k) \xrightarrow{\text{out}(v)} (\text{skip}, \sigma, k)
\end{array}
\]

\[
\begin{array}{c}
(x = \text{scan}(), \sigma, k) \xrightarrow{\text{in}(v)} (\text{skip}, \sigma[x \mapsto v], k)
\end{array}
\]
Interaction and Concurrency

\[ E := \langle \text{ident} \rangle | \langle \# \rangle | E + E | \ldots \]
\[ C := \text{skip} | C; C | \ldots | \text{print}(E) | x = \text{scan()} \]

\[
\begin{array}{c}
(e, \sigma) \downarrow v \\
\hline
\text{console!}v
\end{array}
\]

\[
\begin{array}{c}
\text{console!}v \quad (\text{print}(e), \sigma, k) \xrightarrow{} (\text{skip}, \sigma, k)
\end{array}
\]

\[
\begin{array}{c}
\text{console?}v \quad (x = \text{scan()}, \sigma, k) \xrightarrow{} (\text{skip}, \sigma[x \mapsto v], k)
\end{array}
\]

• A label is \( a! v \) or \( a? v \), where \( a \) is the name of a channel
Interaction and Concurrency

\[ E := \langle \text{ident}\rangle \mid \langle \#\rangle \mid E + E \mid \ldots \]

\[ C := \text{skip} \mid C; C \mid \ldots \mid \langle \text{ident}\rangle \triangleright E \mid \langle \text{ident}\rangle \leftarrow \langle \text{ident}\rangle \]

\[
\frac{(e, \sigma) \downarrow v}{(a! e, \sigma, k) \rightarrow (\text{skip}, \sigma, k)}
\]

\[
\frac{(a ? x, \sigma, k) \rightarrow (\text{skip}, \sigma[x \mapsto v], k)}{(a?v)}
\]

• A label is \( a! v \) or \( a? v \), where \( a \) is the name of a channel
Interaction and Concurrency

\[ E ::= \langle \text{ident} \rangle \mid \langle \# \rangle \mid E + E \mid \ldots \]

\[ C ::= \text{skip} \mid C; C \mid \ldots \mid \langle \text{ident} \rangle ! E \mid \langle \text{ident} \rangle ? \langle \text{ident} \rangle \mid C \mid | C \]

\[
\begin{align*}
(e, \sigma) & \Downarrow v \\
(a ! e, \sigma, k) & \rightarrow (\text{skip}, \sigma, k)
\end{align*}
\]

\[
\begin{align*}
(a ? x, \sigma, k) & \rightarrow (\text{skip}, \sigma[x \mapsto v], k)
\end{align*}
\]

• A label is \( a! v \) or \( a? v \), where \( a \) is the name of a channel
Interaction and Concurrency

\[
\begin{align*}
(c_1, \sigma_1, k_1) & \xrightarrow{a!v} (c_1', \sigma_1', k_1') & & (c_2, \sigma_2, k_2) & \xrightarrow{a?v} (c_2', \sigma_2', k_2') \\
(c_1, \sigma_1, k_1) \parallel (c_2, \sigma_2, k_2) & \rightarrow (c_1', \sigma_1', k_1') \parallel (c_2', \sigma_2', k_2')
\end{align*}
\]

\[
\begin{align*}
(c_1, \sigma_1, k_1) & \xrightarrow{b} (c_1', \sigma_1', k_1') & & (c_2, \sigma_2, k_2) & \xrightarrow{b} (c_2', \sigma_2', k_2')
\end{align*}
\]

\[
\begin{align*}
(c_1, \sigma_1, k_1) \parallel (c_2, \sigma_2, k_2) & \rightarrow (c_1', \sigma_1', k_1') \parallel (c_2, \sigma_2, k_2) \\
(c_2, \sigma_2, k_2) & \xrightarrow{b} (c_2', \sigma_2', k_2')
\end{align*}
\]

\[
\begin{align*}
(c_1, \sigma_1, k_1) \parallel (c_2, \sigma_2, k_2) & \rightarrow (c_1, \sigma_1, k_1) \parallel (c_2', \sigma_2', k_2')
\end{align*}
\]
Internal and External Communication

```python
print(“Name:”);
name = scan();
print(“Message:”);
msg = scan();
message = name + “ says ” + msg;
network_send(message);
response = network_recv();
```
Internal and External Communication

console!“Name:”;  
console?name;  
console!“Message:”;  
console?msg;  
message = name + “ says ” + msg;  
network!message;  
network?response;

• c!“Name:” c?name c!“Message:” c?msg n!(name + “ says ” + msg) n?resp
console!“Name:”;
console?name;

console!“Message:”;
console?msg;
message = name + “ says ” + msg;
network!message;

network?response;

while(true){
    console?x;
    if(x = “Name:”)
        console!“Alice”;
    else if(x = “Message:”)
        console!“hi”;
}

Internal and External Communication
Internal and External Communication

```javascript
console!"Name:";
console?name;
console!"Message:";
console?msg;
message = name + " says " + msg;
network!message;
network?response;

while(true){
  console?x;
  if(x = "Name:")
    console!"Alice";
  else if(x = "Message:")
    console!"hi";
}
```
Internal and External Communication

```javascript
console!“Name:”;  while(true){
console?name;
console!“Message:”;  console?x;
console?msg;  if(x = “Name:”){
message = name + “ says ” + msg;  console!“Alice”;
console?msg;  }else if(x = “Message:”){
network!message;  console!“hi”;  }
network?response;
}
```
Internal and External Communication

```javascript
while (true) {
  console.log("Name:");
  name = readline();
  console.log("Message:");
  msg = readline();
  message = name + " says " + msg;
  network!message;
  network?response;
  if (x = "Name:".)
    console.log("Alice");
  else if (x = "Message:".)
    console.log("hi");
}
```
Internal and External Communication

```javascript
while(true){
  console?x;
  if(x = "Name:")
    console!"Alice";
  else if(x = "Message:")
    console!"hi";
  }
```
Internal and External Communication

```javascript
while(true){
  console?x;
  if(x = "Name:"
    console!"Alice";
  else if(x = “Message:”)
    console!“hi”;
  }
```
Internal and External Communication

console!“Name:”;                           | network?x;
console?name;                               | network!“Acknowledged.”
console!“Message:”;                          | network?response;
console?msg;                                |                                           
message = name + “ says ” + msg;             |                                           
network!message;                             |                                           
network?response;                            |                                           
• console!“Name:” console?name  console!“Message:” console?msg
Internal and External Communication

• Console I/O, network communication, message-passing, etc. can be considered to be either *internal* or *external* to a system

• Internal communication has known values; external communication is visible to and affected by the outside world

• Composing processes that communicate on the same channel turns it from external into internal

• We can draw the lines differently for different purposes

• All of this is made possible by treating I/O as labels!
Semantics of I/O: Summary

• If we’re only interested in the I/O behavior of one single-threaded program, we can represent I/O as internal state.
• To model concurrency or interaction, we can represent I/O as labels on small-steps.
• Labels have no meaning in and of themselves, but can be composed with a way of *interpreting* the labels.
• To describe the meaning of a program with I/O, we have to consider not just the program but its external environment!