Using Concurrency for Structuring

Jayadev Misra

Department of Computer Science
University of Texas at Austin

http://orc.csres.utexas.edu
Why concurrency?

- To speed up things
- To model an inherently concurrent system
- To structure a system (e.g. operating systems)
Quick Intro to Orc; Parallel Composition

1

:: 1 — publishes 1

1 | 2

:: 1 — publishes both 1

:: 2 — and 2
Quick Intro to Orc; Sequential Composition

1 \( >x> x + 3 \)

:: 4

(1 \| 2) \( >x> x \)

:: 1

:: 2

(1 \| 2) \( \gg 3 \)

:: 3

:: 3
Quick Intro to Orc; Pruning

\[ x + 1 < x < 1 \]

:: 1

\[ x < x < (1 | 2) \]

:: 2

val \ x = (1 | 2)
Example: Fibonacci numbers

\[ \text{def } H(0) = (1, 1) \]

\[ \text{def } H(n) = \]
\[ \text{val } (x, y) = H(n - 1) \]
\[ (y, x + y) \]

\[ \text{def } Fib(n) = H(n) > (x, _)> x \]

– Goal expression

\[ Fib(5) \]
Quick Intro to Orc; Otherwise Combinator

1 ; 2
:: 1

stop ; 2
:: 2

1 \gg stop ; 2
:: 2
• An Orc program calls sites to carry out some of its work.
• Fundamental Site \( \text{if}(b) \), where \( b \) is boolean:
  publish signal if \( b \) is true, silent otherwise.
• \( \text{if}(\text{false}) = \text{stop} \)
Subset Sum

Given is a list of positive integers $xs$ and an integer $n$.

Enumerate all sublists of $xs$ that add up to $n$. 
Enumerate All Solutions to Subset Sum

\[
\text{def } \text{sums}(0, \_ ) = [] \quad \text{— } n = 0
\]

\[
\text{def } \text{sums}(\_, [\]) = \text{stop} \quad \text{— } n \neq 0 \text{ and } xs = []
\]

\[
\text{def } \text{sums}(n, x : xs) = \quad \text{— } n \neq 0 \text{ and } xs \neq []
\]
\[
\text{if } (n > 0) \Rightarrow
\]
\[
(\text{sums}(n - x, xs) > ys > x : ys \mid \text{sums}(n, xs))
\]
Completing the Program

```python
def enum(n, xs) = sums(n, xs) >ys> Some(ys) ; None()

enum(10, [2, 4, 1, 2, 3])

:: Some([2, 4, 1, 3])
:: Some([4, 1, 2, 3])```

Enumerate at most one solution

```python
def sums(0, _) = [] — n = 0

def sums(_, [[]]) = stop — n ≠ 0 and xs = []

def sums(n, x : xs) = — n ≠ 0 and xs ≠ []
    if(n > 0) ≫
        (sums(n - x, xs) ≫ ys ≫ x : ys | sums(n, xs))

def one(n, xs) = (Some(ys) ≪ ys ≪ sums(n, xs)) ; None()

one(10, [2, 4, 1, 2, 3])

:: Some([2, 4, 1, 3])
```
The first lexicographic solution

def \textit{sum}(0, \_ ) = [ \] — \( n = 0 \)

def \textit{sum}(\_, [\]) = \textit{stop} — \( n \neq 0 \) and \( x s = [\] \)

def \textit{sum}(n, x : x s ) =
\quad \text{if} (n > 0) \implies
\quad (x : \textit{sum}(n - x, x s ) ; \textit{sum}(n, x s ))

def \textit{first}(n, x s ) = \textit{Some}(\textit{sum}(n, x s )) ; \textit{None}()

\textit{first}(15, [2, 4, 1, 2, 3])

:: \textit{None}()
Consider the grammar:

\[
\begin{align*}
expr &::= term \mid term + expr \\
term &::= factor \mid factor * term \\
factor &::= literal \mid (expr) \\
literal &::= 3 \mid 5
\end{align*}
\]
Parsing strategy

For each non-terminal, say $expr$, define $expr(xs)$: publish all suffixes of $xs$ such that the prefix is a $term$.

$$
def isexpr(xs) = expr(xs) >[]> true \ ; false$$

To avoid multiple publications (in ambiguous grammars),

$$
def isexpr(xs) =
  val res = expr(xs) >[]> true \ ; false
  res$$

\[
\text{isexpr} \smallskip
\left("(\",\",\"3\",\" \ast \",\"3\",\")\",\"\)\",\"\)\",\"\),\"\+\",\"(\",\"3\",\" \+ \",\"3\",\")\")\]]
\quad\text{—}\quad((3\ast3)+(3+3))
\]

$$::\quad true$$
Function for each non-terminal

Given: \[ expr ::= \text{term} \mid \text{term + expr} \]

Rewrite: \[ expr ::= \text{term (\(\epsilon\) \mid + expr)} \]

```python
def expr(xs) = term(xs) >ys> (ys \mid ys >"+" : zs > expr(zs))
def term(xs) = factor(xs) >ys> (ys \mid ys >"*" : zs > term(zs))
def factor(xs) = literal(xs) 
                | xs >"(" : ys > expr(ys) >")" : zs > zs

def literal(n : xs) = n >"3" > xs \mid n >"5" > xs
def literal([]) = stop
```
Exception Handling; callback

- A client requests a service from a server.
- Typically, the server fulfills the request.
- Sometimes, server requests authentication.
def request() =
    val exc = Buffer() — returns a buffer site

    server.req(exc) >v> Some(v)
    | exc.get() >r> exc.put(auth(r)) >> stop