How to say a lot with few words

May 12, 2006

IRCAM

Peter Van Roy
Université catholique de Louvain
Louvain-la-Neuve, Belgium

Goal of the talk

- The goal of this talk is to show that programming languages can be both concise and powerful
- We will show how adding a few powerful concepts can greatly increase the expressiveness of a programming language
- At the same time, we will give a comprehensive overview of concurrent programming and how simple it can be if done properly
- The language we will use is called Oz

Prerequisites

- I assume some familiarity with programming
  - Preferably, in at least two languages
  - Familiarity with algorithmic thinking
- I am going to cover a lot of ground quickly
  - I hope some concepts will be new to you
  - Some may be familiar concepts in a new jacket
  - I will use simple examples to make everything intuitive
- For many more examples and techniques, see the book "Concepts, Techniques, and Models of Computer Programming", MIT Press, March 2004
  - All the example programs run on the Mozart system, available at http://www.mozart-oz.org

Programming language power

- How do programming languages get their expressive power?
- There are two main ways:
  - By libraries: with a large number of libraries that provide extra functionality
  - By design: with a small number of concepts that can be combined in many ways
- The library approach soon hits a brick wall
  - It is limited by the underlying language, e.g., Java always uses objects with mutable state
- The concept approach can go much further
  - We have used this approach since the early 1990s to design the Oz language
  - This talk is a practical introduction to the approach

Choosing the right concepts

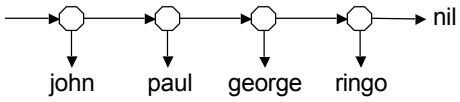
- Oz provides a large set of basic concepts
- Choose the concepts you need, for the paradigm you need
  - Functional programming
  - Declarative concurrency
  - Lazy functional programming
  - Message-passing concurrency
  - Asynchronous dataflow programming
  - Relational programming
  - Constraint programming
  - Object-oriented programming
  - …
- All these paradigms work together well because they differ in just a few concepts

Symbolic data structures

Symbolic data structures

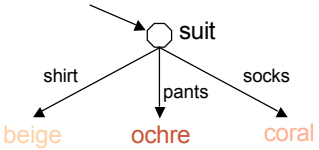
- Lists: the simplest linear structure
  [france belgium colombia]
- Records: a way to group data together
  nations(france:paris belgium:brussels colombia:bogota)
- Atoms: simple constants
  nations, france, paris, belgium, brussels, colombia, bogota
- Numbers: integers (true integers) or floating point
- All these data structures are first-class values
  - First class: Full range of operations to calculate with them
  - Values: They are constants (this is very important!)

Lists



- Lists: the simplest linear structure
  - A list is either an empty list or an element followed by a list
    L=nil % Empty list
    L=john|nil % Element followed by a list
    L=john|(paul|nil)
    L=john|(paul|(george|(ringo|nil)))
  - Lists are used so often that we give them special syntax
    L=john|paul|george|ringo|nil
    L=[john paul george ringo]
- List operations
  - First element: L.1 (sometimes known as “head” or “car”)
  - Rest of list: L.2 (sometimes known as “tail” or “cdr”)

Records



- Records: a way of grouping data together
  R=suit(shirt:beige pants:ochre socks:coral)
- Records have a label (“suit”) and a set of field names (“shirt”, “pants”, “socks”) and their values (“beige”, “ochre”, “coral”)
- Calculations with records

```
{Browse R.shirt}        % Displays beige
{Browse {Label R}}      % Displays suit
{Browse {Arity R}}      % Displays [pants shirt socks]
{Browse {Width R}}      % Displays 3 (number of fields)
R2={AdjoinAt R shirt mauve} % Record with new field
```

- Browse is a tool for displaying data structures

Functional programming

Functional programming

- We use a simple functional language as the starting point
  - (Actually it is a process calculus with procedures, but you don't really need to know that yet)
- This is a powerful way to begin a programming language
- Functions are building blocks
  - This is called higher-order functional programming and it gives an enormous expressive power (the whole area of functional programming is based on this)
  - A function is a value in the language (like an integer), sometimes called a "lexically scoped closure"

Examples of functions

- Here is a simple factorial function

```
fun {Fact N}
   if N==0 then 1 else N*{Fact N-1} end
end
```

- We can use it to define combinations

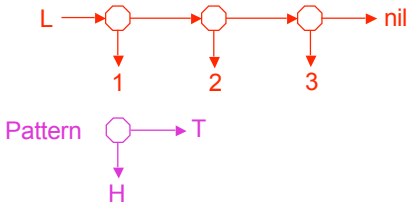
```
fun {Comb N K}
   {Fact N} div ({Fact K} * {Fact N-K})
end
```

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

- This follows exactly the mathematical definition of combinations
- Because Oz integers are true integers (arbitrary precision), this definition really works!
  - For example, {Comb 52 5} returns 2598960 (number of poker hands)
  - This does not work in C++ or Java since Comb will overflow (they only have integers modulo 2^{32})
  - This shows why a language should have a simple semantics

Pattern matching

```
fun {SumList L}
   case L
   of nil then 0
   [] HIT then H+{SumList T}
   end
end
```



- Pattern matching takes apart a data structure by matching it against a corresponding shape
- {Sum [1 2 3]} will try to match [1 2 3] first against nil and then against H|T
- Matching [1 2 3] against nil fails (no way to make them equal)
- Matching [1 2 3] against H|T succeeds and gives H=1 and T=2|3|nil
  - Remember that [1 2 3]=1|2|3|nil
  - H+{Sum T} becomes 1+{Sum [2 3]}
- Final result is 1+(2+(3+0))=6

Higher-order programming in one slide

- Higher-order programming uses functions of any order!
  - A function whose arguments and results are not functions is of first order
    - **fun** {$ X Y} X+Y **end** is a first-order function (note: this function has no name!)
  - A function that has a function of order *n* in an argument or result is of order *n+1*
- A function that returns a function that adds N to a number:

```
fun {MakeAdder N}
   fun {$ X} N+X end
end
Add1={MakeAdder 1}
```

  (Note that the Add1 function has memorized the value of N, which is 1)
- A function that takes a function F of two arguments and an argument N, and returns a function of one argument X that does F on N and X

```
fun {MakeOneArg F N}
   fun {$ X} {F N X} end
end
Add1={MakeOneArg fun {$ X Y} X+Y end 1}
```

  (This Add1 function has memorized the values of F and N)
- What is the order of MakeAdder and the order of MakeOneArg?

Generic programming in one slide

- Summing the elements of a list:

```
fun {SumList L}
   case L of nil then 0
   [] H|T then H+{SumList T} end
end
```

- We make this generic by replacing 0 and +:

```
fun {FoldR L F U}
   case L of nil then U
   [] H|T then {F H {FoldR T F U}} end
end
```

  Why FoldR? It associates to the right: $\{F\ X_0\ \{F\ X_1\ \{F\ X_2\ \ldots\ \{F\ X_{n-1}\ U\}\ldots\}\}\}$

- Now we can define many variations:

```
fun {SumList L} {FoldR L fun {$ X Y} X+Y end 0} end
fun {ProdList L} {FoldR L fun {$ X Y} X*Y end 1} end
fun {Some L} {FoldR L fun {$ X Y} X orelse Y end false} end
fun {All L} {FoldR L fun {$ X Y} X andthen Y end true} end
```


Dataflow and concurrency

Dataflow variables

- Single-assignment store
  - Variables are initially unbound and can be bound to just one value

```
declare X in
{Browse X}        % Displays "X"
X=100             % X is bound, display becomes "100"
```

  - Data structures with holes that are filled in later ("partial values")

```
declare X K V L R in
X=tree(K V L R)   % Build tree with holes in it
K=dog V=chien     % Fill key and value
L=tree(cat chat leaf leaf)          % Fill left subtree
R=tree(mouse souris leaf leaf)      % Fill right subtree
```

- This is an important concept for many paradigms
  - Functional programs can be simpler and more efficient (tail recursion)
  - Declarative concurrency becomes possible (streams)

Concurrency

- Concurrency is a language concept that allows to express when two computations are independent
  - This is very important and should be taught early
- Concurrency should be easy to use
  - It's hard in the usual object-oriented languages
- We will see just how easy concurrency can be
  - Let us add just one concept: the thread
  - Declarative concurrency
  - Message-passing concurrency
  - Asynchronous dataflow programming

Dataflow computation

- A calculation proceeds when its inputs become available

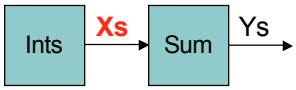
```
thread Z=X+Y {Browse Z} end
thread {Delay 1000} X=25 end
thread {Delay 2000} Y=144 end
```

- When this is executed, nothing is displayed right away
- After 1000 milliseconds, X is bound
  - Still nothing is displayed!
- After 2000 milliseconds, Y is bound
  - X+Y can proceed, and the Browse then displays 169

Dataflow with streams

- Eager producer/consumer example with dataflow synchronization

```
fun {Ints N Max}
   if N<Max then
      {Delay 1000}
      N|{Ints N+1 Max}
   else nil end
end
```




```
fun {Sum S Xs}
   case Xs of X|Xr then
      S|{Sum S+X Xr}
   [] nil then nil end
end
```

```
local Xs Ys in
   thread Xs={Ints 1 1000} end
   thread Ys={Sum 0 Xs} end
end
```

- Ints and Sum threads share the dataflow variable **Xs**, which is a list with unbound tail (stream)
- Monotonic dataflow behavior of **case** statement (synchronize on data availability) gives **stream communication**
- No race conditions

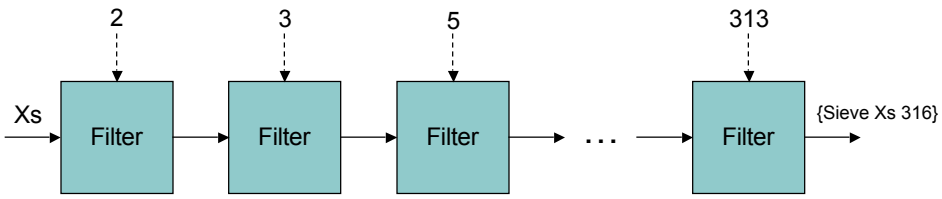
Concurrency can be cheap

- You might wonder whether this is practical
  - Aren't threads expensive?
  - They are expensive in some languages (e.g., Java), but that is an artifact of their implementation
- Threads are cheap in Oz; you can use them whenever you need them

```
fun {Fibo N}
   if N=<2 then 1 else
      thread {Fibo N-1} end + {Fibo N-2}
   end
end
```

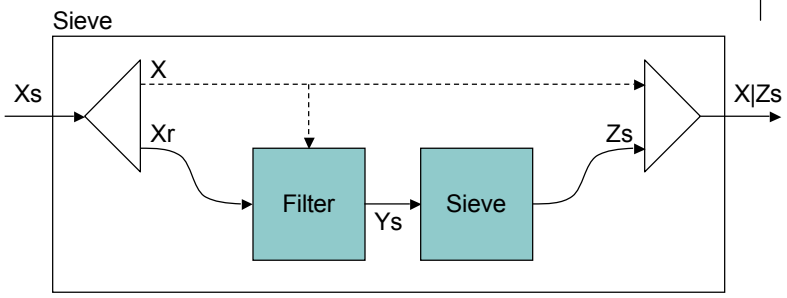
- {Fibo N} creates an exponential number of threads without changing the result of the calculation

Sieve of Eratosthenes (1)



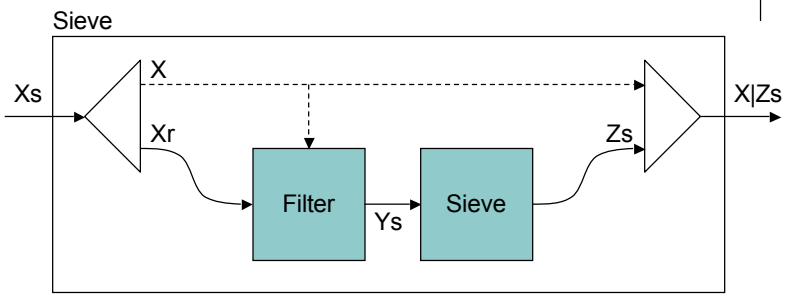
- Let us build a pipeline that implements a prime-number sieve
- At one end, we introduce a sequence of integers starting from 2
- Each pipe element removes multiples of some number
- Only primes will come out the other end

Sieve of Eratosthenes (2)



- Take input stream Xs, decompose into first element X and rest of stream Xr
- Create a filter element with input stream Xr that removes multiples of X
- Call Sieve recursively with output Ys of filter
- Combine X with output Zs of inner Sieve, to make output of outer Sieve

Sieve of Eratosthenes (3)




```
fun {Sieve Xs}
   case Xs of nil then nil
   [] X|Xr then Ys in
      thread Ys={Filter Xr fun {$ Y} Y mod X \= 0 end} end
      X|{Sieve Ys}
   end
end
```

function to check multiple

Sieve of Eratosthenes (4)

```
fun {Sieve Xs}
   case Xs of nil then nil
   [] X|Xr then
      X|{Sieve thread {Filter Xr fun {$ Y} Y mod X \= 0 end} end}
   end
end
```

- We can make the definition shorter by nesting the call to Filter
- We don't really need to declare Ys explicitly

Sieve of Eratosthenes (5)

```
fun {Sieve Xs M}
   case Xs of nil then nil
   [] X|Xr then
      if X=<M then Ys in
         thread Ys={Filter Xr fun {$ Y} Y mod X \= 0 end} end
         X|{Sieve Ys M}
      else
         Xs
      end
   end
end
```

- Generating primes up to n only requires √n filter elements
- This version of Sieve does this optimization
- Most of the work is done in the early filters!

Lazy evaluation

Lazy functional programming

- Lazy evaluation is another natural way to evaluate a functional program
  - Do a calculation only if we need the result
  - Control flows from the output to the input (!)
- Lazy evaluation can be added easily to declarative concurrency: just add one concept “wait until needed”
  - {WaitNeeded X} : wait until X is needed by another calculation
- We can sprinkle calls to WaitNeeded in a program to make it lazy
  - The sprinkling will not change the results of the program. It will only change how much computation is done and when. A very nice way to make a program incremental!

Lazy functions

- A lazy function is executed only when its result is needed

```
fun lazy {Fact N}
   if N==0 then 1 else N*{Fact N-1} end
end
F={Fact 100}      % F is not needed yet
Y=F+1             % F is needed
```

- Lazy functions can be implemented with threads and WaitNeeded

```
proc {Fact N F}
   thread
      {WaitNeeded F}
      F=(if N==0 then 1 else N*{Fact N-1} end)
   end
end
```

- Note that function syntax is short-hand for a procedure with one more argument that is bound to the output ("**fun** {Fact N}" is short-hand for "**proc** {Fact N F}")

Lazy producer/consumer

- With lazy functions we can calculate with infinite data structures

```
fun lazy {Ints N}
   N|{Ints N+1}
end
```

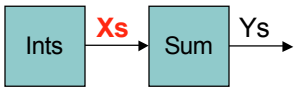
- Lazy list of factorials: each factorial is only calculated once!

```
fun lazy {Facts F N}
   F|{Facts F*N N+1}
end
FactList={Facts 1 2}
{Browse {Nth FactList 69}} % Get 69th element
{Browse {Nth FactList 52}} % Get 52nd element (no extra work!)
```

Lazy producer/consumer

- Lazy producer/consumer example with dataflow synchronization

```
fun lazy {Ints N}
   {Delay 1000}
   N|{Ints N+1}
end
```




```
fun lazy {Sum S Xs}
  case Xs of X|Xr then
     S|{Sum S+X Xr}
  [] nil then nil end
end
```

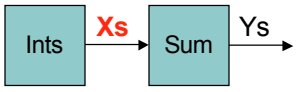
```
local Xs Ys in
  thread Xs={Ints 1} end
  thread Ys={Sum 0 Xs} end
  {Browse {Nth 1000 Ys}}
end
```

- Difference with eager version: it is the *final consumer* that decides how much to calculate, not the initial producer
- Stream communication with shared dataflow variable **Xs**, just like before
- No race conditions, just like before

Eager producer/consumer

- Eager producer/consumer example with dataflow synchronization

```
fun {Ints N Max}
  if N<Max then
    {Delay 1000}
    N|{Ints N+1 Max}
  else nil end
end
```

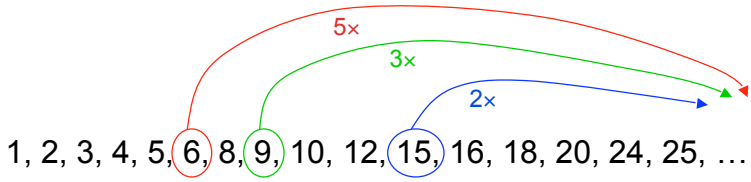



```
fun {Sum S Xs}
  case Xs of X|Xr then
     S|{Sum S+X Xr}
  [] nil then nil end
end
```

```
local Xs Ys in
  thread Xs={Ints 1 1000} end
  thread Ys={Sum 0 Xs} end
end
```

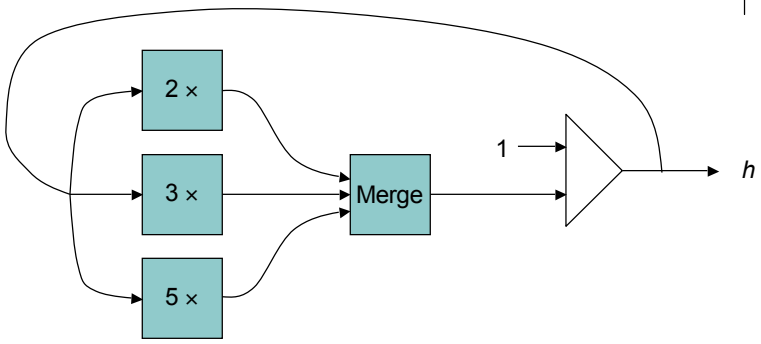
- Ints and Sum threads share the dataflow variable **Xs**, which is a list with unbound tail (stream)
- Monotonic dataflow behavior of **case** statement (synchronize on data availability) gives **stream communication**
- No race conditions

Hamming problem



- The problem is to generate all integers of the form $2^a3^b5^c$ in increasing order
- Here is one way to generate the stream
  - Assume we know a finite part, *h*, of the stream
  - Take the smallest *x* of *h* such that *2x* is bigger than all of *h*
  - Do the same for 3 and 5, giving *y* and *z*
  - Then the next element of h is min*(2x,3y,5z)*

Hamming problem



- We can program this with streams
- We take the stream h, multiply it by 2, 3, and 5, and merge the results into a single output
- The calculation has to be lazy, otherwise it goes into an infinite loop

Hamming problem

```
fun lazy {Times N H}
   case H of X|H2 then
      N*X|{Times N H2}
   end
end

fun lazy {Merge Xs Ys}
   case Xs#Ys of (X|Xr)#(Y|Yr) then
      if X<Y then X|{Merge Xr Ys}
      elseif X>Y then Y|{Merge Xs Yr}
      else X|{Merge Xr Yr} end
   end
end
```

```
H=1|{Merge {Times 2 H}
           {Merge {Times 3 H}
                  {Times 5 H}}}
```

- At first, all the calls to Merge and Times will wait
- When the second value of H is needed, then some calculation will be done
  - The first Merge is activated
  - This will activate Times and the second Merge
  - The second Merge will activate the last two Times
  - This will cause the second value to be calculated

Importance of declarative concurrency

Why is declarative concurrency important?

- Declarative concurrency is much easier to program with than more standard paradigms (e.g., Java style with monitors)
  - Programs have no race conditions, i.e., results that depend on exact timing, which makes them unpredictable
  - Programs have no memory, i.e., internal state that can get a wrong value
- It does have a limitation, though
  - It cannot express nondeterminism, e.g., when programs have multiple independent inputs from the external world
  - This is not usually a problem, because nondeterminism can be isolated to a small part of the program
  - We recommend this programming style!

Declarative concurrent model

<s> ::=	
skip	*Empty statement*
<s>$_1$ <s>$_2$	*Sequential composition*
proc {<x> <x>$_1$ … <x>$_n$} <s> **end**	*Procedure creation*
{<x> <x>$_1$ … <x>$_n$}	*Procedure invocation*
thread <s> **end**	*Thread creation*
local <x> **in** <s> **end**	*Variable creation*
<x>=<value>	*Variable binding*
if <x> **then** <s>$_1$ **else** <s>$_2$ **end**	*Conditional (synchronizes on bind)*
case <x> **of** <p> **then** <s>$_1$ **else** <s>$_2$ **end**	*Pattern matching (synchronizes on bind)*
{WaitNeeded <x>}	*By-need synchronization*

- Declarative concurrency adds threads and single-assignment variables with dataflow synchronization to a simple functional language
  - This is a process calculus that is a subset of Oz
  - Declarative concurrency adds “slack” between producer and consumer
- Lazy evaluation adds by-need synchronization
  - Lazy evaluation does coroutining between producer and consumer

Message passing and multiagent systems

Message-passing concurrency

- Multiagent systems
  - In this paradigm, programs consist of independent entities (called “agents”) that communicate through asynchronous message passing
  - The agents work together to achieve a common goal
- We can implement agents by adding just one new concept, a communication channel
  - Note that this removes the limitation of the declarative concurrent model: the channel can accept inputs from the external world

Communication channel

- We add a simple communication channel, called a port

```
declare S P in
{NewPort S P}
```

- A port P has a corresponding stream S
- Messages sent to the port will appear on S

```
{Browse S}
{Send P alpha}      % S is alpha|_
{Send P beta}       % S is alpha|beta|_
```

- With a port and a thread we can make an agent

Defining an agent (1)

- We define an agent with a port, a thread, and a function
  - The thread reads messages M from the port's stream Msgs and calls the function Fun for each message
  - The function has two arguments, the agent's internal state State and the message M, and it returns the new agent state
-

```
fun {NewAgent Init Fun}
   proc {AgentLoop State Msgs}
      case Msgs of M|Msgs2 then
         {AgentLoop {Fun State M} Msgs2}
      [] nil then skip end
   end
   Msgs
in
   thread {AgentLoop Init Msgs} end
   % The NewPort call returns the port as its result:
   {NewPort Msgs}
end
```

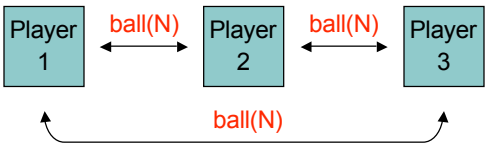
Defining an agent (2)

- A clever programmer will realize that we can define NewAgent with FoldL

```
fun {NewAgent Init Fun}
   Msgs Out in
      thread {FoldL Msgs Fun Init Out} end
      {NewPort Msgs}
end
```

- FoldL is exactly a loop with accumulator: it starts with Init, the second value is {Fun Init M1}, the third value is {Fun {Fun Init M1} M2}, and so forth
  - Each new value Mi on the message stream is accumulated
- Out is the final state when the stream terminates

Three agents playing ball



- Let us define a simple multiagent system with three agents
- Each agent upon receiving a ball(N) message will send a ball(N+1) message to a randomly chosen other player
- Each agent will count the number of ball(N) messages it has received and keep track of N
- Each agent also accepts a getstate(S) message and will bind its internal state to S. This lets us observe the agent's behavior.

A ball-playing agent

```
fun {Player Others}
   {NewAgent state(0 0)
      fun {$ state(M B) Msg}
         case Msg of ball(N) then
            Ran=({OS.rand} mod {Width Others})+1
         in
            {Send Others.Ran ball(N+1)}
            state(M+1 N)
         [] getstate(S) then S=state(M B)
            state(M B)
         end
      end
   end}
end
```

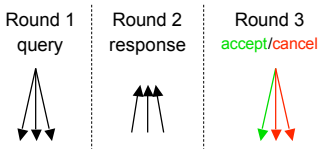

Playing a game

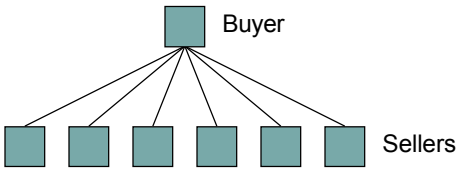
- Create the three players
  P1={Player others(P2 P3)}
  P2={Player others(P1 P3)}
  P3={Player others(P1 P2)}
- Start the game by tossing in a ball
  {Send P1 ball(0)}
- Observe a game in progress
  {Browse {Send P1 getstate($)}}

Functional building blocks as concurrency patterns

- We can combine the expressive power of functional programming with message-passing concurrency
- Functional building blocks
  - {ForAll L F}: Apply a function to all elements of a list
  - L2={Map L1 F}: Transform all elements of a list
  - X={Fold L F U}: Merge elements of a list together
  - L2={Filter L1 F}: Filter out elements of a list
- We can use these building blocks in message-passing programs
  - They were originally designed for sequential programs, but used in a dataflow setting they become powerful concurrency patterns
- Let us show one example: a contract net protocol

Contract net protocol (1)





- A contract net protocol is a simple negotiation protocol
  - A buyer sends a query to a set of sellers
  - Each seller sends a response with the price
  - The buyer then chooses the best price, sends an accept to that seller, and a cancel to the others

Contract net protocol (2)

- Assume that Sellers is a list of sellers
- Then we can program a contract net protocol in just four lines of code:

```
% Send queries and collect seller/price responses
Rs={Map Sellers fun {$ S} S#{Send S query($)} end}

% Find seller/price pair with lowest price
S1#R1={FoldL Rs.2 fun {$ S1#R1 S#R}
                    if R<R1 then S#R else S1#R1 end end Rs.1}

% Send accept to best seller, cancel to others
for S#R in Rs do {Send S if S==S1 then accept else cancel end} end
```

- Map is both a broadcast and convergecast (send and collect responses)
- FoldL combines all the results
- ForAll (**for**) is a broadcast

Contract net protocol (3)

- This example may seem straightforward, but there is more here than meets the eye
  - Everything is asynchronous
- For example, the Map causes messages to be sent and responses to be collected in a list right away, without waiting for them to arrive
  - What happens if the FoldL is executed before all the responses arrive? Some of the elements in the list Rs can still be unbound variables when FoldL executes.
  - This is not a problem: the FoldL operation will suspend and wait whenever it encounters a response that is not available yet
  - So everything works out right, even though the messages are sent asynchronously and the responses can come at any time
  - The reason why everything works out right is the dataflow synchronization

Asynchronous dataflow programming

- The programming style illustrated by the contract net is quite general and useful
  - A combination of asynchronous communication, dataflow synchronization, and functional programming
    - Asynchronous communication (messages between independent entities) ensures loose coupling
    - Dataflow synchronization exactly where needed and not before (implicit synchronization when the variable values are needed, no explicit synchronization operations)
    - Functional programming makes the code compact and easy to reason about (higher-order building blocks and symbolic data structures)
- This style deserves to be more widely used
  - It should be supported by the language

State and objects

Mutable state

- Mutable state consists of variables that can be assigned multiple times
  - We have avoided them so far
- Most languages use them from square one
  - We don't, because they make life complicated, especially in concurrent programs!
  - The usual object-oriented techniques rely too much on them
- Why do we need them?
  - Their main use is for achieving modularity
  - They don't really have another use

Modularity

- A program is a set of building blocks ("components") that communicate with each other
- A component can have an internal memory (the mutable state) and a way to change that memory
- If done right, changing the internal memory lets us update the component without changing the rest of the system
  - This is what we mean by modularity
  - Mutable state allows to change components and reconfigure the system
- This is explained in detail in the textbook

Object-oriented programming

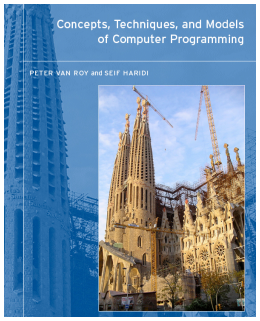
- We have almost reached the end of the talk and I have not mentioned object-oriented programming
  - What's going on here?
  - Since we're talking about concurrency, where are the monitors (synchronized objects, in Java terminology)?
- Object-oriented programming is a way to structure programs
  - I have given only small examples, which don't need these structuring mechanisms
  - Larger programs use object-oriented techniques
    - Modularity comes from using objects with mutable state
    - Polymorphism helps to apportion responsibility
    - Inheritance helps to organize data abstractions
  - Monitors are cumbersome and error-prone

Conclusions

- We have shown how to pack a lot of power into a few concepts
  - Functions and higher-order programming
  - Symbolic data structures
  - Dataflow variables
  - Threads and declarative concurrency
  - Lazy evaluation
  - Communication channels and multiagent systems
  - Asynchronous dataflow programming
  - Mutable state, modularity, and object-oriented programming
- These concepts and many others are explained in our programming textbook
- There are many other concepts that we have not touched in this talk; they are ongoing work
  - Software transactional memory
  - Functional reactive programming

Appendix

Teaching programming as a unified discipline



Goal of the talk revisited

- This talk has given a fast overview of many programming concepts
  - We emphasized intuition and expressiveness
- But there is much more: these concepts and others are part of a comprehensive programming framework and they can be used to teach programming
  - We have developed a way to teach programming based on gradually introducing new concepts and showing what they are good for
  - We show how all major programming paradigms fit in a uniform framework
- This appendix explains and motivates the approach

Teaching programming (1)

- What is programming?
  - We define it broadly as “extending or changing a computer system’s functionality” or “the activity that starts from a specification and leads to a running system over its lifetime”
- How can we teach programming without being affected by historical accidents of current languages and systems?
- We can teach programming by starting with a simple language and **adding features** (Holt 1977)
- A more principled approach is to **add programming concepts**, not language features, e.g., Abelson & Sussman (1985, 1996) in “Structure and Interpretation of Computer Programs”: add mutable state to a functional language, leading to object-oriented programming

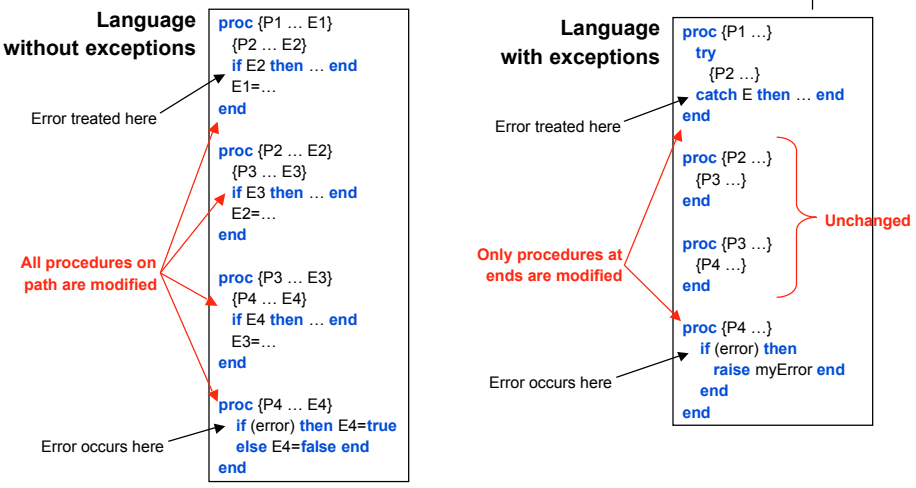
Teaching programming (2)

- In 1999, Seif Haridi and I realized that we could apply this approach in a very broad way by using the Oz language
  - The Oz language was explicitly designed to contain many concepts in a factored way (long-term design effort by Gert Smolka and many others)
  - For example, we realized that **a good second concept is *concurrency*** (Kahn 1974). This lets us keep the good properties of functional programming in a concurrent setting. It works well when there are no external sources of nondeterminism.
- We have written a textbook that reconstructs Oz in a layered way according to a general principle that indicates when to add a concept and what concepts to add
  - Our reconstruction can be seen as **a partially ordered set of process calculi based on *programmer-significant concepts***: they avoid the clutter of the encodings needed by compilers (to map to physical architectures) and by other process calculi (to map program abstractions)
  - Textbook: "Concepts, Techniques, and Models of Computer Programming", MIT Press, 2004, 929 pages

Creative extension principle

- A general principle to design a language in layered fashion by overcoming limitations in expressiveness
- With a given language, when programs start getting complicated for technical reasons unrelated to the problem being solved (non-local changes are needed), then there is a new programming concept waiting to be discovered
  - Adding this concept to the language recovers simplicity (local changes)
- A typical example is exceptions
  - If the language does not have them, all routines on the call path need to check and return error codes (non-local changes)
  - With exceptions, only the ends need to be changed (local changes)
- We rediscovered this principle when writing our textbook
  - Originally defined by (Felleisen 1990)
- This principle applies to all the programming concepts we cover

Example of creative extension principle



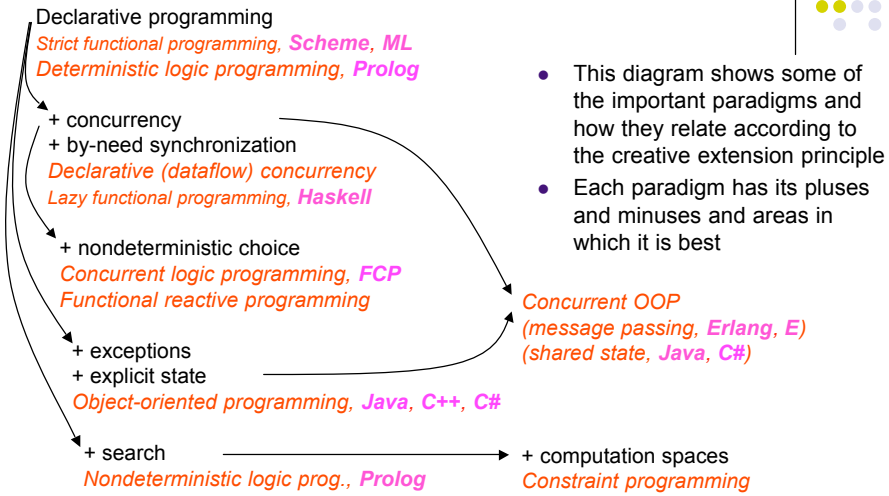
Complete set of concepts (so far)

<s> ::=		
skip	*Empty statement*	Descriptive declarative
<x>$_1$=<x>$_2$	*Variable binding*	
<x>=<record> | <number> | <procedure>	*Value creation*	
<s>$_1$ <s>$_2$	*Sequential composition*	
local <x> **in** <s> **end**	*Variable creation*	
if <x> **then** <s>$_1$ **else** <s>$_2$ **end**	*Conditional*	Declarative
case <x> **of** <p> **then** <s>$_1$ **else** <s>$_2$ **end**	*Pattern matching*	
{<x> <x>$_1$ … <x>$_n$}	*Procedure invocation*	
thread <s> **end**	*Thread creation*	
{WaitNeeded <x>}	*By-need synchronization*	
{NewName <x>}	*Name creation*	Less and less declarative
<x>$_1$= !!<x>$_2$	*Read-only view*	
try <s>$_1$ **catch** <x> **then** <s>$_2$ **end**	*Exception context*	
raise <x> **end**	*Raise exception*	
{NewPort <x>$_1$ <x>$_2$}	*Port creation*	
{Send <x>$_1$ <x>$_2$}	*Port send*	
<space>	*Encapsulated search*	

Complete set of concepts (so far)

<s> ::=	
skip	*Empty statement*
$\langle x \rangle_1 = \langle x \rangle_2$	*Variable binding*
<x>=<record> | <number> | <procedure>	*Value creation*
$\langle s \rangle_1$ $\langle s \rangle_2$	*Sequential composition*
local <x> **in** <s> **end**	*Variable creation*
if <x> **then** $\langle s \rangle_1$ **else** $\langle s \rangle_2$ **end**	*Conditional*
case <x> **of** <p> **then** $\langle s \rangle_1$ **else** $\langle s \rangle_2$ **end**	*Pattern matching*
{<x> $\langle x \rangle_1$ … $\langle x \rangle_n$}	*Procedure invocation*
thread <s> **end**	*Thread creation*
{WaitNeeded <x>}	*By-need synchronization*
{NewName <x>}	*Name creation*
$\langle x \rangle_1$ = !!$\langle x \rangle_2$	*Read-only view*
try $\langle s \rangle_1$ **catch** <x> **then** $\langle s \rangle_2$ **end**	*Exception context*
raise <x> **end**	*Raise exception*
{NewCell $\langle x \rangle_1$ $\langle x \rangle_2$}	***Cell creation*** } **Alternative**
{Exchange $\langle x \rangle_1$ $\langle x \rangle_2$ $\langle x \rangle_3$}	***Cell exchange*** } **Alternative**
<space>	*Encapsulated search*

Taxonomy of paradigms



- This diagram shows some of the important paradigms and how they relate according to the creative extension principle
- Each paradigm has its pluses and minuses and areas in which it is best

History of Oz

- The design of Oz distills the results of a long-term research collaboration that started in the early 1990s, based on concurrent constraint programming (Saraswat, Maher, Ueda)
  - ACCLAIM project 1991-94: SICS, Saarland University, Digital PRL, …
    - AKL (SICS): unifies the concurrent and constraint strains of logic programming, thus realizing one vision of the Japanese FGCS
    - LIFE (Digital PRL): unifies logic and functional programming using logical entailment as a delaying operation (logic as a control flow mechanism)
    - Oz (Saarland U): breaks with Horn clause tradition, is higher-order, factorizes and simplifies previous designs
  - After ACCLAIM, several partners decided to continue with Oz
  - Mozart Consortium since 1996: SICS, Saarland University, UCL
- The current language is Oz 3
  - Both simpler and more expressive than previous designs
  - Distribution support (transparency), constraint support (computation spaces), component-based programming
  - High-quality open source implementation: Mozart Programming System, http://www.mozart-oz.org