Convergence in Language Design: A Case of Lightning Striking Four Times in the Same Place

April 25, 2006

FLOPS 2006

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

Language convergence

- This talk will present four case studies of big research projects that tackled important problems in computer science
  - The four projects all considered language design to be part of their solution
- The surprise is that the four projects came up with language structures that have much in common
  - We will speculate on what this means in terms of one possible *definitive* programming language: a language that provides "good enough" solutions so that computer scientists can move on
  - We believe that some day, a small set of definitive languages will exist
- This talk is intended to provoke discussion!

The four projects

- Programming **highly available systems** for telecommunications
  - By Joe Armstrong and his colleagues at the Ericsson Computer Science Laboratory
  - Designed the Erlang language and system, which was used to build significant products (e.g., AXD 301 ATM switch, Bluetail Mail Robustifier, Alteon SSL Accelerator)
- Programming **secure distributed systems** with multiple users and multiple security domains
  - By Doug Barnes, Mark S. Miller, and the E community
  - Designed the E language and system; ideas originate in the Actor model
- Making **network-transparent distribution** practical
  - By Seif Haridi, Peter Van Roy, Per Brand, Gert Smolka, and their colleagues
  - Designed the Distributed Oz language and system
- Teaching **programming as a unified discipline** covering all popular programming paradigms
  - By Peter Van Roy and Seif Haridi, aided by their colleagues
  - "Reconstructed" Oz and wrote a textbook organized according to programming concepts

Our bias

- Both the network transparency project and the teaching programming project involved some of the same people
- Both projects were undertaken because we believed that the factored design of Oz would be an adequate starting point
  - Concurrent constraints at the core, with orthogonal extensions
- In the final analysis, both projects give good reasons why their solutions are appropriate
  - We will let you judge!
- (In fact, it was thinking about these projects that led us to see the coincidences that this talk presents)

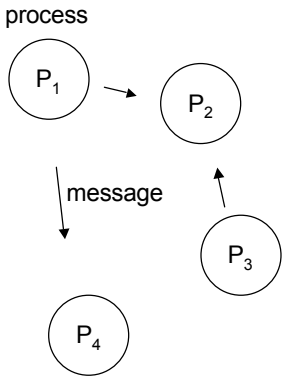
Common language structure

- Each language has a layered structure with 3 or 4 layers
- The inner layers are used most but the outer layers cannot be missed
- Let us give some of the basic insights of each solution

Erlang	E	Oz (distribution)	Oz (teaching)	
Database (Mnesia) for fault tolerance	—	State with global coherence for sharing and collaboration; transactions for latency and fault tolerance	Mutable state for modularity	**Shared-state concurrency**
Fault tolerance through isolation; linking and supervisors to handle failures	Messages between objects in different vats, security through isolation	Asynchronous messages to hide latency between processes; no global state	Multiagent systems are easy to program and reason about and widely applicable	**Message-passing concurrency**
—	"Event loop" concurrency inside a vat (OS process): all objects share one thread (no interleaving)	Dataflow concurrency with efficient distributed unification protocol	A form of concurrency that preserves functional properties if there are no sources of nondeterminism	**Deterministic concurrency**
Lightweight process defined by function, hot code updating	Objects are recursive functions with local state	Functions, classes, and components are values with efficient distributed protocols	Lexically scoped closure: central concept in programming	**Strict functional**

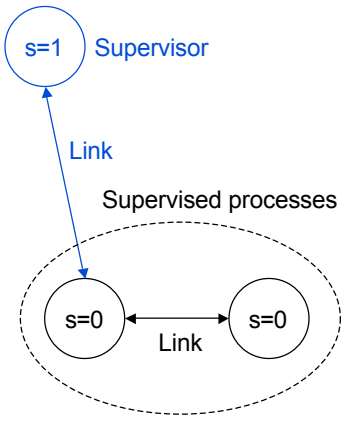
Fault-tolerant programming

Concurrency in Erlang



- Erlang is a dynamically typed language with processes and functions
- An Erlang program consists of a possibly very large number of processes, which each executes in its own address space
- Processes send messages asynchronously and receive them through mailboxes (message queues with extraction through pattern matching and optional time-out)
- The behavior of a process is defined by a transition function f:M×S→S where M is the set of messages and S is the set of states
- Process identities and functions are values that can be put inside data structures
- A process can replace its defining function during execution ("hot update")

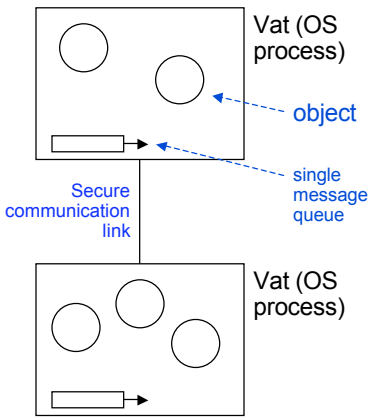
Fault tolerance in Erlang



- Erlang is designed to write programs that survive both software and hardware faults
  - **Software faults** are considered to be the most important to correct
- Two processes can be **linked**; if one fails then both are terminated
  - Programming methodology encourages to let processes fail if anything goes wrong: "**let it fail**" philosophy
- If a linked process has its supervisor bit set, then the process is sent a message instead of failing
- This basic mechanism is used to notify programs for recovery
  - Recovery using **supervisor trees**
- Erlang has a **database**, Mnesia, that supplements the supervisor tree abstraction (maintain invariants)

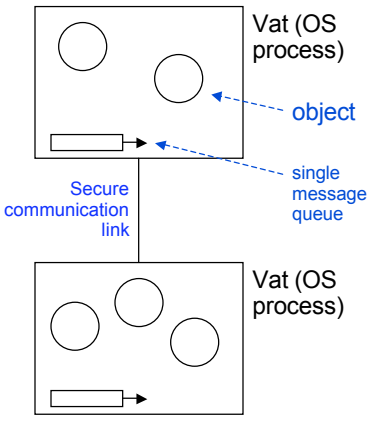
Secure distributed programming

Security with E and POLA



- An E program consists of objects (functions encapsulating a state) hosted in secure processes called **vats** that communicate through a secure message-passing protocol based on encryption
- All language references (including object references) are **capabilities**: they are unforgeable and unbreakably combine designation with authority
- E programs support the **Principle of Least Authority**: the only way to get a capability is if you are passed it by a capability that you already have (no "ambient authority")
- E researchers have built a capability-aware desktop, **CapDesk**, and applications on top of it to show that capability security can be done from the user interface down to single object references without compromising programmability and usability
- What is programming in E like? It is like OOP but (1) **without cheating** and (2) **with both asynchronous and synchronous calls**.

Concurrency in E



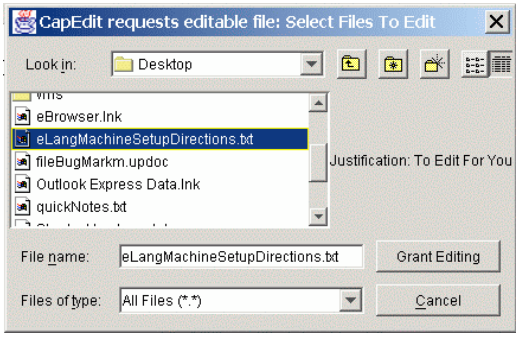
- Within each vat there is a single thread with its message queue; all asynchronous object invocations in the vat pass through that queue
  - Deterministic concurrency for objects in the same vat
  - Synchronous object invocations behave like sequential procedure calls
- Vats are concurrent; an object in a vat can send an asynchronous message to an object in another vat; the message is added to the queue
  - Message-passing concurrency for objects in different vats
- Vats communicate through a secure protocol called Pluribus

Lessons from E

- **Security in depth**: even if one level of security is breached, there is still much protection
  - Completely different from the usual approach, in which breaking through one level ("getting in") gives you all the rights of a legitimate user
  - **The E thesis**: if operating systems and languages were built in this way, the virus problem would largely go away
  - **The E problem**: since this is impractical, how can we add POLA to existing systems?
- **POLA in the GUI**: modify user interface so that GUI selection = giving authority, then this level of security is gained without reducing usability
  - Requires application written in CapDesk (e.g., CapEdit)
- **Polaris project** (HP Labs): add POLA to existing applications

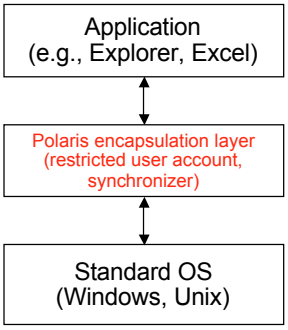
Example: POLA in the GUI

- Does more security have to mean less usability?
  - Not necessarily!
- In CapEdit, clicking on a file both designates the file and gives the editor the right to change it
- The editor only has the rights you give it, no more



Example: Polaris project

- Redesigning existing systems for POLA is impractical
- The Polaris project gets part of the way
  - On a standard OS (e.g., Windows or Unix), it encapsulates an application and manages the authority given to the application
    - It launches the application in a restricted user account
    - It replaces dialog boxes by capability-enabled dialog boxes
    - Files are kept synchronized between the standard user account and restricted account
    - Authority is transferred through the synchronizer, which is a reference monitor
  - Works with standard, unaltered applications
- If the application (e.g., Explorer or Excel) gets a virus, it can only do restricted damage
- Alpha version available



Network-transparent distributed programming

Network-transparent distributed programming

- Network transparency is impossible or undesirable, right?
  - (Waldo *et al*, 1994) give four critiques: pointer arithmetic, partial failure, latency, and concurrency
- Right, if you want to hide the network completely
- Wrong, if your goal is to simplify distributed programming
  - The goal is to separate the functionality of a program from its network behavior (performance, partial failure). Even if only partly realized, it is a gain.
  - "Network transparency with network awareness" (Cardelli, 1995)
- Given proper language design, Waldo's critiques no longer hold and distributed programming is indeed simplified
  - Distributed execution model for Oz, implemented in Mozart system (since 1995)

Basic principles

- *Refine* language semantics with a distributed semantics
  - Separates functionality from distribution structure (network behavior, resource localization, fault behavior)
- Three properties are crucial:
  - Transparency
    - Language semantics identical independent of distributed setting
    - Controversial, but let's see how far we can push it, *if* we can also think about language issues
  - Awareness
    - Well-defined distribution behavior for each language entity: simple and predictable
  - Control
    - Choose different distribution behaviors for each language entity
    - Example: objects can be stationary, cached (mobile), asynchronous, or invalidation-based, with same language semantics

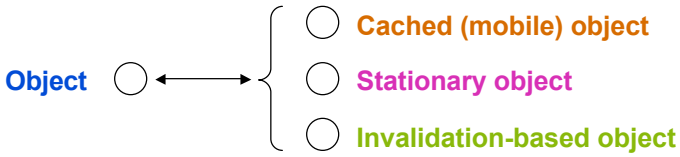
Layered language design

- Oz language has a layered structure with three layers:
  - Strict functional core (stateless): exploit the power of lexically scoped closures
  - Single-assignment extension (dataflow variables + concurrency + laziness): provides a simple but powerful model for concurrent programming ("declarative concurrency")
  - State extension (mutable pointers / communication channels): provides the advantages of state for modularity (object-oriented programming, many-to-one communication and active objects, transactions)
- Layered structure is well-adapted for distributed programming
  - Distributed implementation is more efficient for the inner layers
  - Layered structure is not new: see, e.g., pH (Haskell + I-structures + M-structures), Smalltalk (blocks are closures), even Java (support for immutable objects, concurrency layer)
- Dataflow extension is well-integrated with state: to a first approximation, it can be ignored by the programmer (it is not observable whether a thread temporarily blocks while waiting for a variable's value to arrive)

Distributed implementation

- Use different distributed algorithms according to layer
  - Stateless (records, classes, procedures, software components)
    - Coherence assured by copying (eager immediate, eager, lazy algorithms)
  - Single-assignment (dataflow variables)
    - Allows to decouple communications from object invocations
    - To first approximation: can be completely ignored by the programmer
    - Uses distributed unification algorithm (in between stateless and stateful!)
  - Stateful (objects, communication channels, component instances)
    - Synchronous: stationary, cached (mobile state), invalidation protocols
    - Asynchronous FIFO: channels, asynchronous object calls
    - Transaction protocols
- Each language entity is implemented with one or more distributed algorithms
  - Choice of distributed algorithm allows tuning of network performance
- Does it give a warm, fuzzy feeling?
  - Well, yes (I can give a demo)

Distributed objects



- Each object is implemented with a distributed algorithm
  - Performance and fault behavior depend on the algorithm
- The programmer has just one new operation, passing a language reference from one process (called "site") to another. This means that all processes conceptually form one store.
  - We provide an ASCII representation of language references (called "ticket"), which allows passing references through any medium that accepts ASCII (Web, email, files, phone conversations, …)
- How do we do fault tolerance? By reflecting the faults in the language.

Distributed object example (1)

```
class Coder
  attr seed
  meth init(S) seed:=S end
  meth get(X)
    X=@seed
    seed:=(@seed*23+49) mod 1001
  end
end

% Create a new object C
C={New Coder init(100)}

%  Create a ticket for C
T={Connection.offer C}
```

- Define a simple random number class, Coder
- Create one instance, C
- Create a ticket for the instance, T
- The ticket is an ASCII representation of the object reference

Distributed object example (2)

```
% Use T to get a reference to C
C2={Connection.take T}

local X in
  % invoke the object
  {C2 get(X)}
  % Do calculation with X
  ...
end
```

- Let us use the object C on a second site
- The second site gets the value of the ticket T (through the Web or a file, etc.)
- We convert T back to an object reference, C2
- C2 and C are references to the same object

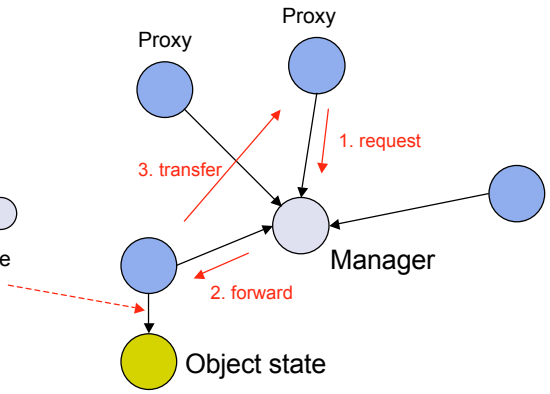
What distributed algorithm is used to implement the object?

Distributed object example (3)

- For the programmer, C and C2 are the **same object**: the distributed algorithm guarantees coherence
- Many distributed algorithms are possible, as long as the language semantics are respected
- By default, we use a ***mobile state algorithm***: the object state moves synchronously to the invoking site using a distributed token passing protocol. This makes the semantics easy, since all object execution is local (e.g., exceptions are raised in local threads).
- Other possibilities are a stationary object (behaves like a server, similar to RMI), an invalidation-based object, etc.

Mobile state algorithm

- Distributed token passing algorithm implements state updates for the object state (Van Roy *et al* 1997)
- Formalize the distributed algorithm as message passing between active entities
  - One "proxy" per site
  - "Manager" serializes state requests
- The object state is *mobile*; to be precise, the *right to update the object state* is mobile, moving synchronously to the invoking site
- All object execution is local



Asynchronous objects (1)

- Objects with mobile state still have the synchronous behavior of centralized objects (they keep the same semantics)
  - This means that a **round trip delay** is needed for the first invocation
- In a distributed system, asynchronous communication is more natural
  - To achieve it, we use dataflow variables: single-assignment variables that can be in one of two states, unbound (the initial state) or bound
- The use of a dataflow variable is transparent: it can be used as if it were the value
  - If the value is not yet available when it is needed, then the thread that needs it will simply suspend until the value arrives
  - Example:

```
thread X=100 end          Y=X+100
(binds X)                 (uses X)
```

- A distributed rational tree unification algorithm is used to implement this behavior: the key ideas are an arbitration (first binding wins) and a broadcast of the binding

Asynchronous objects (2)

- Used just like normal objects
- Return values are passed with dataflow variables:

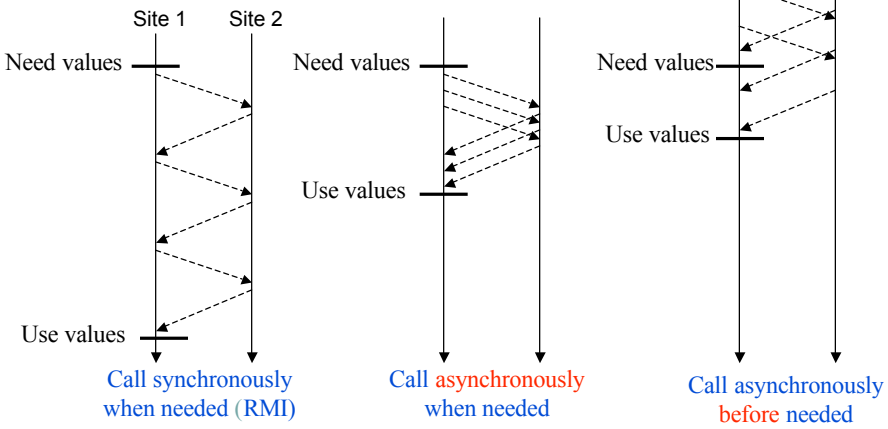
```
C={NewAsync Coder Init}
(create on site 1)

{C get(X1)}     % Asynchronous calls
{C get(X2)}
{C get(X3)}
X=X1+X2+X3  % Synch happens here
(calls on site 2)
```

- Can synchronize on error
  - Exception raised by object:
    {C get(X1) E}
    (synchronize on E)
  - Error due to system fault (crash or network problem):
    - Attempt to use return variable (X1 or E) will signal error (lazy detection)
    - Eager detection also possible

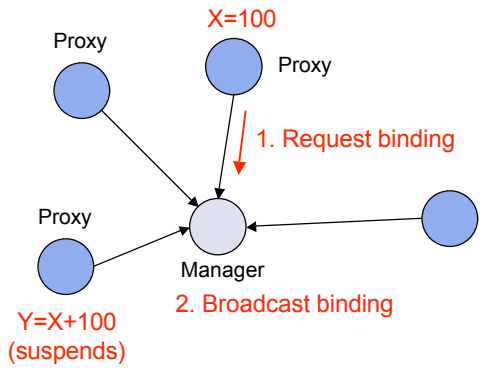
Asynchronous objects (3)

Improve network performance without changing the program!



Distributed unification algorithm

- Distributed rational tree unification implements each dataflow variable (Haridi *et al* 1999)
- It is a remarkable fact that this algorithm requires only one distributed operation: a distributed bind
  - Request binding to manager, which arbitrates (first request wins)
  - Manager broadcasts binding
- All other unification operations are local
- Laziness is supported by a "need" message to the manager



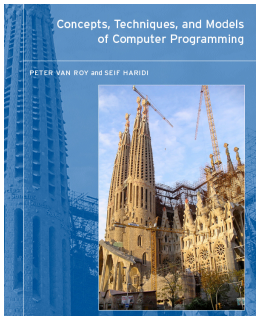
Fault tolerance

- **Reflective failure detection**
  - Reflected into the language, at the level of single language entities
  - We have looked at two kinds: permanent process failure and temporary network failure
  - Both synchronous and asynchronous detection
    - Synchronous: exception when attempting language operation
    - Asynchronous: language operation blocks; user-defined operation started in new thread
    - Our experience: asynchronous is better for building abstractions
- Building fault-tolerant abstractions
  - Using reflective failure detection we can build abstractions in Oz
  - Example: *transactional store*
    - Set of objects, replicated and accessed by transactions
    - Provides both fault tolerance and latency tolerance
    - Lightweight: no persistence, no dependence on file system
  - Example: *structured overlay network (peer-to-peer)*

Beyond network transparency

- Network transparency is only the first step
  - It works well for small numbers of nodes
  - But it doesn't handle larger systems because the programming abstractions of centralized computing are not enough
- Abstractions for large systems: start with structured overlay networks
  - Provide decentralized communications and storage services in a scalable and robust manner
  - An outgrowth of peer-to-peer that adds guarantees and efficiency
- Next step: self-managing systems
  - When "abnormal" behavior becomes frequent, the system architecture must handle self-{configuration, healing, protection, tuning} at all levels
  - Our vision (SELFMAN project starting in 2006): Combine structured overlay networks with an advanced component model; build monitoring services using techniques from physics (e.g., belief propagation) and structure systems according to general system theory (cybernetics!)

Teaching programming as a unified discipline



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) add mutable state to a functional language, leading to object-oriented programming

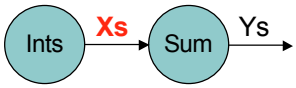
Teaching programming (2)

- In 1999, Seif Haridi and I realized that we could apply this approach in a very broad way by using Oz
  - The Oz language was explicitly designed to contain many concepts in a factored way (long-term design effort by Gert Smolka and many others)
  - 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 wrote 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

The second model: declarative concurrency

- 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 A Xs}
   case Xs of X|Xr then
      A|{Sum A+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

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
  - The above example 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

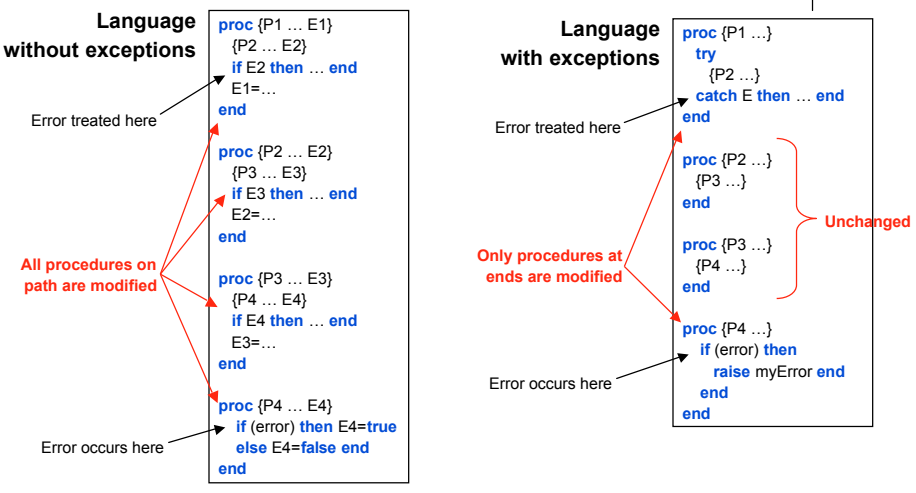
Why is declarative concurrency important?

- A typical programming style consists of concurrent entities that read input streams and write output streams
  - A stream is an incrementally constructed value with a single writer and multiple readers
- However, this model has a strong limitation: it cannot express nondeterminism
  - It cannot be used for programs that have to handle nondeterminism, e.g., that have multiple independent inputs from the external world
- So why is it useful?
  - It is a form of functional programming
  - "Partial termination": for a given momentary configuration of input streams, the output streams eventually reach a stable configuration that is a function of the input streams (a.k.a. "resting point")
  - In most cases the nondeterminism can be isolated to a small part of the program

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*
<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*
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*
<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*

Descriptive declarative

Declarative

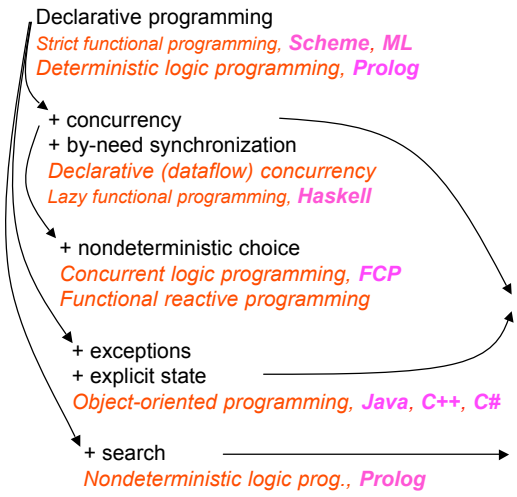
Less and less declarative

Complete set of concepts (so far)

<s> ::=	
skip	*Empty statement*
<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*
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*
<x>$_1$= !!<x>$_2$	*Read-only view*
try <s>$_1$ **catch** <x> **then** <s>$_2$ **end**	*Exception context*
raise <x> **end**	*Raise exception*
{NewCell <x>$_1$ <x>$_2$}	***Cell creation***
{Exchange <x>$_1$ <x>$_2$ <x>$_3$}	***Cell exchange***
<space>	*Encapsulated search*

Alternative

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

Conclusions

Conclusions

- We have presented four substantial research projects that did language design as part of their solution
  - Because they did language design, they give deeper insights than projects that use a fixed language
  - Doing language design is difficult and risky, not a light undertaking
- The four languages have a common, layered structure: a functional core, then deterministic concurrency, then message-passing concurrency, then shared-state concurrency
  - Deterministic concurrency keeps the good properties of functional programming, but can't handle external sources of nondeterminism
  - Message passing generalizes this to handle nondeterminism
  - State adds an important modularity property but is hard to program when combined with concurrency (transactions are one solution)
- Does this tell us something about what a definitive language might look like?
  - Determinism (functional) < monotonicity < isolation < modularity?