HOPL IV

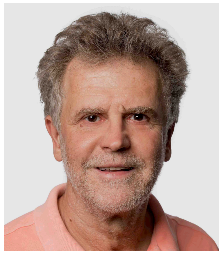
mozart

A History of the Oz Multiparadigm Language

Peter Van Roy and Seif Haridi

June 20, 2021





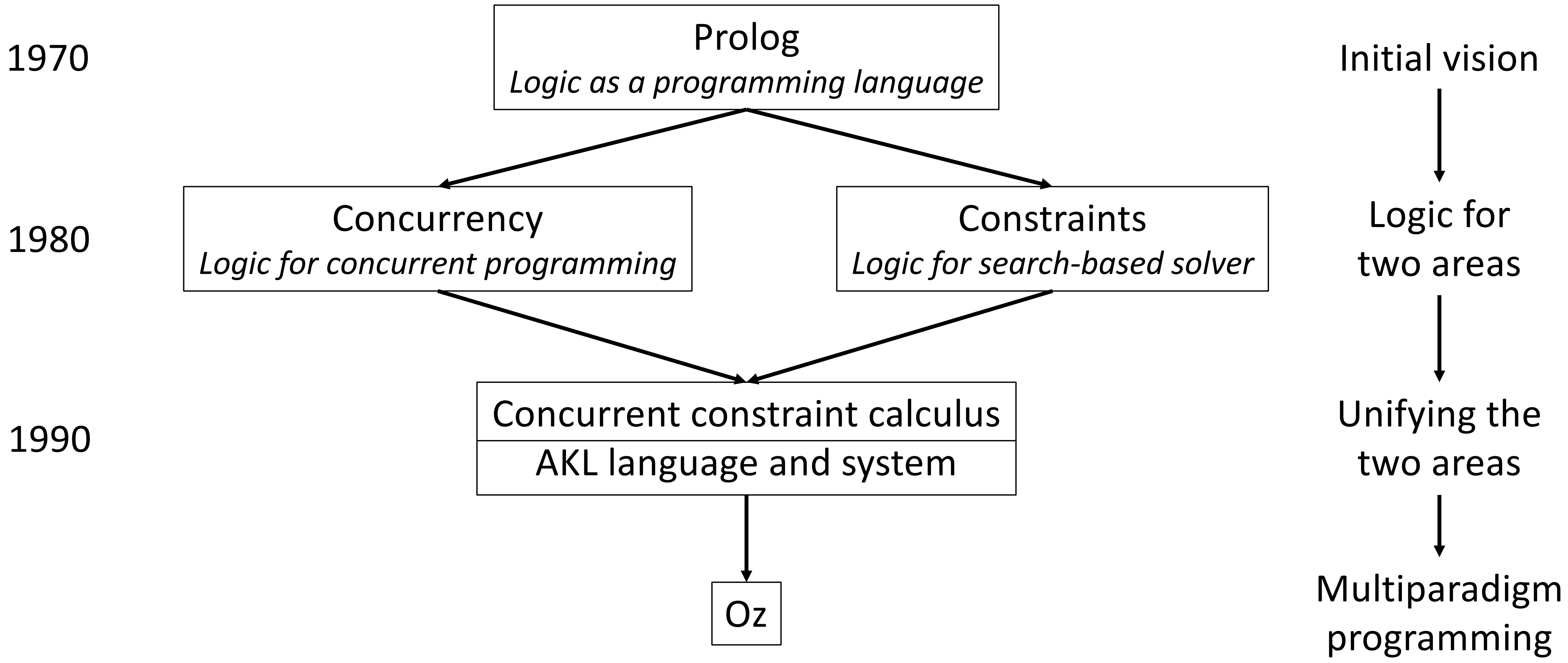

UCLouvain

RI.SE

KTH VETENSKAP OCH KONST

Flashbacks

The prehistory of Oz



HOPL IV

The French Connection

- The place: Marseille, France
- The time: early 1970s

- An opinionated French computer scientist sitting in a bar drinking his Pastis
- Next to him, a dignified Englishman drinking his tea

- Suddenly, lightning strikes!
- Programming in logic, the Prolog language!

HOPL IV

The Rising Sun

- The place: Tokyo, Japan
- The time: early 1980s

- A lanky computer sensei slowly sipping his saké
- Next to him, a samurai of fast systems chugging on his Asahi

- Suddenly, lightning strikes!
- The path to enlightenment is concurrent logic!

HOPL IV

mozart

The Vision and the Foundation

HOPL IV

The Oz vision: multiparadigm programming

- All large programs need more than one paradigm
  - The program may have a database with relational (logical) structure, it may do (functional) transformations, it can use object-oriented principles to structure its data, and concurrency to connect its independent parts
- We would like to support this
  - How can we do it? Where do we start?
- Oz project starting points
  - We design and implement a single language
  - We use concurrent constraints as the semantic foundation

What is a paradigm?

- What is a programming paradigm? Is there even such a thing?
  - Programming is a huge bag of languages and concepts
  - In practice, there are different ways of programming a computer
- For the Oz design, we defined a paradigm as follows:
  - A programming paradigm is an approach to program a computer based on a coherent set of principles or a mathematical theory
- Examples
  - Functional programming: based on lambda calculus
  - Logic programming: based on a formal logic such as Horn-clause logic
  - Object-oriented programming: an approach to organizing data and its operations based on data abstraction, mutable state, and polymorphism

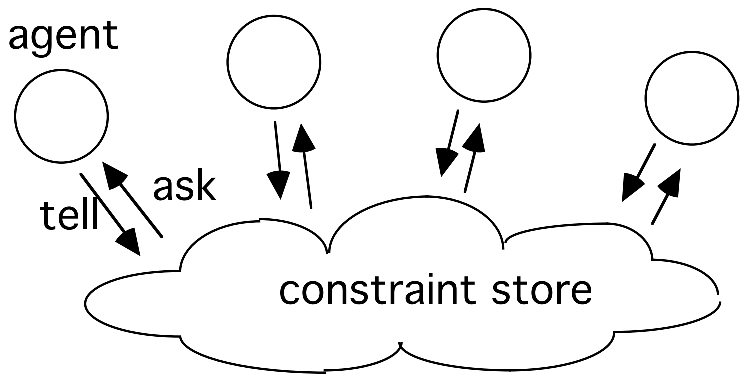
Why do we need a single language?

- Why not just use different languages or libraries?
  - This facilitates upward migration from existing systems
  - Many examples exist, for example the Gecode constraint solver is a C++ library
- This approach has disadvantages
  - Cognitive load on the programmer, increased system complexity
  - Cross-paradigm optimizations are extremely difficult
- This is not a long-term solution
  - As researchers, we aimed for a fundamental, conceptual solution

What is the foundation?

- How do we combine four different paradigms?
  - Functional: lambda calculus
  - Logic: first-order logic
  - Object-oriented: data abstraction and polymorphism
  - Concurrent programming
- We need a foundation in which all these paradigms can fit naturally
  - We chose concurrent constraints as our foundational model
- Why concurrent constraints?
  - All four paradigms can be done in a straightforward way, as we will show
  - It has powerful properties: constraint domains can be defined separately, and synchronization is defined with logical entailment

Concurrent constraint model

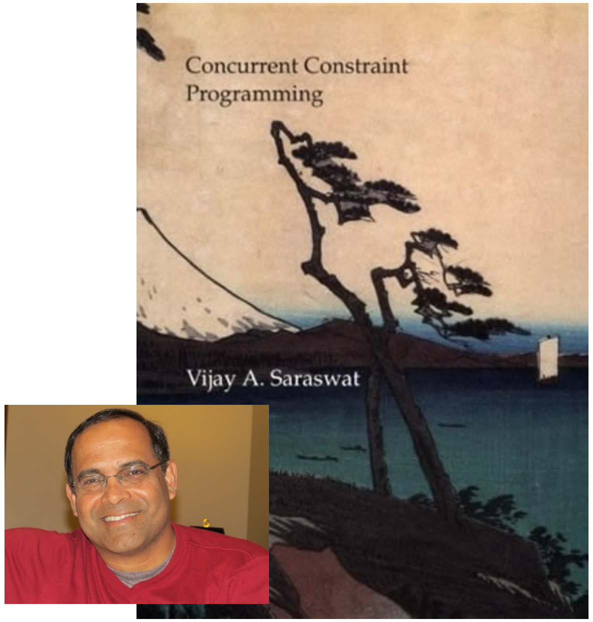


- Agents are programmed in a small language:
  - Ask and tell
  - Concurrent composition, variable definition, procedure definition

- The concurrent constraint model is a calculus for general computation
- The model consists of a shared constraint store observed by concurrent agents
- The store σ contains constraints:
  - $\sigma = c_1 \land c_2 \land \cdots \land c_n$
- There are two basic operations:
  - Tell: add c to σ, which becomes $\sigma \land c$
  - Ask: wait until σ has enough information to decide c or $\neg c$
- Technically, the ask is doing logical entailment which checks $\sigma \vDash c$ and $\sigma \vDash \neg c$

Concurrent constraint language

S	::=	$S_1\ S_2$	*Concurrent composition*
	|	X **in** S	*Variable introduction*
	|	c	*Tell constraint*
	|	**if** C_1 [] C_2 [] ⋯ [] C_n **else** S **end**	*Conditional*
	|	p(X_1 ⋯ X_n)	*Procedure call*
C	::=	X_1 ⋯ X_n **in** c **then** S	*Ask clause*
D	::=	**proc** p(X_1 ⋯ X_n) S **end**	*Procedure definition*




```
proc times2(L1, L2)
   if L1=nil then L2=nil
   [] X M1 in L1=X|M1 then M2 in
        L2=2*X|M2 times2(M1, M2)
end end
```

L1 L2 **in** times2(L1,L2) times2(1|2|3|nil,L1)

First agent *Second agent*

- The first agent initially suspends because the store does not entail L1=nil or L1=X|M1 or their negations
- The second agent will build L1
- As L1 is built, the first agent incrementally builds L2

Synchronization as logical entailment

Concurrent constraints

- Synchronization is when one concurrent computation waits on another
  - Logical entailment to define synchronization
- Consider a store σ={x,y} with variables x, y
  - To do the addition y=x+1, we must know x
  - We do an ask operation:
    if number(x) **then** y=x+1 **end**
  - This tests σ⊨number(x) :
    Does σ know that x is a number?
- This is dataflow behavior
  - Knowledge about x drives the computation

Oz

- Dataflow concurrency is the core of Oz
  - Dataflow variables are the glue that ties together concepts from all paradigms
- Consider this sequential Oz program:

```
local Y in
  Y=X+1
  {Browse Y}  % Display Y
end
```

- Execution will wait at X+1 because X is unbound (ask operation)
  - When X is bound in another thread (tell operation), like X=5, execution continues

HOPL IV

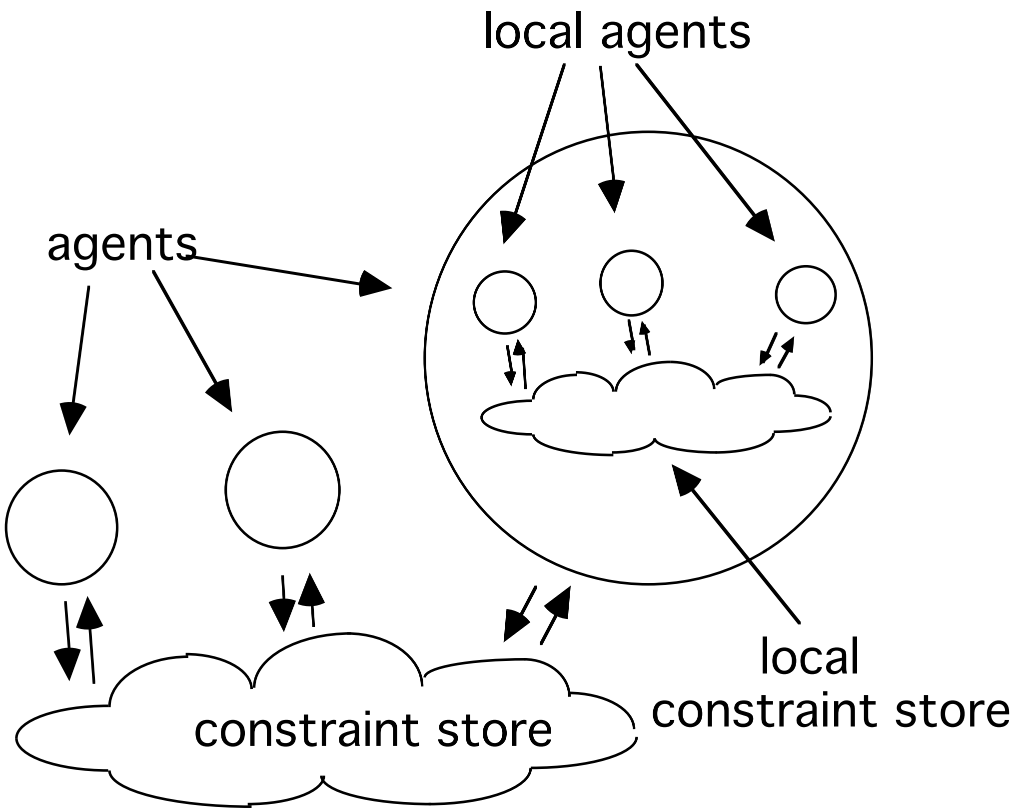
The Road to Oz

Unifying the two areas of logic programming

- In the 1980s, logic programming was divided into two disjoint research areas
  - Concurrent logic programming
  - Constraint logic programming
- A major research problem was how to unify them
  - This would combine reasoning systems with parallel computing
  - This would lead to a society of independent reasoning agents
- The AKL project successfully realized this unification
  - It defined a computation model giving composition of search and concurrency
  - It built a high-quality system that successfully demonstrated the model

AKL: the Swedish ancestor to Oz (1990)

HOPL IV



- The AKL project had the explicit goal of unifying concurrent logic programming and constraint logic programming
- The AKL system was released in 1990 for single and multiprocessor systems
- AKL defines computation spaces, each of which consists of a constraint store and its agents
  - An agent can itself be a computation space; agents see their enclosing stores
  - This gives full compositionality of search and concurrency: First-class search engines can be run concurrently, and conversely, concurrent computations can be used inside search engines

Oz: the German step after AKL (>1990)

- Oz is a direct successor of AKL
  - Oz provides a much richer set of concepts for the programmer
  - Oz layers its kernel language, giving a well-factored language and semantics

- Compared to AKL, Oz adds the following:
  - Full compositionality of all language concepts
  - Compositional syntax (break with AKL's Horn clause syntax from Prolog)
  - Higher-order programming and ubiquitous first-class values
  - Mutable versus immutable data types

Oz language design cycles

- "Oz 0" (1991)
  - This initial language did not yet use the concurrent constraint model
- Oz 1 (1993)
  - The first language influenced by AKL and using the concurrent constraint model
- The Swedish and German groups now joined efforts on a common system
- Oz 2 (1996)
  - The first language that succeeded in the multiparadigm vision
- Oz 3 (1998)
  - Conservative extension for components and distributed computing

Timeline up to Mozart 1.0

HOPL IV

- HYDRA German project (1991) led by Gert Smolka (DFKI)
  - Complex deductive problem solving (needed for many DFKI projects)
- ACCLAIM European project (1992-1995) led by Seif Haridi (SICS, Sweden)
  - Advancing concurrent constraint programming
  - Collaboration between DFKI, SICS, DEC PRL: Oz, AKL, and LIFE languages
  - Oz 1 language (1993): first language, many experimental features → DFKI Oz 1.0 (Jan. 1995)
- PERDIO German project + PERDIO Swedish project (1996-1999)
  - Joined efforts of DFKI and SICS on Oz, Peter Van Roy from PRL to DFKI to UCL
  - First-class computation spaces and constraint programming (Christian Schulte)
  - Oz 2 language (1996): first stable language → DFKI Oz 2.0 (Sep. 1996)
  - Oz 3 language (1998): conservative extension for components and distribution
  - Mozart 1.0 full system (180000 lines C/C++, 140000 lines Oz) → Mozart 1.0 (Jan. 1999) Major release

CCL Workshop (Oct. 1992)

Informal release (Nov. 1993)

DFKI Oz 1.0 (Jan. 1995)

WOz '95 Workshop (Nov. 1995)

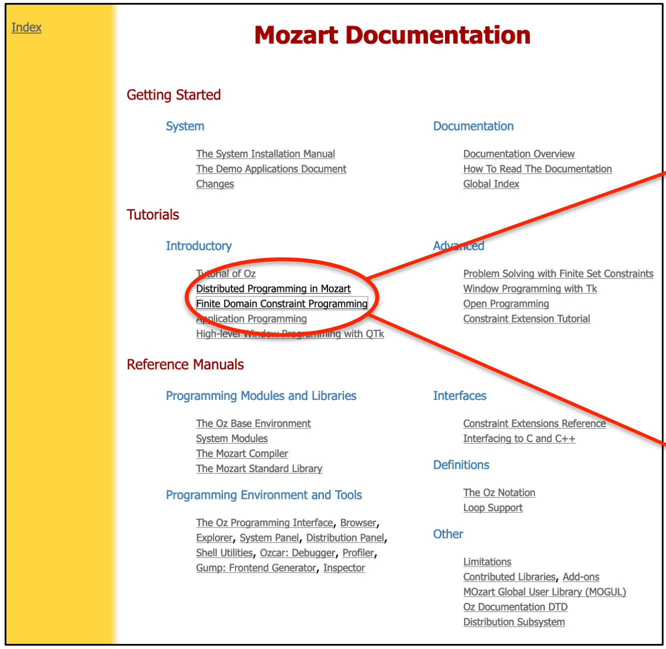
DFKI Oz 2.0 (Sep. 1996)

Mozart 1.0 (Jan. 1999)

Major release

HOPL IV

Mozart 1.0


Index

Mozart Documentation

Getting Started

System

The System Installation Manual
The Demo Applications Document
Changes

Documentation

Documentation Overview
How To Read The Documentation
Global Index

Tutorials

Introductory

Tutorial of Oz
Distributed Programming in Mozart
Finite Domain Constraint Programming
Application Programming
High-level Window Programming with QTk

Advanced

Problem Solving with Finite Set Constraints
Window Programming with Tk
Open Programming
Constraint Extension Tutorial

Reference Manuals

Programming Modules and Libraries

The Oz Base Environment
System Modules
The Mozart Compiler
The Mozart Standard Library

Programming Environment and Tools

The Oz Programming Interface, Browser, Explorer, System Panel, Distribution Panel, Shell Utilities, Ozcar: Debugger, Profiler, Gump: Frontend Generator, Inspector

Interfaces

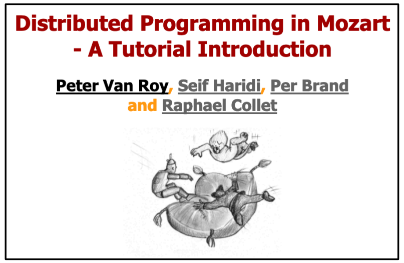
Constraint Extensions Reference
Interfacing to C and C++

Definitions

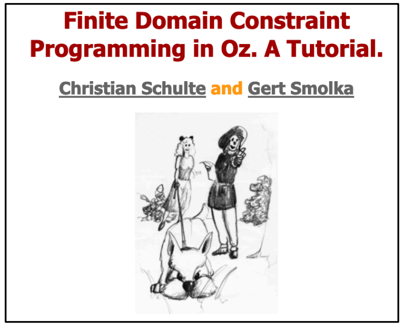
The Oz Notation
Loop Support

Other

Limitations
Contributed Libraries, Add-ons
MOzart Global User Library (MOGUL)
Oz Documentation DTD
Distribution Subsystem


Distributed Programming in Mozart - A Tutorial Introduction

Peter Van Roy, Seif Haridi, Per Brand and Raphael Collet


Finite Domain Constraint Programming in Oz. A Tutorial.

Christian Schulte and Gert Smolka

- Mozart 1.0 released in Jan. 1999
  - BSD style open-source language
  - 10000 downloads during 1999-2001
- Widely used during period 1999-2009
- Last major release Mozart 1.4.0 in 2008

HOPL IV

mozart

The Design Approach

UCLouvain

RISE

KTH VETENSKAP OCH KONST

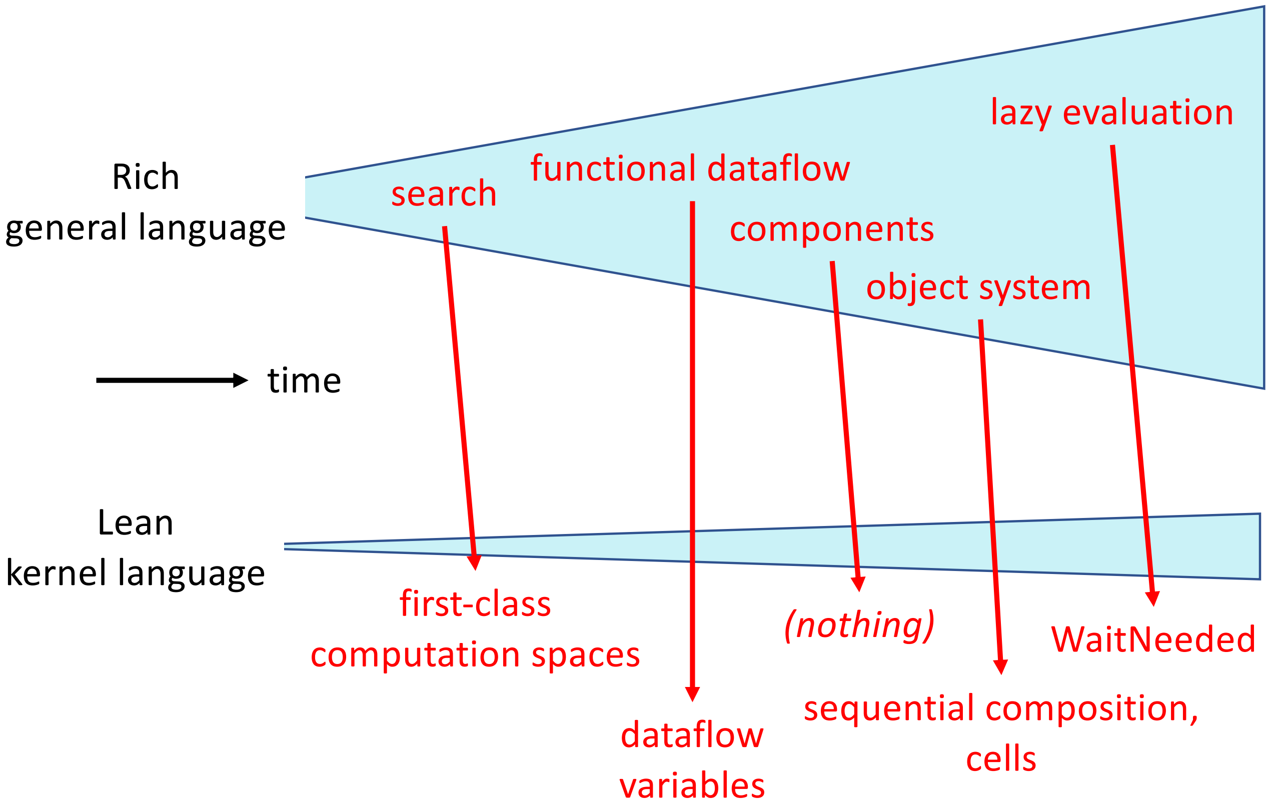
Language design approaches

- Historically, there are two basic approaches to design a language

- One way is to start from the machine and build the language on top
  - Examples are Fortran, C, C++, and recently, Rust
  - Basically, a bottom-up approach: efficiency first

- Another way is to start from a principled design and implement it
  - Examples are Lisp, Prolog, Smalltalk, Erlang, Haskell, Javascript, Python, and Oz
  - Basically, a top-down approach: principles first

Oz design approach: a form of top-down



- The general language is very rich; yet the kernel language was always kept very small
- The developers continuously introduce new abstractions as solutions to practical problems
  - The abstraction is first simplified as much as possible; often it vanishes!
  - A new abstraction is accepted if its implementation is efficient and its semantics is simple
- This methodology achieved the goal of multiparadigm programming

From the concurrent constraint language …

S	::=	S_1 S_2	*Concurrent composition*
	|	X **in** S	*Variable introduction*
	|	c	*Tell constraint*
	|	**if** C_1 [] C_2 [] ⋯ [] C_n **else** S **end**	*Conditional*
	|	p(X_1 ⋯ X_n)	*Procedure call*
C	::=	X_1 ⋯ X_n **in** c **then** S	*Ask clause*
D	::=	**proc** p(X_1 ⋯ X_n) S **end**	*Procedure definition*

… to concurrent constraints in Oz

S	::=	S_1 S_2	*Sequential composition*
	|	X **in** S	*Variable introduction*
	|	c	*Tell constraint*
	|	**if** C_1 [] C_2 [] ⋯ [] C_n **else** S **end**	*Conditional*
	|	{X X_1 ⋯ X_n}	*Procedure call*
	|	**thread** S **end**	*Thread introduction*
C	::=	X_1 ⋯ X_n **in** c **then** S	*Ask clause*
c	::=	X=**proc** {$ X_1 ⋯ X_n} S **end** | …	*Constraints*

- Oz made two major changes to the concurrent constraint model
  - Higher-order procedures (procedures are constraints)
  - Concurrency is explicit (threads) instead of implicit

Rich general language…

- Powerful foundation
  - First-class values (functions, procedures, classes, objects, components, modules, spaces)
  - Compositional factored syntax
  - Lightweight threads and dataflow variables
  - Deep embedding for distributed computing
- Multiple paradigms
  - Functional, functional dataflow, lazy functional dataflow, actor dataflow
  - Data abstraction, polymorphism, inheritance
  - Dataflow concurrency, multiagent programming, shared state concurrency
  - Relational programming, constraint programming, programmable search engines

...lean kernel language

Functional			
S	::=	**skip**	*empty statement*
	|	$S_1\ S_2$	*sequential composition*
	|	**local** X **in** S **end**	*variable introduction*
	|	X_1=X_2	*variable-variable equality*
	|	X=V	*variable-value equality*
	|	**if** X **then** S_1 **else** S_2 **end**	*conditional*
	|	$\{X\ Y_1 \cdots\ Y_n\}$	*procedure call*
	|	**case** X **of** *Record* **then** S_1 **else** S_2 **end**	*pattern matching*
Functional dataflow			
	|	**thread** S **end**	*thread introduction*
Lazy functional dataflow			
	|	`{WaitNeeded` X`}`	*by-need synchronization*
Relational and constraint			
	|	*Space*	*computation spaces*
Exceptions			
	|	**try** S_1 **catch** X **else** S_2 **end**	*exception scope introduction*
	|	**raise** X **end**	*raise exception*
Actor dataflow			
	|	`{NewPort` $X\ Y$`}`	*port introduction*
	|	`{Send` $X\ Y$`}`	*port send*
Mutable state			
	|	`{NewCell` $X\ Y$`}`	*cell introduction*
	|	`{Exchange` $X\ Y\ Z$`}`	*cell exchange*

X, Y, Z	::=	*(identifiers)*
V	::=	*Number* | *Procedure* | *Record* | **true** | **false**
Number	::=	*Int* | *Float*
Procedure	::=	**proc** $\{\$\ X_1 \cdots\ X_n\}$ S **end**
Record	::=	$f(l_1{:}X_1 \cdots\ l_n{:}X_n)$
Space	::=	*(space operations are listed in Figure 16)*

- We start with a simple kernel language that underlies our first paradigm, functional programming
    - We then add concepts one by one to give the other paradigms
    - Vastly different paradigms have quite similar kernel languages
    - The final kernel language is much simpler than the sum of all paradigms
- It is possible to program in each paradigm separately and to combine them where necessary

Salient Features of Oz

Functional programming

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

fun {Map L F}
   case L of X|M then
      {F X}|{Map M F}
   [] nil then nil end
end
```

- Generate a list of integers and map them:

```
local L1 L2 in
   L1={Ints 1 10}
   L2={Map L1 fun {$ X} X*X end}
   {Browse L2} % Display [1 4 9 16 … 81]
end
```

- Because of dataflow variables, both functions are tail-recursive:

```
proc {Map L F R}
   …
   local R1 in R={F X}|R1
      R1={Map M F}
   end
   …
end
```

Create list R with unbound tail R1 that is bound inside the recursive call to Map

From functional to functional dataflow

- A stream is a list that ends in an unbound variable
  - S=a|b|c|d|S2
  - A stream can be extended with new elements as necessary
    - The stream can be closed by binding the end to nil

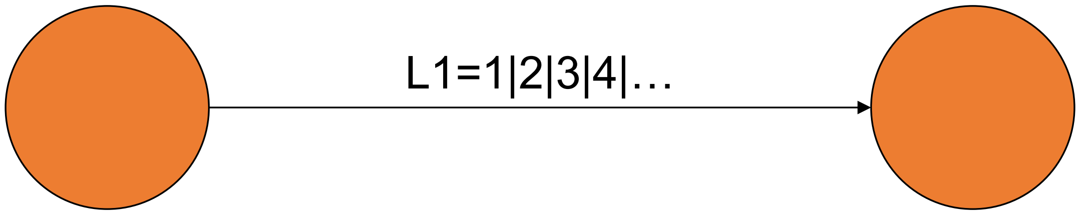
- A stream can be used as a communication channel between two threads
  - The first thread adds elements to the stream
  - The second thread reads the stream

Functional dataflow

- We run the program concurrently without changing the definitions:

```
local L1 L2 in
    {Browse L1}
    {Browse L2}
    thread L1={Ints 1 10} end
    thread L2={Map L1 fun {$ X} X*X end} end
end
```

- This turns a batch computation into an incremental (streaming) computation
  - In general, any functional program can be made more incremental by adding threads anywhere, without changing the final results

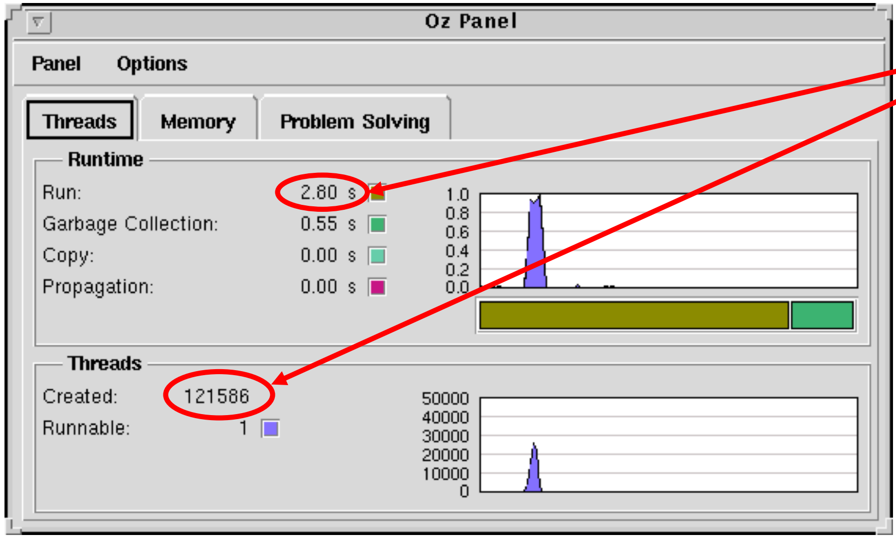


thread L1={Ints 1 10} **end**

thread L2={Map L1 **fun** {$ X} X*X **end**} **end**

(functional dataflow demo)

Ultralightweight threads



Execution time: 2.80 seconds
Number of threads: 121586 (in 1996)

```
fun {Fib X}
   if X=<2 then 1
   else thread {Fib X-1} end + {Fib X-2} end
end

{Browse {Fib 26}}
```

- Fibonacci with two recursive calls; first call creates a thread, dataflow synchronization correctly combines the results
- In the functional paradigm of Oz, any expression can be executed in its own thread without changing the result

HOPL IV

Ports and multi-agent programming

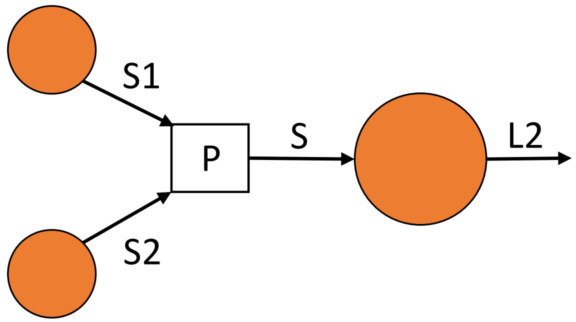
- We want to do multi-agent programming
  - It cannot be done in functional dataflow, because of nondeterminism!
- We add one new concept to do multi-agent programming
  - A named communication stream that we call a port

```
% Connect name P to stream S
declare P S in
{NewPort P S}

% Read the stream S
thread L2={Map S fun … end} end
```

```
% Send S1 to the port
thread S1={Ints 1 1000} end
for X in S1 do {Send P X} end

% Send S2 to the port
thread S2={Ints 1001 2000} end
for X in S2 do {Send P X} end
```



- Multi-agent programming = functional dataflow + port

Actors

```
fun {NewActive Class Init}
   S
   Port={NewPort S}
   Object={New Class Init}
in
   thread
      for M in S do {Object M} end
   end
   Port
end
```

- We combine multi-agent programming with object-oriented programming
- This gives a new abstraction, an active object
  - Concurrency behavior of an agent
  - Computation behavior of an object

HOPL IV

Software transaction manager using strict two-phase locking with ordered timestamps for deadlock avoidance

```
declare
class TMClass
   attr timestamp tm
   meth init(TM) timestamp:=0 tm:=TM end

   meth Unlockall(T RestoreFlag)
      for save(cell:C state:S) in {Dictionary.items T.save} do
         (C.owner):=unit
         if RestoreFlag then (C.state):=S end
         if {Not {C.queue.isEmpty}} then
         Sync2#T2={C.queue.dequeue} in
            (T2.state):=running
            (C.owner):=T2 Sync2=ok
         end
      end
   end

   meth Trans(P ?R TS) /* See next figure */ end
   meth getlock(T C ?Sync) /* See next figure */ end

   meth newtrans(P ?R)
      timestamp:=@timestamp+1 {self Trans(P R @timestamp)}
   end
   meth savestate(T C ?Sync)
      if {Not {Dictionary.member T.save C.name}} then
         (T.save).(C.name):=save(cell:C state:@(C.state))
      end Sync=ok
   end
   meth commit(T) {self Unlockall(T false)} end
   meth abort(T) {self Unlockall(T true)} end
end

proc {NewTrans ?Trans ?NewCellT}
TM={NewActive TMClass init(TM)} in
   fun {Trans P ?B} R in
      {TM newtrans(P R)}
      case R of abort then B=abort unit
      [] abort(Exc) then B=abort raise Exc end
      [] commit(Res) then B=commit Res end
   end
   fun {NewCellT X}
      cell(name:{NewName} owner:{NewCell unit}
           queue:{NewPrioQueue} state:{NewCell X})
   end
end
```

```
meth Trans(P ?R TS)
   Halt={NewName}
   T=trans(stamp:TS save:{NewDictionary} body:P
           state:{NewCell running} result:R)
   proc {ExcT C X Y} S1 S2 in
      {@tm getlock(T C S1)}
      if S1==halt then raise Halt end end
      {@tm savestate(T C S2)} {Wait S2}
      {Exchange C.state X Y}
   end
   proc {AccT C ?X} {ExcT C X X} end
   proc {AssT C X} {ExcT C _ X} end
   proc {AboT} {@tm abort(T)} R=abort raise Halt end end
in
   thread try Res={T.body t(access:AccT assign:AssT
                           exchange:ExcT abort:AboT)}
          in {@tm commit(T)} R=commit(Res)
          catch E then
             if E\=Halt then {@tm abort(T)} R=abort(E) end
   end end
end
```

```
meth getlock(T C ?Sync)
   if @(T.state)==probation then
      {self Unlockall(T true)}
      {self Trans(T.body T.result T.stamp)} Sync=halt
   elseif @(C.owner)==unit then
      (C.owner):=T Sync=ok
   elseif T.stamp==@(C.owner).stamp then
      Sync=ok
   else /* T.stamp\=@(C.owner).stamp */ T2=@(C.owner) in
      {C.queue.enqueue Sync#T T.stamp}
      (T.state):=waiting_on(C)
      if T.stamp<T2.stamp then
         case @(T2.state) of waiting_on(C2) then
         Sync2#_={C2.queue.delete T2.stamp} in
            {self Unlockall(T2 true)}
            {self Trans(T2.body T2.result T2.stamp)}
            Sync2=halt
         [] running then
            (T2.state):=probation
         [] probation then skip end
      end
   end
end
```

This is the full implementation of the transaction manager; this gives an idea of what Oz programs look like

(transaction demo)

```
\switch +gumpparseroutputsimplified +gumpparserverbose

declare
parser LambdaParser from GumpParser.'class'
   meth error(VS) Scanner in
      GumpParser.'class', getScanner(?Scanner)
      {System.showInfo 'line '#{Scanner getLineNumber($)}#': '#VS}
   end

   token
      'define' ';' '=' ')'
      '.': leftAssoc(1)
      'APPLY': leftAssoc(2)
      'lambda': leftAssoc(2)
      '(': leftAssoc(2)
      'id': leftAssoc(2)
      'int': leftAssoc(2)

   syn program(?Definitions ?Terms)
      !Definitions={ Definition($) }*
      !Terms={ Term($) // ';' }+
   end
   syn Definition($)
      'define' 'id'(I) '=' Term(T) ';' => definition(I T)
   end
   syn Term($)
      'lambda' 'id'(I) '.' Term(T)      => lambda(I T)
   [] Term(T1) Term(T2) prec('APPLY')  => apply(T1 T2)
   [] '(' Term(T) ')'                  => T
   [] 'id'(I) Line(L)                  => id(I L)
   [] 'int'(I)                         => int(I)
   end
   syn Line($)
      skip => {GumpParser.'class', getScanner($) getLineNumber($)}
   end
end
```

Parser specification in Oz using the gump DSL parser generator tool creating an LL(1) parser

This is an example of one of the tools provided with the Mozart system

HOPL IV

mozart

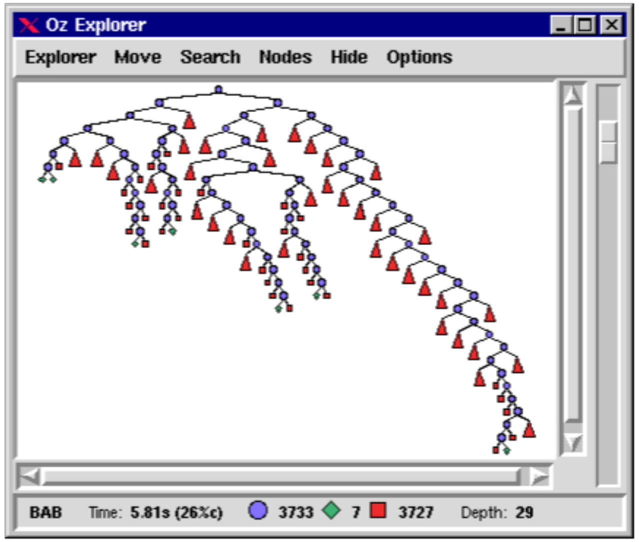
Constraint Programming

HOPL IV

Constraint programming

Christian Schulte

- Mozart 1.0 supports constraint programming
  - Constraint programming is a powerful approach to solve complex combinatoric problems; it is a kind of glue for operations research algorithms
  - Problems are specified as logical relations and solved with an incremental solver
- Mozart was the most advanced constraint system at its release in 1999
  - First-class computation spaces allows programming of custom solvers in Oz
  - Supports nested concurrent solvers (as in AKL)



Explorer tool for interactive exploration of a search tree

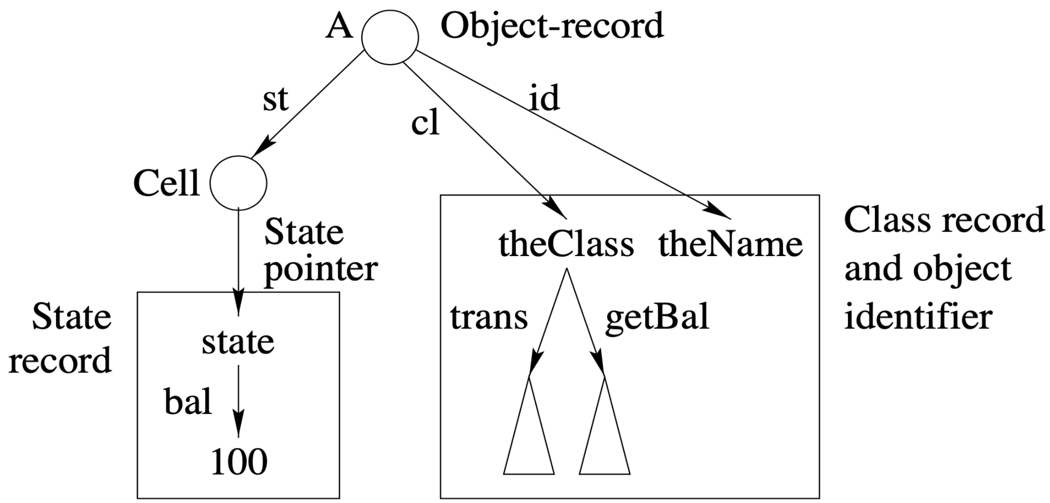
Distributed Computing

Distributed computing

- Work on Distributed Oz started in 1995 and was part of the Mozart 1.0 release in 1999
  - It is based on the clean separation between immutable data, dataflow variables, and mutable data in Oz
  - This separation facilitates deep embedding of distribution
    - Each language entity is implemented with its own distributed algorithm, which defines the network behavior and failure behavior while preserving kernel language semantics
- Distributed behavior of general language entities (like objects and classes) follows from the distributed behavior of their kernel language parts
- Application behavior is independent of distribution structure, except for operation timing and partial failure, for which extensions are provided

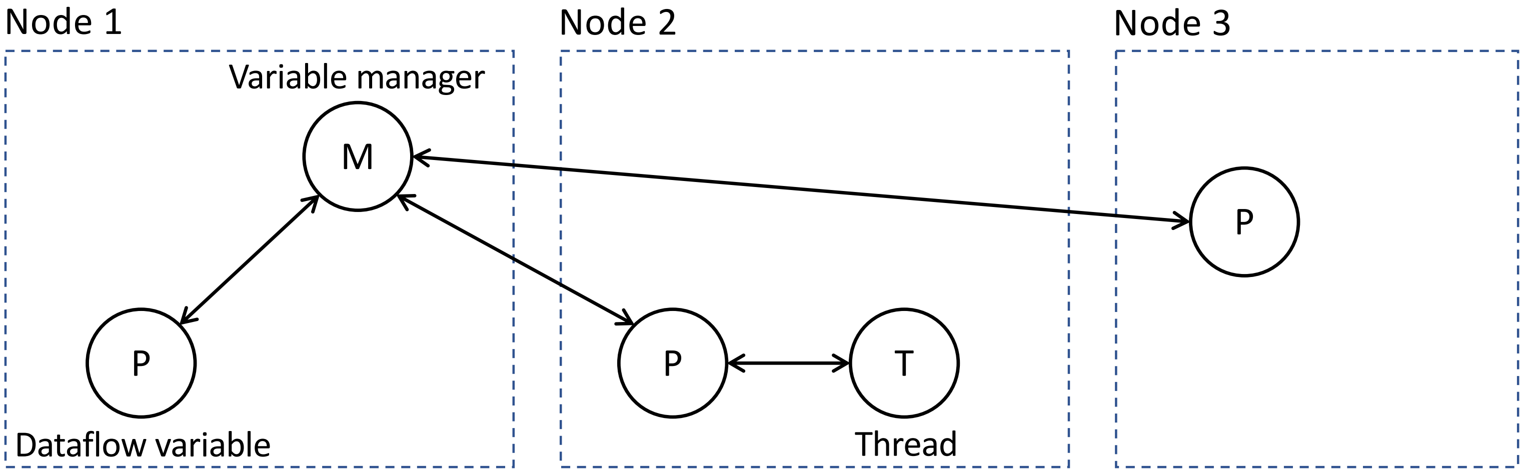
Distributed objects in Oz

```
class Account
   attr bal:0
   meth trans(Amt)
      bal:=@bal+Amt
   end
   meth getBal(B)
      B=@bal
   end
end
A={New Account trans(100)}
```



- The object's distributed behavior is defined by the distributed behavior of its parts
  - An object consists of an object-record which contains a class and the object's mutable state
  - Both object-record and class are immutable, so can be copied across the network
  - Mutable state (cell) obeys a consistency protocol

Distributed dataflow variables



- This figure shows how a dataflow variable is distributed over three compute nodes
- Generalizes remote futures, allows broadcast, maintains consistency of distributed store
- Consistency means that results are always the same, no matter in what order the dataflow variables are bound
  - This is a consequence of the semantics, which is distributed unification

HOPL IV

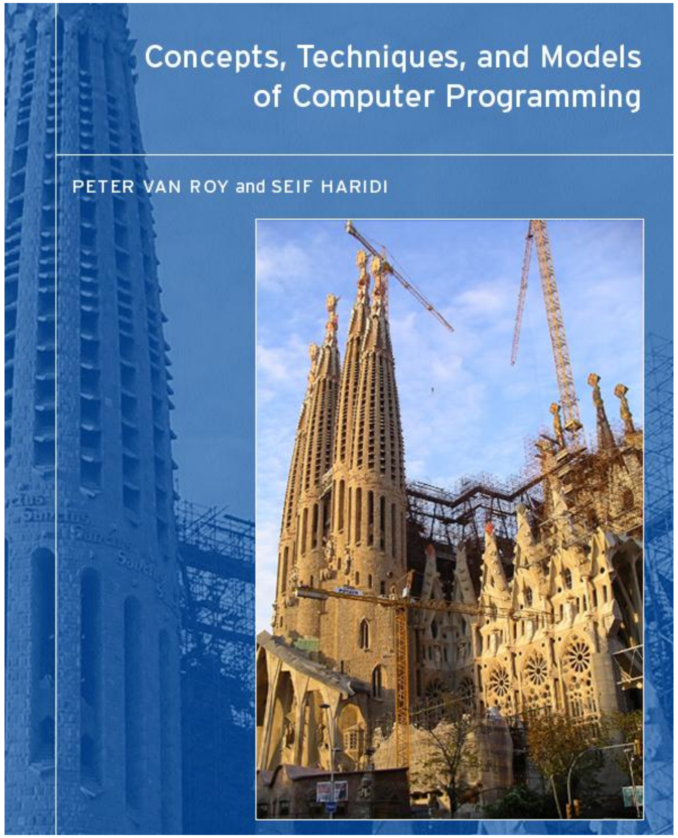
Important Oz Applications

Main Oz applications

	Concurrency Multi-agent	Distribution Fault tolerance	Constraints Symbolic	Higher-order
FriarTuck			X	
Strasheela			X	
NLP			X	X
iCities	X	X		
Beernet		X		
DIVE	X	X		
TransDraw		X	X	
Ωmega	X	X	X	
LogOz	X			X

- This table shows the largest applications that we know of in terms of how they use the strengths of the Oz language and Mozart system

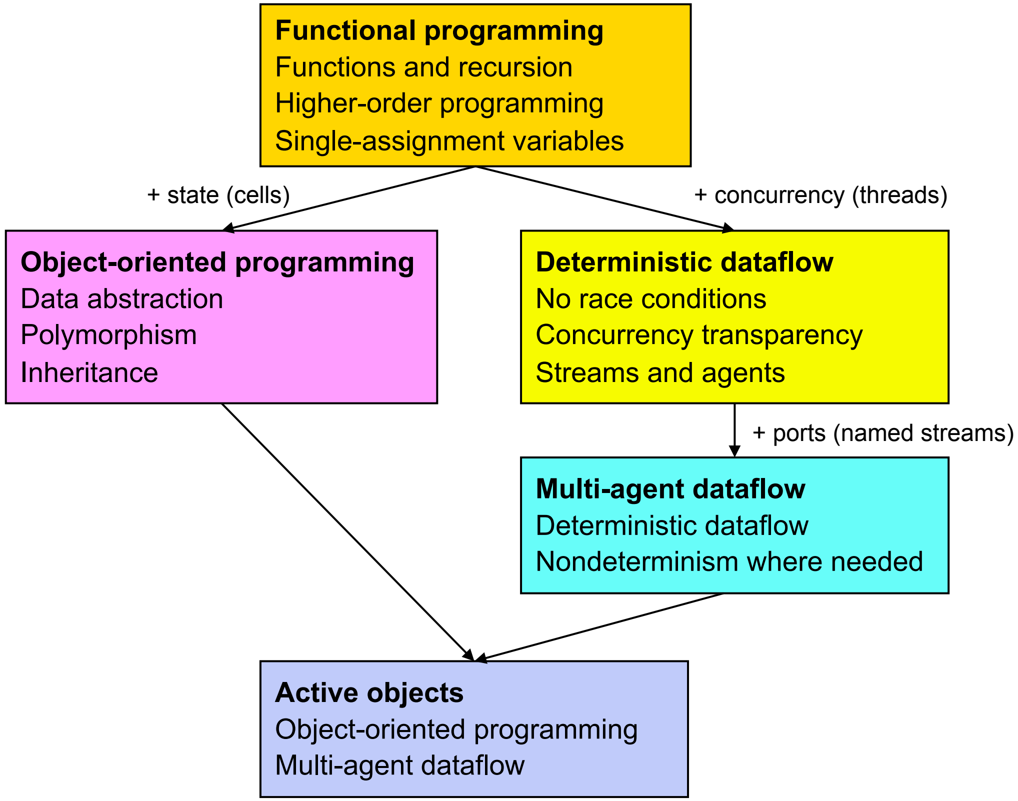
Programming textbook



- General programming textbook based on Oz published by MIT Press (2004), 929 pages
  - Chapters organized according to paradigms
  - Main theme is concurrency (1/3 of the book)

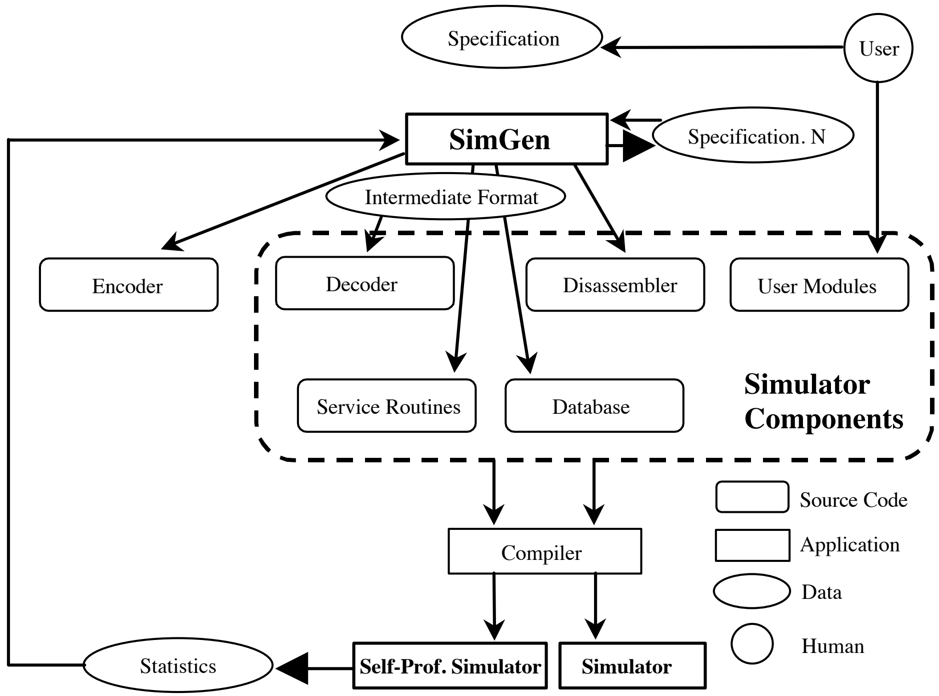
- "This book follows in the fine tradition of Abelson/ Sussman and Kamin's book on interpreters, but goes well beyond them, covering functional and Smalltalk-like languages as well as more advanced concepts in concurrent programming, distributed programming, and some of the finer points of C++ and Java."
  – Peter Norvig, Google Inc.

Oz in programming education



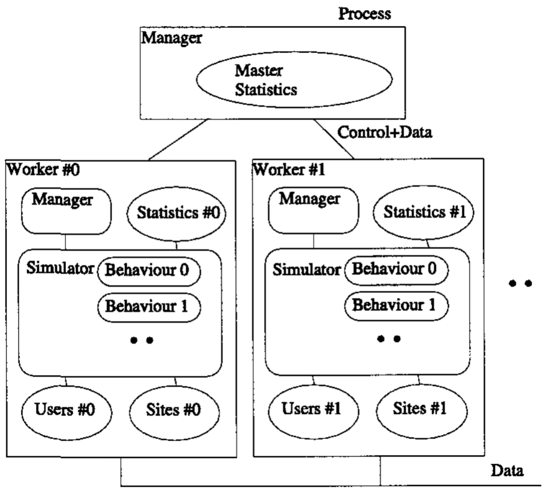
- The textbook and Oz were used in many university-level programming courses
  - At KTH, NUS, UCL in 2001-2003
  - At UCL up to present day
  - At ≥16 universities worldwide
- Concepts-based approach
  - Five paradigms in the second year
  - Formal semantics for all paradigms
- MOOCs on edX platform
  - Louv1.1x and Louv1.2x 2013-2018

SimICS architecture simulator

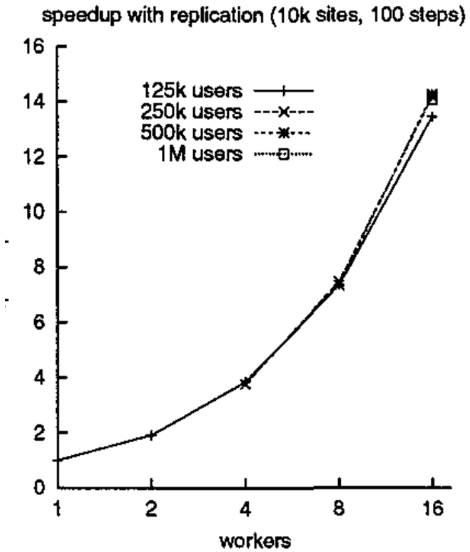


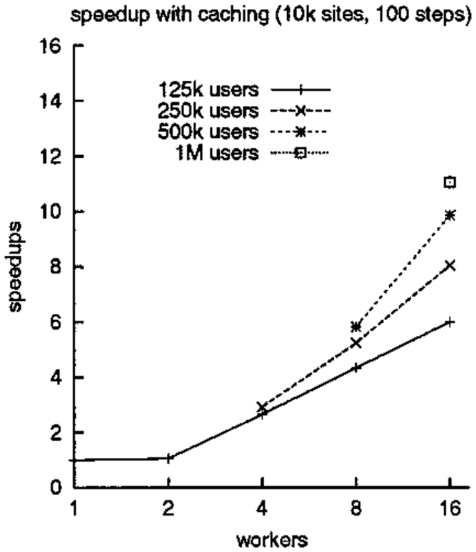
- SimICS was the first system-level simulator that could boot a non-modified commercial operating system at the instruction level
- The core of SimICS is SimGen, written in Oz on Mozart since 1997, which compiles an architecture specification into the components necessary for its operation
- SimGen is still being used today (by Intel) on Mozart 2 and it is probably one of the longest lasting projects using Mozart

iCities agent simulator



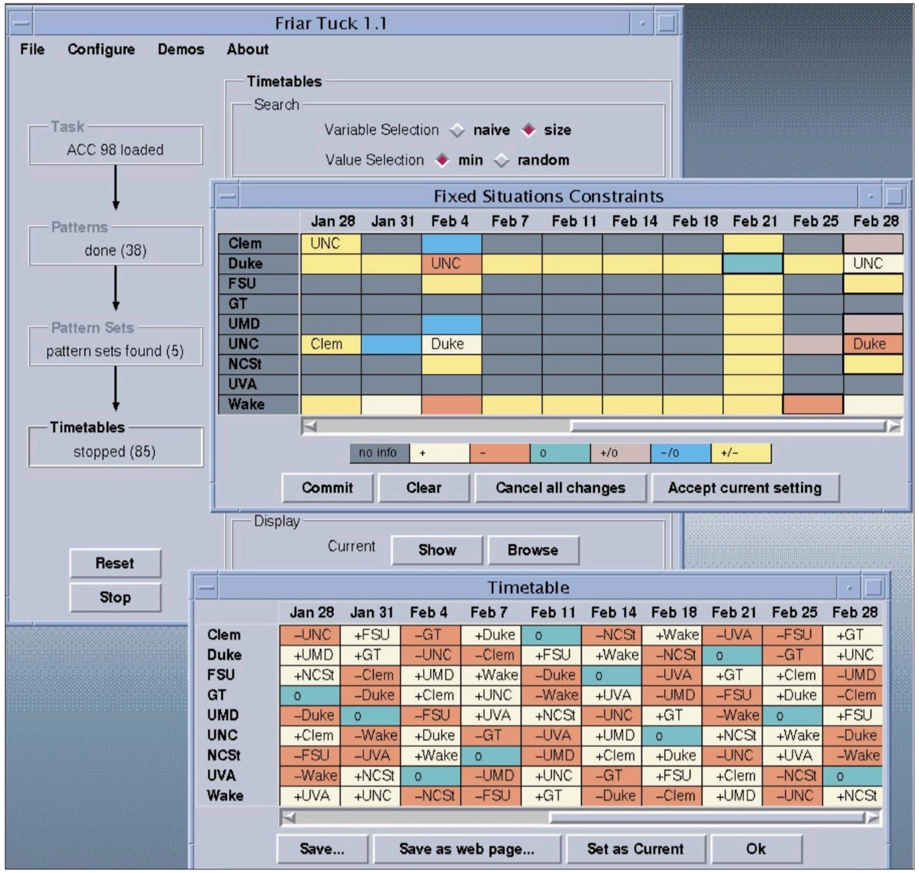
simulator architecture





- iCities was a European project (2001-2003) to study emergence in on-line communities
- iCities used Oz on Mozart to implement a parallel agent simulation platform for clusters
  - In 2002, the system ran on 16 AMD Athlon 1900+ computers with 100 Mbit Ethernet under Linux, achieved speedup of 11 to 14

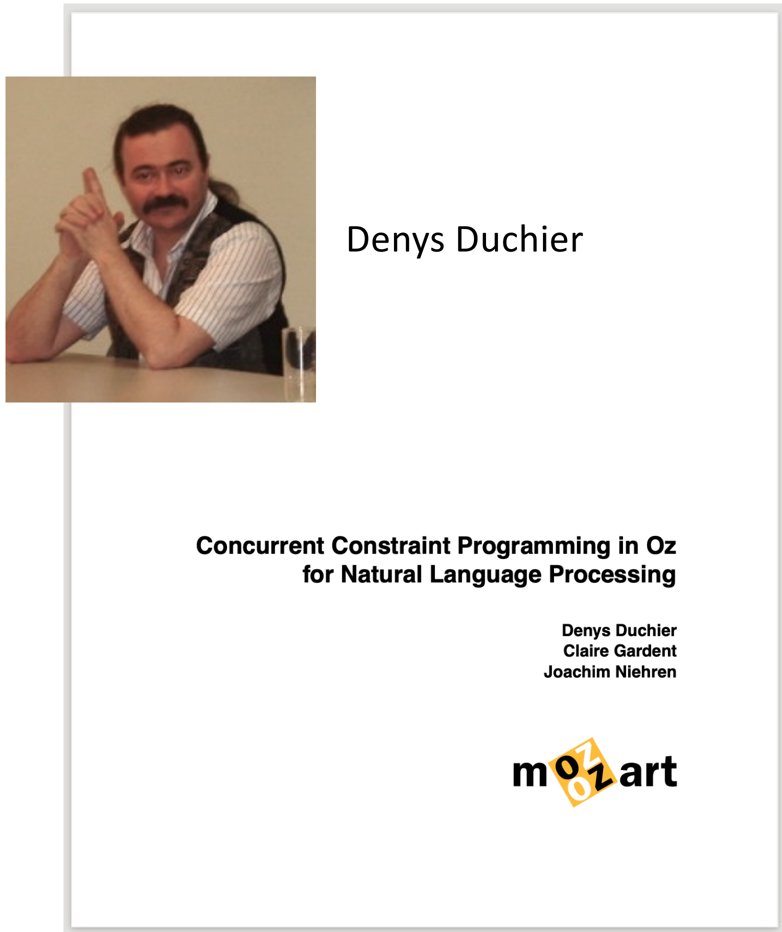
FriarTuck tournament scheduler



- FriarTuck is a round-robin sports tournament scheduling application based on constraint programming, which was initially implemented in Oz on Mozart in 1999
- This software scheduled several sports tournaments in England and the USA in 1999 and 2000
- The company still exists today and is called Workforce Optimizer

NLP in Oz

- 323-page book for computational linguistics in Oz published in 1999
- Some chapters:
  - **A Chart Parser for Context Free Grammars**
  - **Chart Parsing for Unification Grammars**
  - **Active Chart Parsing**
  - **Constraints in Semantic Underspecification**
  - **Word-Order and Dependency Structure**
  - **Constraint-Based Dependency Parsing**
  - **Concurrent Chart Parsing**



Conclusions

Successes

- The initial goal of multiparadigm programming was largely achieved
- The Oz language successfully integrates many paradigms and the Mozart system is a high-quality efficient implementation
- The Oz approach to concurrency successfully simplifies writing concurrent applications
- The programming textbook successfully presents programming as a unified discipline integrating many paradigms
- The book and Mozart system were successfully used in education
- The Mozart system was successfully used to build large applications
- The deep embedding approach of Oz for distribution is practical for cluster computing
- Mozart 1.4.0 is a high-quality system that successfully combines multiparadigm programming with constraints and deep embedding of distribution

Failures

- The Oz project failed in creating a self-sustaining community
  - We failed to navigate the transition between funded research and open-source development
  - Most of the key developers left the project and were not replaced
  - The open-source culture was in its infancy when Mozart was first released
  - We failed to navigate timely the transition to 64-bit architectures due to lack of resources
  - Funding support for programming language research in Europe diminished
- The Oz syntax was unusual and the object syntax was not polished
  - This created a threshold for new users to join the community
  - We failed to recognize this and modernize the syntax

Legacy

- Oz was a pioneer in many ways
- In programming education, Oz was a successful foundation for a concepts-based approach
- Oz pioneered several important programming concepts
  - Lightweight threads (with shared data)
  - Dataflow variables, as a tool for fine-grained asynchronous programming
  - The distinction between mutable and immutable data types
  - Functional dataflow, which is now standard for streaming analytics
  - Programming with actors and futures
  - Techniques for efficient constraint solving including computation spaces
  - Deep embedding of distributed computing

The people

Iliès Alouini, Per Brand, Thorsten Brunklaus, Raphaël Collet, Benoit Daloze, Guillaume Derval, Sébastien Doeraene, Chris Double, Frej Drejhammar, Denys Duchier, Sameh El-Ansary, François Fonteyn, Nils Franzén, Anthony Gégo, Kevin Glynn, Donatien Grolaux, Gustavo Gutiérrez, Seif Haridi, Dragan Havelka, Martin Henz, Martin Homik, Yves Jaradin, Sverker Janson, Erik Klintskog, Leif Kornstaedt, Simon Lindblom, Benjamin Lorenz, Stewart Mackenzie, Guillaume Maudoux, Michael Mehl, Boriss Mejías, Valentin Mesaros, Johan Montelius, Martin Müller, Tobias Müller, Anna Neiderud, Joachim Niehren, Konstantin Popov, Mahmoud Rafea, Ralf Scheidhauer, Christian Schulte, Andreas Simon, Gert Smolka, Alfred Spiessens, Ralf Treinen, Peter Van Roy, Jörg Würtz, Andres Zarza Davila