Overview

- Short history of Oz
- Development methodology
- Textbook and education
- How to teach programming
- Concurrency for dummies
- Functional dataflow with ports
- Understanding data abstraction
- Understanding mutable state
- Distributed programming in Oz
- Logic programming in Oz
- Conclusions
Focus on programming

- Oz is a recent descendant of a series of logic-based languages that started with Prolog
- Oz design started in 1991 focused on logic programming but rapidly increased its ambition to the full range of programming concepts
- We tried hard to reinvent the world
- Oz made many contributions at different levels, but in this talk I will focus on programming: the contributions as seen by a programmer

Short history of Oz
“A man, a plan, a language: Oz”

- A joint German-Swedish initiative (1991)
  - DFKI in Saarbrücken (led by Gert Smolka)
  - SICS in Stockholm (led by Seif Haridi)
  - UCL in Louvain-la-Neuve (from 1996, led by Peter Van Roy)
  - Advanced Concurrent Constraint Languages: Application, Implementation, and Methodology
- An ambitious goal
  - “A coherent design of a language that combines the programming abstractions of various paradigms in a clean and simple way” (Oz tutorial, Nov. 1995)
Concurrent constraint model

- Oz semantics is based on the concurrent constraint model, which is a process calculus for concurrent logic programming (1990)
  - Defined by Vijay Saraswat based on work of Michael Maher and Kazunori Ueda
- A major advance because it defines synchronization in terms of logic!
- CC defines a constraint store $\sigma = c_1 \land c_2 \land \ldots \land c_n$ with primitive constraints $c_i$ and two basic operations, ask and tell
  - Tell $c$: $\sigma$ becomes $\sigma \land c$ if $\sigma \land c$ is satisfiable
  - Ask $c$: wait until $\sigma \models c$ or $\sigma \models \lnot c$

AKL (Andorra Kernel Language)

- AKL is a synthesis of search-based logic programming (Prolog) and concurrent logic programming (1991, Sverker Janson and Seif Haridi)
  - Solves a major unsolved problem of the Japanese Fifth Generation Computer System project (1982-1992)
- New idea: nested computation spaces
  - Computation space: constraint store with its agents, observes the parent spaces
  - Stability property: local termination of the computation space w.r.t. parent spaces
- AKL is compositional
  - Use search with concurrency
  - Use concurrency with search
- Oz is a direct descendant of AKL
  - AKL still has Prolog-like syntax; Oz breaks with Prolog through compositional syntax and higher-order semantics
Releases, releases

- Oz design proceeded intensely from 1991: Oz 1, Oz 2, Oz 3
  - Oz 1 explored many ideas; Oz 2 and Oz 3 made them practical
  - We released many software platforms, all available as open source
- Oz 1 released as DFKI Oz 1.0 (Jan. 1995)
- Oz 2 released as DFKI Oz 2.0 (Sep. 1996)
- Oz 3 the (mostly) final language design
  - First released as Mozart 1.0 (Jan. 1999)
  - 180000 lines of C/C++, 140000 lines of Oz
  - >10000 downloads in 1999-2000-2001
  - Final definition of declarative laziness (2003)
  - Final definition of fault tolerance primitives (2007)
  - Successive 32-bit releases until Mozart 1.4.0 (2008)
  - Clean 64-bit reimplementation as Mozart 2 (2012)

Development methodology
Development methodology

- Oz pioneered a unique development methodology that is key to its innovative design
  - Safety: At all times there exist both a solid implementation and a simple formal semantics
  - Liveness: Designers continuously introduce new language extensions as solutions to practical problems
  - Update criterium: An extension must either simplify the system or greatly increase expressive power
    - If this is the case, intense discussion to simplify the extension
    - Often, it simply vanishes
    - Otherwise, a core concept remains that is added to the language

Examples of the methodology

- Constraint programming
  - Originally with a special-purpose search combinator
  - Replaced by first-class computation spaces, simpler and much more expressive
- Component programming (modules and functors)
  - Can be completely expressed using higher-order, records, and names, no language extension required
- Lazy evaluation
  - Originally inside of unification, by making it a blocking operation
  - Replaced by WaitNeeded, seamless integration with functional dataflow
- Object system
  - Oz 1 had implicit thread creation, with state threading inside objects
  - Oz 2 made threads explicit, greatly simplifying the object system
- Single-assignment variables ("dataflow variables")
  - Single assignment is a weak form of mutable state that does not affect functional purity
  - All list functions trivially become tail-recursive, and furthermore usable as concurrent agents (!?)
In 1999, Seif Haridi and I realized we had something to say about programming. We started writing a textbook, even without knowing who would publish it (“if you build it, they will come”). We decided to write it at its natural size (929 pages), it took four exhausting years. MIT Press agreed in 2003 to publish the book in 2004 (editor Bob Prior). The textbook continues to sell well until the present day and has achieved some recognition. Widely considered as a worthy successor to “Structure and Interpretation of Computer Programs,” Mentioned by Peter Norvig on site “Teach Yourself Programming in Ten Years.”
Education

- We pioneered the "concepts-first" approach to teaching programming
  - Second and third-year courses starting in 2001 (KTH, NUS, UCL)
  - UCL taught it to all engineering students (not just CS majors) in a second-year course until 2018 (>5000 students in all)
  - Around 20 universities worldwide; two MOOCs in edX from 2013 to 2018

- Insights
  - Importance of the second year: students are mature enough but not yet too conservative to accept a broad and deep view of programming
  - Importance of a formal semantics: students of computing need to learn the formal semantics of programming languages, just like students of electronics learn Maxwell's equations
  - Concurrent programming can be taught in the first year ("Concurrency for Dummies", which is a functional dataflow paradigm)

Correcting MOOC programming exercises

- The MOOCs provided online programming exercises in Oz
  - We used the INGInious platform developed at UCL for hosting exercises and Mozart for executing them

- We have developed a tool, CorrectOz, that provides useful feedback to students when programs have errors
  - The tool contains an Oz parser and abstract machine emulator, so it can detect both syntactic and semantic errors
  - The parser implements an extended Oz syntax that includes ambiguity, so it can detect common errors such as missing keywords and misspellings
  - The tool detects around one hundred error markers, which we identified by analyzing actual student errors from the previous year’s course
  - The tool gives useful feedback for around 80% of incorrect programs
  - The tool is available as open-source software. For more information, see http://www.info.ucl.ac.be/~pvr/Magrofuoco_Nathan_33521000_&_Paquot_Arthur_60371000_2016.pdf
How to teach programming

Hundreds of programming languages are in use...

- Python
- C++
- Scala
- Java
- C#
- Erlang
- OCaml
- PHP
- Ruby
- Lua
- Haskell
So many, how can we teach them all?

- Key insight: languages are based on paradigms, and there are many fewer paradigms than languages
- We can teach many languages by teaching few paradigms!

But what is a programming paradigm anyway?

- A programming paradigm is an approach to programming a computer based on a coherent set of principles or a mathematical theory
  - Logic programming (Horn clause logic), functional programming (lambda calculus), object-oriented programming (polymorphism and inheritance), concurrent programming, transactional programming, constraint programming, multi-agent programming, distributed programming, fault tolerance, user interface, data scalability, …
- Any realistic program needs to use many paradigms
  - A program may use a relational (logic) database, do (functional) transformations, structure its (object-oriented) abstractions, …
  - So we need multiple paradigms and we need to combine them in the same program
How can we teach multiple paradigms?

- How can we teach multiple paradigms without teaching multiple languages (since most languages only support one, or sometimes two paradigms)?
- Each language has its own syntax, its own semantics, its own system, and its own quirks
  - Picking some languages, like Java, Scala, Erlang, Prolog, Scheme, and Haskell, and structuring a course around them would be terribly inefficient
- Our pragmatic solution is to use one language
- How can one language cover many paradigms?
  - It seems we have « escaped goblins only to be caught by wolves! », as Bilbo Baggins once said

How can one language cover many paradigms?

- Each paradigm is a different way of thinking
  - How can we combine different ways of thinking in one program?
- We do it using the concept of a kernel language
  - Each paradigm has a simple core language, its kernel language, containing its essential concepts
    - Every practical language, even very complicated, can be translated into a simple kernel language
  - Even very different paradigms have kernel languages that have much in common; often there is only one concept difference
- 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
  - This is how Oz was designed since its conception in 1991
Summary of the approach

- **Hundreds of languages** are used in practice: we cannot teach them all in one course or in one lifetime
  - Solution: **focus on paradigms**, since each language is based on a paradigm and there are many fewer paradigms than languages
- **One language per paradigm is too much** to teach in a course, since each language is already complicated by itself
  - Solution: **use one research language**, Oz, to express many paradigms
- **Realistic programs need to combine paradigms**, but how can we do it since each paradigm is a different way of thinking?
  - Solution: **define paradigms using kernel languages**, since different paradigms have kernel languages that are almost the same
  - Kernel languages allow us to define many paradigms by focusing on their differences, which is much more economical in time and effort

→ In 2020, supporting multiple paradigms is finally considered to be important for mainstream languages (Scala, Java 8, ...)

Five paradigms in one course

- **My second-year course** LINFO1104 (Spring 2020) covers five paradigms
  - With a practical system and a formal semantics
- **Five kernel languages:**
  - $K_{FP}$
  - $K_{OOP} = K_{FP} + \text{cell}$
  - $K_{FD} = K_{FP} + \text{thread}$
  - $K_{AD} = K_{OOP} + \text{port}$
  - $K_{AO} = K_{OOP} + K_{AD}$
- **Course slides are online**
- **It uses our textbook**

Many roads to concurrency

- Concurrency is hard
- The **wrong way** to do concurrent programming
  - Shared-state concurrency (a.k.a. monitors)
  - Monitors should be deprecated in all languages
- A **better way** to do concurrent programming
  - Message-passing concurrency (a.k.a. actors)
  - Actor model works well and is popular (e.g., Erlang)
- The **best ways** to do concurrent programming
  - Functional dataflow (for noninteractive programs)
  - Functional dataflow with ports (for interactive programs)
Concurrency is hard

- Concurrency introduces many difficulties such as nondeterminism, race conditions, reentrancy, deadlocks, livelocks, fairness, handling shared data, and concurrent algorithms to manage them can be complicated
  - Java’s `synchronized objects (monitors)` are tough to program with
  - Erlang’s and Scala’s `actors` are better, but they still have race conditions
  - Libraries can hide some of these problems, but they always peek through
- Adding distribution makes it even harder
- Adding partial failure makes it even much harder than that
- The Holy Grail: can we make concurrent programming as easy as sequential programming?
  - Yes, it can be done, if the paradigm is chosen wisely
  - We show functional dataflow, which is as pure as functional programming

Insights from Oz

27

From functional to dataflow

- Let’s define a very simple concurrent paradigm
- Start with pure functional programming (in Oz syntax!)

```
fun {Map L F} defines a function Map with arguments L and F
  fun {Map L F}
    case L of H|T then {F H}|{Map T F}
    [] nil then nil end
  end
end
fun {Sqr X} X*X end
L={Map [1 2 3] Sqr} % Computes [1 4 9]
```

- Now add dataflow variables and threads…

Insights from Oz

28
Dataflow variables and threads

- **Dataflow variable:**
  \[ B = A \times 10 \quad \% \text{Suspends execution until } A \text{ is bound to a value} \]
  \{Display B\}

- **Thread:**
  \textbf{thread} A=5 end \% Executes concurrently

- **Dataflow synchronization:**
  - Because the multiplication \( A \times 10 \) needs two values, the thread containing the multiplication will wait until another thread binds \( A \).
  - When \( A \) is bound, the thread continues and displays 50.

Functional dataflow paradigm

- An agent is a recursive list function running in its own thread.
- We define a pipeline with two concurrent agents using \texttt{Map} and \texttt{Prod}:
  \begin{verbatim}
  fun {Map L F}
      case L of H|T then {F H}|{Map T F}
      | nil then nil end
  end

  fun {Prod N} N|{Prod N+1} end
  \end{verbatim}
In a functional dataflow program, adding threads does not change the result but only the order in which computations are done.

- Threads can be added at will without introducing bugs (race conditions are not possible).
- The only effect of extra threads is to make the program more incremental (by removing sequential dependencies and deadlocks).
- Any tail-recursive function can be converted into a concurrent agent.

Functional dataflow was invented in 1974 by Gilles Kahn.

- It was largely forgotten until resuscitated in the early 2000s (by Oz?).
- Big-data analytics libraries often use functional dataflow.

It’s important to understand this paradigm.

- Especially since it leads to the best general concurrency paradigm...
Functional dataflow with ports

- Functional dataflow is a great paradigm, but it can’t always be used
  - It can’t be used for programs that interact with the real world
  - This is because real-world interactions introduce nondeterminism (the program must react on an external event, which can appear at any time)

- We fix this by extending functional dataflow with one simple concept, called a port, which is a simple communication channel
  - This leads to the best overall paradigm to write concurrent programs, functional dataflow with ports, which uses functional dataflow as much as possible and adds ports only where needed
  - To explain this paradigm, we give one example, namely a client/server, which can be written easily in functional dataflow with ports
  - For a more thorough explanation, we recommend you look at the course slides for LINFO1104

Client/server in functional dataflow with ports

- A client/server cannot be written in functional dataflow
  - But it can be written easily in functional dataflow with ports

- A client/server application consists of a set of clients all communicating to one server
  - When a client sends a message to the server, the server receives it, does a local computation, and then replies immediately. The total delay depends on the round-trip travel time and the local computation time.

- Client/server applications are ubiquitous on the Internet
Client/server

- A client/server cannot be written in functional dataflow
  - It is because to satisfy client liveness, the server must nondeterministically wait for each client request
- To implement the server, we add a many-to-one channel (called a port)
  - At the server:
    ```
    P={NewPort S} % Create a port with channel S
    proc {Server S} % Define the server
      case S of
        M|T then
          (...local computation for M...)
          {Server T}
        end
      end
    thread {Server S} % Start the server
    ```
  - At the client:
    ```
    {Send P M} % Message M appears on S
    ```

One port needed where the server receives client requests

Insights from Oz

35

Understanding data abstraction

Insights from Oz

36
Definition of a data abstraction

- A data abstraction is a part of a program that has an inside, an outside, and an interface in between.
- The inside is hidden from the outside.
  - All operations on the inside must pass through the interface, i.e., the data abstraction must use encapsulation.
- The interface is a set of operations that can be used according to certain rules.
  - Correct use of the rules guarantees that the results are correct.
- The encapsulation must be supported by the programming language.
  - We will see how the language can enforce the separation between inside and outside.

The two main kinds of data abstraction

- There are two fundamental kinds of data abstraction, namely objects and abstract data types (ADT).
  - An object groups together value and operations in a single entity.
  - An abstract data type keeps values and operations separate.
- Some real world examples:
  - A television set is an object: it can be used directly through its interface (on/off, channel selection, volume control).
  - Coin-operated vending machines are abstract data types: the coins and products are the values and the operations are the vending machines (feed coins to the machine, get a product).
- Real-world languages use both kinds:
  - And sometimes combine them in strange ways, e.g., a Java object is also an abstract data type.
Let’s compare ADTs and objects

- To fix our ideas, we focus on one simple data abstraction, namely a stack:
  - Create a new stack
  - Push an element on a stack
  - Pop an element off a stack
  - Check if a stack is empty
- We will implement this stack both ways, as an abstract data type and as an object

Abstract data type

- An ADT has a set of values and a set of operations on those values
  - Usually, the values and operations are stateless
- Here is our stack as an ADT:

```
local Wrap Unwrap in
  {NewWrapper Wrap Unwrap} % Create secure wrap/unwrap functions
  fun {NewStack} {Wrap nil} end
  fun {Push W X} {Wrap X|{Unwrap W}} end
  fun {Pop W X} S={Unwrap W} in X=S.1 {Wrap S.2} end
  fun {IsEmpty W} {Unwrap W}==nil end
end
S1={NewStack}
S2={Push {Push S1 a} b}
```

- Secure wrapping and unwrapping functions enforce encapsulation
Object

- An object has just one kind of entity, called an object, which encapsulates both the value and the operations
  - Usually, the entity is stateful (it has mutable state), but it can be stateless
- Here is our stack as an object:

  ```
  fun {NewStack} C={NewCell nil} % A cell is a mutable variable
  proc {Push X} C:=X|@C end
  proc {Pop X} S=@C in X=S.1 C:=S.2 end
  fun {IsEmpty} @C==nil end
  in stack(push:Push pop:Pop isEmpty:IsEmpty) end
  S1={NewStack}
  {S1.push a} {S1.push b}
  ```

- There are three methods, push, pop, and isEmpty, combined in one object

Understanding mutable state

Insights from Oz
Why is mutable state important?

- Do languages really need mutable state?
  - Pure functional programming has no mutable state and programs are (fairly) easy to prove correct
  - Imperative programming (like object-oriented programming) has mutable state and programs are much harder to prove correct
  - Why not just use pure functional programming?
- In fact, mutable state cannot be avoided
  - It is essential for programs that interact with the real world
  - Since it cannot be avoided, the language should add it in a good way
- We give just one example here
  - We explain why mutable state is needed for modularity
- For a deeper discussion of this topic, see my keynote talk:

Mutable state is needed for modularity

- We say that a program (or system) is modular with respect to a given part if that part can be changed without changing the rest of the program
  - “part” = function, procedure, component, module, class, library, package, file, …
- We will show by means of an example that the use of mutable state allows us to make a program modular
  - This is not possible in pure functional programming
An Oz fairy tale (1)

- Once upon a time there were three developers, P, U1, and U2
- P has developed module M that implements two functions F and G
- U1 and U2 are both happy users of module M

fun {MF}  % Module definition
fun {F ...}  
\langle Definition of F\rangle
end
fun {G ...}  
\langle Definition of G\rangle
end
in 'export'(f:F g:G)
end
M = {MF}  % Module instantiation

Insights from Oz

An Oz fairy tale (2)

- One day, developer U2 writes an application that runs slowly because it does too much computation
- U2 would like to extend M to count the number of times F is called by the application
- U2 asks P to make this extension, but to keep it modular so that no programs have to be changed to use it

fun {MF}  
fun {F ...}  
\langle Definition of F\rangle
end
fun {G ...}  
\langle Definition of G\rangle
end
in 'export'(f:F g:G)
end
M = {MF}
Oops!

- This is impossible in functional programming because F does not remember what happened in previous calls: it cannot count its calls
  - The only solution is to change the interface of F by adding two arguments, F\textsubscript{in} and F\textsubscript{out}:
    \[
    \text{fun} \{ F \ldots F_{\text{in}} F_{\text{out}} \} \ F_{\text{out}} = F_{\text{in}} + 1 \ldots \end
    \]
  - The rest of the program has to make sure that the F\textsubscript{out} of each call to F is passed as F\textsubscript{in} to the next call of F
  - This means that M’s interface has changed
  - All M’s users, even U1, have to change programs
    - U1 is especially unhappy, since it means a lot of extra work for nothing

Solution using a cell

- A cell is a mutable variable
  - In Oz, mutable variables must be declared explicitly as cells
  - C:=X updates a cell and @C reads it
  - Create a cell when MF is called and increment it inside F
    - Because of static scope, the cell is hidden from the rest of the program: it is only visible inside M
  - M’s interface is extended without changing existing calls
    - M.f stays the same
      - A new function M.c appears that can safely be ignored
  - P, U1, and U2 live happily ever after

\[
\text{fun} \{ \text{MF} \}
\]
\[
\begin{align*}
X &= \text{NewCell 0} \\
\text{fun} \{ F \ldots \} \\
\quad X &= @X + 1 \\
\end{align*}
\]
\[
\langle \text{Definition of F} \rangle
\]
\[
\text{end}
\]
\[
\text{fun} \{ G \ldots \} \\
\langle \text{Definition of G} \rangle
\]
\[
\text{end}
\]
\[
\text{fun} \{ \text{Count} \} @X \text{ end}
\]
\[
\text{in} \ '\text{export'}(f:F \ g:G \ c:Count) \\
\text{end}
\]
\[
M = \{ \text{MF} \}
\]
Because of its well-factored structure, we realized early on that Oz would be a good starting point for building a distributed programming system.

- Oz layered structure makes possible a network-transparent programming model
- We make a deep embedding: each language operation is distributed
- We implement distributed lexical scoping and open distribution

Distributed algorithms
- Distributed values are implemented with lazy & eager copying and unique names
- Dataflow variables (logic variables) are implemented using distributed unification
- Ports (channels) are implemented using asynchronous message sending
- Cells and objects require a global consistency protocol; we implemented “cached” mobile objects and stationary objects

Extensions
- Distributed garbage collection is implemented using local garbage collection extended with weighted reference counting and a lease mechanism for failures
- Failure detection is reflected for each language operation
Distributed unification and remote futures

- Mozart implements distributed rational tree unification
  - The algorithm was proved correct (TOPLAS 1999), the first proved distributed unification algorithm

- Today’s actor-oriented databases have a concept called "remote futures" which allow a node to send a query asynchronously to a remote node which returns a “future” immediately before the result is computed
  - The future is a placeholder for the result that can be passed as an argument and stored in data structures
  - Synchronization on the return value can be done on the future, and is only needed in the thread that uses the result, nowhere else
    - No synchronization is done when the future is returned to the calling node
    - The future can be sent to a third node, which is the only one to synchronize
Logic programming in Oz

- Given the pedigree of Oz, it is expected to support logic programming, and that is completely correct!
- Oz has several levels of support for logic programming:
  - The functional paradigm is a deterministic logic language, and so-called “dataflow variables” are actually full logic variables
  - For nondeterministic logic programming, Prolog-style search can be added exactly where it is needed and queried using an on-demand solver (first-class Prolog top level), and most well-written Prolog programs can be directly translated into Oz programs
  - For more complex problems, there is full support for constraint programming including custom programmable search, several constraint domains, and the Explorer tool for interactive solving

Deterministic append

- We can write append deterministically as follows:
  ```prolog
  proc {Append A B C}
    case A of
      nil then C=B
      [] X|As then Cs in C=X|Cs {Append As B Cs}
    end
  end
  ```
  Or else as follows:
  ```prolog
  proc {Append A B C}
    if B==C then A=nil
    else case C of
      X|Cs then As in A=X|As {Append As B Cs}
    end
  end
  ```
  Both correspond to the logical definition:
  \[ \forall a,b,c. \text{append}(a,b,c) \leftrightarrow (b=c \land a=\text{nil}) \lor (\exists x,c'.a'.a=x[a'=x' \land c=x'[c'=x' \land \text{append}(a',b',c'])] \]
Nondeterministic append

- We can write append nondeterministically as follows:
  
  ```prolog
  proc {Append A B C}
  choice
  A=nil B=C
  [] As Cs X in
  A=X|As C=X|Cs {Append As B Cs}
  end
  end
  ```

- Using the Solve operation this returns a lazy list of solutions:
  
  ```prolog
  L1={Solve proc {X} {Append [1 2] [3 4 5] X} end}
  L2={Solve proc {S} X#Y#Z=S in {Append X Y Z} end}
  ```

- L2 contains a list of triples X#Y#Z giving solutions in order of increasing length, exactly like a Prolog depth-first search.

Translating Prolog to Oz

- For pure Prolog:
  - Deterministic predicates use if and case but not choice
  - Nondeterministic predicates use choice

- For predicates with cut (green or blue cut):
  - If the guard is deterministic, use if or case
  - If the guard is nondeterministic, encapsulate it in SolveOne

- For bagof/3 and setof/3:
  - Bagof/3 corresponds to SolveAll in Oz, and its extension setof/3 sorts the result and removes duplicates
  - Bagof/3 with existential quantifier can be translated directly into Oz by keeping the quantifier local to the problem procedure
How it works in Oz

- Oz has a formal semantics as a concurrent constraint process calculus
  - It directly supports a constraint store with logical ask and tell operations: values and variables are defined in constraints
  - This gives a logical semantics to the functional dataflow

- This is extended with first-class computation spaces as an abstraction for the programmer
  - Spaces allow implementing deep guards, search, and all-solutions search
  - The Oz implementation supports nested computation spaces

Conclusions
Conclusions

- We explained the history of Oz and gave some examples of programming insights that we gained from the Oz project
  - There were many insights, some have become mainstream and others not, but it was not easy to disseminate them
  - The textbook was a key milestone and it is still considered an important book on programming
  - For more information, we recommend the HOPL IV article on Oz: http://www.info.ucl.ac.be/~pvr/hopl20main-p14-p-329dcad–final.pdf

- We still use Oz for teaching programming up to the present day
  - In my view, Oz is still the best language for introducing students to programming paradigms and formal semantics
  - For more information, we recommend the course slides for LINFO1104: http://www.info.ucl.ac.be/~pvr/linfo1104-handouts.pdf