A History of the Oz Multiparadigm Language

PETER VAN ROY, Université catholique de Louvain, Belgium
SEIF HARIDI, Royal Institute of Technology and RISE, Sweden
CHRISTIAN SCHULTE, Royal Institute of Technology, Sweden
GERT SMOLKA, Saarland University, Germany

Shepherd: Vijay Saraswat, Goldman Sachs R&D, USA

Oz is a programming language designed to support multiple programming paradigms in a clean factored way that is easy to program despite its broad coverage. It started in 1991 as a collaborative effort by the DFKI (Germany) and SICS (Sweden) and led to an influential system, Mozart, that was released in 1999 and widely used in the 2000s for practical applications and education. We give the history of Oz as it developed from its origins in logic programming, starting with Prolog, followed by concurrent logic programming and constraint logic programming, and leading to its two direct precursors, the concurrent constraint model and the Andorra Kernel Language (AKL). We give the lessons learned from the Oz effort including successes and failures and we explain the principles underlying the Oz design. Oz is defined through a kernel language, which is a formal model similar to a foundational calculus, but that is designed to be directly useful to the programmer. The kernel language is organized in a layered structure, which makes it straightforward to write programs that use different paradigms in different parts. Oz is a key enabler for the book *Concepts, Techniques, and Models of Computer Programming* (MIT Press, 2004). Based on the book and the implementation, Oz has been used successfully in university-level programming courses starting from 2001 to the present day.

CCS Concepts: • **Social and professional topics** → **Computing education**; • **Theory of computation** → **Constraint and logic programming**; *Process calculi*; *Operational semantics*; • **Software and its engineering** → **Distributed programming languages**; **Object oriented languages**; **Functional languages**; **Constraint and logic languages**; **Data flow languages**; **Multiparadigm languages**; **Semantics**; *Graphical user interface languages*; • **Human-centered computing** → *User interface programming*; *User interface toolkits*.

Additional Key Words and Phrases: Computer programming, multiparadigm programming, concurrent programming, dataflow, functional programming, logic programming, lazy evaluation, programming education, distributed programming

ACM Reference Format:
Peter Van Roy, Seif Haridi, Christian Schulte, and Gert Smolka. 2020. A History of the Oz Multiparadigm Language. *Proc. ACM Program. Lang.* 4, HOPL, Article 83 (June 2020), 56 pages. https://doi.org/10.1145/3386333

Authors' addresses: Peter Van Roy, Université catholique de Louvain, Louvain-la-Neuve, 1348, Belgium, peter.vanroy@uclouvain.be; Seif Haridi, Royal Institute of Technology and RISE, Stockholm, Sweden, haridi@kth.se; Christian Schulte, Royal Institute of Technology, Stockholm, Sweden, cschulte@kth.se; Gert Smolka, Saarland University, Saarbrücken, Germany, smolka@ps.uni-saarland.de.

2475-1421/2020/6-ART83
https://doi.org/10.1145/3386333

Contents

1 INTRODUCTION

The Oz programming language is designed to support different programming paradigms with equal ease. This vision was conceived in the early 1990s, when a research community was created around programming languages based on concurrent constraint programming, first in Swedish and German national projects, and then combined in the ACCLAIM European research project which took place from 1992 to 1995. From the beginning, this community was convinced that all large programs need to use more than one paradigm, and that concurrent constraint programming was a suitable foundation to build a multiparadigm system. This initially resulted in three languages and their implementations, namely AKL, Oz 1, and LIFE. AKL was developed by Sverker Janson and Seif Haridi at KTH (Royal Institute of Technology in Stockholm) and SICS (Swedish Institute of Computer Science), Oz 1 was developed by Gert Smolka at Saarland University and DFKI (German Research Center for Artificial Intelligence), and LIFE was developed by Hassan Aït-Kaci at Digital PRL (Paris Research Laboratory). In 1995, the Swedish and German groups realized that they needed to combine their efforts and that Oz 1 was a good starting point [NA Haridi 1994]. We quote from the Oz 2.0 tutorial (November 1996) [NA Haridi 1996]:

> When we start writing programs in any existing language, we quickly find ourselves confined by the concepts of the underlying paradigm. Oz tries to attack this problem by a coherent design of a language that combines the programming abstractions of various paradigms in a clean and simple way.

This led to a continued development of the Oz language, first leading to Oz 2 and its implementation DFKI Oz 2.0 (released in 1996), and then reaching nearly its final shape with Oz 3 and its implementation, the Mozart Programming System (released in January 1999). This system and its successors are available as open-source software up to the present day [NA Mozart Consortium 2018]. This article tells the story of Oz first during the period that it was conceived (1991-1999), then during the period when it was widely used (1999-2009), and subsequently, when it was used mostly for computer science education.

The Oz design effort started in 1991 and initially focused on designing a language for knowledge-based multi-agent applications, starting from logic and concurrent programming. But this initial goal rapidly became much more ambitious, to support as many programming paradigms as possible in an equitable way, with a uniform syntax, simple semantics, and efficient implementation. In this context, we defined a programming 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 include functional programming, based on the λ calculus, logic programming, based on Horn clause logic, and object-oriented programming, based on a set of principles including data abstraction, polymorphism, and inheritance.

Following this definition, we understood early on that large programs often use many paradigms together. For example, the program may have a database that uses a relational (logical) structure, it may have to do (functional) transformations, it can use object-oriented principles to structure its

data abstractions, and it can use a concurrent paradigm (such as dataflow) to connect its independent parts. Many present-day programming systems support this to some degree, either inside a language, with libraries, or by combining several languages.

The Oz language supports the combination of paradigms within a single language by using a kernel language approach. Starting with a small language containing just a few concepts, we first explore how to program and reason in this language. We then add concepts one by one to overcome limitations in expressiveness. For example, object-oriented programming, as usually done in mainstream languages, has a kernel language with just one additional concept with respect to functional programming, namely mutable state. Functional dataflow programming has just two concepts in addition to functional programming, namely dataflow variables and threads. This design process resulted in a uniform framework that covers all major programming paradigms that were known at that time. A large number of practical paradigms are supported, each represented as a subset of Oz, and each defined with its own kernel language [NA Van Roy 2008]. Note that most of these kernel languages have very much in common, even for programming paradigms that superficially seem very different. In this way, the Oz language cleanly and simply supports programs that use multiple paradigms. It is very natural to write programs with multiple paradigms in Oz, as is shown with many substantive examples in Section 5.2.

Structure of the Paper. We divide this paper into four parts.

- The first part (Section 2) gives the history of Oz. Section 2.1 situates Oz with respect to its roots in logic programming and concurrent constraint programming. Section 2.2 gives a detailed account of the Oz project, from its inception in 1991 up to the release of the Mozart 1.0 system in 1999. Section 2.3 presents the period when Mozart was widely used in projects and applications, from 1999 to 2009. Section 2.4 explains what happened after that, from 2009 to the present, when Mozart was mostly used for education (courses and MOOCs). Finally, Section 2.5 gives an assessment of the Oz project including its successes and failures.
- The second part (Section 3) explains four major principles underlying the Oz project: making operations explicit, evolutionary development based on efficient implementation and simple formalization, supporting multiparadigm programming, and combining dynamic and static typing. These four principles are important to characterize the originality of the Oz project.
- The third part (Section 4) gives the impact of Oz. Section 4.1 presents the textbook published by MIT Press, which contributed to the visibility and use of Mozart. Section 4.2 summarizes the use of Oz in computer science education (including university courses and a MOOC). Section 4.3 summarizes the main projects and applications that used Oz.
- The fourth part (Section 5) gives an overview of the main technical ideas of Oz, so that curious readers can see the final results of all this work. Section 5.1 defines the Oz kernel language with its concepts and operations, and explains how it organizes the paradigms. Section 5.2 gives some highlights of multiparadigm programming by presenting examples of various paradigms and how they can be used together. We show examples of functional dataflow, lazy functional dataflow, actor dataflow, mutable state, and relational programming. Section 5.3 explains how we extended Oz to supported distributed programming. The layered design of Oz lends itself to an efficient deep embedding of distribution in the language.

2 HISTORY

The history of Oz can be divided into three periods: the gestation period (1991-1999), when Oz was created and developed into a full-featured robust language and implementation (the Mozart system, released in January 1999), the Mozart period (1999-2009), when Oz was widely downloaded (as the Mozart system) and used and supported by key developers and active user groups, and the

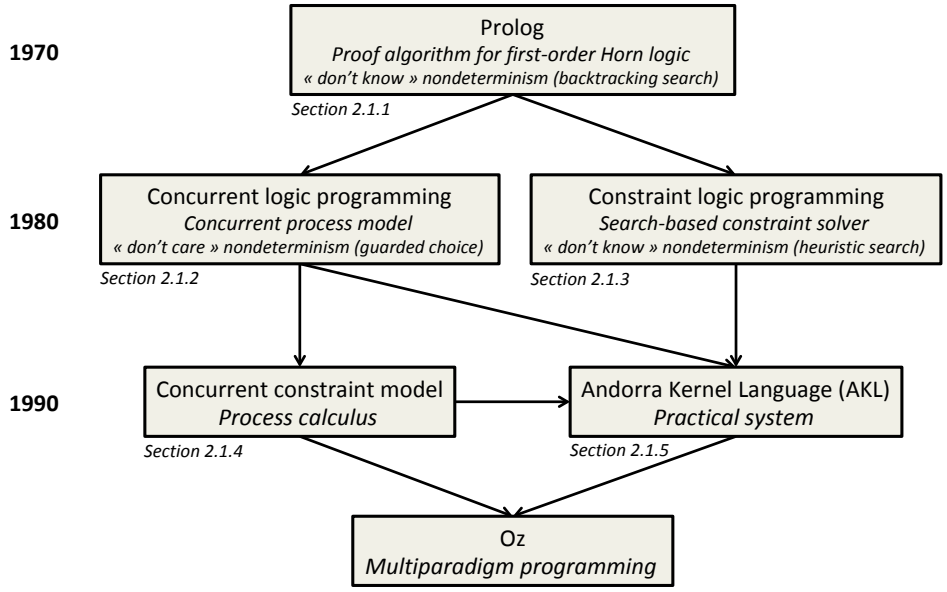

Fig. 1. Precursors of Oz

education period (2009-present), when Oz use diminished and it was mainly used and supported as a platform for education.

2.1 Precursors

Oz is a recent descendant of a series of logic-based languages that originated with Prolog (see Figure 1). We give a brief overview of the evolutionary steps starting from Prolog that resulted in Oz. Around 1980, research based on Prolog bifurcated into two separate directions of language design, namely concurrent logic programming (Section 2.1.2) and constraint logic programming (Section 2.1.3). The former introduced the process model of logic programming and was used for operating system design and parallel processing. The latter generalized Prolog's execution to use general constraints and constraint solvers, and was used to solve complex combinatoric problems. These two directions evolved separately until 1990. At that time, the concurrent constraint model was introduced as a clean formalization of concurrent logic programming (Section 2.1.4), and the Andorra Kernel Language (AKL) was designed to combine the concepts of the two areas (Section 2.1.5). Oz is a direct descendant of both AKL and concurrent constraint programming, and was from the beginning designed to be multiparadigm.

2.1.1 Prolog. Since the construction of the first automatic computers, one of the ultimate goals of computer science was to build a system that would allow programming in logic, i.e., to use deduction as computation. Efforts in this direction led to the first practical logic programming language, Prolog, which was conceived and built in the early 1970's by Alain Colmerauer, Robert Kowalski, and Philippe Roussel [Colmerauer and Roussel 1996]. This system uses a simple depth-first proof algorithm for first-order logic, with programs written as conjunctions of Horn clauses. When attempting to prove a predicate, the search algorithm will successively try all Horn clauses. If a proof is not possible with a given clause, the system will try the next clause. This selection process is called *don't know* nondeterminism. The initial system and its followups were interpreters.

The first Prolog compiler was built by David H.D. Warren in 1977 [Warren 1977]. By 1983, Warren had developed the New Prolog Engine, which was soon called the WAM (Warren Abstract Machine) [Warren 1983]. It became the de facto standard implementation technique and led to a proliferation of sequential WAM-based Prolog systems [Van Roy 1994]. These systems are widely used up to the present day [NA Carlsson et al. 2001–2020].

The Prolog language itself, despite being a sweet spot in the trade-off between expressiveness and efficiency, only partly lived up to the initial promise of programming in logic. Prolog programs, despite being logical specifications (if written in an appropriate style) needed to be highly algorithmic in order to achieve efficiency [O'Keefe 1990; Sterling and Shapiro 1986]. Therefore, in addition to much work on Prolog implementation, there was much language design work focused on increasing the abstraction level of logic programming while maintaining high execution efficiency. This work went primarily in two directions, which are known as concurrent logic programming and constraint logic programming.

2.1.2 Concurrent Logic Programming. The invention of concurrent logic programming was a major step in logic programming research. This was started by systems that added coroutining to Prolog, such as IC-Prolog developed by Keith Clark at Imperial College. Finally, concurrent logic programming was introduced with Parlog and Concurrent Prolog, developed by Clark and Ehud Shapiro, respectively [Clark 1987; Shapiro 1983, 1987]. These systems replace the sequential depth-first search of Prolog by a process model, in which each predicate is defined by guarded clauses. Invoking a predicate creates a concurrent process. This process first chooses a clause, using a nondeterministic choice inspired by Dijkstra's guarded command language. This selection process is called *don't care* nondeterminism, since any single clause with a true guard may be chosen and committed to. When the clause is chosen, each predicate in the body of the clause is in turn invoked. Concurrent processes communicate and synchronize by means of shared logic variables, which act as communication channels. In Oz, this use of logic variables is very common, and we call logic variables used in this way *dataflow variables*.

This model was used as the basis for the Japanese Fifth Generation Computer Systems (FGCS) initiative, which took place from 1982 to 1992 [NA Wikipedia 2020b]. Groups in this initiative and elsewhere designed many different languages that were variations on this theme. In particular, the design of GHC (Guarded Horn Clauses) by Kazunori Ueda simplified concurrent logic programming considerably by introducing the notion of quiet guards [Ueda 1985]. In a *quiet guard*, a clause matching a goal will fire only if the guard is logically entailed by the constraint store (explained below in Section 2.1.4). The concept of quiet guard led to the concept of logical entailment as a way to define synchronization, a key innovation that was first formalized by Michael Maher [Maher 1987].

Originally, any predicate could be in a guard, which is called a *deep* guard, but eventually most of the work focused on systems with *flat* guards, which are limited to basic constraints or system-provided tests [Tick 1995]. Note that the flat/deep distinction is orthogonal to the notion of quietness: both flat and deep guards can use logical entailment as the synchronization condition. The difference is that in a deep guard, the set of constraints whose entailment is being checked is determined dynamically, by running the guard, instead of being statically known as in a flat guard. Flat guards are easy to implement efficiently and allow to express most of the programming techniques necessary for implementing systems based on networks of concurrent processes. The flat versions of Concurrent Prolog and GHC, called FCP and FGHC respectively, were developed into large systems [Institute for New Generation Computer Technology 1992; Shapiro 1989]. The KL1 (Kernel Language 1) language, derived from FGHC, was implemented in the high-performance

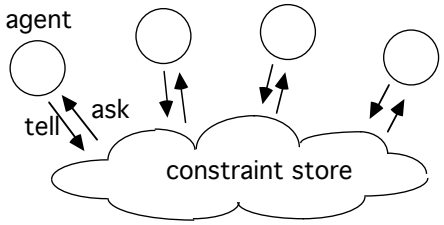

Fig. 2. The concurrent constraint model

KLIC system by Takashi Chikayama and his group. This system runs on sequential, parallel, and distributed machines [Fujise et al. 1994; Ueda and Chikayama 1990].

2.1.3 Constraint Logic Programming. Another major step in logic programming research was the invention of constraint logic programming as a generalization of Prolog. This started with the Prolog II language invented by Alain Colmerauer, which supported rational trees and disequalities (equations of the form $X \neq Y$) [Colmerauer 1982], followed by theoretical development by Joxan Jaffar and Jean-Louis Lassez [Jaffar and Lassez 1987; Marriott et al. 2006], the CLP($\mathcal{R}$) system and Pascal Van Hentenryck's work on finite domain constraints [Jaffar et al. 1992; Van Hentenryck 1994], and subsequently by a large body of work on constraints and their solvers. Constraint logic programming generalizes both the basic data operations and the control flow of Prolog. To explain this, we first summarize briefly the basic data operation of Prolog. This operation, called unification, is an algorithm to solve systems of equality constraints. Unification rewrites a conjunction of equality relations into a canonical solved form where each variable X is either bound to a term t or remains unbound.

A constraint logic programming system generalizes unification: equality relations are replaced by other relations (such as disequalities, inequalities, arithmetic relations on finite integer domains, or symbolic relations on symbolic data structures such as graphs or trees), and the unification algorithm is replaced by an algorithm to solve these relations. In this way, the constraint logic system can be used to solve complex combinatoric optimization problems. In principle this is a straightforward generalization, but in practice it is highly technical and complex, because useful relations and domains often do not have efficient solution algorithms. The efficiency of a constraint solver depends on the efficiency of the constraint solving algorithm and on the effectiveness of search heuristics that generalize Prolog's depth-first search. Choosing useful relations and domains and designing efficient solution algorithms often requires doing significant research and development. Modern constraint solving systems provide abilities that are competitive with and complement traditional areas such as operations research [NA Schulte et al. 2019].

2.1.4 Concurrent Constraint Programming. The semantic foundation of concurrent logic programming, and subsequently of Oz, is the concurrent constraint model originally developed by Vijay Saraswat [Saraswat and Rinard 1990; Saraswat 1993]. Conceptually, the model consists of a shared constraint store observed by concurrent agents, as illustrated in Figure 2. Agents do not communicate directly, but indirectly through the store. Given is a constraint store σ that consists of a conjunction of primitive logical constraints, $\sigma = c_1 \wedge c_2 \wedge \ldots \wedge c_n$. The model defines two basic operations on the constraint store, called ask and tell. Both operate on a constraint c that can also be a conjunction of primitive constraints. The tell operation c adds c to the store σ, which then

S	::=	$S_1\ S_2$	*concurrent composition*
	|	X **in** S	*variable introduction*
	|	c	*tell constraint*
	|	**if** C_1 `[]` $\cdots$ `[]` C_n **else** S **end**	*conditional*
	|	$\mathrm{p}(X_1 \cdots X_n)$	*procedure call*
C	::=	$X_1 \cdots X_n$ **in** c **then** S	*ask clause*
D	::=	**proc** $\mathrm{p}(X_1 \cdots X_n)\ S$ **end**	*procedure definition*

Fig. 3. The concurrent constraint language

becomes $\sigma \wedge c$, and it checks that the result is satisfiable. The ask operation waits until σ entails c (written as $\sigma \models c$) or the negation of c (written as $\sigma \models \neg c$). If σ entails the negation of c, we say that σ *disentails* c. Intuitively, the ask operation waits until we know enough to decide whether c is satisfied or whether it will never be satisfied. If neither can be decided, then the agent waits. This is how synchronization happens in concurrent constraint systems. In general, a constraint can be any formula in first-order logic. In practice, the constraints are chosen so that the ask and tell operations are computable, and preferably efficient.

This conceptual model inspired a series of process calculi for communicating concurrent agents. The work on AKL and Oz was based on a specific calculus that came from this model. In this calculus, the agents are programmed using the language defined in Figure 3, shown with a syntax similar to Oz. This language adds several operations in addition to the basic ask and tell, namely concurrent composition, variable introduction, and procedures. The ask is written as a conditional **if** ... **end** that contains ask clauses. When the constraint c in an ask clause is entailed, the statement S in the ask clause can execute. If more than one ask clause is entailed, a nondeterministic choice is made between them. The **else** statement can only be executed when all ask clauses are disentailed.

The concurrent constraint model can be used with any constraint system. Oz supports several constraint systems, but the most commonly used one is equality constraints on records, which generalizes Prolog terms. This is a logical way to represent standard data structures such as lists, tuples, and trees. To understand how this model works, we give a simple example of a concurrent constraint program that generates two lists of given lengths and appends them. We assume the system supports list notation and simple arithmetic constraints. The program defines two procedures:

```
proc append(L1,L2,L3)
   if L1=nil then L2=L3
   [] X M1 in L1=X|M1 then M3 in L3=X|M3 append(M1,L2,M3)
   end
end
proc list(N,L)
   if N=0 then L=nil
   [] N>0 then L1 N1 in N1=N-1 L=N|L1 list(N1,L1)
   end
end
```

In this definition the vertical bar, e.g., as used in `X|M1`, denotes infix list construction, analogous to cons in Lisp. Let us execute the following expression:

```
L1 L2 L3 in list(100,L1) list(100,L2) append(L1,L2,L3)
```

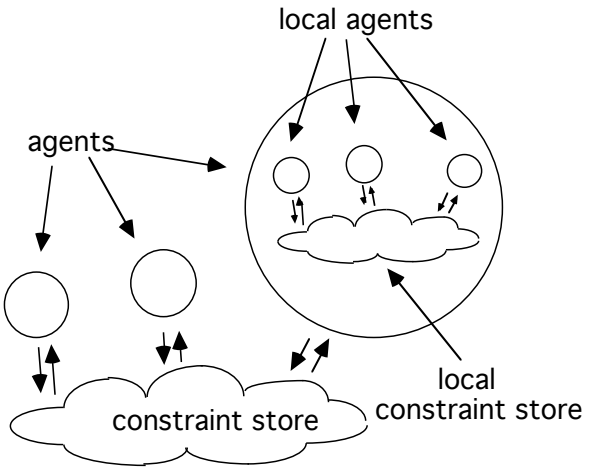

Fig. 4. The AKL computation model

This initially creates three agents using the concurrent composition. Note that the language does not contain sequential composition: all operations are done concurrently. The agents execute concurrently and communicate through the constraint store. Synchronization is done by entailment, i.e., the **`if`** constraint conditions. Execution order is constrained by the data dependencies between telling a constraint and asking a constraint. The `list` agents will add constraints, resulting in `L1` and `L2` eventually being equal to the lists `100|99|98|...|2|1|nil`. The `append` agent will wait until something is known about `L1`. As soon as part of `L1`'s structure is known in the store, for example `L1=X|M1`, then the second condition inside the **`if`** of the `append` is entailed, and the corresponding body is executed. This tells the constraint `L3=X|M3`, which incrementally constructs part of the output, and it also reduces to `append(M1,L2,M3)`, which is an agent that appends the remainder. Notice that the `append` agent can be made tail-recursive, since `L3=X|M3` can be added to the store independently of the recursive agent call. In an efficient implementation that reuses the stack space of `append`, we see that the stack space is constant. The Oz kernel language also has the property that list-building functions are tail-recursive. The essential difference between the concurrent constraint model and Oz 3 is that concurrency in Oz 3 is explicit (explicit thread creation) instead of implicit (see Section 2.2 below). As a final remark, note that the three agents, namely the two `list` agents and the `append` agent, can potentially run in parallel, if the implementation supports it. An appropriate scheduler will stream the output of the first `list` agent to the `append` agent, and run the second `list` agent in parallel with this. Because of data dependencies, however, each agent internally will run sequentially.

2.1.5 Andorra Kernel Language (AKL). Around 1988, Sverker Janson and Seif Haridi started a project at SICS with the explicit goal of defining a single language that provides the abilities of both search-based logic programming (as in Prolog) and concurrent logic programming (as in GHC) [Janson and Haridi 1991]. This work was originally based on a general And-Or computation model first proposed by David H.D. Warren to support parallel execution of Prolog. The result of this work was AKL, which generalizes the concurrent constraint language of the previous section. (AKL also adds ports for actor-based programming, but we postpone their discussion until Section 5.1.) Because it provides a unified formal model, AKL allows combining the two paradigms of search and concurrency in a compositional way, namely using search in a concurrent setting, and concurrency

S	::=	$S_1\ S_2$	*concurrent composition*
	|	X **in** S	*variable introduction*
	|	c	*basic constraint*
	|	**if** C_1 [] $\cdots$ [] C_n **else** S **end**	*conditional*
	|	$\mathrm{p}(X_1 \cdots X_n)$	*procedure call*
	|	**choice** C_1 [] $\cdots$ [] C_n **end**	*disjunction*
	|	**bagof** $X\ S\ Y$ **end**	*aggregation*
C	::=	$X_1 \cdots X_n$ **in** S_1 **then** S_2	*clause*
D	::=	**proc** $\mathrm{p}(X_1 \cdots X_n)\ S$ **end**	*procedure definition*

Fig. 5. The AKL kernel language

in a search setting. Figure 5 shows the AKL kernel language. It generalizes the concurrent constraint language of Figure 3 in two ways:

- *Deep guards.* The guard in a clause C can be an arbitrary computation, defined by a statement S_1 instead of just a basic constraint. We call this a *deep guard*, as opposed to the *flat guard* of the concurrent constraint language. The deep guard's computation builds a conjunction of basic constraints at run-time. A guard can be chosen nondeterministically when this computation terminates and when the basic constraints are entailed by the constraint store.
- *Encapsulated search.* Two new statements are added, namely **choice** and **bagof**. These statements work together: the **bagof** $X\ S\ Y$ **end** executes a statement S that contains a **choice**. The **choice** gives a sequence of alternative computations, C_1 to C_n and the **bagof** can choose these alternatives in order. Individual solutions are referred to by X and the list of solutions is returned in Y. The **bagof** of Figure 5 is one example of many possible built-in aggregators in AKL. It is similar in spirit to list comprehensions in functional languages.

Nested computation spaces. The major technical contribution of AKL, which underlies both deep guards and encapsulated search, and which lets it merge concurrent and constraint programming, is the concept of nested computation space that includes the semantic concept of stability needed for its correct operation [Janson 1994]. We call *computation space* the constraint store together with the agents observing it, as used in concurrent constraint programming (see Figure 2). Note that in AKL, the computation space is an implementation concept that is not directly visible to the programmer, unlike Oz which made the computation space a first-class language concept. AKL allows computation spaces to be nested, which means that an agent can itself be a computation space (as shown in Figure 4). With respect to the parent computation space, we call such an agent a *local* computation space. A local space sees the constraints of its parent space in conjunction with its own constraints, but a parent space cannot see the constraints of a local space.

A local space is *stable* when it contains no reducible instructions and no additional parent constraints could make it reducible again. Stability is a termination condition on the local space. The two conditions are both important. First, the computation inside the local space must not be reducible. Second, the computation must not become reducible for any possible added constraint in the parent space (because parent space constraints are visible to the local space). More precisely, given a local space defined by an initial statement S_0 and an initial store σ_0. The local space executes by reduction steps, giving $(S_0, \sigma_0) \rightarrow (S_1, \sigma_1) \rightarrow \cdots \rightarrow (S_n, \sigma_n)$. Each reduction step can add constraints to the local constraint store. The local space is *stable* when for all satisfiable parent constraints σ, the local statement S_n is not reducible with the store $\sigma \wedge \sigma_n$. This can be implemented efficiently by tracking variable bindings.

Deep guards and encapsulated search can both be implemented with stability of local spaces:

- Execution of a deep guard. A deep guard can be chosen when it is stable and when its constraints are entailed by the parent store.
- Execution of encapsulated search. The **`bagof`** executes each alternative of the **`choice`** in a new local space. When an alternative is stable, **`bagof`** detects this and merges the solution (if it exists) into the parent space.

In its simplest form, **`bagof`** adds a Prolog-style depth-first search to the concurrent constraint model. It is more general than Prolog in that the solutions are provided incrementally and concurrently with other activities in the execution. We remark that **`bagof`** can provide other forms of search, such as breadth-first, iterative deepening, and so on, by changing how the alternatives of a **`choice`** statement are chosen. As we will see later, computation spaces were further developed in Oz, making them first class, which allowed user-programmable search strategies for constraint solving.

2.2 Gestation Period: 1991–1999

In the beginning. Oz was conceived in the HYDRA project, a German national project that started in June 1991 at DFKI in Saarbrücken, led by Gert Smolka and including three graduate students working on Oz, Martin Henz, Michael Mehl, and Ralf Scheidhauer. HYDRA was initially motivated to address complex deductive problem solving (which was needed for many DFKI projects). When the project started, its goals were to overcome limitations in logic programming, namely to improve control strategies and add constraints, and also to add concurrency and object orientation. By the end of 1991, the Saarbrücken group had made connections with groups in SICS in Sweden, led by Seif Haridi, and DEC PRL in France, led by Hassan Aït-Kaci, who were working in similar directions (with the AKL and LIFE languages). These three national groups then became part of the ACCLAIM European research project, which started in September 1992 [NA ACCLAIM 1992–1995]. Inspired by the concurrent constraint model of Vijay Saraswat, ACCLAIM's goal was to advance the area of concurrent constraint programming, at all levels from foundation to languages and frameworks, and implementation technology.

First design of Oz and start of ACCLAIM. At the same time that ACCLAIM started, HYDRA had already made a first prototypical implementation of Oz, with some key ideas including named first-class procedures and an early object system. The name Oz was suggested by Martin Henz. Identifying procedures with unique unforgeable names avoids the need to do higher-order unification, which is undecidable. A new record data type was added that was inspired by LIFE [Smolka and Treinen 1994; Van Roy et al. 1996]. This implementation consisted of four components: a programming environment based on Emacs, a compiler implemented in the typed logic programming language TEL (a logic programming language with types and functions designed and implemented in a previous project), an abstract machine implemented in C++, and a graphics system based on the public domain package Interviews. This implementation was successfully demonstrated at two international workshops: the CCL Workshop in Val d'Ajol, France in October 1992, and the ACCLAIM kickoff workshop in Stockholm in November 1992. The new language and its prototype implementation allowed experimentation with multiple programming paradigms.

Moving to concurrent constraints. However, this initial design of Oz, pre-ACCLAIM, was not based on concurrent constraints. When ACCLAIM started, the Oz design took the ideas of concurrent constraints coming from AKL. Furthermore, the initial design started with synchronous communication between concurrent agents (CSP style), which was abandoned in favor of asynchronous communication as provided by the concurrent constraint model.

Supporting constraint programming. Moving to concurrent constraints had the important consequence that it greatly facilitated strengthening the constraint programming part of Oz. In 1993, HYDRA added finite domain constraints and encapsulated search, which are needed for many applications (scheduling, production planning, natural language processing). By the end of 1993, a higher-order combinator for encapsulated search was added into the system [Schulte and Smolka 1994; Schulte et al. 1994; Smolka 1995a]. This made Oz into the first concurrent and higher-order programming language that integrates Prolog-style problem solving. It's clear that Oz and AKL are by now converging: the encapsulated search problem was first solved by AKL in 1991, and Oz took this solution and improved it by making it a higher-order combinator.

First informal release. Although there was not yet an official release, by November 1993 the system was released informally to interested researchers and supported by a 350-page Oz Handbook. Several DFKI projects, AKA-MOD and AKA-TACOS, used Oz, as well as Daimler Benz Research in Berlin and around one dozen other places worldwide. It is interesting to see that SICS had similar goals with its AKL project; as part of ACCLAIM, HYDRA by now has established a close collaboration with SICS, which will eventually lead to a joint effort on a common language and implementation.

Oz 1. The system was now moving towards a public release. Both the language and its implementation were evolving rapidly. For objects and concurrency, Oz 1 made the following design decisions:

- *Implicit thread creation.* For reasons of efficiency, Oz 1 abandoned the original concurrency model that had maximal concurrency where all are agents are concurrent by default (as in Section 2.1.4). This was replaced by implicit thread creation. By default, statements execute sequentially. When a statement blocks, a new thread is created that contains only the blocked statement. The main thread is not suspended but continues with the next statement.
- *Object system and mutable state.* The Oz 1 object system was based on two principles. First, the object's state was stored in a mutable container, called a *cell*. Second, all methods had two additional (hidden) arguments, which contained the input and output states. This was needed to guarantee the correct serialization of stateful operations in object programs, given implicit thread creation.

As a consequence of the above, the graphical subsystem was completely redesigned. A high-level window interface was built on top of Tcl/Tk, providing Oz programmers with a concurrent object-oriented interface to graphical objects. Among other things, this allows for multiple inheritance (Tcl/Tk has no inheritance). The Oz 1 language is well summarized in the major paper on the Oz Programming Model [Smolka 1995b].

Initial evaluation for multi-agent programming. For Daimler Benz Research at Berlin, HYDRA conducted a study evaluating the benefits of Oz as a basis for their multi-agent programming projects. The study and a prototype were presented to Daimler Benz on October 20, 1994. The study argues that Oz is superior to Allegro Common Lisp, their current platform at that date. For some applications the study showed that implementation effort can be reduced by one order of magnitude. Moreover, the Oz prototype was significantly more efficient in time and memory usage.

Release of DFKI Oz 1.0 (January 23, 1995). The Oz 1 implementation was now called DFKI Oz 1.0 and publicly released. The main components of this system are an interactive programming interface, a concurrent browser (interactive dataflow display of data structures), powerful interoperability features, an incremental compiler, an emulator executing the code produced by the compiler, and support for stand-alone applications. The system came with a garbage collector and object-oriented support for constructing graphical user interfaces based on Tcl/Tk. The efficiency

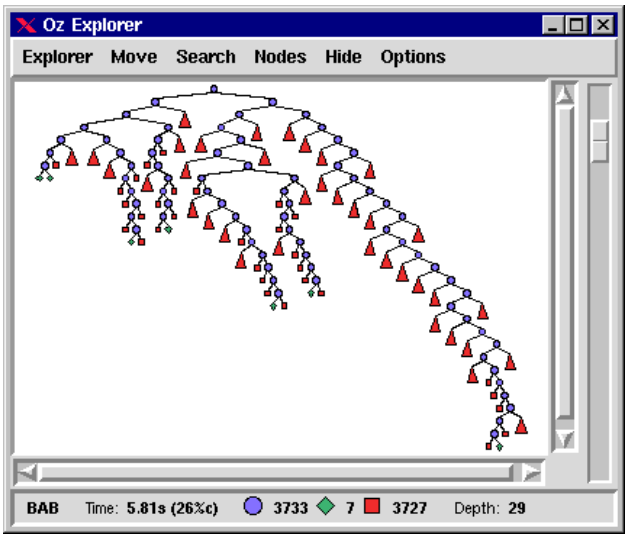

Fig. 6. Explorer tool for interactive exploration of a search tree.

of this implementation is competitive with commercial Lisp and Prolog systems of the time. In the first two months since the release, about 300 installations were registered worldwide. The release came with 11 documents (about 630 pages), consisting of tutorials, reference manuals, and the language definition. The survey of Oz document was requested about 740 times via ftp and the World Wide Web. Among the requests, commercial Internet sites made up about 40% of the US domain.

Extending the support for constraint programming. During 1995 and early 1996 the constraint system was again substantially strengthened through the work of Christian Schulte [Schulte 1997a,b, 1999, 2000a,b, 2002].

- *First-class computation spaces.* The basic operations of computation spaces are made available to programmers as an abstraction that allows programming inference engines directly in Oz [Schulte 1997b].
- *Reified constraints.* To facilitate construction of more complex constraints, the system was extended with reified propagators. A *propagator* is a concurrent agent that implements a constraint in the solver. A reified propagator adds a boolean control variable that represents the truth of the propagator's constraint. This allows to program high-level constraint programming abstractions such as a cardinality constraint and constructive disjunction. Constructive disjunction is an example of a deep constraint combinator [Schulte 2000b]. It is important for constraint applications like scheduling and timetabling. This extension was evaluated with real-world data, by constructing a timetable for a German college.
- *Explorer tool.* The Explorer tool allows real-time interactive exploration of a constraint problem's search tree [Schulte 1997a]. Constraints are attached to the tree's nodes, subtrees can be expanded or collapsed, and search can be directed interactively. Figure 6 gives a screenshot of the original Explorer. User-defined tools can be attached to the Explorer to display and interact with the constraints. For example, a timetabling solver could display partially solved timetables. At this time, DFKI Oz was the first system to provide an interactive tool to assist the development of constraint-based problem solving.

- *Recomputation-based search.* Recomputation trades space for time, which is essential for many applications that otherwise require too much memory [Schulte 1999]. All search functionality was reimplemented to support recomputation.
- *C++ interface.* An interface for writing finite domain propagators in C++ was designed and implemented in the system.

Joining efforts. The connection between Oz and AKL became closer in 1995 during the final period of ACCLAIM. The Swedish and German groups decided to join their efforts and continue work on a common system, which would be a successor to DFKI Oz 1.0. Two national projects, both called PERDIO, were started in early 1996, to continue the work as a collaboration [NA Haridi et al. 1996; Smolka et al. 1995]. On the Swedish side, Erik Klintskog, Per Brand, and Nils Franzén joined the project, and Konstantin Popov moved from Saarbrücken to Stockholm in 1997. In addition, Peter Van Roy, who was originally a member of DEC PRL working with Hassan Aït-Kaci on LIFE [Aït-Kaci and Podelski 1993], wanted to be part of this work and joined the DFKI when DEC PRL closed in October 1994. The stated goal for the PERDIO funding was to develop a system for open, distributed, fault-tolerant, reactive, and knowledge-intensive applications. The actual goal that was realized was to finalize the common language and prepare a major release, which became the Mozart system. Peter Van Roy visited SICS for one month in January 1996 to work on the Oz distribution model with the Swedish group. Van Roy and Seif Haridi continued to work together, first in Saarbrücken when Haridi came there for a sabbatical stay from May to November 1996, and subsequently after Van Roy joined UCL (Belgium) in October 1996.

First International Workshop on Oz Programming (WOz'95, Nov. 1995). The Dalle Molle Institute for Artificial Intelligence Research organized a workshop for Oz programming which was held in Martigny (Switzerland) with around 40 participants from both academia and industry [NA Cochard 1995]. Eighteen papers were accepted, of which 11 were by people other than the language developers, and there were three tutorials on Oz programming.

Oz 2. During 1996, the Oz 1 language changed radically to become Oz 2. Oz 2 made the following design decisions:

- *Explicit thread creation.* The implicit thread creation of Oz 1, while solving the efficiency problem, introduces many other problems. It makes reasoning about sequentiality and termination difficult, it gives no control over concurrency (e.g., ability to create concurrent agents explicitly), and it increases the complexity of the object system (which requires a separate mechanism to enforce sequentiality of state updates). Oz 2 abandons this in favor of explicit thread creation. Sequential composition is the default, and concurrency is never introduced implicitly but must be explicitly introduced by the program as a thread. As a result, Oz 2 can be seen as a higher-order multithreaded language with dataflow synchronization.
- *New object system.* Given explicit thread creation, the object system was redesigned and simplified. Experiments confirm that the new design results in a major efficiency improvement both with respect to space and time [Henz 1997a,b].
- *Exception handling.* Given explicit thread creation, an efficient and lean exception-handling mechanism was designed for Oz 2.
- *Actor support.* Oz 2 adds the port concept (see Section 5.1.4), which is a primitive providing efficient many-to-one communication and is a basic building block for actor programming. Experience with AKL showed that ports are an effective primitive [Janson et al. 1993].

All libraries and the object system were redesigned according to these changes.

Release of DFKI Oz 2.0 (September 1996). The Oz 2 implementation was called DFKI Oz 2.0 and publicly released.

Beginning of PERDIO project. Just after the HYDRA project was finished, the PERDIO project started in April 1996. From the viewpoint of a programmer building a networked application, PERDIO's goal was to combine network transparency with network awareness. Network transparency means that the semantics of computations does not depend on the node they execute on, and that the possible interconnections between two computations do not depend on whether they execute on the same or on different nodes. Network awareness means that distribution is achieved by explicit acts that give full control over communication patterns. While network transparency makes it easy to write distributed software, network awareness makes it possible to avoid undesirable network traffic and to fully plan the exploitation of the resources of a distributed computing system. When PERDIO started we considered that Oz was a good base language (targeting locally networked computers, not Internet, in the original proposal) because it combines expressiveness with simplicity:

- *Expressiveness.* Concurrency is essential for distributed systems. Logic variables provide for dataflow synchronization. Objects provide for concurrent data abstractions. Threads can communicate computational tasks through first-class procedures and classes. Code transfer between nodes is automatic, since classes are first-class values that can be copied across the network. Distributed lexical scoping is a direct consequence of network transparency. This concept was advocated by Luca Cardelli in his work on Obliq [Cardelli 1995]. Computations behave identically independent of the node they execute on.
- *Simplicity.* Simplicity comes from a design that expresses higher-level abstractions in a lean kernel language. Once distribution issues are understood for the kernel language, they will extend readily to the rest of the language. This holds true both for conceptual and implementational concerns. This is explained in Section 5.3.

In the first year of PERDIO (up to March 1997), we developed: (1) a distribution model for Oz, (2) a first version of an open, Internet-based module system for Oz, called components; (3) a first version of Ozlets, which are applications embedded into Web pages and written in Oz; and (4) a prototype system implementing all of the above. PERDIO eventually targeted partially synchronous systems, like the Internet, but not the complexities related to NATs and firewalls (a good explanation of this complexity is given in [Roverso et al. 2009]).

Oz 3. We now made a conservative extension of the Oz 2 language by adding read-only views of logic variables (for encapsulating abstractions based on logic variables, such as ports), and by adding abilities for large-scale and distributed programming. The resulting language is called Oz 3. The support for large-scale programming is based on components and was originally designed by Gert Smolka, Denys Duchier, and Leif Kornstaedt [Duchier et al. 1998]:

- *Functors and modules.* We added support for component-based programming with two new concepts, called functors and modules. A module is a running instance of a component, including data structures, objects, and threads. A functor is a module specification that defines a function whose arguments are modules and whose result is a new module. Instantiating a functor means to call this function with the correct modules as inputs. All libraries were then rewritten to become modules. A running application is a graph of modules.
- *Module managers and lazy loading.* The module manager is the part of the system that creates modules by instantiating functors. The module manager is considered part of the Oz 3 language definition. It is written in Oz using one additional language concept, namely lazy evaluation. In lazy evaluation, a function is associated to a logic variable, and when

there is an attempt to bind the variable, the binding is blocked and the function is called first, and the variable is bound to the result. This was the first attempt to add lazy evaluation to Oz. Because it does blocking inside of unification, it is not declarative, but this was fixed in 2003 (see Section 5.1.3 for the declarative version). The module manager uses lazy evaluation to implement lazy loading, namely that a module is created at first use. This is completely transparent to the program. This supports very short application startup times (fractions of a second), and together with the added support for standalone applications, it allows Mozart applications to be used in scripts.

The support for distributed programming builds on network transparency:

- *Connection.* Semantically, all Mozart processes live in the same shared store. But they do not share any references. We add support for connections, which allows a Mozart process to export an arbitrary language reference to another Mozart process. To initiate a connection, a Mozart application creates a *ticket*, which is a text string representation of an Oz language reference. This allows the reference to be communicated using any infrastructure that supports communication of text strings (such as sockets, Web pages, postal letters, phone conversations, and so on). Once the initial connection is made, a rich communication structure using Oz data structures can be built, in the same way that this is done by threads in a single application. This is supported in the Mozart system by distributed protocols specific to each language type, e.g., for logic variables, stateful entities such as objects, cells, and ports, and stateless values such as procedures and records. This is explained in Section 5.3.
- *Pickle.* A stateless data structure can be converted into a byte sequence called a pickle. A data structure can be recovered from its pickle, where the original and the recovered data structures are semantically identical. Pickles can be stored on files and can be loaded from URLs. This supports Mozart's repository of third-party contributed components, called MOGUL (Mozart Global User Library). Many Oz entities, including procedures and objects, receive upon creation a unique and fresh name fixing their identity. Since pickles can be moved between Mozart processes, the uniqueness of names must now be maintained globally. The original scheme we implemented relies on the assumption that the clocks and IP addresses of the involved computers are not manipulated, and that pickles are not faked.

Parallel search engines. Combining the distribution support and computation spaces, we now started to experiment with a parallel search engine for constraint solving. The new parallel engine takes the same problem definitions as the existing sequential engines, so there is no additional burden on the programmer. The parallel engine consists of a few hundred lines of Oz code and is a clear demonstration of the power of the high-level building blocks for distribution and constraints.

Release of Mozart 1.0 (January 25, 1999). The Oz 3 implementation was called Mozart since 1998, and was now publicly released with a BSD style open-source license. It came with an interactive Emacs developer interface and a set of libraries and tools for building standalone applications. It was released as a 32-bit system with binaries for Mac OS X, Linux, and Windows. Within the first three days after the announcement there were about 17000 page requests and 250 downloads of the system. Producing Mozart 1.0 was a major engineering effort. The system consists of 180000 lines of C/C++ and 140000 lines of Oz. The documentation comes in the form of Web pages and consists of 65000 source lines. Mozart 1.0 is an application-strength and complete implementation of Oz and improves considerably on the efficiency and stability of previous Oz implementations. Detailed information about this Mozart implementation including an evaluation is given in the two Ph.D. dissertations about the Oz Virtual Machine by Michael Mehl and Ralf Scheidhauer [Mehl 1999; Scheidhauer 1998]. The close collaboration between SICS and DFKI was essential to this effort.

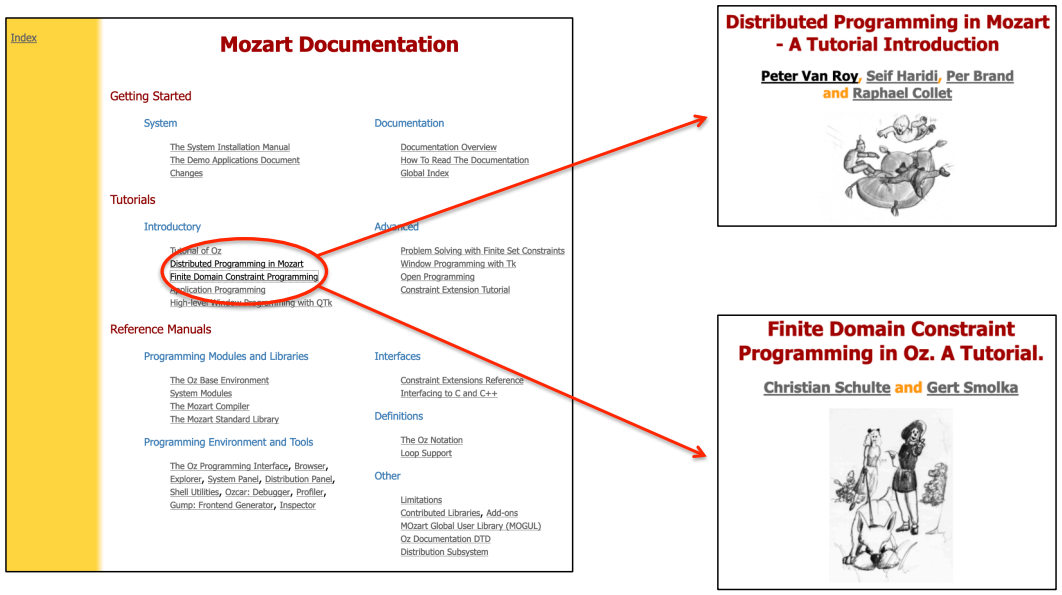

Fig. 7. Mozart 1.4.0 documentation page (in 2008)

2.3 Mozart Period: 1999–2009

During the period 1999-2009, Mozart was widely downloaded and used, and supported by key developers and the active user groups `mozart-users` and `mozart-hackers`.

- In the years 1999-2000-2001, our archives show that Mozart was downloaded more than 10000 times (more than 5000 times in the year 1999 alone). We have archives of the mozart-users mailing list, which show that it was highly active from 2001 to 2009 and declined after that. After 2009, Peter Van Roy continued to use Oz for his university courses and it was maintained mainly for that purpose.
- Oz was used as primary language in programming courses given at 16 universities worldwide that we are aware of, by people other than the language creators: Iowa State University, University of Central Florida, Brigham Young University, Texas A & M University, Rensselaer Polytechnic Institute, Dartmouth College, Norwegian University of Science and Technology, Universidad del Valle (Cali, Colombia), Università degli Studi del Sannio, California State University Los Angeles, Xavier University, Linköping University, National University of Singapore, Cairo University, University of Dortmund, and New Mexico State University.
- In 2004 we organized the MOZ2004 conference on Oz and its applications, which was held in Charleroi, Belgium in October 2004. There were 49 registered participants (some are shown in Figure 8). Out of 23 accepted papers, 16 were written by people other than the language creators. The proceedings were published in the Springer LNCS series as volume 3389 [Van Roy 2004].
- During this period, we set up the Mozart third-party library (called MOGUL: Mozart Global User Library), which accepted many packages by third parties. This library had 68 packages, of which 58 are by people other than language creators.
- From 1999 to 2003, we wrote a programming textbook organized according to the multi-paradigm structure of Oz and distilling our experience in the Oz project (see Section 4.1).

Fig. 8. Some participants at the MOZ2004 conference

This book was published by MIT Press (2004) and has sold more than 10000 copies as of 2018. It continues to sell well (hundreds of copies per year up to the present, not including licensed editions and translations). It has more than 600 citations in Google Scholar.

- During this period the Oz language remained stable, but we evolved the support for distributed programming. This support was completely redesigned to be modular, so that the distribution protocols needed for deep embedding were cleanly separated from the Oz emulator. This is documented in the 2005 dissertation of Erik Klintskog [Klintskog 2005]. We also completely redesigned the failure detectors and how they are visible in the language, which greatly improves the ability to design resilient distributed abstractions completely inside the language. This is documented in the 2007 dissertation of Raphaël Collet [Collet 2007].
- There were many other research publications around Oz and Mozart, including more Ph.D. dissertations, most of which are listed on the Mozart web site [NA Mozart Consortium 2018].

The last major release of Mozart 1 was Mozart 1.4.0, which was released in 2008 (see Figure 7). After this release, the maintenance of the Mozart 1 system diminished as key developers left the project (the system was mostly maintained by Ph.D. students) and research funding ran out.

Feature overview. At its release in 1999, the Mozart system mainly innovated in three areas: multiparadigm programming, open distributed computing, and constraint-based inference. We developed applications in scheduling and timetabling, in placement and configuration, in natural language and knowledge representation, and we also built multi-agent systems and sophisticated collaborative tools (see Section 4). The multiparadigm design, in particular, was spectacularly successful. It made it possible to use all the features of Oz together in a simple way, for example:

- Ultralightweight threads. Mozart implements ultralightweight threads and easily supports applications with hundreds of thousands of threads in a single operating system process. Because of dataflow synchronization using logic variables, programming highly concurrent applications is easy (Section 5.2.1 shows how it works).
- Logic programming. Mozart supports logic programming going beyond Horn clauses. Because of encapsulated search and concurrency, Mozart provides first-class Prolog top levels, i.e., it

is possible to run multiple concurrent instances of a Prolog search and ask for successive solutions inside a program.
- GUI programming. Mozart provides an object-oriented interface to the Tcl/Tk GUI library. This was later extended by QTk, which makes it possible to define graphic user interfaces much more compactly (see Section 4.3).
- Distributed components. Components are values in the Oz language, and the distribution model transfers them using copying (eager or lazy). This allows transparent and efficient transfer of program code between Mozart processes (as shown in Section 5.3).

Many code examples of multiparadigm programming are given in Section 5.2.

Language fixes. During this period, the Oz 3 language remained mostly stable. We fixed the semantics of lazy evaluation so that it would be declarative and fit well into the dataflow synchronization. The restructuration of the distribution subsystem changed the language interface to distribution, in particular for failure detection.

2.4 Education Period: 2009–Present

During this period, the core development of Mozart was finished and maintenance focused on using Mozart as a platform for education. Peter Van Roy continued to teach programming courses at UCL at the engineering faculty and supported Mozart for this purpose, using the textbook mentioned above (see Section 4.2). Oz is well-suited as a language for education, with its interactive interface and well-factored multiparadigm design.

Mozart 2. All versions of Mozart 1 are 32-bit systems. As personal computers migrated to 64-bit architectures in the years following the Mozart 1.4.0 release, a need was felt for a 64-bit Mozart version. Mozart 2 is a clean 64-bit reimplementation of Mozart that was realized at UCL by Sébastien Doeraene and Yves Jaradin and released in 2012. This system is being used and maintained up to the present day [NA Jaradin and Doeraene 2013]. Both Mozart 1 and Mozart 2 are available at the Mozart 2 site [NA Mozart Consortium 2018]. Mozart 2.0.1 was released in September 2018 by Guillaume Maudoux with support from other developers. This system still has significant use, e.g., more than 800 downloads in the six months since its release. Mozart 2 is a full implementation of Oz with the following differences:

- It lacks the constraint and distribution support. The UCL team did not have the resources to implement this in Mozart 2, so we focused on the essential support for education.
- It has a reflection interface that allows defining new implementations of kernel language operations within the Oz language. This was intended to enable a reimplementation of the distribution support within the Oz language, but this was never completed.
- It has a language extension to support list comprehensions designed by François Fonteyn in June 2014 [Fonteyn 2014]. The list comprehension extension seamlessly supports the eager and lazy dataflow paradigms illustrated in Section 5.2.
- It has a multicore extension designed by Benoit Daloze in June 2014 that allows taking advantage of multicore processors [Daloze 2014]. Each Mozart process, which is implemented as an operating system process, has a dedicated port that allows receiving messages sent from another process. Messages can be any values in the language (including procedures and classes) and are copied between Mozart processes in a similar way to Erlang processes. There are no shared references between Mozart processes.

2.5 Successes and Failures

Let us now take a step back to assess the results of the Oz project. The project started with very ambitious goals. It is interesting to see how this turned out in the long term. The initial goal of multiparadigm programming was largely achieved. The Oz language successfully integrates many paradigms and Mozart is a high-quality efficient implementation, as witnessed by the wide variety of large-scale applications (Section 4.3) and the longstanding use in education (Section 4.2). Both the Oz design and the CTM textbook successfully used the kernel language approach to present many paradigms in an integrated framework (see Section 5 and Section 4.1). During its heyday, Oz had impact in many areas, as explained in Section 4. We showed that the deep embedding used in Distributed Oz is a practical approach for distributed cluster computing (see Section 5.3 and the iCities project in Section 4.3). We showed that Oz is well-suited for teaching programming using a concepts-based approach (see Section 4.2). On the other hand, we failed in creating a self-sustaining Oz community. There were two main reasons for this.

2.5.1 Syntax. The first reason was the unusual syntax, and we failed to recognize this and fix it. The syntax had several problems: it was widely different from existing syntaxes, it was verbose in some common cases (e.g., lambda expressions), and the object system syntax was not polished. Feedback from users and potential users confirms our suspicion that the unusual syntax was one reason that Oz was not used as much as it could have been. It created a threshold that made it harder for potential new users to join the community. We note that this was only a threshold effect and did not hold for proficient users. Once a new user had overcome the threshold, they came to appreciate the system's power and simplicity.

The syntax design was a difficult issue because of the project's ambition: we aimed to support as many programming paradigms as possible. This put a strong constraint on the syntax: it needed to support all paradigms in a clean and factored manner. For example, we used parentheses for records and brackets for lists; this left us with braces for functions and procedures. This is an important lesson for future language designers: be especially careful about syntax, and be prepared to make big changes in the syntax when evolving the system. For example, the Erlang community developed a new language, Elixir, that runs on the same virtual machine as Erlang and is interoperable with Erlang. This was a very beneficial development for them.

2.5.2 Community. The second reason was much more important than the threshold effect of an unusual syntax. We failed to navigate the transition between funded research, during which Mozart was developed, and open-source development, which is mainly driven by social interactions and in which most developers are volunteers. During the gestation period, the project had a strong sense of purpose and was driven by a powerful development methodology (Section 3.2). When Mozart was released in 1999, the goal of multiparadigm programming was considered to be achieved. At that point, this sense of purpose diminished. A new sense of purpose did come into being with the Mozart release, namely to create a community around the newly released system, but this new sense was not as strong as the original. The original developers (including the authors) were primarily researchers, and we did not make a successful transition to an open-source project organized around a community of volunteer developers. Most of the key developers gradually lost interest and left the project, and were not replaced by other developers. The German and Swedish teams moved on to other projects, and only the Belgian team at UCL led by Peter Van Roy was left to evolve the system. As a result, system support diminished, and we were slow to evolve the system: 64-bit support came only in 2012 (with Mozart 2), and multicore support came only in 2014.

2.5.3 Legacy. Despite these failures, Oz was a pioneer in many ways. In programming education, Oz was a successful foundation for the concepts-based approach (see Section 4.2). During the

education period, the Oz project was refocused on education by the UCL team. This was moderately successful: UCL used Mozart to teach programming to all students of engineering for 15 years and developed several popular MOOCs in the edX consortium. Mozart was and continues to be used in other universities. We also attracted some outside developers and continued to evolve the system. As a result, Mozart maintained a presence in the general developer community.

Oz also pioneered several important programming concepts: lightweight threads (with shared data, unlike Erlang) which are used in Go, the distinction between mutable and immutable data which is used in Scala, deterministic dataflow, which is commonly used in cloud analytics tools, actors that return futures, which is being used in actor-oriented databases, and constraint logic programming, where Oz introduced first-class computation spaces that allow programming search strategies and deep constraint combinators independent of the declarative specification of the problem. Whereas in the 1990s, languages were mostly single-paradigm, in the present day this has changed and supporting multiple paradigms is now considered to be an important language property (see Section 3.3) [Van Roy 2006]. Some of the programming concepts pioneered by Oz have become commonplace in mainstream languages, as illustrated in Section 5.2.

3 PRINCIPLES

From the beginning, the Oz effort was guided by several strong principles. The coherence of the Oz 3 language can be attributed to these principles as well as to the enthusiasm and vision of the Oz developers. Here we explain four of these principles:

- *Explicitness*. We initially tried to make many operations implicit, but experience showed that this is often wrong.
- *Development methodology*. The development methodology was characterized by a combination of efficient implementation and simple formal semantics. This was possible because the Oz developers were both practical implementors and theoretical computer scientists.
- *Support multiparadigm programming*. The need to support multiple programming paradigms was recognized from the start, in the HYDRA project, and quickly broadened to cover much more than improvements in logic programming.
- *Combine dynamic and static typing*. Because of the goal of multiparadigm programming, we initially started with dynamic typing, but we recognized that this must be complemented with static typing, even for exploratory programming.

An early discussion of these principles is given in [Van Roy et al. 2003b].

3.1 Explicitness

During the AKL and Oz projects, we initially tried to make many operations implicit. For example, Oz 1 has implicit concurrency, where the program is sequential by default and threads are created whenever an operation blocks. Experience showed that this was almost always wrong and that it is better to make operations explicit, where they are inserted by an explicit programmer command. We give four examples of this that we observed during the Oz project:

- Provide explicit concurrency, i.e., sequential composition by default with explicit thread creation (see *Oz 2* paragraph on page 14). This is important for interaction with the environment, efficiency, reasoning about termination, and debugging. The Oz design introduced explicit concurrency in two steps. The original concurrency model of concurrent constraint programming had maximal concurrency, where all operations are concurrent and execution is directed by dataflow dependencies. Oz 1 replaced this maximally implicit model by a model with implicit thread creation, where threads are created only when statements block. This

solved the efficiency problem of the maximally concurrent model, but it had many other problems. Oz 2 replaced the implicit thread creation of Oz 1 by explicit thread creation.

- Provide explicit mutable state, i.e., variables are immutable by default and mutable variables (cells) and communication channels (ports) must both be declared explicitly. Cells and ports are equivalent in expressiveness, as explained in Section 5.1.4 (*Mutable state* paragraph). Our experience shows that cells should not be defined implicitly, as is often done in popular languages such as Java and Python, because this complicates program development and maintenance. Ports are important for actor communication and composition of concurrent components, and are declared for that purpose.
- Provide explicit laziness, i.e., execution is eager by default and laziness must be declared explicitly. In our experience, this greatly simplifies reasoning about complexity and it allows using laziness exactly where it is needed. Oz uses laziness for on-demand loading of software components (see *Oz 3* paragraph on page 15) and on-demand computation of constraint solutions (see Section 5.2.6). Explicit laziness is also important for efficient functional data structures [Okasaki 1998].
- Provide explicit search, i.e., programs should by default not use search. This is a problem with older logic languages such as Prolog, where search was implicit. With explicit search such as provided by first-class computation spaces in Oz 3, it can be used exactly where needed and exactly in the right form [Van Roy et al. 2003b]. This is important because of the high computational cost of search. However, search cannot be eliminated entirely because it is essential for completeness. It is interesting, however, to note that the search is used in a lazy fashion, where solutions are computed on demand using heuristics to reduce the need for search.

These four cases can be seen as examples of Clarke's Second Law [NA Wikipedia 2020a]:

> The only way of discovering the limits of the possible is to venture a little way past them into the impossible.

The Oz project started by trying to do many things implicitly, but as our experience showed that this did not work well, we took a step back from the brink and made the operations explicit in Oz 2. In a few cases (such as mutable state and laziness), we did not feel the need to implement implicitness in our first design so we avoided it from the start.

3.2 Development Methodology

The development methodology used in the Oz project has been refined over many years, and is largely responsible for the combination of expressive power, semantic simplicity, and implementation efficiency found in Mozart. The methodology was first explained in [Van Roy et al. 2003b]; partial explanations of it are given in [Smolka 1995b; Van Roy 1999]. We summarize it here.

At all times during development, there exist both a solid implementation and a simple formal semantics. However, the system's design is in continuous flux. The developers continuously introduce new abstractions as solutions to practical problems. The burden of proof is on the developer proposing the abstraction: he must prototype it and show an application for which it is necessary. The net effect of a new abstraction must be either to simplify the system or to greatly increase its expressive power. If this seems to be the case, then intense discussion takes place among all developers to simplify the abstraction as much as possible. Often it vanishes: it can be completely expressed without modifying the system. This is not always possible. Sometimes it is better to modify the system: to extend it or to replace an existing abstraction by a new one.

The decision whether to accept an abstraction is made according to several criteria including aesthetic ones. Two major acceptance criteria are related to implementation and formalization. The abstraction is acceptable only if its implementation is efficient and its formalization is simple.

To make this explanation concrete, we give a nonexhaustive list of examples of this methodology in the design of Oz:

- A first example occurs with constraint programming and encapsulated search. The original design used a special-purpose search combinator that itself spawned a local computation space [Schulte et al. 1994]. This complex design was eventually replaced by the much simplified computation space abstraction (see Section 5.1.6) [Schulte 1997b, 2002]. The latter provides the ability to program many relational and constraint programming abilities without requiring any additional support.
- A second example occurs with component-based programming using functors and modules, as mentioned in the *Oz 3* paragraph in Section 2.2. These new abilities did not require any extension to the kernel language; higher-order programming with records and names (unforgeable constants, see Section 5.1.1) is sufficient to define them.
- A third example is the support for lazy evaluation. As explained in the *Oz 3* paragraph, this was originally defined by making unification a blocking operation. In addition to making unification more complex, this also made it nondeclarative. This was later replaced by a single operation, `WaitNeeded`, which was simpler and easier to implement, and declarative to boot (see paragraph *Lazy functional dataflow* in Section 5.1.3). By using `WaitNeeded` exactly where it is needed the programmer can combine the techniques of deterministic dataflow and by-need synchronization (see Section 5.2.3).
- A fourth example is the object system of Oz 2 as opposed to Oz 1. As explained in Section 2.2, Oz 2 has explicit thread creation, as opposed to implicit thread creation in Oz 1. In addition to simplifying reasoning about programs, this also had the additional consequence of simplifying the object system: the Oz 1 object system had to explicitly serialize an object's updates, whereas in Oz 2 this was no longer necessary, since threads provided a sufficient serialization.
- A fifth example is the use of single assignment variables in a concurrent functional setting (called "dataflow variables" throughout this article, see definition in Section 2.1.2). One advantage is that single assignment is a weak form of mutable state that does not affect the purity of a functional program, while allowing some bindings. A second advantage is that all list functions trivially become tail-recursive without needing code transformations or other techniques (see Section 5.1.5). A third advantage is that all list functions can be used in a concurrent setting on streams, where a *stream* is a list with an unbound tail. A list function running inside a thread immediately becomes a concurrent agent, where input lists are used as input channels and output lists as output channels. A fourth, major advantage is that all higher-order functions become concurrency patterns; for example the `FoldL` function is the heart of a concurrent agent with internal state and the `Map` function combines a broadcast and convergecast (both in Section 5.2.4).

This methodology extends the approaches put forward by Hoare, Ritchie, and Thompson [Hoare 1987; Ritchie 1987; Thompson 1987]. Hoare advocates designing a program and its specification concurrently. He also explains the importance of having a simple core language. Ritchie advises having the designers and others actually use the system during the development period. In Mozart this is possible because the development environment is part of the run-time system. Thompson shows the power of a well-designed abstraction. The success of Unix was made possible due to its simple, powerful, and appropriate abstractions.

With respect to traditional software design processes, this methodology is closest to *exploratory programming*, which consists in developing an initial implementation, exposing it to user comment, and refining it until the system is adequate [Sommerville 1992]. The main defect of exploratory programming, that it results in systems with ill-defined structure, is avoided by the way the abstractions are refined and by the double requirement of efficient implementation and simple formalization.

The two-step process of first generating abstractions and then selecting among them is analogous to the basic process of evolution. In evolution, an unending source of different individuals is followed by a filter, survival of the fittest [Darwin 1859]. In the analogy, the individuals are abstractions and the filters are the two acceptance criteria of efficient implementation and simple formalization. Some abstractions thrive (e.g., compositionality with lexical scoping), others die (e.g., the "generate and test" approach to search is dead, being replaced by propagate and distribute), others are born and mature (e.g., dynamic scoping, see Section 5.3), and others become instances of more general ones (e.g., deep guards, once basic, are now implemented with spaces).

3.3 Support Multiparadigm Programming

In any large programming project, it is almost always a good idea to use more than one paradigm:

- Different parts are often best programmed in different paradigms.[1] For example, an event handler may be defined as an actor whose new state is a function of its previous state and an external event. This uses both the object-oriented and functional paradigms and encapsulates the concurrency in the actor.
- Different levels of abstraction are often best expressed in different paradigms. For example, consider a multi-agent system programmed in a concurrent logic language. At the language level, the system does not have the concept of state. But there is a higher level, the agent level, consisting of stateful entities called "agents" sending messages to each other. Strictly speaking, these concepts do not exist at the language level. To reason about them, the agent level is better specified as a graph of communicating actors.

Section 5.2 gives examples of what can be done in Oz when combining multiple paradigms in the same program. Note that it is always possible to *encode* one paradigm in terms of another. Usually this is not a good idea. We explain why in one particularly interesting case, namely pure concurrent constraint programs with state [Janson et al. 1993]. The canonical way to encode mutable state in a concurrent constraint program without state is by using streams. A stream represents a communication channel; a send is done by binding the tail to a pair of a message and new tail (see also Section 5.2.1). In this case, an actor object is a recursive predicate that reads a stream. A *reference* to an actor is a stream that is read by the actor. This reference can only be used by one sender object, which sends messages by binding the stream's tail. Two sender objects sending messages to an actor are coded as two streams feeding a *stream merger*, whose output stream then feeds the actor. Whenever a new reference is created, a new stream merger has to be created. The system as a whole is therefore more complex than a system with state:

- The communication graph of the actors is encoded as a network of streams and stream mergers. In this network, each object has a tree of stream mergers feeding into it. The trees are created incrementally during execution, as object references are passed around the system.
- To regain efficiency, the compiler and run-time system must be smart enough to discover that this network is equivalent to a much simpler structure in which senders send directly to receivers. This "decompilation" algorithm is so complex that to our knowledge no concurrent constraint or concurrent logic system ever implemented it.

[1] Another approach is to use multiple languages with well-defined interfaces. This is more complex, but sometimes works.

On the other hand, a much simpler solution is to add named streams to the language, such as ports in AKL or Oz. This supports many-to-one communication directly and hence directly supports communicating actors.

Similar examples can be found for other concepts, e.g., higher-orderness, concurrency, exception handling, and laziness [Van Roy and Haridi 2004]. We explain these examples:

- In a first-order language, a higher-order function can be encoded by explicitly representing the environment frame needed for the closure. This complexity is unnecessary if the language is higher-order.
- In a sequential language, concurrency can be encoded by explicitly representing task pools and implementing a scheduler that chooses which task to execute. This complexity is unnecessary if threads are added to the kernel language.
- In a language without exceptions such as the original C language, exception handling can be encoded by implementing each operation as a function that returns an error code, and checking the error code at each function call and returning immediately if there is an error [Kernighan and Ritchie 1988]. Section 2.3 of [Van Roy 2009] gives an example of this encoding. This complexity is unnecessary if exceptions are added to the kernel language.
- In a language without laziness, it is possible to encode lazy evaluation by program transformation. For example, the [Van Roy and Haridi 2004] textbook gives two implementations of a bounded buffer in functional programming: in the textbook, Section 4.3.3 (Figure 4.14) shows a bounded buffer in a concurrent functional language without laziness and Section 4.5.4 (Figure 4.27) shows a bounded buffer in a lazy concurrent functional language. In the former definition, the consumer binds a dataflow variable as a signal to the producer whenever it needs a new value. The latter definition is much simpler.

In each case, encoding the concept increases the complexity of both the program and the system implementation. In each case, adding the concept to the language gives a simpler and more uniform system. In general, when a specific problem is being solved, it can happen that programs become complicated for technical reasons that are not directly related to the problem. This is a sign that there is a new concept waiting to be discovered and added to the kernel language. We call this the *creative extension principle*. It was first discovered by Felleisen [1990].

3.4 Combine Dynamic and Static Typing

Our goal was to explore multiple paradigms and how they interact. This was mostly unknown territory when we started the Oz project, which is why we favored exploring expressiveness over the compile-time verification given by static typing.

We define a *type* as a set of values along with a set of operations on those values. We say that a language has *checked types* if the system enforces that operations are only executed with values of correct types. There are two basic approaches to checked typing, namely dynamic and static typing. In *static typing*, all variable types are known at compile time. No type errors can occur at run-time. In *dynamic typing*, the variable type is known with certainty only when the variable is bound. If a type error occurs at run-time, then an exception is raised. Oz is a dynamically-typed language. Let us examine the trade-offs in each approach.

Dynamic typing puts fewer restrictions on programs and programming than static typing. For example, it allows Oz to have an incremental development environment that is part of the run-time system. It allows to test programs or program fragments even when they are in an incomplete or inconsistent state. It allows truly open programming, i.e., independently-written components can come together and interact with as few assumptions as possible about each other. It allows programs, such as operating systems, that run indefinitely and grow and evolve.

On the other hand, static typing has at least three advantages when compared to dynamic typing. It allows to catch more program errors at compile time. It allows for a more efficient implementation, since the compiler can choose a representation appropriate for the type. Last but not least, it allows for partial program verification, since some program properties can be guaranteed by the type checker.

In our experience, we find that neither approach is always clearly better. Sometimes flexibility is what matters; at other times having guarantees is more important. It seems therefore that the right type system should be "mixed", that is, be a combination of static and dynamic typing. This allows the following development methodology, which is consistent with our experience. In the early stages of application development, when we are building prototypes, dynamic typing is used to maximize flexibility. Whenever a part of the application is completed, then it is statically typed to maximize correctness guarantees and efficiency. For example, module interfaces and procedure arguments could be statically typed to maximize early detection of errors. The most-executed part of a program could be statically typed to maximize its efficiency.

Much work has been done to add some of the advantages of dynamic typing to a statically-typed language, while keeping the good properties of static typing:

- Polymorphism adds flexibility to functional and object-oriented languages.
- Type inferencing, pioneered by ML, relieves the programmer of the burden of having to type the whole program explicitly.

In the Oz project, we chose to go in the opposite direction by having the default be dynamic typing. The Oz compiler does static type inference to improve compile-time error detection, but this is not otherwise made available to the programmer. This can be seen as a precursor to gradual typing, which defines a system where a type can be partially known or unknown at compile-time [Siek and Taha 2006]. Oz makes a start in this direction, but much more work could be done on the topic of static typing for multiparadigm programming.

4 IMPACT

4.1 CTM Textbook

Oz is used as the main language in the textbook *Concepts, Techniques, and Models of Computer Programming*, published in March 2004 by MIT Press [Van Roy and Haridi 2004] and translated into Japanese, Polish, French, and Spanish. The book is organized according to the kernel language approach throughout. It has more than 1000 programs and program fragments, all of which run on the Mozart System. The book was favorably received and is often compared to *Structure and Interpretation of Computer Programming* [Abelson et al. 1996; Deville 2005; Gammie 2009]. It is still popular at the present time and is widely referenced on the Internet. For example, it is listed on the Web page "Teach Yourself Programming in Ten Years" by Peter Norvig, Research Director at Google [NA Norvig 2006–2020]. The review by Peter Gammie in the Journal of Functional Programming [Gammie 2009] contains the following memorable assessment:

> The overarching achievement of this book is to be so provocative that one wants to engage the authors in debate about almost everything they say. Partly this is due to the chirpy writing style [...] but mostly it is their delicious iconoclasm.

Work on the book started in 1999 when Peter Van Roy and Seif Haridi realized that Oz was comprehensive enough to support a programming textbook. We decided to write a book at its natural size, explaining clearly all we had learned during the Oz project. We greatly underestimated the effort this would take, to organize and write down what we learned and give it a coherent

structure, but eventually the book was completed (the text was completed in Summer 2003).[2] We acknowledge Raphaël Collet for his help on Chapter 13, which gives a complete structural operational semantics for Oz, and Kevin Glynn who wrote the introduction to Haskell (Section 4.7). During the writing of the book, significant parts of the material were tested in programming courses (as explained in Section 4.2). The Oz language was stable during the writing except for one change, namely the fix to lazy evaluation which was completed just in time to make it in the book. It was done by adding the operation `WaitNeeded` to the kernel language.

The last public draft of the book is from June 2003. On June 18, a message was posted by user `timothy` to the Slashdot technology news site about the book, with title "A New Bible For Programmers?", linking to this draft. This link was "slashdotted": within 24 hours, there were more than 6000 downloads, causing the UCL computing science departmental Web server to crash. We subsequently received useful feedback that helped us correct some errors in the draft. We would especially like to thank William Cook for noticing and helping us correct a major error in the terminology for data abstractions. Thanks to his help, the book uses the correct terminology for the two fundamentally different forms of data abstraction, namely abstract data types (ADTs) and procedural abstractions (objects). The discussion with William Cook is archived on the programming languages blog Lambda the Ultimate for June 18, 2003.

4.2 Education

4.2.1 Traditional Courses. We have elaborated the book's approach into a second-year university level programming course. This was done before the book's publication by Seif Haridi, Christian Schulte, and Peter Van Roy, through courses given at KTH (Sweden), NUS (Singapore) (during Haridi's sabbatical stay), and UCL (Belgium), in 2001-2003. Van Roy has continued to teach and refine courses based on the book up to the present day. Since its inception, more than 5,000 engineering students at UCL have followed this course as part of their core curriculum in the five-year engineering degree. In addition, a number of other universities worldwide have taught a similar course based on the CTM book and the course material we provide (Section 2.3 gives a partial list).

Concepts-based approach. The course uses a concepts-based approach that progressively introduces new concepts and organizes them into kernel languages [Van Roy 2011; Van Roy et al. 2003a]. When the course was first taught to all engineering students at UCL, in Fall 2004, there was initial skepticism in the computing science department about the use of Oz since it was a research language and not an industrially popular language. The course was provisionally accepted for two years, to be followed by an evaluation to see whether it should be continued. After two years and positive evaluations from the students, the course has been taught each fall semester ever since (15 times up to and including Fall 2018). In the current version of the course, there are around four hundred students each year. The didactic team consists of Peter Van Roy assisted by around four teaching assistants and thirteen student monitors. The student monitors are third-year (or later) students who have successfully taken the course previously. Each student monitor manages one lab session per week. This pipelined structure has advantages for both cost and quality: it allows the course to be taught with limited resources, and it allows junior students to be taught by their seniors who

[2] The first author kept all the drafts during the writing process. This led to the following curious result. Plotting the number of distinct words w in a draft (where a "word" is a maximal sequence containing only letters or digits) versus the draft size s (in bytes) gives a function very close to a square root $w = c \cdot s^{0.5}$. A phone book (almost no repetition) would give an exponent of 1, whereas repeating the same text would give an exponent of 0. The actual exponent 0.5 seems to show that new information added during the writing process was always integrated into the existing information. Since c does not vary significantly with the size, it also seems to indicate that the thoroughness of the integration was relatively constant during the writing process.

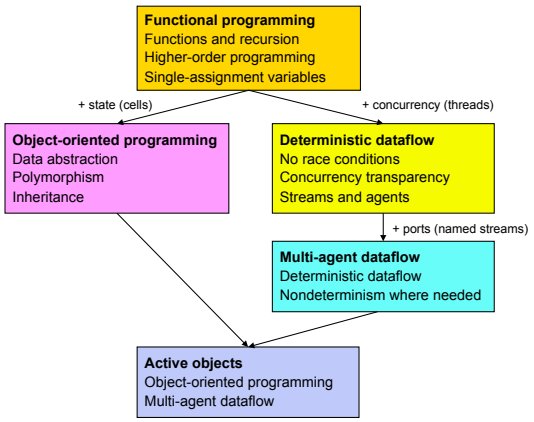

Fig. 9. Five paradigms covered in the MOOCs Louv1.1x and Louv1.2x

understand well the difficulties encountered when learning programming concepts. We have been using this approach for almost twenty years and we have gained two important insights about how to teach programming:

- The first insight is the importance of the second year. In the first year, the students are given an introductory course (which can be based on Java or Python). The language used in the first year is not critical, since its role is to give the students an introduction to algorithmic thinking. In the second year, the students are mature enough to understand a more abstract course organized around programming concepts. The language is more important in the second year, since we aim to give the students a broad and deep view of programming. Oz was successful because it directly supports all the concepts of the course in one system, which avoids the need for installing and using several systems.
- The second insight is the importance of formal semantics. University-level students in computer science need to understand the semantics of their subject, in the same way that students of electrical engineering need to understand the formal foundations of electromagnetism such as given by Maxwell's equations. We recommend that students be taught programming language semantics in the second year. While the students can be given some semantics in the first year, it is more important to start by learning algorithmic thinking. Giving a formal semantics in the second year can succeed if the semantics is properly designed to be simple and factored, such as the abstract machine semantics given in CTM. The semantics covers all the concepts we teach in the course.

4.2.2 Massive Open Online Courses (MOOCs). Since 2013, when UCL joined the edX Consortium, until 2018, the course was taught as two MOOCs (Massive Open Online Courses), called Louv1.1x and Louv1.2x [Combéfis et al. 2014; Combéfis and Van Roy 2015]. We split the course into two MOOCs, with 6 and 7 weekly lessons respectively, since there is too much material for a single MOOC. Around 50,000 students have attended these MOOCs up to the present day. The two MOOCs together cover five programming paradigms, as shown in Figure 9. All the material of our university course is taught in the MOOCs and the weekly on-campus lecture is organized as a flipped classroom. For legal reasons, we keep the same proctored on-campus evaluation as for a non-MOOC course.

The MOOCs provide programming exercises in Oz using the INGInious hosting platform for online exercises [Derval et al. 2015]. We have also developed a tool, CorrectOz, that is able to provide useful feedback to students when their programs have errors [Paquot and Magrofuoco 2016]. The tool includes an Oz parser and an abstract machine emulator, so it can detect both syntactic and semantic errors. The parser accepts an extended Oz syntax that includes ambiguity, so it can detect common errors such as missing keywords. The tool identifies around a hundred error markers, which we found by analyzing actual student errors from the previous year's course. It is able to give useful feedback for around 80% of incorrect programs in the MOOC exercises.

The two MOOCs are also open to external students. One of the general problems that universities face when organizing MOOCs is how to make them sustainable, since they require local resources. We solved this problem for our course in two steps. The first edition of the MOOCs was funded by the university (for the creation of the initial videos and exercises). All later editions of the MOOCs require no additional funding because it is part of a university course that already has significant resources allocated to it. One of the course's four teaching assistants is assigned to manage the MOOC for the UCL students as well as the external students. The main responsibility of this assistant is maintenance of the MOOC's support for programming exercises. In addition, all of the course's teaching assistants and student monitors help in managing the discussions on the MOOC forums.

4.3 Projects and Applications

Since its release, the Mozart system was widely used in research projects and innovative applications. We made an effort to track down the most important of these applications, and we hope that we did not miss any. Oz was initially designed to support knowledge-based multi-agent applications, and many of the applications using Mozart were in this area. The natural language processing community widely used Mozart and this was supported by a book of more than 300 pages [Duchier et al. 1999]. The Strasheela constraint-based music composition system was implemented in Mozart by Torsten Anders as part of his Ph.D. research [Anders 2007]. The TransDraw collaborative graphic editor was implemented by Donatien Grolaux and used a transaction protocol to mask network delays to give quick user interaction even on slow networks while guaranteeing a globally consistent drawing [Grolaux 1998]. Fractalide is an implementation of HyperCard combined with Flow-Based Programming done by Denis Michiels and Stewart MacKenzie [Michiels and MacKenzie 2014]. In addition to these applications, Mozart was used as the main development vehicle in several European and national research projects [Van Roy and Haridi 1999]. Three Swedish projects were COORD, DMS, and ToCEE. The COORD project developed techniques for agent coordination using decentralized market-based interaction models. The DMS project developed a multi-agent platform inspired by the FIPA model, using the properties of Mozart to improve the expressiveness of agent interaction models. The ToCEE project worked on a distributed environment for cooperative engineering in the construction industry. The SELFMAN European project worked on self-managing distributed systems based on structured overlay networks and first-class components [NA SELFMAN 2006–2009][Lienhardt et al. 2007]. The PIRATES project in Wallonia (Belgium) developed collaborative tools and libraries that built on Mozart's distribution support [NA PIRATES 1997–2003].

SimICS system-level architecture simulator. SimICS was the first system-level simulator that could boot a non-modified commercial operating system at the instruction level [Magnusson et al. 2002, 1998]. In 1998 it was able to boot unmodified operating systems including Linux 2.0.30 and Solaris 2.6. Mozart was used to implement SimGen, which is a key part of the SimICS system-level architecture simulator. Starting from a specification of the target instruction set, SimGen generates

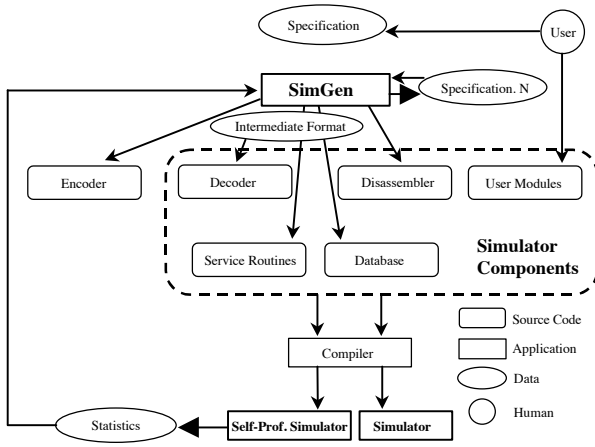

Fig. 10. The SimICS system-level simulator showing the SimGen instruction-level compiler written in Mozart (from [Magnusson et al. 1998]). This figure shows the role of SimGen in generating the various components needed for system-level architecture simulation.

the core components necessary for the operation of SimICS, as shown in Figure 10. The company Virtutech was created in 1998 to commercialize the SimICS technology, and Virtutech was acquired by Intel in 2010, who are continuing to use the technology up to the present day. SimGen was written by Fredrik Larsson starting in May 1996, originally in C but rewritten in Oz starting in the beginning of 1997 (before the official release of Mozart 1) because of the requirements of symbolic manipulation and high-level programming. SimGen was originally written for Ericsson in the APZ project (proprietary processor for telephone switches) and later used to generate the interpreter core of SimICS. The constraint part of Mozart was found useful for checking valid instruction patterns. SimGen was improved over the years and eventually moved to Mozart 2. SimGen uses a custom finite domain solver in the Mozart 2 version, since the latter no longer supports constraint programming. SimGen is still being used today and it is probably one of the longest lasting projects using Mozart.

Friar Tuck tournament scheduler. Friar Tuck is a round-robin sports tournament scheduling application based on constraint programming that lets tournament coordinators enter a variety of constraints to compute optimal solutions to complex tournament-planning problems (see Figure 11) [Henz 1999, 2000]. This software was used to schedule several sports tournaments in England and the USA in 1999 and 2000, including the West of England Club Cricket Championship and the Wisconsin Intercollegiate Athletic Conference. This software showed its ability to schedule the 1997/1998 Atlantic Coast Conference (ACC) in basketball by outperforming the solution by Nemhauser and Trick that was accepted by the ACC (1 minute for Friar Tuck versus 24 hours for the other solution) [Henz 2001; Nemhauser and Trick 1998]. Friar Tuck was initially implemented in Mozart by Martin Henz using Mozart's constraint solver and GUI tools, and used as the initial justification for creating a company also called Friar Tuck. However, the Mozart constraint solver did not scale to the problem sizes that the company encountered in workforce management, which was the area it focused on commercially. It was replaced by a custom solver using heuristic local search using pseudo-boolean 0/1 models with max-SAT (maximum satisfiability). The company still exists today and is called Workforce Optimizer.

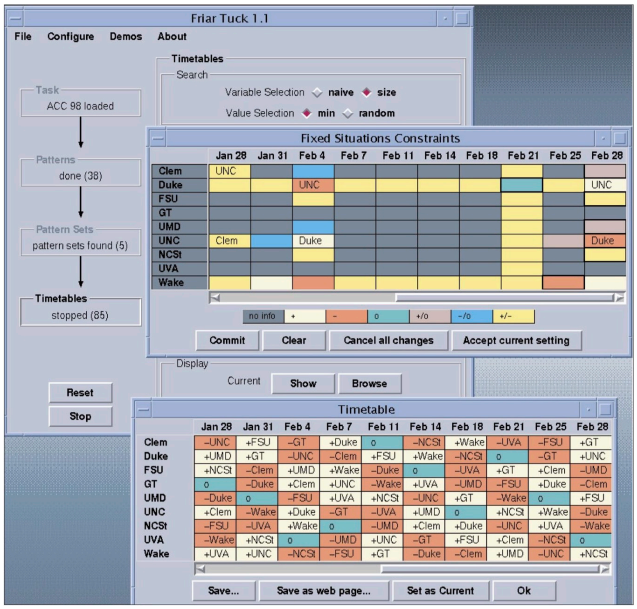

Fig. 11. The Friar Tuck tournament-scheduling application written in Mozart using constraint solving and GUI programming (from [Henz 2000]). This figure shows the constraint editor for fixing patterns and opponents, the interface to the solver phases, and the display of a generated schedule.

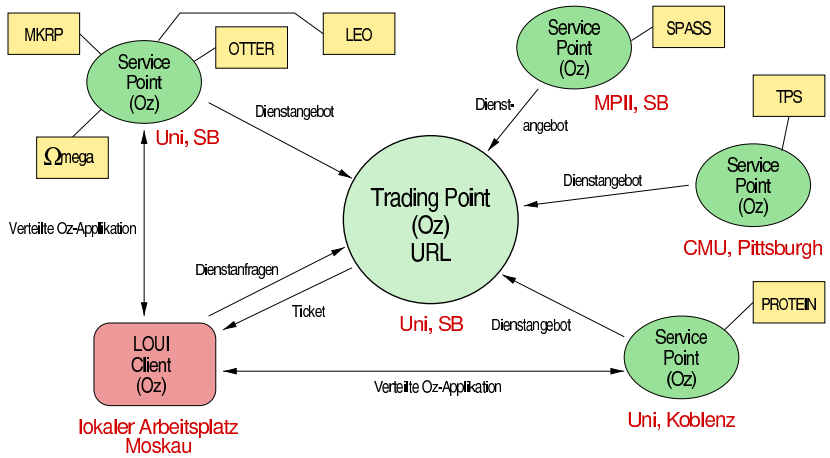

Fig. 12. The ΩMEGA proof assistant as part of a distributed application implemented in Oz, using multiple abilities of Mozart including distributed programming, constraint programming, and GUI programming (from ΩMEGA project, see [Siekmann et al. 2006]).

ΩMEGA proof assistant. ΩMEGA is a proof development system designed with the ultimate goal of supporting theorem proving in mathematical research practice and mathematics education, i.e., it is a mathematical assistant system that supports the user in the various tasks related to mathematical theorem proving (see Figure 12). This was one of the first big applications to be implemented in

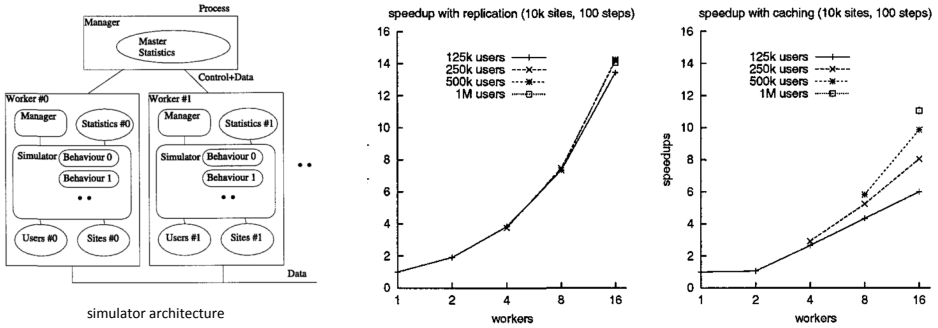

Fig. 13. Architecture and measured speedup of the iCities parallel agent simulator written in Mozart using the network-transparent distribution model (from [Popov et al. 2003])

Mozart, by Jörg Siekmann and his group at DFKI and Saarland University [Siekmann et al. 2006]. The system consists of a proof planner and a distributed collection of tools for formulating problems, proving subproblems, and for proof presentation. The distributed graphical user interface, LOUI, provides advanced communication facilities; the MBase tool is an active mathematical database; the constraint solver LINEQ helps to construct mathematical objects; several traditional automated theorem proving systems such as OTTER and SPASS and an induction prover, INKA, may tackle simple subproblems; diverse computer algebra systems can be used to simplify expressions and to oracle the instantiations of variables.

iCities agent simulation. The iCities European project ran from 2000 to 2003 and studied the evolution of inhabitants of virtual worlds (on-line communities), to understand emergent organizational patterns for the information society. This project used Mozart to develop a large-scale agent simulation platform (supporting millions of concurrent agents) running on workstation clusters [Popov et al. 2003]. This work has shown empirically that the distribution of users on sites follows a universal power law [Lelis et al. 2001]. The system did discrete-time simulation to capture the behavior of Web users and Web sites. Sites and users are connected in small-world graphs resembling the social network. Users are agents that exhibit behavior of real users, including visiting bookmarked sites, exchanging information in "word-of-mouth" style, and updating bookmarks. The simulator was completely written in Oz on Mozart using the network-transparent distribution model and taking advantage of lightweight threads, dataflow synchronization, and component-based programming. In 2002, the simulator supported up to 10^6 Web users on 10^4 Web sites. Hardware used was a cluster of 16 AMD Athlon 1900+ computers connected by 100Mbit switched Ethernet and running under Linux. On a single computer at 1GHz, the simulator required about 1 minute for 10^4 users to do 10^2 steps. On the cluster, the simulator achieved parallel speedups of 11 to 14, due to the high parallelism and efficient communication. Figure 13 shows the simulator architecture and speedup numbers ("replication" means keeping local copies of stateless data, "caching" means agents migrate locally for each simulation step).

Virtual reality programming. Mozart was used in the DIVE system for virtual reality programming [Axling et al. 1996; McGlashan and Axling 1996]. Mozart is designed to support multiple concurrent agents with lightweight concurrency, which makes it well-suited for VR-applications. DIVE is an interface between Oz applications and a toolkit for building distributed VR applications. It was part of a framework for Agent Oriented Programming specialized for defining agents in virtual environments for simulations. The framework was used to develop a system allowing collaborative

configuration of virtual battlefields and battle simulations where the computer generated forces are controlled with spoken natural language.

LogOz first-year programming course. The LogOz project created a first-year university programming course based on progressive enrichment of multi-agent microworlds [Cambron and Cuvelier 2006; de le Vingne et al. 2007]. The LogOz vision is that first-year programming should focus on concepts important for Internet computing such as concurrency, distribution, and fault tolerance. The microworlds are inspired by the layered structure of Oz, with practical content inspired by concepts from related efforts such as Logo, Toontalk, and Squeak. Students learn multi-agent programming, which is used as a framework to introduce concepts from higher-order programming, concurrency, graphic user interfaces, first-class software components, and fault-tolerant distributed computing. The LogOz environment was implemented in Oz in two master's projects and takes advantage of the lightweight concurrency and symbolic programming abilities of Oz. We created a complete one-semester course and tested it successfully on an audience of volunteer students.

QTk user interface toolkit. QTk was built as a frontend to Mozart's Tk library [Grolaux et al. 2001, 2002] and is explained in chapter 10 of the CTM textbook [Van Roy and Haridi 2004]. QTk uses the multiparadigm approach to simplify interface design. A user interface is defined by a combination of declarative and imperative paradigms. The static part of the interface is defined by a nested record structure. This record contains references to active objects to handle incoming and outgoing events and to dynamically modify the interface. Our measurements show that coding the same interface in QTk takes around one third the number of lines of code as in Tk.

Dataflow concurrency for Scala. Ozma is a conservative extension to the Scala programming language that supports Oz dataflow concurrency and lazy execution [Doeraene 2011; Doeraene and Van Roy 2013]. Ozma adds dataflow values, lightweight threads, lazy execution and ports, as a conservative extension to Scala semantics. We have designed Ozma and built a compiler and runtime environment that implements the full language. The implementation combines the frontend of the Scala compiler together with the backend of the Mozart system.

5 TECHNICAL OVERVIEW

Previous parts of this article give the historical development of Oz and its impact. In this section, we give a brief presentation of the technical contributions of Oz, to get a deeper understanding of the historical development. All of the technical concepts presented in this section have their backgrounds in the history.

5.1 Oz Language

5.1.1 Kernel Language Approach. Figure 14 shows three popular ways to define programming languages, namely by a kernel language, a foundational calculus, or a virtual machine. We explain the differences between these three approaches. Two approaches, the foundational calculus and the virtual machine, are well-known. A foundational calculus gives a formal model of the language that is designed to facilitate proving properties. For example, the λ calculus is a formal model for functional programming that can be used to prove the Church-Rosser property (confluence). A virtual machine defines an implementation of a language that maps in straightforward fashion to existing processor architectures, to facilitate practical implementation. For example, the Java Virtual Machine defines an emulator that can be used to implement the Java language. The third approach, the kernel language, is less well-known.

A *kernel language* is a formal model with a mathematical semantics, like the foundational calculus, but it has a different goal. Whereas the foundational calculus is designed to simplify mathematical

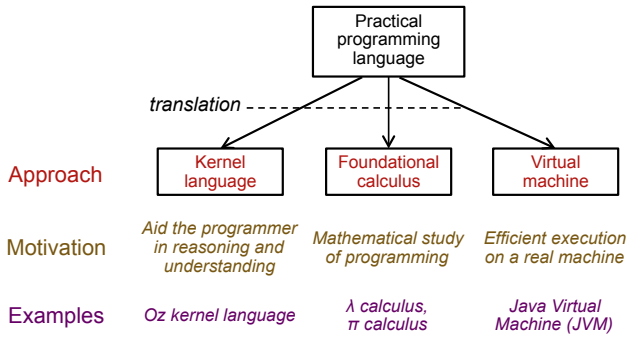

Fig. 14. Three ways to define a programming language

study, the kernel language is designed to be directly useful to the programmer. The foundational calculus is usually small, and often it is minimal (its constructs cannot be encoded by other constructs in the calculus). Programmer-visible concepts need to be encoded in it, which makes it cumbersome for programmers to write realistic programs in it. The kernel language tends to be larger, and while it is usually not minimal, it does respect other formal requirements such as factorization and compositionality. The kernel language contains concepts directly useful to programmers. The programmer can write practical programs directly in the kernel language, and define linguistic abstractions to make these programs more compact and readable (see Section 5.1.5). With a kernel language, the programmer can immediately see how different paradigms relate to one another, and how they can be used separately and together, since each paradigm corresponds to specific kernel language constructs. Based on the kernel language, it is straightforward to define a simple operational semantics for Oz (see Chapter 13 of [Van Roy and Haridi 2004]). The execution state consists of a multiset of instructions (where each instruction corresponds to a thread) that observe a shared constraint store. An execution step consists of choosing one instruction and performing a single reduction on it. With this operational semantics, we can reason about computational complexity and resource usage and we can define garbage collection.

The Oz 3 language, which we will refer to simply as Oz, is defined in terms of the kernel language by adding syntactic sugar and linguistic abstractions. Since the kernel language is a subset of Oz 3, it is possible to write programs directly in the kernel language. To make explicit the multiparadigm nature of Oz, the kernel language is organized in a layered structure. Each paradigm is defined by a set of concepts within this structure. Figure 15 shows most of the Oz kernel language. This kernel language was designed in stepwise fashion during the Oz gestation period (see Section 2.2). The CTM textbook presented in Section 4.1 is organized around this kernel language: each chapter corresponds to a particular subset of the kernel language. We now briefly explain the operations of the kernel language and organize them according to the principal programming paradigms supported by Oz. To simplify the presentation in this article, Figure 15 leaves out two concepts related to encapsulation of abstractions (unforgeable constants, called *names*, and read-only views of dataflow variables), and the concept of a failed value (to handle unification failure). For interested readers, the complete kernel language and its operational semantics are defined in Chapter 13 of the CTM textbook [Van Roy and Haridi 2004].

5.1.2 Records, Atoms, Tuples, and Lists. A *record* in Oz is a compound data structure that consists of a label f and a fixed collection of fields indexed by their field names l_1, ..., l_n. Records are the only compound data structure in the kernel language. Atoms, tuples, and lists are defined in terms

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)*

Fig. 15. The Oz kernel language: This figure shows the layered structure of the Oz kernel language. Each **boldface** heading introduces a programming paradigm. All paradigms above the double line are declarative whereas paradigms below this line are nondeclarative. The usefulness of each paradigm is explained and justified in the main text body. Because of this layered structure, paradigms coexist in the same language and Oz programs can be written as a combination of several paradigms with well-defined interactions.

of records. An *atom* is a record with no fields. A *tuple* is a record whose field names are consecutive integers starting from 1. The tuple `t(S R)` corresponds to the record `t(1:S 2:R)`. A *list* is either the atom `nil` or a tuple with label '`|`' and two fields containing an element and a list. The list `a|nil`, which can also be written `[a]`, corresponds to the record `'|'(1:a 2:nil)`.

5.1.3 Declarative Paradigms. The kernel language layers above the double line in Figure 15 all define declarative paradigms, which are either pure functional or pure relational. From the viewpoint of logic, all these paradigms are forms of logic programming that perform deductions which add information to the constraint store. The relational paradigms extend the functional paradigms

with search. The functional paradigms are related to the λ calculus as follows. We define the term "declarative" to mean confluence of program executions. In many variants of the λ calculus, which formalizes pure functional programming, the Church-Rosser theorem states that program executions are confluent, from which it follows that the final result of an execution is the same for all reduction orders, up to variable renaming. Functional programming in Oz is a variant of the λ calculus with dataflow variables and threads. Dataflow variables are single-assignment variables that allow synchronization on binding (wait until the variable is bound to a value). This is a restricted use of logic variables. Threads define sequential (applicative order) executions within a concurrent program execution. The Church-Rosser property was proved for this variant in [Niehren et al. 2006].

It is interesting to note that both concurrency and laziness can be modeled by restrictions on the order in which reductions are done. In a functional dataflow program, it is possible to create threads. The scheduler will only choose reduction orders that correspond to thread executions. For each thread, the scheduler guarantees fairness and a guaranteed minimum percentage of processor steps, according to a policy defined by the Mozart system. In a similar way, a lazy functional dataflow program allows only reduction orders that correspond to its threads and its by-need synchronizations.

In a multiparadigm setting, it is often useful to define declarativeness as an observational property, where a program is declarative if all possible final results of its executions are logically equivalent. This is an important modularity property for declarative programming. For example, it happens regularly that we would like to define a declarative component but the component can only be implemented in a nondeclarative paradigm. A simple example is memoization, where a function keeps a cache of previously computed results, so that future calls can check the cache instead of doing a computation. This trades off computation time for memory space and its implementation requires a paradigm with mutable state.

Functional. In the basic functional paradigm, programs are executed sequentially and arguments are evaluated before functions are called. This allows the standard programming techniques of eager functional programming. In addition, using the single-assignment property of logic variables allows functions that construct data structures (such as lists or trees) to be tail-recursive (example given in Section 5.1.5).

Functional dataflow. The functional dataflow paradigm adds dataflow concurrency. This allows to unnest function calls, so that they can run concurrently. For example:

```
Y={F1 {F2 X}}
```

can be replaced by:

```
local Z in
   thread Z={F2 X} end
   thread Y={F1 Z} end
end
```

This allows `F2` to build its result incrementally and `F1` to use this result incrementally. This is especially useful when the shared argument `Z` between the two functions is a *stream*. Then `F2` can build the stream incrementally while `F1` reads the stream incrementally. In effect, `F1` and `F2` have become concurrent agents and `Z` is a communication channel connecting the two. Because list-building functions are tail-recursive, this is efficient (i.e., concurrent agents use constant stack space). Section 5.2 gives more examples of this programming style.

Lazy functional dataflow. The lazy functional dataflow paradigm adds the `WaitNeeded` operation, which does by-need synchronization in a form compatible with dataflow variables [Spiessens et al. 2003]. `{WaitNeeded X}` suspends until `X` is needed by another operation. To be precise, an operation *needs* `X` if `X` must be bound to a nonvariable for that operation to continue. We can add calls to `WaitNeeded` in a program to add by-need evaluation where necessary. This will not change the result of the program, but only how much computation is done to obtain the result. Section 5.1.5 defines a linguistic abstraction for lazy functions. This allows the standard programming techniques of lazy evaluation to be used. In addition, it can be used together with concurrency, which gives extra expressiveness (see discussion in Section 5.2).

Relational and constraint. Relational and constraint programming are both forms of logic programming, in which a program is defined in first-order logic and executed by means of an efficient proof algorithm. Relations are first-order predicates; the general term *relational programming* refers to programming with relations, which includes both Prolog-style logic programming and relational database programming. The term *constraint programming* is used when the execution depends on solution algorithms that target specific kinds of relations, which are referred to as constraints (see Section 2.1.3). Constraint programming is often used for combinatoric optimization in tandem with approaches coming from operations research. As indicated in Figure 15, the relational and constraint paradigms are both implemented with first-class computation spaces.

Because functional programming in Oz is based on concurrent constraint programming, the relational paradigm can be seen an extension to functional programming that adds a lazy depth-first search ability to programs. Section 5.1.6 gives a linguistic abstraction for relational programs, by introducing the **`choice`**, **`fail`**, and `Solve` operations and defining them with computation spaces.[3] This lets us write programs that behave exactly the same as pure Prolog programs. Section 5.2.6 gives a programming example in this paradigm.

5.1.4 *Nondeclarative Paradigms.*

The layers below the double line in Figure 15 add nondeclarative expressiveness to the language. There are good reasons why such expressiveness is needed and pure functional or relational programming does not suffice. This is explained below for each paradigm.

Exceptions. Functions plus exceptions are the first nondeclarative paradigm (first layer below the double line in Figure 15). Exceptions are an important extension of declarative programming because they allow a program to detect when its execution becomes nondeclarative. This can happen for internal reasons (run-time errors) or external reasons (resource problems). Examples of run-time errors are taking the square root of a negative number, dividing by zero, accessing a nonexistent record field, or attempting to bind a variable to two different values. An exception can also be raised when there is a external problem such as power interruption or memory exhaustion. In all cases, raising an exception allows the program to manage problems without stopping program execution. In the case of a binding conflict, one binding will succeed and the other will fail, which in addition to being erroneous can introduce an observable nondeterminism in the program (if the bindings are done in different threads). In this case, the exception handler can encapsulate the nondeterminism and ensure that the rest of the program sees a deterministic result. In that way, the program becomes declarative again.

Actor dataflow. Actor dataflow extends functional dataflow with the ability for multiple senders to send messages to a given endpoint. The implementation serializes these messages so that the receiver sees a single stream of messages. We add one concept, called a *port*, to the kernel language. A port is defined as a unique unforgeable constant (the port's name) associated with an unbound

[3] For an implementation of the constraint paradigm, see Chapter 12 of [Van Roy and Haridi 2004].

dataflow variable (the end of the port's stream). Sending a message to the port's name atomically binds the unbound variable to a list pair (cons cell) that contains the message and the new end of stream. Two operations are provided for ports, namely `NewPort` and `Send`. Given two fresh unbound dataflow variables `P` and `S`, the `NewPort` operation creates a port:

```
P={NewPort S}
```

Here, `P` is bound to a new unique port name and `S` is the end of the stream. The `Send` sends a message to the port, which extends the stream by binding `S`:

```
{Send P alpha}    % S = alpha|_
{Send P beta}     % S = alpha|beta|_
```

For a computation to send a message to a port, it only needs to know the port's name `P`. Repeated sends will create a stream. Section 5.2.4 gives examples of actor dataflow.

Mutable state. There are many reasons why mutable state is important. A program may need to distinguish separate invocations of the same function. A function may need to remember information from past invocations, for example because it is part of an interaction with the real world. A function's implementation may need to be replaced by another implementation, without changing the interface. Furthermore, the mutable state must be named because it must be possible for a program to manage such changes during its own execution. In the general case, it must be possible for data structures to contain references to mutable entities. To realize all these abilities, we extend functional programming with the ability to store an updatable reference to the constraint store. We add one concept, called a *cell*, to the kernel language. A cell consists of a pair of a unique unforgeable constant, the cell's name, and a variable reference into the constraint store, with two operations `NewCell` and `Exchange`. Given a variable `R`, the `NewCell` operation creates a cell with initial content `R`:

```
C={NewCell R}
```

The `{Exchange C Old New}` atomically does two operations: it binds `Old` to the old content of `C`, and it sets `New` as the new content. The atomic combination of these two operations means that `Exchange` can be used correctly on cells shared between multiple threads.

From a theoretical point of view, ports and cells are equivalent in expressiveness in the sense that each can be implemented using the other. Therefore, in a foundational calculus, only one is strictly necessary. In our kernel language, however, we have found it useful to include both as separate concepts. From a programmer viewpoint, actor dataflow and mutable state are conceptually different ideas that are used in different ways: actor dataflow is fundamentally asynchronous (`Send` completes immediately and the message is added eventually to the port's stream) whereas mutable state is synchronous (`Exchange` completes when the cell has been updated). This distinction shows up clearly in the deep embedding approach used for distribution (see Section 5.3).

5.1.5 Linguistic Abstractions. The kernel language is important to define precisely the Oz language and its semantics. However, we do not expect programmers to use this language for practical program development. The practical Oz language seen by programmers extends the kernel language by adding syntactic sugar and linguistic abstractions. Syntactic sugar is just a convenient shortcut for frequently occurring syntactic patterns. Linguistic abstractions are more important because they introduce new programmer concepts. The kernel language is useful to understand how programs execute (in fact, the Oz implementation can run kernel language programs directly), but it is usually not the best way to write practical programs.

A linguistic abstraction is a construct that defines a syntax for an abstraction, i.e., a new programmer concept. Oz provides common linguistic abstractions such as functions (keyword **`fun`**),

classes (keywords **class**, **self**, **meth**), loops (keyword **for**), and locks (keyword **lock**). The relational programming primitives **choice** and **fail** are also provided as linguistic abstractions. The translation of these linguistic abstractions into kernel language defines their semantics.

Functions as linguistic abstractions. We explain how functions are provided in Oz as linguistic abstractions. A function with n arguments is translated into a procedure with $n + 1$ arguments, where the function result corresponds to the procedure's final argument. Function definitions and calls are translated into procedure calls and definitions. The function definition:

```
fun {F X} X*X end
```

is translated into the following procedure:

```
F=proc {$ X R} R=X*X end
```

Here and in the kernel language of Figure 15, the dollar sign `$` in the procedure's header is used as placeholder for an identifier, to indicate an anonymous procedure.

Lazy functions as linguistic abstractions. Translating function syntax to procedure syntax is more than just a technical convenience. Because the function result is available explicitly, it allows the translation to access it directly inside the procedure definition. This is used to translate lazy functions. The lazy function definition:

```
fun lazy {F X} X*X end
```

is translated into the following procedure:

```
F=proc {$ X R} thread {WaitNeeded R} R=X*X end end
```

As explained before, the `WaitNeeded` call waits until `R` is needed by an operation.

Tail-recursive functions for list construction. Functions that construct lists are translated into tail-recursive procedures using a similar technique: because the output can be accessed directly in the translation, it can be created before the recursive call. This also requires the output to be created as an unbound dataflow variable, which will be bound later in the recursive call. The definition of `Append`:

```
fun {Append L1 L2}
   case L1 of nil then L2
   [] H|M1 then H|{Append M1 L2}
   end
end
```

is translated into the following procedure:

```
Append=proc {$ L1 L2 L3}
   case L1 of nil then L3=L2
   [] H|M1 then
      local M3 in
         L3=H|M3
         {Append M1 L2 M3}
      end
   end
end
```

The first element of the output list is constructed with `L3=H|M3` and the rest of the output list is constructed when `M3` is bound in the recursive call to `Append`. For brevity we omit the straightforward translation of the **case** statement into a nested sequence of primitive **case** statements in the kernel language.

Space	::=	*X*=`{NewSpace` *P*`}`	*create new space X running program P*
	|	*I*=`{Choose` *N*`}`	*create a choice point with N alternatives and choose I*
	|	*Y*=`{Ask` *X*`}`	*ask space X for its status Y*
	|	`{Commit` *X I*`}`	*make choice I in space X's choice point*
	|	*Y*=`{Clone` *X*`}`	*create an identical copy Y of space X*
	|	*Y*=`{Merge` *X*`}`	*return space X's constraints referenced by Y*

Fig. 16. The computation space abstraction

Relational		
|	**choice** S_1 `[]` $\cdots$ `[]` S_n **end**	*disjunction*
|	**fail**	*failure*
|	*Y*=`{Solve` *X*`}`	*encapsulated search*

Fig. 17. The relational kernel language

5.1.6 The Computation Space Abstraction. Oz computation spaces are inspired by AKL computation spaces (see Section 2.1.5), except that instead of providing a built-in operation like AKL's **bagof**, the operations needed for Oz computation spaces are deconstructed and provided to the programmer as a first-class abstraction in the language. Figure 16 gives the main operations of computation spaces. We briefly explain what these operations do and how they are used and we show how to implement relational programming with them. For a more detailed explanation, see [Schulte 2002] or Chapter 12 of [Van Roy and Haridi 2004].

Relational programming with computation spaces. We explain how to implement a relational program written in a functional paradigm with an additional instruction that allows to choose among *n* alternatives (an example program is shown in Section 5.2.6). The parent space first creates a new local space with `NewSpace`. Execution then continues as follows:

- The local space runs the relational program, and when it needs to choose it calls `Choose`, which suspends. Concurrently, the parent space uses `Ask` to wait until the local space achieves stability. When the local space is stable, the call to `Ask` returns with one of `failed`, `succeeded`, or `alternatives(N)`. In the third case, `Choose` was called with *n* alternatives, so the parent space calls `Commit` to communicate a choice number to the local space (an integer from 1 to *n*) and resume the suspended `Choose` call. Inside the local space, the `Choose` call then returns with the choice number and the local space resumes execution.
- When a local space runs to completion with no more choice points, then the relational program has possibly found a solution. This is detected with `Ask` in the parent space when it returns `failed` or `succeeded`. In the latter case, there is a solution and the parent space brings it up from the local space with `Merge`.
- To enumerate over more than one possible choice, the parent space uses `Clone` to create clones of the local space before doing `Commit`. For each clone, the `Commit` is called with a different choice number.
- Multiple solutions can be collected by the parent space and returned in a lazy list, which behaves similarly to a Prolog interactive top-level query.

A linguistic abstraction for relational programming. Figure 17 shows a linguistic abstraction that defines a kernel language for relational programming which allows Oz to do logic programming

```
fun {Solve F}
   {SolveStep {NewSpace F} nil}
end

fun {SolveStep S SolTail}
   case {Ask S}
   of failed          then SolTail
   [] succeeded       then {Merge S}|SolTail
   [] alternatives(N) then {SolveLoop S 1 N SolTail}
   end
end

fun lazy {SolveLoop S I N SolTail}
   if I>N then
      SolTail
   elseif I==N then
      {Commit S I}
      {SolveStep S SolTail}
   else
      C={Clone S}
      NewTail={SolveLoop S I+1 N SolTail}
   in
      {Commit C I}
      {SolveStep C NewTail}
   end
end
```

Fig. 18. The Solve operation defined using computation spaces: lazy all-solution search engine

in the Prolog style. This extends the kernel language of Figure 15 with disjunction (**choice**), failure (**fail**), and a `Solve` function. The **choice** groups together a set of alternative statements. Executing the **choice** provisionally picks one alternative and continues execution. If this leads to a failure later on (**fail**), then another alternative is picked. The `Solve` function is given a zero-argument function as input and returns a lazy list of solutions. The input function calls **choice** internally. Note that `Solve` calls can be nested: it is possible for an execution of a relational program to itself run a local relational program.

Figure 18 gives the Oz code for the function `Solve`. For brevity, the Oz code of this figure is given in the syntax of the practical Oz language, which can be translated into kernel language. We do not give this translation, but it should be clear how it works. The **choice** statement is defined with the `Choose` operation as follows:

choice S_1 [] S_2 [] ... [] S_n **end**

translates into:

```
case {Choose N}
of 1 then S1
[] 2 then S2
...
[] N then Sn
end
```

When a **choice** statement is run, `Choose` first creates a choice point and suspends, waiting for a `Commit` in the parent space. When the parent space commits, the `Choose` call returns the alternative chosen by the parent and execution continues. The **fail** operation can be defined as a binding of two incompatible values, e.g., `1=2`.

The relational paradigm defined here is a simple example of computation spaces. The Mozart 1 system defines much more general uses. It provides many other search strategies, including breadth first, iterative deepening, limited discrepancy search, branch and bound, and so on. It also provides additional constraint domains and constraint operations, so that it can be used to do full-fledged constraint programming.

5.2 Examples of Multiparadigm Synergy

Oz can express many programming techniques of different paradigms and use them together cleanly in the same program. We do not present all these techniques here; they are explained in the CTM textbook and elsewhere [Van Roy 2009; Van Roy and Haridi 2004]. This section focuses on techniques that we consider to be uniquely important to Oz. They illustrate the expressive power that comes from supporting multiple paradigms in the same language.

It is important to remember that these techniques were possible in the DFKI Oz 1.0 release in 1995 and were fully supported by the high-quality Mozart system released in 1999 (20 years before the writing of this article). Since the first release of Mozart, Oz efficiently supported a programming style based on declarative dataflow concurrency and asynchronous calls to objects where the result was returned as a dataflow variable. In 1999, no other language with an implementation of comparable quality to Mozart supported this style. The style has spread in the last two decades, and these ideas are now available in widely used languages and systems. Both dataflow concurrency and asynchronous object calls returning futures are now commonplace, e.g., in big-data analytics tools and actor-oriented database systems. Supporting multiple paradigms within a language is now commonplace, e.g., both Scala and Java 8 support functional programming with lambda expressions.

5.2.1 Functional Dataflow. This paradigm extends pure functional programming with two concepts, namely dataflow variables and threads. We can write concurrent functional agents that communicate through streams. This paradigm is widely applicable, despite the fact that it cannot express nondeterministic programs. This is because it keeps all the advantages of functional programming in a concurrent setting. Race conditions, which are observable effects of nondeterminism, are impossible. The final results of an execution are always the same, no matter how the threads are scheduled (assuming the scheduler is fair). It is possible to take any functional program and add threads to it arbitrarily, without changing the result. The only effects of adding threads are to remove deadlocks (give results when the original program did not) and to make the program more incremental (give partial results when the inputs are built incrementally). In the programming courses (Section 4.2) we call this "Concurrency for Dummies".

The advantage of functional dataflow. To compare the functional paradigm with the functional dataflow paradigm, let us run two programs that perform the same computation, first sequentially and then concurrently. We define the function `Prod` that creates a list of elements from `L` to `H`, and takes 100 ms to create each element:

```
fun {Prod L H}
   {Delay 100}  % Delay at least 100 ms, to simulate a big computation
   if L>H then nil else L|{Prod L+1 H} end
end
```

Using `Prod` we define a program that creates a list and then maps its elements. Here is the sequential version:

```
L1={Prod 1 1000}
L2={Map L1 fun {$ X} X*X end}
```

The list `L2` contains the squares of integers from 1 to 1000. Here is the concurrent version:

```
thread L1={Prod 1 1000} end
thread L2={Map L1 fun {$ X} X*X end} end
```

This is a practical concurrent program because both `Prod` and `Map` are tail-recursive, which implies that they use constant stack size. It is generally true that all list-building functions in Oz can be converted into concurrent executions, because they are tail-recursive. In the concurrent version, the list `L2` will also contain the squares of successive integers from 1 to 1000. Both programs perform the same computation. So what difference does concurrency make? It is very simple: in the sequential version, the dataflow variable `L2` will remain unbound for 1000×100 ms = 100 seconds, and it will then be bound to the whole list at once. In the concurrent version, a new element of `L2` will appear every 100 ms, for 1000 times. "Concurrency for Dummies" has converted a batch program into an incremental program.

Streams. Let us examine closely how the previous example works. During the execution, both threads are active simultaneously, but not always runnable. Whenever the first thread adds an element to `L1`, the second thread becomes runnable and can read the element. The intermediate state of `L1` is an incomplete list, i.e., it has an unbound dataflow variable at its tail. This is an important concept that we have mentioned before (see Section 3.2). We give it a name: a *stream* is a list with an unbound dataflow variable as its tail. The stream `S`:

```
S=a|b|c|d|S2
```

can be extended by binding the tail `S2`:

```
S2=e|f|S3
```

A stream can be used as a communication channel between two threads in the functional dataflow paradigm. The first thread adds elements to the stream (it is a producer), and the second thread reads the stream (it is a consumer). Any function that builds a list can be used as a producer and any function that reads a list can be used as a consumer.

5.2.2 Higher-Order Functional Dataflow. In the functional dataflow paradigm, we can use functional building blocks as concurrency patterns. As shown in the above example, we can use the `Map` function to read a stream and build a new stream. In fact, all functions on lists can also be used on streams. This is especially useful for higher-order functions:

- **`for X in L do`** *S* **`end`**: For all elements `X` of list `L`, execute statement *S* in the scope of `X`. We call this a declarative **`for`** loop.
- `L2={Map L1 F}`: Transform `L1` by applying `F` to all its elements, giving `L2`.
- `X={FoldL L F U}`: Combine all elements of `L` together with transformation function `F` and initial value `U`, giving `X`. We summarize this with the equation $x = (\cdots((u\ f\ l_0)\ f\ l_1)\ \cdots\ f\ l_{n-1})$ where `L=[`$l_0\ \cdots\ l_{n-1}$`]` and f is used as a binary infix operator.

These functions were originally designed for pure functional programming, but they get new meaning as concurrency abstractions when used with dataflow variables in a concurrent setting. We give several examples of this in the following sections.

5.2.3 Lazy Functional Dataflow. This paradigm extends functional dataflow with by-need evaluation, where we only evaluate an expression if its result is needed by another expression. Control flows from output to the input: the computation is driven by the need to generate an output. In the kernel language, this requires adding one concept to the functional dataflow paradigm, namely a by-need wait which we call `WaitNeeded`. As a simple example, let us define the lazy function

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

that creates a lazy list of successive integers. `Ints` is defined as follows in the kernel language:

```
proc {Ints N R}
   thread {WaitNeeded R} R=N|{Ints N+1} end
end
```

In this way, all the programming techniques of traditional lazy evaluation become available to the Oz programmer. For example, all the techniques that use lazy evaluation to build efficient functional data structures are possible [Okasaki 1998]. Furthermore, lazy evaluation can be used together with concurrency giving additional techniques. Combining lazy evaluation with concurrency, we can write a bounded buffer where the producer does not run in lockstep with the consumer, which is not possible with lazy evaluation only (code for the bounded buffer is given in Section 4.5.4 of [Van Roy and Haridi 2004]). The combination of lazy evaluation and concurrency has been known at least since 1977 [Kahn and MacQueen 1977].

5.2.4 Higher-Order Actor Dataflow. Higher-order functional dataflow has the limitation that it is deterministic. To overcome this limitation, the kernel language adds one concept called a *port*, which can be understood as a named stream.

Actors. By using ports, it is possible to define actor programming. We define an *actor* as a named concurrent entity with an internal state, which transforms its state whenever a message is sent to it. An actor can be defined with a port and a thread, using `FoldL` to read the message stream `S`:

```
fun {NewActor I F}
   local S Out in
      Out = thread {FoldL S F I} end
      {NewPort S}
   end
end
```

The name of the actor is the name of its internal port. `I` is the initial state and `F` is the state transition function. The `FoldL` executes as a loop with an accumulating parameter. The first value of this parameter is `I`. The second value is `{F I M1}`, where `M1` is the first message sent to the actor. The third value is `{F {F I M1} M2}`, and so forth. Each new message sent to the actor appears on the message stream and causes a state transition. `Out` is the final state when the stream terminates, which causes the actor to terminate.

Contract net protocol. A contract net is a simple negotiation protocol (see Figure 19). There are three phases. First, a buyer sends a query to a set of sellers. Second, each seller sends a response with its price. Third, the buyer then chooses the best price, sends an accept to that seller, and a cancel to the others. This contract net protocol can be written very simply in the higher-order dataflow concurrency paradigm. We implement the buyer and the sellers as actors. We use a higher-order function to implement each phase of the protocol:

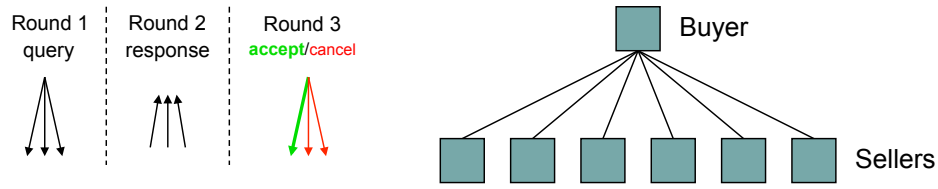

Fig. 19. Contract net protocol

- Map sends the query to all sellers and collects the responses. The expressive power of Map is remarkable in this example. It does both a broadcast (send to sellers) and convergecast (collect responses).
- FoldL calculates the lowest price by iterating over all responses.
- ForAll sends the final decision (accept or cancel) to the sellers.

With this understanding, we can program the contract net protocol as shown in Figure 20. This example is interesting because it is completely asynchronous. The `Map` sends messages and collects responses asynchronously. The list of responses `L` is created immediately, without waiting for responses to arrive, so it may contain unbound dataflow variables. This list is given as argument to `FoldL`. What happens if the `FoldL` is executed before all the responses arrive? This is not a problem: the `FoldL` operation will wait, in dataflow fashion, whenever it encounters a response that is not available yet. So everything works out correctly, even though all messages are sent asynchronously and responses can come at any time.

```
% First phase: send queries and collect the seller/price pairs.
L={Map Sellers fun {$ S} R in {Send S query(R)} t(S R) end}

% Second phase: find seller/price pair with lowest price.
t(S1 R1)={FoldL L.2
 fun {$ t(S1 R1) t(S R)} if R<R1 then t(S R) else t(S1 R1) end end
L.1}

% Third phase: send accept to best seller, cancel to others.
for t(S R) in L do {Send R if S==S1 then accept else cancel end} end
```

Fig. 20. Contract net protocol (source code)

This implementation can be easily extended to handle issues that may arrive in practice, for example to time out if a seller does not reply for a long time, or to reduce latency by sending a cancel message to a seller as soon as a price is received that is lower than that seller's price, without waiting until the whole computation is terminated.

5.2.5 Mutable State and Data Abstraction. Data abstraction is the main organizing principle for building complex software systems. A data abstraction is a part of a program that has an inside, an outside, and an interface between the two. The inside is inaccessible to the outside, except through the interface. The interface is a set of operations that can be used according to certain rules. Using data abstractions has three main advantages. First, a guarantee that the abstraction gives correct results, if the rules are respected. Second, a simplification of the program, since the user of an abstraction does not have to understand the implementation in order to use it. Third, division of labor that enables the development of programs by teams.

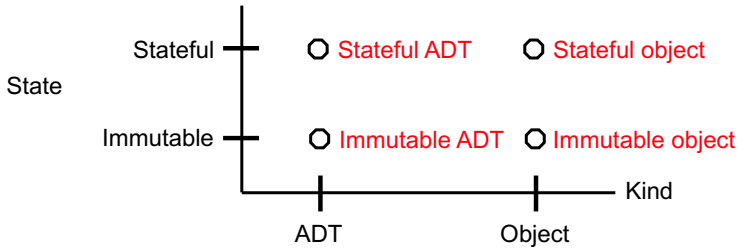

Fig. 21. Four ways to package a data abstraction

There are two main kinds of data abstraction, namely objects and abstract data types. An object groups together value and operations in a single entity. An abstract data type keeps values and objects separate. Orthogonally to this, an abstraction can be stateless or stateful. A stateless abstraction is a pure value (a constant). A stateful abstraction encapsulates a mutable state that can be changed during its operation. Mutable state is used in Oz for abstractions that interact with the real world [Van Roy 2018]. For a given functionality, this means that there are four ways to package the functionality in a data abstraction (see Figure 21). Modern programming languages use all of these ways. Both stateful objects and abstract data types (ADTs) are ubiquitous; numbers are typically implemented as ADTs (they are constants) and many languages allow defining stateful objects. Stateful ADTs are widely used for large data structures such as arrays. Immutable objects have become popular in big-data frameworks. For example, Spark uses a *fluent* interface which builds on method chaining of objects: an object invocation returns a new object. Some stateful systems use a fluent interface where an object invocation returns the same object with updated state, which is then available for a chained invocation.

Immutable object. All four possibilities for packaging data abstractions are given and compared in the CTM textbook [Van Roy and Haridi 2004]; here we show just the immutable object because it is an unusual approach as compared to the other three.

```
local
   fun {StackObject S}
      fun {Push E} {StackObject E|S} end
      fun {Pop E} case S of X|S1 then E=X {StackObject S1} end end
      fun {IsEmpty} S==nil end
   in record(push:Push pop:Pop isEmpty:IsEmpty) end
in
   fun {NewStack} {StackObject nil} end
end
```

Calling `NewStack` creates an empty stack and calling `IsEmpty` returns a boolean. Calling `Push` or `Pop` returns a new stack. The following code:

```
S={{{{{NewStack}.push 1}.push 2}.pop X}.pop Y}
```

binds `X`=2 and `Y`=1 and `S` to an empty stack. It is perhaps a remarkable fact that immutable objects can be defined completely inside the functional programming paradigm; no other concepts are needed.

ForCollect abstraction. We define a loop abstraction that can accumulate results over loop iterations. This abstraction is interesting because it uses mutable state to implement functional dataflow.

We start by introducing the declarative **for** loop which defines independent iterations. For example, the expression `for I in [1 2 3] do {Browse I*I} end` is identical to:

```
local I=1 in {Browse I*I} end
local I=2 in {Browse I*I} end
local I=3 in {Browse I*I} end
```

(where `Browse` is a Mozart operation that displays its argument.) Each iteration is independent; the identifier `I` references one element of the list in each iteration. Let us extend this declarative **for** loop with the ability to accumulate results. We would like the statement:

```
R=for I in [1 2 3] do
     ... % accumulate I*I
  end
```

to return `R=[1 4 9]`. We specify a new abstraction, `ForCollect`, that can do this:

```
R={ForCollect L proc {$ C I} S end}
```

This loops over all elements in `L` and executes statement *S* with loop index `I` and collect procedure `C`. Calling `{C X}` inside *S* will accumulate `X` in `R`. The statement:

```
R={ForCollect [1 2 3] proc {$ C I} {C I*I} end}
```

will return `R=[1 4 9]`. Seen from the outside, `ForCollect` is a functional abstraction. Furthermore, because of the deep integration of programming paradigms in Oz, `ForCollect` works correctly in a concurrent setting with streams, and it is also possible to write a version that creates a lazy stream as output. Running `ForCollect` in its own thread will create a functional agent:

```
S2=thread
      {ForCollect S1 proc {$ C X} if X mod 2 == 0 then {C X*X} end end}
   end
```

This takes an input stream of integers `S1` and creates an output stream `S2` containing only the squares of the even integers. We now show how to implement `ForCollect`. The collect procedure `C` cannot be written in the functional paradigm because it has memory: each call to `{C X}` appends `X` to the output list. `C` can only be defined using mutable state, i.e., a cell or a port. In our implementation, each running instance of `ForCollect` uses one cell internally to store the current end of its output list. This gives the following definition:

```
proc {ForCollect L P R}
   local
      Acc={NewCell R}
      proc {C X} local R2 in {Exchange Acc X|R2 R2} end end
   in
      for X in L do {P C X} end
      {Exchange Acc nil _}
   end
end
```

The result is terminated by `nil` when the `ForCollect` operation terminates. In this code, the `{Exchange C X|R2 R2}` binds the end of the output list to `X|R2`, which adds `X` to the list, and sets the new end of the list to `R2`. The output list is terminated with `nil` when all executions of `P` terminate. The use of `Exchange` ensures that `ForCollect` will work correctly when the collect procedure is called from more than one thread. This is because `Exchange` atomically does both a read and write; using separate read and write would allow race conditions where a second thread inserts an operation in between the read and write. `Exchange` is the simplest way to avoid this

problem; another solution would be to use explicit locks, by defining locks in the kernel language (which also requires mutable state).

5.2.6 Relational Programming. To illustrate relational programming in the Prolog style, we show a nondeterministic list append operation. In Prolog syntax this can be written as follows:

```
append([], L, L).
append([X|M1], L2, [X|M3]) :- append(M1, L2, M3).
```

In Oz the relational program is written as follows:

```
proc {Append L1 L2 L3}
   choice L1=nil L2=L3
   [] M1 M3 X in L1=X|M1 L3=X|M3 {Append M1 L2 M3}
   end
end
```

It is instructive to compare this program with the `append` predicate in Section 2.1.4 and the `Append` function in Section 5.1.5. In the relational paradigm, we search for solutions by calling `Solve`:

```
L={Solve fun {$} X Y in {Append X Y [1 2 3]} t(X Y) end}
```

This computes a lazy list that contains all solutions to the query `{Append X Y [1 2 3]}`. It corresponds exactly to the following Prolog top level query:

```
|?- append(X, Y, [1,2,3]).
```

Asking for all elements of `L` (for example, by printing them) will result in:

```
L=[t(nil [1 2 3]) t([1] [2 3]) t([1 2] [3]) t([1 2 3] nil)]
```

These results are ordered according to a depth-first traversal of the solution tree, as in Prolog.

5.3 Distribution

As Oz 1 was being completed, the Oz research community realized that the Oz language design would be a good starting point for building a distributed programming system. Because the language cleanly separates immutable, dataflow, and mutable language entities, it would be possible to use a *deep embedding* approach, where each language entity is implemented with its own distributed algorithm. The design and implementation of Distributed Oz started in earnest in 1995 in the two PERDIO projects mentioned in Section 2.2. The stated goal of these two projects was to build a system for cluster computing. The distribution support as explained below was part of the Oz 3 language and its implementation was part of the Mozart Programming System first released in 1999 [Haridi et al. 1999, 1998, 1997; Van Roy 1999][NA Van Roy et al. 1999].

5.3.1 Deep Embedding of Distribution in Oz. The first goal of Distributed Oz was to separate the language semantics from the distribution structure. As far as possible, each language operation should have the same semantics independent of the distribution structure. This separation means that applications can be initially written in one distribution structure and run in another distribution structure without changing the source code. The only differences would be operation timing and possibility of partial failure. During the application's lifetime, the distribution structure could changed to accommodate changing requirements. For example, consider the contract net protocol of Section 5.2.4. This program has the same behavior if the buyer and the sellers are located on different nodes.

While this separation is an important goal, it cannot be perfect for the two reasons mentioned, namely performance and failure. Changing distribution structure will change the communication patterns needed during execution, which changes performance. Changing distribution structure also

changes the failure behavior. When a compute node or a communication link fails, the execution of a language entity may no longer obey the semantics. As a consequence, Distributed Oz consists of three parts:

- First, the language semantics is obeyed precisely if there are no failures.
- Second, to control performance the developer can choose for each language entity the desired protocol. Immutable entities, such as records and procedure values, are copied eagerly or lazily. Dataflow variables can be bound once, eagerly or lazily. Mutable entities require a consistency protocol. We provided two consistency protocols, namely stationary state and migratory state.
- Third, to add fault tolerance, the system incorporates failure detection which allows building fault-tolerance abstractions within the language.

5.3.2 Consequences of Deep Embedding. We show how to use deep embedding for distributed computing.

Sharing references between stores. To initiate a distributed computation, two Oz processes need to share one reference. We provide the ability to export a reference outside of an Oz process and to import a reference that was previously exported. At this point, the two processes behave as if they shared the same store. For example, let us define the function `Add` that keeps track of a running sum. The call `{Add X}` adds `X` to the sum and then returns the updated sum.

```
C={NewCell 0}
fun {Add X} Old New in {Exchange C Old New} New=Old+X New end
{Offer Add URI}
```

Exporting is done by `Offer` which stores a serialization of `Add` as a character string in the uniform resource identifier `URI`. The function `Add` can be imported from another process by calling `Take`:

```
Add2={Take URI}
Sum={Add2 25}
```

In the implementation, `Add2` is a local reference on the second process, but because of deep embedding, `Add2` is semantically identical to `Add`. In this way, `Add` can be imported into any number of processes and called from them. Semantically, it is as if `Add` was called from multiple threads. Because the definition of `Add` uses `Exchange`, this is a correct concurrent execution.

Distributed lexical scoping. A procedure value that references a language entity will continue to reference that entity, independent of where the procedure value is transferred across the network. This implies that entities that are moved or copied across compute nodes will continue to behave according to their specification. The function `Add` defined above will always reference the cell `C` no matter from where it is executed. To achieve distributed lexical scoping, the implementation transparently translates local references into remote references, and vice versa, when references are moved between processes.

Dataflow across nodes. The producer and consumer example of Section 5.2.1 can be run on different compute nodes. Because of the distributed implementation of streams (which uses the distributed binding protocol explained in Section 5.3.3) this creates an asynchronous pipeline between the two compute nodes.

Automatic code transfer between nodes. Let us define a compute server that uses its computational resources to do any computation that we give it:

```
P={NewPort S}
thread
```

```
   for M in S do {M} end
end
{Offer P URI}
```

To use this server, the client imports `P` and sends it the computation to be performed:

```
P2={Take URI}
fun {Fibo N} if N<2 then 1 else {Fibo N-1}+{Fibo N-2} end end
local F in
   {Send P2 proc {$} F={Fibo 30} end}
   {Browse F}  % Display F on the local screen
end
```

The code of the zero-argument procedure is transferred to the compute node and executed there. When the computation is complete, `F` is bound to the result, which is sent back to the client because of distributed lexical scoping.

Components with local resources. Resource dependencies are specified in Oz 3 in terms of functors and modules. A *module* provides a well-defined package of functionality with an interface and an implementation. Modules are localized to single compute nodes. A *functor* is a linguistic abstraction that specifies a module. It defines the module's interface, the implementation, and the other modules that it needs. Functors are first-class values. For example, the functor `Timer` specifies a simple timer that uses the underlying operating system functionality provided by the module `OS`:

```
functor Timer
   export starttime:S endtime:E
   import OS
define
   T={NewCell 0} in
   proc {S} T:={OS.time} end
   fun {E} {OS.time}-@T end
end
```

The functor `Timer` is a value that can be exported, imported by another node, and installed there using the local `OS` module of that node. This is an example of dynamic scoping. To implement a sandbox, the destination node can provide a version of `OS` with restricted functionality.

5.3.3 Implementation. A wide range of distributed protocols were used to implement Distributed Oz. Several of these are interesting in their own right and are presented below.

Distributed binding protocol (distributed unification). The distributed binding protocol is used to bind dataflow variables that have references on several compute nodes. This algorithm is efficient in common cases. For example, when the variable is bound on one node and read on a second node then the latency is the same as explicit message passing. The general binding algorithm is called *distributed unification on rational trees* and it was first defined formally and proved correct in [Haridi et al. 1999]. Distributed Oz implements an extended version of this algorithm that is well-behaved in case of network and node failures.

Migratory state protocol. This protocol migrates an object to the node where it is called, while maintaining consistency and lexical scoping [Van Roy et al. 1997]. This does consistent caching of the object state.

Distributed garbage collection. Distributed Oz implements garbage collection at three levels:

- Each compute node has a local garbage collector. This collector coordinates with the distributed collectors described below.

- To handle distributed references, this is extended with a weighted reference counting algorithm. This algorithm is efficient and scalable, but it does not handle cycles between mutable entities on different nodes. These are left as a programmer responsibility.
- To handle failure, this is further extended with a time-lease mechanism. This handles permanent or long-lived temporary failures.

6 CONCLUSIONS

This paper gives a retrospective view of the history of the Oz project, including both the community aspects and the technical aspects. We show how Oz has its roots in the logic programming community and how its development broadened to cover many programming paradigms. We explain the Oz design process with the people involved and the decisions made, we show how the language evolved over the years, and we give the principles of its design. Oz had impact on both research and education, and we give highlights in both these areas. We show what Oz programming is like and we gives examples of synergy when combining several paradigms in one program. We hope that the ideas of this article will be an inspiration to future language designers.

ACKNOWLEDGMENTS

We acknowledge the people who contributed to Oz and Mozart: 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, (Al)fred Spiessens, Ralf Treinen, Peter Van Roy, Jörg Würtz, Andres Zarza Davila, and many others who contributed directly and indirectly.

REFERENCES

Harold Abelson, Gerald Jay Sussman, and Julie Sussman. 1996. *Structure and Interpretation of Computer Programs, Second Edition.* The MIT Press, Cambridge, Mass.

Hassan Aït-Kaci and Andreas Podelski. 1993. Towards a Meaning of LIFE. *Journal of Logic Programming* 16, 3-4 (July-August), 195–234.

Torsten Anders. 2007. *Composing Music by Composing Rules: Design and Usage of a Generic Music Constraint System.* Ph.D. Dissertation. Queen's University Belfast(Feb.).

Robert L. Ashenhurst and Susan Graham (Eds.). 1987. *ACM Turing Award Lectures: The First Twenty Years.* ACM Press.

Tomas Axling, Seif Haridi, and Lennart Fáhlen. 1996. Virtual reality programming in Oz. In *Virtual Environments and Scientific Visualization (Eurographics).* Springer-Verlag Wien(May).

Isabelle Cambron and Mathieu Cuvelier. 2006. *La programmation en première année basée sur l'enrichissement progressif de micromondes multi-agents (First-year programming based on progressive enrichment of multi-agent microworlds).* Master's thesis. ICTEAM Institute, Université catholique de Louvain(May). https://www.info.ucl.ac.be/~pvr/micromondes.html In French.

Luca Cardelli. 1995. A language with distributed scope. *ACM Transactions on Computer Systems* 8, 1 (Jan.), 27–59.

Keith L. Clark. 1987. PARLOG: the language and its applications. In *Proceedings of the Conference on Parallel Architectures and Languages Europe (PARLE). Volume II: Parallel Languages (Lecture Notes in Computer Science)*, A. J. Nijman J. W. de Bakker and P. C. Treleaven (Eds.), Vol. 259. Springer Verlag, Eindhoven, The Netherlands(June), 30–53.

Raphaël Collet. 2007. *The Limits of Network Transparency in a Distributed Programming Language.* Ph.D. Dissertation. Université catholique de Louvain(Dec.).

Alain Colmerauer. 1982. *PROLOG II Reference Manual and Theoretical Model.* Technical Report. Université Aix-Marseille II, Groupe d'Intelligence Artificielle(Oct.).

Alain Colmerauer and Philippe Roussel. 1996. The Birth of Prolog. In *History of Programming Languages–II.* 331–367.

Sébastien Combéfis, Adrien Bibal, and Peter Van Roy. 2014. Recasting a Traditional Course into a MOOC by Means of a SPOC. In *Second MOOC European Stakeholders Summit (EMOOCS'14)*. Lausanne, Switzerland(Feb.).

Sébastien Combéfis and Peter Van Roy. 2015. Three-Step Transformation of a Traditional University Course into a MOOC: A LouvainX Experience. In *Third MOOC European Stakeholders Summit (EMOOCS'15)*. Mons, Belgium(May), 76–80.

Benoit Daloze. 2014. *Extending Mozart 2 to Support Multicore Processors.* Master's thesis. Université catholique de Louvain, Louvain-la-Neuve, Belgium(June).

Charles Darwin. 1859. *On the Origin of Species by means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life.* Harvard University Press 1964 (originally John Murray, London, 1859).

Gregory de le Vingne, Maxime Romain, and Cécile Toint. 2007. *La programmation en première année basée sur l'enrichissement progressif de micromondes multi-agents, partie II (First-year programming based on progressive enrichment of multi-agent microworlds, part II).* Master's thesis. ICTEAM Institute, Université catholique de Louvain(Aug.). https://www.info.ucl.ac.be/~pvr/micromondes.html In French.

Guillaume Derval, Anthony Gégo, Pierre Reinbold, Benjamin Frantzen, and Peter Van Roy. 2015. Automatic Grading of Programming Exercises in a MOOC Using the INGInious Platform. In *Third MOOC European Stakeholders Summit (EMOOCS'15)*. Mons, Belgium(May), 86–91.

Yves Deville. 2005. Book Review: Concepts, Techniques, and Models of Computer Programming. *Journal of Theory and Practice of Logic Programming* 5, 4-5 (July), 595–600.

Sébastien Doeraene. 2011. *Ozma: Extending Scala with Oz Concurrency.* Master's thesis. Université catholique de Louvain, Louvain-la-Neuve, Belgium(June).

Sébastien Doeraene and Peter Van Roy. 2013. A New Concurrency Model for Scala Based on a Declarative Dataflow Core. In *Fourth Annual Scala Workshop (colocated with ECOOP, ECMFA, and ECSA)*. Montpellier, France(July).

Denys Duchier, Claire Gardent, and Joachim Niehren. 1999. *Concurrent Constraint Programming in Oz for Natural Language Processing.* Programming Systems Lab and Department of Computational Linguistics, Universität des Saarlandes.

Denys Duchier, Leif Kornstaedt, Christian Schulte, and Gert Smolka. 1998. *A Higher-order Module Discipline with Separate Compilation, Dynamic Linking, and Pickling.* Technical Report. Programming Systems Lab, DFKI and Universität des Saarlandes.

Matthias Felleisen. 1990. On the Expressive Power of Programming Languages. In *3rd European Symposium on Programming (ESOP 1990)*. (May), 134–151.

François Fonteyn. 2014. *Comprehensions in Mozart.* Master's thesis. ICTEAM Institute, Université catholique de Louvain, Louvain-la-Neuve, Belgium(June).

Tetsuro Fujise, Takashi Chikayama, Kazuaki Rokusawa, and Akihiko Nakase. 1994. KLIC: A Portable Implementation of KL1. In *Fifth Generation Computing Systems (FGCS '94)*. (Dec.), 66–79.

Peter Gammie. 2009. Book Review: Concepts, Techniques, and Models of Computer Programming. *Journal of Functional Programming* 19, 2 (March).

Donatien Grolaux. 1998. *Editeur graphique réparti basé sur un modèle transactionnel (A distributed graphic editor based on a transactional model).* Master's thesis. Université catholique de Louvain(June). In French.

Donatien Grolaux, Peter Van Roy, and Jean Vanderdonckt. 2001. QTk: A Mixed Declarative/Procedural Approach for Designing Executable User Interfaces. In *IFIP International Conference on Engineering for Human-Computer Interaction (EHCI 2001) (Lecture Notes in Computer Science)*, Vol. 2254. Springer Verlag, 109–110.

Donatien Grolaux, Peter Van Roy, and Jean Vanderdonckt. 2002. FlexClock: A Plastic Clock Written in Oz with the QTk Toolkit. In *1st International Workshop on Task Models and Diagrams for User Interface Design (TAMODIA 2002)*. Bucharest, Romania(July).

Seif Haridi, Peter Van Roy, Per Brand, Michael Mehl, Ralf Scheidhauer, and Gert Smolka. 1999. Efficient Logic Variables for Distributed Computing. *ACM Transactions on Programming Languages and Systems* 21, 3 (May), 569–626.

Seif Haridi, Peter Van Roy, Per Brand, and Christian Schulte. 1998. Programming Languages for Distributed Applications. *New Generation Computing* 16, 3 (May), 223–261.

Seif Haridi, Peter Van Roy, and Gert Smolka. 1997. An Overview of the Design of Distributed Oz. In *Proceedings of the 2nd International Symposium on Parallel Symbolic Computation (PASCO 97)*. ACM(July). Early version appeared in Workshop on Multi-Paradigm Logic Programming, JICSLP 96.

Martin Henz. 1997a. *Objects for Concurrent Constraint Programming.* The Kluwer International Series in Engineering and Computer Science, Vol. 426. Kluwer Academic Publishers, Boston(Oct.).

Martin Henz. 1997b. *Objects in Oz.* Ph.D. Dissertation. Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, Germany(June).

Martin Henz. 1999. Constraint-Based Round Robin Tournament Planning. In *International Conference on Logic Programming (ICLP 99)*. The MIT Press, Las Cruces, New Mexico, 545–557.

Martin Henz. 2000. Friar Tuck–A Constraint-Based Tournament-Scheduling Tool. *IEEE Intelligent Systems* (Jan./Feb.), 5–7.

Martin Henz. 2001. Scheduling a Major College Basketball Conference–Revisited. *Journal of Operations Research* 49, 1 (Feb.), 163–168.

Charles Antony Richard Hoare. 1987. The Emperor's Old Clothes. See [Ashenhurst and Graham 1987]. 1980 Turing Award Lecture.

Institute for New Generation Computer Technology (Ed.). 1992. *Fifth Generation Computer Systems 1992*. Vol. 1,2. Ohmsha Ltd. and IOS Press. ISBN 4-274-07724-1.

Joxan Jaffar and Jean-Louis Lassez. 1987. Constraint logic programming. In *14th ACM Symposium on Principles of Programming Languages (POPL 87)*. (Jan.), 111–119.

Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. 1992. The CLP(R) Language and System. *ACM Transactions on Programming Languages and Systems* 14, 3 (July).

Sverker Janson. 1994. *AKL–A Multiparadigm Programming Language*. Ph.D. Dissertation. Uppsala University and SICS.

Sverker Janson and Seif Haridi. 1991. Programming Paradigms of the Andorra Kernel Language. In *1991 International Symposium on Logic Programming (ISLP)*. The MIT Press, San Diego, CA, USA(Oct.), 167–183.

Sverker Janson, Johan Montelius, and Seif Haridi. 1993. Ports for objects in concurrent logic programs. In *Research Directions in Concurrent Object-Oriented Programming*. The MIT Press(July), 211–231.

Gilles Kahn and David B. MacQueen. 1977. Coroutines and Networks of Parallel Processes. In *IFIP Congress*. 993–998.

B. W. Kernighan and D. M. Ritchie. 1988. *The C Programming Language (ANSI C)*, 2nd edition. Prentice Hall, Englewood Cliffs, NJ.

Erik Klintskog. 2005. *Generic Distribution Support for Programming Systems*. Ph.D. Dissertation. KTH and SICS, Stockholm, Sweden(April).

Stelios Lelis, Petros Kavassalis, Jakka Sairamesh, Seif Haridi, Fredrik Holmgren, Mahmoud Rafea, and Antonis Hatistamatiou. 2001. Regularities in the formation and evolution of information cities. In *Second Kyoto Workshop on Digital Cities (Lecture Notes in Computer Science)*, Vol. 2362. Springer Verlag, 41–55.

Michael Lienhardt, Alan Schmitt, and Jean-Bernard Stefani. 2007. Oz/K: A Kernel Language for Component-Based Open Programming. In *6th International Conference on Generative Programming and Component Engineering (GPCE '07)*. (Oct.), 43–52.

Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. 2002. Simics: A Full System Simulation Platform. *IEEE Computer* (Feb.), 50–58.

Peter S. Magnusson, Fredrik Larsson, Andreas Moestedt, Bengt Werner, Fredrik Dahlgren, Magnus Karlson, Fredrik Lundholm, Jim Nilsson, Per Stenström, and Håkan Grahn. 1998. SimICS/sun4m: A Virtual Workstation. In *USENIX Annual Technical Conference*. (June).

Michael Maher. 1987. Logic Semantics for a Class of Committed-Choice Programs. In *Fourth International Conference on Logic Programming (ICLP 87)*. The MIT Press, Melbourne, Australia(May), 858–876.

Kim Marriott, Peter J. Stuckey, and Mark Wallace. 2006. *Constraint Logic Programming*. Vol. 2. Chapter 12, 409–452. Foundations of Artificial Intelligence, Handbook of Constraint Programming.

Scott McGlashan and Tomas Axling. 1996. Talking to Agents in Virtual Worlds. In *Third UK VR-SIG Conference*. Leicester, UK.

Michael Mehl. 1999. *The Oz Virtual Machine–Records, Transients, and Deep Guards*. Ph.D. Dissertation. Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, Germany(May).

Denis Michiels and Stewart MacKenzie. 2014. *Fractalide: HyperCard on Flow-Based Programming*. Master's thesis. ICTEAM Institute, Université catholique de Louvain, Louvain-la-Neuve, Belgium(June).

George L. Nemhauser and Michael A. Trick. 1998. Scheduling a major college basketball conference. *Journal of Operations Research* 46, 1 (Feb.), 1–8.

Joachim Niehren, Jan Schwinghammer, and Gert Smolka. 2006. A Concurrent Lambda Calculus with Futures. *Journal of Theoretical Computer Science* 364, 338–356.

Chris Okasaki. 1998. *Purely Functional Data Structures*. Cambridge University Press, Cambridge, UK.

Richard O'Keefe. 1990. *The Craft of Prolog*. The MIT Press.

Arthur Paquot and Nathan Magrofuoco. 2016. *CorrectOz: Recognizing Common Mistakes in the Programming Exercises of a Computer Science MOOC*. Master's thesis. ICTEAM Institute, Université catholique de Louvain, Louvain-la-Neuve, Belgium(June).

Konstantin Popov, Mahmoud Rafea, Fredrik Holmgren, Per Brand, Vladimir Vlassov, and Seif Haridi. 2003. Parallel Agent-Based Simulation on a Cluster of Workstations. *Parallel Processing Letters* 13, 4, 629–641.

Dennis M. Ritchie. 1987. Reflections on Software Research. See [Ashenhurst and Graham 1987]. 1983 Turing Award Lecture.

Roberto Roverso, Sameh El-Ansary, and Seif Haridi. 2009. NATCracker: NAT Combinations Matter. In *18th IEEE International Conference on Computer Communications and Networks*. IEEE Computer Society, San Francisco, CA(Aug.).

Vijay Saraswat and Martin Rinard. 1990. Concurrent Constraint Programming. In *17th ACM Symposium on Principles of Programming Languages (POPL 90)*. San Francisco, CA(Jan.), 232–245.

Vijay A. Saraswat. 1993. *Concurrent Constraint Programming*. The MIT Press.

Ralf Scheidhauer. 1998. *Design, Implementierung und Evaluierung einer virtuellen Maschine für Oz (Design, Implementation, and Evaluation of a Virtual Machine for Oz)*. Ph.D. Dissertation. Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, Germany(Dec.). In German.

Christian Schulte. 1997a. Oz Explorer: A Visual Constraint Programming Tool. In *14th International Conference on Logic Programming (ICLP 97)*, Lee Naish (Ed.). The MIT Press, Leuven, Belgium(July), 286–300.

Christian Schulte. 1997b. Programming Constraint Inference Engines. In *3rd International Conference on Principles and Practice of Constraint Programming (Lecture Notes in Computer Science)*, Gert Smolka (Ed.), Vol. 1330. Springer Verlag, Schloß Hagenberg, Austria(Oct.), 519–533.

Christian Schulte. 1999. Comparing Trailing and Copying for Constraint Programming. In *1999 International Conference on Logic Programming (ICLP 99)*, Danny De Schreye (Ed.). The MIT Press, Las Cruces, NM, USA(Nov.), 275–289.

Christian Schulte. 2000a. *Programming Constraint Inference Services*. Ph.D. Dissertation. Universität des Saarlandes, Fachbereich Informatik, Saarbrücken, Germany.

Christian Schulte. 2000b. Programming Deep Concurrent Constraint Combinators. In *Practical Aspects of Declarative Languages, Second International Workshop, PADL 2000 (Lecture Notes in Computer Science)*, Enrico Pontelli and Vítor Santos Costa (Eds.), Vol. 1753. Springer Verlag, Boston, MA, USA(Jan.), 215–229.

Christian Schulte. 2002. *Programming Constraint Services: High-Level Programming of Standard and New Constraint Services*. Lecture Notes in Artificial Intelligence, Vol. 2302. Springer Verlag, Berlin, Germany. Ph.D. dissertation, Universität des Saarlandes.

Christian Schulte and Gert Smolka. 1994. Encapsulated Search for Higher-order Concurrent Constraint Programming. In *International Symposium on Logic Programming*, Maurice Bruynooghe (Ed.). The MIT Press, Ithaca, NY, USA(Nov.), 505–520.

Christian Schulte, Gert Smolka, and Jörg Würtz. 1994. Encapsulated Search and Constraint Programming in Oz. In *Second Workshop on Principles and Practice of Constraint Programming (Lecture Notes in Computer Science)*, Alan H. Borning (Ed.), Vol. 874. Springer Verlag, Orcas Island, WA, USA(May), 134–150.

Ehud Shapiro. 1983. *A subset of Concurrent Prolog and its Interpreter*. Technical Report TR-003. Institute for New Generation Computer Technology (ICOT), Cambridge, Mass.(Jan.).

Ehud Shapiro (Ed.). 1987. *Concurrent Prolog: Collected Papers*. Vol. 1-2. The MIT Press, Cambridge, Mass.

Ehud Shapiro. 1989. The Family of Concurrent Logic Programming Languages. *Comput. Surveys* 21, 3 (Sept.), 413–510.

Jeremy G. Siek and Walid Taha. 2006. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop (SFP)*. (Sept.), 81–92.

Jörg Siekmann, Christoph Benzmüller, and Serge Autexier. 2006. Computer Supported Mathematics with ΩMEGA. *Journal of Applied Logic* 4, 533–559.

Gert Smolka. 1995a. The Definition of Kernel Oz. In *Constraints: Basics and Trends*. Lecture Notes in Computer Science, Vol. 910. Springer Verlag, 251–292. Also DFKI Technical Report 1994.

Gert Smolka. 1995b. The Oz Programming Model. In *Computer Science Today*. Lecture Notes in Computer Science, Vol. 1000. Springer Verlag, 324–343.

Gert Smolka and Ralf Treinen. 1994. Records for Logic Programming. *Journal of Logic Programming* 18, 3 (April), 229–258.

Ian Sommerville. 1992. *Software Engineering*. Addison-Wesley.

Alfred Spiessens, Raphaël Collet, and Peter Van Roy. 2003. Declarative Laziness in a Concurrent Constraint Language. In *2nd International Workshop on Multiparadigm Constraint Programming Languages (MultiCPL 2003)*. Kinsale, Ireland(Sept.). Colocated with 9th International Conference on Principles and Practice of Constraint Programming (CP2003).

Leon Sterling and Ehud Shapiro. 1986. *The Art of Prolog–Advanced Programming Techniques*. The MIT Press.

Ken Thompson. 1987. Reflections on Trusting Trust. See [Ashenhurst and Graham 1987]. 1983 Turing Award Lecture.

Evan Tick. 1995. The Deevolution of Concurrent Logic Programming. *Journal of Logic Programming* 23, 2 (May), 89–123.

Kazunori Ueda. 1985. Guarded Horn Clauses. In *4th International Conference on Logic Programming '85 (Lecture Notes in Computer Science)*, Eiti Wada (Ed.), Vol. 221. Springer Verlag, Tokyo, Japan(July), 168–179.

Kazunori Ueda and Takashi Chikayama. 1990. Design of the Kernel Language for the Parallel Inference Machine. *Comput. J.* 33, 6, 494–500.

Pascal Van Hentenryck. 1994. Constraint logic programming. *The Knowledge Engineering Review* 6, 3 (Sept.), 151–194.

Peter Van Roy. 1994. 1983–1993: The Wonder Years of Sequential Prolog Implementation. *Journal of Logic Programming* 19/20 (May/July), 385–441. Also Digital PRL Research Report 36, Dec. 1993.

Peter Van Roy. 1999. On the separation of concerns in distributed programming: Application to distribution structure and fault tolerance in Mozart. In *International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications (PDSIA 99)*. Tohoku University, Sendai, Japan(July).

Peter Van Roy (Ed.). 2004. *2nd International Conference on Multiparadigm Programming in Mozart/Oz (MOZ2004)*. Lecture Notes in Computer Science, Vol. 3389. Springer Verlag, Charleroi, Belgium(Oct.).

Peter Van Roy. 2006. Convergence in Language Design: A Case of Lightning Striking Four Times in the Same Place. In *8th International Symposium on Functional and Logic Programming (FLOPS 2006) (Lecture Notes in Computer Science)*, Vol. 3945. Springer Verlag(April), 2–12.

Peter Van Roy. 2009. *Programming Paradigms for Dummies: What Every Programmer Should Know.* IRCAM/Delatour France(June).

Peter Van Roy. 2011. *The CTM Approach for Teaching and Learning Programming*. Vol. 2. Nova Science Publishers(Jan.), Chapter 5, 101–126.

Peter Van Roy. 2018. A Software System Should be Declarative Except Where it Interacts with the Real World. In *Workshop on Logic and Practice of Programming (LPOP 2018)*. Oxford, UK(July). Colocated with FLoC 2018.

Peter Van Roy, Joe Armstrong, Matthew Flatt, and Boris Magnusson. 2003a. The Role of Language Paradigms in Teaching Programming. In *34th Technical Symposium on Computer Science Education (SIGCSE 2003)*. (Feb.).

Peter Van Roy, Per Brand, Denys Duchier, Seif Haridi, Martin Henz, and Christian Schulte. 2003b. Logic Programming in the Context of Multiparadigm Programming: The Oz Experience. *Journal of Theory and Practice of Logic Programming* (Nov.), 715–763.

Peter Van Roy and Seif Haridi. 1999. Mozart: A Programming System for Agent Applications. *AgentLink News* 4 (Nov.). Esprit Network of Excellence for Agent-Based Computing.

Peter Van Roy and Seif Haridi. 2004. *Concepts, Techniques, and Models of Computer Programming*. The MIT Press.

Peter Van Roy, Seif Haridi, Per Brand, Gert Smolka, Michael Mehl, and Ralf Scheidhauer. 1997. Mobile Objects in Distributed Oz. *ACM Transactions on Programming Languages and Systems* 19, 5 (Sept.), 804–851.

Peter Van Roy, Michael Mehl, and Ralf Scheidhauer. 1996. Integrating Efficient Records into Concurrent Constraint Programming. In *8th International Symposium on Programming Languages, Implementations, Logics, and Programs (PLILP '96)*. Springer Verlag, Aachen, Germany(Sept.).

David H. D. Warren. 1977. *Applied Logic–Its Use and Implementation as a Programming Tool.* Ph.D. Dissertation. University of Edinburgh. DAI Research Reports 39 & 40, Also reprinted as Technical Note 290, SRI International, 1983.

David H. D. Warren. 1983. *An Abstract Prolog Instruction Set.* Technical Report. (Oct.). Technical Note 309, SRI International Artificial Intelligence Center.

NON-ARCHIVAL REFERENCES

ACCLAIM. 1992–1995. Advanced Concurrent Constraint Languages: Application, Implementation, and Methodology. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.51.1542&rep=rep1&type=pdf Esprit Basic Research project (PE7195).

Mats Carlsson et al. 2001–2020. SICStus Prolog. https://sicstus.sics.se/ Research Institutes of Sweden.

Jean-Luc Cochard (Ed.). 1995. *WOz'95, International Workshop On Oz Programming*. Institut Dalle Molle d'Intelligence Artificielle Perceptive (IDIAP), CP 592, CH-1920 Martigny(Nov).

Seif Haridi. 1994. Development of Concurrent Constraint Languages for Multiparadigm Programming. (Dec.). https://www.info.ucl.ac.be/~pvr/acclaim-95.pdf Invited talk, Fifth Generation Computer Systems (FGCS 1994).

Seif Haridi. 1996. *A Tutorial of Oz 2.0.* Technical Report. Swedish Institute of Computer Science(Nov.). https://www.info.ucl.ac.be/~pvr/Tutorial_of_Oz_2.pdf

Seif Haridi, Per Brand, Nils Franzén, and Erik Klintskog. 1996. PERDIO: Persistence and Distributed Programming Systems(Feb.). Nutek project proposal, Sweden.

Yves Jaradin and Sébastien Doeraene. 2013. Mozart VM 2.0: Design of the New Virtual Machine of Mozart, An Implementation of Oz. (April). https://www.info.ucl.ac.be/~pvr/mozart2-design.pdf Technical presentation.

Mozart Consortium. 2018. The Mozart Programming System (Oz 3), Version 2. (Sept.). http://mozart2.org/

Peter Norvig. 2006–2020. Teach Yourself Programming in Ten Years. https://norvig.com/21-days.html

PIRATES. 1997–2003. Méthodes et Outils pour la Programmation Répartie Transparente et Sûre (Methods and Tools for Dependable Transparent Distributed Programming). Wallonia Region (Belgium) project (Convention 9713540).

Christian Schulte, Guido Tack, and Mikael Z. Lagerkvist. 2019. Modeling and Programming with Gecode. (May). https://www.gecode.org/doc-latest/MPG.pdf This document will always correspond to the latest Gecode version, which is available at https://www.gecode.org.

SELFMAN. 2006–2009. Self Management for Large-Scale Distributed Systems based on Structured Overlay Networks and Components. http://www.ist-selfman.org European Sixth Framework Programme project (Number 034084).

Gert Smolka, Christian Schulte, and Peter Van Roy. 1995. PERDIO—Persistent and Distributed Programming in Oz(March). BMBF project proposal, Germany.

Peter Van Roy. 2008. Programming Paradigms Poster. https://www.info.ucl.ac.be/~pvr/paradigms.html

Peter Van Roy, Seif Haridi, and Per Brand. 1999. Distributed Programming in Mozart – A Tutorial Introduction. http://mozart2.org/mozart-v1/doc-1.4.0/dstutorial/index.html Part of Mozart 1 documentation.
Wikipedia. 2020a. Clarke's Three Laws. https://en.wikipedia.org
Wikipedia. 2020b. Fifth Generation Computer. https://en.wikipedia.org