Multiparadigm Programming in Oz

Martin Müller, Tobias Müller, and Peter Van Roy

RR-95-16
Research Report

Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

Research
Report
RR-95-16

Multiparadigm Programming in Oz

Martin Müller, Tobias Müller, and Peter Van Roy

7 December 1995

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Postfach 20 80
67608 Kaiserslautern, FRG
Tel.: + 49 (631) 205-3211
Fax: + 49 (631) 205-3210

Stuhlsatzenhausweg 3
66123 Saarbrücken, FRG
Tel.: + 49 (681) 302-5252
Fax: + 49 (681) 302-5341

Multiparadigm Programming in Oz

Martin Müller, Tobias Müller, and Peter Van Roy

DFKI-RR-95-16

This paper has been presented at the workshop “Visions for the Future of Logic Programming: Laying the Foundations for a Modern Successor to Prolog”, in association with ILPS’95, December 7, Portland, Oregon.

This work has been supported by the Bundesminister für Forschung und Technologie (contract ITW 9105), the Basic Esprit Project ACCLAIM (contract EP 7195), and the Esprit Working Group CCL (contract EP 6028).

ISSN 0946-008X

Multiparadigm Programming in Oz

Martin Müller, Tobias Müller, Peter Van Roy
Programming Systems Lab
German Research Center for Artificial Intelligence (DFKI)
Stuhlsatzenhausweg 3, 66123 Saarbrücken, Germany
email: {`mmueller,tmueller,vanroy`}`@dfki.uni-sb.de`

November 29, 1995

Abstract

The foundation of Prolog's success is the high abstraction level of its declarative subset, namely first-order Horn clause logic with SLDNF resolution. What's missing from Prolog is that little attempt is made to give the same foundation to anything *outside* the declarative subset. We argue that multiparadigm programming can remedy this lack. We give a foundation for multiparadigm programming and we show how it is realized in the Oz language and system. Oz naturally encompasses multiple styles of programming, including (constraint) logic, functional, and concurrent object-oriented programming, by providing a common underlying foundation for these styles through a simple formal model in the concurrent constraint paradigm. We illustrate the integrative programming model with paradigmatical code examples.

Contents

1 Introduction

> Six blind sages were shown an elephant and met to discuss their experience. "It's wonderful," said the first, "an elephant is like a snake: slender and flexible." "No, no, not at all," said the second, "an elephant is like a tree: sturdily planted on the ground." "Marvelous," said the third, "an elephant is like a wall." "Incredible," said the fourth, "an elephant is a tube filled with water." "What is then this mythical beast that is all things to all?," said the fifth. "Hell if I know," said the sixth, "but let us now disperse and proclaim our wisdom."

Programming languages are a means to describe computational behavior. Different languages may rely on very different credos or *paradigms* as to what computation is all about. A rough classification of paradigms distinguishes between stateful and stateless programming. Stateful programming explicitly represents data that change over time. Stateless programming represents data that can be created but that never change. The distinction is useful because stateful programming is closer to the way the external world works, whereas stateless programming is much simpler to reason about.

Stateful programming has evolved from traditional imperative programming into object-oriented programming. Among the stateless paradigms (by and large coinciding with the so-called declarative ones) one can distinguish between directed and undirected computation. The former has been made most popular by functional programming languages, the latter by logic languages.

1.1 Why Multiparadigm Programming?

Multiparadigm programming is *not* feature stacking as in PL/I or C++. Multiparadigm programming is the integration of several programming paradigms in a *simple* model. From a foundational point of view, this allows to understand various forms of computation as facets of a single phenomenon. Multiparadigm programming eases the coding of algorithms in a "natural" style. This may be the functional, the constraint, or the object-oriented paradigm, depending on whether just functions are evaluated, partial information is used, or state manipulation is necessary.

From a systems point of view, using a single language avoids a proliferation of interfaces between components addressing different concerns. In a distributed setting, using a single language allows the communication of arbitrary data structures. The set of data structures which can be transferred between different languages is usually greatly restricted.

Finally, multiparadigm languages are advantageous for the teaching of algorithms. They introduce only those new concepts necessary for each language and they keep syntax changes minimal when different classes of algorithms are considered. The language Leda [3], which supports multiple paradigms, was strongly motivated by this educational argument.

1.2 Programming Paradigms

Let us look more closely to the specific contributions of the mentioned paradigms to the field of programming.

Logic programming (e.g., Prolog [45]) was devised as realization of computation as *deduction*. Its major contributions to programming language design are logic variables and search. More generally speaking, it pioneered speculative computation with partial information, which subsequently was generalized into constraint programming.

Functional programming (e.g., Haskell [16]) rests on the idea that computation is the (referentially transparent) *evaluation* of expressions. Among its contributions are functions as first-class citizens, full compositionality, lexical scoping, and the development of type systems for computing.

Object-oriented programming (e.g., Smalltalk [12]) organizes computational activities into entities called *objects* which encapsulate state and methods to manipulate it. Often, inheritance is supported as a means for incremental development and code reuse.[1] Objects are a powerful concept to structure stateful (e.g., imperative) computation, which is particularly important in applications that react with the external world.

The choice of a programming paradigm depends on the particular task to be solved. It strongly influences what a solution to a problem looks like. Much of programming proficiency in a given language is the ability to cast a program in this language's paradigm. The typical applications for a language are those which can be expressed in it with the least effort. Other problems may resist an elegant or concise realization (e.g., procedural abstraction in Prolog or imperative data structures in stateless paradigms), or enforce the use of concepts which complicate reasoning about the program (e.g., to consider arithmetics as an object interaction instead of function evaluation).

1.3 How Multiparadigm Programming can Save Prolog

Prolog maintains a stable niche position in industry for applications based on rapidly changing, structured data. Typical examples are natural language, program generation, expert systems, data transformation, and theorem proving. Prolog has many advantages for such applications. The bulk of programming can be done cleanly in its pure declarative subset. The code is compact due to the expressiveness of unification and the term notation. Memory management is dynamic and implicit. Efficient, robust, and interoperable implementations exist. Primitives exist for useful non-declarative operations, e.g., call/1 provides a form of higher-orderness (i.e., first-class procedures) and setof/3 provides a form of encapsulated search.

[1] Languages that support objects but no inheritance are usually called *object-based*.

The foundation of Prolog's success is the high abstraction level of its declarative subset, namely first-order Horn clause logic with SLDNF resolution. What's missing from Prolog is that little attempt is made to give the same foundation to anything *outside* the declarative subset. Two decades of research have resulted in a solid understanding of the declarative subset and only a partial understanding of the rest.[2] This results in two main flaws of Prolog: First, the operational aspects are too deeply intertwined with the declarative. The control is naive (depth-first search) and eager. The interactive top level has a special status: it is lazy and is not accessible to programs. Second, to express anything beyond the declarative subset requires ad hoc primitives that are limited and do not always do the right thing. Freeze/2 provides coroutining as a limited form of concurrency. Call/1 and setof/3 provide a limited form of higher-orderness [47].

Multiparadigm programming can remedy these defects. Its goal is to provide a firm foundation for *all* facets of computation, not just the declarative subset. The semantics should be fully defined and bring the operational aspects out into the open. For example, encapsulating search gives first-class Prolog top levels, each with its own user-programmable search strategy. True higher-orderness results in compact, modular programs. Concurrency and stateful execution make it easy to write programs that interact with the external world.

In the rest of this paper we give a foundation for multiparadigm programming and we show how it is realized in the Oz language and system. Oz is fully defined and has an efficient implementation competitive with Prolog systems. Oz has much in common with Prolog, but it is not a superset of Prolog. Oz does not have the reflective syntax of Prolog, nor does it have the meta-programming facilities (like call/1, assert/1) or the user-definable syntax (operator declarations). Like Prolog, Oz has a declarative subset. Like Prolog, Oz has been generalized to arbitrary constraint systems (currently implemented are finite domains and open feature structures). Oz provides a clean implementation of logic programming that can be used for many of the tasks for which Prolog and CLP are used today.

1.4 Plan of the Paper

Section 2 introduces the programming model underlying Oz. Section 3 quickly surveys related approaches to the integration of programming paradigms. Section 4 illustrates programming in Oz by paradigmatical code examples in the functional, logic, and object-oriented styles. Section 5 presents the Oz Programming Model in an informal manner and relates its components to the examples discussed earlier.

2 A Uniform Model for Multiparadigm Programming

Our approach to multiparadigm programming is to start with concurrent constraint programming (ccp) as a foundation. Concurrency is important because it is the most general

[2] The non-declarative aspect has received some attention, e.g., [27, 19, 34, 2].

form of control and can easily be restricted to the requirements of any programming style. Constraints are important because they provide a simple and uniform way to express communication and synchronization. The ccp model [23, 37] combines ideas from research in concurrent [39] and constraint logic programming [20], as well as from CCS [24]:

Concurrent constraint programming consists of primitive concurrent agents which communicate and synchronize through a monotonically growing constraint store. It relies on Maher's characterization of synchronization as logical entailment between constraints.

We extend ccp with higher-order procedures (i.e., first-class procedures). The resulting higher-order ccp model is simple and theoretically clean. Its restriction to the pure functional paradigm recovers nice formal properties such as confluence [28], both in the lazy and eager cases. For a deeper introduction, see the Oz programming model [41] and the calculi underlying Oz [40, 29, 28]. A closely related model of concurrency is the π calculus [25]. See [28] for a formal comparison between the two models.

The higher-order ccp core is minimally extended with state and search (see Section 5). The programming style arising from the resulting model can be sketched as follows: Computation is organized as a network of concurrent objects which react with each other and with the external world (I/O). Inside the objects, functions and predicates are used to process stateless knowledge in algorithmic or declarative (constraint) form.

3 Related Work

Integration of paradigms is a very active area of research and has produced a variety of different languages. The development of Oz has benefited from this fertile atmosphere. This section can only take a glimpse into the field.

Most of the research on integration has originated from within the stateless paradigm. There has been very little research originating from within the stateful paradigm to integrate stateless programming. Typed imperative languages (e.g., Algol, Eiffel, or C++) may be viewed as an exception since types add a level of stateless semantics to a possibly stateful program. They do not, however, provide for stateless *programming*.

A short-term solution to integrate different paradigms is to use a *coordination model* [4, 5]. The paradigmatic coordination model is Linda, which provides a uniform global tuple space that can be accessed with a small set of basic operations from any process that is connected to it. Linda layers have been added to languages of various paradigms. Only primitive data objects (numbers and strings) can be put in the tuple space.

The easiest way to allow imperative programming in a functional framework is to add locations with destructive assignment. This route was taken by languages such as Lisp [44], Scheme [7], and SML [15]. Also the M-structures of Id [30] and its successor pH [31] fall in this category. In Oz, the location primitive is the cell, introduced to provide state for the object system. In contrast to the mentioned languages, stateful programs in Oz never use cells directly but only the higher-level concept of objects.

In Haskell, state is integrated using the monadic style of programming [46, 33] which generalizes the continuation-passing style. The monadic style allows to control the sequentialization necessary for various kinds of side effecting (I/O, error handling, non-deterministic choice).

Concurrent logic programming has investigated in depth the use of logic variables for synchronization and communication. Since logic variables are constrained in an incremental manner, they allow one to express monotonic synchronization and hence determinism in a concurrent setting. The concurrent logic language Strand has evolved into the coordination language PCN [10] for imperative languages. In the functional programming community, the I-structures of Id [30] realize a restricted form of logic variables. Further, the Goffin project [6] uses a first-order concurrent constraint language as a coordination language for Haskell processes.

A constraint logic language with a concurrent flavor is LIFE [1]. It provides automatic delaying of functions and "demons" that wait until data structures have a particular form. LIFE is designed around record constraints (called ψ-terms) that live in a stateless inheritance hierarchy. They are especially useful for knowledge representation and natural language processing. DFKI Oz supports record constraints much in the style of LIFE.

There have been many proposals to incorporate an object system into (concurrent) logic languages [9]. These proposals remain unsatisfactory due to the absence of first-class procedures and mutable state in these languages.

The logic language Gödel [19] aims at being a more declarative Prolog. It provides for a strong type system and a module system. The control facilities provided by Gödel allow for coroutining, a pruning (or commit) operator related to parallel conditionals in concurrent Prolog, and constraint solving capabilities in the domains of integers and rationals.

Also in the logic programming tradition, the design rationale of Lambda Prolog [26] is to develop more powerful logic than Horn logic as a basis for programming. In particular, functional programming is supported by providing λ terms as data structures, which are handled by higher-order unification. In contrast, Oz combines first-class procedures with *first-order* constraints via atomic constants (called *names*), which represent abstraction in the constraint solving process.

Another stream of research aims at integrating functions into first-order logic programming by extending resolution towards deterministic evaluation of functions (called *narrowing*) [13].

The language Leda [3] has been developed mainly for educational purposes. It makes different programming styles available in a single language but does not provide a simple operational model for this integration.

4 Examples

This section presents Oz programs exemplifying the logic, functional, and object-oriented style. Based on the Oz programming model, these styles smoothly integrate to yield, e.g., a first-order logic language, a higher-order constraint language, a concurrent object-oriented language, or a functional language. Since explanation of the examples has to be brief, we refer the reader to the Oz documentation [43]. The sample applications in the DFKI Oz release illustrate how paradigms mix in larger programs. For instance, a multi-agent transportation scenario is set up as a collection of concurrent objects which interact with each other while locally performing speculative computation.

The Oz programming model is defined in terms of a simple kernel language [42] (see Section 5). When translated into the kernel language, the different paradigms show up as different idioms. Since these tend to be verbose, Oz provides notational support for the most common idioms [17]. To increase readability, we present the examples in the Oz notation.

4.1 Higher-order Functional Programming

For a long period functional languages have been the first choice for symbolic computation. Therefore symbolic mathematics and parsing are among the typical applications. Consider the problem of encoding a text (list of characters) with respect to a Huffman tree. A Huffman tree T is an ordered binary rooted tree whose leaves carry pairwise distinct characters, the *domain* of T.

```
fun {CEncode T C}
   case T
   of n(L R) then
      case {CEncode L C} of y(P) then y(1|P)
      elsecase {CEncode R C} of y(P) then y(0|P)
      else n end
   elseof l(D) then case D==C then y(nil) else n end
   else n end
end
```

Given a Huffman tree T, the function application `{CEncode T C}` maps the character C from the domain of T to the unique path of the leaf carrying C. The function `CEncode` assumes binary trees to be encoded with constructors `n/2` for nodes and `l/1` for leaves. An application `{CEncode T C}` will return `y(P)` where `P` is the path of `C` in `T` if `C` lies in the domain of `T`. Otherwise, `{CEncode T C}` will return `n`.

Functions are not primitives of Kernel Oz. Rather, the basic form of first-class procedures is relational. All n-ary functions (using keyword **`fun`**) are syntactically expanded to n+1-ary relations (using keyword **`proc`**) (see Section 5). Oz supports different conditionals as derived from the basic guarded conditional **`if`**`...`**`then`**`...`**`else`**`...`**`fi`** (see Section 5). The **`case`** statement is a notational variant that can be used as a conditional expecting a boolean condition. There is an optional **else** branch which is selected if none of the patterns match.

```
fun {Encode T}
   fun {CAppend C Cs}
     case {CEncode T C} of y(P) then {Append P Cs} else Cs end
   end
   fun {E Cs} {FoldR Cs CAppend nil} end
in E end
```

The function `Encode` takes a tree `T` and returns the function `E` which maps a string of characters `Cs` to its encoding with respect to `T`. This is done by appending all single character encodings using a `FoldR`, where the function `CAppend` strips the bookkeeping information `y` or `n` (and simply ignores illegal characters).

Assuming some Huffman tree over the alphabet of vowels, the following expression will bind `E` to its corresponding encoding function:

```
declare E={Encode n(n(l(a) l(e)) n(n(l(u) l(i)) l(o)))}
```

Applying the function `E` to an input list as in `C={E [a e i o u]}` binds `C` to its Huffman code `[1 1 1 0 0 1 0 0 0 0 1 1]`.

4.2 Logic Programming

A driving motivation for the development of Prolog was grammars for natural language [8]. Logic grammars are also easily written in Oz. In Oz, the first-class procedures, concurrency, and local computation spaces add much expressiveness to what is easily expressed in Prolog. For example, an HPSG (head-driven phrase structure grammar) parser can be written declaratively and executed efficiently. The DFKI Oz release comes with an example HPSG parser. For simplicity, we consider a grammar for Lisp s-expressions. A code segment, using Prolog's DCG (Definite Clause Grammar) notation, is given below. It defines an attribute grammar that can build an s-expression from a list of tokens. This notation has a straightforward translation into standard Prolog. We assume that the notation is close enough to standard BNF to be understandable even with no Prolog knowledge.

```
main(S) :- sexpr(S, ['(','(',c,d,')',e,')'], []). % ((c d) e)

sexpr(S) --> atom(S).
sexpr([U|V]) --> ['('], sexpr(U), sexpr_star(V), [')'].

sexpr_star([]) --> [].
sexpr_star([U|V]) --> sexpr(U), sexpr_star(V).

atom(S) --> [S], {S=c; S=d; S=e}.
```

To be as clear as possible about what is actually executed, we present the above grammar as a self-contained Oz program that does not need a preprocessor. The procedure `{SExpr S T1 T0}` is a predicate that defines the relation between an s-expression `S` as a tree and the token list `T1` that represents it. `T0` is the remaining list of tokens, i.e., which are not consumed by `SExpr`. We show that this program can be used not only as a parser but also as a generator of token lists.

```
proc {SExpr S T1 T0}
   or {Atom S T1 T0}
   [] T2 T3 T4 U V in T1=´(´|T2 S=U|V
      then {SExpr U T2 T3} {SExprStar V T3 T4} T4=´)´|T0
   ro
end

proc {SExprStar S T1 T0}
   or T1=T0 S=nil
   [] U V T2 in S=U|V then {SExpr U T1 T2} {SExprStar V T2 T0}
   ro
end

proc {Atom S T1 T0}
   T1=S|T0 or S=c [] S=d [] S=e ro
end
```

The `or...[]...ro` construct is a logical disjunction (see Section 5). Each branch of the disjunction corresponds to a Horn clause and has a guard consisting of simple constraints such as `T1=T0`, `S=nil`, and `S=U|V`. The execution of a disjunction follows the so-called *Andorra principle* [36, 14], suitably extended to concurrency and encapsulation. A disjunction reduces if it is *deterministic*, i.e., if all branches but one have failing guards. The last remaining branch is then executed. Execution continues until all disjunctions that remain have more than one non-failing guard. This is able to find a solution in all cases in which Prolog does so, and also in many cases where Prolog would deadlock or go into an infinite loop. However, this strategy is not *complete*, i.e., it does not always find a solution when one exists. To make it complete, it is necessary to add a search primitive to the computation model.

The procedure `SExpr` specifies the grammar's constraints without implying any particular search strategy. With the appropriate strategy, `SExpr` can be used both as a parser and as a generator. This can be done with one of the predefined search procedures in the module `Search`, which implement various search strategies using the `Solve` combinator (see Section 5). The use of `Search` illustrates the separation of logic and control in Oz. The function `Parse` parses a given token list, using a first-solution search with depth-first strategy:[3]

[3] For generality, solutions are returned as procedures wrapped in a tuple. To actually access a solution, the procedure has to be selected and applied.

```
fun {Parse T}
   {Search.one.depth proc {$ S} {SExpr S T nil} end}
end

S={Parse ['(' '(' c d ')'  e ')']} {Browse {S.1}}    % ((c d) e)
```

The browser `Browse` is a concurrent tool used to inspect Oz data structures [35]. The function `Generate` lazily generates pairs of parses and their token lists using an all-solution search. Laziness means that solutions are generated on demand, which is expressed by incrementally constraining the output to a list.

```
fun {Generate}
   {Search.all.lazy proc {$ Sol} S T in Sol=S#T {SExpr S T nil} end}
end

{Generate}=S1|_|_|_|_|S2|_ {Browse s({S1.1} {S2.1})}    % s(c#[c] [e]#['(' e
')'])
```

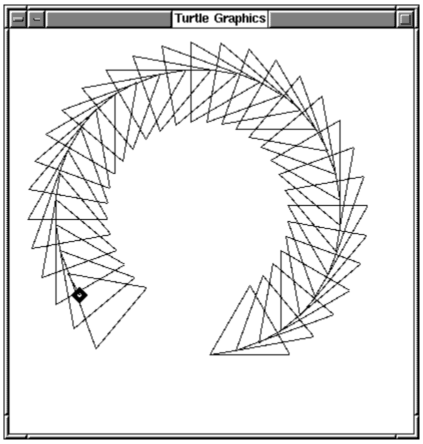
This example illustrates that logic programming in Oz is based on the notion of *encapsulated search*, which was originated in the AKL language [21].

4.3 Object-oriented Programming

A standard application for object-oriented languages from their beginning is window programming. The Smalltalk programming environment [22] was one of the earliest successful window systems. Inspired by Logo's turtlegraphics [32] and Smalltalk's Pen class, we sketch a turtle class in Oz. Note that Oz objects are based completely on a single state primitive (see Section 5). The full concurrent object system of Oz is defined in terms of higher-order ccp with state [18].

```
class Turtle from Tk.canvasTag
   attr x y ang:0.0 pen:True size:400 canvas
   meth init
      x<-(@size div 2)  y<-(@size div 2)  <<display>>
   end
   meth up          pen <- False end
   meth down        pen <- True  end
   meth left(A)     ang <- @ang-{ToRad A} end
   meth right(A)    ang <- @ang+{ToRad A} end
   meth forward(D)
      DX={F2I {I2F D}*{Cos @ang}}  DY={F2I {I2F D}*{Sin @ang}}
   in
      case @pen then <<line(DX DY)>> else <<move(DX DY)>> end
   end
   meth move(DX DY) <<tk(move DX DY)>> end
```

```
    %% ... further methods (display, line) left out
end
```



DFKI Oz provides an object-oriented interface to the Tcl/Tk graphics package. This interface provides a class `Tk.canvasTag` from which the class `Turtle` inherits. Instances of the `Turtle` class have as *attributes* (instance variables) the coordinates `x` and `y`, an angle `ang`, and a flag `pen`. Turtles move and turn on the screen on receipt of messages and leave a trail if the `pen` flag is set. The `init` method places a turtle in the center of its associated `canvas` and applies its method `display` for Tk interfacing. Methods `up` and `down` set the `pen` flag. Methods `left(A)` and `right(A)` take an argument `A` in degrees and update the current angle `ang` accordingly. Method `forward(D)` moves the turtle forward by `D` units. The explicit coercions `I2F` and `F2I` are used to convert between integers and floats.

Modeling Objects in Kernel Oz. We now sketch how Oz objects are modeled by the kernel language. Objects are procedures that reference a *cell* whose content designates the current state. A cell is the primitive that incorporates stateful computation into Oz. Cells may be seen as locations which can be atomically updated (see Section 5).

The object's state is a record whose fields are the object's attributes. Lexical scoping guarantees encapsulation of the state. After executing a class definition as for `Turtle`, the predefined procedure `New` can be used like `{New Turtle init T}` to create an instance `T` of `Turtle` and send it the initial message `init`. Since objects are procedures, *message sending* amounts to application, for example:

```
{T forward(50)} {T left(90)}
```

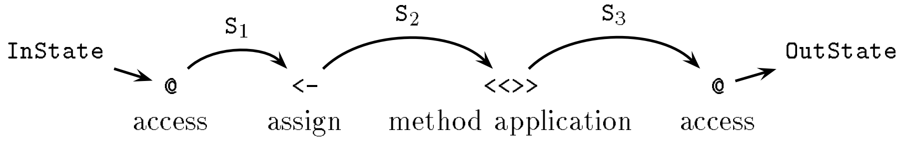
At any point in time, the *current state* of an object is designated by the logic variable in its cell. On receipt of a message like `forward(50)`, the cell is updated to hold a fresh logic variable `OutState` and the method selected by the label of the message (`forward`) is scheduled to compute the new state, which is put in `OutState`:

```
case {DetB InState} then {Method InState Msg Object OutState} end
```

This conditional defers the execution of the method until the object's current state is available (i.e., until the `DetB` test succeeds). This setup results in a queue of messages waiting for mutually-exclusive access to the object's state. Note that messages are enqueued even if the current state of the object is not yet available, i.e., if its cell holds an unbound variable. A message queue may be depicted as:

$$s_1 \xrightarrow{m_1} s_2 \xrightarrow{m_2} s_3 \xrightarrow{m_3} s_4$$

where the circled states have become available, s_4 is the current state, and methods m_1 through m_3 have been scheduled to compute the state sequence s_1 through s_4. Note that the methods may execute concurrently. *Methods* are modeled as procedures `{Method InState Msg Object OutState}` computing an output state from an input state, a message, and the object invoking the message (the self reference). The body of a method *threads* the input state through attribute accesses, assignments and method applications to ensure the correct sequentialization of state accesses in a concurrent context:



Attribute access, assignment and method application is modeled as application of the procedures `@`, `<-` and `<<>>`, accordingly, to an intermediate state. E.g., `{'<-' InState A V OutState}` waits until `InState` is determined, checks `InState` for the field `A` and then computes `OutState` by replacing `InState`'s value at `A` by `V`. By using the expression `<<display>>` inside a method body, the method `display` is called without relinquishing the exclusive access to the state. This is realized by threading the intermediate state through the body of the called method, in the same way as for state accesses.

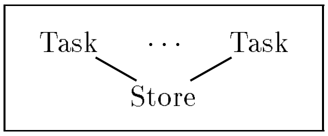
5 The Oz Programming Model

This section briefly summarizes the ingredients of the Oz Programming Model OPM with its extension by encapsulated search, and relates them to the programming abstractions above [41]. For more background, see [40, 42, 18, 38]. The paradigms are compared on the calculus level in [29, 28].

Concurrent Constraints. The core of OPM is similar to Saraswat's concurrent constraint model [37], providing for concurrent control and synchronization via logic variables.

$$\begin{array}{lcl} E & ::= & \mathbf{local}\ x\ \mathbf{in}\ E\ \mathbf{end} \mid E \wedge F \mid C \mid \mathbf{if}\ C\ \mathbf{then}\ E\ \mathbf{else}\ F\ \mathbf{fi} \\ C & ::= & x = y \mid x = f(\overline{y}) \mid \ldots \end{array}$$

Computation in OPM takes place in a *computation space* hosting a number of *tasks* (also called actors) connected to a shared *store*. Computation advances by *reduction of tasks* which can manipulate the store and create new tasks. When a task is reduced it disappears. As computation proceeds, the store accumulates constraints on variables in the *constraint store*.



Concurrent composition $E \wedge F$ and *declaration* **local** x **in** E **end** provide the glue for tasks E and F. Reduction of $E \wedge F$ yields two tasks E and F, and reduction of **local** x **in** E **end** generates a fresh variable y and reduces to $E[y/x]$. Reduction of a *constraint* C (*tell*) advances the constraint store S to the logical conjunction $S \wedge C$, provided $S \wedge C$ is consistent. Otherwise, the constraint store is not changed and *failure* is announced.

Conditional tasks are the observers of the constraint store (*ask*). Telling a constraint communicates it to all tasks, while asking means to synchronize on the constraint in the store. Assuming C in the constraint store, the conditional **`if`** D **`then`** E **`else`** F **`fi`** reduces to E if C logically entails D, and to F if C and D are inconsistent. Otherwise, the conditional cannot reduce.

Higher-Order Abstraction. Procedures in Oz are used to model functions, predicates, methods, and objects. *Procedures* in OPM are triples of the form $\xi{:}x/E$ consisting of a *name* ξ, *formal argument* x and a body E.[4] Procedures may be defined and applied as follows:

$$E \quad ::= \quad \ldots \mid \mathbf{proc}\ \{x\ y\}\ E\ \mathbf{end} \mid \{x\ y\}$$

Reduction of a definition **`proc`** `{`x y`}` E **`end`** binds the variable x to a freshly created name ξ and enters the procedure $\xi{:}y/E$ into the *procedure store*. An application task `{`x z`}` must wait until the procedure store contains a procedure $\xi{:}y/E$ such that the constraint store entails $x = \xi$. Then, it can reduce to the task $E[z/y]$, which is obtained from the procedure body E by replacing the actual argument z for the formal one y.

State. OPM supports state by means of *cells* $\xi{:}x$. A cell represents a mutable binding of a name to a variable, i.e., a location. There are two primitives for creation of and operation on cells:

$$E \quad ::= \quad \ldots \mid \{\texttt{NewCell}\ x\ y\} \mid \{\texttt{Exchange}\ x\ y\ z\}$$

Cells are created by reducing a task of the form `{NewCell` x y`}`. Similar to procedure definition, reduction of this task picks a fresh name ξ, binds x to ξ and enters the cell $\xi{:}y$ into a third compartment of OPM's store, the *cell store*.

Cell store observers are tasks of the form `{Exchange` x y z`}` which wait until the cell store contains a cell $\xi{:}u$ such that the constraint store entails $x = \xi$. In that case, the cell binding is updated to $\xi{:}y$ and the equation $z = u$ is entered into (told to) the constraint store.

Cells plus higher-order procedures are sufficient to give an operational explanation of a powerful object system [40, 18].

Constraint Programming and Search. OPM has been extended to subsume full constraint logic programming. This is done by means of a non-deterministic choice combinator and a solve combinator, which together allow to encapsulate search in a concurrent setting [38]:

$$E \quad ::= \quad \ldots \mid \mathbf{or}\ E\ [\,]\ F\ \mathbf{ro} \mid \{\texttt{Solve}\ x\ E\}$$

[4] The restriction to single arguments is for notational simplicity.

The task `{Solve` x E`}` opens a new, *local* computation space for reduction of E in search of solutions for x (roughly, in the predicate $E(x)$). Local computation proceeds just as global computation until the local space is either *failed* or *stable*. A computation space is called failed after reduction of a constraint which is inconsistent with the store. Failure is a run-time error unless in a local space where it signals the absence of solutions. Stability of a local space means that the computation has run to completion and that the global space cannot influence the local space. If a stable local space contains a non-deterministic *choice* **`or`** E `[]` F **`ro`**, it is *distributed* into two spaces where the choice is replaced by its right or left alternative, respectively. Both spaces are returned as procedures which can be explored with a suitable strategy.

6 Conclusion

We have presented and justified multiparadigm programming in Oz by means of programming examples and an introduction to the underlying formal model. There are two reasons why multiparadigm programming is important:

- In the theoretical development of computer science, all computational phenomena must be understood as aspects of a single underlying model. The design of Oz is an attempt to provide such a model.
- Independently of the theoretical importance of multiparadigm programming, we have to investigate whether it can be realized in a practical system and whether this provides advantages over single-paradigm languages such as Prolog. The DFKI Oz system is a realization of Oz that can be used for many of the tasks for which Prolog is used today.

Future research on Oz will be continued in two major directions: to use constraints (e.g., finite domains and feature structures) to solve real-world problems and to extend the model for transparent open distributed programming.

Acknowledgement

We thank the members of the Programming Systems Lab at DFKI. We thank Christian Schulte for his comments on this paper. The research reported in this paper has been supported by the Bundesminister für Bildung, Wissenschaft, Forschung und Technologie (FTZ-ITW-9105), the Esprit Project ACCLAIM (PE 7195), and the Esprit Working Group CCL (EP 6028). The DFKI Oz system is available through WWW at `http://ps-www.dfki.uni-sb.de/oz/` or through anonymous ftp from `ps-ftp.dfki.uni-sb.de`.

References

[1] Hassan Aït-Kaci and Andreas Podelski. Towards a meaning of LIFE. *Journal of Logic Programming*, 16:195–234, 1993.

[2] James Andrews. The logical semantics of the Prolog cut. In *ILPS 95*, December 1995.

[3] Timothy A. Budd. *Multi-Paradigm Programming in LEDA*. Addison-Wesley, Reading, MA, 1995.

[4] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

[5] Nicholas Carriero and David Gelernter. Coordination languages and their significance. *Communications of the ACM*, 35(2):96–107, February 1992.

[6] M. Chakravarty, Y. Guo, and M. Köhler. Goffin: Higher-Order Functions meet Concurrent Constraints. In *First International Workshop on Concurrent Constraint Programming*, Venice, Italy, May 29–31 1995.

[7] William Clinger and eds. Jonathan Rees. The Revised^4 Report on the Algorithmic Language Scheme. *LISP Pointers*, IV(3):1–55, July-September 1991.

[8] A. Colmerauer. The Birth of Prolog. In *The Second ACM-SIGPLAN History of Programming Languages Conference*, pages 37–52. ACM SIGPLAN Notices, March 1993.

[9] Andrew Davison. A Survey of Logic Programming-based Object Oriented Languages. In *Research Directions in Concurrent Object-Oriented Programming*. The MIT Press, Cambridge, MA, 1993.

[10] Ian Foster. Strand and PCN: Two Generations of Compositional Programming Languages. Preprint MCS-P354-0293, Argonne National Laboratories, 1993.

[11] D.M. Gabbay, C.J. Hogger, and J.A. Robinson, editors. *Handbook of Logic in Artificial Intelligence and Logic Programming*. Oxford University Press, Oxford, UK, 1995.

[12] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, MA, 1983.

[13] M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *The Journal of Logic Programming*, (19,20):583–628, 1994.

[14] Seif Haridi and Sverker Janson. Kernel Andorra Prolog and its computation model. In *7th International Conference on Logic Programming*, pages 31–48. MIT Press, 1990.

[15] Robert Harper, Dave MacQueen, and Robin Milner. Standard ML. Technical Report ECS-LFCS-86-2, Department of Computer Science, University of Edinburgh, 1986.

[16] Report on the Programming Language Haskell: A Non-Strict, Purely Functional Language. *ACM SIGPLAN Notices*, 27(5), May 1992.

[17] Martin Henz. The Oz notation. DFKI Oz documentation series, German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, 1994.

[18] Martin Henz, Gert Smolka, and Jörg Würtz. Object-oriented concurrent constraint programming in Oz. In V. Saraswat and P. Van Hentenryck, editors, *Principles and Practice of Constraint Programming*, chapter 2, pages 29–48. The MIT Press, Cambridge, MA, 1995.

[19] P. M. Hill and J. W. Lloyd. *The Gödel Programming Language*. The MIT Press, Cambridge, MA, 1994.

[20] Joxan Jaffar and Michael J. Maher. Constraint Logic Programming: A Survey. *The Journal of Logic Programming*, 19/20:503–582, May-July 1994.

[21] Sverker Janson. *AKL–A Multiparadigm Programming Language*. PhD thesis, Uppsala University, Box 1263, S-164 28 Kista, Sweden, June 1994.

[22] Wilf R. Lalonde and John R. Pugh. *Inside Smalltalk*, volume 1. Prentice-Hall, Englewood Cliffs, N.J., 1991.

[23] Michael J. Maher. Logic Semantics for a Class of Committed-Choice Programs. In Jean-Louis Lassez, editor, *International Conference on Logic Programming*, pages 858–876. The MIT Press, Cambridge, MA, 1987.

[24] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Germany, 1980.

[25] Robin Milner. The Polyadic π-Calculus: A Tutorial. In F. L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Proceedings of the 1991 Marktoberndorf Summer School on Logic and Algebra of Specification*, NATO ASI Series. Springer-Verlag, Berlin, Germany, 1993.

[26] G. Nadathur and D. Miller. Higher-Order Logic Programming. In Gabbay et al. [11].

[27] Lee Naish. *Negation and Control in Prolog*. Springer-Verlag, 1986. Lecture Notes in Computer Science Volume 238.

[28] Joachim Niehren. Functional Computation as Concurrent Computation. In *23^{rd} ACM Symposium on Principles of Programming Languages*, 1996.

[29] Joachim Niehren and Martin Müller. Constraints for Free in Concurrent Computation. In *Asian Computer Science Conference*, Pathumthani, Thailand, 11–13 December 1995. Springer-Verlag, Berlin, Germany. LNCS, to appear.

[30] Rishiyur S. Nikhil. ID Language Reference Manual Version 90.1. Computation Structures Group Memo 284-2, MIT, July 1991.

[31] Rishiyur S. Nikhil. An Overview of the Parallel Language Id - A Foundation for pH, a Parallel Dialect of Haskell. Technical report, Digital Cambridge Research Laboratory, 1994.

[32] Seymour Papert. *MindStorms: Children, Computers and Powerful Ideas.* Basic Books, 1981.

[33] Simon L. Peyton-Jones and Philip Wadler. Imperative Functional Programming. In *20^{th} ACM Symposium on Principles of Programming Languages*, pages 71–84. ACM Press, January 1993.

[34] Andreas Podelski and Gert Smolka. Operational semantics of constraint logic programs with coroutining. In *ICLP 95*, pages 449–463, 1995.

[35] Konstantin Popov. An exercise in concurrent object-oriented programming: The Oz Browser. In *WOz'95, International Workshop on Oz Programming*, Martigny, Switzerland, November 1995.

[36] Vitor Santos Costa, David H. D. Warren, and Rong Yang. Andorra-I: A parallel Prolog system that transparently exploits both And- and Or-parallelism. In 3rd ACM SIGPLAN Conference on Principles and Practice of Parallel Programming, pages 83–93, 1991.

[37] Vijay A. Saraswat. *Concurrent Constraint Programming.* The MIT Press, Cambridge, MA, 1993.

[38] Christian Schulte and Gert Smolka. Encapsulated search in higher-order concurrent constraint programming. In Maurice Bruynooghe, editor, *Logic Programming: Proceedings of the 1994 International Symposium*, pages 505–520, Ithaca, New York, USA, 13–17 November 1994. The MIT Press.

[39] E. Shapiro. The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys*, 21(3):413–510, September 1989.

[40] Gert Smolka. A Foundation for Concurrent Constraint Programming. In Jean-Pierre Jouannaud, editor, *Conference on Constraints in Computational Logics*, volume 845 of *Lecture Notes in Computer Science*, pages 50–72, München, Germany, 7–9 September 1994.

[41] Gert Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Computer Science Today*, Lecture Notes in Computer Science, vol. 1000. Springer-Verlag, Berlin, 1995. to appear.

[42] Gert Smolka. The Definition of Kernel Oz. In Andreas Podelski, editor, *Constraints: Basics and Trends*, volume 910 of *Lecture Notes in Computer Science*, pages 251–292. Springer-Verlag, Berlin, Germany, 1995.

[43] Gert Smolka and Ralf Treinen, editors. *DFKI Oz Documentation Series*. German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, 1995.

[44] Guy L. Steele. *Common LISP: The Language*. Digital Press, second edition, 1990.

[45] L. Sterling and E. Shapiro. *The Art of Prolog*. The MIT Press, Cambridge, MA, Cambridge, MA, 1986.

[46] Philip Wadler. The Essence of Functional Programming. In *19th ACM Symposium on Principles of Programming Languages*, New York, January 1992. Invited Talk.

[47] D.H.D. Warren. Higher-order Extensions to Prolog: Are they Needed? In J.E. Hayes, Donald Michie, and Y.-H. Poa, editors, *Machine Intelligence 10*, volume 125 of *Lecture Notes in Mathematics*, chapter 22, pages 441–454. Wiley, Chichester, England, 1982.