Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

Research
Report
RR-94-23

The Definition of Kernel Oz

Gert Smolka

November 1994

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

Deutsches Forschungszentrum für Künstliche Intelligenz

The German Research Center for Artificial Intelligence (Deutsches Forschungszentrum für Künstliche Intelligenz, DFKI) with sites in Kaiserslautern and Saarbrücken is a non-profit organization which was founded in 1988. The shareholder companies are Atlas Elektronik, Daimler-Benz, Fraunhofer Gesellschaft, GMD, IBM, Insiders, Mannesmann-Kienzle, Sema Group, Siemens and Siemens-Nixdorf. Research projects conducted at the DFKI are funded by the German Ministry for Research and Technology, by the shareholder companies, or by other industrial contracts.

The DFKI conducts application-oriented basic research in the field of artificial intelligence and other related subfields of computer science. The overall goal is to construct systems with technical knowledge and common sense which - by using AI methods - implement a problem solution for a selected application area. Currently, there are the following research areas at the DFKI:

- ☐ Intelligent Engineering Systems
- ☐ Intelligent User Interfaces
- ☐ Computer Linguistics
- ☐ Programming Systems
- ☐ Deduction and Multiagent Systems
- ☐ Document Analysis and Office Automation.

The DFKI strives at making its research results available to the scientific community. There exist many contacts to domestic and foreign research institutions, both in academy and industry. The DFKI hosts technology transfer workshops for shareholders and other interested groups in order to inform about the current state of research.

From its beginning, the DFKI has provided an attractive working environment for AI researchers from Germany and from all over the world. The goal is to have a staff of about 100 researchers at the end of the building-up phase.

Dr. Dr. D. Ruland
Director

The Definition of Kernel Oz

Gert Smolka

DFKI-RR-94-23

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

© Deutsches Forschungszentrum für Künstliche Intelligenz 1994

This work may not be copied or reproduced in whole of part for any commercial purpose. Permission to copy in whole or part without payment of fee is granted for nonprofit educational and research purposes provided that all such whole or partial copies include the following: a notice that such copying is by permission of the Deutsche Forschungszentrum für Künstliche Intelligenz, Kaiserslautern, Federal Republic of Germany; an acknowledgement of the authors and individual contributors to the work; all applicable portions of this copyright notice. Copying, reproducing, or republishing for any other purpose shall require a licence with payment of fee to Deutsches Forschungszentrum für Künstliche Intelligenz.

ISSN 0946-008X

The Definition of Kernel Oz

Gert Smolka
Programming Systems Lab
German Research Center for Artificial Intelligence (DFKI)
Stuhlsatzenhausweg 3
D66123 Saarbrücken, Germany
email: `smolka@dfki.uni-sb.de`

Abstract

Oz is a concurrent language providing for functional, object-oriented, and constraint programming. This paper defines Kernel Oz, a semantically complete sublanguage of Oz. It was an important design requirement that Oz be definable by reduction to a lean kernel language.

The definition of Kernel Oz introduces three essential abstractions: the Oz universe, the Oz calculus, and the actor model. The Oz universe is a first-order structure defining the values and constraints Oz computes with. The Oz calculus models computation in Oz as rewriting of a class of expressions modulo a structural congruence. The actor model is the informal computation model underlying Oz. It introduces notions like computation spaces, actors, blackboards, and threads.

Contents

1 Introduction

Oz is a concurrent language providing for functional, object-oriented, and constraint programming. It is defined by reduction to a lean sublanguage, called Kernel Oz, which is defined in this paper. The fact that we can elegantly define the semantic essence of Kernel Oz in less than ten pages indicates that such a multi-paradigm language is feasible. Further evidence is provided by the existence of a complete and efficient implementation.

The research behind Oz is driven by practical and theoretical considerations.

On the practical side, we see the need for a concurrent high-level language. Clearly, such a language should subsume higher-order functional programming, and organize state and concurrent functionality by means of objects. For tasks that involve search, the problem solving capabilities known from constraint logic programming would be advantageous.

On the theoretical side, we would like to advance towards a unified computation model subsuming and explaining seemingly incompatible programming paradigms. Concurrency and constrained-based problem solving are particularly challenging. Important considerations in the development of a unified programming model are its simplicity and generality as a mathematical construction, its usefulness as a basis for designing practical programming languages, and the existence of simple and efficient implementation models.

Programming languages can be classified by the computation model they are based on. Imperative programming, functional programming, logic programming, and concurrent logic programming are established classes based on different computation models. Oz does not fit in any of these classes. Rather, it is based on a new computation model incorporating ideas from functional programming, logic programming, and concurrent computation (the π-calculus, in particular). Here are some principles realized in Kernel Oz:

- Expressions are composed concurrently, with references made by lexically scoped logic variables.
- All values are defined as the elements of a first-order structure, called the Oz universe.
- Values are described by constraints, which are logic formulas over the Oz universe, and are combined automatically by means of constraint simplification and propagation.
- Reference to fresh names is possible, where names are special values of the Oz universe.
- Procedural abstraction is provided with full generality, where abstractions are referred to by names.

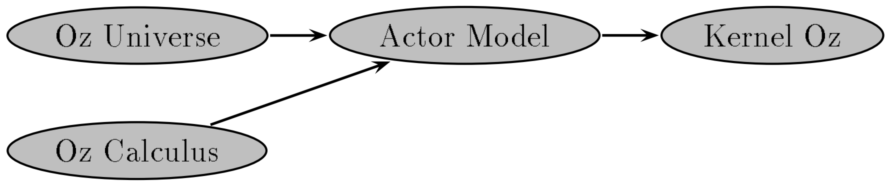
- State is provided through cells, which are primitive concurrent agents holding a reference.
- Speculative constraint computation is delegated to local computation spaces.
- Search is provided by a combinator spawning a local computation space and returning nondeterministic alternatives as procedural abstractions.

A guiding principle in the design of Oz was the requirement that Oz be definable by reduction to a kernel language as lean as possible. This led us to look for minimal primitives for expressing computational concepts such as functions, objects, and search. The search for a coherent collection of such primitives has been an exciting journey through a jungle of complexity to a glade of simplicity.

Structure of the Definition

The formal definition of a real programming language is a complex task. To be useful it must be simple. To be simple, it must introduce different abstractions, identifying different concerns that can be treated independently. For powerful abstractions to exist, the language design must be based on these abstractions. Thus designing a programming language subsumes creating the abstractions explaining the language.

The definition of Kernel Oz introduces three essential abstractions: the Oz universe, the Oz calculus, and the actor model.



The Oz universe defines values and constraints. It is a structure of first-order predicate logic whose elements are the values and whose formulas are the constraints Oz computes with. The values of Oz are closed under tuple and record construction and include numbers, atoms and names. The fact that Oz provides for full higher-order programming but has first-order values only is a radical departure from the established models of functional computation.

The actor model[1] is the informal computation model underlying Oz. It introduces notions like computation spaces, actors, blackboards, and threads. Computations can be described by a class of elaborable expressions.

The Oz Calculus formalizes the actor model, with the exception of the reduction strategy and input and output. It models concurrent computation as rewriting of

[1]The actor model for Oz is quite different from Hewitt's actor model of computation [6]. However, both models have in common that they are inherently concurrent (Hewitt speaks of ultra concurrency).

a class of expressions modulo a structural congruence. This set-up, which is also employed in more recent presentations of the π-calculus [8, 7], proves particularly useful for Oz since constraint propagation and simplification can be accommodated elegantly by means of the structural congruence.

Kernel Oz itself consists of a class of expressions whose semantics is defined by a translation into the elaborable expressions of the actor model. Kernel Oz restricts the expressivity of constraints so that an efficient implementation becomes possible.

How to read the Definition

This report gives a complete and concise definition of Kernel Oz. Supplementary literature is needed to understand the language design and programming in Oz. The reader is expected to have an intuitive understanding of Oz, as conveyed by [14]. More thorough introductions to programming in Oz are [5, 9]. The document [4] defines Oz by reduction to Kernel Oz.

On first reading, we recommend to ignore the constraint programming aspects of Oz (disjunctions, solvers, finite domains). The study of the Oz calculus should be prepared by reading [13], which introduces a simplified calculus not covering constraints and search. Other aspects of the calculus, in particular deep guards and the relationship to logic programming, are discussed in [12]. The search combinator is introduced in [10, 11].

Acknowledgements

The following persons have contributed directly to the design and/or implementation of Oz: Martin Henz, Michael Mehl, Martin Müller, Tobias Müller, Joachim Niehren, Konstantin Popow, Ralf Scheidhauer, Christian Schulte, Gert Smolka, Ralf Treinen, Jörg Würtz.

2 The Oz Universe

The Oz universe is a mathematical model of the data structures Oz computes with. It is defined as a structure of first-order predicate logic with equality. All variables in Oz range over the elements of the Oz universe. The elements of the Oz universe are called **values**, and the first-order formulas over its signature are called **constraints**.

2.1 Values

Values are classified as shown in Figure 1. A value is either a primitive or a compound value. A **primitive value** is either a literal or a number. A **literal** is either an atom or a name. A **number** is either an integer or a float. A **compound value** is either a proper tuple or a proper record.

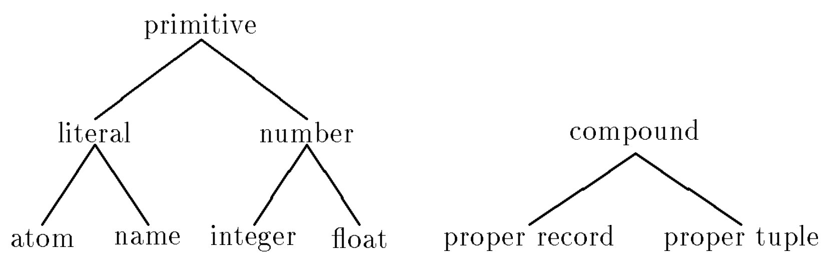


Figure 1: Classification of values.

It remains to define the basic classes of atoms, names, integers, floats, proper records, and proper tuples, which are pairwise disjoint. Every value is in one and only one basic class.

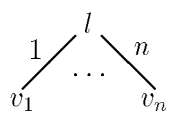
Atoms are finite sequences of positive integers between 1 and 255. For convenience, atoms are usually written as strings, exploiting a mapping specified in [4] (e.g., `'fred'`, `'Atom'`, `''`, or `'$#@\'78'`). For alphanumeric atoms starting with a lower case letter we usually omit the quotes; for instance, we may write `fred` for `'fred'`. Atoms are totally ordered by the lexical order induced by the canonical order on integers.

Names are primitive values without any structure. There are infinitely many names. There is no order on names.

The **integers** are the integers you know from school. They are ordered as usual.

The **floats** are the finitely many floating point numbers defined by the implementation. They are totally ordered.

A **tuple** is either a literal or a proper tuple. A **proper tuple** is an ordered tree



where l is a literal, $v_1, \ldots, v_n$ are values, and $n > 0$. Tuples are written as $l(v_1 \ldots v_n)$, where $l()$ stands for l. Two tuples are equal if and only if they have the same linear notation.

Given a tuple $t = l(v_1 \ldots v_n)$, we call the literal l the **label**, the values $v_1, \ldots, v_n$ the **components**, the integer n the **width**, and the integers $1, \ldots, n$ the **features** of t. Moreover, we call v_i the **component** or **subtree of** t **at** i.

A **record** is either a literal or a proper record. A **proper record** is an unordered tree

$$\begin{tikzcd} & l \arrow[dl, dash, "l_1"'] \arrow[dr, dash, "l_n"] & \\ v_1 & \cdots & v_n \end{tikzcd}$$

where l is a literal, $l_1, \ldots, l_n$ are pairwise distinct literals, $v_1, \ldots, v_n$ are values, and $n > 0$.

Records are written as $l(l_1{:}\, v_1 \ldots l_n{:}\, v_n)$, where $l()$ stands for l. Two proper records are equal if and only if they have the same linear notation up to permutation of named fields $l_i{:}\, v_i$.

Given a record $t = l(l_1{:}\, v_1 \ldots l_n{:}\, v_n)$, we call the literal l the **label**, the values $v_1, \ldots, v_n$ the **fields**, the integer n the **width**, and the literals $l_1, \ldots, l_n$ the **features** of t. Moreover, we call v_i the **field** or **subtree of** t **at** l_i.

By definition, every literal is both a nullary tuple and a zero-width record. In other words, the intersection of the set of all tuples with the set of all records is exactly the set of all literals.

The Oz universe is closed under tuple and record construction. It also contains all infinite trees that can be obtained from primitive values by tuple and record construction. Consequently, the equation

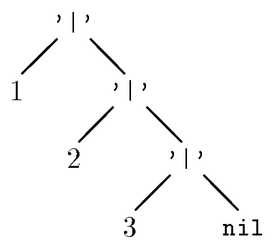
$$x \quad = \quad l(l_1{:}\, x \; l_2{:}\, v)$$

has exactly one solution for x given l, l_1, l_2, v. The straightforward mathematical details of the underlying construction can be found in [15].

An important operation on records is adjunction. The **adjunction** of two records s and t is the record $s * t$ defined as follows: the label of $s * t$ is the label of t; the features of $s * t$ are the features of s together with the features of t; and v is the subtree of $s * t$ at l if and only if either v is the subtree of t at l, or if l is not a feature of t and v is the subtree of s at l. Thus record adjunction amounts to record concatenation, where for common features the right argument takes priority. For instance,

$$l(a{:}\,1 \;\; b{:}\,2 \;\; c{:}\,3) \; * \; k(b{:}\,7 \;\; d{:}\,4) \quad = \quad k(a{:}\,1 \;\; b{:}\,7 \;\; c{:}\,3 \;\; d{:}\,4).$$

Lists are special tuples defined inductively as follows: the atom `nil` is a list (called the **empty list**); and if v is a value and w is a list, then the tuple `'|'`$(v\ w)$ is a list (where v is called the **head** and w is called the **tail**). For instance,



is the list containing the integers 1, 2, 3 in ascending order.

2.2 Constraints

The **signature of the Oz universe** consists of all primitive values and of finitely many predicates called **constraint predicates**. Every primitive value is a constant denoting itself. Note that the signature of the Oz universe contains no proper function symbol. The first-order formulas over the signature of the Oz universe are called **constraints**. The variables occurring in constraints are taken from a fixed infinite set.

The constraint predicates of the Oz universe are defined as follows:

- `isAtom`(x), `isName`(x), `isLiteral`(x), `isInt`(x), `isFloat`(x), `isNumber`(x), `isRecord`(x), and `isTuple`(x) are defined as one would expect from their names.
- `intPlus`(x, y, z) and `intTimes`(x, y, z) are the predicates corresponding to integer addition and multiplication. For instance, the formulas `intMult`$(3, 6, 18)$, $\neg$`intMult`$(3.0, 6, 18)$, and $\neg$`intMult`$(3.0, 6, 18.0)$ are all true in the Oz universe.
- `floatPlus`(x, y, z), `floatMinus`(x, y, z), `floatTimes`(x, y, z), `floatDiv`(x, y, z), `floatPow`(x, y, z), `floatAbs`(x, y), `floatCeil`(x, y), `floatFloor`(x, y), `floatExp`(x, y), `floatLog`(x, y), `floatSqrt`(x, y), `floatSin`(x, y), `floatASin`(x, y), `floatCos`(x, y), `floatACos`(x, y), `floatTan`(x, y), and `floatATan`(x, y) are the predicates corresponding to the respective functions on the floats (defined by the implementation).
- `floatToInt`(x, y) and `intToFloat`(x, y) are the predicates corresponding to the respective conversion functions (defined by the implementation).
- `atomString`(x, y) holds if and only if y is the list of integers corresponding to the atom x. Note that `atomString`(x, y) is functional from left to right and from right to left.
- `intLE`(x, y), `floatLE`(x, y), and `atomLE`(x, y) are the predicates corresponding to the respective total orders on integers, floats, and atoms.
- `label`(x, y) holds if and only if x is a tuple or record whose label is y.

- `width`(x, y) holds if and only if x is a tuple or record whose width is y.
- `subtree`(x, y, z) holds if and only if x is a tuple or record, y is a feature of x, and z is the subtree of x at y.
- `extendTuple`(x, y, z) holds if and only if x and z are tuples and z is obtained from x by adding y as additional rightmost component.
- `adjoin`(x, y, z) is the predicate corresponding to record adjunction.
- `adjoinAt`(x, y, z, u) holds if and only if y is a literal and x and u are records such that $x * l(y{:}\, z) = u$, where l is the label of x.
- `arity`(x, y) holds if and only if x is a record and y is the list of the atomic features of x (i.e., all features of x that are atoms) in ascending order.
- `finiteDomainBound(x)` holds if and only if x is the upper bound for finite domains defined by the implementation. The upper bound must be an integer larger or equal than 255.

An important property of the Oz universe is the fact that validity of sentences is preserved under permutation of names; that is, given two first-order sentences S, T over the signature of the Oz universe such that T is obtained from S by a bijective renaming of names, S is valid in the Oz universe if and only if T is valid in the Oz universe. To obtain this property, no order on names is defined, and consequently the arity of a record does not contain those features that are names.

We write $\top$ for the trivial constraint true, and $\bot$ for the trivial constraint false. We say that

- a constraint ϕ **entails** a constraint ψ if the implication $\phi \rightarrow \psi$ is valid in the Oz universe
- a constraint ϕ is **equivalent** to a constraint ψ if the equivalence $\phi \leftrightarrow \psi$ is valid in the Oz universe
- a constraint ϕ is **satisfiable** if it does not entail $\bot$.

For convenience, we write

$$x \doteq l(l_1{:}\, y_1,\ \ldots,\ l_n{:}\, y_n)$$

for the constraint

$$\texttt{label}(x, l) \ \wedge\ \texttt{width}(x, n) \ \wedge\ \texttt{subtree}(x, l_1, y_1) \ \wedge\ \ldots\ \wedge\ \texttt{subtree}(x, l_n, y_n).$$

and

$$x \doteq l(y_1,\ \ldots,\ y_n)$$

for the constraint

$$\texttt{label}(x, l) \;\wedge\; \texttt{width}(x, n) \;\wedge\; \texttt{subtree}(x, 1, y_1) \;\wedge\; \ldots \;\wedge\; \texttt{subtree}(x, n, y_n).$$

Moreover, we write

$$x \doteq y_1|\ldots|y_k|\texttt{nil}$$

for the constraint constraining x to the list $y_1, \ldots, y_k$.

Since the Oz universe has integers with addition and multiplication, satisfiability of constraints is undecidable, even for conjunctions of atomic integer constraints (Hilbert's Tenth Problem). Furthermore, satisfiability of constraints involving no other predicate but the subtree predicate is undecidable [16].

Kernel Oz restricts the use of constraints such that satisfiability and entailment of the occurring constraints is efficiently decidable.

More about the logic and algorithmic properties of record and tuple constraints can be found in [15, 3, 1, 2].

3 The Actor Model

The actor model is the informal computation model underlying Oz. It can be seen as a computational metaphor for the Oz calculus, the formal computation model underlying Oz. The two models formulate complementary views of computation in Oz supporting different intuitions. The actor model takes an operational perspective introducing notions like computation spaces, actors, and blackboards.

An important aspect of an inherently concurrent computation model like the one underlying Oz is the reduction strategy determining the partial order according to which possible reduction steps are to be performed. The reduction strategy has semantical significance as it comes to fairness, and practical significance as it comes to efficient implementation. Finding the right reduction strategy has been one of the more difficult issues in designing Oz.

By its nature, an informal model must rely on the intuition of the reader and cannot compete with the rigor of a formal model. Thus the Oz calculus is taken as the defining model, except for the reduction strategy and input and output, which are only formulated in the actor model. The formally inclined reader may prefer to study the Oz calculus first.

3.1 Computation Spaces, Blackboards, and Actors

Computation in Oz takes place in a computation space called the **top level**. A **computation space** consists of a finite number of **actors** connected to a **blackboard**. Computation proceeds by reduction of actors. When an actor reduces, it may create new actors and write information on the blackboard. As long as an actor does not reduce, it does not have an outside effect. Actors are short-lived: once they reduce they disappear.

A **blackboard** is a store containing a constraint and a partial function **binding** names to abstractions and variables. Names, variables, and constraints are defined by the Oz universe, and abstractions will be defined later. The blackboard stores its constraint only up to logical equivalence in the Oz universe. A blackboard is called **empty** if it binds no name and if its constraint is $\top$.

A blackboard **entails** a constraint ψ if the constraint of the blackboard entails ψ. A blackboard **binds a variable** x to a variable or constant $s \neq x$ if the constraint $x \doteq s$ is entailed by the constraint of the blackboard.

As computation proceeds, the information stored by the constraint of a blackboard increases monotonically. More precisely, if a blackboard evolves from a state B to a state B', then:

- If B entails a constraint ϕ, then B' entails ϕ.
- If B binds a name a to an abstraction $\overline{y}/E$, then B' binds a to $\overline{y}/E$.

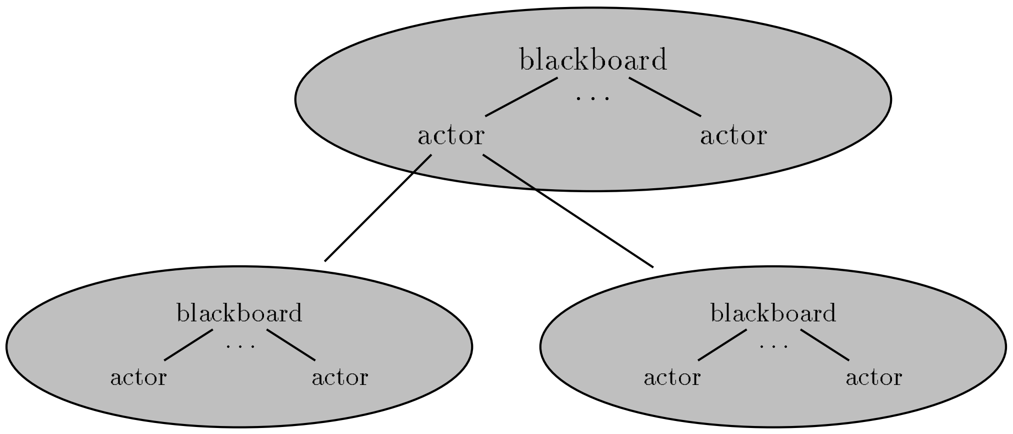


Figure 2: A tree of computation spaces.

- If B binds a name a to an once-only abstraction $\overline{y}/\!\!/E$, then B' binds a to either $\overline{y}/\!\!/E$ or $\overline{y}/\bot$.
- If B binds a name a to a variable, then B' binds a to a (possibly different) variable.

Abstractions will be defined shortly.

Some actors spawn **local computation spaces**, thus creating a tree of computation spaces taking the top level as root (see Figure 2). As computation proceeds, new local computation spaces are created and existing local spaces are discarded or merged with their parent space.

We say that a computation space S' is **subordinated** to a computation space S if $S' = S$ or if S' is subordinated to a local computation space of an actor of S. We say that a blackboard B is **subordinated** to a computation space S if B is the blackboard of a space subordinated to S. We say that a blackboard B' is **subordinated** to a blackboard B if B' is subordinated to the computation space of B. We say that X is **superordinated** to Y if Y is subordinated to X.

Every computation space is equipped with a possibly empty set of **local variables** and **local names**. As computation proceeds, computation spaces may acquire fresh local variables and names. A variable or name can be local to at most one computation space. If a variable or name is local to a space, the space is called its **home space**. Every occurring name must have a home space. Moreover, every variable having occurrences that are not statically bound (defined below) must have a home space. Variables and names with a home space can only occur in spaces subordinated to their home space.

The hierarchy of computation spaces rooted in the top level satisfies the following invariants:

- The constraints of subordinated blackboards entail the constraints of superordinated blackboards (slogan: local spaces know the constraints of global spaces).
- Names can only be bound by the blackboard of their home space, and there is at most one binding for a name.

An important operation on a blackboard is the **imposition of a constraint**. A constraint ψ is **imposed** on a blackboard storing a constraint ϕ by making the blackboard store the constraint $\psi \wedge \phi$, where imposing ψ includes imposing ψ on all subordinated blackboards (thus the invariants are maintained).

We say that a computation space **fails** if a constraint is imposed such that the constraint of the blackboard becomes unsatisfiable. If a computation space fails, all its subordinated spaces fail. When a computation space fails, the actors of the space are discarded and computation in the space is aborted. While failure of a local space is a regular event, failure of the top level is considered a run-time error.

While some actors can reduce immediately once they have been created, others have to wait until the blackboard contains sufficient information. Once an actor becomes reducible, it remains reducible, except if its computation space fails or is discarded. An actor has an outside effect only once it reduces.

We assume that actors are reduced one after the other, an important assumption known as **interleaving semantics**. While we anticipate that an implementation reduces actors in parallel, we insist that the effect of such a parallel computation must always be achievable by a sequence of single actor reductions. Interleaving semantics separates concurrency from parallelism such that parallelism has no semantic significance and is only visible at the implementation level.

An Oz computation takes places concurrently with other computations, with some of which it may have to communicate and synchronize. To model this essential aspect of concurrent computation within the actor model, we assume that every top level computation space is equipped with an **input** and an **output stream**. An actor computation may read from the input stream and write on the output stream. Agents in the outside world may write on the input stream and read from the output stream. We assume that the tokens communicated on the input and output streams are atoms.

There are four kinds of actors: Elaborators, conditionals, disjunctions, and solvers. Conditionals, disjunctions and solvers are called **proper actors**. While proper actors spawn local computation spaces, elaborators do not.

3.2 Elaboration of Expressions

Elaborators are actors that **elaborate** a class of **elaborable expressions** de-

E	$::=$	ϕ	constraint
	$\mid$	$x\colon \overline{y}/E$	abstractor
	$\mid$	$x\colon \overline{y}/\!\!/E$	once-only abstractor
	$\mid$	$x\colon y$	cell creation
	$\mid$	$E_1\ E_2$	composition
	$\mid$	**local** $\overline{x}$ **in** E **end**	declaration
	$\mid$	`newName`$[x]$	name creation
	$\mid$	`apply`$[x\overline{y}]$	application
	$\mid$	**if** C_1 `[]` ... `[]` C_n **else** E **fi**	conditional
	$\mid$	**or** C_1 `[]` ... `[]` C_n **ro**	disjunction
	$\mid$	**OR** C_1 `[]` ... `[]` C_n **RO**	nondistributing disjunction
	$\mid$	`solve`$[x\colon E, y_1y_2y_3]$	solver
	$\mid$	`det`$[x]$	determination
	$\mid$	`getDomain`$[x, y]$	
	$\mid$	`input`$[x]$ $\mid$ `output`$[x]$	
	$\mid$	`setThreadPriority`$[x]$	
	$\mid$	`getThreadPriority`$[x]$	
C	$::=$	$\overline{x}$ **in** E_1 **then** E_2	clause
x, y, z	$::=$	$\langle$*variable*$\rangle$	
$\overline{x}$, $\overline{y}$	$::=$	$\langle$*possibly empty sequence of variables*$\rangle$	

Figure 3: Elaborable expressions.

fined in Figure 3. Every constraint as defined by the Oz universe is an elaborable expression, *provided* it contains no names. There is the further side-condition that the formal arguments $\overline{y}$ of an abstractor expression $x\colon\overline{y}/E$ or $x\colon\overline{y}/\!\!/E$ be pairwise distinct.

An **abstraction** takes the form $\overline{y}/E$ or $\overline{y}/\!\!/E$, where the variables in $\overline{y}$ are called the **formal arguments** and the elaborable expression E is called the **body** of the abstraction. The formal arguments are required to be pairwise distinct. An abstraction $\overline{y}/\!\!/E$ is called a **once-only abstraction** and can only be applied once.

Given an abstractor $x\colon\overline{y}/E$ or $x\colon\overline{y}/\!\!/E$, we call x the **designator** and $\overline{y}/E$ or $\overline{y}/\!\!/E$ the **abstraction** of the abstractor.

Given an application `apply`$[x\overline{y}]$, we call x the **designator** and $\overline{y}$ the **actual arguments** of the application.

Given a solver `solve`$[x\colon E, y_1y_2y_3]$, we call x the **root variable**, E the **guard**, and

y_1, y_2, y_3 the **control variables** of the solver.

Given a clause $\overline{x}$ **`in`** E_1 **`then`** E_2, we call the variables in $\overline{x}$ the **local variables**, the expression E_1 the **guard**, and the expression E_2 the **body** of the clause.

The elaborable expressions come with the following variable binders:

- universal and existential quantification in constraints
- an abstractor $x{:}\,\overline{y}/E$ or $x{:}\,\overline{y}/\!\!/E$ binds its formal arguments $\overline{y}$ with scope E
- a declaration **`local`** $\overline{x}$ **`in`** E **`end`** binds its **declared variables** $\overline{x}$ with scope E
- a solver `solve`$[x{:}\,E, y_1y_2y_3]$ binds its root variable x with scope E
- a clause $\overline{x}$ **`in`** E_1 **`then`** E_2 binds its local variables $\overline{x}$ with scope E_1 and E_2.

Free and bound variables are defined accordingly. An elaborable expression is **closed** if it has no free variable.

Computation spaces also act as variable binders: They bind their local variables. Every variable occurrence must be bound either **statically** by a binder in an elaborable expression or a constraint, or **dynamically** by a computation space. In particular, all free variables of the elaborable expression of an elaborator must be bound dynamically.

Note that we heavily overload the term “binding”. First, a blackboard can bind a name to a variable or an abstraction. Second, a blackboard can bind a variable x to a variable or constant s, which means that it entails the constraint $x \doteq s$. Third, a variable occurrence in an elaborable expression can be bound by a variable binder as defined above. Fourth, a variable occurrence can be bound by a computation space.

By **elaboration** of an expression E we mean the reduction of an elaborator for E. Elaboration of

- a constraint ϕ imposes ϕ on the blackboard of the computation space where the elaboration takes place. Recall that imposing a constraint on a blackboard means to impose it on all subordinated blackboards. Elaboration of a constraint in a space may result in the failure of some subordinated spaces.
- an abstractor $x{:}\,\overline{y}/E$ or $x{:}\,\overline{y}/\!\!/E$ chooses a fresh name a, binds a to the abstraction $\overline{y}/E$ or $\overline{y}/\!\!/E$, and imposes the constraint $x \doteq a$. Everything is done in the computation space where the elaboration takes place, which also acts as the home space of the fresh name a.
- a cell creation $x{:}y$ chooses a fresh name a, binds a to y, and imposes the constraint $x \doteq a$. The home space of a is the computation space where the elaboration takes place.
- a composition $E_1\ E_2$ creates two separate elaborators for E_1 and E_2.

- a declaration **`local`** x **`in`** E **`end`** chooses a fresh variable y and creates an elaborator for the expression $E[y/x]$. The notation $E[y/x]$ stands for the expression that is obtained from E by replacing all free occurrences of x with y. The home space of y is the space where the elaboration takes place. A multiple variable declaration **`local`** $x\ \overline{x}$ **`in`** E **`end`** is treated as a nested declaration **`local`** x **`in`** **`local`** $\overline{x}$ **`in`** E **`end`** **`end`**.

- `newName`$[x]$ chooses a fresh name a and imposes the constraint $x \doteq a$. The home space of a is the space where the elaboration takes place.

- `apply`$[x\overline{y}]$ must wait until there is a name a such that the blackboard entails $x \doteq a$. Then we distinguish three cases:

  1. If a is bound to an abstraction $\overline{z}/E$ by a superordinated blackboard and the number of the actual arguments $\overline{y}$ agrees with the number of the formal arguments $\overline{z}$, an elaborator for $E[\overline{y}/\overline{z}]$ is created (a copy of the body of the abstraction, where the actual arguments replace the formal arguments).
  2. If a is bound to an once-only abstraction $\overline{z}/\!/E$ by a superordinated blackboard and the number of the actual arguments $\overline{y}$ agrees with the number of the formal arguments $\overline{z}$, an elaborator for $E[\overline{y}/\overline{z}]$ is created. Moreover, a is rebound to the abstraction $\overline{y}/\bot$.
  3. If a is bound to a variable z by the blackboard of the space where the elaboration takes place and the actual arguments are $\overline{y} = y_1\, y_2$, then a is rebound to y_2 and the constraint $z \doteq y_1$ is imposed.

  In all other cases the elaborator for the application cannot reduce.

- a conditional **`if`** C_1 `[]` ... `[]` C_n **`else`** E **`fi`** creates a conditional actor spawning a local computation space for every clause $C_1, \ldots, C_n$ (see Figure 4). A local space for a clause $\overline{x}$ **`in`** E_1 **`then`** E_2 is created with a blackboard containing the constraint of the parent board, and a single elaborator for the expression **`local`** $\overline{x}$ **`in`** E_1 **`end`**. Moreover, the conditional actor carries the else expression E and associates with every local computation space the body E_2 of the corresponding clause. Since the scope of the local variables $\overline{x}$ includes both the guard E_1 and the body E_2 of the clause, the local variables $\overline{x}$ must be replaced consistently in the guard and in the body when **`local`** $\overline{x}$ **`in`** E_1 **`end`** is elaborated in the local space.

- a disjunction **`or`** C_1 `[]` ... `[]` C_n **`ro`** or **`OR`** C_1 `[]` ... `[]` C_n **`RO`** creates a disjunctive actor spawning a local space for every clause $C_1, \ldots, C_n$. The local spaces are created in the same way as for conditionals. A disjunctive actor created by **`or`** ... **`ro`** is called **distributing**, and a disjunctive actor created by **`OR`** ... **`RO`** is called **nondistributing**.

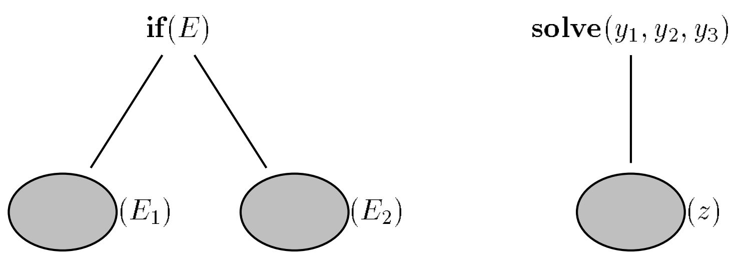


Figure 4: A conditional actor with two local spaces and a solver.

- `solve`$[x: E, y_1 y_2 y_3]$ creates a solver actor spawning a single local computation space (see Figure 4). The local computation space is created with a blackboard containing the constraint of the parent blackboard, and a single elaborator for $E[z/x]$, where z is a fresh variable taking the local computation space as home. The solver actor carries the root variables z and the control variables y_1, y_2, y_3.

- `det`$[x]$ must wait until the blackboard entails $x \doteq c$ for some constant c of the signature of the Oz universe. When this is satisfied, the elaborator for `det`$[x]$ reduces without further action.

- `getDomain`$[x, y]$ must wait until there are nonnegative integers $n_1, \ldots, n_k$ such that the blackboard entails the disjunctive constraint $x \doteq n_1 \vee \ldots \vee x \doteq n_k$. When this is satisfied, the elaborator for `getDomain`$[x, y]$ reduces by imposing the constraint
$$y \doteq n_1 | \ldots | n_k | \texttt{nil}$$
where $n_1, \ldots, n_k$ is the shortest list in ascending order such that the blackboard entails $x \doteq n_1 \vee \ldots \vee x \doteq n_k$.

- `input`$[x]$ waits until there is an atom s on the input stream, consumes it, and imposes the constraint $x \doteq s$.

- `output`$[x]$ waits until the blackboard entails $x \doteq s$ for some atom s and then puts s on the output stream.

- `setThreadPriority`$[x]$ and `getThreadPriority`$[x]$ will be defined in Subsection 3.4.

We have now seen all reduction rules for elaborators.

3.3 Reduction of Proper Actors

We will now specify the reduction rules for proper actors. Recall that a proper actor is either a conditional, a disjunction, or a solver.

We say that a local computation space is **entailed** if it contains no actors anymore, and if the parent blackboard entails $\exists\overline{x}\,\phi$, where $\overline{x}$ are the local variables and ϕ is the constraint of the blackboard of the local space.

Reduction of Conditionals

A conditional actor can reduce if one of its local computation spaces is entailed, or if all its local computation spaces have failed.

If one of its local computation spaces is entailed, the conditional can reduce as follows: discard the other local spaces, merge the local blackboard with the global blackboard (there cannot be any conflicts), and create an elaborator for the associated clause body.

If several local computation spaces of a conditional are entailed, the conditional can choose with which one it reduces.

If all local computation spaces of a conditional have failed, it can reduce to an elaborator for the else expression.

Reduction of Disjunctions

A disjunctive actor can reduce if all but possibly one of its local computation spaces have failed, or if a local space whose associated clause body is the constraint $\top$ is entailed.

If all local computation spaces of a disjunction have failed, the disjunction can reduce to an elaborator for the constraint $\bot$.

If all but one local computation space of a disjunction have failed, the disjunction can reduce with the unfailed space. This is done by merging the unfailed local space with the global space (there cannot be any conflicts), and by creating an elaborator for the associated clause body.

If a local space with associated clause body $\top$ is entailed, the disjunctive actor can reduce without further action.

Reduction of Solvers

A local computation space is called **blocked** if it is unfailed and no actor in the space or a subordinated space can reduce. A local computation space is called **stable** if it is blocked and remains blocked for every satisfiable strengthening of the constraint of the parent blackboard.

A solver actor can reduce if its local computation space is either failed or stable.

If the local computation space of a solver is failed, the solver can reduce to an elaborator for the application `apply`$[y_1]$, where y_1 is the first control variable of the solver.

If the local computation space of a solver is stable and does not contain a distributing disjunctive actor, the solver can reduce to an elaborator for the expression

```
local $x$ in
    $x$: $z/F$
    apply[ $y_2$ $x$ $S$ ]
end
```

where z is the root variable of the solver, F is an elaborable expression representing the stable local computation space, y_2 is the second control variable of the solver, and S (the so-called **status**) is either the atom `entailed` or `stable`, depending on whether the local space is entailed or not. That F represents the local computation space means that elaboration of `solve`[z: F, $y_1 y_2 y_3$] will recreate the local space up to renaming of local variables and names by reduction of elaborators only. The transformation of a computation space into an elaborable expression is called **reflection**.

If the local computation space of a solver is stable and contains a distributing disjunctive actor **or** C_1 [] ... [] C_n **ro**, the solver can reduce to an elaborator for

```
local $x_1$ $x_2$ in
    $x_1$: $z$// local $\overline{y}$ in $F$ or $C_1$ ro end
    $x_2$: $z$// local $\overline{y}$ in $F$ or $C_2$ [] ... [] $C_n$ ro end
    apply[$y_3$ $x_1$ $x_2$ $S$]
end
```

where **local** $\overline{y}$ **in** F **or** C_1 [] ... [] C_n **ro end** is an elaborable expression representing the stable local computation space, z is the root variable of the solver, y_3 is the third control variable of the solver, and S (the so-called **status**) is either the atom `last` or `more`, depending on whether $n = 1$ or not. The alternatives are returned as once-only abstractions to allow for an efficient implementation.

For distributing disjunctive expressions and actors the order of clauses and local computation spaces is significant and preserved by elaboration and reflection.

3.4 Reduction Strategy

So far we have not made any assumptions about the order in which actors are reduced. Such assumptions are needed, however, so that one can write fair and efficient programs. Without such assumptions a single infinite computation could starve all other computations.

Oz's reduction strategy organizes actors into threads, where every thread is guaranteed to make progress if it can reduce and has sufficient priority. Threads are equipped with priorities to provide for asynchronous real time programming.

A **thread** is a nonempty sequence of actors. Every actor belongs to exactly one thread. When a computation space fails or is discarded, its actors are discarded, which includes their removal from the threads they reside on. The actors on a thread may belong to different computation spaces.

Every thread has a **priority** that can be changed. The priority is an integer, where a larger integer means a higher priority.

Threads are scheduled by means of a **priority queue**, which is served by one or several **workers**. A free worker picks the first thread from the queue and starts reducing it. If the thread cannot reduce anymore, or the worker has spent more than a given time limit reducing it, the worker puts the thread back into the queue, at the position determined by the current priority of the thread.

Although there may be several workers, only one actor can reduce at a time. Thus reductions performed by different workers are interleaved into a sequence of single reductions (so-called **interleaving semantics**).

A thread can reduce by reducing one of its actors, or by moving its first actor to a new thread. If possible, a thread reduces a proper actor. If it contains no reducible proper actor, the thread must reduce with its first actor. Every thread that contains more than one actor is reducible.

To reduce an actor on a thread means to reduce the actor and replace it with the possibly empty sequence of actors it has reduced to. Proper actors reduce to a single elaborator or no actor at all. Elaborators may reduce to more than one actor. For them the order of the replacing actors is defined as follows:

- For the elaborator of a composition $E_1\ E_2$, the elaborator for E_1 goes before the elaborator for E_2.
- For the elaborator of a conditional [disjunction], the elaborators for the clauses $e_1, \dots, e_n$ go before the conditional [disjunctive] actor a,

  $$e_1\ \dots\ e_n\ a$$

  where the order of the elaborators $e_1, \dots, e_n$ is given by the order of the clauses in the conditional [disjunctive] expression
- For the elaborator of a solver, the elaborator of the local computation space goes before the solver actor.

Reduction of threads is defined as follows:

1. if a thread contains a reducible proper actor, reduce it
2. if a thread contains no reducible proper actor and the first actor is reducible, reduce the first actor
3. if a thread contains no reducible proper actor, the first actor is not reducible, and the thread contains further actors, move the first actor to a newly created thread; the newly created thread inherits the priority of the creating thread.

We say that the third rule **suspends** the first actor of a thread. Note that suspension of an actor creates a new thread, and that this is the only way to create a new thread.

The strategy gives priority to the reduction of proper actors, where the position in the thread does not matter. Since proper actors reduce to elaborators, a thread will quickly run out of reducible proper actors. Elaborators are reduced with a strategy reminiscent of sequential execution.

Elaboration of an expression

- `setThreadPriority`$[x]$ must wait until there is an integer n such that the blackboard entails $x \doteq n$. When this is satisfied, the elaborator of `setThreadPriority`$[x]$ can reduce by changing the priority of its thread to n. If the priority is not stricly increased, the worker must return the thread to the priority queue.
- `getThreadPriority`$[x]$ imposes the constraint $x \doteq n$, where n is the current priority of the thread elaborating the expression.

Concerning solvers, there is a further assumption about order: When a solver reduces by distributing a disjunctive actor, the distributing disjunctive actor that was created last is distributed.

3.5 Computations

A computation space is called **irreducible** if no actor in the space or a subordinated space can reduce. Note that a space is irreducible if and only if it is either blocked or failed.

A **finite computation** issuing from a closed elaborable expression E is a sequence $S_1, \ldots, S_n$ of states of a top level computation space such that:

- The initial state S_1 consists of the empty blackboard and an elaborator for E.
- Every state S_{i+1} is obtained from its predecessor S_i by reduction of a single actor, possibly in a subordinated space.
- The final state S_n is irreducible.
- The state sequence respects the reduction strategy.

Since failure prevents further reduction, none of the states $S_1, \ldots, S_{n-1}$ can be failed.

A **infinite computation** issuing from a closed elaborable expression E is an infinite sequence S_1, S_2, S_3, ... of states of a top level computation space such that:

- The initial state S_1 consists of the empty blackboard and an elaborator for E.
- Every state S_{i+1} is obtained from its predecessor S_i by reduction of a single actor, possibly in a subordinated space.

- The state sequence respects the reduction strategy.

Since failure prevents further reduction, none of the states in an infinite computation can be failed.

Example 3.1 There are both finite and infinite computations issuing from the closed elaborable expression

```
local X Y in
   X:/ apply[X]
   if X≐Y then ⊤ [] X≐Y then apply[X] else ⊤ fi
   X≐Y
end
```

However, due to the reduction order imposed by threads, there are no infinite computations issuing from

```
local X Y in
   X:/ apply[X]
   X≐Y
   if X≐Y then ⊤ [] X≐Y then apply[X] else ⊤ fi
end
```

□

3.6 Success, Failure, and Termination of Actors

The **direct descendants** of an actor A are the actors A creates when it reduces. The **descendants** of an actor are obtained by taking the reflexive and transitive closure of the direct descendant relation. The actors in the local spaces of a proper actor A are not considered descendants of A, and neither are their reductions considered reductions of A.

We say that an actor has

- **succeeded** if all its descendants have reduced without failing the computation space
- **failed** if one of its descendants has failed the computation space
- **terminated** if all its descendants have reduced.

Note that an actor has terminated if and only if it has succeeded or failed.

4 Kernel Oz

This section defines Kernel Oz, a semantically complete sublanguage of Oz. Every Oz program can be translated into an expression of Kernel Oz. In fact, the meaning of Oz programs is defined by a reduction to Kernel Oz.

Kernel Oz consists of a class of expressions whose semantics is defined by a translation into the elaborable expressions of the actor model.

Kernel Oz restricts the elaborable constraints such that the actor model can be implemented efficiently. This is necessary since properties such as satisfiability and entailment of constraints are undecidable in general. Kernel Oz provides most of the available constraints only indirectly through predefined procedures. All predefined procedures are defined by an elaborable expression called prelude.

Although Oz is semantically defined by reduction to Kernel Oz, it cannot be implemented efficiently this way. In particular, implementations are supposed to realize objects and finite domains more efficiently than it is suggested by their translation to Kernel Oz.

4.1 Syntax

The abstract syntax of Kernel Oz is defined in Figure 5. It introduces a class of expressions called **kernel expressions**. The kernel expressions are less expressive than the elaborable expressions. Except for constraints, the missing expressivity is regained by means of predefined procedures.

E	$::=$	**false** $\mid$ **true** $\mid$ x=s	constraints
	$\mid$	**proc** $\{x\ \overline{y}\}$ E **end**	procedure definition
	$\mid$	$\{x\ \overline{y}\}$	procedure application
	$\mid$	$E_1\ E_2$	composition
	$\mid$	**local** $\overline{x}$ **in** E **end**	declaration
	$\mid$	**if** C_1 [] ... [] C_n **else** E **fi**	conditional
	$\mid$	**or** C_1 [] ... [] C_n **ro**	disjunction
	$\mid$	**OR** C_1 [] ... [] C_n **RO**	nondistributing disjunction
C	$::=$	$\overline{x}$ **in** E_1 **then** E_2	clause
x, y	$::=$	$\langle$*variable*$\rangle$	
$\overline{x}$, $\overline{y}$	$::=$	$\langle$*possibly empty sequence of variables*$\rangle$	
s	$::=$	x $\mid$ $\langle$*atom*$\rangle$ $\mid$ $\langle$*number*$\rangle$	

Figure 5: Kernel expressions.

A concrete syntax for kernel expressions is inherited from the concrete syntax of Oz (defined in [4]).

Every kernel expression can be rewritten into an elaborable expression by applying the following rules:

- A procedure definition **proc** $\{x\ y_1\ \ldots\ y_n\}$ E **end** rewrites into

```
local A R in
    A: $y_1 \ldots y_n/E$
    R $\doteq$ rec(abstraction: A, arity: $n$)
    $x \doteq$ ChunkLabel($\mathit{Proc}$ : R)
end
```

- A procedure application $\{x\ y_1\ \ldots\ y_n\}$ rewrites into

```
if R in subtree($x$, $\mathit{Proc}$, R)
then
    local A in
        subtree(R, arity, $n$)
        subtree(R, abstraction, A)
        apply[A $y_1 \ldots y_n$]
    end
else false fi
```

- The constraint expressions **false**, **true**, x=s rewrite into the constraints $\bot$, $\top$, and $x \doteq s$, respectively.

The symbol *Proc* is a variable that must not occur in kernel expressions. Whenever possible, we use Oz's lexical syntax [4]; for instance, `abstraction`, `arity` and `rec` are atoms, and `A`, `R`, and `ChunkLabel` are variables. Moreover, `subtree(R, arity,` n`)` is a constraint, and `R` $\doteq$ `rec(abstraction: A, arity:` n`)` and $x \doteq$ `ChunkLabel(`*Proc* `: R)` abbreviate constraints (see Section 2).

The variable binders of the kernel expressions are clear from the translation to the elaborable expressions. It is understood that the declarations of `A` and `R` introduced by the above translation rules do not capture variables.

By providing abstractors and applications only indirectly through procedure definitions and applications, Kernel Oz establishes a recognizable class of first-order values acting as procedures (see the definition of the kernel procedure `IsProcedure`).

A kernel expression E is **admissible** if its free variables are among the **kernel variables**, which are the following:

```
ChunkLabel  NewName  NewCell  Exchange  Det  SolveCombinator
IsInt  IsFloat  IsNumber  IsAtom  IsName  IsLiteral  IsProcedure
IsCell  IsChunk  IsTuple  IsRecord  IsNoNumber
Label  Width  Subtree  ExtendTuple  Adjoin  AdjoinAt  Arity
AtomToString  StringToAtom  ProcedureArity
`=<`  `+`  `-`  `*`  `/`  Pow Abs
FloatToInt  IntToFloat  Ceil  Floor
Exp  Log  Sqrt  Sin  Cos  Tan  Asin  Acos  Atan
FiniteDomainBound  FiniteDomain  FiniteDomainNE  GetFiniteDomain
`Input`  `Output`  `SetThreadPriority`  `GetThreadPriority`
```

4.2 Semantics

The semantics of an admissible kernel expression E is defined as the semantics of the closed elaborable expression

```
local Proc ⟨Kernel Variables⟩ in
   newName[ChunkLabel]
   newName[Proc]
   ⟨Prelude⟩
   ⟨E rewritten into an elaborable expression⟩
end
```

where the elaborable expression ⟨*Prelude*⟩ is defined in the next section.

computation issuing from an admissible kernel expression E is a computation issuing from the closed elaborable expression obtained from E by the above translation.

4.3 Prelude

Below we define several elaborable expressions that composed together yield the expression ⟨*Prelude*⟩ needed for the semantic translation above. The procedures defined in the prelude are called **kernel procedures**. We use Oz's lexical syntax for variables and atoms (with the exception of the variable *Proc*, which has no concrete syntax), and Kernel Oz's syntax for procedure definitions and applications (to be expanded as defined in Section 4.1). Moreover, we write E_1 **then** E_2 for a clause $\overline{x}$ **in** E_1 **then** E_2 whose variable prefix $\overline{x}$ is empty.

Names, Determination, Procedures, and Cells

```
proc {NewName X}
   newName[X]
end
```

```
proc {Det X}
   if isInt(X) then det[X] else true fi
end

proc {IsProcedure P}
   if R in subtree(P,Proc,R) then true else false fi
end

proc {ProcedureArity P N}
   if R in subtree(P,Proc,R) then subtree(R,arity,N) else false fi
end

local Cell in
   {NewName Cell}

   proc {NewCell X C}
      local Z in
          Z: X
         C≐ChunkLabel(Cell:Z)
      end
   end

   proc {IsCell X}
      if Z in subtree(X,Cell,Z) then true else false fi
   end

   proc {Exchange C X Y}
      if Z in subtree(C,Cell,Z) then apply[Z X Y] else false fi
   end
end
```

Procedures and cells are modelled as special records called chunks, where a field holds the name bound to an abstraction or variable. Procedures and cells cannot be faked since their features *Proc* and `Cell`, respectively, cannot be accessed by admissible kernel expressions. This means that every value that qualifies as a procedure or cell must have been introduced by a procedure definition or an application of the kernel procedure `NewCell`, or must have been derived from such a value by possibly repeated adjunction.

Classification Predicates

The classification predicates classify the values of Kernel Oz according to the hierarchy shown in Figure 6. The classes value, number, noNumber, and literal are obtained by union of their subclasses. All leaf classes are disjoint. The classification predicates for procedures and cells were already defined in Section 4.3.

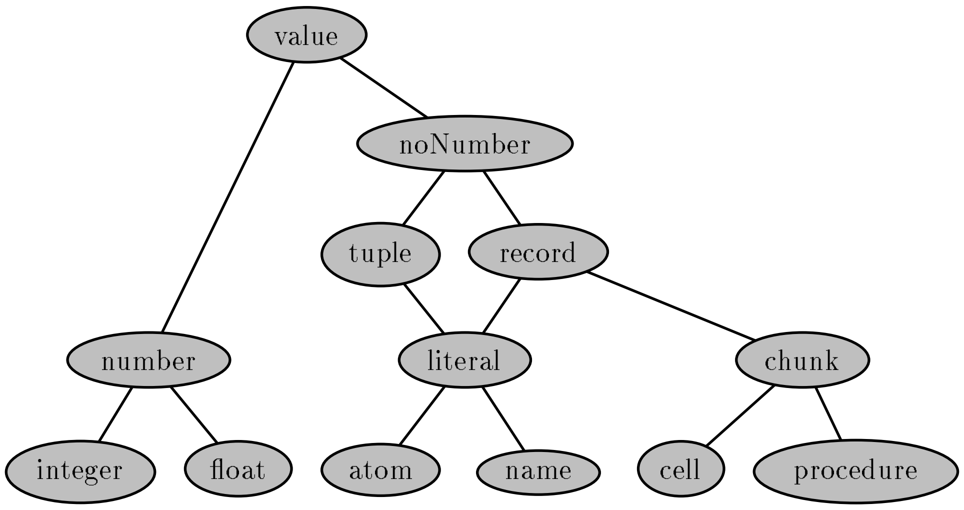


Figure 6: Classification of values in Kernel Oz.

```
proc {IsInt X}
   if isInt(X) then det[X] else false fi
end

proc {IsFloat X}
   if isFloat(X) then true else false fi
end

proc {IsNumber X}
   if {IsInt X} then true else {IsFloat X} fi
end

proc {IsAtom X}
   if isAtom(X) then true else false fi
end

proc {IsName X}
   if isName(X) then true else false fi
end

proc {IsLiteral X}
   if {IsAtom X} then true else {IsName X} fi
end
```

```
proc {IsTuple X}
   if isTuple(X) then true else false fi
end

proc {IsRecord X}
   if isRecord(X) then true else false fi
end

proc {IsChunk X}
   if {IsRecord X} then label(X,ChunkLabel) else false fi
end

proc {IsNoNumber X}
   if {IsTuple X} then true else {IsRecord X} fi
end
```

Order

```
proc {'=<' X Y}
   if {IsInt X} {IsInt Y} then intLE(X,Y)
   [] {IsFloat X} {IsFloat Y} then floatLE(X,Y)
   [] {IsAtom X} {IsAtom Y} then atomLE(X,Y)
   else false fi
end
```

Tuples, Records, and Atoms

```
proc {Label X L}
   if {IsNoNumber X} then label(X,L) else false fi
end

proc {Width X N}
   if {IsNoNumber X} then width(X,N) else false fi
end

proc {Subtree X F Y}
   if {IsRecord X} {IsLiteral F} then subtree(X,F,Y)
   [] {IsTuple X} {IsInt F} then subtree(X,F,Y)
   else false fi
end

proc {ExtendTuple X Y Z}
   if {IsTuple X} then extendTuple(X,Y,Z) else false fi
end
```

```
proc {Adjoin X Y Z}
   if {IsRecord X} {IsRecord Y}
   then if {IsChunk X} then false else adjoin(X,Y,Z) fi
   else false fi
end

proc {AdjoinAt X F Y Z}
   if {IsRecord X} {IsLiteral F}
   then if {IsChunk X} then false else adjoinAt(X,F,Y,Z) fi
   else false fi
end

proc {Arity X L}
   if {IsRecord X} then arity(X,L) else false fi
end

proc {AtomToString A L}
   if {IsAtom A} then atomString(A,L) else false fi
end

local ListDet in
   proc {StringToAtom L A}
      if {ListDet L} then  atomString(A,L) else false fi
   end
   proc {ListDet Xs}
      if X Xr in Xs≐'|'(X,Xr) {Det X}
      then {ListDet Xr} else Xs=nil fi
   end
end
```

Arithmetic

```
proc {`+` X Y Z}
   if {IsInt X} {IsInt Y} then intPlus(X,Y,Z)
   [] {IsFloat X} {IsFloat Y} then floatPlus(X,Y,Z)
   else false fi
end

proc {`-` X Y Z}
   if {IsInt X} {IsInt Y} then intPlus(Y,Z,X)
   [] {IsFloat X} {IsFloat Y} then floatMinus(X,Y,Z)
   else false fi
end
```

```
proc {`*` X Y Z}
   if {IsInt X} {IsInt Y} then intTimes(X,Y,Z)
   [] {IsFloat X} {IsFloat Y} then floatTimes(X,Y,Z)
   else false fi
end

proc {`/` X Y Z}
   if {IsFloat X} {IsFloat Y} then floatDiv(X,Y,Z) else false fi
end

proc {Pow X Y Z}
   if {IsInt X} Y=0 then Z=1
   [] {IsInt X} Y>0 then
      local A B in {`-` Y 1 A} {Pow X A B} {`*` X B Z} end
   [] {IsFloat X} {IsFloat Y} then floatPow(X,Y,Z)
   else false fi
end

proc {Abs X Y}
   if {IsInt X} then if X<0 then {`-` 0 X Y} else X=Y fi
   [] {IsFloat X} then floatAbs(X,Y)
   else false fi
end

proc {FloatToInt X Y}
   if {IsFloat X} then floatToInt(X,Y) else false fi
end

proc {IntToFloat X Y}
   if {IsInt X} then intToFloat(X,Y) else false fi
end

proc {Ceil X Y}
   if {IsFloat X} then floatCeil(X,Y) else false fi
end
```

The remaining kernel procedures for floating point arithmetic

```
Floor  Exp  Log  Sqrt  Sin  Cos  Tan  Asin  Acos  Atan
```

are defined analogously to `Ceil`.

Finite Domains

```
finiteDomainBound(FiniteDomainBound)
```

```
proc {FiniteDomain X}
   isInt(X) ∧ intLE(0,X) ∧ intLE(X,FiniteDomainBound)
end

proc {FiniteDomainNE X N}
   if {FiniteDomain X} {IsInt N} then ¬(X≐N) else false fi
end

proc {GetFiniteDomain X L}
   if {FiniteDomain X} then getDomain[X,L] else false fi
end
```

Solve Combinator

```
proc {‘SolveCombinator‘ Query Answer}
   local Failed Solved Distributed in
      Failed:/
         Answer≐failed
      Solved:X S/
         local P Q in
            proc {P Y} apply[X Y] end
            adjoinAt(P,status,S,Q)
            Answer≐solved(Q)
         end
      Distributed:X Y S/
         local P Q L R in
            proc {P Z} apply[X Z] end
            proc {Q Z} apply[Y Z] end
            adjoinAt(P,status,last,L)
            adjoinAt(Q,status,S,R)
            Answer≐distributed(L,R)
         end
      solve[X: {Query X}, Failed Solved Distributed]
   end
end
```

Thread Priorities

The kernel procedures ‘SetThreadPriority‘ and ‘GetThreadPriority‘ must only be used in system programs.

```
proc {'SetThreadPriority' N}
   if {IsInt N} then setThreadPriority[N] else false fi
end

proc {'GetThreadPriority' N}
   getThreadPriority[N]
end
```

Input and Output

The kernel procedures `'Input'` and `'Output'` serve as a semantic model for the higher-level input-output functions provided by Oz implementations. They are not meant for real use. When applied in a local computation space, `'Input'` and `'Output'` fail.

```
local IsTopLevel TopLevelCell in
   proc {'Input' X}
      {IsTopLevel} input[X]
   end
   proc {'Output' X}
      if {IsAtom X} then {IsTopLevel} output[X] else false fi
   end
   {NewCell top TopLevelCell}
   proc {IsTopLevel}
      local X in
         {ExchangeCell TopLevelCell X X}
         if X=top then true [] true then false else false fi
      end
   end
end
```

4.4 Normal Computation Spaces

Kernel Oz restricts the elaborable constraints such that failure, entailment, and stability of local computation spaces become efficiently decidable. This is done by providing the necessary constraints in weakened form through predefined procedures, where the weakened forms can be defined by elaborable expressions.

In the following we define a property called **normality** that is satisfied by all computation spaces occurring in computations issuing from admissible kernel expressions. For normal computation spaces, satisfiability, entailment, and stability can be decided efficiently.

A **determinant** for a variable x is a constraint that has one of the following forms:

1. $x \doteq s$, where s is an atom, a name, an integer, or a float

2. $x \doteq l(l_1{:}\, y_1,\ \ldots,\ l_n{:}\, y_n)$, where l and $l_1, \ldots, l_n$ are literals and $y_1, \ldots, y_n$ are variables

3. $x \doteq l(y_1,\ \ldots,\ y_n)$, where l is a literal and $y_1, \ldots, y_n$ are variables.

We say that a constraint ϕ **determines** a variable x if ϕ entails a determinant for x.

In the following it will become clear that the kernel procedure `Det` is defined such that an elaborator for `{Det X}` succeeds if and only if `X` is determined by the constraint of the blackboard.

A **normal constraint** for a variable x is either a determinant for x or a disjunction $x \doteq n_1 \vee \ldots \vee x \doteq n_k$, where $n_1, \ldots, n_k$ are $k > 1$ integers between 0 and the upper bound for finite domains.

A **solved constraint** is a constraint of the form

$$x_1 \doteq y_1 \ \wedge\ \ldots\ \wedge\ x_k \doteq y_k \ \wedge\ \phi_{k+1} \ \wedge\ \ldots\ \wedge\ \phi_n$$

where there exist variables $x_{k+1}, \ldots, x_n$ such that

- $x_1, \ldots, x_n$ are pairwise distinct
- $\phi_{k+1}, \ldots, \phi_n$ are normal constraints for $x_{k+1}, \ldots, x_n$
- the variables $x_1, \ldots, x_k$ are different from the variables $y_1, \ldots, y_k$ and do not occur in the normal constraints $\phi_{k+1}, \ldots, \phi_n$.

Proposition 4.1 *Every solved constraint is satisfiable in the Oz universe.*

Theorem 4.2 *The conjunction of two solved constraints is either unsatisfiable or logically equivalent (in the Oz universe) to a solved constraint. Moreover, entailment between two possibly existentially quantified solved constraints can be decided in quasi-linear time.*

Proof. Follows from the results in [15]. □

A computation space S is **normal** if all its subordinated spaces are normal, and if it satisfies one of the following conditions:

1. S is failed.

2. The constraint of S's blackboard is solved.

3. S contains no actors and the constraint of its blackboard is a conjunction of a solved constraint and an atomic formula obtained with one of the following constraint predicates: `subtree`, `isInt`, `IsFloat`, `IsAtom`, `IsName`, `IsTuple`, `IsRecord`.

Claim 4.3 *Every computation space occurring in a computation issuing from an admissible kernel expression is normal (up to logical equivalence of constraints in the Oz universe).*

4.5 Logical Semantics

A pair of a constraint ψ and n pairwise distinct variables $x_1, \ldots, x_n$ is called a **logical semantics** of a procedure p taking n arguments if the following two conditions are satisfied:

- If an elaborator for an application $\{p\ x_1\ \ldots\ x_n\}$ fails, where no actors but the descendents of the elaborator are reduced, then the initial constraint of the blackboard entails $\neg\psi$.
- If an elaborator for an application $\{p\ x_1\ \ldots\ x_n\}$ succeeds, where no actors but the descendents of the elaborator are reduced, then the equivalence

  $$\phi_1 \wedge \psi \ \leftrightarrow\ \exists\overline{y}\,\phi_2$$

  is valid in the Oz universe, where ϕ_1 is the initial constraint of the blackboard, ϕ_2 is the final constraint of the blackboard, and $\overline{y}$ are the new local variables created during the reduction.

With the exception of

```
NewName  NewCell  Exchange  SolveCombinator  GetFiniteDomain
'Input'  'Output'  'SetThreadPriority'  'GetThreadPriority'
```

all predefined kernel procedures have a logical semantics. A logical semantics for, say `'=<'`, is

$$\texttt{intLE}(x, y)\ \vee\ \texttt{floatLE}(x, y)\ \vee\ \texttt{atomLE}(x, y).$$

4.6 Interactive Programming

So far we have assumed that computation starts from a single admissible kernel expression. It is straightforward to generalize to an incremental regime elaborating expressions arriving on a stream. To be useful, the arriving expressions must be allowed to share variables.

A **kernel program** is a kernel expression with a hole $\bullet$ defined as follows:

$$\mathcal{E} \quad ::= \quad \bullet \;\mid\; E\,\mathcal{E} \;\mid\; \texttt{local}\ \overline{x}\ \texttt{in}\ \mathcal{E}\ \texttt{end}$$

Kernel programs are compositional in that we can obtain from two programs $\mathcal{E}_1$ and $\mathcal{E}_2$ a composed program $\mathcal{E}_1[\mathcal{E}_2]$ by replacing the hole of $\mathcal{E}_1$ with $\mathcal{E}_2$. The idea is now to replace the initial expression E by a stream

$$\mathcal{E}_1[\mathcal{E}_2[\ldots\mathcal{E}_n[\texttt{true}]\ldots]]$$

of nested programs, where elaboration of the hole $\bullet$ must wait until the next program arrives. To work as expected, a substitution must be maintained for the hole $\bullet$ mapping the statically bound variables to their dynamic replacements. (Recall that elaboration of **`local`** x **`in`** E **`end`** creates an elaborator for $E[y/x]$, where y is a fresh variable replacing x within its static scope.)

A convenient syntax for entering a program is

`declare` $\overline{x}$ **`in`** E

which stands for

`local` $\overline{x}$ **`in`** $E\ \bullet$ **`end`**

5 The Oz Calculus

Oz has been designed hand in hand with a formal model consisting of the Oz universe and the Oz calculus. It is fair to say that Oz could not have been conceived without a formal model. This becomes evident, for instance, with the notion of constraint entailment, or the semantics of solvers. As Oz evolved, its formal model evolved. Ideas for new combinators evolved by trying different formulations in the calculus, which provided the ground for arguing their simplicity and generality. The solve combinator was the one that came last and took longest to evolve.

It takes intuition and effort to understand a new formal system, even if it is mathematically seen simple. This is the reason for presenting the calculus last, although it certainly comes first in our understanding of Oz. The actor model can be seen as a computational metaphor for the calculus providing motivation and intuition.

The Oz calculus models concurrent computation as rewriting of a class of expressions modulo a structural congruence. This set-up, which is also employed in more recent presentations of the π-calculus [8, 7], proves particularly useful for Oz since constraint propagation and simplification can be accommodated elegantly by means of the structural congruence.

The Oz calculus is not committed to a particular constraint system; instead, it is parameterized with respect to a general and straightforward notion of constraint system. This divide and conquer approach simplifies things considerably since we can now deal with the complexities of the Oz universe separately and independently.

In the interest of a smooth presentation, the calculus is somewhat simplified. It will be extended with the missing expressivity when the connection with the actor model is made. The extended calculus formalizes all aspects of the actor model, with the exception of the reduction strategy and input and output. Not specifying the order in which actors are reduced greatly simplifies the formal machinery.

The expressions of the Oz calculus model the computation spaces of the actor model. Thus their purpose is different from the purpose of elaborable expressions, which can only occur within elaborators. While the elaborable expressions model only static aspects of the actor model, the expressions of the calculus model both the dynamic and static aspects of the actor model. Elaborators are not modelled explicitly but are expressible with the other primitives of the calculus.

The study of the Oz calculus should be prepared by reading [13], which introduces a simplified calculus not covering constraints and search.

5.1 Constraint Systems

We base our notion of constraint system on first-order predicate logic with equality. A **constraint system** consists of

1. a signature Σ (a set of constant, function and predicate symbols)

2. a satisfiable theory Δ (a set of sentences over Σ having a model)
3. an infinite set of constants in Σ called **names** satisfying two conditions:
   (a) $\Delta \models \neg(a \doteq b)$ for every two distinct names a, b
   (b) $\Delta \models \phi \leftrightarrow \psi$ for every two sentences ϕ, ψ over Σ such that ψ can be obtained from ϕ by permutation of names.

The Oz universe defines a constraint system as follows: take its signature and its names as they are, and let the constraint theory Δ be the set of all sentences valid in the Oz universe. It is not difficult to verify that the two conditions for names are satisfied. Note that the second condition on names prevents us from having an order on names; this explains why the predicate $\texttt{arity}(x, y)$ ignores those features of x that are names.

Given a constraint system, we will call every formula over its signature a **constraint**. We use $\bot$ for the constraint that is always false, and $\top$ for the constraint that is always true. We say that a constraint ϕ **entails** a constraint ψ if $\Delta \models \phi \rightarrow \psi$, and that a constraint ϕ is **equivalent** to a constraint ψ if $\Delta \models \phi \leftrightarrow \psi$. We say that a constraint is **satisfiable** if it does not entail $\bot$.

5.2 Syntax

Figure 7 defines the syntax of the Oz calculus. The definition assumes that a constraint system is given, which fixes infinite sets of variables, names and constraints. Variables and names are jointly referred to as **references**.

We use $\overline{u}$ to denote a possibly empty sequence of references. A sequence $\overline{u}$ is called **linear** if its elements are pairwise distinct. If $\overline{u} = u_1 \ldots u_n$, we often write $\exists\overline{u}\,E$ for $\exists u_1 \ldots \exists u_n E$.

Although our notation suggests the contrary, we do distinguish between a composition $\phi_1 \wedge \phi_2$ and a conjunction $\phi_1 \wedge \phi_2$, and also between a declaration $\exists x\phi$ and an existential quantification $\exists x\phi$. If we want to make the distinction explicit, we will use the symbols $\dot{\wedge}$ for conjunction and $\dot{\exists}$ for existential quantification.

Both variables and names can be declared. Declaration of names provides for reference to fresh names.

An expression $a{:}\,\overline{x}/E$ models a binding of the name a to the abstraction $\overline{x}/E$. For convenience, we call the entire expression $a{:}\,\overline{x}/E$ an abstraction. We call a the **designator**, $\overline{x}$ the **formal arguments**, and E the **body** of the abstraction. We sometimes write $a{:}\,A$, where $A = \overline{x}/E$.

Given a cell $a{:}u$, we call a the **designator** and u the **reference** of the cell. A cell models a binding of a name to a reference.

Given an application $u\overline{v}$, we call u the **designator** and $\overline{v}$ the **actual arguments** of the application.

Symbols

x, y, z	:		*variable*	
a, b, c	:		*name*	
u, v, w	$::=$	$x \mid a$	*reference*	

Expressions

ϕ, ψ	:		*constraint*	
E	$::=$	ϕ	*constraint*	
	$\mid$	$E_1 \wedge E_2$	*composition*	
	$\mid$	$\exists u E$	*declaration*	
	$\mid$	$a{:}\,\overline{x}/E$	*abstraction*	*($\overline{x}$ linear)*
	$\mid$	$a{:}u$	*cell*	
	$\mid$	$u\overline{v}$	*application*	
	$\mid$	**if** D **else** E	*conditional*	
	$\mid$	**or** (D)	*disjunction*	
	$\mid$	**solve**$(x{:}\, E, uvw)$	*solver*	
D	$::=$	$C \mid \bot \mid D_1 \vee D_2$	*collection*	
C	$::=$	E_1 **then** $E_2 \mid \exists u C$	*clause*	

Figure 7: Syntax of the Oz calculus.

Given a solver **solve**$(x{:}\, E, uvw)$, we call x the **root variable**, E the **guard**, and u, v, w the **control references** of the solver.

Given a clause $\exists \overline{u}\,(E_1$ **then** $E_2)$, we call $\overline{u}$ the **local references**, E_1 the **guard**, and E_2 the **body** of the clause.

The syntactic category D represents multisets of clauses, where $\bot$ stands for the empty multiset and $\vee$ for multiset union.

The Oz calculus has the following binders for references:

- A declaration $\exists u E$ binds the **declared reference** u with scope E.
- An abstraction $a{:}\,\overline{x}/E$ binds its formal arguments $\overline{x}$ with scope E.
- A clausal declaration $\exists u C$ binds the local reference u with scope C.
- A solver **solve**$(x{:}\, E, uvw)$ binds its root variable x with scope E.
- Universal and existential quantification in constraints.

The **free and bound references** of expressions are defined accordingly. An expression is **closed** if it has no free variable.

The notation $E[u/x]$ stands for the expression that is obtained from E by replacing every free occurrence of x with u. The notation $E[\overline{u}/\overline{x}]$ is defined accordingly, where the elements of the sequence $\overline{x}$ are replaced simultaneously, and $\overline{x}$ is assumed to be linear.

A context is an expression having a hole $\bullet$ at a reducible position. **Contexts** are defined as follows:

$$\begin{array}{lcl}
\mathcal{E} & ::= & \bullet \mid \mathcal{E} \wedge E \mid E \wedge \mathcal{E} \mid \exists u\, \mathcal{E} \mid \textbf{if}\ \mathcal{D}\ \textbf{else}\ E \mid \textbf{or}\,(\mathcal{D}) \mid \textbf{solve}(x{:}\,\mathcal{E}, uvw) \\
\mathcal{D} & ::= & \mathcal{C} \mid \mathcal{D} \vee D \mid D \vee \mathcal{D} \\
\mathcal{C} & ::= & \mathcal{E}\ \textbf{then}\ E \mid \exists u \mathcal{C}.
\end{array}$$

We write $\mathcal{E}[E]$ for the expression obtained by replacing the hole in the context $\mathcal{E}$ with the expression E (capturing of free variables in E is OK). An expression E is called **free for a context** $\mathcal{E}$ if no free reference of E is captured at the position of the hole in $\mathcal{E}$.

5.3 Structural Congruence

A **congruence** is an equivalence relation on the expressions of the Oz calculus (i.e., the syntactic categories ϕ, E, D, and C) that is compatible with all syntactic combinators (e.g., if $E_1 \equiv E_1'$ and $E_2 \equiv E_2'$, then $E_1 \wedge E_2 \equiv E_1' \wedge E_2'$). The **structural congruence** $E_1 \equiv E_2$ of the Oz calculus is defined as the least congruence satisfying the congruence laws in Figure 8.

Proposition 5.1 *Given two constraints ϕ_1 and ϕ_2, the composition $\phi_1 \wedge \phi_2$ is congruent to the conjunction $\phi_1 \dot\wedge \phi_2$.*

Proof. We have $\phi_1 \wedge \phi_2 \equiv \phi_1 \wedge (\phi_1 \dot\wedge \phi_2) \equiv (\phi_1 \dot\wedge \phi_2) \wedge \phi_1 \equiv (\phi_1 \dot\wedge \phi_2) \wedge \top \equiv \phi_1 \dot\wedge \phi_2$ by relative simplification, commutativity of composition, relative simplification, and neutrality of $\top$. □

For declaration and existential quantification an analogous proposition does not hold.

An expression E is called **failed** if $E \equiv \bot \wedge E$. A clause $\exists\overline{u}\,(E_1\ \textbf{then}\ E_2)$ is called **failed** if E_1 is failed. A collection D is called **failed** if D is $\bot$, or D is a failed clause, or $D = D_1 \vee D_2$, where both D_1 and D_2 are failed.

An expression E is called **nilpotent** if it has the form

$$\exists\overline{x}\exists\overline{a}\exists\overline{b}\exists\overline{c}(\phi \wedge \overline{a}{:}\,\overline{A} \wedge \overline{b}{:}\overline{u})$$

where $\Delta \models \exists\overline{x}\phi$. (The notation $\overline{a}{:}\,\overline{A}$ stands for a composition $a_1{:}\,A_1 \wedge \ldots \wedge a_n{:}\,A_n$ of abstractions, and $\overline{b}{:}\overline{u}$ stands for a composition of cells.)

Renaming

- $E_1 \equiv E_2$ if E_1 and E_2 are equal up to renaming of bound references

Composition and Collection

- $\wedge$ is associative, commutative and satisfies $E \wedge \top \equiv E$
- $\vee$ is associative, commutative and satisfies $D \vee \bot \equiv D$

Declaration

- $\exists u \exists v E \;\equiv\; \exists v \exists u E$
- $\exists u E_1 \wedge E_2 \;\equiv\; \exists u\,(E_1 \wedge E_2)$ if u does not occur free in E_2

Relative Simplification

- $\phi_1 \wedge \mathcal{E}[\phi_2] \;\equiv\; \phi_1 \wedge \mathcal{E}[\phi_2']$ if ϕ_1 is free for $\mathcal{E}$ and $\Delta \models \phi_1 \wedge \phi_2 \leftrightarrow \phi_1 \wedge \phi_2'$

Equality

- $x \doteq u \wedge E \;\equiv\; x \doteq u \wedge E[u/x]$ if u is free for x in E

Figure 8: Structural congruence in the Oz calculus.

5.4 Induced Constraints

The Relative Simplification Law makes it possible to propagate constraints in an expression downward, provided no free variables of the constraint are captured. Given a context, whose bound references are renamed apart, there is a strongest constraint (unique up to equivalence) that can be propagated to the hole. Below we define this induced constraint for a class of contexts that is exhaustive modulo structural congruence.

An expression is called **basic** if it is no constraint, composition or declaration.

The following defines a partial function from constraints and contexts to constraints:

$$
\begin{array}{rcll}
\mathrm{Ind}_\phi\,(\bullet) & = & \phi & \\
\mathrm{Ind}_\phi\,(\psi \wedge \mathcal{E}) & = & \mathrm{Ind}_{\phi\wedge\psi}\,(\mathcal{E}) & \\
\mathrm{Ind}_\phi\,(E \wedge \mathcal{E}) & = & \mathrm{Ind}_\phi\,(\mathcal{E}) & \text{if } E \text{ basic} \\
\mathrm{Ind}_\phi\,(\exists u \mathcal{E}) & = & \mathrm{Ind}_\phi\,(\mathcal{E}) & \text{if } u \text{ is not free in } \phi \\
\mathrm{Ind}_\phi\,(\textbf{if } \mathcal{D} \textbf{ else } E) & = & \mathrm{Ind}_\phi\,(\mathcal{D}) & \\
\mathrm{Ind}_\phi\,(\textbf{or}\,(\mathcal{D})) & = & \mathrm{Ind}_\phi\,(\mathcal{D}) & \\
\mathrm{Ind}_\phi\,(\textbf{solve}(x{:}\,\mathcal{E}, uvw)) & = & \mathrm{Ind}_\phi\,(\mathcal{E}) & \text{if } x \text{ is not free in } \phi
\end{array}
$$

$$\begin{array}{rcll}
\mathrm{Ind}_\phi\,(\mathcal{C} \vee D) & = & \mathrm{Ind}_\phi\,(\mathcal{C}) & \\
\mathrm{Ind}_\phi\,(\mathcal{E}\ \textbf{then}\ E) & = & \mathrm{Ind}_\phi\,(\mathcal{E}) & \\
\mathrm{Ind}_\phi\,(\exists u \mathcal{C}) & = & \mathrm{Ind}_\phi\,(\mathcal{C}) & \quad \text{if } u \text{ is not free in } \phi.
\end{array}$$

If $\mathrm{Ind}_\phi\,(\mathcal{E})$ is defined, we call $\mathrm{Ind}_\phi\,(\mathcal{E})$ the constraint **induced by $\mathcal{E}$ under ϕ**. If $\mathrm{Ind}_\top\,(\mathcal{E})$ is defined, we call $\mathrm{Ind}_\top\,(\mathcal{E})$ the constraint **induced by $\mathcal{E}$**.

Proposition 5.2 *If $\mathcal{E}$ induces ψ under ϕ, then $\mathcal{E}[\psi] \wedge \phi \equiv \mathcal{E}[\top] \wedge \phi$.*

The next proposition says that our definition of induced constraints is exhaustive on contexts modulo structural congruence. Structural congruence on contexts is the least congruence on contexts satisfying all congruence laws in Figure 8 rewritten for contexts. Note that $\mathcal{E} \equiv \mathcal{E}'$ does not imply $\mathcal{E}[E] \equiv \mathcal{E}'[E]$ (because of the Renaming Law).

Proposition 5.3 *For every context $\mathcal{E}$ and every constraint ϕ there exists a context $\mathcal{E}'$ such that $\mathcal{E} \equiv \mathcal{E}'$ and the induced constraint of $\mathcal{E}'$ under ϕ is defined.*

We say that a context $\mathcal{E}$ is **admissible** if $\mathcal{E}[\bot] \not\equiv \mathcal{E}[\top]$.

Proposition 5.4 *A context $\mathcal{E}$ is admissible if and only if there exists a context $\mathcal{E}'$ and a satisfiable constraint ϕ such that $\mathcal{E} \equiv \mathcal{E}'$ and $\mathcal{E}'$ induces ϕ.*

5.5 Reduction

The reduction relation $E \rightarrow E'$ of the Oz calculus is defined by the inference system in Figure 9 and the following definitions:

- An expression E is called **reducible** if there exists an expression E' such that $E \rightarrow E'$.
- An expression E is called **stable** if, for every abstraction and for every satisfiable constraint π, the expression $\pi \wedge E$ is neither reducible nor failed. (The definition is by induction on the number of nested solvers in E.)
- An expression E is called **distributable** if there exist $\overline{u}$, E' and D such that $E \equiv \exists \overline{u}\,(E' \wedge \textbf{or}\,(D))$.

A **finite computation** issuing from a closed expression E_1 is a sequence $E_1, \ldots, E_n$ of closed expressions such that the final expression E_n is irreducible and $E_i \rightarrow E_{i+1}$ for all i. Since failure prevents further reduction, none of the expressions $E_1, \ldots, E_{n-1}$ can be failed.

An **infinite computation** issuing from a closed expression E_1 is an infinite sequence E_1, E_2, E_3, ... of closed expressions such that $E_i \rightarrow E_{i+1}$ for all i. Since failure prevents further reduction, none of the expressions E_i can be failed.

Structure

- $\dfrac{E \;\equiv\; \mathcal{E}[E_1] \quad E_1 \overset{\phi}{\Rightarrow} E_1' \quad \mathcal{E}[E_1'] \;\equiv\; E'}{E \;\to\; E'}$ if $\mathcal{E}$ is admissible and induces ϕ
- $\dfrac{E \;\Rightarrow\; E'}{E \overset{\phi}{\Rightarrow} E'}$

Application

- $\mathcal{E}\,[a\overline{u}] \;\wedge\; a{:}\,\overline{x}/E \;\overset{\phi}{\Rightarrow}\; \mathcal{E}\,[E\,[\overline{u}/\overline{x}]] \;\wedge\; a{:}\,\overline{x}/E$

  if $\mathcal{E} \wedge \phi$ is admissible, $\overline{x}$ and $\overline{u}$ have equal length,
  $\overline{u}$ is free for $\overline{x}$ in E, and $a{:}\,\overline{x}/E$ is free for $\mathcal{E}$

Exchange

- $avw \;\wedge\; a{:}u \;\Rightarrow\; v \doteq u \;\wedge\; a{:}w$

Conditional

- **if** $\exists\overline{u}\,(E_1$ **then** $E_2) \vee D$ **else** $E_3 \;\Rightarrow\; \exists\overline{u}\,(E_1 \wedge E_2)$ if $\exists\overline{u}\,E_1$ nilpotent
- **if** D **else** $E \;\Rightarrow\; E$ if D failed

Disjunction

- **or** $(\exists\overline{u}\,(E_1$ **then** $E_2) \vee D) \;\Rightarrow\; \exists\overline{u}\,(E_1 \wedge E_2)$ if D failed
- **or** $(\exists\overline{u}\,(E$ **then** $\top) \vee D) \;\Rightarrow\; \top$ if $\exists\overline{u}\,E$ nilpotent

Solver

- **solve**$(x{:}\,E, uvw) \;\Rightarrow\; u$ if E failed
- **solve**$(x{:}\,E, uvw) \;\Rightarrow\; \exists a \exists b(vab \wedge a{:}\,x/E_2 \wedge b{:}\,x/E_3)$

  if $\exists x E$ is stable, $E = \exists\overline{u}\,(E_1 \wedge$ **or** $(C \vee D))$,
  $E_2 = \exists\overline{u}\,(E_1 \wedge$ **or** $(C))$, and $E_3 = \exists\overline{u}\,(E_1 \wedge$ **or** $(D))$
- **solve**$(x{:}\,E, uvw) \;\Rightarrow\; \exists a(wa \wedge a{:}\,x/E)$ if $\exists x E$ stable and not distributable

Annulment

- $E \;\Rightarrow\; \top$ if E nilpotent and $E \not\equiv \top$

Figure 9: Reduction in the Oz calculus.

As one would expect of a concurrent computation model with indeterministic choice, there are closed expressions that permit both finite and infinite computations. For instance:

$$\exists a\,(a{:}/a \;\wedge\; \textbf{if}\ \top\ \textbf{then}\ a \;\vee\; \top\ \textbf{then}\ \top\ \textbf{else}\ \top).$$

The following proposition says that conditionals can reduce with clauses whose guards are entailed.

Proposition 5.5 *Suppose ϕ_1 is satisfiable and entails $\exists\overline{x}\phi_2$. Then*

$$\phi_1 \;\wedge\; \textbf{if}\ \exists\overline{x}(\phi_2\ \textbf{then}\ E_1)\ \textbf{else}\ E_2 \quad\to\quad \phi_1 \;\wedge\; \exists\overline{x}(\phi_2 \;\wedge\; E_1).$$

Proof. It will be convenient to use the congruence relation (on constraints)

$$\phi \models\!\!\!\dashv_\Delta \psi \quad :\Longleftrightarrow \quad \Delta \models \phi \leftrightarrow \psi.$$

Because of the Renaming Law we can assume without loss of generality that no variable in $\overline{x}$ occurs in ϕ_1. It suffices to show that there exists a constraint ϕ_3 such that $\phi_1 \wedge \phi_2 \models\!\!\!\dashv_\Delta \phi_1 \wedge \phi_3$ and $\exists\overline{x}\phi_3 \models\!\!\!\dashv_\Delta \top$ since

$$\begin{aligned}
\phi_1 \;\wedge\; \textbf{if}\ \exists\overline{x}(\phi_2\ \textbf{then}\ E_1)\ \textbf{else}\ E_2 &\equiv \phi_1 \;\wedge\; \textbf{if}\ \exists\overline{x}(\phi_3\ \textbf{then}\ E_1)\ \textbf{else}\ E_2\\
&\to \phi_1 \;\wedge\; \exists\overline{x}(\phi_3 \;\wedge\; E_1)\\
&\equiv \phi_1 \;\wedge\; \exists\overline{x}(\phi_2 \;\wedge\; E_1).
\end{aligned}$$

by the Relative Simplification Law, the first reduction rule for conditionals, and once more the Relative Simplification Law. Let $\phi_3 := \phi_1 \to \phi_2$ (here $\to$ is implication, not reduction). Then $\phi_1 \wedge \phi_2 \models\!\!\!\dashv_\Delta \phi_1 \wedge \phi_3$ is obviously satisfied. Moreover, $\exists\overline{x}\phi_3 \models\!\!\!\dashv_\Delta \exists\overline{x}(\phi_1 \to \phi_2) \models\!\!\!\dashv_\Delta \phi_1 \to \exists\overline{x}\phi_2 \models\!\!\!\dashv_\Delta \top$. □

5.6 Examples

The following examples give a first impression of how the Oz calculus models concurrent computation.

Example 5.6 Consider the expression

$$\exists x\exists y(\exists a(x \doteq a) \;\wedge\; \exists a(y \doteq a) \;\wedge\; \textbf{if}\ x \doteq y\ \textbf{then}\ E_1\ \textbf{else}\ E_2)$$

and suppose that x and y are distinct variables that do not occur free in E_1 and E_2. We will show that this expression reduces in four steps to E_2.

First we move the left declaration of the name a to the outside of the expression using the congruence laws for declarations and compositions.

$$\equiv\ \exists a\exists x\exists y(x \doteq a \;\wedge\; \exists a(y \doteq a) \;\wedge\; \textbf{if}\ x \doteq y\ \textbf{then}\ E_1\ \textbf{else}\ E_2)$$

This is of course only possible if a does not occur free in E_1 or E_2. Should this be the case, renaming a within its scope as justified by the Renaming Law is necessary. Next we apply the Equality Law to $x \doteq a$.

$$\equiv\ \exists a \exists x \exists y (x \doteq a\ \wedge\ \exists a(y \doteq a)\ \wedge\ \textbf{if}\ a \doteq y\ \textbf{then}\ E_1\ \textbf{else}\ E_2)$$

Now we move the declaration of x inside using the laws for composition and declaration (we exploit that x does not occur free in E_1 and E_2 and that x is different from y).

$$\equiv\ \exists a \exists y (\exists x (x \doteq a)\ \wedge\ \exists a(y \doteq a)\ \wedge\ \textbf{if}\ a \doteq y\ \textbf{then}\ E_1\ \textbf{else}\ E_2)$$

Since $\exists x(x \doteq a)$ is nilpotent, we can delete $\exists x(x \doteq a)$ using the Annulment Rule and the laws for compositions (in particular $E \wedge \top \equiv E$).

$$\rightarrow \exists a \exists y (\exists a(y \doteq a)\ \wedge\ \textbf{if}\ a \doteq y\ \textbf{then}\ E_1\ \textbf{else}\ E_2)$$

Next we rename the inner name a to the different name b using the Renaming Law.

$$\equiv\ \exists a \exists y (\exists b(y \doteq b)\ \wedge\ \textbf{if}\ a \doteq y\ \textbf{then}\ E_1\ \textbf{else}\ E_2)$$

This brings us in a position where we can eliminate $\exists b(y \doteq b)$ in the same way we did it before for $\exists a(x \doteq a)$.

$$\rightarrow\ \exists a \exists b (\textbf{if}\ a \doteq b\ \textbf{then}\ E_1\ \textbf{else}\ E_2)$$

Now, since $a \doteq b$ is failed, we obtain

$$\rightarrow\ \exists a \exists b E_2$$

using the second rule for conditionals. It remains to get rid of the declarations of the names a and b. This can be done using the Annulment Rule together with the laws for compositions and declarations:

$$\equiv\ \exists a \exists b (\top \wedge E_2)\ \equiv\ (\exists a \exists b \top)\ \wedge\ E_2\ \rightarrow\ \top\ \wedge\ E_2\ \equiv\ E_2.$$

□

Example 5.7 Nilpotence and relative simplification model entailment of clauses in the presence of local abstractions and cells. For instance, consider the reduction

$$\begin{array}{rl} & y \doteq b\ \wedge\ \textbf{if}\ \exists x\ (\exists a(x \doteq a\ \wedge\ y \doteq b\ \wedge\ a{:}\, y/y \doteq x)\ \textbf{then}\ E_1)\ \textbf{else}\ E_2 \\ \rightarrow & y \doteq b\ \wedge\ \exists x\ (\exists a(x \doteq a\ \wedge\ a{:}\, y/y \doteq x)\ \wedge\ E_1) \end{array}$$

which is justified by relative simplification, the first rule for conditionals, and the fact that $\exists x \exists a(x \doteq a\ \wedge\ a{:}\, y/y \doteq x)$ is nilpotent. □

5.7 Relationship with the Actor Model

The expressions of the Oz calculus model the computation spaces of the actor model, provided we take the Oz universe as the constraint system underlying the calculus. Conditionals, disjunctions, and solvers model the respective proper actors. However, we need to extend the calculus so that it

- can express nondistributing disjunctions
- can express once-only abstractions
- captures solvers fully (need to provide status and to return alternatives as once-only abstractions)
- can express elaborators.

Nondistributing disjunctions are incorporated easily: they have the same reduction rules as distributing disjunctions, but they are not distributed by solvers.

Once-only abstractions are incorporated by extending the expressions of the calculus with the form $a{:}\,\overline{x}/\!/E$ and the reduction relation with the rule

- $\mathcal{E}\,[a\overline{u}] \,\wedge\, a{:}\,\overline{x}/\!/E \;\overset{\phi}{\Rightarrow}\; \mathcal{E}\,[E\,[\overline{u}\,/\overline{x}]] \,\wedge\, a{:}\,\overline{x}/\!/\bot$
  if $\mathcal{E} \wedge \phi$ is admissible, $\overline{x}$ and $\overline{u}$ have equal length,
  $\overline{u}$ is free for $\overline{x}$ in E, and $a{:}\,\overline{x}/E$ is free for $\mathcal{E}$.

Moreover, it is necessary to strengthen the notion of nilpotence to all expressions of the form

$$\exists\overline{x}\exists\overline{a}\exists\overline{b}\exists\overline{c}\exists\overline{d}(\phi \,\wedge\, \overline{a}{:}\,\overline{A} \,\wedge\, \overline{b}{:}\overline{u} \,\wedge\, \overline{c}{:}\,\overline{B})$$

where $\Delta \models \exists\overline{x}\phi$ and $\overline{c}{:}\,\overline{B}$ stands for a conjunction of once-only abstractions.

Now it is easy to modify the second and third reduction rule for solvers such that the solvers of the actor model are faithfully modelled.

To model elaborators, we first extend the expressions of the calculus with **det**(u) and **getDomain**(u, v). The semantics of **det**(u) is captured by the reduction rule

- **det**$(u) \;\overset{\phi}{\Rightarrow}\; \top$
  if there exists a constant c such that ϕ entails $u \doteq c$.

The semantics of **getDomain**(u, v) is captured by

- **getDomain**$(x, u) \;\overset{\phi}{\Rightarrow}\; u \doteq n_1|\ldots|n_k|\texttt{nil}$
  if $n_1, \ldots, n_k$ is the shortest list of nonnegative integers in ascending order such that ϕ entails $x \doteq n_1 \vee \ldots \vee x \doteq n_k$.

How do we model an elaborator for a constraint ϕ? It cannot be modeled by the constraint ϕ itself since this would impose ϕ immediately (consider the inconsistent constraint $\bot$). However, the conditional **if** $\top$ **then** ϕ **else** $\top$ behaves exactly like an elaborator for ϕ: Only when it is reduced, the constraint ϕ is imposed.

In the following we write $\langle E\rangle$ for **if** $\top$ **then** E **else** $\top$.

An elaborable expression is called **translatable** if it does not contain subexpressions of the form `input`$[x]$, `output`$[x]$, `setThreadPriority`$[x]$, or `getThreadPriority`$[x]$.

The function $[\![E]\!]$ translates a translatable elaborable expression E into an expression of the extended Oz calculus. The translation is such that $[\![E]\!]$ models an elaborator for E.

$$\begin{array}{rcl}
[\![\phi]\!] & = & \langle\phi\rangle \\
[\![x\colon\overline{y}/E]\!] & = & \langle\exists a(x \doteq a \wedge a\colon\overline{y}/[\![E]\!])\rangle \\
[\![x\colon\overline{y}/\!/E]\!] & = & \langle\exists a(x \doteq a \wedge a\colon\overline{y}/\!/[\![E]\!])\rangle \\
[\![x\colon y]\!] & = & \langle\exists a(x \doteq a \wedge a\colon y)\rangle \\
[\![E_1\ E_2]\!] & = & \langle[\![E_1]\!] \wedge [\![E_2]\!]\rangle \\
[\![\texttt{local}\ \overline{x}\ \texttt{in}\ E\ \texttt{end}]\!] & = & \langle\exists\overline{x}\,[\![E]\!]\rangle \\
[\![\texttt{newName}[x]]\!] & = & \langle\exists a(x \doteq a)\rangle \\
[\![\texttt{apply}[x\overline{y}]]\!] & = & x\overline{y} \\
[\![\texttt{if}\ C_1\ \texttt{[]}\ \ldots\ \texttt{[]}\ C_n\ \texttt{else}\ E\ \texttt{fi}]\!] & = & \langle\textbf{if}\ [\![C_1]\!] \vee \ldots \vee [\![C_n]\!]\ \textbf{else}\ [\![E]\!]\rangle \\
[\![\texttt{or}\ C_1\ \texttt{[]}\ \ldots\ \texttt{[]}\ C_n\ \texttt{ro}]\!] & = & \langle\textbf{or}\,([\![C_1]\!] \vee \ldots \vee [\![C_n]\!])\rangle \\
[\![\texttt{OR}\ C_1\ \texttt{[]}\ \ldots\ \texttt{[]}\ C_n\ \texttt{RO}]\!] & = & \langle\textbf{OR}\,([\![C_1]\!] \vee \ldots \vee [\![C_n]\!])\rangle \\
[\![\texttt{solve}[x\colon E, y_1y_2y_3]]\!] & = & \langle\textbf{solve}(x\colon [\![E]\!], y_1y_2y_3)\rangle \\
[\![\texttt{det}[x]]\!] & = & \textbf{det}(x) \\
[\![\texttt{getDomain}[x,y]]\!] & = & \textbf{getDomain}(x,y) \\
[\![\overline{x}\ \texttt{in}\ E_1\ \texttt{then}\ E_2]\!] & = & \exists\overline{x}\,([\![E_1]\!]\ \textbf{then}\ [\![E_2]\!]) \qquad \text{if } E_2 \neq \top \\
[\![\overline{x}\ \texttt{in}\ E\ \texttt{then}\ \top]\!] & = & \exists\overline{x}\,([\![E]\!]\ \textbf{then}\ \top)
\end{array}$$

The special treatment of the constraint $\top$ in clause bodies is needed so that reduction of disjunctions by clause entailment is modelled correctly.

We can now state the relationship between the actor model and the Oz calculus. For every closed and translatable elaborable expression E we have the following:

- If there is a finite computation issuing from E in the actor model, then there is a finite computation issuing from $[\![E]\!]$ in the extended Oz calculus.

- If there is an infinite computation issuing from E in the actor model, then there is an infinite computation issuing from $[\![E]\!]$ in the extended Oz calculus.

The converse of each of the two statements is wrong in general. This is because the reduction strategy employed by the actor model excludes some of the computations of the calculus.

References

[1] Hassan Aït-Kaci, Andreas Podelski, and Gert Smolka. A feature-based constraint system for logic programming with entailment. *Theoretical Computer Science*, 122(1–2):263–283, January 1994.

[2] Rolf Backofen and Gert Smolka. A complete and recursive feature theory. *Theoretical Computer Science*. To appear 1995; a preliminary version is available as DFKI Research Report RR-92-30, German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany.

[3] Rolf Backofen and Ralf Treinen. How to win a game with features. In Jean-Pierre Jouannaud, editor, *1st International Conference on Constraints in Computational Logics*, Lecture Notes in Computer Science, vol. 845, pages 320–335, München, Germany, 7–9 September 1994. Springer-Verlag.

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

[5] Martin Henz and Martin Müller. Programming in Oz. DFKI Oz documentation series, German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, 1994.

[6] Carl Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8:323–364, 1977.

[7] Robin Milner. The polyadic π-calculus: A tutorial. ECS-LFCS Report Series 91-180, Laboratory for Foundations of Computer Science, University of Edinburgh, Edinburgh EH9 3JZ, October 1991.

[8] Robin Milner. Functions as processes. *Journal of Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[9] Tobias Müller, Konstantin Popow, Christian Schulte, and Jörg Würtz. Constraint programming in Oz. DFKI Oz documentation series, German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, 1994.

[10] 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*, Ithaca, New York, USA, November 1994. MIT-Press. To appear.

[11] Christian Schulte, Gert Smolka, and Jörg Würtz. Encapsulated search and constraint programming in Oz. In A.H. Borning, editor, *Second Workshop on Principles and Practice of Constraint Programming*, Lecture Notes in Computer

Science, vol. 874, Orcas Island, Washington, USA, 2-4 May 1994. Springer-Verlag.

[12] Gert Smolka. A calculus for higher-order concurrent constraint programming with deep guards. Research Report RR-94-03, German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, February 1994.

[13] Gert Smolka. A foundation for higher-order concurrent constraint programming. In Jean-Pierre Jouannaud, editor, *1st International Conference on Constraints in Computational Logics*, Lecture Notes in Computer Science, vol. 845, pages 50–72, München, Germany, 7–9 September 1994. Springer-Verlag.

[14] Gert Smolka. An Oz primer. DFKI Oz documentation series, German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, 1994.

[15] Gert Smolka and Ralf Treinen. Records for logic programming. *Journal of Logic Programming*, 18(3):229–258, April 1994.

[16] Ralf Treinen. Feature constraints with first-class features. In Andrzej M. Borzyszkowski and Stefan Sokołowski, editors, *Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science, vol. 711, pages 734–743, Gdańsk, Poland, 30 August–3 September 1993. Springer-Verlag.

Index