Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

Research
Report
RR-95-08

An Abstract Machine for Oz

Michael Mehl, Ralf Scheidhauer, and Christian Schulte

June 1995

An Abstract Machine for Oz

Michael Mehl, Ralf Scheidhauer, and Christian Schulte

DFKI-RR-95-08

This work appears in: Programming Languages, Implementations, Logics and Programs, Seventh International Symposium, PLILP'95, Springer-Verlag, LNCS, September 20-22, 1995, Utrecht, The Netherlands.

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

ISSN 0946-008X

An Abstract Machine for Oz

Michael Mehl, Ralf Scheidhauer, and Christian Schulte
Programming Systems Lab
German Research Center for Artificial Intelligence (DFKI)
Stuhlsatzenhausweg 3, D–66123 Saarbrücken, Germany
{mehl,scheidhr,schulte}@dfki.uni-sb.de

June 27, 1995

Abstract

Oz is a concurrent constraint language providing for first-class procedures, concurrent objects, and encapsulated search. DFKI Oz is an interactive implementation of Oz competitive in performance with commercial Prolog and Lisp systems. This paper describes AMOZ, the abstract machine underlying DFKI Oz. AMOZ implements rational tree constraints, first-class procedures, local computation spaces for deep guards, and preemptive and fair threads.

Contents

1 Introduction

Oz is a concurrent constraint language [20, 19, 17, 6, 22] providing for functional, object-oriented, and constraint programming. It has a simple yet powerful computation model [19, 20], which extends the concurrent constraint model [10, 16] by first-class procedures, deep guards, concurrent state, and encapsulated search.

DFKI Oz [11] is an interactive implementation of Oz based on an incremental compiler and an abstract machine. It features a programming interface based on GNU Emacs, an object-oriented interface to Tcl/Tk, powerful interoperability features, a garbage collector, and support for stand-alone applications. Performance is competitive with commercial Prolog and Lisp systems.

This paper describes an abstract machine, called AMOZ, which covers important aspects of the DFKI Oz abstract machine. AMOZ implements rational tree constraints, first-class procedures, deep guards, and threads, leaving aside mutable state for objects [19], record constraints [21], as well as finite domain constraints and encapsulated search [18].

Constraint Store. By the very idea of concurrent constraint programming, computation emerges from adding constraints to a store. In this paper, we consider constraints over rational trees (as in Prolog II [5]) that enjoy a *variable-centered* normal form: adding constraints results in binding variables. This is utilized in AMOZ: binding variables triggers procedure application, reduction of conditionals, and readiness of threads.

First-class Procedures. Oz provides for first-class procedures typical of modern functional languages (e.g., Haskell [7], Scheme [3], and SML [12]). First-class procedures in Oz support higher-order functional programming [19], concurrent object-oriented programming [6], and encapsulated search [17]. In AMOZ, execution of a procedure definition dynamically creates a procedure (called closure in functional languages) and stores the procedure under a so-called name. Procedure application is triggered by binding a variable to a name, from which the procedure to be applied is retrieved.

Deep Guards. Deep guards allow any expression in the guard of a conditional. Reduction of a deep guard is done in a local computation space. The main point of discussing deep guards here is to show implementation techniques for local computation spaces. Local computation spaces are needed to encapsulate search, and encapsulation of search is a must in a concurrent and reactive language. It is well known that the problem has not been solved in the Japanese Fifth Generation Project, leaving them with two incompatible language designs: concurrent logic programming and (constraint) logic programming. AKL was the first language that solved this problem, employing a design based on deep guards [8]. Oz, on the other side, employs a higher-order search combinator that uses local computation spaces but does not presuppose deep guards [17].

Threads. Languages like Prolog II and AKL have a single thread of control in which all computations are performed. However, this is insufficient for the fine grained concurrency found in concurrent constraint languages. Since general fairness does not seem practical, Oz provides for multiple threads that are scheduled fairly. In AMOZ threads are lightweight, implemented as multiple stacks of tasks that are scheduled preemptively and fairly.

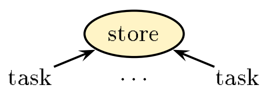
The design of abstract machines for constraint based languages has been pioneered by the Warren Abstract Machine (WAM) [25, 1]. The implementation of DFKI Oz has been influenced by the AGENTS implementation of the concurrent constraint language AKL [8]. AKL is a deep guard language providing for encapsulated search. However, AKL does not provide for first-class procedures and threads. `cc(FD)` [23] is a constraint programming language specialized for finite domain constraints. It is a compromise between a flat and a deep guard language in that combinators (i.e., cardinality, disjunction, and implication) can be nested in guards, but procedure applications cannot. As AKL, it does not support first-class procedures and threads.

The paper is organized as follows. Section 2 gives an informal presentation of the computation model, and Sect. 3 gives an example. Section 4 shows unification for rational trees, and Sect. 5 introduces AMOZ. Threads and a limited case of conditional are introduced in Sect. 6. Section 7 extends the abstract machine for local computation spaces. Procedures are introduced in Sect. 8.

2 An Informal Computation Model

This section gives an informal presentation of the computation model underlying the sublanguage of Oz considered in this paper. A full description of Oz's computation model can be found in [20].

The notion of a *computation space* is central to the computation model. A computation space consists of a number of *tasks*[1] connected to a *store*.



Computation proceeds by reducing tasks with respect to the information contained in the store. A task is reduced as soon as the store contains sufficient information. When a task is reduced new information may be written to the store or new tasks may be created. Tasks are short-lived: they cease to exist once they are reduced. Some tasks may spawn local computation spaces, thus creating a tree of computation spaces. As computation proceeds, new local computation spaces are created and existing spaces are removed or merged with their parent space.

The store consists of a *constraint store* and a *procedure store*. The constraint

[1]In other papers on Oz tasks are called actors.

store contains constraints $x = y$ and $x = f(\overline{y})$ in a normal form. The constraint store grows monotonically. The constraints are interpreted in a fixed first-order structure, called the *universe*. The universe contains rational trees (as in Prolog II [4, 5]), an extension to records is straightforward [21]. Suppose that ϕ is the conjunction of all constraints in the store. We say that the *store entails* a constraint ψ, if $\phi \to \psi$ is valid in the universe. The procedure store contains the bindings of names to procedures (to be explained later).

The tree of computation spaces satisfies the invariant that constraints of a local computation space entail constraints of their parent space ("local spaces know the constraints of global spaces"). A constraint is *imposed* by adding it to the local store and all stores below in the tree of computation spaces. Hence, imposition maintains the invariant on the tree of spaces. A computation space *fails*, if a constraint is imposed such that the constraints in the store become unsatisfiable in the universe. If a computation space fails, all spaces below fail. If a space fails, all its tasks are discarded.

There are two kinds of tasks: *elaborators* and *conditional tasks*. An elaborator is a task that executes an expression. Expressions are:

E, F, G	$::=$	$x = y \quad \mid \quad x = f(\overline{y})$	*constraints*
	$\mid$	**local** x **in** E **end**	*declaration*
	$\mid$	$E\,F$	*composition*
	$\mid$	**proc** $\{x\,\overline{y}\}$ E **end**	*procedure definition*
	$\mid$	$\{x\,\overline{y}\}$	*procedure application*
	$\mid$	**if** $\overline{x}$ **in** E **then** F **else** G **fi**	*conditional*

Elaboration of a *constraint* $x = y$ or $x = f(\overline{y})$ imposes it.

Elaboration of a *declaration* **local** x **in** E **end** creates a new variable local to the computation space and an elaborator for E. Within E the new variable is referred to by x. The space is called the *home* of x. Declaration of multiple variables **local** $x\,\overline{y}$ **in** E **end** abbreviates **local** x **in local** $\overline{y}$ **in** E **end end**.

Elaboration of a *composition* $E\,F$ creates separate elaborators for E and F.

Elaboration of a *procedure definition* **proc** $\{x\,\overline{y}\}$ E **end** chooses a new name a, writes the binding $a : \overline{y}/E$ to the procedure store, and creates an elaborator for the constraint $x = a$. A *name* is a constant in the universe. There are infinitely many different names. Since procedures are associated with new names when they are written to the procedure store, a name cannot refer to more than one procedure.

Elaboration of a *procedure application* $\{x\,y_1 \cdots y_n\}$ waits until there is a name a, such that the constraint store entails $x = a$ and the procedure store contains a binding $a : z_1 \cdots z_n/E$. When this is the case, an elaborator for the expression $E[y_1/z_1, \ldots, y_n/z_n]$, where the formal parameters have been replaced by the actual parameters, is created.

The elaboration of a *conditional* is more involved. We will proceed in two steps. First, we consider the special case **if** $\overline{y}$ **in** $x = f(\overline{y})$ **then** F **else** G **fi**, where variables in $\overline{y}$ are pairwise different ("pattern matching"). This case is especially instructive for AMOZ in Sect. 6. Its elaboration creates a *conditional task*. The conditional task waits until the store either entails $\exists\overline{y}\; x = f(\overline{y})$, in which case an elaborator for **local** $\overline{y}$ **in** $x = f(\overline{y})\; F$ **end** is created, or entails $\neg\exists\overline{y}\; x = f(\overline{y})$, in which case an elaborator for G is created.

The general conditional **if**$\overline{x}$**in**E**then**F**else**G**fi** subsumes the previous simplified case. Its elaboration creates a conditional task spawning a local computation space. We call the expression $\overline{x}$ **in** E the *guard* of the conditional. A guard is called *deep* if E is not a constraint. The local computation space is created with a store containing the constraints from the parent store and an elaborator for **local** $\overline{x}$ **in** E **end**.

We say that the guard is *entailed* if its associated computation space S is not failed, S has no tasks left, and its parent store entails $\exists\overline{y}\;\phi$, where $\overline{y}$ are the local variables of S and ϕ is the conjunction of constraints of S's store. Due to the monotonic growth of the constraint store, entailment of a guard is a stable property, i.e., it continues to hold when computation proceeds. A conditional task must wait until its guard is either entailed or failed (i.e., its corresponding local computation space is failed). If the guard is failed, the conditional task reduces to an elaborator for the expression G (its **else** constituent). If the guard is entailed, the constraints of the local store are merged with its parent store's constraints. Merging amounts to changing the local variables' home space to the parent space. By this, local variable bindings are made global. Then, the expression F (its **then** constituent) is elaborated.

So far we have not made any assumptions about the order in which tasks are executed. Such assumptions are necessary, however, so that one can write fair and efficient programs. Without such assumptions a single infinite computation, e.g., a data base query server, which is intended to run forever, could lead to starvation of all other computations.

A *thread* is a nonempty sequence of tasks. Each task belongs to exactly one thread. When a computation space is failed, its tasks are discarded, which includes their removal from the threads they reside on.

A thread can *run* by reducing its first task if it is reducible, or otherwise by moving its first task to a newly created thread. Reducing a task on a thread means to reduce the task and replace it with the possibly empty sequence of tasks it has reduced to. The order of replacing tasks is defined as follows. For the task of a composition $E\,F$ the task for E goes before the task for F. For the task of a conditional, the task for the guard goes before the task of the conditional itself.

If a thread contains a single not yet reducible task it is called *suspended*, and *runnable* otherwise. Upon creation of a thread it is suspended. If the task of a

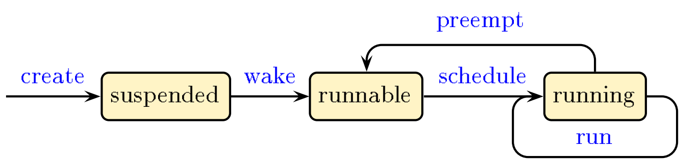


Figure 1: Different states of threads.

suspended thread becomes reducible, the thread becomes runnable. We say it is *woken*.

Amoz is sequential, based on a single worker, where multiple runnable threads are scheduled preemptively and fairly. Only one thread can run at a time, it is called *running*. Making a runnable thread running is called to *schedule* the thread. Figure 1 sketches the handling of threads.

3 An Example: Mapping Lists

A procedure mapping a list `Xs` to a list `Ys` by applying a procedure `P` to all elements of both lists can be written as follows:

```
proc {Map Xs P Ys}
  if Xr X in Xs=c(X Xr) then
     local Y Yr in Ys=c(Y Yr) {P X Y} {Map Xr P Yr} end
  else Xs=nil Ys=nil fi
end
```

Lists are represented as trees `c(`t_1 `c(`t_2 ... `c(`t_n `nil)))`. The procedure is referred to by a variable `Map`, as to be expected in a language with first-class procedures.

To illustrate the operational semantics of `Map`, assume that the procedure definition has been elaborated. Now we enter the expression

```
declare Xs P Ys in {Map Xs P Ys}
```

whose elaboration creates new variables for `Xs`, `P`, and `Ys` and reduces the procedure application `{Map Xs P Ys}` to a conditional task. The **`declare`** expression is a variant of the **`local`** expression whose scope extends to expressions the programmer enters later. The conditional task cannot be reduced since there is no information about the variable `Xs` in the store.

Now we enter the constraint (every occurrence of '`_`' creates a fresh variable)

```
Xs=c(_ c(_ _))
```

Since `Xs=c(_ c(_ _))` entails the constraint in the guard of the conditional, it is reduced with its **`then`**-part. This imposes the constraint `Ys=c(Y Yr)`, applies `P` to `X` and `Y`, and elaborates the recursive application `{Map Xr P Yr}`. A new conditional task is created which immediately reduces. Once more a conditional task is created which this time cannot be reduced. The store now entails `Xs=c(_ c(_ _))` and `Ys=c(_ c(_ _))`. Two elaborators for the application of `P` have been created, but cannot reduce, since no definition for `P` has been elaborated yet. Both have been moved to newly created threads.

By entering the constraint `Xs=c(s(o) c(s(s(o)) nil))` the conditional task is reduced to its **`else`**-constituent. Now the store entails `Xs=c(s(o) c(s(s(o)) nil))` and `Ys=c(_ c(_ nil))`. Then we enter a procedure definition for `P`.

```
proc {P X Y} if Z in X=s(Z) then Y=Z else Y=o fi end
```

Both threads where the tasks for the application of `P` reside on are run, each creating a conditional task. After their reduction, the store entails:

```
Xs=c(s(o) c(s(s(o)) nil)) Ys=c(o c(s(o) nil))
```

Suppose that the definition entered for `P` would be more involved, e.g., prime factorization of large integers. In this case, prime factorization of each list element would proceed in a round-robin fashion.

Threads are created implicitly. However, by using **`thread`** E **`end`** as abbreviation for **`local`** x **`in if in`** $x = a$ **`then`** E **`else`** $a = a$ **`fi`** $x = a$ **`end`**, we can explicitly state that reduction of E must advance fairly.

4 The Constraint Store

This section explains how rational trees are represented in the constraint store and how they are unified. More details on this can be found in [21].

The constraint store consists of various kinds of *nodes*: tuples, names (explained in Sect. 8), variables, and references. Rational trees are composed of these nodes. The constraint store is a dynamic memory area, thus nodes must be explicitly allocated. Nodes are built according to the following definition, where the code of the abstract machine is presented in a C++-like notation.

```
struct Node {enum {TUPLE, NAME, VAR, REF} tag;
             union {Node *ref;
                    struct {Label label; Node *args[];} tuple;};};
```

As presentation proceeds, the **`union`** part will be extended to host information used with names and variables. Representation of nodes in this paper are chosen with simplicity rather than efficiency in mind. The DFKI Oz asbtract machine employs tagged pointers instead of a tagged data objects.

Unification of two trees residing in the constraint store works as follows:

```
Bool unify(Node *xin, Node *yin) {
 Node *x = deref(xin); Node *y = deref(yin);
 if (x==y) return True;
 if (x→tag==VAR) { bind(x,y); return True; }
 if (y→tag==VAR) { bind(y,x); return True; }
 if (x→tag==NAME || y→tag==NAME || x→tuple.label≠y→tuple.label)
  return False;
 Node *xargs[] = x→tuple.args; Node *yargs[] = y→tuple.args;
 rebind(x,y);
 for (int i = 0; i<width(y→tuple.label); i++)
  if (unify(xargs[i],yargs[i]) == False) return False;
 return True;
}

Node *deref(Node *n) { return (n→tag==REF) ? deref(n→ref) :  n; }
void bind(Node *f, Node *t) { f→tag = REF; f→ref = t; }
void rebind(Node *f, Node *t) { f→tag = REF; f→ref = t; }
```

The function `deref` follows a chain of references until a non-reference node is reached. The functions `bind` and `rebind` make their first arguments into a reference pointing to its second argument. Note that `bind` and `rebind` are identical, but as the presentation proceeds they will be enhanced in different ways. After dereferencing both arguments, `x` and `y` point to variable, tuple, or name nodes. In case they point to the same node, unification is done. If one of them points to a variable, then `bind` binds the variable to the other node. If both point to tuples with the same label (which implies the same width, i.e., the same number of subtrees, as well), `x` is made pointing to `y` by `rebind`. Unification continues recursively for all subtrees of the tuples. Otherwise, `False` is returned. Note that two names can be unified only if they are identical.

In each recursive call the number of tuple and variable nodes in the constraint store is decremented by one. Additionally the invariant holds that chains of references are acyclic. This implies termination of unification. In AMOZ it is important that unification is *variable-centered*: it results in binding variables.

5 Introducing AMOZ

This section introduces AMOZ by presenting compilation and execution of declaration, composition, equation, and tuple construction.

```
#define DISPATCH { PC++; goto emulate; }
engine() {
 emulate:  switch (*PC) {
           case ALLOCATE(n):
            E = new Node*[n];
            DISPATCH;
           /* further instructions will be filled in here */
          }
 fail:  /* handling of failure in unification */
}
```

Figure 2: The emulator loop of AMOZ.

Elaboration is implemented by execution of abstract machine instructions. Elaborating an expression E corresponds to executing the corresponding instructions $\mathcal{C}[\![E]\!]$ as given by the compiler. In the following we consider only expressions, that are closed (i.e., without free variables) and renamed apart (i.e., each variable is declared only once).

To compile an expression E, the set $\mathcal{V}$ of variables declared in E is computed, where variables declared in procedure definitions are left aside (they are treated in Sect. 8). For each variable $x \in \mathcal{V}$ an index $\mathcal{A}[\![x]\!]$ is allocated, so that AMOZ can refer to a variable x by its index $\mathcal{A}[\![x]\!]$.

AMOZ needs several registers. The *program counter* `PC` points to the currently executed instruction. The *environment* `E` is an array mapping the index $\mathcal{A}[\![x]\!]$ of variable x to the variable's node in the store: `E[`$\mathcal{A}[\![x]\!]$`]`. The emulator loop shown in Fig. 2 contains the single instruction `ALLOCATE(`n`)`. Instructions for an expression E whose set of variables $\mathcal{V}$ has n elements are preceded by an instruction `ALLOCATE(`n`)` to allocate memory for the variables in E.

Execution of **local** x **in** E **end** creates a fresh variable node in the store and writes a reference to it to the environment `E[`$\mathcal{A}[\![x]\!]$`]`. On the left of the diagram below the instructions obtained by compilation are shown, whereas on the right the implementation of newly introduced instructions is shown.

$\mathcal{C}[\![$**local** x **in** E **end**$]\!] \equiv$

`CREATE_VAR(`$\mathcal{A}[\![x]\!]$`)`	`case CREATE_VAR(i):`
$\mathcal{C}[\![E]\!]$	`E[i] = new Node ⟨tag: VAR⟩;`
	`DISPATCH;`

Composition is compiled into concatenation of the respective instruction sequences: $\mathcal{C}[\![E\,F]\!] \equiv \mathcal{C}[\![E]\!]\mathcal{C}[\![F]\!]$.

An equality constraint is translated to an instruction calling the unification algorithm as presented in Sect. 4.

$\mathcal{C}[\![x = y]\!] \equiv$

```
    UNIFY(𝒜⟦x⟧,𝒜⟦y⟧)  | case UNIFY(i,j):
                      |  if (unify(E[i],E[j])==False) goto fail;
                      |  DISPATCH;
```

Tuple construction $x = f(\overline{y})$ proceeds in three steps. First, a node holding the tuple to be constructed is allocated. A reference to it is held in register `S`. Second, the arguments are constructed and entered to the tuple's node. The last step unifies the constructed tuple with x.

$\mathcal{C}[\![x = f(y_1 \ldots y_n)]\!] \equiv$

```
    CREATE_TUPLE(f/n)  | case CREATE_TUPLE(f/n):
    PUT_ARG(1,𝒜⟦y₁⟧)   |  S=new Node ⟨tag:TUPLE,
    ...                |              tuple:⟨label:f/n,
    PUT_ARG(n,𝒜⟦yₙ⟧)   |                     args:new Node∗[n]⟩⟩;
    UNIFY_S(𝒜⟦x⟧)      |  DISPATCH;
                       | case PUT_ARG(i,j):
                       |  S→tuple.args[i] = E[j];
                       |  DISPATCH;
                       | case UNIFY_S(i):
                       |  if (unify(S,E[i])==False) goto fail;
                       |  DISPATCH;
```

The compilation scheme presented above is simplified; the integration of optimization techniques known from the WAM [25, 1, 24] like read/write mode unification, and allocation of temporary variables to registers is straightforward and is not detailed.

6 Threads and Matching

This section introduces threads through a restricted form of conditional expression that implements pattern-matching. Conditionals considered herein are of the form **if** $\overline{y}$ **in** $x = f(\overline{y})$ **then** E **else** F **fi**, where the variables in $\overline{y}$ are pairwise distinct. They can be reduced, if and only if the variable x is bound. The case where x is not bound, introduces threads into AMOZ.

Threads in AMOZ are stacks of tasks. They have the type `Thread` and feature the common operations `push`, `pop`, and `isEmpty`. A *task* `TASK(l)` points to an abstract machine instruction located at label `l`. AMOZ is extended by three registers: `running` (of type `Thread`) for the currently running thread, `runnable` (of type `ThreadQueue`) for the queue of runnable threads, and `timeOver` for a flag that will be set to `TRUE` by an external source (e.g., the operating system) after a certain amount of time.

Adding a task to the currently running thread is performed by the `PUSH` instruction. The `RETURN` instruction tries to execute the topmost task from the currently running thread. If the currently running thread has no tasks left, another thread is

```
emulate: ...
    case PUSH(l):
     push(running, TASK(l)); PC++;
     if (timeOver) goto preempt; else goto emulate;
    case RETURN:
     goto run;

run:
  if (isEmpty(running)) goto schedule;
  TASK(l) = pop(running); PC = l; goto emulate;
schedule:
  if (isEmpty(runnable)) // terminate AMOZ
  running = dequeue(runnable); goto run;
preempt:
  push(running, TASK(PC)); timeOver=FALSE;
  enqueue(runnable, running); goto schedule;
```

Figure 3: Extending the emulator loop for threads.

scheduled. Preemption is checked in the `PUSH` instruction only, since it is the only instruction by which tasks can be added dynamically to a thread.

The emulator loop is extended as shown in Fig. 3. Note that the loop deals only with runnable threads, creation and waking of threads is explained below.

We will need in the following that variables are extended such that suspended threads can be attached to them:

```
struct Node {... {struct {ThreadQueue *susp} var; ...} ...};
```

The special form of conditional compiles as follows:

$$\mathcal{C}[\![\textbf{if } y_1 \cdots y_n \textbf{ in } x = f(y_1 \cdots y_n) \textbf{ then } E \textbf{ else } F \textbf{ fi}]\!] \equiv$$

```
     PUSH(L1)                | case DELAY(i):
     DELAY($\mathcal{A}[\![x]\!]$)              |  S=deref(E[i]);
     MATCH($f/n$,Le)           |  if (S→tag≠VAR) DISPATCH;
     GET_ARG(1,$\mathcal{A}[\![y_1]\!]$)       |  enqueue(S→var.susp,new Thread ⟨TASK(PC)⟩);
     ...                     |  goto run;
     GET_ARG(n,$\mathcal{A}[\![y_n]\!]$)       | case MATCH(f/n,le):
     $\mathcal{C}[\![E]\!]$                     |  if (S→tag==TUPLE && S→label==f/n) DISPATCH;
     RETURN                  |  PC=le;
 Le: $\mathcal{C}[\![F]\!]$                     |  goto emulate;
     RETURN                  | case GET_ARG(i,j):
 L1:                         |  E[j]=S→tuple.args[i];
                             |  DISPATCH;
```

The first instruction pushes a task on the running thread, thus fixing where execution proceeds after the conditional. The instruction DELAY checks whether the variable x is bound. In case x is bound, it is matched against the pattern and execution continues with the instructions for either E or F depending on the outcome of the match. Otherwise, the conditional must wait until x is bound. In this case a new thread consisting of the single task to reexecute the DELAY instruction is created. This thread is attached to the node of x. Execution continues by popping the next task from the current thread.

Binding a variable wakes all suspended threads attached to it by adding them to the queue of runnable threads. This is implemented by extending the procedure bind (cf. Sect. 4):

```
void bind(Node *f, Node *t) {
 runnable=concat(runnable,f→susp);
 f→susp=⟨⟩; f→tag=REF; f→ref=t;
}
```

7 Local Computation Spaces

This section introduces local computation spaces to AMOZ, and shows how they are used for implementing conditionals with deep guards.

Local computation spaces are represented in the machine as follows:

```
struct Space {Space *parent;
              NodePair *script[];
              enum {ALIVE, FAILED, ENTAILED} state;
              int counter;
              Instr* entailed, failed;};
```

The parent component points to the space directly above, linking spaces to the tree of spaces. In the *root* space, i.e., the topmost space, it is NULL. AMOZ is equipped with a register curSpace pointing to the current computation space, which is initialized with the root space. A task TASK(s,l) now also carries the space s to which it belongs.

Local constraints are maintained in the script, consisting of pairs of nodes (to be explained later).

The field entailed (failed) points to an instruction where execution proceeds in case the local space is entailed (failed). In our case of a conditional, these fields point to the first instruction of its **then** respectively **else** constituent.

Entailment of computation spaces. A computation space is entailed if no spaces exist below, it has no tasks left, and its local constraints are entailed by its parent's constraints. To check the first two conditions for a space, the field counter counts its tasks and the spaces below. The counter is maintained upon creation of new spaces, failure of spaces, merging of entailed spaces, and upon pushing and popping of tasks to the currently running thread.

A local constraint is entailed if it does not bind any global variables (this is a well known property of rational tree constraints [21], sometimes also referred to as quietness). Binding of variables needs to support entailment checking: the direction of binding becomes important, that is, global variables must not be bound to local variables [21]. This introduces the need to check for locality of variables. Therefore, a variable node in the store contains its home space, which is initialized upon variable creation:

```
struct Node {... struct {ThreadQueue *susp; Space *home;} var; ...};

case CREATE_VAR(i):
 E[i]=new Node ⟨tag:VAR var:⟨susp:⟨⟩ home:curSpace⟩⟩;
 DISPATCH;
```

Checking locality of a variable must take into account that a computation space is merged with its parent's space upon entailment. Testing whether a variable is local to a space is implemented by applying the function `isCurrent` to the variable's home field.

```
Bool isCurrent(Space *s) {
 return (s==curSpace || s→state==ENTAILED && isCurrent(s→parent));
}
```

In DFKI Oz, the garbage collector shortens `parent` chains, such that memory used by entailed spaces can be reclaimed.

As in the previous section, upon binding of variables suspended threads need to be woken. DFKI Oz incorporates an important optimization, that only threads below the current space are woken. Maintaining the script will be explained later.

```
void bind(Node *f, Node *t) {
 if (t→tag==VAR && isCurrent(t→home)) swap(f,t);
 if (!isCurrent(f→home)) add(curSpace→script,⟨f,t⟩);
 f→tag=REF; f→ref=t;
 wake(f);
 if (t→tag==VAR) wake(t);
}

void wake(Node *n) {
 runnable=concat(runnable,n→susp);
}
```

Bindings done by the procedure `rebind` must be done local to a space as well. This can be achieved by doing them only temporarily during unification and undoing them after finishing `unify`. For sake of brevity we omit the straightforward redefinition of `unify` and `rebind`.

Finally, the procedure `isEntailed` tests whether a space is entailed:

```
Bool isEntailed(Space *s) {
 return s→state==ALIVE && s→counter==0 && s→script==⟨⟩
}
```

Maintaining multiple computation spaces. In a space, all constraints from spaces above must be visible. In a sequential implementation this can be achieved by doing variable bindings in place and maintaining a script of globally visible changes, supporting fast access to both local and global bindings. The globally visible changes are bindings of global variables. They are written to the script in the procedure `bind`. Other schemes for multiple constraint stores are known, e.g., [15, 13, 14].

Suppose that the current space is S_1, and a task in a different space S_2 must be run. All constraints local to spaces between S_1 and the root must be removed, and all constraints between the root and S_2 must be made visible[2]. Removal of constraints is called *leaving*, whereas making constraints visible is called *entering*.

```
void leave(Space *s) {
 if (s==NULL) return;
 leaveSpace(s);
 leave(s→parent);
}
```

```
Bool enter(Space *s) {
 if (s==NULL) return True;
 if (!enter(s→parent)) return False;
 return enterSpace(s);
}
```

Leaving a single space removes all bindings contained in its script. Entering updates `curSpace`, and performs unification of all pairs in the script. Note, that it is not sufficient to perform binding, since the left hand side of a script entry may be bound already.

```
void leaveSpace(Space *s) {
 foreach ⟨x,t⟩ in s→script
  x→tag = VAR;
}
```

```
Bool enterSpace(Space *s) {
 curSpace=s;
 if (s→state==FAILED) return False;
 foreach ⟨x,t⟩ in s→script
  if (unify(x,t)==False)
   return False;
 s→script=⟨⟩; return True;
}
```

[2]DFKI Oz and the AGENTS implementation of AKL use the straightforward optimization not to go up to the root space, but to the closest common ancestor of S_1 and S_2.

Deep guards. Now we consider conditionals with deep guards. Their compilation is as follows:

$\mathcal{C}[\![$**if** $x_1 \cdots x_n$ **in** E **then** F **else** G **fi**$]\!] \equiv$

```
     PUSH(L1)
     CREATE_SPACE(Lt,Le)
     CREATE_VAR(A[[x1]])
     ...
     CREATE_VAR(A[[xn]])
     C[[E]]
     CHECK_ENTAILED
 Lt: C[[F]]
     RETURN
 Le: C[[G]]
     RETURN
 L1:
```

```
case CREATE_SPACE(lt,le):
 Space *s=new Space ⟨state:ALIVE,
                     entailed:lt, failed:le,
                     script:⟨⟩, counter:0,
                     parent:curSpace⟩;
 curSpace→counter++; curSpace = s;
 DISPATCH;
case CHECK_ENTAILED:
 if (isEntailed(curSpace)) {
  merge(); goto emulate;
 } else {
  foreach ⟨x,t⟩ in s→script
   enqueue(x→var.susp,newThread(PC));
  goto run;
 }
```

The instruction `CREATE_SPACE` links the newly created space to the tree, updates the current space's counter, and enters the created space. By `newThread(l)` a new thread with one task to execute the instruction at `l` in the current space is created. The instruction `CHECK_ENTAILED` checks if the space is entailed. In case the space is not yet entailed, new threads are added to the global variables bound in this space. These threads contain the task to reexecute the `CHECK_ENTAILED` instruction. Otherwise, the function `merge` merges the current space with its parent space.

```
void merge() {
 curSpace→state=ENTAILED;
 PC=curSpace→entailed;
 curSpace=curSpace→parent;
 curSpace→counter--;
}
```

Running a thread now needs to enter the computation space of the task. When popping a task `TASK(s,l)`, then the space `s` is entered and its counter is decremented. When this task has been executed (i.e., when reaching `run` again) entailment is checked, because it could have been the space's last task. AMOZ also needs to handle failure of a space, because failed spaces must be discarded from the tree of spaces. Spaces below a failed space are not marked as failed immediately, instead `enter` detects them and discards their tasks. The extended emulator loop is shown in Fig. 4.

The cost of pushing and popping tasks of the form `TASK(s,l)` can be reduced by having two kinds of tasks `TASK_S(s)` and `TASK_L(l)`. The latter will simply jump to the instruction at label `l`, where the (costly) former will enter space `s` and maintain

```
fail:
 if (curSpace→status==ALIVE) {
  curSpace→state = FAILED;
  PC = curSpace→failed;
  curSpace=curSpace→parent;
  curSpace→counter--;
  goto emulate;
 }
 goto run;
```

```
run:
 if (isEntailed(curSpace)) {
  merge(); goto emulate;
 }
 if (isEmpty(running)) goto schedule;
 TASK(s,l)=pop(running);
 s→counter--;
 leave(curSpace);
 if (enter(s)==False) goto fail;
 PC=l;
 goto emulate;
```

Figure 4: Extending the emulator loop for local computation spaces.

`counter` as explained above. Now tasks of kind `TASK_S(curSpace)` need to be pushed just before `curSpace` is left.

Pattern matching conditionals as shown in the previous section can be used as an optimization to conditionals with deep guards. This makes indexing techniques applicable as known for Prolog [24], Concurrent Logic Programming languages [9], and CC languages [2]. Further techniques for optimization of flat guards including composition of equations and arithmetic tests have been integrated into DFKI Oz.

8 First-class Procedures

This section introduces procedure definition and application to AMOZ. Nodes carrying the tag `NAME` are extended to support binding to procedures. Procedures resemble closures known from functional programming languages.

```
struct Node {... struct {Instr *lb; int arity; Node *free[];} proc;};
```

A procedure definition is compiled to instructions creating the procedure and instructions for the body of the procedure ($z_1, \cdots, z_k$ denote the *free* variables of **local** $y_1 \cdots y_n$ **in** E **end**):

$\mathcal{C}[\![$**proc** $\{x\ y_1 \cdots y_n\}\ E$ **end**$]\!] \equiv$

```
     PROCDEF(Lb,$n$,$k$) | case PROCDEF(l,n,k):
     UNIFY_S($\mathcal{A}[\![x]\!]$)    |  S=new Node ⟨tag:NAME, proc:⟨lb:l,arity:n,
     MOVE_C($\mathcal{A}[\![z_1]\!]$,1)  |                            free:new Node∗[k]⟩⟩;
     ...                 |  DISPATCH;
     MOVE_C($\mathcal{A}[\![z_k]\!]$,$k$) | case MOVE_C(i,j):
     JUMP(L1)            |  S→proc.free[j]=E[i];
 Lb: $\mathcal{B}[\![y_1 \cdots y_n\ ,\ E]\!]$      |  DISPATCH;
 L1:                     | case JUMP(l):
                         |  PC=l; goto emulate;
```

The `PROCDEF` instruction creates a new `NAME` node. The `free` field is filled by `MOVE_C` instructions. Free variables are addressed by a new register `F` within the body of a procedure similarly to how other variables are addressed by `E`. Thus we allow access to `F` in instructions like `UNIFY`.

$\mathcal{B}[\![y_1 \cdots y_n, E]\!]$ compiles the body: it allocates an environment of size $k = m + n$, where m is the number of declared variables within E (cf. Sect. 5), and n is the number of arguments. Parameters, which are passed in *argument registers* `A[1]` to `A[`n`]` as in the WAM, are first saved into the environment. Then the compiler creates code for the body.

$\mathcal{B}[\![y_1 \cdots y_n, E]\!] \equiv$

```
     ALLOCATE($k$)         | case MOVE_E(i,j):
     MOVE_E(1,$\mathcal{A}[\![y_1]\!]$)  |  E[j]=A[i];
     ...                 |  DISPATCH;
     MOVE_E($n$,$\mathcal{A}[\![y_n]\!]$)  |
     $\mathcal{C}[\![E]\!]$                |
     RETURN              |
```

An application $\{x\ y_1 \cdots y_n\}$ checks whether x is bound to a name, moves $y_1 \cdots y_n$ into `A[1]` to `A[`n`]`, pushes the instruction following the application on the currently running thread, sets register `F` to point to x's procedure, and jumps to the body of the procedure. Since an `ALLOCATE` instruction in the procedure's body will change `E` we have to modify the task data structure to also contain the environment `E` and free variable `F` registers.

```
                      |       ALLOCATE(2)
                      |       CREATE_VAR(E[0])   % Z → E[0]
                      |       CREATE_VAR(E[1])   % P → E[1]
                      |       CREATE_TUPLE(a/0)
local Z P in          |       UNIFY_S(E[0])      % Z=a
   Z=a                |       PROCDEF(Lp,1,1)    % Z is free
   proc {P X}         |       UNIFY_S(E[1])
      X=Z             |       MOVE_C(E[0],0)     % Z→F[0]
   end                |       JUMP(Le)
end                   |   Lp: ALLOCATE(1)
                      |       MOVE_E(1,E[0])     % X → E[0]
                      |       UNIFY(E[0],F[0])   % X=Z
                      |       RETURN
                      |   Le: RETURN
```

Figure 5: Example code for a procedure definition.

```
$\mathcal{C}[\![\{x\ y_1 \cdots y_n\}]\!] \equiv$
        PUSH(L1)                          | case PUSH(l):
        DELAY($\mathcal{A}[\![x]\!]$)            |  push(running,TASK(curSpace,l,E,F));
        MOVE_A($\mathcal{A}[\![y_1]\!]$,1)       |  DISPATCH;
        $\cdots$                               | case MOVE_A(i,j):
        MOVE_A($\mathcal{A}[\![y_n]\!]$,$n$)     |  A[j]=E[i];
        APPLY($\mathcal{A}[\![x]\!]$,$n$)        |  DISPATCH;
    L1:                                   | case APPLY(i,n):
                                          |  S=deref(E[i]);
                                          |  if (S→tag≠NAME || S→proc.arity≠n) goto fail;
                                          |  F=S→proc.free;
                                          |  PC=S→proc.lb; goto emulate;
```

In Fig. 5 we show the compilation of a procedure `P`, which unifies its argument with the variable `Z`.

The scheme presented above imposes an overhead to the handling of procedures compared to the first order case as exemplified by Prolog. Therefore the DFKI Oz compiler performs some important optimizations to eliminate these extra costs, as we will describe now.

The `ALLOCATE` instruction takes memory from a heap (and not from a stack as Prolog does) and there is no `DEALLOCATE` instruction at the end of a procedure. This is due to concurrency: when the end of a procedure is reached, there may be suspended threads referring to the environment. DFKI Oz does the following: the compiler inserts a `DEALLOCATE` instruction, and environments are allocated from a free-list. Every environment has a flag which is set if there exist suspended threads. The `DEALLOCATE` instruction checks this flag, and frees the environment only if the flag is not set.

The execution of the instructions for a procedure definition is quite costly due to procedure creation. The instructions are only executed once per procedure, whereas the procedure itself can be applied many times. To reduce the cost of a procedure application the instructions `PUSH`, `DELAY` and `APPLY` can be collapsed into one instruction. Additionally, the compiler tries to determine by static analysis whether x in $\{x\,y_1 \cdots y_n\}$ is bound to a procedure with arity n. Experience shows that it succeeds in most cases, especially in all cases where procedures are used as in Prolog: the compiler replaces the `PUSH`, `DELAY`, `APPLY` sequence by a special instruction `FASTAPPLY` taking as argument the instruction address for the procedure definition of x. This eliminates the extra costs of the general scheme.

Acknowledgements

We would like to thank Tobias Müller, Konstantin Popov, and Gert Smolka for helping us with the design and/or implementation of the abstract machine for Oz. We are grateful to the Programming Systems Group at SICS for sharing their experiences in implementing SICStus Prolog and AKL with us. Martin Henz, Martin Müller, Tobias Müller, Joachim Niehren, Gert Smolka, Ralf Treinen, Peter Van Roy, and the anonymous referees provided helpful comments on this paper. The research reported in this paper has been supported by the Bundesminister für Bildung, Wissenschaft, Forschung und Technologie (FTZ-ITW-9105), the Esprit Project ACCLAIM (PE 7195), and the Esprit Working Group CCL (EP 6028).

References

[1] Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* Logic Programming Series. The MIT Press, Cambridge, MA, 1991.

[2] Per Brand. A decision graph algorithm for CCP languages. In Leon Sterling, editor, *Proceedings of the 1995 International Conference on Logic Programming*, pages 433–448, Kanagawa, Japan, June 1995. The MIT Press.

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

[4] Alain Colmerauer. Prolog and infinite trees. In K.L. Clark and S.-A. Tärnlund, editors, *Logic Programming*, pages 153–172. Academic Press, 1982.

[5] Alain Colmerauer. Equations and inequations on finite and infinite trees. In *Proceedings of the 2nd International Conference on Fifth Generation Computer Systems*, pages 85–99, 1984.

[6] Martin Henz, Gert Smolka, and Jörg Würtz. Object-oriented concurrent constraint programming in Oz. In Vijay Saraswat and Pascal Van Hentenryck, editors, *Principles and Practice of Constraint Programming*, chapter 2, pages 27–48. The MIT Press, Cambridge, MA, 1995.

[7] Paul Hudak, Philip Wadler, et al. Report on the programming language Haskell. Technical Report YALEU/DCS/RR/777, Yale University, 1990.

[8] Sverker Janson. *AKL - A Multiparadigm Programming Language.* Dissertation, SICS Swedish Institute of Computer Science, Uppsala University 1994, SICS Box 1263, S-164 28 Kista, Sweden, 1994.

[9] Shmuel Kliger and Ehud Shapiro. From decision trees to decision graphs. In Saumya Debray and Manuel Hermenegildo, editors, *North American Conference on Logic Programming*, pages 97–116, Austin, TX, 1990. The MIT Press.

[10] Michael J. Maher. Logic semantics for a class of committed-choice programs. In Jean-Louis Lassez, editor, *Logic Programming, Proceedings of the Fourth International Conference*, pages 858–876, Melbourne, 1987. The MIT Press.

[11] Michael Mehl, Tobias Müller, Konstantin Popov, and Ralf Scheidhauer. DFKI Oz user's manual. DFKI Oz documentation series, German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, 1994.

[12] Robin Milner, Mads Tofte, and Robert Harper. *Definition of Standard ML.* The MIT Press, Cambridge, MA, 1990.

[13] Johan Montelius and Khayri A. M. Ali. An And/Or-parallel implementation of AKL. *New Generation Computing*, 13–14, August 1995.

[14] Andreas Podelski and Gert Smolka. Situated simplification. In *Proceedings of the First International Conference on Principles and Practice of Constraint Programming*, LNCS, Marseille, France, September 1995. Springer-Verlag. To appear.

[15] Andreas Podelski and Peter Van Roy. The beauty and beast algorithm: quasi-linear incremental tests of entailment and disentailment over trees. In Maurice Bruynooghe, editor, *Logic Programming: Proceedings of the 1994 International Symposium*, pages 359–374, Ithaca, NY, November 1994. The MIT Press.

[16] Vijay A. Saraswat and Martin Rinard. Concurrent constraint programming. In *Proceedings of the 7th Annual ACM Symposium on Principles of Programming Languages*, pages 232–245, San Francisco, CA, January 1990. ACM Press.

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

[18] Christian Schulte, Gert Smolka, and Jörg Würtz. Encapsulated search and constraint programming in Oz. In Alan H. Borning, editor, *Second Workshop on Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science, vol. 874, pages 134–150, Orcas Island, WA, May 1994. Springer-Verlag.

[19] 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, September 1994. Springer-Verlag.

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

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

[22] Gert Smolka and Ralf Treinen (ed.). DFKI Oz documentation series. Deutsches Forschungszentrum für Künstliche Intelligenz, Stuhlsatzenhausweg 3, D–66123 Saarbrücken, Germany, 1994.

[23] Pascal Van Hentenryck, Vijay Saraswat, and Yves Deville. Design, implementation and evaluation of the constraint language cc(FD). In Andreas Podelski, editor, *Constraint Programming: Basics and Trends*, Lecture Notes in Computer Science, vol. 910, pages 293–316. Springer-Verlag, 1995.

[24] Peter Van Roy. 1983-1993: The wonder years of sequential Prolog implementation. *Journal of Logic Programming*, 19/20:385–441, May/July 1994.

[25] David H. D. Warren. An abstract Prolog instruction set. Technical Note 309, SRI International, Artificial Intelligence Center, Menlo Park, CA, October 1983.

Remark

The DFKI Oz system and papers of authors from the Programming Systems Lab at DFKI are available through WWW at `http://ps-www.dfki.uni-sb.de/` or through anonymous ftp from `ps-ftp.dfki.uni-sb.de`.