Attributed Variables in Oz

Michael Mehl, **Martin Müller** and **Gert Smolka**
Programming Systems Lab
German Research Center for Artificial Intelligence (DFKI)
Stuhlsatzenhausweg 3, D–66123 Saarbrücken, Germany
{mehl,mmueller,smolka}@dfki.uni-sb.de

Abstract

Attributed variables have proved to be a flexible mechanism to extend the functionality of unification based languages towards new constraint systems or meta programming. We present our approach of integrating attributed variables into Oz and illustrate it with a number of examples. The main surprise is the fact that attributed variables can be given an elegant formal operational semantics at the high level of the Oz Programming Model (Opm). This is in contrast to previous work, which conceived attributed variables as an implementation driven low-level concept.

Keywords Attributed Variables, Operational Semantics, Concurrent Constraint Programming, Constraint Systems, Oz Programming Model, Oz

Correspondence

Martin Müller
Programming Systems Lab
German Research Center for Artificial Intelligence (DFKI)
Stuhlsatzenhausweg 3, D–66123 Saarbrücken, Germany,
mmueller@dfki.uni-sb.de,
Tel (Fax): +49 681 302-5329 (5341)

1 Introduction

Constraint logic programming (CLP) and concurrent constraint programming (CCP) are based on a parametrized computational model. This model can be instantiated by any constraint system for which one has at least an incremental algorithm for deciding satisfiability [15].

Of course, every concrete CLP or CCP system can support only specific constraint systems immediately. Almost all constraint languages contain a solver for equational constraints over some tree domain (finite, rational or feature trees), and the recent ones also offer finite domain constraint solving.

Whenever a new constraint solver needs to be incorporated, one might provide direct support from the abstract machine. This is likely to be a fast approach, but definitely not the most flexible one. To add a new solver without direct machine support, one has two immediate options:

1. Find a high-level representation of the constraints and use a meta-interpreter to implement the constraint solver.

2. Implement a new constraint system on top of an existing one. This often requires to restrict the expressivity of the new one.[1]

Both options are not fully satisfactory. Meta-interpretation is costly, and in most cases a sufficiently powerful partial evaluator is not at hand which could eliminate the extra complexity. Further, meta-interpretation cannot easily build upon a given constraint solver and exploit its efficient implementation, even if the new solver is a proper extension of a given one. The restriction approach is not an option if the new constraints are needed in full expressiveness, and if it is, the restriction may be contrived and difficult for a programmer to understand.

To tackle such problems, many logic programming systems provide a form of *attributed variables* [13] which support the amalgamation of object-level and meta-level programming.[2] Based on attributed variables, logic programming systems have been enhanced by constraint solvers over rational numbers [11, 12], feature trees (records) [26], and sets[3]. Attributed variables are also used for the implementation of distributed CLP systems [10].

Attributed variables are variables with an associated data structure, the *attribute*, which may represent user-defined constraints on the variable. On unification of two attributed variables, a *unification handler* (user-defined as well) is called to replace both attributes by another one which represents their conjunction. For example, consider a set constraint system encoding sets as list-valued attributes. Two variables `X` and `Y` with attributes `[1 2 3]` and `[2 1 3]` should be unifiable since both lists describe the same set. Thus, an appropriate unification handler would check whether two lists are permutations of each other, and pick one of them freely as the new attribute.

Attributed variables have proven to be a powerful mechanism for the extension of CLP systems, but they have not yet been given a rigorous semantics. In this paper we describe how attributed variables have been added into the concurrent constraint

[1] For instance, in the ProFIT system [4], Prolog has been extended by compiling more expressive (feature and sort) constraints into finite tree unification.

[2] A phrase from [19], which contains a broad discussion on extensibility of PROLOG.

[3] Unpublished work by S. Manandhar

language Oz [31]. The main contribution of this paper is a formal operational semantics at the high level of the Oz Programming Model (OPM). This turns attributed variables from an implementation driven low-level concept into a first-class feature of the language.

The development of OPM took from research in concurrent constraint programming [17, 28] and the pi calculus [20], and builds on design and implementation of AKL [16]. OPM can be seen as an extension of the concurrent constraint model by higher-order procedures and cells (state). Computation in OPM proceeds by tasks reducing over a shared store which contains compartments for constraints, procedures, and cells. During reduction, tasks may add (tell) information to the store (as long as it remains consistent) and wait until certain constraints are logically entailed by the store (ask). OPM allows the addition of a search combinator [29] which makes it general enough to encompass the computation models of other logic languages such as PROLOG II [3].

OPM accomodates attributed variables in a special compartment of the store, called the *attribute store*. Although the attribute store is intended as a (user-defined) extension of the constraint store, entailment is still decided relative to the constraint store only. Instead, there is an interaction between attribute and constraint store which enables attributes to affect the constraint store by means of the user-defined handlers. More specifically, the constraint store activates the attribute store as soon as certain equations become entailed, which in turn emits tasks in order to execute user-provided procedures. This mechanism has been implemented within DFKI OZ as a minimal extension of its abstract machine [18].

Related Work. Concepts similar to attributed variables have appeared in the literature under different names: The earliest reference is to Carlsson's *suspension variables* [2] with which coroutining facilities were integrated into SICSTUS PROLOG. Le Huitouze [13] coined the name *attributed variables* for an abstract data type developed for the implementation of PROLOG II in MALI [27]. The hook into unification provided by attributed variables could be used to optimize memory management. Neumerkel's *metastructures* [23] are ordinary (PROLOG) terms which are recognizable as members of a special sort (built from special constructors). Roughly, metastructures do not attach attributes to a variable, but embed the variable into its attributes. When unification encounters metastructures, it passes control to user-defined predicates. Neumerkel specified metastructures via a PROLOG meta interpreter which made unification fully explicit. The ECL^{i}PS^e system [1] supports the attribution of variables by *metaterms* (which are just attributed variables rather than metastructures). The constraint handling library (CHR) provided in ECL^{i}PS^e makes use of metaterms [5]. Holzbaur [11, 12] implemented a complete CLP language based on attributed variables. His system has been integrated into SICSTUS PROLOG [6], and is being added to ECL^{i}PS^e.

Our driving motivation to investigate attributed variables was the implementation of a set-constraint system as an extension of rational tree unification [21]. We have been inspired by work of T. Müller who implemented an experimental C interface providing attributed variables for DFKI OZ [22]. While this paper focusses on the implementation of constraint solvers, ongoing research investigates the use of attributed variables for the implementation of a distributed variable unification protocol.

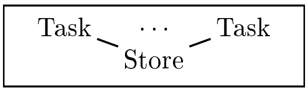
E	$::=$	φ	constraint
		$E_1 \mid E_2$	composition
		$\texttt{if}\ \varphi\ \texttt{then}\ E_1\ \texttt{else}\ E_2$	conditional
		$\texttt{local}\ x\ \texttt{in}\ E$	declaration
		$\texttt{proc}\ \texttt{\{}x\ z\texttt{\}}\ E$	definition
		$\texttt{\{}x\ y\texttt{\}}$	application
φ, ψ	$::=$	$x = y \mid x = f(\overline{y}) \mid \varphi \wedge \psi$	constraints

Figure 1: A Simple cc Language

Plan of the Paper. In Section 2 we briefly recall the Oz Programming Model Opm. Section 3 illustrates by example the implementation of a new constraint system using attributed variables. Section 4 specifies attributed variables for Opm rigorously, and justifies design decisions by considerations of confluence and atomicity. Figure 5 illustrates the resulting programming practice in Oz, by giving encodings of freeze and lazy evaluation, and Section 6 concludes.

2 The Oz Programming Model

The Oz Programming Model (Opm) [31] is a concurrent programming model subsuming higher-order functional [24] and object-oriented programming [9] as facets of a general model.



Computation in OPM takes place in a *computation space* hosting a number of *tasks* connected to a shared *store*. Computation advances by *reduction of tasks*. The reduction of a task can manipulate the store and create new tasks. When a task is reduced it disappears. Reduction of tasks is an atomic operation, and tasks are reduced one by one. Typically, many tasks are reducible in a given computation space. To obtain fairness, reactivity, and efficiency, a *reduction strategy* selects the next task to be reduced.

Tasks can *synchronize* on the store in that they become reducible only once the store satisfies certain conditions. *Task synchronization* in OPM *is monotonic*: a reducible task stays reducible if other tasks are reduced earlier.

Monotonic task synchronization is achieved as follows: Assume a set of *constraints*, *i.e.*, logic formulas φ, ψ closed under conjunction, for which a logic entailment relation ($\varphi \models \psi$) is defined. The store holds a constraint in a special compartment called the *constraint store*. The only way to update the constraint store is to *tell* it a constraint φ, which strengthens the constraint store from ψ to the conjunction $\psi \wedge \varphi$, provided that $\psi \wedge \varphi$ is logically consistent. This provision makes the satisfiability test an integral part of the tell operation.[4] Tasks may synchronize another task on a constraint, called the task's *guard*. A synchronized task becomes reducible if its guard is entailed by the constraint store.

[4] See also *incremental tell* [31].

We consider the sublanguage of OPM defined by the grammar in Figure 1. We pick equational constraints over rational (infinite) trees; this is for concreteness and because most CLP or CCP systems support constraints over some tree domain. We say that the constraint store φ *determines* a variable x, if φ knows the top-level constructor of x: $\varphi \models \exists \overline{y}(x = f(\overline{y}))$ for some f.

Tasks E which are constraints, compositions or declarations, or definitions, are unsynchronized. Conditional tasks synchronize on the constraint store and become reducible only once their guard is entailed or disentailed by the constraint store.

Reduction of a constraint φ *tells* φ to the constraint store. Reduction of a composition $E_1 \wedge E_2$ creates two tasks E_1 and E_2. Reduction of a conditional `if` φ `then` E_1 `else` E_2 creates the task E_1 if φ is entailed and the task E_2 if φ is disentailed. Reduction of a declaration `local` x `in` E chooses a fresh variable y and creates the task $E[y/x]$ (E with y replaced for x). Reduction of a definition `proc {`x z`}` E chooses a fresh *name* ξ (which can be seen as a globally unique constant), tells $x = \xi$ to the constraint store and writes the procedure $\xi{:}z/E$ to the procedure store. An application `{`x y`}` waits until the procedure store contains a procedure $\xi{:}z/E$ such that the constraint store entails $x = \xi$. If this is the case, the task `{`x y`}` reduces to $E[y/z]$ which is obtained by replacing y for z in E.

3 An Example

As a didactic example, we discuss the implementation of a simple solver over finite domains of constants C. We consider constraints of the form

$$\begin{array}{lll} \varphi, \psi & ::= & x = a \mid x \in S \mid x = y \mid \bot \mid \varphi \wedge \psi \mid \exists x \varphi \\ a & ::= & \text{elements of } C \\ S & ::= & \text{finite subsets of } C \end{array}$$

Variables are interpreted by elements of C. A valuation v of the variables in a constraint φ satisfies φ ($v \models \varphi$) if the primitive constraints in φ are different from *inconsistency* $\bot$ and are simultaneously satisfied by v:

$$v \models x = y \quad \text{if } x^v{=}y^v \qquad v \models x = a \quad \text{if } x^v{=}a \qquad v \models x \in S \quad \text{if } x^v{\in}S$$

Satisfiability of a constraint can be checked by the simplification rules in Figure 2. This algorithm transforms a constraint into a normal form, which contains for every variable x at most one constraint of the form $x = y$, $x = a$, or $x \in S$; that is, the constraint store can be implemented in a *variable centered* manner. Whenever a constraint is added (told), the rules restore this property.

While the constraints $x = y$ and $x = a$ are directly supported in most CLP or CCP systems, the constraint $x \in S$ must be represented explicitly (assuming no direct finite domain support). This can be done by attaching S as an *attribute* to x, making x an *attributed variable*. We assume a binary procedure `NewDom` to perform this operation: `{NewDom S X}`

Rules (V), $(C1)$, and $(C2)$ are implemented as part of the tree unification algorithm. For $(U1)$, $(U2)$, $(D1)$ and $(D2)$, user-defined code has to be provided.

Rule (U) applies when there are two constraints of the form $x \in S$. Given variable-centeredness, this can happen only by unification of two variables. Both constraints are discarded and replaced by a new one. The replacement is defined by the following procedure, as the so-called *unification handler*.

$$(D1)\ \frac{x = a \wedge x \in S}{x = a}\ a \in S \qquad (D2)\ \frac{x = a \wedge x \in S}{\perp}\ a \notin S$$

$$(U1)\ \frac{x \in S_1 \wedge x \in S_2}{x \in S_1 \cap S_2}\ S_1 \cap S_2 \neq \emptyset \qquad (U2)\ \frac{x \in S_1 \wedge x \in S_2}{\perp}\ S_1 \cap S_2 = \emptyset$$

$$(V)\ \frac{x = y \wedge \varphi}{x = y \wedge \varphi[y/x]}\ x \in \mathcal{V}(\varphi)$$

$$(C1)\ \frac{x = a \wedge x = a}{x = a} \qquad (C2)\ \frac{x = a \wedge x = b}{\perp}\ a \neq b$$

Figure 2: Deciding Satisfiability of Finite Domain Constraints

```
proc {Unify S1 S2 S3}
   {Intersect S1 S2 S3}
   if S3=nil then false else true end
end
```

When there are two constraints of different form $x = a$ and $x \in S$, then rules $(D1)$ and $(D2)$ check their consistency. In both cases $x \in S$ is deleted, and $x = a$ survives unless $a \notin S$. Hence, deletion of $x \in S$ can be specified by the following procedure, called *deletion handler*:

```
proc {Delete X S}
   if {Member X S} then true else false end
end
```

The service provided by attributed variables can be summarized as follows: enable the attachment of attributes to variables, and call the unification or the deletion handler when an attributed variable becomes determined or unified with another attributed variable.

4 Attributed Variables for OPM

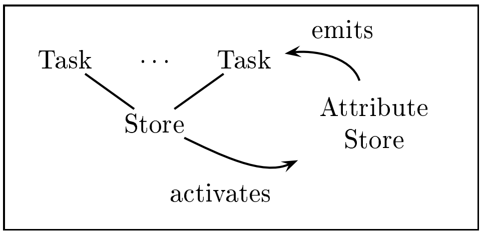
4.1 The Attribute Store

An *attributed variable* is an ordinary variable which has an *attribute* associated with it. Attributes are triples of variables u,d,v which are held in a new compartment of an Opm store, the *attribute store*. The attribute store contains *attributions* which are quadruples mapping variables to attributes:

$$x \rightsquigarrow u,d,v$$

The attribute components are called *unification handler* u, *deletion handler* d, and the *attribute value* v associated with x. A variable with an associated attribute is called an *attributed variable*.

The attribute store may contain multiple attributions for the same variable, and also attributions for a variable determined by the constraint store. Tasks do not interact directly with the attribute store. Instead, the attribute store emits tasks when activated by the constraint store. The emission of new tasks due to interaction of constraint and attribute store has priority over reduction of tasks, *i.e.*, is done eagerly. This makes the emission of new tasks an integral part of the tell operation just like the test for satisfiability.



4.2 Reduction of Attributions

There are two possible reductions involving attributed variables. The *unification* case applies when the attribute store contains for the same non-determined variable two attributions with the same unification handler. The other one deals with determination of an attributed variable.[5]

Unification. Assume the attribute store to contain the attributions

$$x \rightsquigarrow u,d,v \qquad \text{and} \qquad x' \rightsquigarrow u',d',v'.$$

If the constraint store entails $x = x'$ and $u = u'$ but does not determine x and x' (i.e., does not entail $\mathsf{det}(x)$ or $\mathsf{det}(x')$), then the following happens atomically: One of the attributions is discarded, the value of the other attribute is replaced with a fresh variable w yielding the attribution $x \rightsquigarrow u,d,w$, and the new task $\{u\ v\ v'\ w\}$ is created, which computes w from v and v'.

Deletion. If the attribute store contains an attribution

$$x \rightsquigarrow u,d,v$$

for a variable x which is determined by the constraint store, then the attribution is discarded, and the new task $\{d\ x\ v\}$ is created.

Note that attributed variables cannot be transformed into free (i.e., non-attributed) variables unless triggered by the constraint store via the deletion rule. That is, we do not provide an operation for explicit "detachment" of attributes, as used, *e.g.*, in [10].

4.3 Multiple Attributes and Confluence

A variable is allowed to have multiple attributes, provided they have different unification handlers.[6] This makes the unification handler a "type tag" for an attribution. The unification rule states that attributions of different type attached to the same

[5] Carlsson's suspension variable mechanism [2] is more general since it also calls a handler if an attributed variable is unified with a free variable.

[6] This design is similar to the one in ECL^{i}PS^e [1] or SICSTUS PROLOG [6], and in contrast to the approach taken in [22] where each variable can have just one attribute.

variable do not affect each other. This construction allows one to have, at the same time, multiple constraint solvers which do not interfere, even if they do not "know about each other".

This is an essential precondition for the design of confluent unification and deletion handlers. If two constraints from systems with overlapping domain affect the same variable, then eventually their consistency must be checked. Our design decision delays this compatibility check to the deletion case.

For illustration why this may affect confluence, assume that we implement a finite domain and a boolean solver such that the boolean domain $\{0,1\}$ is a valid finite domain. Given the attribute store

$$x \rightsquigarrow U_b, D_b, \{0,1\} \ \wedge \ y \rightsquigarrow U_f, D_f, \{0,\ldots,5\},$$

we tell the constraints $x{=}1$, $y{=}1$, and $x{=}y$ to the constraint store (in some order). Both deletion handlers are defined as in our running FD example.

Then essentially two reductions are possible: Either $x = 1$ and $y = 1$ are told before $x = y$, or vice versa.

If $x = 1$ and $y = 1$ are told first, both attributions are discarded and deletion handlers for booleans and finite domains are applied.

$$x = 1 \ \wedge \ \{D_b\, x\, \{0,1\}\} \ \wedge \ y = 1 \ \wedge \ \{D_b\, y\, \{0,\ldots,5\}\}$$

Unification of x and y is then uncritical.

If $x = y$ is told first, no unification handler is called since both variables carry different attributes.

$$x = y \ \wedge \ x \rightsquigarrow U_b, D_b, \{0,1\} \ \wedge \ y \rightsquigarrow U_f, D_f, \{0,\ldots,5\},$$

When x and y are bound to 1 (by whichever of $x = 1$ or $y = 1$), *both* deletion handlers are called.

$$x = y \ \wedge \ x = 1 \ \wedge \ \{D_b\, x\, \{0,1\}\} \ \wedge \ \{D_b\, y\, \{0,\ldots,5\}\}$$

If only one attribute were allowed per variable, the second branch would fail while the first would not, a blatant violation of confluence.

4.4 The Data Type

To incorporate attributed variables into Oz, we add just one kernel procedure `NewAttr` to Kernel Oz, which creates the corresponding attribution.

`{NewAttr U D V X}` must wait until `U` and `D` are bound to ternary and binary procedures, respectively. Then it reduces to $X \rightsquigarrow U, D, V$,

As an Oz procedure, `NewAttr` is specified as follows:

```
proc {NewAttr U D V X}
   case {IsProcedure U} andthen {Procedure.arity U 3}
      andthen {IsProcedure D} andthen {Procedure.arity U 2}
   then X⇝U,D,V else false end
end
```

```
local
   ... %% definitions of Unify and Delete on page 7
   proc {MyUnify V1 V2 V3}
      case V1 # V2 of av(S1) # av(S2) then
         local S3 in V3=av(S3) {Unify S1 S2 S3} end
      [] get(S1)# av(S2)  then V=av(S1)  S1=S2
      [] av(S1) # get(S2) then V=av(S1)  S1=S2
      [] get(S1)# get(S2) then V=get(S1) S1=S2
      end
   end
   proc {MyDelete X V}
      case V of get(S) then S = X
      [] av(W) then {Delete X W}
      end
   end
in
   proc {NewDom S X}
      {NewAttr MyUnify MyDelete av(S) X}
   end
   proc {GetDom X S}
      {NewAttr MyUnify MyDelete get(S) X}
   end
end
```

Figure 3: Encoding Attribute Access

This procedure is used to define the procedure `NewDom` based on the definitions of `Unify` and `Delete` from our running finite domain example.

```
proc {NewDom S X}
   {NewAttr Unify Delete S X}
end
```

Note that we need not specify kernel procedures for reading and writing access to an attribute, since we can encode them in presence of simple data structures.

The encoding relies on the following discipline. The attribute values are wrapped into a tuple with label `av`, and a wrapper `get` indicates a pseudo attribute value used just for the interaction via the unification rule. Figure 3 extends our finite domain system by reading access to attribute values.

4.5 Atomicity

The unification rule operates on attributions by discarding one and replacing the other. Note that this operation on attributions is specified to be atomic. Since attributes are *resources*, atomicity is essential in concurrent systems to avoid race conditions. This holds for Oz as well as for systems with corouting and preemption like Prolog II and Sicstus Prolog.

For illustration, consider the following generic unification handler (in a form as it would appear in Sicstus Prolog).

```
handle_unify(X, Y) :- get_atts(X, av(VX)), get_atts(Y, av(VY)),
                      do_sth(VX, VY, VZ), put_atts(Y, av(VZ)).
```

To handle a unification of variables `X` and `Y`, both attribute values are read by `get_atts`, "unified" by an operation `do_sth` and written back by `put_atts`. Assume that the variables `X`, `Y`, and `Z` are attributed with structures containing the unbound variable `U`, and consider the following goal.

```
..., freeze(U, X=Z), Y=Z, ...
```

Unification of `X` and `Z` is delayed until `U` becomes determined. On unification `Y=Z`, `handle_unify(Y Z)` is called. Since `U` is part of the values `VY` and `VZ`, it may become bound during the consecutive execution of `do_sth(VY, VZ, Vnew)`, such that `X=Z` is woken. If this is done eagerly, then the corresponding call to `handle_unify(Y Z)` overlaps dangerously with `handle_unify(X Z)`.

```
... get_attrs(Z, av(VZ)),      % subgoal of handle_unify(Y, Z)
  ... get_attrs(Z, av(VZ2)),   % subgoal of handle_unify(X, Z)
  ... put_attrs(Z, av(Vnew2)),
... put_attrs(Z, av(Vnew1)),
```

4.6 Implementation

The implementation of attributed variables within DFKI OZ is separated into an emulator and an OZ part. The emulator provides only minimal support: It knows of two OZ procedurs only (handling the unification and the deletion case) and provides the hooks to call them from the unification routine. The full semantics as specified above is implemented by these OZ procedures using record constraints [32]. Additionally, two built-ins are provided to create an attribution or to return the attribute of an attributed variable.

5 Further Examples: Freeze and Lazy Evaluation

In this Section we encode delayed evaluation with attributed variables. We first encode freeze, the coroutining primitive from PROLOG II [3] which has become a standard in logic programming. Then we adapt it to delayed evaluation known from lazy functional programming (*e.g.*, HASKELL [7]) This latter encoding has been motivated by the development of the functional language OZFUN [14], which offers a coarse-grained mixture of eager and lazy evaluation. OZFUN has been compiled into OZ in [14], based on encodings of lazy functional programming from [30, 25, 24].

5.1 Freeze

To freeze a goal on a variable means to suspend its evaluation until this variable becomes determined. This is, a variable gets associated with a list of closures, *i.e.*, goals. To delay the evaluation of an expression `E` we have two options in OZ, corresponding to the two expressions with irreducible subexpressions. These are procedures **`proc {P} E end`** and conditionals **`if Trigger=ok then E fi`**.

Assume `Call` to be defined by **`proc {Call P} {P} end`**. Figure 4 encodes *delay by abstraction* (the other case is analogous). An application

```
{NewFreeze proc {$} E end X}
```

```
local
   proc {FreezeUnify Xs Ys Zs}
      {Append Xs Ys Zs}
   end
   proc {FreezeDelete X F}
      {ForAll F Call}
   end
in
   proc {NewFreeze P X}
      {NewAttr FreezeUnify FreezeDel [P] X}
   end
end
```

Figure 4: Encoding Freeze

```
local
   proc {LazyUnify P1 P2 P3}
      {P1 X} {P2 X} P3=Succeed
   end
   proc {LazyDelete X P}
      {P X}
   end
in
   proc {NewLazy P X}
      {NewAttr LazyUnify LazyDelete P X}
   end
   proc {Force X}
      {NewLazy Succeed X}
   end
end
```

Figure 5: Encoding Delayed Execution by Abstraction

suspends the evaluation of `E` until `X` becomes determined, by attributing `X` with the singleton list `[proc {$} E end]`.

On unification of two attributed variables, the lists of frozen goals are simply appended. Determination of an attributed variable triggers a call to `FreezeDelete`, which wakes all frozen goals by passing them to `Call` via the higher-order procedure `ForAll`.

5.2 Lazy Evaluation

In lazy functional programming, an expression is evaluated only if it is needed. This is the case, *e.g.*, if the expression is about to be applied as a function. A functional program can be encoded into the relational setting of Oz by mapping each of its subexpressions `M` to an expression `X=M` which binds a fresh variable `X` to what `M` eventually evaluates to (see [30, 25, 24] for a formal treatment). To delay the evaluation of `X=M`, we attribute `X` with a unary procedure of the form `proc {P Y} Y=M end`. To request the evaluation of `X=M`, we simply apply it: `{P X}`.

The code in Figure 5 allows the delay of `X=M` by executing the following.

```
local proc {P Y} Y=M end in {NewLazy P X} end    %% delay
```

```
fun {Delay E M}
   {NewLazy proc {$ Y} Y={Interpret E M} end}
end
proc {Force X Y}
   X={NewLazy Succeed} X = Y
end
fun {Interpret E M}
   case M of lam(X N) then
      fun {$ XV} {Interpret (X#XV)|E N} end
   [] app(N1 N2) then
      {{Force {Interpret E N1}} {Delay E N2}}
   elsecase {IsInt M} then
      M
   else {Lookup M E} end
end
```

Figure 6: An Interpreter for the Lazy Lambda Calculus

Now assume both `X` and `Y` to be such "lazy variables". When `X` becomes determined, its associated assignment must be executed. When `X` and `Y` are unified, *both* associated assignments must be executed. In lazy functional languages this roughly corresponds to the fact that evaluating a conditional `if X = Y then ... fi` implies to evaluate both `X` and `Y` completely.

Figure 5 defines the corresponding handler, assuming the definition `proc {Succeed X} true end`. The replacement of `P1` and `P2` by this dummy procedure ensures that the delayed assigments are only forced once.

The encoding of `Delay` and `Force` in Figure 5 suggests a more natural and lightweight (*i.e.*, efficient) compilation scheme for OzFun [14]. More specifically, it avoids high-level data structures to distinguish proper data from delayed expressions, which unduly increases memory usage and makes conversion procedures necessary.

Figure 6 shows how the standard interpreter for the lazy lambda calculus based on delay/force (see, *e.g.*, [8]) looks like in Oz. To avoid verboseness, we allow some, hopefully intuitive, syntactic sugar to express functions and to nest subexpressions.

6 Conclusion

We have discussed and illustrated the integration of attributed variables into Oz. While the concept of attributed variables and most of their applications have existed before, we claim to have given the first rigorous semantics for attributed variables in constraint logic programming. The surprising fact that this is possible on the high level of Opm promotes them from an implementation-driven low-level concept to a first-class language feature.

Acknowledgements

We thank Peter Van Roy and Seif Haridi for discussions on attributed variables, and Tobias Müller for the inspiration we drew from his C implementation of attributed variables [22]. Manuel Hermenegildo commented on the state of the art in attributed variables. Discussions with Thom Frühwirth and Joachim Niehren on constraint handling also helped. The research reported in this paper has been supported by the Bundesminister für Forschung und Technologie (FTZ-ITW-9105), the Esprit Project ACCLAIM (PE 7195), the Esprit Working Group CCL (EP 6028), and the Graduiertenkolleg Kognitionswissenschaft at the Universität des Saarlandes.

References

[1] P. Brisset, T. Frühwirth, P. Lim, M. Meier, T. L. Prost, J. Schimpf, and M. Wallace. ECL^{i}PS^e 3.4 extensions user manual. Technical report, ECRC GmbH 1994, Jan. 1994.

[2] M. Carlsson. Freeze, Indexing, and Other Implementation Issues in the Wam. In *4th ICLP*, pp. 40–58. University of Melbourne, MIT Press, May 1987.

[3] A. Colmerauer, H. Kanoui, and M. Caneghem. Prolog, Theoretical Principles and Current Trends. *Technology and Science of Informatics*, 2(4):255–292, 83.

[4] G. Erbach. ProFIT: Prolog with Features, Inheritance and Templates. In *7th Conference of the EACL*, Dublin, 1995.

[5] T. Frühwirth and P. Brisset. High-Level Implementation of Constraint Handling Rules. ECRC Technical Report ECRC-TR-95-20, June 1995.

[6] P. S. Group. *SICStus Prolog Users's Manual, Release 3.0.* SICS, PO Box 1263, 164 28 Kista, Sweden, June 1995.

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

[8] J. Hatcliff and O. Danvy. A Generic Account of Continuation-Passing Styles. In *21st ACM Symposium on POPL*, pp. 144–154. ACM Press, New York, 1994.

[9] M. Henz, G. Smolka, and J. Würtz. Object-Oriented Concurrent Constraint Programming in Oz. V. Saraswat and P. Hentenryck, eds., *Principles and Practive of Constraint Programming*, pp. 27–48. MIT Press, Cambridge, MA, 1995.

[10] M. Hermenegildo, D. Cabeza, and M. Carro. Using Attributed Variables in the Implementation of Concurrent and Parallel Logic Programming Systems. In *12th ICLP*, pp. 631–645. MIT Press, June 1995.

[11] C. Holzbaur. *Specification of Constraint Based Inference Mechanisms through Extended Unification.* PhD thesis, Technische Universität Wien, 1990.

[12] C. Holzbaur. *SICStus 2.1/DMCAI Clp 2.1.1 User's Manual.* University of Vienna, 1994.

[13] S. L. Huitouze. A New Data Structure for Implementing Extensions to Prolog. In P. Deransart and J. Małuszyński, editors, *International Symposium on PLILP*, LNCS 456, pp. 136–150. Springer, Berlin, Germany, Aug. 1990.

[14] K. Ibach. OzFun: A Functional Language for Mixed Eager and Lazy Programming. Master's thesis, Univ. des Saarlandes, FB Informatik, Oct. 1995. In German.

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

[16] S. Janson and S. Haridi. Programming Paradigms of the Andorra Kernel Language. In *International Logic Programming Symposium*, pp. 167–186, 1991.

[17] M. J. Maher. Logic Semantics for a Class of Committed-Choice Programs. In J.-L. Lassez, editor, *International Conference on Logic Programming*, pp. 858–876. The MIT Press, Cambridge, MA, 1987.

[18] M. Mehl, R. Scheidhauer, and C. Schulte. An Abstract Machine for Oz. In *International Symposium on PLILP*, LNCS 982, pp. 151–168, Utrecht, Netherlands, Sept. 20–22 1995. Springer, Berlin, Germany.

[19] M. Meier and J. Schimpf. An Architecture for Prolog Extensions. ECRC Technical Report ECRC-95-06, 1995.

[20] R. Milner. The Polyadic π-Calculus: A Tutorial. In F. L. Bauer, W. Brauer, and H. Schwichtenberg, eds., *Proc. 1991 Marktoberndorf Summer School on Logic and Algebra of Specification*, NATO ASI Series. Springer, Berlin, 1993.

[21] M. Müller, J. Niehren, and A. Podelski. Inclusion constraints over non-empty sets of trees. 1996. Submitted for publication.

[22] T. Müller. Integrating New Constraint Systems into DFKI Oz. *1^{st} Int. Workshop on Oz Programming*, Martigny, Switzerland, 29 Nov.–1 Dec. 1995.

[23] U. Neumerkel. Extensible Unification by Metastructures. In *Proc. META*, 1990.

[24] J. Niehren. Functional Computation as Concurrent Computation. *23^{rd} ACM Symposium on POPL*, pp. 333–343, St. Petersburg Beach, Florida, Jan. 1996.

[25] J. Niehren and M. Müller. Constraints for Free in Concurrent Computation. In K. Kanchanasut and J.-J. Lévy, eds., *ACSC*, LNCS, pp. 171–186, Pathumthani, Thailand, 11–13 Dec. 1995. Springer, Berlin, Germany.

[26] B. Pfahringer and J. Matiasek. A CLP Schema to Integrate Specialized Solvers and its Application to Natural Language Processing. Technical report, Austrian Research Institute for Artificial Intelligence, Vienna, 1992.

[27] O. Ridoux. MALIv06: Tutorial and Reference Manual. Technical Report 611, IRISA, Rennes, France, 1991.

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

[29] C. Schulte, G. Smolka, and J. Würtz. Encapsulated Search and Constraint Programming in Oz. In A. Borning, editor, *2^{nd} PPCP*, LNCS 874, pp. 134–150, Orcas Island, Washington, USA, 2-4 May 1994. Springer, Berlin, Germany.

[30] G. Smolka. A Foundation for Concurrent Constraint Programming. In J.-P. Jouannaud, ed., *Proc. CCL*, LNCS 845, pp. 50–72, München, 7–9 Sep. 1994.

[31] G. Smolka. The Oz Programming Model. In J. van Leeuwen, editor, *Computer Science Today*, LNCS 1000, pp. 324–343. Springer, Berlin, Germany, 1995.

[32] G. Smolka and R. Treinen. Records for Logic Programming. *The Journal of Logic Programming*, 18(3):229–258, Apr. 1994.