Oz—A Programming Language for Multi-Agent Systems*

Martin Henz, Gert Smolka, Jörg Würtz

German Research Center for Artificial Intelligence (DFKI)
Stuhlsatzenhausweg 3
D-6600 Saarbrücken
Germany
E-mail: {henz, smolka, wuertz}@dfki.uni-sb.de

Abstract

Oz is an experimental higher-order concurrent constraint programming system under development at DFKI. It combines ideas from logic and concurrent programming in a simple yet expressive language. From logic programming Oz inherits logic variables and logic data structures, which provide for a programming style where partial information about the values of variables is imposed concurrently and incrementally. A novel feature of Oz is that it accommodates higher-order programming without sacrificing that denotation and equality of variables are captured by first-order logic. Another new feature of Oz is constraint communication, a new form of asynchronous communication exploiting logic variables. Constraint communication avoids the problems of stream communication, the conventional communication mechanism employed in concurrent logic programming. Constraint communication can be seen as providing a minimal form of state fully compatible with logic data structures.

Based on constraint communication and higher-order programming, Oz readily supports a variety of object-oriented programming styles including multiple inheritance.

1 Introduction

Oz is an attempt to create a high-level concurrent programming language bringing together the merits of logic and object-oriented programming in a unified language.

Our natural starting point was concurrent constraint programming [Saraswat and Rinard, 1990], which brings together ideas from constraint and concurrent logic programming. Constraint logic programming [Jaffar and Lassez, 1987, Colmerauer and Benhamou, 1993], on the one hand, originated with Prolog II [Colmerauer *et al.*, 1983] and was prompted by the need to integrate numbers and data structures in an operationally efficient, yet logically sound manner. Concurrent logic programming [Shapiro, 1989], on the other hand, originated with the Relational Language [Clark and Gregory, 1981] and was promoted by the Japanese Fifth Generation Project, where logic programming was conceived as the basic system programming language and thus had to account for concurrency, synchronization and indeterminism. For this purpose, the conventional SLD-resolution scheme had to be replaced with a new computation model based on the notion of committed choice. At first, the new model developed as an ad hoc construction, but finally Maher [Maher, 1987] realized that commitment of agents can be captured logically as constraint entailment. A major landmark in the new field of concurrent constraint programming is AKL [Janson and Haridi, 1991], the first implemented concurrent constraint language accommodating search and deep guards.

Saraswat's concurrent constraint model [Saraswat and Rinard, 1990] can accommodate object-oriented programming along the lines of Shapiro's stream-based model for Concurrent Prolog [Shapiro and Takeuchi, 1983]. However, this model is intolerably low-level due to the clumsiness of stream communication and the lack of higher-order programming facilities. This becomes fully apparent when the model is extended to provide for inheritance [Goldberg *et al.*, 1992].

Thus the two essential innovations Oz has to provide to be well-suited for object-oriented programming are better communication and a facility for higher-order programming. Both innovations require stepping outside of established semantical foundations. The semantics of Oz is thus specified by a new mathematical model, called the Oz Calculus, whose technical set-up was inspired by the π-calculus [Milner, 1991], a recent foundationally motivated model of concurrency.

The way Oz provides for higher-order programming is unique in that denotation and equality of variables are captured by first-order logic only. In fact, denotation of variables and the facility for higher-order programming are completely orthogonal concepts in Oz. This is in contrast to existing approaches to higher-order logic programming [Nadathur and Miller, 1988, Chen *et al.*, 1993].

Constraint communication is asynchronous and indeterministic. A communication event replaces two complementary communication tokens with an equation linking the partners of the communication. Constraint communication introduces a minimal form of state that

*This work has been supported by the Bundesminister für Forschung und Technologie, contract ITW-9105.

is fully compatible with logic data structures. Efficient implementation of fair constraint communication is straightforward.

The paper is organized as follows. The next section outlines a simplified version of the Oz Calculus. Section 3 shows how Oz accommodates records as a logic data structure. The remaining sections present one possible style of concurrent object-oriented programming featuring multiple inheritance.

2 The Oz Calculus

The operational semantics of Oz is defined by a mathematical model called the Oz Calculus [Smolka, 1993]. In this section we outline a simplified version sufficing for the purposes of this paper.

The basic notion of Oz is that of a computation space. A computation space consists of a number of agents connected to a blackboard (see Fig. 1). Each agent reads the blackboard and reduces once the blackboard contains the information it is waiting for. The information on the blackboard increases monotonically. When an agent reduces, it may put new information on the blackboard and create new agents. Agents themselves may have one

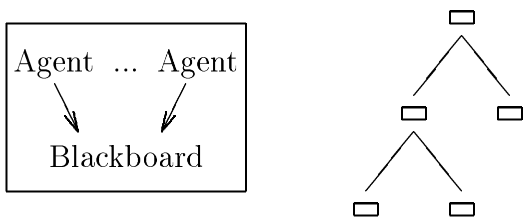

Figure 1: Computation Model

or several local computation spaces. Hence the entire computation system is a tree-like structure of computation spaces (see Fig. 1).

The agents of a computation space are agents at the micro-level. They are used to program agents at the macro-level. One interesting form of macro-agents are the objects we will introduce in a later section of this paper.

Formally, a computation state is an expression according to Fig. 2. (If ξ is a syntactic category, $\overline{\xi}$ denotes a possibly empty sequence $\xi \ldots \xi$.) Constraints, abstractions and communication tokens reside on the blackboard. Applications and conditionals are agents. Composition and quantification are the glue assembling agents and blackboard items into a computation space. Quantification introduces local variables. Abstractions may be seen as procedure definitions and applications as procedure calls.

The clauses of a conditional are unordered. Their guards, i.e., σ in $\exists\overline{x}\,(\sigma$ **then** $\tau)$, constitute local computation spaces. Note that any expression can be taken as a guard; one speaks of a flat guard if the guard is a constraint.

There are two variable binders: quantification $\exists x\sigma$ binds x with scope σ and abstraction $x{:}\overline{y}/\sigma$ binds the variables in $\overline{y}$ with scope σ. Free variables of an expression are defined accordingly.

x, y, z	:		*variables*
σ, τ, μ	$::=$		
	ϕ		*constraint*
	$x{:}\overline{y}/\sigma$		*abstraction*
	$x\,!\,y$		*put token*
	$x?y$		*get token*
	$x\overline{y}$		*application*
	if $\omega_1 \ \ldots\ \omega_n$ **else** σ		*conditional*
	$\sigma \wedge \tau$		*composition*
	$\exists x\sigma$		*quantification*
ω	$::=$	$\exists\overline{x}\,(\sigma$ **then** $\tau)$	*clause*
ϕ, ψ	$::=$	$\bot \mid \top \mid s \doteq t \mid r(\overline{s}) \mid \phi \wedge \psi$	

Figure 2: Expressions of the Oz Calculus

Computation is defined as reduction (i.e., rewriting) of expressions. A reduction step is performed by applying a reduction rule to a subexpression satisfying the application conditions of the rule. There is no backtracking. Control is provided by the provision that reduction rules must not be applied to *mute* subexpressions, i.e., subexpressions that occur within bodies of clauses, else parts of conditionals, or bodies of abstractions. It is up to the implementation which non-mute subexpression is rewritten with which applicable rule.

Reduction "$\sigma \to \tau$" is defined modulo structural congruence "$\sigma \equiv \tau$" of expressions, that is, satisfies the inference rule

$$\frac{\sigma \equiv \sigma' \quad \sigma' \to \tau' \quad \tau' \equiv \tau}{\sigma \to \tau}\,.$$

Structural congruence is an abstract equality for computation states turning them from purely syntactic objects into semantical objects. Structural congruence provides for associativity and commutativity of composition, renaming of bound variables, quantifier mobility

$$\exists x\sigma \wedge \tau \equiv \exists x(\sigma \wedge \tau) \qquad \text{if } x \text{ does not occur free in } \tau,$$

constraint simplification, and information propagation from global blackboards to local blackboards.

2.1 Constraints

Constraints (ϕ,ψ in Figure 2) are formulas of first-order predicate logic providing for data structures. Logical conjunction of constraints coincides with composition of expressions. Constraints express partial information about the values of variables. The semantics of constraints is defined logically by a first-order theory Δ and imposed with the congruence law

$$\phi \equiv \psi \qquad \text{if } \Delta \models \phi \leftrightarrow \psi.$$

This law closes the blackboard under entailed constraints (since $\Delta \models \phi \to \psi$ iff $\Delta \models \phi \leftrightarrow \phi \wedge \psi$). The congruence law

$$x \doteq y \wedge \sigma \;\equiv\; x \doteq y \wedge \sigma[y/x] \qquad \text{if } y \text{ is free for } x \text{ in } \sigma$$

imposes equalities on the blackboard to the rest of the computation space ($\sigma[y/x]$ is obtained from σ by replacing every free occurrence of x with y). Equality of variables is strictly first-order: Two variables x, y are equal if the constraints on the blackboard entail $x \doteq y$, and different if the constraints on the blackboard entail $\neg(x \doteq y)$. Of course, the information on the blackboard may be insufficient to determine whether two variables are equal or different. Moreover, an inconsistent blackboard entails both $x \doteq y$ and $\neg(x \doteq y)$.

The Anullation Law

$$\exists \overline{x}(\phi \;\wedge\; \overline{y}{:}\overline{\alpha}) \;\equiv\; \top$$

$$\text{if } \Delta \models \exists \overline{x}\, \phi \text{ and } \overline{y} \subseteq \mathcal{L}(\overline{x}, \phi), \text{ where}$$
$$\mathcal{L}(\overline{x}, \phi) \;:=\; \{y \in \overline{x} \mid \forall z{:}\; \phi \models_\Delta y \doteq z \Rightarrow z \in \overline{x}\}$$

provides for the deletion of quantified constraints and abstractions not affecting visible variables.

2.2 Application

An application agent $x\overline{y}$ waits until an abstraction for its *link* x appears on the blackboard and then reduces as follows:

$$x\overline{y} \;\wedge\; x{:}\overline{z}/\sigma \;\rightarrow\; \exists \overline{z}\,(\overline{z} \doteq \overline{y} \wedge \sigma) \;\wedge\; x{:}\overline{z}/\sigma$$

$$\text{if } \overline{x} \text{ and } \overline{y} \text{ are disjoint and of equal length.}$$

Note that the blackboard $y{:}\overline{z}/\sigma \wedge x \doteq y$ contains an abstraction for x due to the congruence laws stated above. Since the link x of an abstraction $x{:}\overline{y}/\sigma$ is a variable like any other, abstractions can easily express higher-order procedures. Note that an abstraction $x{:}\overline{y}/\sigma$ does not impose any constraints (e.g., equalities) on its link x.

2.3 Constraint Communication

The semantics of the two communication tokens is defined by the Communication Rule:

$$x\,!\,y \wedge z?y \;\rightarrow\; x \doteq z.$$

Application of this rule amounts to an indeterministic transition of the blackboard replacing two complementary communication tokens with an equality constraint. The Communication Rule is the only rule deleting items from the blackboard. Since agents read only constraints and abstractions, the information visible to agents nevertheless increases monotonically.

2.4 Conditional

It remains to explain the semantics of a conditional agent

$$\textbf{if } \exists \overline{x}_1\,(\sigma_1 \textbf{ then } \tau_1) \;\cdots\; \exists \overline{x}_n\,(\sigma_n \textbf{ then } \tau_n) \textbf{ else } \mu.$$

The guards σ_i of the clauses are local computation spaces reducing concurrently. For the local computations to be meaningful it is essential that information from global blackboards is visible on local blackboards. This is achieved with the Propagation Law (recall that the clauses are unordered):

$$\pi \;\wedge\; \textbf{if } \exists \overline{x}\,(\sigma \textbf{ then } \tau) \;\; \overline{\omega} \textbf{ else } \mu$$
$$\equiv$$
$$\pi \;\wedge\; \textbf{if } \exists \overline{x}\,(\pi \wedge \sigma \textbf{ then } \tau) \;\; \overline{\omega} \textbf{ else } \mu$$

if π is a constraint or abstraction and no variable in $\overline{x}$ appears free in π.

Read from left to right, the law provides for copying information from global blackboards to local blackboards. Read from right to left, the law provides for deletion of local information that is present globally. An example verified by employing the Propagation Law in both directions (as well as constraint simplification) is

$$x \doteq 1 \;\wedge\; \textbf{if } (x \doteq 1 \textbf{ then } \sigma)\; (x \doteq 2 \textbf{ then } \tau) \textbf{ else } \mu$$
$$\equiv\; x \doteq 1 \;\wedge\; \textbf{if } (\top \textbf{ then } \sigma)\; (\bot \textbf{ then } \tau) \textbf{ else } \mu.$$

The example assumes that the constraint theory entails that 1 and 2 are different.

Operationally, the constraint simplification and propagation laws can be realized with a so-called relative simplification procedure. Relative simplification for the constraint system underlying Oz is investigated in [Smolka and Treinen, 1992].

There are two distinguished forms a guard of a clause may eventually reduce to, called satisfied and failed. If a guard of a clause is satisfied, the conditional can reduce by committing to this clause:

$$\textbf{if } \exists \overline{x}\,(\sigma \textbf{ then } \tau) \;\; \overline{\omega} \textbf{ else } \mu \;\rightarrow\; \exists \overline{x}\,(\sigma \wedge \tau) \qquad \text{if } \exists \overline{x}\, \sigma \equiv \top.$$

Reduction puts the guard on the global blackboard and releases the body of the clause.

A guard is failed if the constraints on its blackboard are unsatisfiable. If the guard of a clause is failed, the clause is simply discarded:

$$\textbf{if } \exists \overline{x}\,(\bot \wedge \sigma \textbf{ then } \tau) \;\; \overline{\omega} \textbf{ else } \mu \;\rightarrow\; \textbf{if } \overline{\omega} \textbf{ else } \mu.$$

Thus a conditional may end up with no clauses at all, in which case it reduces to its else part:

$$\textbf{if else } \mu \;\rightarrow\; \mu.$$

The reduction

$$x \doteq 1 \;\wedge\; \textbf{if } (x \doteq 1 \textbf{ then } \sigma)\; (x \doteq 2 \textbf{ then } \tau) \textbf{ else } \mu$$
$$\rightarrow\; x \doteq 1 \;\wedge\; \sigma$$

is an example for the application of the first rule, and

$$x \doteq 3 \;\wedge\; \textbf{if } (x \doteq 1 \textbf{ then } \sigma)\; (x \doteq 2 \textbf{ then } \tau) \textbf{ else } \mu$$
$$\rightarrow^*\; x \doteq 3 \;\wedge\; \mu$$

is an example employing the other two reduction rules.

2.5 Logical Semantics

The subcalculus obtained by disallowing communication tokens and conditionals with more than one clause enjoys a logical semantics by translating expressions into formulas of first-order predicate logic as follows (composition is interpreted as conjunction, and quantification is interpreted as existential quantification):

$$x{:}\overline{y}/\sigma \;\Longrightarrow\; \forall \overline{y}\,(\mathsf{apply}(x\overline{y}) \leftrightarrow \sigma)$$
$$x\overline{y} \;\Longrightarrow\; \mathsf{apply}(x\overline{y})$$
$$\textbf{if } \exists \overline{x}\,(\sigma \textbf{ then } \tau) \textbf{ else } \mu \;\Longrightarrow\; \exists \overline{x}\,(\sigma \wedge \tau) \;\vee\; (\neg \exists \overline{x} \sigma \wedge \mu).$$

Under this translation, reduction is an equivalence transformation, that is, if $\sigma \to \tau$ or $\sigma \equiv \tau$, then $\Delta \models \sigma \leftrightarrow \tau$. Moreover, negation can be expressed since $\neg\sigma$ is equivalent to **if** σ **then** $\bot$ **else** $\top$.

2.6 Unique Names

A problem closely related to equality and of great importance for concurrent programming is the dynamic creation of new and unique names. Roughly, one would like a construct gensym(x) such that

$$\mathsf{gensym(x)} \wedge \mathsf{gensym(y)}$$

is congruent to a constraint entailing $\neg(x \doteq y)$. For this purpose we assume that there are infinitely many distinguished constant symbols called *names* such that the constraint theory Δ satisfies:

1. $\Delta \models \neg(a \doteq b)$ for every two distinct names a, b
2. $\Delta \models S \leftrightarrow S[a/b]$ for every logical sentence S and every two names a, b ($S[a/b]$ is obtained from S by replacing every occurrence of b with a).

Now gensym(x) is modeled as a generalized quantification $\exists a(x \doteq a)$, where the quantified name a is subject to α-renaming. With that and the quantifier mobility stated above we in fact obtain a constraint in which x and y are different:

$$\begin{aligned}\exists a(x \doteq a) \wedge \exists a(y \doteq a) &\equiv \exists a(x \doteq a) \wedge \exists b(y \doteq b)\\ &\equiv \exists a \exists b(x \doteq a \wedge y \doteq b).\end{aligned}$$

3 Records

The constraint system underlying Oz provides a domain that is closed under record construction [Smolka and Treinen, 1992]. We now outline its constraint theory as far as is needed for the rest of this paper. We will be very liberal as it comes to syntax. The reader may consult [Smolka and Treinen, 1992] for details.

Records are obtained with respect to an alphabet of constant symbols, called *atoms*, and denoted by a, b, f, g. Records are constructed and decomposed by constraints of the form

$$x \doteq f(a_1{:}\,x_1 \ldots a_n{:}\,x_n)$$

where f is the *label*, $a_1, \ldots, a_n$ are the field names, and $x_1, \ldots, x_n$ are the corresponding values of record x. The order of the fields $a_i{:}\,x_i$ is not significant. The semantics of the above constraint is fixed by two axiom schemes

$$\begin{aligned} f(\overline{a}{:}\,\overline{x}) \doteq f(\overline{a}{:}\,\overline{y}) &\leftrightarrow \overline{x} \doteq \overline{y} \\ f(\overline{a}{:}\,\overline{x}) \doteq g(\overline{b}{:}\,\overline{y}) &\to \bot \qquad \text{if } f \neq g \text{ or } [\overline{a}] \neq [\overline{b}] \end{aligned}$$

where $[\overline{a}]$ is the set of elements of the sequence $\overline{a}$.

Field selection $x.y$ is a partial function on records defined by the axiom schemes

$$\begin{aligned} f(\overline{a}{:}\,\overline{x}\ b{:}\,y)\,.\,b &\doteq y \\ f(\overline{a}{:}\,\overline{x})\,.\,b \doteq y &\to \bot \qquad \text{if } b \notin [\overline{a}]. \end{aligned}$$

The function label(x) is defined on records by the scheme

$$\mathsf{label}(f(\cdots)) \doteq f.$$

Finally, record adjunction "adjoinAt(x, y, z)" is defined by the schemes:

$$\begin{aligned} \mathsf{adjoinAt}(f(\overline{a}{:}\,\overline{x}\ b{:}\,y),\ b,\ z) &\doteq f(\overline{a}{:}\,\overline{x}\ b{:}\,z) \\ \mathsf{adjoinAt}(f(\overline{a}{:}\,\overline{x}),\ b,\ z) &\doteq f(\overline{a}{:}\,\overline{x}\ b{:}\,z) \qquad \text{if } b \notin [\overline{a}]\,. \end{aligned}$$

We write $f(x_1 \ldots x_n)$ as a short hand for $f(1{:}\,x_1 \ldots n{:}\,x_n)$. Thus we obtain Prolog terms as a special case of records.

4 Synchronous Communication

Constraint communication is asynchronous. The following program shows how synchronous comunication can be expressed using constraint communication. Computation only proceeds after communication has taken place (signaled by an acknowledgement).

```
proc {Producer}
   exists Ack in
   item('yellow brick' Ack 1) ! Channel
   if Ack = 1 then {Producer} fi
end

proc {Consumer}
   exists X Ack in
   item(X 1 Ack) ? Channel
   if Ack = 1
   then {AddToRoad X} {Consumer} fi
end
```

We have now switched to the concrete syntax of Oz: **pred** $\{x\ \overline{y}\}\ \sigma$ **end** stands for $x{:}\,\overline{y}/\sigma\ \wedge\ \exists a(x \doteq a)$, $\{x\ \overline{y}\}$ for $x\overline{y}$, and juxtaposition for composition. Moreover, nesting is allowed and is eliminated by conjunction and quantification; e.g. *item(X 1 Ack)* ? *Channel* expands to **exists** *Y* **in** *Y=item(X 1 Ack) Y* ? *Channel*. Finally, the default for a missing else part of a conditional is **else true**.

5 Objects

An object has a static aspect, its *method table*, and a dynamic aspect, its *state*. Methods are functions

$$\mathit{method} : \mathit{state} \times \mathit{message} \to \mathit{state}.$$

A method table is a mapping from method names to methods, represented as a record whose field names act as method names. A message is a record, whose label is the name of the method and whose fields are arguments.

It turns out that we can represent an object O by the procedure that sends the message. This representation gives a unique identity to the object since **proc** $\{x\ \overline{y}\}\ \sigma$ **end** stands for $x{:}\,\overline{y}/\sigma\ \wedge\ \exists a(x \doteq a)$.

```
proc {O Message}
   if MethodName Method in
      MethodName={ label Message}
      Method=MethodTable.MethodName
   then exists State in
       State ? C
       if { label State}=state
       then {Method State Message} ! C fi
   fi
end
```

Observe that *nested application* makes programs more concise: {*Method State Message*} ! *C* stands for

```
exists NState in
{Method State Message NState}  NState ! C
```

When a message is received by the object O, the method associated with the method name is retrieved using the method table of the object (i.e., late binding). Then the state of the object is replaced by the state obtained by applying the method.

The following procedure provides a generic scheme for creating objects from a method table and an initial message.

```
proc {Create IMessage MethodTable O}
    exists IMethod C in
    IMethod= MethodTable.{Label IMessage}
    {IMethod state(self:O) IMessage} ! C
    proc {O Message} ... end
end
```

Observe that the notion of "self" is provided in a natural way by starting with the initial state *state*(*self* : *O*). Object initialization is provided by applying an initial message to that state. The resulting state is written on the blackboard. Now, the object is ready to receive messages. We abbreviate message sending of the form {*O M*} by *O* ˆ*M*. Note that quantification of the communication link *C* hides the state and provides for data encapsulation.

6 Methods

Assume that we want to model a counter as an object. First, we fix the methods to be stored in the method table. To initialize the counter we use the method

```
proc {Init InS X OutS}
    if Y in X = init(Y)
    then OutS = { adjoinAt InS val Y} fi
end
```

Observe that *Init* will add the attribute *val* if it is not present in the state *InS* (see the semantics of adjoinAt in Section 3). To ease the treatment of the state and to get a more elegant notation we abbreviate this abstraction by

```
meth ⟪Init init(Y)⟫ val ← Y end
```

Incrementing and retrieving is achieved by

```
proc {Inc InS X OutS}
    if X = inc
    then OutS = { adjoinAt InS val InS.val + 1} fi
end

proc {Get InS X OutS}
    if Y in X = get(Y)
    then OutS = InS   Y = InS.val fi
end
```

which is abbreviated to

```
meth ⟪Inc inc⟫ val ← @val + 1 end
meth ⟪Get get(Y)⟫ Y = @val end
```

A counter is created by

```
MT = mt(init:Init inc:Inc get:Get)
{Create init(0) MT Counter}
```

7 Inheritance

In our framework, inheritance amounts to using the method tables of other objects to build the method table of a new object. We modify the procedure *Create* to provide for inheritance.

```
proc {Create Ancestors IMessage
        NewMethods O}
    exists IMethodName IMethod C
            AllMethods Send in
    ...
    AllMethods =
        {AdjoinAll Ancestors NewMethods}
    O = object(methods:AllMethods
                send:Send)
    proc {Send Message} ... end
end
```

The procedure *AdjoinAll* (not shown) adjoins the method tables of *Ancestors* and *NewMethods* from left to right: For any method name, the rightmost method definition is taken (cf. adjoinAt in Section 3).

To make the methods of objects accessible, an object is now represented as a record containing the methods and the send procedure. Therefore, message sending changes slightly: *Counter* ˆ*inc* stands now for {*Counter*.*send inc*}.

A counter that is displayed in a window (the object *VisibleObject* is defined in Section 9) and that can additionally decrement its value can be created by

```
meth ⟪Dec dec⟫ val ← @val − 1 end
DecCounter =
    {Create Counter|VisibleObject|nil
            init(0) mt(dec:Dec)}
```

for which we introduce the following syntactic sugar.

```
create DecCounter
    from Counter VisibleObject
    with init
    meth dec  val ← @val − 1 end
end
```

8 Method Application

Some languages providing for inheritance support the concept of *super* to address methods overwritten due to the inheritance priority. Oz provides a more general scheme in that an object can apply to its state methods of any other object (regardless of inheritance).

Assume an already defined object *Rectangle*. A square can inherit from a rectangle but needs for initialization only its length but not its width.

```
create Square from Rectangle with init(10)
    meth init(X)
        《 (Rectangle.methods).init init(X X) 》
    end
    ...
end
```

where the method expands to

```
proc {Init InS X OutS}
    if Y in X = init(Y)
    then OutS = {Rectangle.methods.init
               InS init(Y Y)} fi
end
```

Note that 《@*self*.*methods* *m*》 differs from @*self*ˆ*m* in that the former tranforms the local state immediately, whereas other messages can be taken before the latter is eventually executed.

9 Meta Object Protocol

Now, we modify the object system such that the essentials of object creation and message sending can be inherited, providing the object-system with a meta object protocol like in [Kiczales *et al.*, 1991] for CLOS. The new definition of *Create* uses the meta-method *create* to describe the object's behavior.

```
proc {Create Ancestors IMessage NewMethods O}
    exists AllMethods in
    AllMethods =
      {AdjoinAll Ancestors NewMethods}
    {AllMethods.create
      _ create(AllMethods IMessage O) _}
end
```

The underscore "_" denotes an anonymous variable occurring only once.

Like an organism, an object can inherit the way it and its heirs are created, and the basic structure how it communicates with its environment.

We can further modularize the object protocol such that, e.g., each method call is performed by a call to the meta-method *methodCall*. Assume that the meta-methods *create* and *methodCall* are defined in the object *MetaObject*. In this case, a *VisibleObject* that sends a message containing its current state to a *Display* whenever it executes a method, can be created as follows:

```
create VisibleObject from MetaObject
    meth methodCall(InS Meth Mess OutS)
        {Meth InS Mess OutS}
        Display ˆ show(OutS)
    end
end
```

Acknowledgements

We thank all members of the Programming Systems Lab at DFKI for countless fruitful discussions on all kinds of subjects and objects; particularly many suggestions came from Michael Mehl and Ralf Scheidhauer.

References

[Chen *et al.*, 1993] W. Chen, M. Kifer, and D. S. Warren. Hilog: A foundation for higher-order logic programming. *Journal of Logic Programming*, pages 187–230, 1993.

[Clark and Gregory, 1981] K.L. Clark and S. Gregory. A relational language for parallel programming. In *Proc. of the ACM Conference on Functional Programming Languages and Computer Architecture*, pages 171–178, 1981.

[Colmerauer and Benhamou, 1993] A. Colmerauer and F. Benhamou, editors. *Constraint Logic Programming: Selected Research*. 1993. To appear.

[Colmerauer *et al.*, 1983] A. Colmerauer, H. Kanoui, and M. Van Caneghem. Prolog, theoretical principles and current trends. *Technology and Science of Informatics*, 2(4):255–292, 1983.

[Goldberg *et al.*, 1992] Y. Goldberg, W. Silverman, and E. Shapiro. Logic programs with inheritance. FGCS, pages 951–960, 1992.

[Jaffar and Lassez, 1987] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the Annual ACM Symposium on Principles of Programming Languages*, pages 111–119, 1987.

[Janson and Haridi, 1991] S. Janson and S. Haridi. Programming paradigms of the Andorra kernel language. In *Logic Programming, Proceedings of the 1991 International Symposium*, pages 167–186, 1991.

[Kiczales *et al.*, 1991] G. Kiczales, J. des Rivières, and D. Bobrow. *The Art of the Metaobject Protocol*. 1991.

[Kahn, 1989] K.M. Kahn. Objects: A fresh look. In *Proceedings of the European Conference on Object Oriented Programming*, pages 207–223, 1989.

[Maher, 1987] M. J. Maher. Logic semantics for a class of committed-choice programs. In *Logic Programming, Proceedings of the Fourth International Conference*, pages 858–876, 1987.

[Milner, 1991] R. Milner. The polyadic π-calculus: A tutorial. ECS-LFCS Report Series 91-180, University of Edinburgh, 1991.

[Nadathur and Miller, 1988] G. Nadathur and D. Miller. An overview of λProlog. In *Logic Programming: Proceedings of the Fifth International Conference and Symposium, Volume 1*, pages 810–827, 1988.

[Saraswat and Rinard, 1990] V.A. Saraswat and M. Rinard. Concurrent constraint programming. In *Proceedings of the 7th Annual ACM Symposium on Principles of Programming Languages*, pages 232–245, 1990.

[Shapiro and Takeuchi, 1983] E. Shapiro and A. Takeuchi. Object oriented programming in Concurrent Prolog. *New Generation Computing*, 1:24–48, 1983.

[Shapiro, 1989] E. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):413–510, 1989.

[Smolka and Treinen, 1992] G. Smolka and R. Treinen. Records for logic programming. In *Proceedings of the 1992 Joint International Conference and Symposium on Logic Programming*, pages 240–254,1992.

[Smolka, 1993] G. Smolka. A calculus for higher-order concurrent constraint programming. Research report, DFKI, 1993. Forthcoming.