

Oz/Ks: a kernel language for a dynamic FRACTAL ADL

Alan Schmitt, Jean-Bernard Stefani

INRIA Rhône-Alpes

Outline

- 1 Objectives
- 2 Design goals
- 3 Related work & contributions
- 4 Oz/Ks abstract syntax
- 5 Oz/Ks operational semantics
- 6 Programming examples
- 7 Conclusion

- 1 Objectives
- 2 Design goals
- 3 Related work & contributions
- 4 Oz/Ks abstract syntax
- 5 Oz/Ks operational semantics
- 6 Programming examples
- 7 Conclusion

Small kernel language for distributed component-based programming :

- Basis for a full programming language, in the tradition of Pict and Mozart
- Studying the conjunction of multiple programming paradigms with Fractal-like software structures
- Programming support for distributed deployment, distributed dynamic reconfiguration and dynamic software evolution
- Basis for a dynamic Fractal ADL

Programming support for dynamic software architecture :

- manipulating attributes (component meta-data)
- manipulating component structure
- manipulating component packages
- manipulating persistent and control flow state
- exception handling
- event-condition-action (ECA) rules
- distributed workflow patterns (e.g. join, split, iterations, etc)
- rollback and recovery
- atomicity conditions

Outline

- 1 Objectives
- 2 Design goals**
- 3 Related work & contributions
- 4 Oz/Ks abstract syntax
- 5 Oz/Ks operational semantics
- 6 Programming examples
- 7 Conclusion

Multi-paradigm To retain the expressive power, and the multiple programming styles that are available with the MOZART computation model.

Compositionality To retain the compositionality that comes with the procedural abstractions in MOZART.

Reconfigurability To extend the MOZART computation model with a notion of component that can be used as a unit of isolation, and un-planned reconfiguration.

- Sharing** To support the definition of software architectures with shared components.
- Security** To support the construction of sandboxes, allowing the proper isolation of components from others, even in presence of sharing.
- Network** To retain for the Oz/Ks operational semantics the local and network independent character of the Kell calculus abstract machine.

- 1 Objectives
- 2 Design goals
- 3 Related work & contributions**
- 4 Oz/Ks abstract syntax
- 5 Oz/Ks operational semantics
- 6 Programming examples
- 7 Conclusion

- Large body of work on ADLs of various sorts (e.g. C2, Rapide, Wright, Darwin, Unicon, CommUnity, Olan, Fractal)
- Strangely, not that many attempts to embed software architecture concepts into programming languages. Some recent ones : ArchJava, Cell, ComponentJ, Jiazzi.
- Large body of work on languages for distributed programming. Recent relevant efforts : Acute, Mozart, Alice, Nomadic Pict, Erlang, Distributed Scheme
- Much relevant work around Mozart and its extensions : mobile threads, failure semantics and annotations, membranes, security, pickles.

- 1 Extend MOZART core (syntax and formal operational semantics) with support for :
 - Software architecture concepts at different granularities
 - Pickling/unpickling
 - Programmable code and task mobility
 - Dynamic isolation and sandboxing
 - Core support for dynamic reconfiguration
- 2 Extend FRACTAL ADL for the description of dynamic distributed software architectures, with support for :
 - Workflow patterns
 - Dynamic reconfiguration
 - Higher-order constructs
 - Component packages

Outline

- 1 Objectives
- 2 Design goals
- 3 Related work & contributions
- 4 Oz/Ks abstract syntax**
- 5 Oz/Ks operational semantics
- 6 Programming examples
- 7 Conclusion

S ::= skip	<i>empty statement</i>
S₁ S₂	<i>sequential composition</i>
thread{P} S end	<i>thread creation</i>
local X₁ ... X_n in S end	<i>variable creation (n ≥ 1)</i>
X = Y	<i>imposing equality</i>
X = u	<i>binding to a base value</i>
X = f(l₁ : X₁ ... l_n : X_n)	<i>binding to a record (n ≥ 1)</i>
if X then S₁ else S₂ end	<i>branch statement</i>
case X of J then S₁ else S₂	<i>pattern matching</i>
{newCap C Y}	<i>capability & name creation</i>
...	

$S ::= \dots$

proc { $P X_1 \dots X_n$ } S end	<i>procedure creation</i>
{ $P X_1 \dots X_n$ }	<i>procedure call</i>
{ isDet $X Y$ }	<i>testing bound status</i>
{ newCell $X C$ }	<i>cell creation</i>
{ exchange $C X Y$ }	<i>cell read-and-update</i>
{ readOnly $X Y$ }	<i>readonly variable</i>
try S_1 catch X then S_2 end	<i>exception handling</i>
raise X end	
...	

Example : List iterator in MOZART

Illustrating higher-order programming : an iterator `ForAll` that applies a unary procedure `Proc` to all elements of a list `L`.

```
proc{ForAll L Proc}
  case L
  of nil then skip
  [] H | T then
    {Proc H}
    {ForAll T Proc}
  end
end
```

```
proc{ForAll L Proc}
  case L of nil
  then skip
  else case L of H | T
  then {Proc H}
       {ForAll T Proc}
  else skip
  end
  end
end
```

Example : For iterator in MOZART

Illustrating higher-order programming : an iterator `For` that applies a unary procedure `Proc` to integers from `From` to `To`.

```
local
  proc {ForUp C To Step P}
    if C=<To then {P C} {ForUp C+Step To Step P} end
  end
  proc {ForDown C To Step P}
    if C>=To then {P C} {ForDown C+Step To Step P} end
  end
in
  proc {For From To Step P}
    if Step>0 then {ForUp From To Step P}
    else {ForDown From To Step P} end
  end
end
```

Example : Classes in MOZART

A class C can be constructed as a record of a method table and a list of attribute names. A method is a procedure with 3 arguments : invocation message, state of the invoked object, and reference to the invoked object. An invocation message is a record whose label denotes a method name.

```
local MethTable AttTable in
  AttNames = [a1, ..., am]
  MethTable = m(init: P0 op1: P1 ... opn: Pn)
  proc{P0 M State Self} ... end
  proc{P1 M State Self} ... end
  ...
  proc{Pn M State Self} ... end
  C = c(methods: MethTable atts: AttNames)
end
```

Example : Objects in MOZART

An object `Obj` can be constructed as a procedure that encapsulates some state (by lexical scoping). The state of the object (shared by all methods) is constructed as a record of cells. Method invocation consists in calling the procedure whose name in the method table appears as the label of the message `M`.

```
proc {New Class Init Obj}
  local State P in
    proc{P A} local Y in {NewCell Y A} end end
    State = {MakeRecord s Class.atts}
    {Record.forAll State P}
    proc{Obj M}
      {Class.methods.{Label M} M State Obj}
    end
    {Obj Init}
  end
end
end
```

Component basis

$S ::= \dots$

softK { $K X i(X_1 \dots X_n) o(Y_1 \dots Y_m)$ }	<i>soft kell creation</i>
hardK { $K X i(X_1 \dots X_n) o(Y_1 \dots Y_m)$ }	<i>hard kell creation</i>
{ pack $X Y$ }	<i>task passivation</i>
{ unpack $X Y$ }	<i>task unpacking</i>
{ inPort P }	<i>input port creation</i>
{ outPort P }	<i>output port creation</i>
{ bind $X Y$ }	<i>binding ports</i>
{ unbind $X Y$ }	<i>unbinding ports</i>
{ send $P X$ }	<i>message send</i>
{ receive $P X$ }	<i>message receive</i>

$J ::= j$	<i>basic pattern</i>
$\langle X \rangle$	<i>stopped thread</i>
$[X]$	<i>stopped kell</i>
$\text{pack}(X, Y)$	<i>packed value</i>

$j ::= v$	<i>base value</i>
base	<i>base value type</i>
X	<i>pattern variable</i>
$f(l_1 : j_1 \dots l_n : j_n)$	<i>record pattern</i>

Example : An updatable server that listens on ports

A server that listens on two input ports P_1 and P_2 , whose behavior is given by the recursive procedure *Serve*, and which can be changed upon receipt of an update message.

```
local T in
  proc{Serve In1 In2}
    local M1 M2 in
      {receive In1 M1}
      {receive In2 M2}
      case M1 of update(P) then {P In1 In2} else ... end
    end
  end
  {inPort P1}
  {inPort P2}
  thread{T}{Serve P1 P2} end
end
```

Outline

- 1 Objectives
- 2 Design goals
- 3 Related work & contributions
- 4 Oz/Ks abstract syntax
- 5 Oz/Ks operational semantics**
- 6 Programming examples
- 7 Conclusion

- Given through a reduction relation \rightarrow , that relates task configurations, i.e. pairs of the form $\langle \mathcal{T}, \sigma \rangle$, where \mathcal{T} is a task and σ is a store.

- $\langle \mathcal{T}, \sigma \rangle \rightarrow \langle \mathcal{T}', \sigma' \rangle$ is noted $\frac{\mathcal{T} \parallel \mathcal{T}'}{\sigma \parallel \sigma'}$

- Tasks are given by the grammar :

$$\begin{array}{ll} \mathcal{T} ::= t \mid \eta[\mathcal{T}] \mid \mathcal{T} \mathcal{T} & \text{tasks} \\ t ::= \eta\langle \rangle \mid \eta\langle \mathcal{S} \ t \rangle & \text{threads} \end{array}$$

- Stores are sets of variable and name bindings, together with additional execution information (e.g. bindings between ports, between names and capabilities).

Store grammar

$\sigma ::= x = \perp$	<i>unbound variable</i>
$x = u$	<i>variable bound to base value</i>
$x = f(l_1 : x_1 \dots f_n : x_n)$	<i>variable bound to record</i>
$x = \text{pack}(T, \sigma)$	variable bound to packed value
$\xi : \text{proc}\{\$ X_1 \dots X_n\}S$	<i>procedure value</i>
$\xi : \text{thread}$	thread pointer
$\xi : \text{softK}$	soft kell pointer
$\xi : \text{hardK}$	hard kell pointer
$\xi : \text{cell}(x)$	<i>cell value</i>
$\xi : \text{oport}(\eta)$	output port
$\xi : \text{iport}(\eta, z_s, z_e)$	input port
$\text{bound}(\eta, \xi)$	port binding
$\text{cap}(\eta, \xi)$	capability pairing
$\text{read}(x, y)$	<i>read-only variable</i>
$\text{in}(\eta, \xi)$	<i>ownership branch</i>
$\sigma \wedge \sigma$	<i>store conjunction</i>

A bit of notation

$$\frac{S \mid_{\eta} \parallel S'}{\sigma \parallel \sigma'}$$

$$\frac{\zeta \langle S \ T \rangle \mid_{\eta} \parallel T}{\sigma \parallel \sigma'}$$

are shorthand for, respectively,

$$\frac{\eta[\zeta \langle S \ T \rangle \ T] \parallel \eta[\zeta \langle S' \ T \rangle \ T]}{\sigma \parallel \sigma'}$$

$$\frac{\eta[\zeta \langle S \ T \rangle \ U] \parallel \eta[T \ U]}{\sigma \parallel \sigma'}$$

where ζ is arbitrary.

$$[\text{PAR}] \frac{\mathcal{T} \mathcal{U} \parallel \mathcal{T}' \mathcal{U}}{\sigma \parallel \sigma'} \text{ if } \frac{\mathcal{T} \parallel \mathcal{T}'}{\sigma \parallel \sigma'}$$

$$[\text{COMP}] \frac{\eta[\mathcal{T}] \parallel \eta[\mathcal{T}']}{\sigma \parallel \sigma'} \text{ if } \frac{\mathcal{T} \parallel \mathcal{T}'}{\sigma \parallel \sigma'}$$

$$[\text{EQUIV}] \frac{\mathcal{V} \parallel \mathcal{V}'}{\gamma \parallel \gamma'} \text{ if } \frac{\mathcal{U} \parallel \mathcal{U}'}{\sigma \parallel \sigma'} \text{ and } \mathcal{U} \equiv \mathcal{V}, \mathcal{U}' \equiv \mathcal{V}', \sigma \equiv \gamma, \sigma' \equiv \gamma'.$$

Sequential execution

$$[\text{SKIP}] \frac{\eta\langle \mathbf{skip} \ T \rangle}{\sigma} \parallel \frac{\eta T}{\sigma}$$

$$[\text{SEQ}] \frac{\eta\langle S \ T \rangle}{\sigma} \parallel \frac{\eta\langle S' \ T \rangle}{\sigma'} \quad \text{if} \quad \frac{S}{\sigma} \parallel \frac{S'}{\sigma'}$$

$$[\text{SEQTH}] \frac{\eta\langle (S_1 \ S_2) \ T \rangle}{\sigma} \parallel \frac{\eta\langle S_1 \ \langle S_2 \ T \rangle \rangle}{\sigma}$$

Miscellaneous rules

$$\text{[THREAD]} \frac{\zeta \langle \mathbf{thread}\{x\} \mathbf{S} \mathbf{end} \ T \rangle |_{\lambda} \parallel \zeta T \ \eta \langle \mathbf{S} \ \langle \rangle \rangle}{\sigma \models x = \perp \parallel \sigma \wedge \gamma} \quad \xi, \eta \text{ fresh,}$$

where $\gamma \equiv x = \eta \wedge \eta : \mathbf{thread} \wedge \mathbf{in}(\lambda, \eta)$

$$\text{[VAR]} \frac{\mathbf{local} \ X_1 \dots X_n \ \mathbf{in} \ \mathbf{S} \ \mathbf{end}}{\sigma} \parallel \frac{\mathbf{S}\{X_1 \rightarrow x_1, \dots, X_n \rightarrow x_n\}}{\sigma \wedge x_1 = \perp \wedge \dots \wedge x_n = \perp} \quad x_i \text{ fresh}$$

$$\text{[BINDV]} \frac{x = v}{\sigma \models x = \perp \wedge \neg \mathbf{read}(x)} \parallel \frac{\mathbf{skip}}{\sigma \wedge x = v}$$

$$\text{[IFTRUE]} \frac{\mathbf{if} \ x \ \mathbf{then} \ \mathbf{S}_1 \ \mathbf{else} \ \mathbf{S}_2 \ \mathbf{end}}{\sigma \models x = \mathbf{true}} \parallel \frac{\mathbf{S}_1}{\sigma}$$

$$\text{[NEWCAP]} \frac{\{\mathbf{newCap} \ x \ y\}}{\sigma \wedge x = \perp \wedge y = \perp} \parallel \frac{\mathbf{skip}}{\sigma \wedge x = c \wedge y = n \wedge \mathbf{cap}(c, n)} \quad c, n \text{ fresh}$$

Case statement

$$[\text{CASEM}] \frac{\text{case } x \text{ of } J \text{ then } S_1 \text{ else } S_2}{\sigma \models x = v} \parallel \frac{S_1 \theta'}{\sigma \wedge \gamma}$$

if $\text{match}_\sigma(v, J\theta') = \theta$, where $\forall y \in \text{dom}(\theta)$ y fresh and

$$\begin{aligned} \theta' &= \{Y_1 \rightarrow y_1, \dots, Y_m \rightarrow y_m\} & \text{bv}(J) &= Y_1 \mid \dots \mid Y_m \\ \text{dom}(\theta) &= \{y_1, \dots, y_m\} & \gamma &\equiv \bigwedge_{y \in \text{dom}(\theta)} y = \theta(y) \end{aligned}$$

$$[\text{CASEU}] \frac{\text{case } x \text{ of } J \text{ then } S_1 \text{ else } S_2}{\sigma \models x = v} \parallel \frac{S_2}{\sigma}$$

if $\text{match}_\sigma(v, J\theta') = \perp$, where

$$\theta' = \{Y_1 \rightarrow y_1, \dots, Y_m \rightarrow y_m\} \quad \text{bv}(J) = Y_1 \mid \dots \mid Y_m$$

$$[\text{PNEW}] \frac{\text{proc}\{x X_1 \dots X_n\} S \text{ end}}{\sigma \models x = \perp} \parallel \frac{\text{skip}}{\sigma \wedge x = \eta \wedge \text{cap}(\eta, \xi) \wedge \xi : P}$$

where η, ξ fresh, and $P = \text{proc}\{\$ X_1 \dots X_n\} S$

$$[\text{PCALL}] \frac{\{x X_1 \dots X_n\}}{\sigma \models x = \xi \wedge \xi : P} \parallel \frac{S\{X_1 \rightarrow x_1, \dots, X_n \rightarrow x_n\}}{\sigma}$$

where $P = \text{proc}\{\$ X_1 \dots X_n\} S$

$$[\text{INPORT}] \frac{\{\text{inPort } y\} |_{\eta} \parallel \text{skip}}{\sigma \models y = \perp} \parallel \frac{\text{skip}}{\sigma \wedge \gamma} \quad z, \xi, \zeta \text{ fresh}$$

where $\gamma \equiv y = \zeta \wedge \text{cap}(\zeta, \xi) \wedge \xi : \text{oport}(\eta, z, z) \wedge z = \perp$

$$[\text{OUTPORT}] \frac{\{\text{outPort } x\} |_{\eta} \parallel \text{skip}}{\sigma \models x = \perp} \parallel \frac{\text{skip}}{\sigma \wedge x = \zeta \wedge \text{cap}(\zeta, \xi) \wedge \xi : \text{oport}(\eta)} \quad \zeta, \xi \text{ fresh}$$

$$[\text{BINDP}] \frac{\{\mathbf{bind} \ x \ y\} \mid_{\lambda} \quad \parallel \quad \mathbf{skip}}{\sigma \models \phi \wedge \phi' \wedge \phi'' \quad \parallel \quad \sigma \wedge \text{bound}(\xi', \eta')}$$

where

$$\phi \equiv \mathbf{x} = \xi \wedge \text{cap}(\xi, \xi') \wedge \xi' : \text{oport}(\zeta) \wedge \neg \text{bound}(\xi')$$

$$\phi' \equiv \mathbf{y} = \eta \wedge \text{cap}(\eta, \eta') \wedge \eta' : \text{iport}(\zeta', \mathbf{z}_s, \mathbf{z}_e)$$

$$\phi'' \equiv \text{separate}(\zeta, \zeta') \implies \lambda \sqsupseteq \zeta \wedge \lambda \sqsupseteq \zeta'$$

$$[\text{UNBIND}] \frac{\{\mathbf{unbind} \ x \ y\} \mid_{\lambda} \quad \parallel \quad \mathbf{skip}}{\sigma \wedge \text{bound}(\xi', \eta') \models \phi \wedge \phi' \quad \parallel \quad \sigma}$$

where

$$\phi \equiv \mathbf{x} = \xi \wedge \text{cap}(\xi, \xi') \wedge \xi' : \text{oport}(\eta) \wedge \mathbf{y} = \eta \wedge \text{cap}(\eta, \eta') \wedge \eta' : \text{iport}(\eta', \mathbf{z}_s, \mathbf{z}_e)$$

$$\phi' \equiv \text{separate}(\zeta, \zeta') \implies \lambda \sqsupseteq \zeta \wedge \lambda \sqsupseteq \zeta'$$

$$[\text{SEND}] \frac{\{\mathbf{send } p \ x\} |_{\zeta}}{\sigma \wedge \eta : \text{iport}(\zeta', z_s, z_e) \models \phi} \parallel \frac{\mathbf{skip}}{\sigma \wedge \eta : \text{iport}(\zeta', z'_s, z_e) \wedge y = v}$$

where

$$\phi \equiv p = \xi \wedge \xi : \text{oport}(\zeta) \wedge \text{bound}(\xi, \eta) \wedge x = v \wedge \text{strict}_{\sigma}(v) \wedge z_s = y \mid z'_s \wedge y = \perp$$

$$[\text{RECEIVE}] \frac{\{\mathbf{receive } p \ x\} |_{\zeta}}{\sigma \wedge \eta : \text{iport}(\zeta, z_s, z_e) \models \phi} \parallel \frac{x = z}{\sigma \wedge \gamma \wedge \text{read}(x)}$$

where z, z'_e fresh

$$\begin{aligned} \phi &\equiv p = \eta \wedge x = \perp \wedge z_e = \perp \\ \gamma &\equiv z_e = z \mid z'_e \wedge z = \perp \wedge z'_e = \perp \wedge \eta : \text{iport}(\zeta, z_s, z'_e) \end{aligned}$$

Component abstraction

$$[\text{CHNEW}] \frac{\zeta \langle \text{hardK}\{y \ z \ i(x_1 \dots x_n) \ o(y_1 \dots y_m)\} \ \mathbf{S} \ \text{end} \ T \rangle |_\lambda \quad \sigma \models \phi}{\zeta T \ \xi [\zeta \langle \mathbf{S} \ \langle \rangle \rangle] \quad \sigma \wedge \gamma}}$$

where ξ, ζ fresh and

$$\begin{aligned} U &= \{x_1, \dots, x_n\} & I &= \{1, \dots, n\} \\ V &= \{y_1, \dots, y_m\} & J &= \{1, \dots, m\} \\ W &= U \cup V \cup \{y, z\} & \forall k \in I \cup J, \xi_k, \eta_k, z_k & \text{ fresh} \\ \phi &\equiv \forall x \in v(\mathbf{S}, \sigma) \setminus W, \text{strict}_\sigma(x) \wedge \forall x \in W, x = \perp \end{aligned}$$

$$\begin{aligned} \gamma &\equiv \bigwedge_{i \in I} x_i = \xi_i \wedge \xi_i : \text{iport}(\eta, z_i, z_i) \wedge z_i = \perp \wedge y = \xi \wedge \xi : \text{hardK} \\ &\quad \bigwedge_{j \in J} y_j = \eta_j \wedge \eta_j : \text{oport}(\eta) \wedge z = \zeta \wedge \zeta : \text{thread} \\ &\quad \wedge \text{in}(\lambda, \xi) \wedge \text{in}(\xi, \zeta) \end{aligned}$$

Packing and unpacking

$$[\text{PACK}] \frac{\eta\langle\{\mathbf{pack} \ x \ y\} \ T\rangle \ \zeta[T]}{\sigma \wedge \text{in}(\lambda, \zeta) \wedge \xi : \text{comp} \models \phi} \parallel \frac{\eta T}{\sigma \wedge \xi : \text{comp}(\text{packed}) \wedge y = \text{pack}(\zeta[T], \sigma)}$$

where

$$\phi = \text{unbound}(\zeta[T], \sigma) \wedge x = \xi$$

$$[\text{UNPACK}] \frac{\zeta\langle\{\mathbf{unpack} \ y \ x\} \ T\rangle \mid_{\lambda}}{\sigma \models y = \text{pack}(T, \sigma')} \parallel \frac{\zeta T \ \mathcal{T}\theta}{\sigma \wedge \gamma}$$

where

$$\gamma \equiv \bigwedge_{\xi \in \text{tcn}(T\theta)} \text{in}(\lambda, \xi) \wedge \sigma' \theta \wedge x = \text{namelist}(\theta)$$

and where for all $x \in v(\sigma')$, $\theta(x) = y$, with y fresh, for all $\xi \in n(\sigma')$, $\theta(\xi) = \eta$, with η fresh.

- 1 Objectives
- 2 Design goals
- 3 Related work & contributions
- 4 Oz/Ks abstract syntax
- 5 Oz/Ks operational semantics
- 6 Programming examples**
- 7 Conclusion

A perfect sandbox

Receive an untrusted component on a port, and unpack it in a hard kell with a monitoring procedure `Proc` that reports its results on port `Q`.

```
{receive P X}
hardK{K T i() o(Q)}
  local T' Y in
    thread{T'} {unpack X Y} end
    {Proc Q}          % some monitoring procedure
  end
end
```

A pure module in Oz/Ks

A procedure `ProcMod` that creates a component named `Mod` that comprises only a pair of procedures and make them known on an output port `Q` :

```
proc{ProcMod Mod Q}
  softK{Mod T i() o(Q)}
  local P1 P2 Y P in
    proc{P1 ...} S1 end
    proc{P2 ...} Sn end
    proc{P X} {send Q X} {P X} end
    Y = r(proc1: P1 proc2: P2)
    {P Y}
  end
end
end
```

A pure module

Using `Mod` : instantiate it, pack it, and send it on port `P` for future use :

```
local X Z in
  {ProcMod X}
  {pack X Z}
  {send P Z}
end
```

On receiving `Mod` on port `P'` , unpack it, get the set of procedures it contains on port `Q'` , and use it :

```
local X Y Z in
  {receive P' X}
  {unpack X Z}
  case Z of pair(Q R) | Z'
    then {bind Q' R} {receive Q' Y} {Y.procl ...}
    else raise LoadFailure end
  end
end
```

Extending logical variables across soft membranes

The effect of passing an unbound logical variable X across soft membranes can be achieved as follows.

Emitter code:

```
local In in
{inPort In}
{send P Z}
%% Z is value v{cont(In) / X}
{receive In X}
end
```

Receiver code:

```
local X Out In' in
{receive Z}
%% handle Z = v
%% as if v was v{X'/X}
%% at some point
%% extract port name In
%% i.e. In' = In
%% setting X' = w
X' = ...
{outPort Out}
{bind Out In'}
{send Out X'}
end
```

A simple interceptor

A simple interceptor procedure to process and forward messages between ports. Also an example of recursive definition in OZ.

```
proc{Relay In Out Proc}
  local X Y in
    {receive In X}
    {Proc X Y}
    {send Out Y}
    {Relay In Out Proc}
  end
end
```

A component template

A component template *à la Fractal* can be constructed as a record comprising a `port-names` field, a `membrane` field, a `contents` field, and an `attribute-names` field.

```
local InPortNames OutPortNames PubAttNames
      PrivAttNames Contents Membrane
in
  PubAttNames = [att1 ... attm]
  PrivAttNames = [att1 ... attmm]
  InPortNames = [iport1 ... iportn]
  OutPortNames = [oport1 ... oportq]
  Contents = [Templ ... Tempt]
  Membrane = proc{$ InPorts OutPorts State Subs Self} ... end
  Temp = tmp(memb: Membrane
             pub:PubAttNames priv:PrivAttNames
             iports: InPortNames oports: OutPortNames
             cont: Contents init:Init)
end
```

A generic component factory

A component factory is a procedure that takes a component template as input and returns a new component conforming to the template.

```
proc{NewComp Arg}
  local State P Subs Q T in
    case Arg of p(Temp Comp) then
      proc{P A} Y in {NewCell Y A} end
      State = {MakeRecord s Temp.pub#Temp.priv}
              {Record.forAll State P}
      proc{Q A} Y in p(A Y) end
      Subs = {MakeRecord sc {List.map Temp.cont Q}}
      In = {MakeRecord i Temp.iports}
      Out = {MakeRecord o Temp.oports}
      Comp = hardK{$ T In Out}
              {Temp.memb In Out State Subs Comp}
            end
          end
        end
      end
    end
  end
```

Outline

- 1 Objectives
- 2 Design goals
- 3 Related work & contributions
- 4 Oz/Ks abstract syntax
- 5 Oz/Ks operational semantics
- 6 Programming examples
- 7 Conclusion**

- Oz/Ks : attempt at combining multiple programming paradigms, as supported by MOZART, and component-based structures inspired by FRACTAL and the Kell calculus.
- Kell and port abstractions provide a programmatic way to build FRACTAL-like structures.
- Packed values provide a direct representation of component packages.
- Programming evolving structures possible through a combination of higher-order programming, port binding/unbinding, and packing/unpacking.

OZ/Ks is ongoing work. Much remains to investigate :

- Decompose kell and port constructs into finer grain ones :
 - membranes, naming, binding across named membranes, strictness as explicit check
- Relax strictness : components communicating through logical variables
- Inspecting, manipulating and analyzing packed structures
- Assert and prove security properties of the design
- Failure semantics
- Explain MOZART distributed implementation in OZ/Ks
 - distributed unification, mobile objects, failure semantics
- Supporting recoverable, compensatable and atomic actions