κ-VM : a virtual machine for a domain-oriented calculus

Marc Lacoste (FT R&D)
Florence Germain (FT R&D)

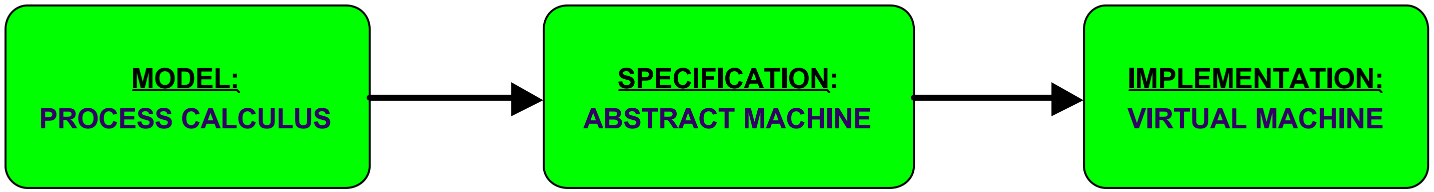
MIKADO Kick-Off Meeting

Introduction

➔ Domain-oriented programming ?

➔ Challenge : can large-scale distributed systems be programmed effectively using the M-calculus ?

➔ French MARVEL Project results :



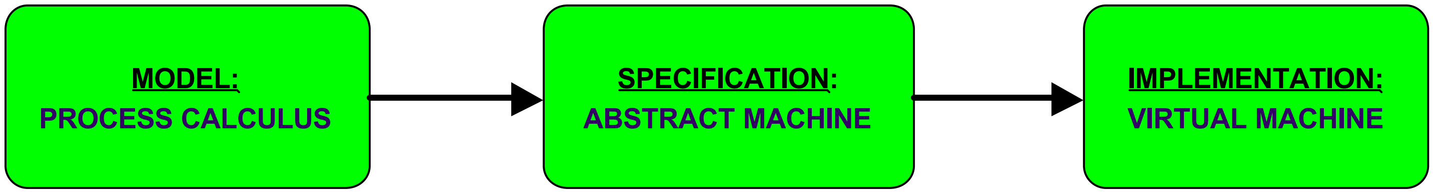
Introduction

> **Definition**: *"A distributed system may be viewed as a partitioned system. Subsystems may be grouped in different, possibly overlapping sets, generally under the control of a single authority. We call such sets of subsystems* ***domains."*** [FMOODS00]

- **Domain-oriented programming ?**

- **Challenge : can large-scale distributed systems be programmed effectively using the M-calculus ?**

- **French MARVEL Project results :**



Introduction

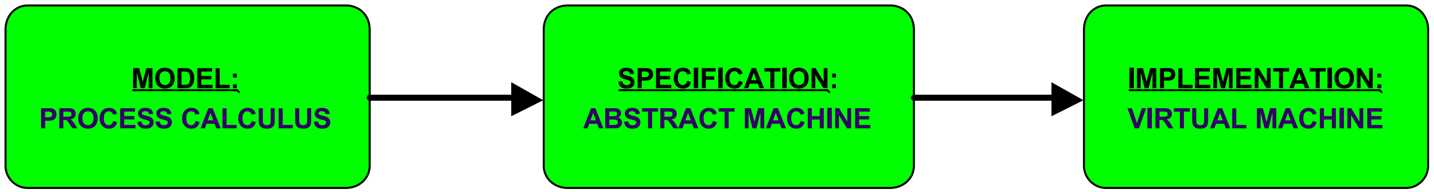
Examples of domains:

- ✓ Failure
- ✓ Naming
- ✓ Technology
- ✓ Resources
- ✓ Security (access control)
- ✓ Trust (authentication)

Definition: *"A distributed system may be viewed as a partitioned system. Subsystems may be grouped in different, possibly overlapping sets, generally under the control of a single authority. We call such sets of subsystems **domains**."* [FMOODS00]

➔ **Challenge : can large-scale distributed systems be programmed effectively using the M-calculus ?**

➔ **French MARVEL Project results :**



Outline of the talk

The $\mathcal{K}$-calculus : a simple model to program with domains

A formal specification : the $\mathcal{K}$ abstract machine

The $\mathcal{K}$-VM implementation

Distributed extension

Conclusion and future work

The κ-calculus

- The κ-calculus : a directly implementable subset of the M-calculus

  - Domains are not named
  - P[Q] instead of a(P)[Q]

  - No domains inside controllers
  - no (P[Q] | R) [S]

  - Restricted application
  - (P Q) is restricted to the case where both P and Q reduce to values

  - No dynamic binding features
  - Fewer reduction rules

The κ-calculus

- Functional core : direct embedding of the λ - calculus

P ::=		Process
	lambda x . P	Lambda-abstraction
|	(P P)	Application
|	x	Variable

- Semantics :

The κ-calculus

- **Functional core : direct embedding of the λ - calculus**

P **::=**		**Process**
	lambda *x* . *P*	**Lambda-abstraction**
|	**(*P* *P*)**	**Application**
|	***x***	**Variable**

- **Semantics :** $(l\ x . P\ \ V) \rightarrow \{V / x\} P$

The κ-calculus

➔ **Imperative and concurrent features (π - calculus, blue - calculus) :**

P ::= …	**Process**
nil	**Inert process**
| **new** *r* *P*	**Creation of new names**
| *Q* | *R*	**Parallel composition**
| **if** *u* = *v* **then** *P* **else** *Q*	**Conditional testing**
| *r*	**Reference**
| **def** *r* = *P*	**Definition**
u ,v ::=	**Names**
r	**Reference**
| *x*	**Variable**

➔ **Semantics :**

The κ-calculus

➔ Imperative and concurrent features (π - calculus, blue - calculus) :

P ::= …	Process
nil	Inert process
| **new** *r* *P*	Creation of new names
| *Q* | *R*	Parallel composition
| **if** *u* = *v* **then** *P* **else** *Q*	Conditional testing
| *r*	Reference
| **def** *r* = *P*	Definition
u ,v ::=	Names
r	Reference
| *x*	Variable

➔ Semantics :

$$\langle r = P \rangle \mid (r\,V) \rightarrow \langle r = P \rangle \mid PV$$

The κ-calculus

- **Synchronization primitives (join-calculus) :**

P ::= …	**Process**
| D	**Definition**
D,D' ::=	**Definitions**
def $J = P$	**Simple definition**
| D ; D'	**Multiple definitions**
J , J' ::=	**Join-pattern**
J | J'	**Synchronization**
| r	**Reference**

- **Semantics (local communication):**

The κ-calculus

➔ Synchronization primitives (join-calculus) :

P ::= …	Process
| *D*	Definition
D,D' ::=	Definitions
def *J* = *P*	Simple definition
| *D* ; *D'*	Multiple definitions
J , *J'* ::=	Join-pattern
J | *J'*	Synchronization
| *r*	Reference

➔ Semantics (local communication):

$$\frac{\langle D \rangle = \langle r_1 \mid \cdots \mid r_n = P \rangle}{\langle D \rangle \mid (r_1\, V_1) \mid \cdots \mid (r_n\, V_n) \rightarrow \langle D \rangle \mid P V_1 \cdots V_n}$$

The κ-calculus

- Distinguishing feature : domain primitives : (κ-calcul)

P ::= …	Process
| **dom** *d* **do** *P* **controls** *Q* **end**	Domain
| **pass** *V*	Passivation

- Semantics :
  - Remote communication:
  - Passivation:

More details …

France Télécom R&D

The κ-calculus

➔ **Distinguishing feature : domain primitives : (κ-calcul)**

P **::= …**	**Process**
| **dom *d* do *P***	**Domain**
controls *Q*	
end	
| pass *V*	**Passivation**

$rV \mid P[Q]$	$\rightarrow$	$(rV \mid P)[Q]$	$r \in \mathbf{dn}(P)$	**[in]**
$(rV \mid P)[Q]$	$\rightarrow$	$P[rV \mid Q]$	$r \in \mathbf{dn}(Q)$	**[intrude]**
$rV \mid P[Q]$	$\rightarrow$	$((i\ l \cdot rV) \mid P)[Q]$	$r \in \mathbf{dn}(Q)$	**[filter.in]**
$(rV \mid P)[Q]$	$\rightarrow$	$rV \mid [Q]$	$r \notin \mathbf{dn}(P[Q])$	**[out]**
$P[rV \mid Q]$	$\rightarrow$	$(rV \mid P)[Q]$	$r \in \mathbf{dn}(P)$	**[extrude]**
$P[rV \mid Q]$	$\rightarrow$	$((o\ l \cdot rV) \mid P)[Q]$	$r \notin \mathbf{dn}(P[Q])$	**[filter.out]**

➔ **Seman**

- **Re**
- **Passivation:**

More details …

The κ-calculus

- **Distinguishing feature : domain primitives : (κ-calcul)**

***P* ::= …**	**Process**
| dom *d* do *P*	**Domain**
controls *Q*	
end	
| pass *V*	**Passivation**

- **Semantics :**
  - **Remote communication:**

$rV \mid P[Q]$	$\rightarrow$	$(rV \mid P)[Q]$	$r \in \mathrm{dn}(P)$	[in]
$(rV \mid P)[Q]$	$\rightarrow$	$P[rV \mid Q]$	$r \in \mathrm{dn}(Q)$	[intrude]
$rV \mid P[Q]$	$\rightarrow$	$((i\ l \cdot rV) \mid P)[Q]$	$r \in \mathrm{dn}(Q)$	[filter.in]
$(rV \mid P)[Q]$	$\rightarrow$	$rV \mid [Q]$	$r \notin \mathrm{dn}(P[Q])$	[out]
$P[rV \mid Q]$	$\rightarrow$	$(rV \mid P)[Q]$	$r \in \mathrm{dn}(P)$	[extrude]
$P[rV \mid Q]$	$\rightarrow$	$((o\ l \cdot rV) \mid P)[Q]$	$r \notin \mathrm{dn}(P[Q])$	[filter.ou t]

  - **Passivation:**

More details …

The κ-calculus

➔ **Distinguishing feature : domain primitives : (κ-calcul)**

P **::= …**	**Process**
| dom *d* do *P*	**Domain**
controls *Q*	
end	
| pass *V*	**Passivation**

➔ **Semantics :**

- **Remote communication:**

$rV \mid P[Q]$	$\rightarrow$	$(rV \mid P)[Q]$	$r \in \mathrm{dn}(P)$	[in]
$(rV \mid P)[Q]$	$\rightarrow$	$P[rV \mid Q]$	$r \in \mathrm{dn}(Q)$	[intrude]
$rV \mid P[Q]$	$\rightarrow$	$((i\ l \cdot rV) \mid P)[Q]$	$r \in \mathrm{dn}(Q)$	[filter.in]
$(rV \mid P)[Q]$	$\rightarrow$	$rV \mid [Q]$	$r \notin \mathrm{dn}(P[Q])$	[out]
$P[rV \mid Q]$	$\rightarrow$	$(rV \mid P)[Q]$	$r \in \mathrm{dn}(P)$	[extrude]
$P[rV \mid Q]$	$\rightarrow$	$((o\ l \cdot rV) \mid P)[Q]$	$r \notin \mathrm{dn}(P[Q])$	[filter.ou t]

- **Passivation:**

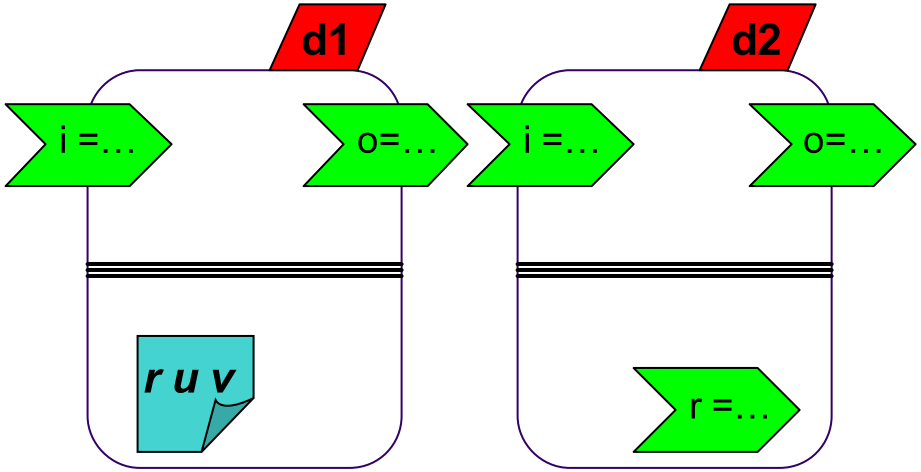
$$(P \mid \mathrm{pass}\ \ V)[Q] \rightarrow V(l \cdot P)(l \cdot Q)$$

More details …

The κ-calculus

➔ **Example:**

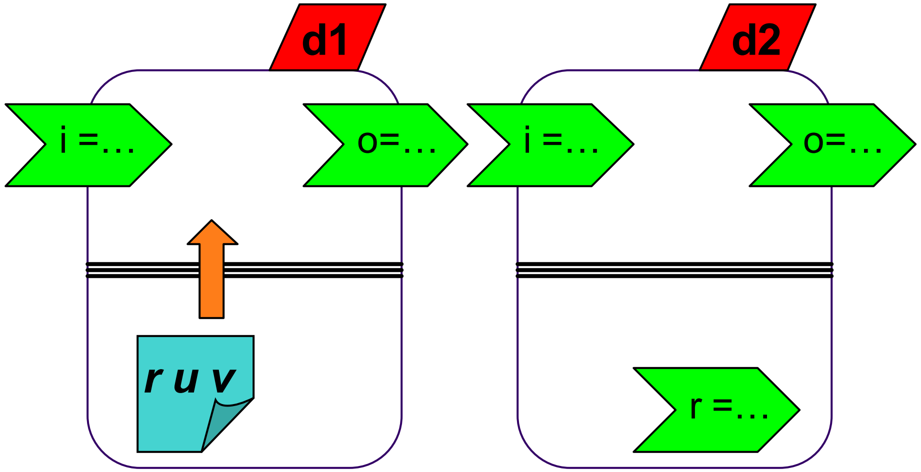
```
new u,v
dom d1 do
  def i = lambda x . x ()
  def o = lambda x. x ()
controls
  ( r  u v )
end
|
dom d2 do
   def i = lambda x . x ()
   def o = lambda x. x ()
controls
   def r = lambda x. lambda y . (y x)
end
```



The κ-calculus

➔ Example:

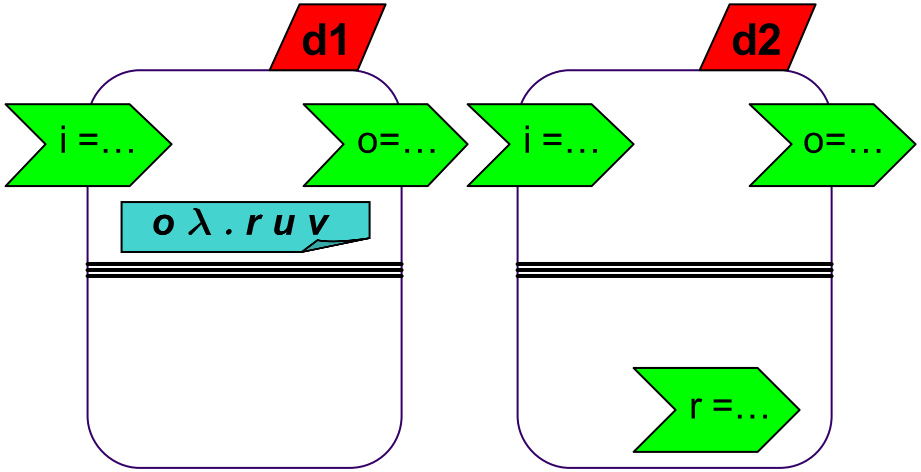
```
new u,v
dom d1 do
  def i = lambda x . x ()
  def o = lambda x. x ()
controls
  ( r  u v )
end
|
dom d2 do
   def i = lambda x . x ()
   def o = lambda x. x ()
controls
   def r = lambda x. lambda y . (y x)
end
```



The κ-calculus

➔ Example:

```
new u,v
dom d1 do
  def i = lambda x . x ()
  def o = lambda x. x ()
controls
  ( r  u v )
end
|
dom d2 do
   def i = lambda x . x ()
   def o = lambda x. x ()
controls
   def r = lambda x. lambda y . (y x)
end
```

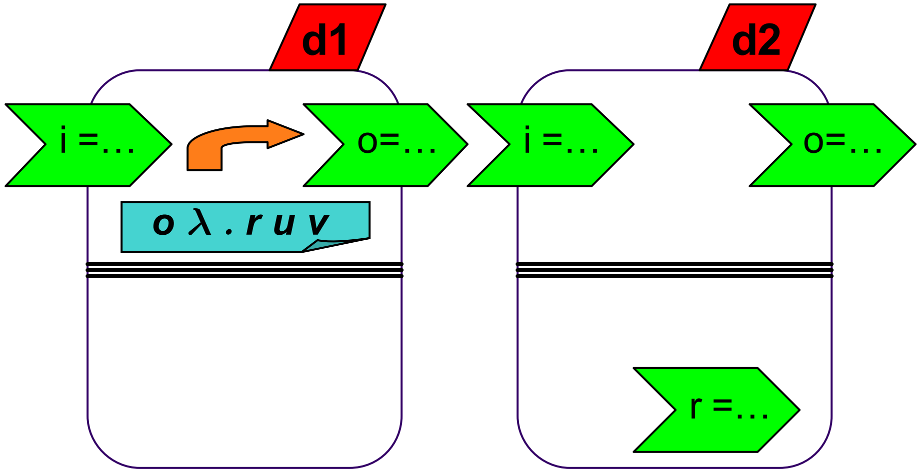


France Télécom R&D

The κ-calculus

➔ **Example:**

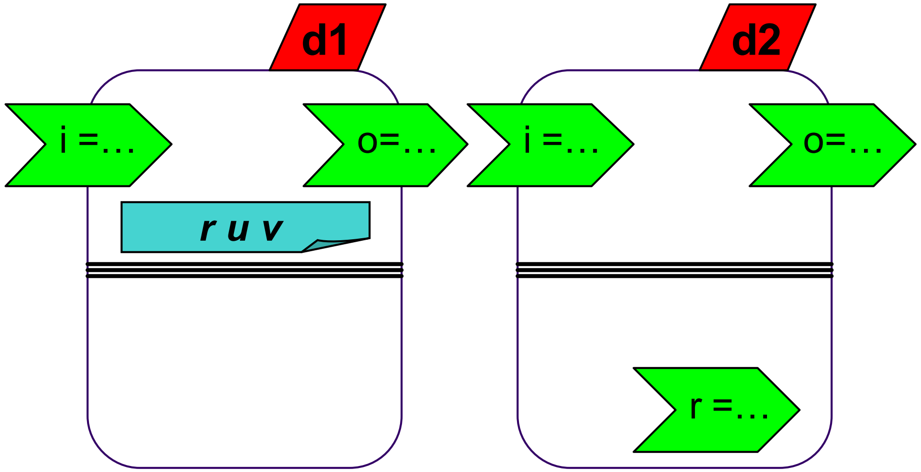
```
new u,v
dom d1 do
  def i = lambda x . x ()
  def o = lambda x. x ()
controls
  ( r  u v )
end
|
dom d2 do
   def i = lambda x . x ()
   def o = lambda x. x ()
controls
   def r = lambda x. lambda y . (y x)
end
```



The κ-calculus

- **Example:**

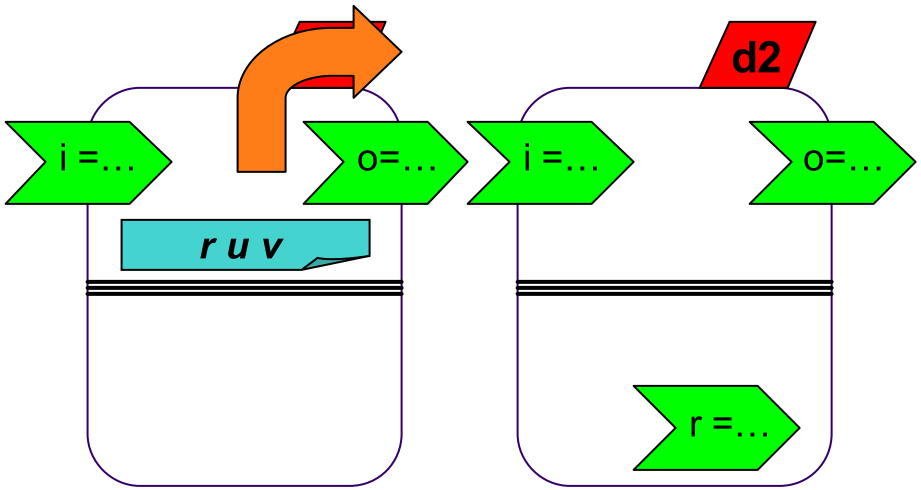
```
new u,v
dom d1 do
  def i = lambda x . x ()
  def o = lambda x. x ()
controls
  ( r  u v)
end
|
dom d2 do
   def i = lambda x . x ()
   def o = lambda x. x ()
controls
   def r = lambda x. lambda y . (y x)
end
```



The κ-calculus

- **Example:**

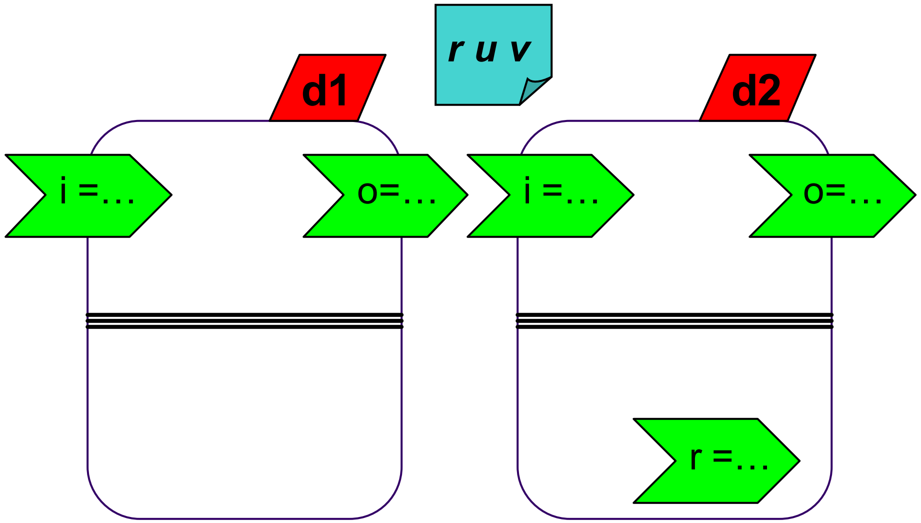
```
new u,v
dom d1 do
  def i = lambda x . x ()
  def o = lambda x. x ()
controls
  ( r  u v)
end
|
dom d2 do
   def i = lambda x . x ()
   def o = lambda x. x ()
controls
   def r = lambda x. lambda y . (y x)
end
```



The κ-calculus

➔ **Example:**

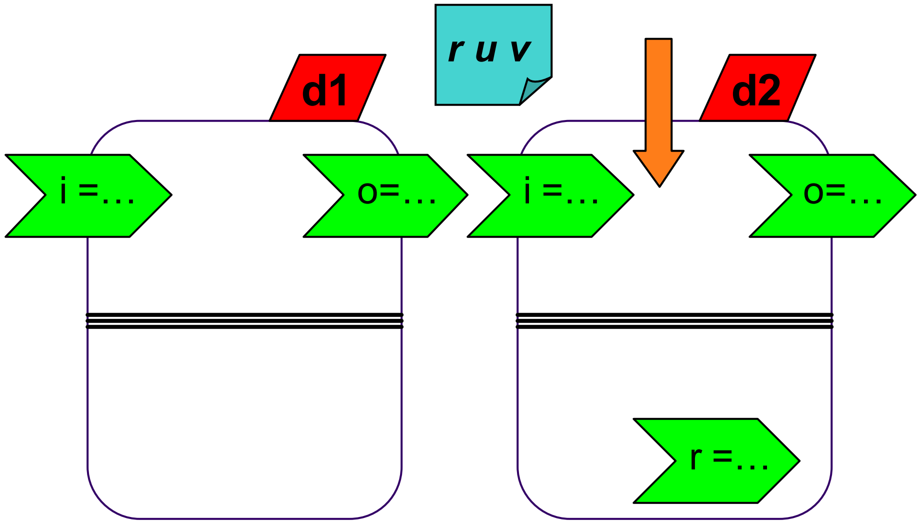
```
new u,v
dom d1 do
  def i = lambda x . x ()
  def o = lambda x. x ()
controls
  ( r  u v)
end
|
dom d2 do
   def i = lambda x . x ()
   def o = lambda x. x ()
controls
   def r = lambda x. lambda y . (y x)
end
```



The κ-calculus

➔ Example:

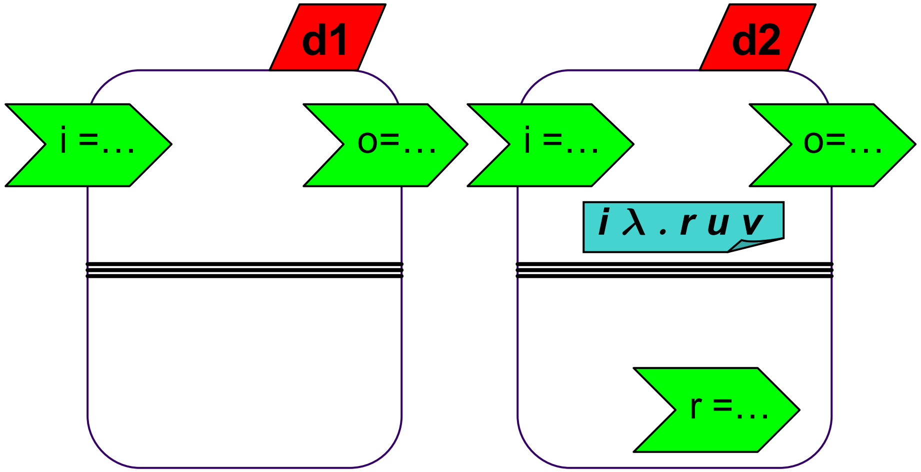
```
new u,v
dom d1 do
  def i = lambda x . x ()
  def o = lambda x. x ()
controls
  ( r  u v)
end
|
dom d2 do
   def i = lambda x . x ()
   def o = lambda x. x ()
controls
   def r = lambda x. lambda y . (y x)
end
```



The κ-calculus

➔ **Example:**

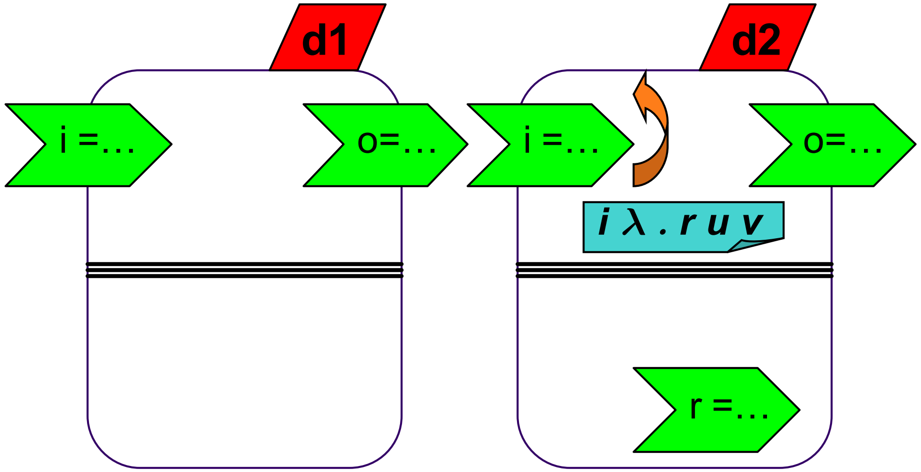
```
new u,v
dom d1 do
  def i = lambda x . x ()
  def o = lambda x. x ()
controls
  ( r  u v)
end
|
dom d2 do
   def i = lambda x . x ()
   def o = lambda x. x ()
controls
   def r = lambda x. lambda y . (y x)
end
```



The κ-calculus

➔ **Example:**

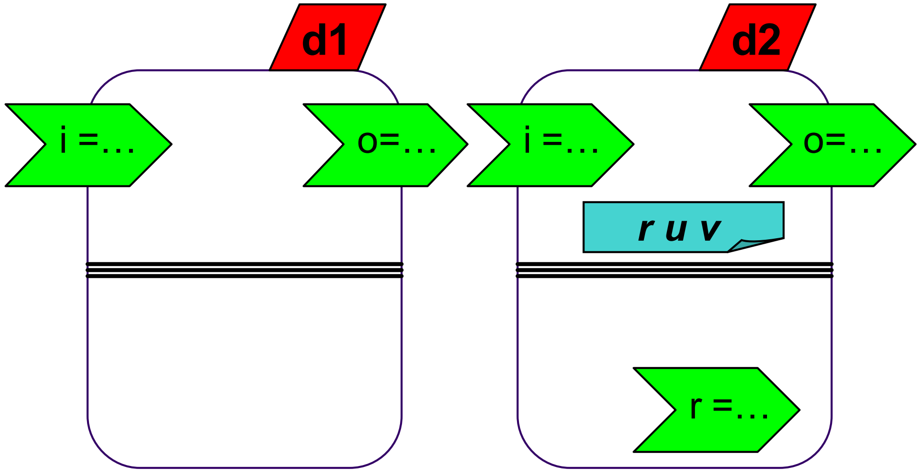
```
new u,v
dom d1 do
  def i = lambda x . x ()
  def o = lambda x. x ()
controls
  ( r  u v)
end
|
dom d2 do
   def i = lambda x . x ()
   def o = lambda x. x ()
controls
   def r = lambda x. lambda y . (y x)
end
```



The κ-calculus

➔ **Example:**

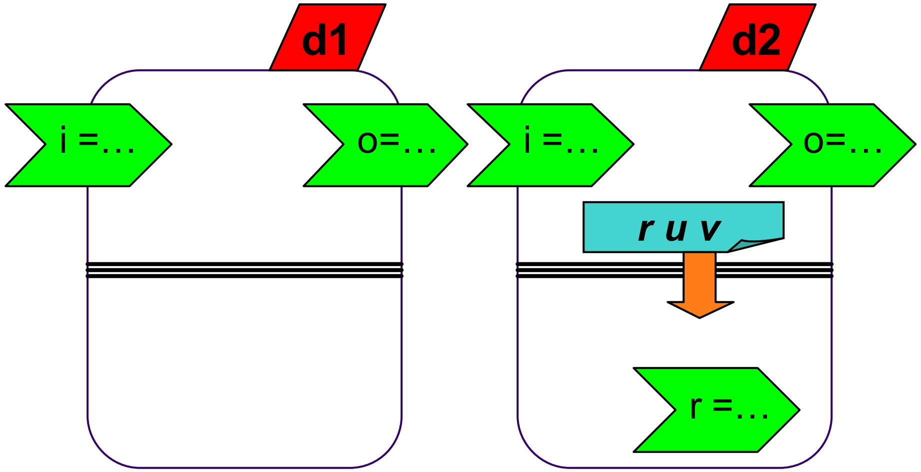
```
new u,v
dom d1 do
  def i = lambda x . x ()
  def o = lambda x. x ()
controls
  ( r  u v)
end
|
dom d2 do
   def i = lambda x . x ()
   def o = lambda x. x ()
controls
   def r = lambda x. lambda y . (y x)
end
```



The κ-calculus

➔ **Example:**

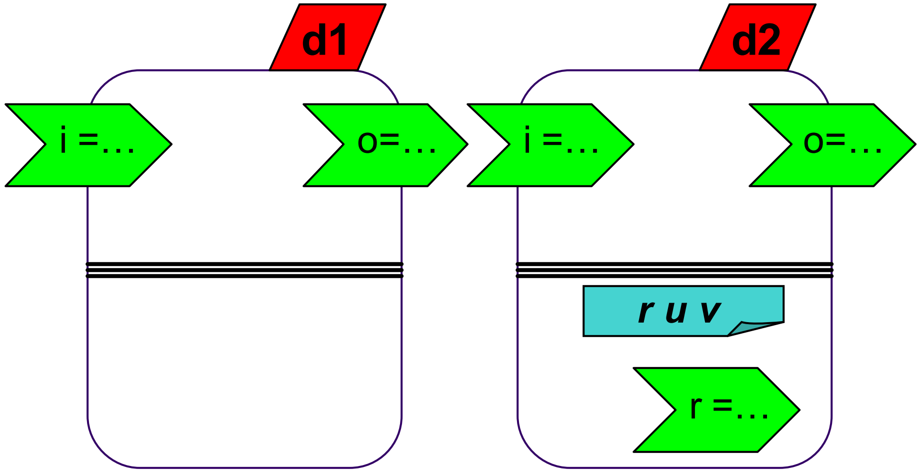
```
new u,v
dom d1 do
  def i = lambda x . x ()
  def o = lambda x. x ()
controls
  ( r  u v)
end
|
dom d2 do
   def i = lambda x . x ()
   def o = lambda x. x ()
controls
   def r = lambda x. lambda y . (y x)
end
```



The κ-calculus

➔ **Example:**

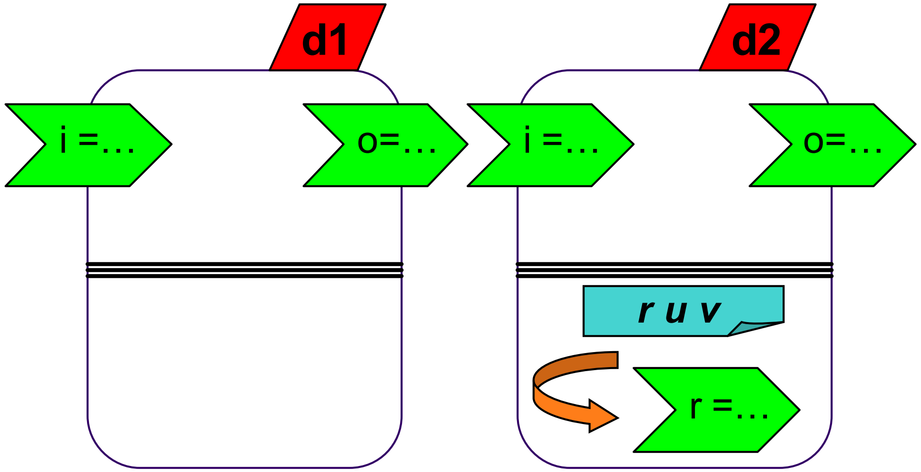
```
new u,v
dom d1 do
  def i = lambda x . x ()
  def o = lambda x. x ()
controls
  ( r  u v)
end
|
dom d2 do
   def i = lambda x . x ()
   def o = lambda x. x ()
controls
   def r = lambda x. lambda y . (y x)
end
```



The κ-calculus

➔ **Example:**

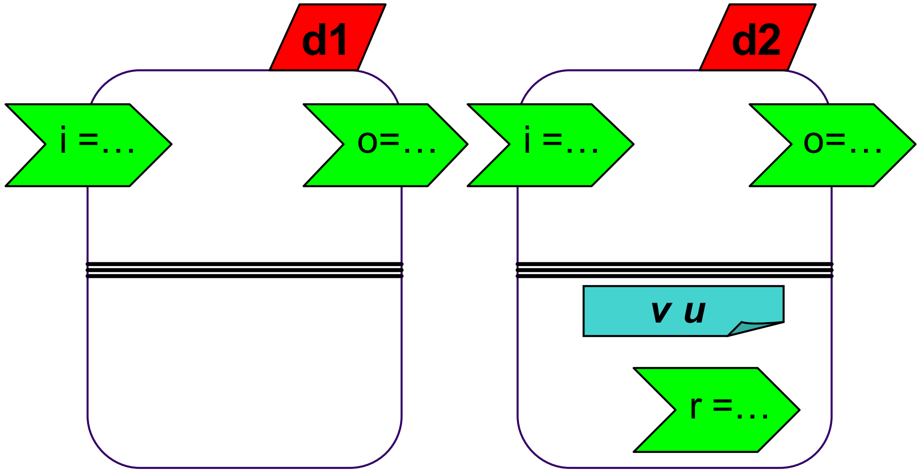
```
new u,v
dom d1 do
  def i = lambda x . x ()
  def o = lambda x. x ()
controls
  ( r  u v)
end
|
dom d2 do
   def i = lambda x . x ()
   def o = lambda x. x ()
controls
   def r = lambda x. lambda y . (y x)
end
```



The κ-calculus

➔ **Example:**

```
new u,v
dom d1 do
  def i = lambda x . x ()
  def o = lambda x. x ()
controls
  ( r  u v)
end
|
dom d2 do
   def i = lambda x . x ()
   def o = lambda x. x ()
controls
   def r = lambda x. lambda y . (y x)
end
```



Outline of the talk

The $\mathcal{K}$-calculus : a simple model to program with domains

➔A formal specification : the $\mathcal{K}$ abstract machine

The $\mathcal{K}$-VM implementation

Distributed extension

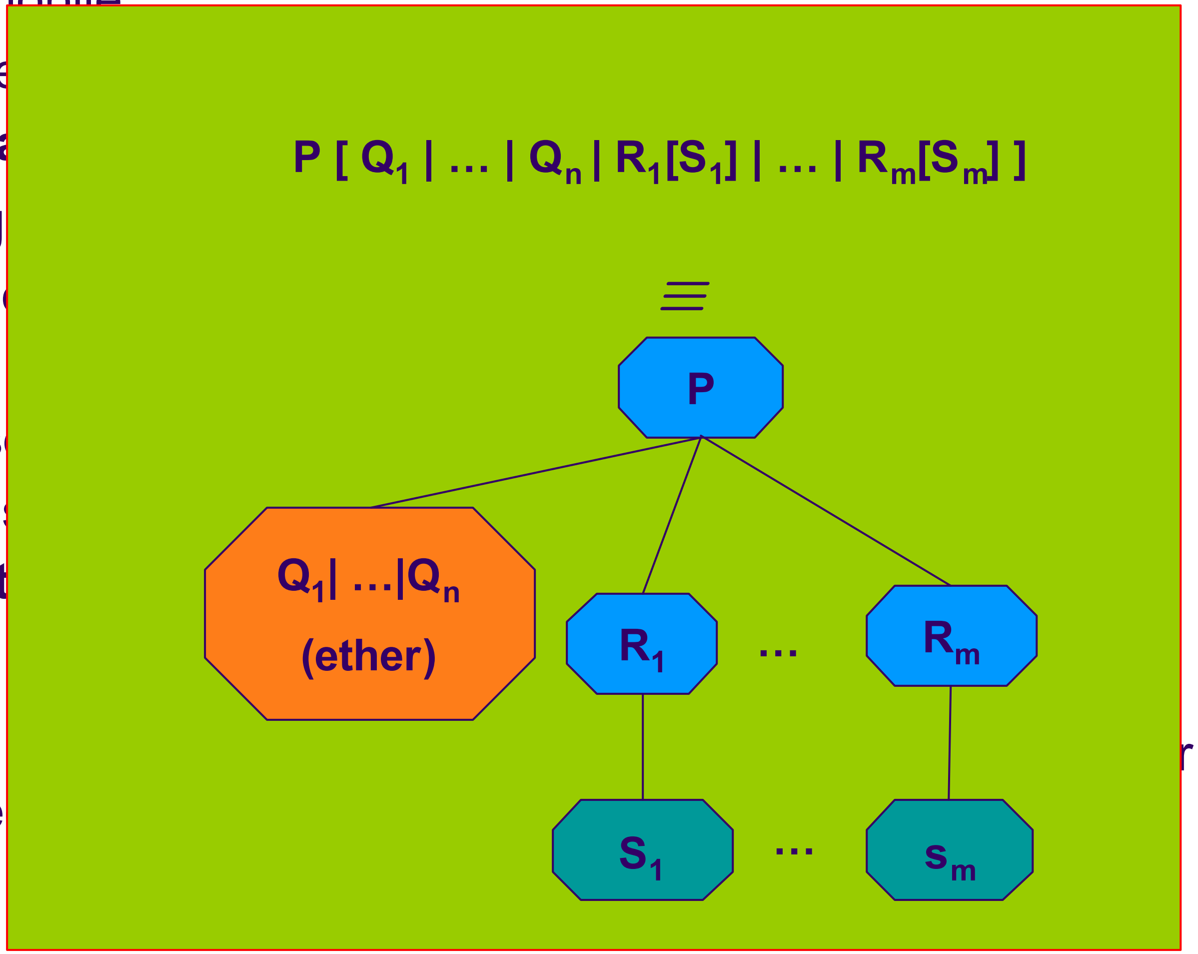
Conclusion and future work

The κ abstract machine

- **General architecture :**
  - Collection of **agents** (locations) :
    - Potentially mobile
    - Potentially residing on several VMs / physical sites
  - One or more **oracles** acting as **look-up services** for definitions
- **Structure of an agent :**
  - A language-based domain = a set of agents
    - Controller
    - Flat processes inside the content $\Rightarrow$ **internal ether**
    - Sub-domains $\Rightarrow$ links towards other agents
  - Agent = **evaluator** + **execution engine**
- **The oracle :**
  - A map { **r** $\Rightarrow$ **l** } where **l** is the location hosting a definition waiting for messages on receiver **r**

The κ abstract machine

- **General architecture :**
  - Collection of **agents** (locations) :
    - Potentially mobile
    - Potentially re
  - One or more **ora**
- **Structure of an ag**
  - A language-bas
    - Controller
    - Flat process
    - Sub-domain
  - Agent = **evaluat**
- **The oracle :**
  - A map { **r** ⇒ **l** } … r
    messages on re

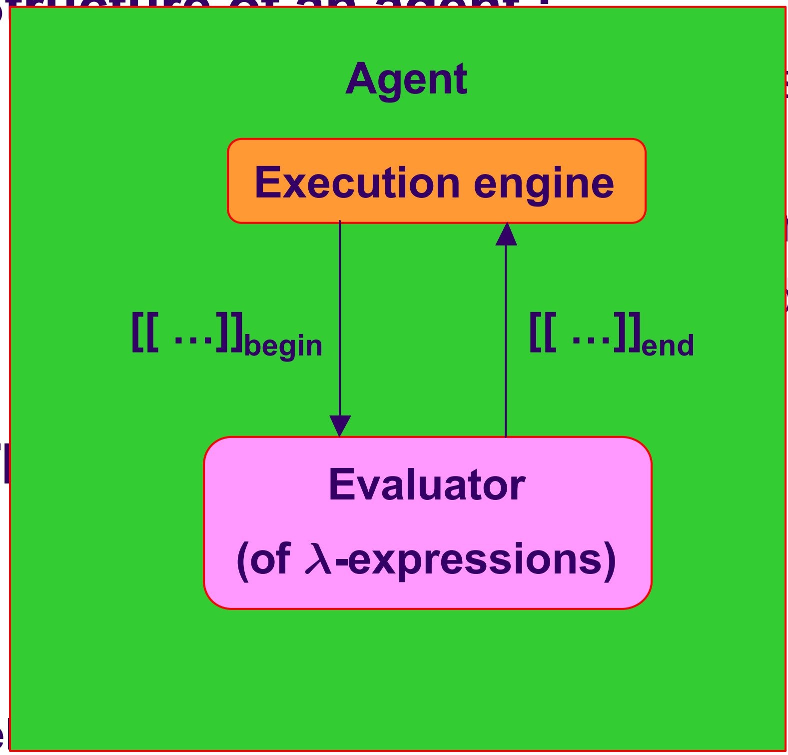


The κ abstract machine

- **General architecture :**
  - Collection of **agents** (locations) :
    - Potentially mobile
    - Potentially residing on several VMs / physical sites
  - One or more **oracles** acting as **look-up services** for definitions
- **Structure of an agent :**
  - A language-based domain = a set of agents
    - Controller
    - Flat processes inside the content $\Rightarrow$ **internal ether**
    - Sub-domains $\Rightarrow$ links towards other agents
  - Agent = **evaluator** + **execution engine**
- **The oracle :**
  - A map { **r** $\Rightarrow$ **l** } where **l** is the location hosting a definition waiting for messages on receiver **r**

The κ abstract machine

- **General architecture :**
  - Collection of **agents** (locations) :
    - Potentially mobile
    - Potentially residing on several VMs / physical sites
  - One or more **oracles** acting as **look-up services** for definitions
- **Structure of an agent :**



set of agents

ntent ⇒ **internal ether**

ls other agents

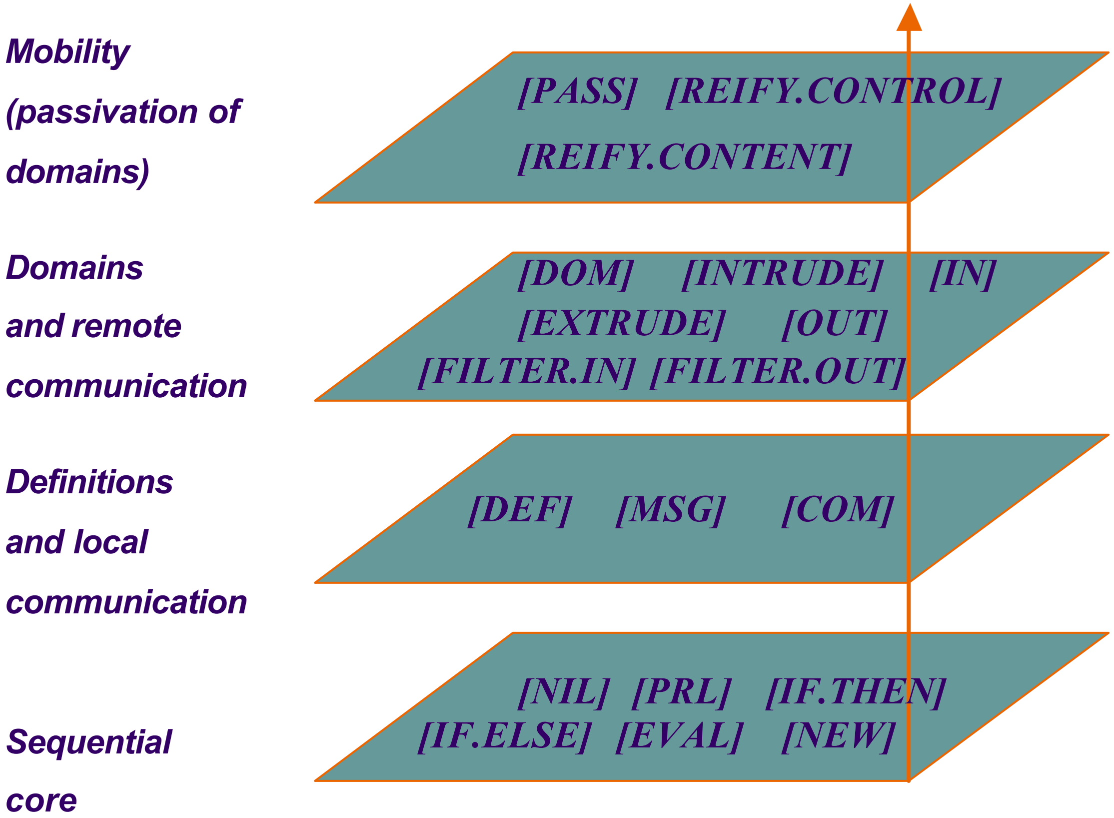
engine

- **Th**

location hosting a definition waiting for

The execution engine

➔ A layered-design:



Outline of the talk

The $\mathcal{K}$-calculus : a simple model to program with domains

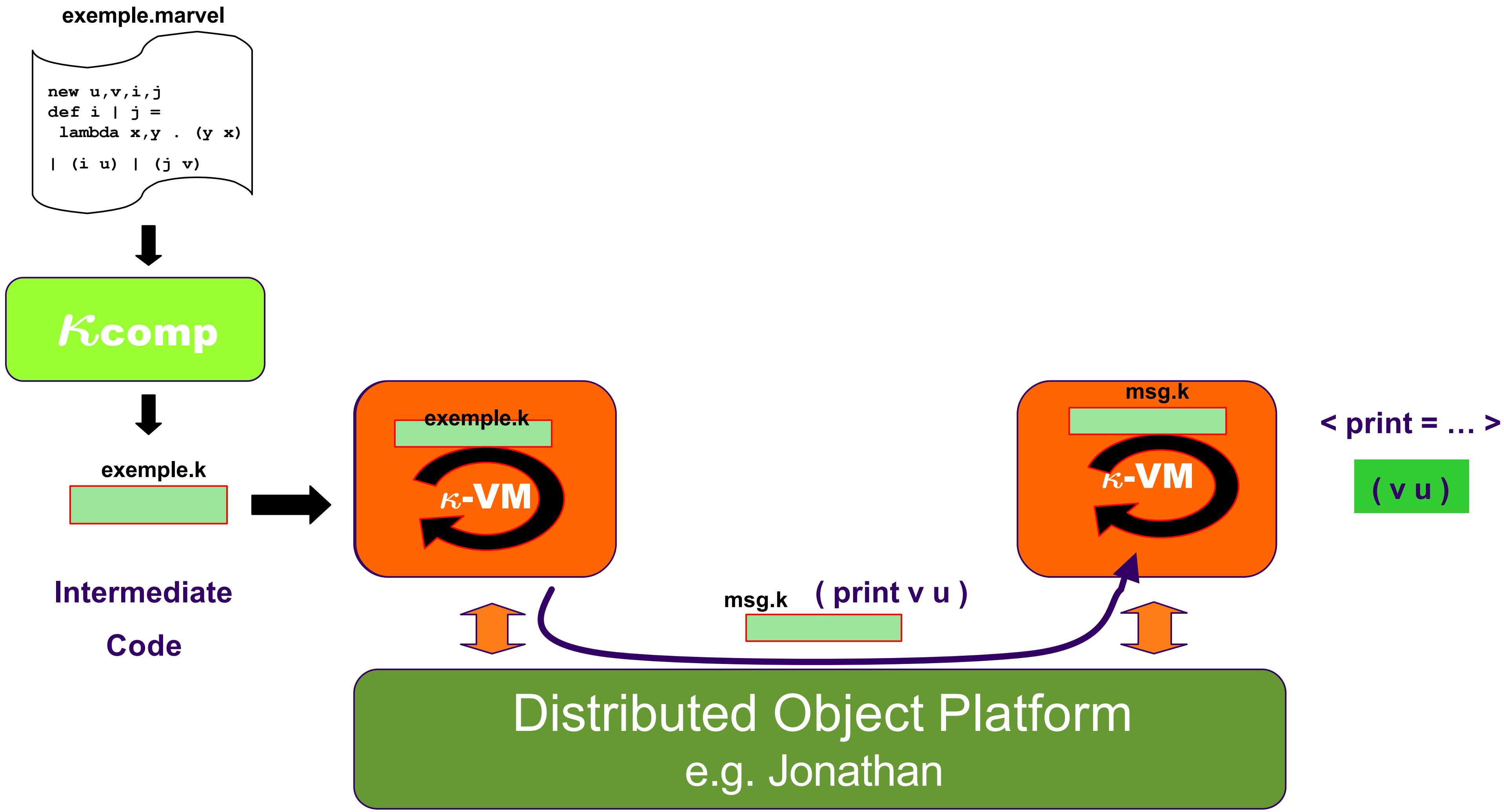
A formal specification : the $\mathcal{K}$ abstract machine

➔ The $\mathcal{K}$-VM implementation

Distributed extension

Conclusion and future work

κ-VM : an overview



The compiler

- **Structure:** (written in OCaml)
  - **Parser**
  - **Transformation of syntactic sugar constructs in core-calculus**
  - **Intermediate code generation**
- **The intermediate code:** (13 instructions)
  - **Fixed size instructions :** (PRL, APP, DOM, NIL, TEST, NEW, LAMBDA, PASS)

P | Q

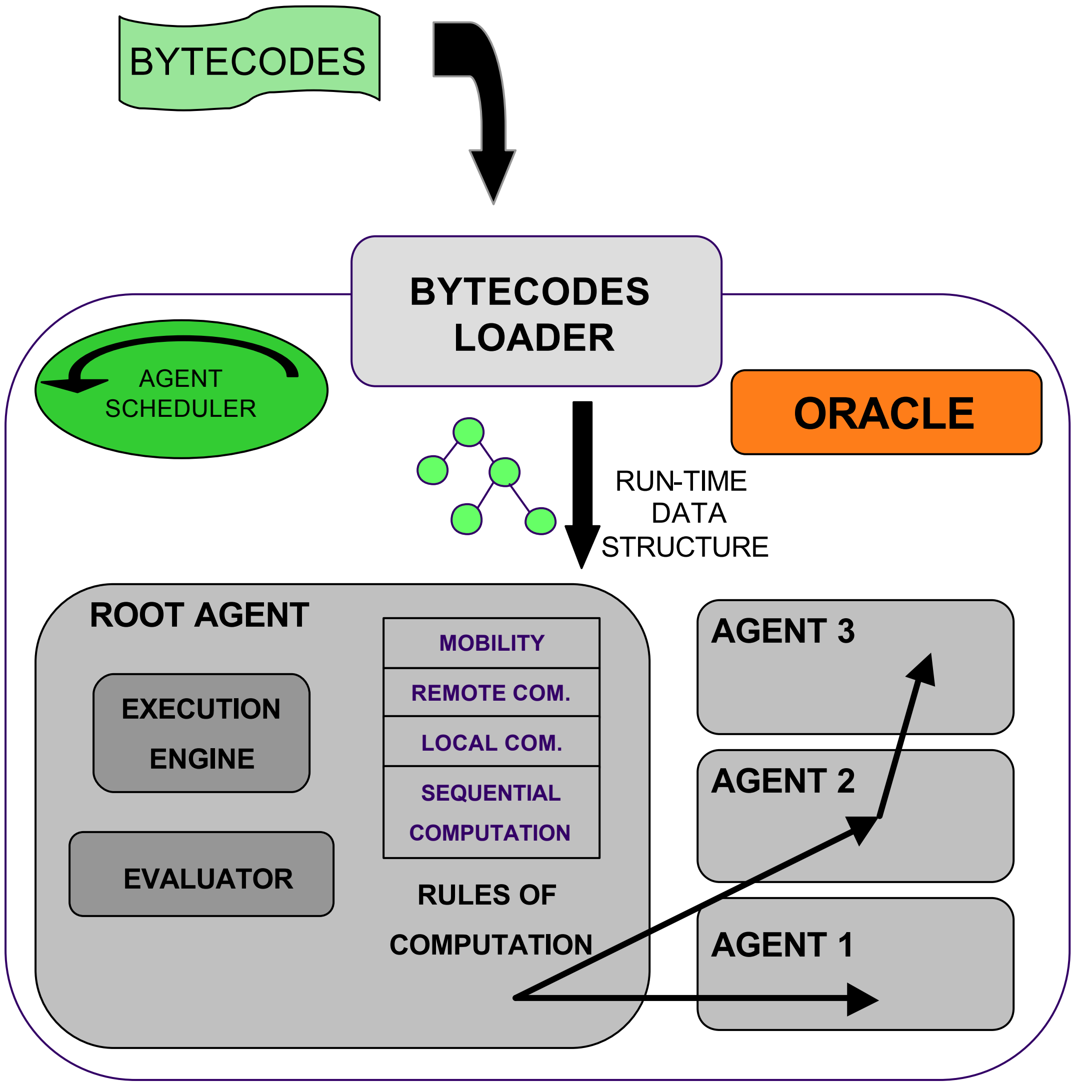
PRL	@P	@Q

  - **Variable size instructions :** (REF, VAR, DOMN, DEF, JOIN)

< r_1 | … | r_n = P >

JOIN	n	@ r_1	…	@ r_n	@ P

The virtual machine : internal structure



Outline of the talk

The κ-calculus : a simple model to program with domains

A formal specification : the κ abstract machine

The κ-VM implementation

➔ Distributed extension   More details …

Conclusion and future work

Conclusion

- **Simple example of an end-to-end formalization approach :**
  - **A simple model for programming with domains**
  - **A formal specification** of the implementation of the model by an **abstract machine**
  - **A centralized implementation** (written in Java+OCaml) **:** κ-VM
  - A **distributed extension** of the specification

Future Work

- **A truly distributed VM :**
  - **Implementation of M - calculus features, e.g., dynamic binding, type system**
  - **Distributed implementation** on top of an **ORB** like Jonathan, or a **MOM**
- **A « component - based virtual machine » :**
  - **Take into account non-functional properties such as naming, mobility, or or security**
  - **Implement the VM using the MIKADO Component Framework**