Questioning the object dogmas

Jornadas Chilenas de Computación

[illegible] September 2004

[illegible] D'Hondt
Programming Technology Lab
Computer Science Department
Faculty of Sciences
Vrije Universiteit Brussel
http://prog.vub.ac.be/~tjdhondt

reuse & improvise

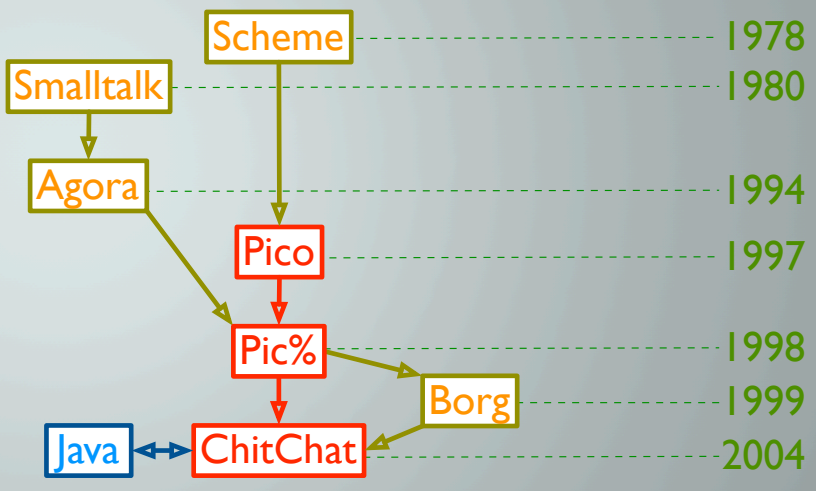
Programming Technology Lab

- **Object-oriented Software Evolution and Reuse**
- **Intentionality in AOP**
- **Domain-specific Aspect Languages**
- **Component Technology for Distributed and Embedded Systems**
- **Language Features for Ambient Intelligence**

Programming Technology Lab

- Logic Meta-programming
- A Prototype-based Language Family
- Reflection and Strong Mobility
- Reuse Contracts
- Software Classification
- Software (Co)-Evolution

Taxonomy of an experiment



Inspiration for the experiment

- Finframe in EDS Belux
- use J2EE as distributed platform
- use DSL for financial transactions
- use Java-based interpreter
- robust base
- configurable + evolvable

FSM

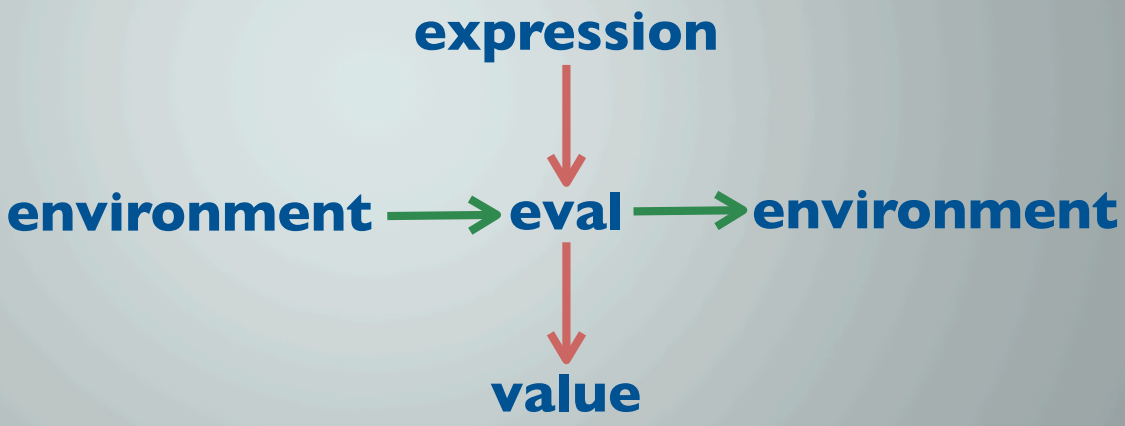
XML

http://www.eds.com/services_offerings/so_finframe.shtml

Everything must be first class

- first class classes
- first class methods
- first class programs
- first class computation
- closures
- environments

A little language



A little language

```
{ tag => fun(tag):
    [tag, fun];

  else: 0;

  case@clauses:
    { default: void;
      siz: size(clauses);
      max: 0;
      for(k: 1, k <= siz, k:= k+1,
        { clause: clauses[k];
          if(clause[1] = else,
             default:= clause[2],
             if(clause[1] > max,
                max:= clause[1],
                void)) });
      tbl[max]: default;
      for(k: 1, k <= siz, k:= k+1,
        { clause: clauses[k];
          if(clause[1] != else,
             tbl[clause[1]]:= clause[2]) });

      select(tag):
        if(tag > max,
          default,
          { fun: tbl[tag]; fun(tag)) }}
```

operator, free and canonical function format

no special forms

call-by-name

tables as compound data structures

higher-order functions

pico = 10^{-12}

A little language: features

- minimal&regular syntax
- infix operators
- tables everywhere
- first-class everything
- call-by-name
- abstract syntax

A little language: features

- minimal&regular syntax

variable	tabulation	application	
`x` variable/constant reference	`t[idx]` table indexing	`f(1, x)` function call	invocation
`v: 123` variable definition	`t[10]: x()` variable table definition	`f(x): x+x` variable function definition	invocation: expression
`c:: 123` constant definition	`t[10]:: y()` constant table definition	`f(x):: x*x` constant function definition	invocation:: expression
`v:= 123` variable assignment	`t[10]:= 0` table modification	`f(x):= -x` function redefinition	invocation:= expression

A little language: features

- minimal&regular syntax
- infix operators
- tables everywhere
- first-class everyth
- call-by-name
- abstract syntax

```
a++b: a+b+1
<function ++>
a**b: a*b*2
<function **>
p<=>q: abs(p-q)<1
<function <=>>
1++2<=>1**2
<native true>
```


A little language: features

```
counter():
 { count:0;
   counter():=
     count:=count+1;
   counter() }
<function counter>
tab[10]: counter()
<table>
display(tab)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

l&regular syntax

erators

- tables everywhere
- first-cla
- call-by-
- abstrac

```
table@arguments: arguments
<function table>
begin@arguments: arguments[size(arguments)]
<function begin>
T: table(1,2,3,4,5)
<table>
display(T)
[1, 2, 3, 4, 5]
begin(X: 1, Y: 2, X+Y)
3
```

A little language: features

- minimal&reg… …ntax
- infix operato…
- tables every…
- **first-class everything**
- call-by-name
- abstract syntax

```
number
fraction
text
function
table
environment
continuation
void
```


A little language: features

- …regular syntax
- …ators
- tables ever…
- **first-class …**
- **call-by-name**
- abstract syntax

```
{ true(p, q())::
    p;
  false(p(), q)::
    q;
  if(p, c(), a())::
    p(c(), a());
  while(p(), e())::
    { loop(value, boolean)::
        boolean(loop(e(), p()), value);
      loop(void, p()) }}
```

```
map(filter(item), table):
  { index: 0;
    filtered_table[size(table)]:
      filter(table[index:= index+1]) }
<function map>
display(map(item^2, [1,2,3,5,7,11]))
[1, 4, 9, 25, 49, 121]
```

A little language: features

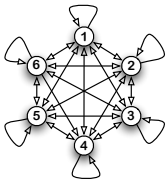
```
<expression>    ::= <void> | ... | <number>

<void>          ::= VOI
<reference>     ::= REF <name>
<application>   ::= APL <expression> <arguments>
<tabulation>    ::= TBL <expression> <indexation>
<declaration>   ::= DCL <invocation> <expression>
<definition>    ::= DEF <invocation> <expression>
<assignment>    ::= SET <invocation> <expression>
<constant>      ::= CST <name> <expression> <dictionary>
<variable>      ::= VAR <name> <expression> <dictionary>
<continuation>  ::= CNT <dictionary> <number> <number> <table>
<native>        ::= NAT <name> <number>
<function>      ::= FUN <name> <arguments> <expression> <dictionary>
<table>         ::= TAB <table>
<text>          ::= TXT <text>
<fraction>      ::= FRC <fraction>
<number>        ::= NBR <number>

<name>          ::= <text>
<indexation>    ::= <table>
<arguments>     ::= <table>
<arguments>     ::= <invocation>
<dictionary>    ::= <variable>
<dictionary>    ::= <constant>
<dictionary>    ::= <void>
<invocation>    ::= <reference>
<invocation>    ::= <application>
<invocation>    ::= <tabulation>
```

- **abstract syntax**

http://www.cs.uni-bonn.de/~costanza/lisp-ecoop/

A little language: architecture

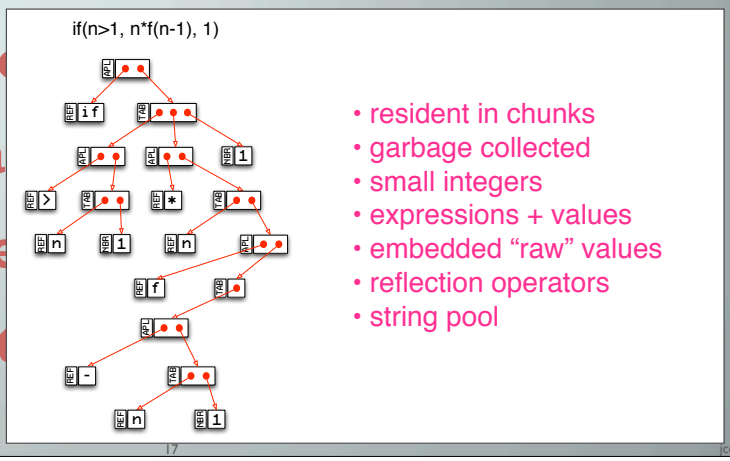
- **uniform memory + gc**
- **abstract grammar driven**
- **environ**
- **thread/**
- **tail recu**
- **smart caching**



- variable-length chunks
- mark-and-compact gc
- tagged size-headers
- single bit per cell for gc
- programs -> chunks
- values -> chunks
- environments -> chunks
- threads -> chunks

A little language: architecture

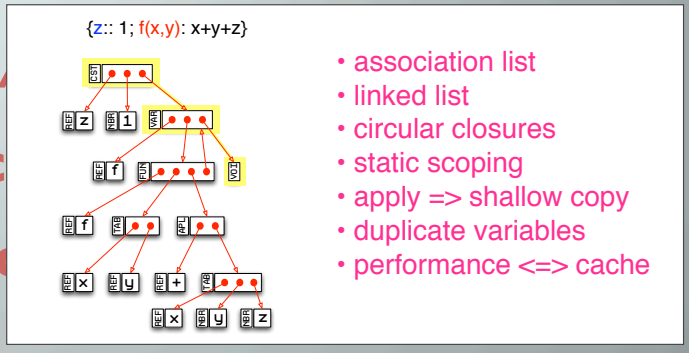
- uniform memory + gc
- abstract grammar driven
- envir
- threa
- tail re
- smart



- resident in chunks
- garbage collected
- small integers
- expressions + values
- embedded “raw” values
- reflection operators
- string pool

A little language: architecture

- uniform memory + gc
- abstract grammar driven
- environments as lists
- thread
- tail rec
- smart



- association list
- linked list
- circular closures
- static scoping
- apply => shallow copy
- duplicate variables
- performance <=> cache

A little language: re

```
NIL_type THR_evaluate(NIL_type)
  { NBR_type nbr;
    UNS_type idx;
    THK_type thk;
    for (;;)
     { GLOBAL_claim_default();
       nbr = THREAD->nbr;
       idx = get_NBR(nbr);
       thk = THUNKS[idx];
       thk(THREAD); }}
```

evaluation engine

infinite loop

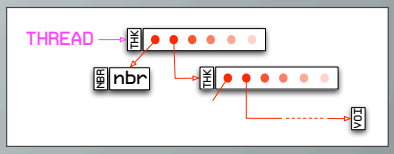
preempt memory overflow

identify thunk

lookup thunk code

execute thunk code

- thread/continuation style
- tail recursion
- smart caching



A little language: architecture

```
static void evaluate_function_body(EXP_type Bod, DCT_type Dct)
  { DCT_type dct;
    dct = DICT;
    DICT = Dct;
    THR_poke_eval_1(rET_thunk, Bod, dct); }
```

+ gc

abstract grammar driven

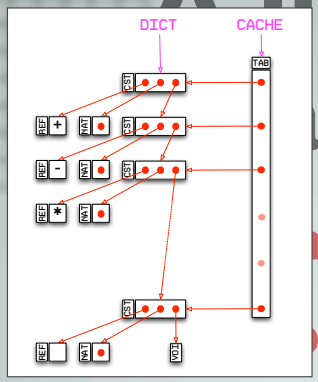
environ

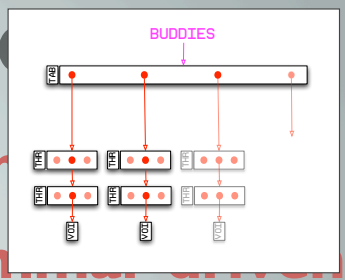
thread

```
static void evaluate_function_body(EXP_type Bod, DCT_type Dct)
  { DCT_type dct;
    THR_zap();
    dct = DICT;
    DICT = Dct;
    if (THR_get_thunk() == rET_thunk)
      { THR_keep_eval(Bod); }
    else
      THR_push_eval_1(rET_thunk, Bod, dct); }
```

- tail recursion
- smart caching

A little language: architecture





- uniform mem
- abstract gram
- environments as lis
- thread
- tail re
- **smart caching**

	DrScheme	Pico
Quicksort(20000)	1.372	1.237
Eratosthenes(50000)	0.380	0.366
Fibonacci(25)	0.436	0.648

little objects

```
counter(n):
  { incr(): n:= n+1;
    decr(): n:= n-1;
    clone() }
<function counter>
c: counter(10)
<dictionary>
c.incr()
11
d: counter(5)
<dictionary>
d.decr()
4
```

capture the dynamic environment

instantiate

qualification

- object = environment
- instantiate = call
- message = lookup

little objects: inheritance

```
counter(n):
  { incr(): n:= n+1;
    decr(): n:= n-1;
    super: void;
    protect(limit):
      { incr():
          if(n = limit,
            error("overflow"),
            super.incr());
        decr():
          if(n = -limit,
            error("underflow"),
            super.decr());
        clone() };
    super:= clone() }
<function counter>
c: counter(10)
<dictionary>
d: c.protect(11)
<dictionary>
d.incr()
11
d.incr()
<user error: overflow>
```

mixin method

overriding methods

sub-prototype

prototype creation

- inheritance = nesting
- overriding = homonyms
- late binding = lookup

little objects: cloning

```
Stack(n):
  { T[n]: void;
    t: 0;
    empty():: t = 0;
    full():: t = n;
    push(x)::
      { T[t:= t+1]:= x;
        self() };
    pop()::
      { x: T[t];
        t:= t-1; x };
    makeProtected()::
      { push(x)::
          if(full(),
            error("overflow"),
            .push(x));
        pop()::
          if(empty(),
            error("underflow"),
            .pop());
        clone() };
    clone() }
<closure Stack>
{ s: Stack(10);
  p: s.makeProtected();
  p.pop() }
<user error: underflow>
```

: : = constant declaration

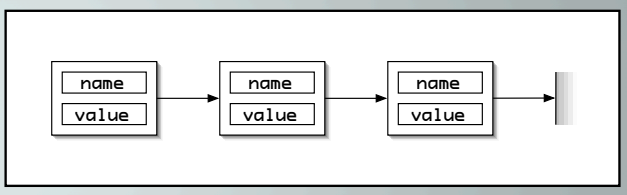
self reference

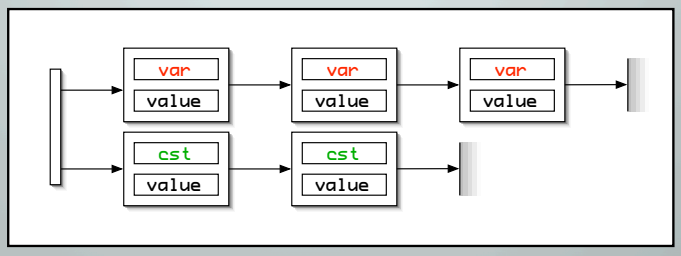
super send

- constant = visible/shared
- variable = hidden/unshared
- clone = shallow/deep copy

little objects: code sharing

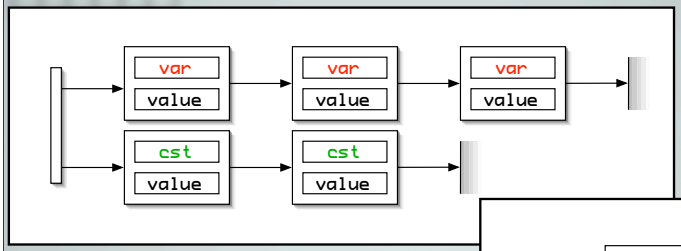
...Before



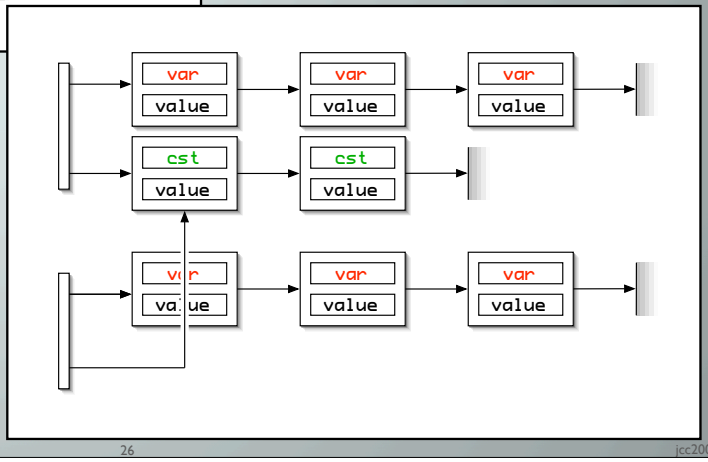


After...

little objects: code sharing



prototypes are cloned with clone(obj)

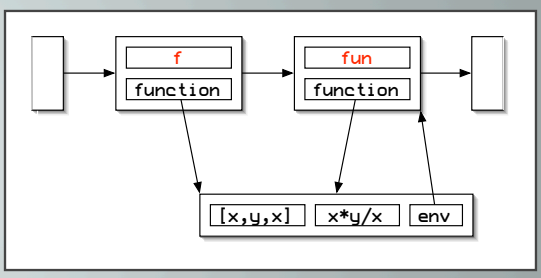


- clone(variables) = deep copy
- clone(constants) = shallow copy

little objects: lazy closures

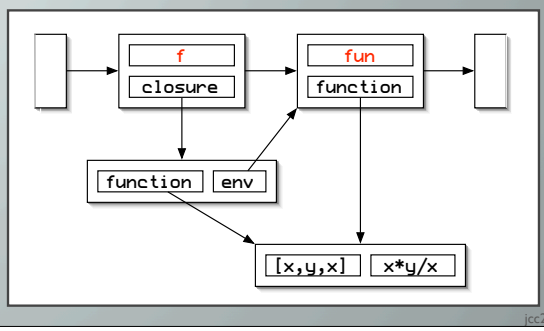
...Before

```
fun(x,y,z): x*y/z
<function fun>
f: fun
<function fun>
```



After...

```
fun(x,y,z): x*y/z
<closure fun>
f: fun
<closure fun>
```



little objects: Pic%

- Stable, consistent object model
- Simple semantics
- Shared attributes
- Multi-paradigm
- First class methods
- Dynamic scope

defined by a metacircular interpreter

convertible into e.g. denotational semantics

no difference between scalars, tables, functions

smooth integration of procedures, methods

can be assigned, passed, returned, etc.

but: overriding semantics imply dynamic scope

Message Oriented Programming
The Case for First Class Messages
Dave Thomas,

First-class values

Procedural
Hybrid
Pure

	functions	methods	classes	arguments	environments	computation
C++	±	±	-	±	-	-
Java	inner classes	-	-	-	-	-
Smalltalk	blocks	-	+	-	+	(+)
Scheme	+	closures / environments		+	-	±
Pic%	+			+	+	+
CLOS	(+)	(+)	+	+	+	±

first class ≠reflection

Aml-oriented programming

Structuring the Object-Soup
Structuring Concurrency
Structuring Distribution
Structuring Mobility

The ChitChat model

Partial Failures
Non-blocking Communication
Rubberband References
Reversible Computations

Future Research

Avoiding classes

Many Technical Inconveniences

Classes =
Information Shared by instances
static variables
identity of methods

Fundamental Problem
Classes are an implicit sharing mechanism between objects that becomes painfully explicit due to distribution in open networks. Class-based languages offer no means to deal with it manually.

Avoiding prototypes

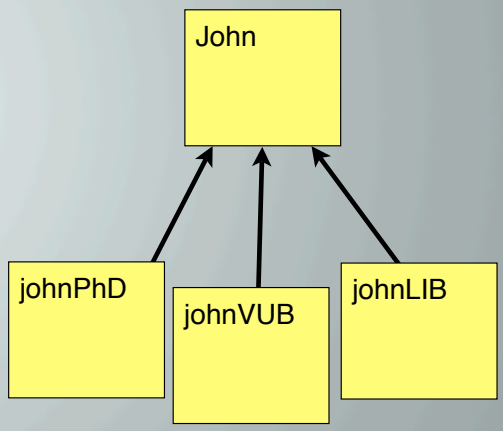
Prototypes=
Objects + Messages
Reflection
Delegation
Cloning

Fundamental Problem
Prototype-based languages apply a variety of encapsulation-breaching and identity breaching operators on objects. Hence a security issue is created.

Parent sharing (e.g. Self)

```
view.Person()::
 { ...
   view.PhDStudent()::
    { ... };
   view.Researcher()::
    { ... };
   view.LibraryUser():: ...;
          { ... } }
```

```
john: this().Person()
johnPhD: john.PhDStudent()
johnVUB: john.Researcher()
johnLIB: john.LibraryUser()
```



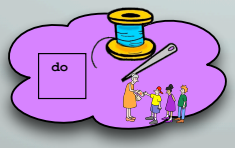
Active objects

asynchronous messages spawn concurrency

```
aview.fib(n)::{
  do()::if(n<2,
        1,
        athis().fib(n-1).do()+athis().fib(n-2).do()) }
```

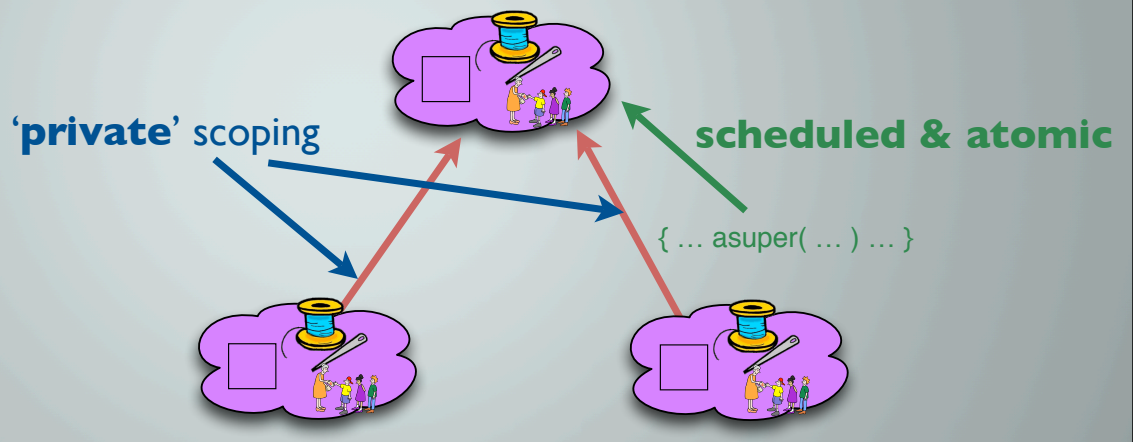
creates an active object (no intra-object concurrency!)

synchronization

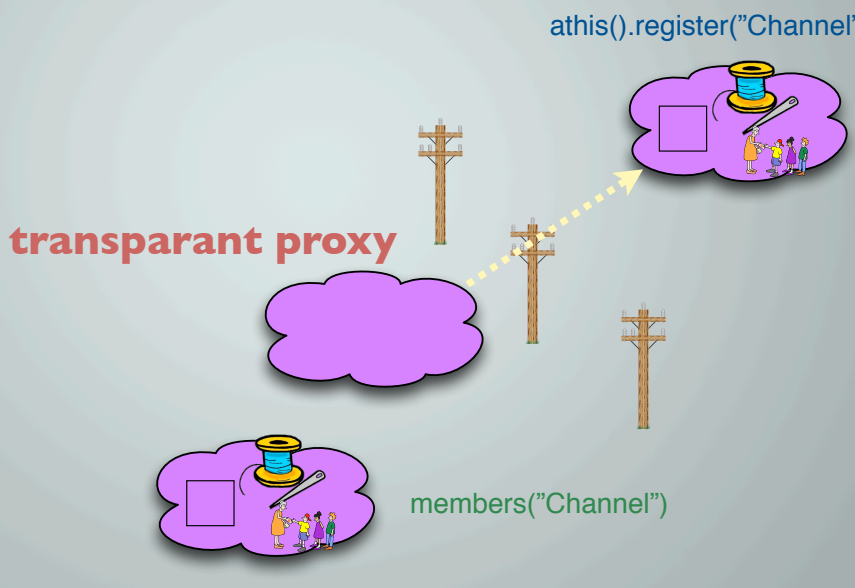


transparant promise-delivery

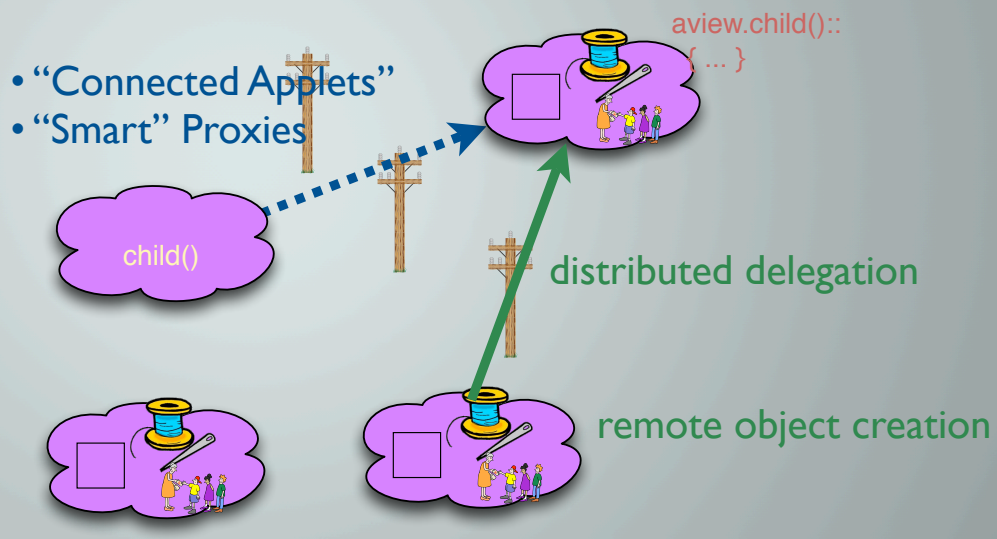
Parent sharing



Remote acquaintances



Remote object creation



Move methods

Move methods are like ordinary methods

- **have a name**
- **can have arguments**
- **have a body**

And when invoked, their body is executed

But

- **all objects on the method lookup path are first moved to the site of the sender**
- **if there is no move, then the body is not executed**

Example: swarms

```
{ e:English man();
  f:Français();
  b:Belg();
  s:Swarm(e.come,f.viens,b.kom)
  s.doMove() }
```

send move message

```
aview.Swarm@methods::{
  move.doMove@args ::
    foreach(methods,elm@args)
  }
```

call first-class move methods

```
aview.English man()::{
  move.come()::{
    display("arrived")}}

aview.Français()::{
  move.viens()::{
    display("arrivé")}}

aview.Belg()::{
  move.kom()::{
    display("aangekomen")}}
```

group first-class move methods

From Pic% to ChitChat

Pic%

Classless : Objects are Idiosyncratic
Expressiveness of prototypes
- First-class methods
- Cloning, dynamic object extension
- Extensible

Secure
- Objects + Messages (!)
- Capability-based

ChitChat

A **Concurrency** Model
A **Mobility** Model
A **Distribution** Model

The ChitChat model

ChitChat is at the junction of

- Prototype-based language design
- Concurrent object-oriented language design
- Distributed and mobile language design

Technical contributions

- Intersecting class-based and prototype-based languages
- Intersecting object-based and lambda-based languages
- Reconciling prototype-based delegation and concurrency
- "Smart" proxies are delegation-based descendants
- Move analysed as harmful; parallel with goto
- A first proposal for structured mobility


```
unify(Ex1, Ex2, Frm):
  void;

unify_fail@Any:
  void;

same_number(Nb1, Nb2):
  Nb1[NBR_NBR_idx] = Nb2[NB

same_fraction(Fr1, Fr2):
  Fr1[FRC_FRC_idx] = Fr2[FRC

same_text(Tx1, Tx2):
  Tx1[TXT_TXT_idx] = Tx2[TXT

same_void(Vo1, Vo2):
  true;

same_fail(Vo1, Vo2):
  false;

unify_values_case: case(NBR_tag # same_number,
                        FRC_tag # same_fraction,
                        TXT_tag # same_text,
                        VOI_tag # same_void,
                          void # same_fail);

unify_values(Val, Exp, Frm):
  { tg1: Val[TAG_idx];
    tg2: Exp[TAG_idx];
    if(tg1 = tg2,
      { cas: unify_values_case(tg1);
        if(cas(Val, Exp),
          Frm,
          void) }) };

referenced(Var, Exp):
  { referenced_variable(Va1, Va2):
      same_variable(Va1, Va2);

    referenced_table_items(Var, Tab, Idx):
      if(Idx > size(Tab),
        false,
        if(referenced(Var, Tab[Idx]),
          true,
          referenced_table_items(Var, Tab, Idx+1)));

    referenced_table(Var, Tab):
      referenced_table_items(Var, Tab[TAB_TAB_idx], 1);

    referenced_pattern(Var, Pat):
      referenced_table(Var, Pat[PAT_TMS_idx]);

    referenced_value(Var, Val):
      false;

    referenced_case: case(VAR_tag # referenced_variable,
                          TAB_tag # referenced_table,
                          PAT_tag # referenced_pattern,
                            void # referenced_value);

    referenced(Var, Exp):=
      { tag: Exp[TAG_idx];
        cas: referenced_case(tag);
          cas(Var, Exp) };

    referenced(Var, Exp) };

unify_variable(Var, Exp, Frm):
  { tag: Exp[TAG_idx];
    cas: unify_variable_case(tag);
    cas(Var, Exp, Frm) };

unify_table_items(Ta1, Ta2, Frm, Idx):
  if(Idx > size(Ta1),
    Frm,
    unify_table_items(Ta1, Ta2, unify(Ta1[Idx], Ta2[Idx], Frm), Idx+1));

unify_2_tables(Ta1, Ta2, Frm):
  { ta1: Ta1[TAB_TAB_idx];
    ta2: Ta2[TAB_TAB_idx];
    if(size(ta1) = size(ta2),
      unify_table_items(ta1, ta2, Frm, 1),
      void) };

unify_table_case: case(VAR_tag # unify_variable,
                       TAB_tag # unify_2_tables,
                         void # unify_fail);

unify_table(Tab, Exp, Frm):
  { tag: Exp[TAG_idx];
    cas: unify_table_case(tag);
    cas(Exp, Tab, Frm) };

unify_2_patterns(Pa1, Pa2, Frm):
  if(Pa1[PAT_SYM_idx] = Pa2[PAT_SYM_idx],
    unify(Pa1[PAT_TMS_idx], Pa2[PAT_TMS_idx], Frm),
    void);

unify_pattern_case: case(VAR_tag # unify_variable,
                         PAT_tag # unify_2_patterns,
                           void # unify_fail);

unify_pattern(Pat, Exp, Frm):
  { tag: Exp[TAG_idx];
    cas: unify_pattern_case(tag);
    cas(Exp, Pat, Frm) };

unify_value(Val, Exp, Frm):
  { tag: Exp[TAG_idx];
    if(tag = VAR_tag,
      unify_variable(Exp, Val, Frm),
      unify_values(Val, Exp, Frm)) };

unify_case: case(VAR_tag # unify_variable,
                 TAB_tag # unify_table,
                 PAT_tag # unify_pattern,
                   void # unify_value);

unify(Tm1, Tm2, Frm):=
  { tag: Tm1[TAG_idx];
    cas: unify_case(tag);
    cas(Tm1, Tm2, Frm) };
```

http://pico.vub.ac.be

Bibliography

- De Meuter, W., D'Hondt, T., and Dedecker, J.
  Intersecting classes and prototypes.
  Ershov Memorial Conference (2003), M. Broy and A.V. Zamulin, Eds.,
  vol. 2890 of LNCS, Springer.
- Dedecker, J., and Van Belle, W. Actors for mobile ad-hoc networks.
  Embedded and Ubiquitous Computing (2004), L. Yang, M. Guo, G. Gao, and N. Jha, Eds.,
  vol. 3207 of LNCS, Springer.
- De Meuter, W.
  Move Considered Harmful: A Language Design Approach to Mobility and Distribution for Open Networks.
  PhD thesis, Vrije Universiteit Brussel, September 2004.
- Van Belle, W.
  Creation of an Intelligent Concurrency Adaptor in order to mediate the Differences between Conflicting Concurrency Interfaces.
  PhD thesis, Vrije Universiteit Brussel, September 2003.
- Steyaert, P.
  Open Design of Object-Oriented Languages, A Foundation for Specialisable Reflective Language Frameworks.
  PhD thesis, Vrije Universiteit Brussel, September 1994.
- Van Cutsem, T., Mostinckx, S., De Meuter, W., Dedecker, J. and D'Hondt, T. On the Performance of SOAP in a Non-Trivial Peer-to-Peer Experiment
  Proceedings of 2nd International Conference of Component Deployment, 2004.
- Van Belle, W., Fabry, J. Verelst, K. and D'Hondt, T.
  Experiences in mobile computing: the CBORG mobile multi-agent system.
  Proceedings of Tools 38, IEEE Computer society press, 2001.
- Van Belle W. and D'Hondt T.
  Agent Mobility and Reification of Computational State, an experiment in migration.
  Infrastructure for Agents, Multi-Agent Systems, and Scalable Multi-Agent Systems,
  LNAI Springer, 2000.
- D'Hondt T. and De Meuter W.
  Of first-class methods and dynamic scope.
  RSTI L'objet 9/2003. LMO 2003.
- **http://pico.vub.ac.be**