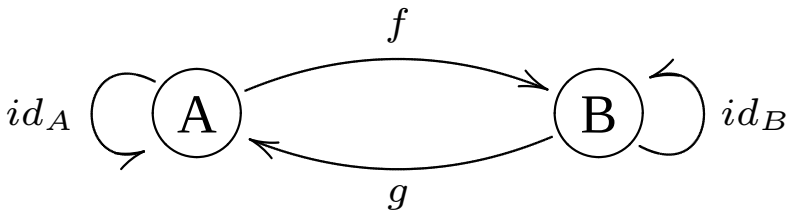
$\cong$ A survey of isomorphisms of types

Roberto Di Cosmo *(INRIA and PPS)*

WIT'02
Toulouse
8-9 November 2002

$\cong$ What is an isomorphism?

- A and B are *isomorphic* iff there exist f and g such that



A and B may be:

- types in a λ-calculus
- objects in a category
- formulae of a logic
- specifications of software components
- • • •

Why isomorphisms of types?

How to find software components?

by name? No: arbitrary

SML	sub_string
Ocaml	String.sub
CamlLight	substring
Haskell	subStr (?)

by type? Better, but still a bit arbitrary:

```
Objective Caml version 3.04
# String.sub;;
- : string -> int -> int -> string = <fun>

Standard ML of New Jersey, Version 0.93
- substring;;
val it = fn : string * int * int -> string
```

A more complex example: fold

```
sumlist [a1;...;an] = 0+a1+a2+...+an  = fold + 0 [a1;...;
```

```
multlist [a1;...;an] = 1*a1*a2*...*an  = fold * 1 [a1;...
```

```
#let rec fold op acc = function [] -> acc
# | (a::r) -> fold op (op acc a) r;;
val fold : ('a -> 'b -> 'b) -> 'b -> 'a
list -> 'b = <fun>
```

Language	Name	Type
ML of Edinburgh LCF	itlist	$('a \to 'b \to 'b) \to 'a\ list \to 'b \to 'b$
CAML	list_it	”
Haskell	foldl	$('a \to 'b \to 'a) \to 'a \to 'b\ list \to 'a$
Ocaml	List.fold_left	”
SML of New Jersey	fold	$('a \times 'b \to 'b) \to 'a\ list \to 'b \to 'b$
Edinburgh SML Library	foldl	$('a \times 'b \to 'b) \to 'a\ list \to 'b \to 'b$

≅ Retrieve *functions* up to **type isomorphisms**

In `CamlLight v0.7+` you get `camlsearch`[a]:

find "fold" :

```
ranger> camlsearch -s -e "'b*'a list*('b->'a->'b) ->
'b"
it_list : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
list_it : ('b -> 'a -> 'a) -> 'b list -> 'a -> 'a
```

find instances :

```
ranger> camlsearch -x -l "_x->string*int->string"
```

find more general cases `ranger> camlsearch -x -m "(int->int) ->`

```
int list -> int list"
```

$\cong$ We try to find all type isomorphisms

But one must be very precise about:

the types under consideration

the language allowed for building converters

for recursive types we do not want to prove

$$int * int \cong int$$

the equational theory used to prove the isomorphism

dont forget extensionality!

[Dezani 1976] the only invertible term of λ_β is $I = \lambda x.x$

For *each language*, we try to find ***all*** the isomorphisms

(swap) $A \to (B \to C) = B \to (A \to C)$ $\}\ Th^1$

1. $A \times B = B \times A$
2. $A \times (B \times C) = (A \times B) \times C$
3. $(A \times B) \to C = A \to (B \to C)$
4. $A \to (B \times C) = (A \to B) \times (A \to C)$
5. $A \times \mathbf{T} = A$
6. $A \to \mathbf{T} = \mathbf{T}$
7. $\mathbf{T} \to A = A$

$\}\ Th^1_{\times T}$

8. $\forall X.\forall Y.A = \forall Y.\forall X.A$
9. $\forall X.A = \forall Y.A[Y/X]$ [a]
10. $\forall X.(A \to B) = A \to \forall X.B$ [b]

$\}\ + \mathbf{swap} = Th^2$

11. $\forall X.A \times B = \forall X.A \times \forall X.B$
12. $\forall X.\mathbf{T} = \mathbf{T}$

$\}\ Th^2_{\times T}$

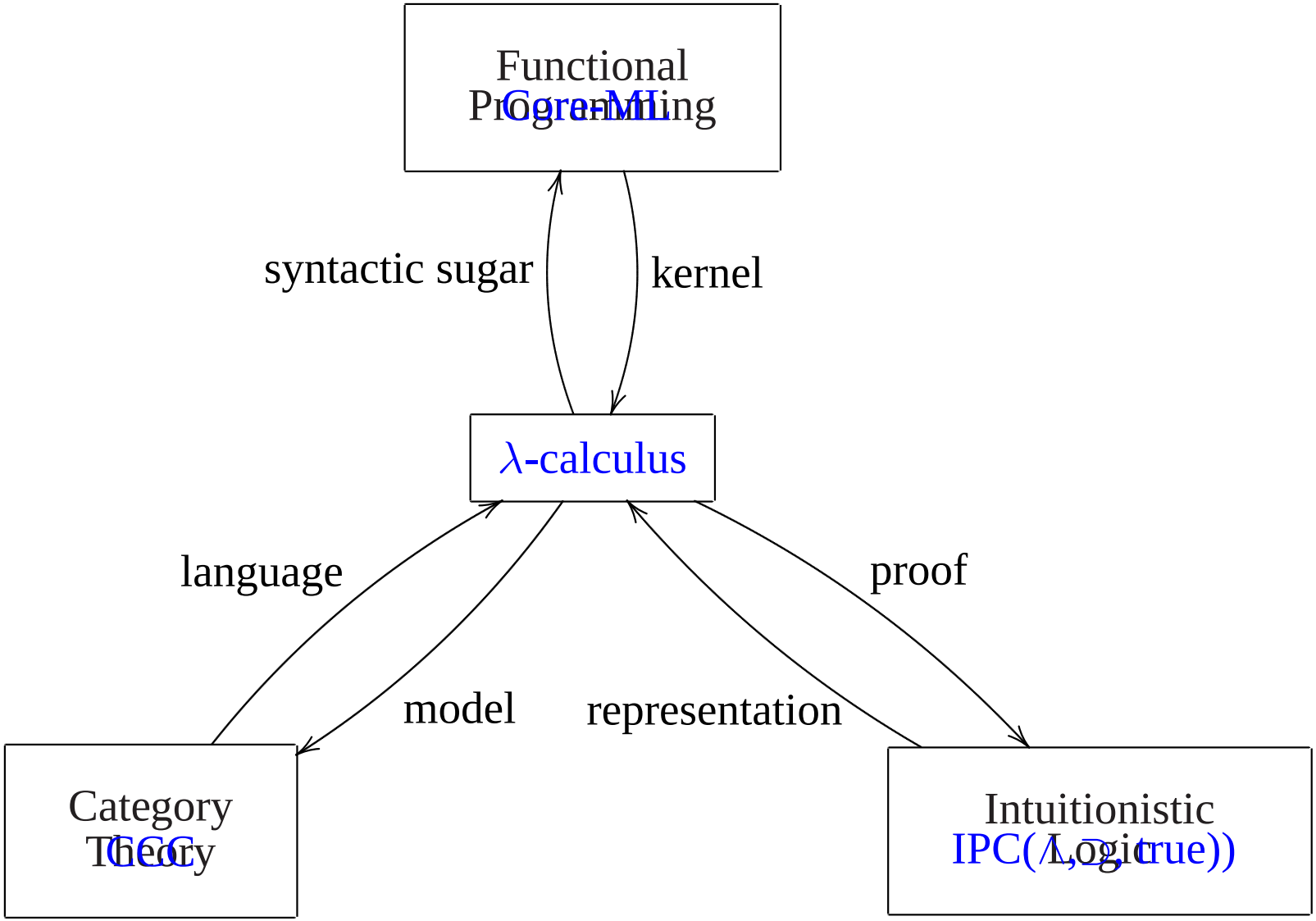
split $\forall X.A \times B = \forall X.\forall Y.A \times (B[Y/X])$

$\}\ -10, 11 = Th^{ML}$

---

[a] X free for Y in A; $Y \notin FTV(A)$ [b] $X \notin FTV(A)$

Isomorphisms and Curry-Howard



$\cong$ A simple dictionary

Type	*Proposition*	*Categorical object*
$A \to B$	$A \supset B$	B^A
$A \times B$	$A \wedge B$	$A \times B$
$Unit$	$True$	1

Example:

$$(A \times B) \to C = A \to (B \to C) \qquad \text{(Curry)}$$

categories

$$C^{(A \times B)} = (C^B)^A$$

proofs

$$(A \wedge B) \supset C = A \supset (B \supset C)$$

ML

```
(A * B) -> C = A -> (B -> C)
```

$\cong$ Proof techniques

syntactic study invertible terms

$$M \text{ invertible iff } \exists M'.M \circ M' = \lambda x.x \text{ and } M' \circ M = \lambda x.x$$

- important proviso:

  no isomorphisms without extensional axioms!
- difficult, technical, but done for a variety of systems
- easier, more geometric, when working in Linear Logic
- impractical for systems with the sum type

semantic study the class of models of the language

- nice connection with number theory and Tarski's High School Algebra Problem for the first order theories
- but breaks down with the empty type

Isomorphisms: completeness-related work

1972 Dezani: FHP are the only invertible λ-terms

1981 Soloviev: completeness for CCCs via **FinOrd**

1985 Bruce-Longo: completeness for $\lambda - \beta\eta^{\tau}$ via Dezani's FHP result

1990 Bruce-Di Cosmo-Longo: completeness for CCCs syntactically

1991 Di Cosmo: completeness for F+SP+terminal object (and subsystems)

1991 Rittri: application of $Th^1_{\times T}$ to library search

1992 Di Cosmo: completeness for core-ML

1992 Rittri: matching up to $Th^1_{\times T}$is decidable

1993 Soloviev: completeness for linear lambda calculus

1993 Rittri: unification up to linear isomorphisms is decidable

1995 Di Cosmo: monography on isomorphisms, question of sum types posed

1996 Andreev-Solovev: $n(log^2(n))$ decision algorithm for linear isomorphisms

1999 Di Cosmo-Balat: completeness in Multiplicative Linear Logic

2000 Gil: subtyping arithmetical types

2001 Balat-Di Cosmo-Fiore: isos with sums is not finitely axiomatisable

2002 Laurent: completeness for LLP

This side of the Completeness Graal

In many cases completeness seems out of reach:

dependent types :

useful for retrieving theorems in Coq/PVS/Mizar/HOL etc.

recursive types :

useful for modeling objects, and for searches in OO libraries

module types :

useful for glueing together pre-existing software modules

Nevertheless, even incomplete theories are immediately useful.

Isomorphisms: “incomplete” results

1993 Rollins-Wing-Zaremsky: naïve isomorphisms for SML modules

1996 Muller: searching SML modules up to naïves isomorphismes

1996 Aponte-Di Cosmo: isomorphisms for Ocaml modules

1997 Delahaye-Di Cosmo-Werner: isomorphisms in Coq

1997 IBM’s mockingbirdMockingbird project

1998 Aponte-Di Cosmo-Dubois: first subtyping of Ocaml modisosmodules up to isomorphism

1998 Di Cosmo-Lopez: sound foundation for Mockingbird (unpublished)

2000 Longo-Milsted-Soloviev: Axiom C collapses retyping functions

2000 Palsberg-Zhao: AC isos for modjavarecursive types via perfect bipartite graph matching

2002 Yakobowski: first implementation of a prototype for Ocaml module subtyping up to isos

2002 Di Cosmo-Pottier-Rémy: subtyping of recursive types up to AC isos via bipartite graph matching (subm.)

2002 Jha-Palsberg-Zhao-Henglein: AC isos for recursive types via stable graph partitions

Conclusions and **perspectives**

- Increasing interest of researchers with different background
- A wealth of open problems to explore, both theoretical and practical ones
- Applications now motivated by the exponetial explosions of resources on the Web both for programmers and mathematicians!
- The right time to have this workshop!

APPENDIX!

calculus

$$\frac{}{\Gamma,\ x:\tau,\ \Gamma' \vdash x:\tau} \qquad \frac{}{\Gamma \vdash () : \mathtt{1}}$$

$$\frac{\Gamma \vdash t_i : \tau_i \quad (i=1,2)}{\Gamma \vdash \langle t_1, t_2 \rangle : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash t : \tau_1 \times \tau_2}{\Gamma \vdash \pi_i(t) : \tau_i}\ (i=1,2)$$

$$\frac{\Gamma,\ x:\tau_1 \vdash t : \tau}{\Gamma \vdash \lambda x:\tau_1.\, t : \tau_1 \to \tau} \qquad \frac{\Gamma \vdash t : \tau_1 \to \tau \qquad \Gamma \vdash t_1 : \tau_1}{\Gamma \vdash t t_1 : \tau}$$

$$\frac{\Gamma \vdash t : \mathtt{0}}{\Gamma \vdash \bot_\tau : \tau}$$

$$\frac{\Gamma \vdash t : \tau_i}{\Gamma \vdash \iota_i^{\tau_1,\tau_2}(t) : \tau_1 + \tau_2}\ (i=1,2) \qquad \frac{\Gamma \vdash t : \tau_1 + \tau_2 \qquad \Gamma,\ x_i:\tau_i \vdash t_i : \tau \quad (i=1,2}{\Gamma \vdash \mathsf{case}(t, x_1:\tau_1.\, t_1, x_2:\tau_2.\, t_2) : \tau}$$

Figure 1: Typing rules.

The calculus and its **extensional** equalities

$$\frac{\Gamma \vdash t : \tau}{\Gamma \vdash t = t : \tau} \qquad \frac{\Gamma \vdash t = t' : \tau}{\Gamma \vdash t' = t : \tau} \qquad \frac{\Gamma \vdash t_1 = t_2 : \tau \qquad \Gamma \vdash t_2 = t_3 : \tau}{\Gamma \vdash t_1 = t_3 : \tau}$$

$$\frac{\Gamma \vdash t : 1}{\Gamma \vdash t = () : 1}$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash \pi_i \langle t_1, t_2 \rangle = t_i : \tau_i} \; i = 1, 2 \qquad \frac{\Gamma \vdash t : \tau_1 \times \tau_2}{\Gamma \vdash t = \langle \pi_1(t), \pi_2(t) \rangle : \tau_1 \times \tau_2}$$

$$\frac{\Gamma, x : \tau_1 \vdash t : \tau \qquad \Gamma \vdash t_1 : \tau_1}{\Gamma \vdash (\lambda x : \tau_1 . t)(t_1) = t[x/t_1] : \tau} \qquad \frac{\Gamma \vdash t : \tau_1 \to \tau}{\Gamma \vdash t = \lambda x : \tau_1 . t(x) : \tau_1 \to \tau} \; x \notin FV(t)$$

$$\frac{\Gamma \vdash t : \tau_1 \to \tau \qquad \Gamma \vdash t_1 = t_1' : \tau_1}{\Gamma \vdash t(t_1) = t(t_1') : \tau} \qquad \frac{\Gamma, x : \tau_1 \vdash t = t' : \tau}{\Gamma \vdash \lambda x : \tau_1 . t = \lambda x : \tau_1 . t' : \tau_1 \to \tau}$$

$$\frac{\Gamma \vdash \bot_0 : 0 \qquad \Gamma \vdash t : \tau}{\Gamma \vdash \bot_\tau = t : \tau}$$

$$\frac{\Gamma \vdash t : \tau_j \qquad \Gamma, x_i : \tau_i \vdash t_i : \tau \quad i = 1, 2}{\Gamma \vdash \mathsf{case}(\iota_j(t), x_1 . t_1, x_2 . t_2) = t_j[x_j/t] : \tau} \; j = 1, 2 \qquad \frac{\Gamma \vdash t : \tau_1 + \tau_2 \qquad \Gamma, x : \tau_1 + \tau_2}{\Gamma \vdash \mathsf{case}(t, x_1 . t'[x/\iota_1(x_1)], x_2 . t'[x/\iota_2(x_2)}$$

Isomorphisms of modisos*Modules* (with Aponte and Dubois)

To search an ML module system we must handle "syntax choices":

in the base language : function names, isomorphic types

in the module language : module names, order of functor parameters, structure nesting, ...

from the composition : product vs structures, functions vs functors

```
module UnifyCurry :
   functor (t:TERMS) functor (s:SUBSTITUTION with s.termtype = t.termty
   sig
        val c: int*string
       unify: t.termtype -> t.termtype -> s.substtype
   end

module UnifyUnCurry :
   functor (sig module t:TERMS module s:SUBSTITUTION with s.termtype =
   sig
        val c1: int
        val c2: string
        findunifier: t.termtype * t.termtype -> s.substtype
   end
```

$\cong$ Isomorphisms of modjava*Recursive Types* to match Java classes in $O(n^2)$ (Palsberg and Zhao)

$\cong$ Isomorphisms of *Recursive Types* to match Java classes in $O(n^2)$ (Palsberg and Zhao)

```
interface I1 {
float m1 (I1 a);
int m2 (I2 a);
}
```
$\mu\alpha.(\alpha \to float) \times (I_2 \to int)$

```
interface I2 {
I1 m3 (float a);
I2 m4 (float a);
}
```
$\mu\delta.(float \to I_1) \times (float \to \delta)$

```
interface J1 {
J1 n1 (float a);
J2 n2 (float a);
}
```
$\mu\beta.(float \to \beta) \times (float \to J_2)$

```
interface J2 {
int n3 (J1 a);
float n4 (J2 a);
}
```
$\mu\eta.(J_1 \to int) \times (\eta \to float)$

$$I_1 \equiv J_2 \ ?$$

$\cong$ Isomorphisms of *Recursive Types* to match Java classes in $O(n^2)$ (Palsberg and Zhao)

```
interface I1 {
float m1 (I1 a);
int m2 (I2 a);
}
```
$\mu\alpha.(\alpha \to float) \times (I_2 \to int)$

```
interface I2 {
I1 m3 (float a);
I2 m4 (float a);
}
```
$\mu\delta.(float \to I_1) \times (float \to \delta)$

```
interface J1 {
J1 n1 (float a);
J2 n2 (float a);
}
```
$\mu\beta.(float \to \beta) \times (float \to J_2)$

```
interface J2 {
int n3 (J1 a);
float n4 (J2 a);
}
```
$\mu\eta.(J_1 \to int) \times (\eta \to float)$

$$I_1 \equiv J_2 \ ?$$

$\cong$ Isomorphisms of *Recursive Types* to match Java classes in $O(n^2)$ (Palsberg and Zhao)

```
interface I1 {
float m1 (I1 a);
int m2 (I2 a);
}
```
$\mu\alpha.(\alpha \to float) \times (I_2 \to int)$

```
interface I2 {
I1 m3 (float a);
I2 m4 (float a);
}
```
$\mu\delta.(float \to I_1) \times (float \to \delta)$

```
interface J1 {
J1 n1 (float a);
J2 n2 (float a);
}
```
$\mu\beta.(float \to \beta) \times (float \to J_2)$

```
interface J2 {
int n3 (J1 a);
float n4 (J2 a);
}
```
$\mu\eta.(J_1 \to int) \times (\eta \to float)$

$$I_1 \equiv J_2 \ ?$$

$\cong$ Isomorphisms of *Recursive Types* to match Java classes in $O(n^2)$ (Palsberg and Zhao)

```
interface I1 {
float m1 (I1 a);
int m2 (I2 a);
}
```
$\mu\alpha.(\alpha \to float) \times (I_2 \to int)$

```
interface I2 {
I1 m3 (float a);
I2 m4 (float a);
}
```
$\mu\delta.(float \to I_1) \times (float \to \delta)$

```
interface J1 {
J1 n1 (float a);
J2 n2 (float a);
}
```
$\mu\beta.(float \to \beta) \times (float \to J_2)$

```
interface J2 {
int n3 (J1 a);
float n4 (J2 a);
}
```
$\mu\eta.(J_1 \to int) \times (\eta \to float)$

$$I_1 \equiv J_2 \ ?$$

$\cong$ Isomorphisms of *Recursive Types* to match Java classes in $O(n^2)$ (Palsberg and Zhao)

```
interface I1 {
float m1 (I1 a);
int m2 (I2 a);
}
```
$\mu\alpha.(\alpha \to float) \times (I_2 \to int)$

```
interface I2 {
I1 m3 (float a);
I2 m4 (float a);
}
```
$\mu\delta.(float \to I_1) \times (float \to \delta)$

```
interface J1 {
J1 n1 (float a);
J2 n2 (float a);
}
```
$\mu\beta.(float \to \beta) \times (float \to J_2)$

```
interface J2 {
int n3 (J1 a);
float n4 (J2 a);
}
```
$\mu\eta.(J_1 \to int) \times (\eta \to float)$

$$I_1 \equiv J_2 \ ?$$

$\cong$ Isomorphisms of *Recursive Types* to match Java classes in $O(n^2)$ (Palsberg and Zhao)

```
interface I1 {
float m1 (I1 a);
int m2 (I2 a);
}
```
$\mu\alpha.(\alpha \to float) \times (I_2 \to int)$

```
interface I2 {
I1 m3 (float a);
I2 m4 (float a);
}
```
$\mu\delta.(float \to I_1) \times (float \to \delta)$

```
interface J1 {
J1 n1 (float a);
J2 n2 (float a);
}
```
$\mu\beta.(float \to \beta) \times (float \to J_2)$

```
interface J2 {
int n3 (J1 a);
float n4 (J2 a);
}
```
$\mu\eta.(J_1 \to int) \times (\eta \to float)$

$$I_1 \equiv J_2 \ ?$$

$\cong$ Isomorphisms of *Recursive Types* to match Java classes in $O(n^2)$ (Palsberg and Zhao)

```
interface I1 {
float m1 (I1 a);
int m2 (I2 a);
}
```
$\mu\alpha.(\alpha \to float) \times (I_2 \to int)$

```
interface I2 {
I1 m3 (float a);
I2 m4 (float a);
}
```
$\mu\delta.(float \to I_1) \times (float \to \delta)$

```
interface J1 {
J1 n1 (float a);
J2 n2 (float a);
}
```
$\mu\beta.(float \to \beta) \times (float \to J_2)$

```
interface J2 {
int n3 (J1 a);
float n4 (J2 a);
}
```
$\mu\eta.(J_1 \to int) \times (\eta \to float)$

$$I_1 \equiv J_2 \ ?$$

$\cong$ Isomorphisms of mockingbird*ML-like types* as an alternative to weak IDLs (Auerbach, Barton, and Raghavachari)

IBM's Mockingbird project: how do we exchange data between different languages?

Java:

```
public class Point {
private float x;
private float y;
...};
public class PVector
extends Vector ;
```

C++:

```
class Point {
float x;
float y;
public:
...};
class PVector {
int len; float *xs;
float *ys; ...};
```

Solution 1: use an IDL (e.g. CORBA)...

But IDLs are restrictive (e.g. CORBA), one needs to agree *beforehand*

Solution 2: program freely, then produce *automatically* the conversion code for each pair of peers.

$\cong$ Isomorphisms of **ML-like types** as an alternative to weak IDLs (Auerbach, Barton, and Raghavachari)

Ibm's Mockingbird project solution 2 for `Java`, `CORBA` et `C++`.

1) analysis of type definitions in each program
2) conversion in internal representation (ML-like types + annotations)
3) test of isomorphism
4) generation of code for the marshaler

Need isomorphisms also with sum and recursion

$\cong$ Isomorphisms of recursive types...

$$\begin{aligned} A \times B\ list &= \mu X.(A \times B) \times X + 1 \\ &= \mu X.(B \times A) \times X + 1 = B \times A\ list \end{aligned}$$

There are 3 kinds of isomorphisms

identity A = B iff the interpretation of A and B coincide, e.g. equivalence among different syntactic representations of the same object

$$\mu X.A \times X = \mu X.A \times (A \times X)$$

Amadio/Cardelli/Fiore/Abadi:

$$\frac{A = F(A)}{A = \mu X.F(X)}$$

$\cong$ Isomorphisms of recursive types...

realised by the identity A and B are isomorphic via terms without computational content, e.g.

$$\forall X.\forall Y.A = \forall Y.\forall X.A$$

proper if the terms have computational content, e.g.

$$A \times B = B \times A$$

(this may be extended to a congruence on recursive types, see Palsberg et al.)

$\cong$ Isomorphisms of recursive types...

They must be kept separated, otherwise... *inconsistence...*

$$A = A \times 1 \Rightarrow A = \mu X.X \times 1 = B \Leftarrow B \times 1 = B$$

Theorem(with P.M. Lopez) The system $(=_{\text{identité}} \cup =_{\text{proper}})^*$ is consistent.

This incomplete system should suffice for Mockigbird.