Contents

Subtyping Recursive Types modulo Associative Commutative Products

Didier Rémy *INRIA-Rocquencourt*

Joint work with:

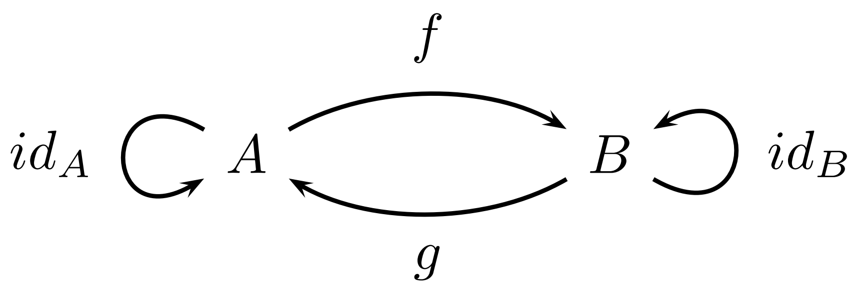
Roberto Di Cosmo *University of Paris 7*

François Pottier *INRIA-Rocquencourt*

IFIP, December 2003

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
- • • •

We strive to find all type isomorphisms

Usually, one tries to be very precise about:

- the types under consideration
- the language allowed for building converters
- the equational theory used to prove the isomorphism

We want, if possible:

- a complete characterization
- an efficient decision algorithm
- a way to build the converters

Sometimes, we know 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. $\mathbf{T} \to A = A$
8. $\forall X.\forall Y.A = \forall Y.\forall X.A$
9. $\forall X.A = \forall Y.A[Y/X]$ $^{(a)}$
10. $\forall X.(A \to B) = A \to \forall X.B$ $^{(b)}$
11. $\forall X.A \times B = \forall X.A \times \forall X.B$
12. $\forall X.\mathbf{T} = \mathbf{T}$

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

(1–7: $Th^1_{\times T}$; 8–10: **+swap** $= Th^2$; 1–12: $Th^2_{\times T}$; 1–12 and split: $- 10, 11 = Th^{ML}$)

$^{(a)}$ X free for Y in A and $Y \notin FTV(A)$. $^{(b)}$ $X \notin FTV(A)$.

Isomorphisms of recursive types

We want to have *explicit* recursive types for

Search in OO libraries

recursive types (μ) are a key tool to describe objects and classes

Automatic adapter synthesis

recursive types (μ) are a key tools in Mockingbird, together with sum types

But isomorphisms of recursive types

is a very tricky subject!

Isomorphism of recursive types

three kinds

Identity A = B because $[\![A]\!]=[\![B]\!]$. *e.g.*

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

Captured by Amadio/Cardelli/Fiore/Abadi's "fix" rule:

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

Identity realised A = B is proved by terms that erase to the identity. *e.g.* $\forall X.\forall Y.A = \forall Y.\forall X.A$

Proper A = B has a computational content, *e.g.* $A \times B = B \times A$

Different kinds must not be mixed!

For any A and B we have the “proper” isomorphisms

$$A = A \times 1 \qquad B = B \times 1$$

If we mix them with “identity” isomorphisms, we can apply fix

$$\frac{A = A \times 1}{A = \mu X.X \times 1} \qquad \frac{B = B \times 1}{B = \mu X.X \times 1}$$

And then, we conclude $A = B$!

The system $Th^1_{\times T} \cup Amadio/Cardelli$ is inconsistent!

Isomorphisms of recursive types. . .

However, we can get some useful results if we give up the quest for completeness:

Side-by-side strategy

Theorem (Di Cosmo-Lopez)
The system $(=_{\text{Amadio/Cardelli}} \cup =_{Th^1_{\times T}})^*$ is consistent.
This seems to suffice to validate many of the Mockingbird rules.

Workable-subsystem strategy

Approach used by Palsberg and Zhao: consider only the isomorphisms of recursive types generated by applying the associativity and commutativity rule to finite sets of products.
This is also the approach we will follow here.

Chronology

1996 Abadi-Fiore's "*Syntactic considerations on recursive types*": find the coercions between "identity" isomorphic types, discover the problem with $A = A \times 1$

1997 IBM's Mockingbird project: motivational examples for proper recursive isomoprhisms

1998 Brandt-Henglein "*Coinductive axiomatization of recursive type equality and subtyping*"

2000 Palsberg-Zhao: efficient equality of recursive types up to $AC(\times)$ via perfect bipartite graph matching in $O(n^2)$

2002 Jha-Palsberg-Zhao: more efficient equality of recursive types up to $AC(\times)$ via size-stable graph partitions in $O(n \log n)$

2002 Jha-Palsberg-Zhao-Henglein: minor variant of above

2002 Di Cosmo-Pottier-Rémy: subtyping of recursive types up to $AC(\times)$ via bipartite graph matching this work

Recursive types

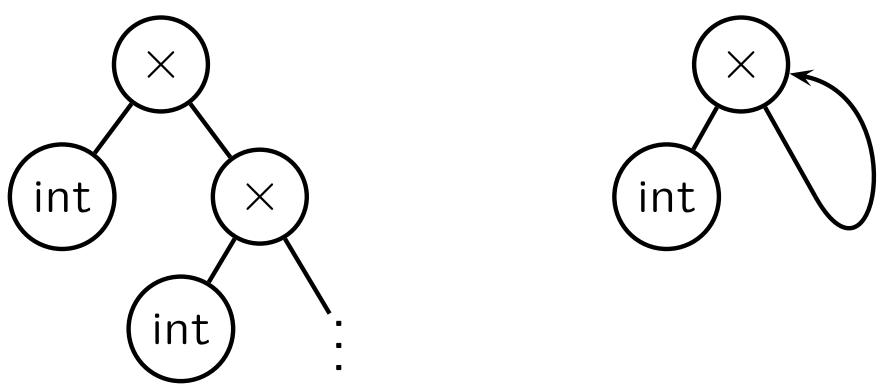
A recursive type can be equivalently presented as:

μ-**notation** a finite set of recursive equations

(can be coded with the μ operator)

$$I = \mathsf{int} \times I \qquad\qquad (\mu\alpha.int \times \alpha)$$

regular trees a (possibly infinite) tree having only a finite number of distinct subtrees (can be represented as a graph)



representable term a (possibly infinite) term whose partial function is related to the set of traces of a finite automaton

Problem:

> *Find a possible implementation of interface I in a Java library S, but abstracting from method and interfaces names.*

Coding interfaces as recursive types, forgetting names, it can be

Restated in terms of recursive types:

> *Given two recursive types A and B, is it possible to reorder the products (using associativity and commutativity) in a way that makes A and B coincide?*

This is precisely equivalence of recursive types up to $AC(\times)$.

As usual, we will get rid of associativity by collapsing trees of binary products into n-ary products $\Pi_{i=1}^{n}$ (just Π in what follows).

$$A \to (A \times B) \times C = A \to \Pi(A, B, C)$$

Matching Java classes (Palsberg-Zhao)

interface I_1 {
  float m_1 (I_1 a);
  int m_2 (I_2 a);
}

$I_1 = \Pi(I_1 \rightarrow float, I_2 \rightarrow int)$

interface I_2 {
  I_1 m_3 (**float** a);
  I_2 m_4 (**float** a);
}

$I_2 = \Pi(float \rightarrow I_1, float \rightarrow I_2)$

interface J_1 {
  J_1 n_1 (**float** a);
  J_2 n_2 (**float** a);
}

$J_1 = \Pi(float \rightarrow J_1, float \rightarrow J_2)$

interface J_2 {
  int n_3 (J_1 a);
  float n_4 (J_2 a);
}

$J_2 = \Pi(J_1 \rightarrow int, J_2 \rightarrow float)$

$I_1 \equiv J_2$?

Equivalence of recursive types

We know how to efficiently test for equality two recursive types:

- define equality coinductively as the largest relation satisfying

Eq-Top
$$\frac{t \mathrel{\mathcal{R}} t'}{t(\epsilon) = t'(\epsilon)}$$

Eq-Arrow
$$\frac{t_1 \to t_2 \mathrel{\mathcal{R}} t_1' \to t_2'}{t_1 \mathrel{\mathcal{R}} t_1' \qquad t_2 \mathrel{\mathcal{R}} t_2'}$$

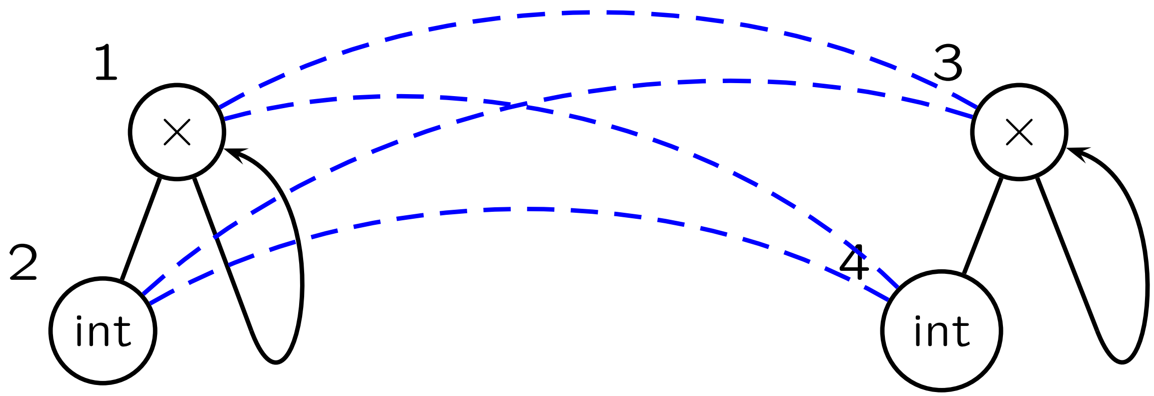
Eq-Pi
$$\frac{\Pi_{i=1}^{n} t_i \mathrel{\mathcal{R}} \Pi_{i=1}^{n} t_i'}{(t_i \mathrel{\mathcal{R}} t_i')^{i \in 1..n}}$$

- to decide $t \mathrel{\mathcal{R}} t'$, start from the full relation $\mathcal{R}_0 = T \times T'$, and propagate inconsistencies with the definition of $\mathcal{R}$

Examples (1)

This can be schematically represented via a bipartite graph, related nodes of both types (represented as graphs).

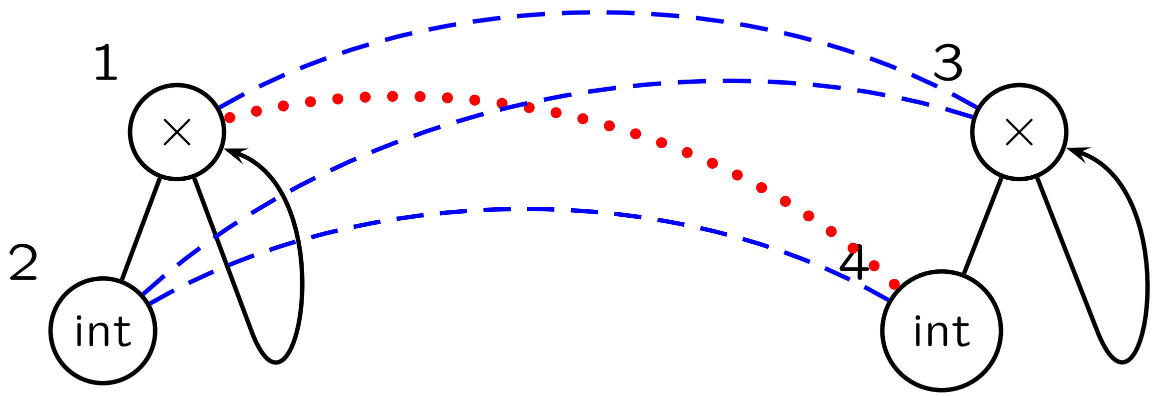
Examples (1)



$$\mathcal{R}_0 = \ (1,3),(1,4),(2,3),(2,4)$$

This can be schematically represented via a bipartite graph, related nodes of both types (represented as graphs).

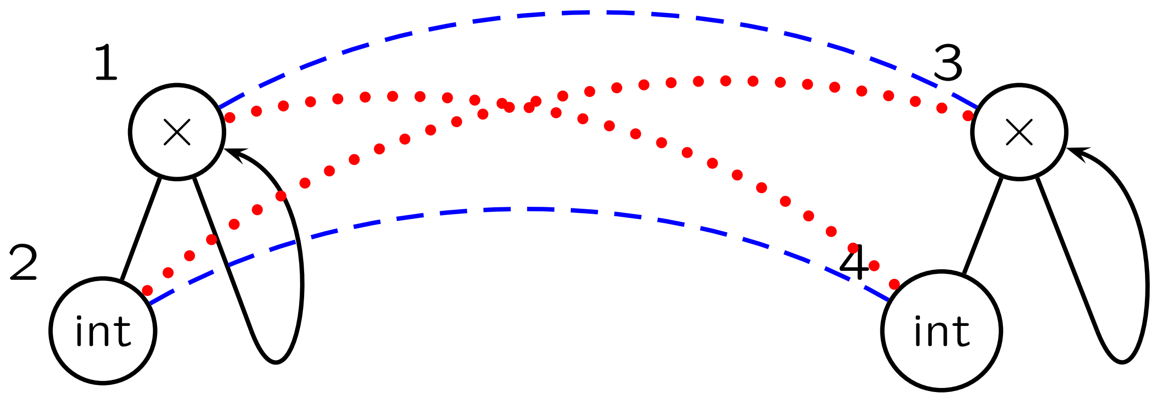
Examples (1)



$$\mathcal{R}_0 = \ (1,3),(1,4),(2,3),(2,4)$$

Immediately invalid relations are removed, . . .

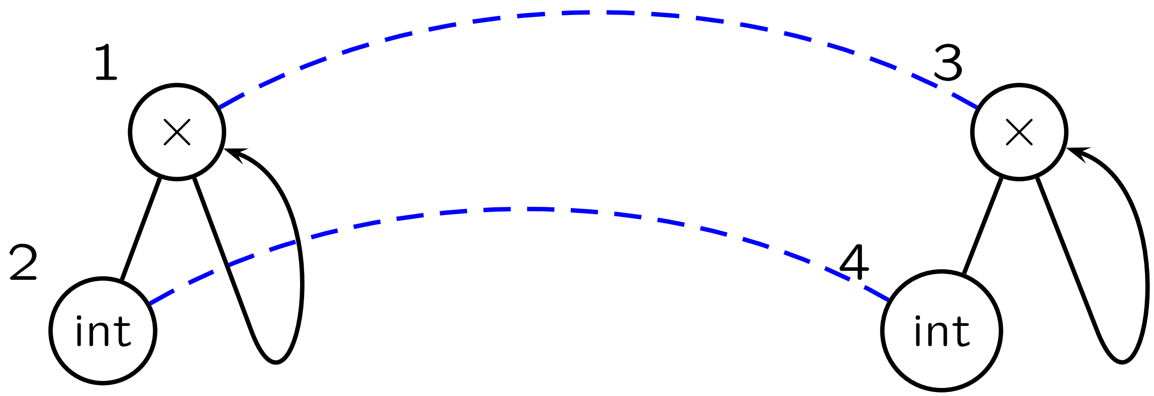
Examples (1)



$$\mathcal{R}_0 = \ (1,3),(1,4),(2,3),(2,4)$$

Immediately invalid relations are removed, . . .

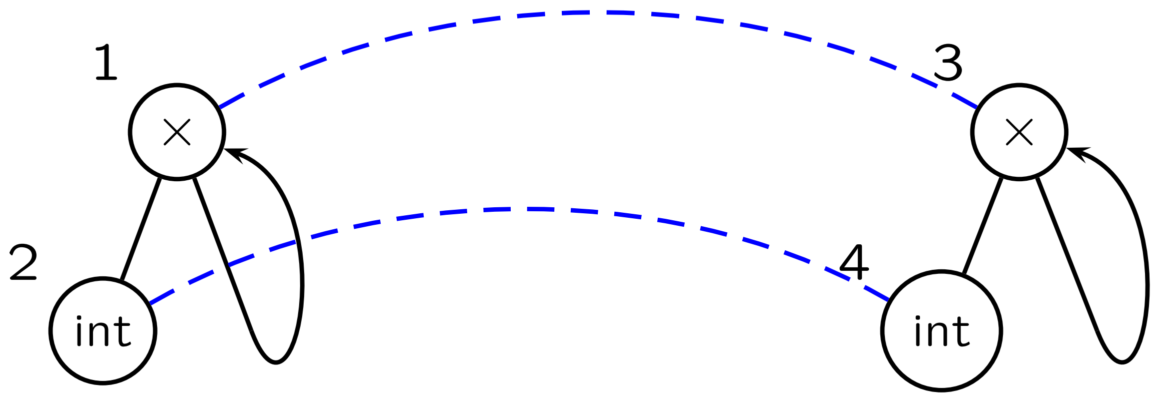
Examples (1)



$$\mathcal{R}_0 = (1,3),(1,4),(2,3),(2,4)$$
$$\mathcal{R}_1 = (1,3),(2,4)$$

. . . , which in turn may immediately invalidate other relations.

Examples (1)

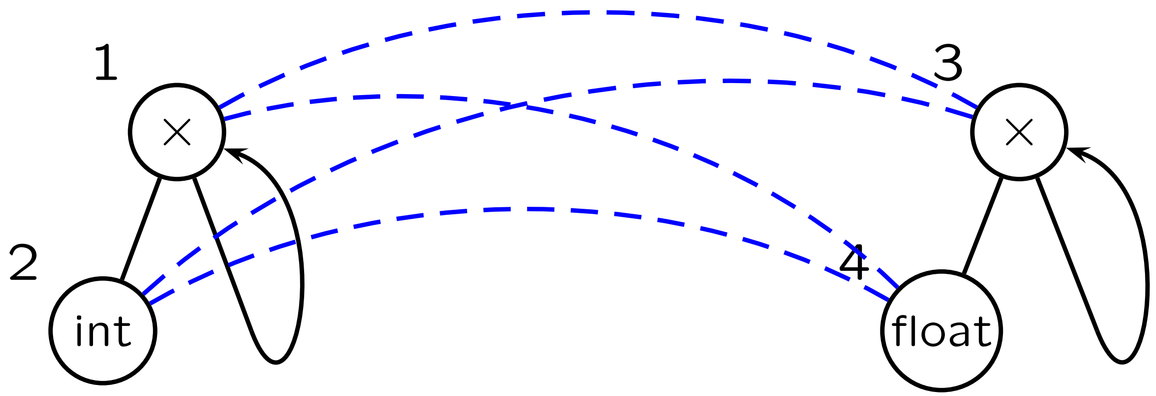


$$\mathcal{R}_0 = (1,3),(1,4),(2,3),(2,4)$$

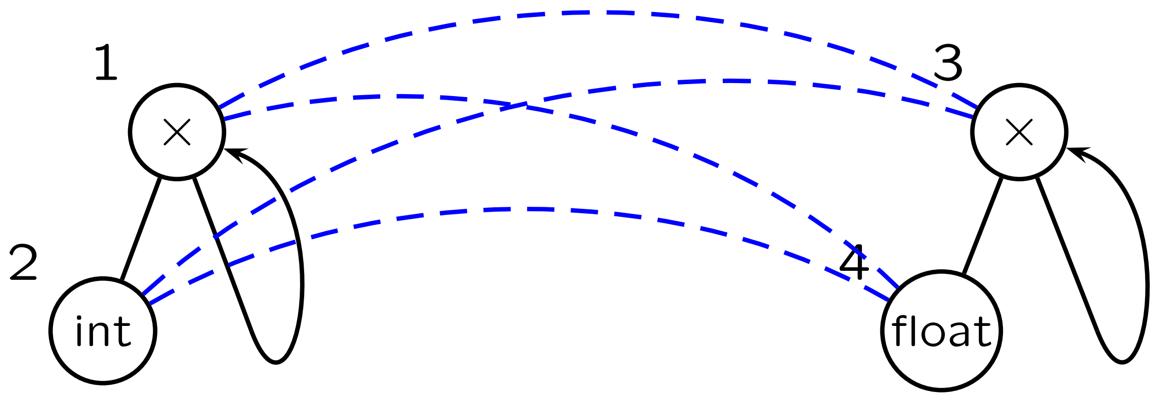
$$\mathcal{R}_1 = (1,3),(2,4)$$

$$R_2 = (1,3),(2,4)$$ *success*

Examples (2)

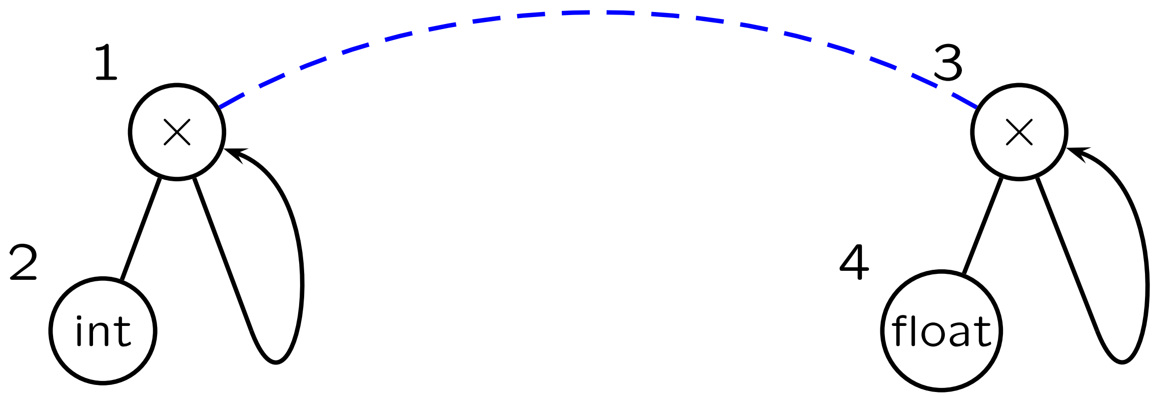


Examples (2)



$$\mathcal{R}_0 = \ (1,3),(1,4),(2,3),(2,4)$$

Examples (2)



$$\mathcal{R}_0 = \ (1,3),(1,4),(2,3),(2,4)$$

$$\mathcal{R}_1 = \ (1,3)$$

Examples (2)

$$\mathcal{R}_0 = (1,3),(1,4),(2,3),(2,4)$$

$$\mathcal{R}_1 = (1,3)$$

$$\mathcal{R}_2 = \emptyset$$

$failure$

Examples (2)

$$\mathcal{R}_0 = (1,3),(1,4),(2,3),(2,4)$$

$$\mathcal{R}_1 = (1,3)$$

$$\mathcal{R}_2 = \emptyset$$

$failure$

Examples (2)

$$\mathcal{R}_0 = (1,3),(1,4),(2,3),(2,4)$$

$$\mathcal{R}_1 = (1,3)$$

$$\mathcal{R}_2 = \emptyset$$

failure

Equivalence of rec. types up to $AC(\times)$

Modify the test for equality of two recursive types:

- $=$ (equality)
  is define coinductively as the largest relation $\mathcal{R}$ satisfying

$$\frac{t \;\mathcal{R}\; t'}{t(\epsilon) = t'(\epsilon)} \qquad \frac{t_1 \to t_2 \;\mathcal{R}\; t'_1 \to t'_2}{t_1 \;\mathcal{R}\; t'_1 \qquad t_2 \;\mathcal{R}\; t'_2} \qquad \frac{\Pi_{i=1}^{n} t_i \;\mathcal{R}\; \Pi_{i=1}^{n} t'_i}{(t_i \quad \mathcal{R}\; t'_i)^{i \in 1..n}}$$

Equivalence of rec. types up to $AC(\times)$

Modify the test for equality of two recursive types:

- $=_{AC}$ (equality up to $AC(\times)$)
  is define coinductively as the largest relation $\mathcal{R}$ satisfying

$$\frac{t \; \mathcal{R} \; t'}{t(\epsilon) = t'(\epsilon)} \qquad \frac{t_1 \to t_2 \; \mathcal{R} \; t'_1 \to t'_2}{t_1 \; \mathcal{R} \; t'_1 \qquad t_2 \; \mathcal{R} \; t'_2} \qquad \frac{\Pi_{i=1}^{n} t_i \; \mathcal{R} \; \Pi_{i=1}^{n} t'_i}{\exists \sigma \in \Sigma_n^n, \; (t_{\sigma(i)} \; \mathcal{R} \; t'_i)^{i \in 1..n}}$$

Equivalence of rec. types up to $AC(\times)$

Modify the test for equality of two recursive types:

- $=_{AC}$ (equality up to $AC(\times)$)
  is define coinductively as the largest relation $\mathcal{R}$ satisfying

$$\frac{t \mathrel{\mathcal{R}} t'}{t(\epsilon) = t'(\epsilon)} \qquad \frac{t_1 \to t_2 \mathrel{\mathcal{R}} t'_1 \to t'_2}{t_1 \mathrel{\mathcal{R}} t'_1 \qquad t_2 \mathrel{\mathcal{R}} t'_2} \qquad \frac{\Pi_{i=1}^n t_i \mathrel{\mathcal{R}} \Pi_{i=1}^n t'_i}{\exists \sigma \in \Sigma_n^n,\ (t_{\sigma(i)} \mathrel{\mathcal{R}} t'_i)^{i \in 1..n}}$$

- to decide $t \mathrel{\mathcal{R}} t'$, proceed as for usual equality, but at Π nodes, use a "perfect graph matching" algorithm to check consistency of $\Pi(a_1, .., a_n) = \Pi(b_1, \ldots, b_n)$ with the relation $\mathcal{R}_n$.

Stocktaking

From previous work we have a very efficient algorithm for $=_{AC}$.

What is missing?

Subtyping up to $AC(\times)$: the type of a queried interface may be very complex: the user wants to ask only for a *supertype*.

A reasonable query for the Collection type (with 15 methods) is

```
public interface SomeCollection {
  public void    add (Object o);
  public void    remove (Object o);
  public boolean contains (Object o);
  public int     size ();
}
```

Bad news: the optimizations used in Palsberg *et al.* fail here.

Glue code We want the search tool to also build coercions...

Subtyping up to $AC(\times)$

Let $\leq_0$ be the ordering on symbols generated by:

$$\bot \leq_0 s \qquad s \leq_0 \top \qquad \rightarrow \leq_0 \rightarrow \qquad \frac{n \geq m}{\Pi^n \leq_0 \Pi^m}$$

Definition 1 ($=_{AC}$-simulation) *[Reminder]*

A relation $\mathcal{R}$ is an $=_{AC}$-simulation if it satisfies

$$\frac{t_1 \; \mathcal{R} \; t_2}{t_1(\epsilon) = \; t_2(\epsilon)} \qquad \frac{t_1 \rightarrow t_2 \; \mathcal{R} \; t_1' \rightarrow t_2'}{t_1' \; \mathcal{R} \; t_1 \qquad t_2 \; \mathcal{R} \; t_2'} \qquad \frac{\Pi_{i=1}^{n} t_i \; \mathcal{R} \; \Pi_{i=1}^{m} t_i'}{\exists \sigma \in \Sigma_n^{\;m} \, , \; (t_{\sigma(i)} \; \mathcal{R} \; t_i')^{i \in 1..\, m}}$$

Subtyping up to $AC(\times)$

Let $\leq_0$ be the ordering on symbols generated by:

$$\bot \leq_0 s \qquad s \leq_0 \top \qquad \rightarrow \leq_0 \rightarrow \qquad \frac{n \geq m}{\Pi^n \leq_0 \Pi^m}$$

Definition 2 ($\leq_{AC}$-simulation)
A relation $\mathcal{R}$ is an $\leq_{AC}$-simulation if it satisfies

$$\frac{t_1 \; \mathcal{R} \; t_2}{t_1(\epsilon) \leq_0 t_2(\epsilon)} \qquad \frac{t_1 \rightarrow t_2 \; \mathcal{R} \; t'_1 \rightarrow t'_2}{t'_1 \; \mathcal{R} \; t_1 \qquad t_2 \; \mathcal{R} \; t'_2} \qquad \frac{\Pi^n_{i=1} t_i \; \mathcal{R} \; \Pi^m_{i=1} t'_i}{\exists \sigma \in \Sigma^m_n, \; (t_{\sigma(i)} \; \mathcal{R} \; t'_i)^{i \in 1..m}}$$

Subtyping up to $AC(\times)$

Let $\leq_0$ be the ordering on symbols generated by:

$$\bot \leq_0 s \qquad s \leq_0 \top \qquad \rightarrow \leq_0 \rightarrow \qquad \frac{n \geq m}{\Pi^n \leq_0 \Pi^m}$$

Definition 2 ($\leq_{AC}$-simulation)
A relation $\mathcal{R}$ is an $\leq_{AC}$-simulation if it satisfies

$$\frac{t_1 \;\mathcal{R}\; t_2}{t_1(\epsilon) \leq_0 t_2(\epsilon)} \qquad \frac{t_1 \rightarrow t_2 \;\mathcal{R}\; t'_1 \rightarrow t'_2}{t'_1 \;\mathcal{R}\; t_1 \qquad t_2 \;\mathcal{R}\; t'_2} \qquad \frac{\Pi_{i=1}^{n} t_i \;\mathcal{R}\; \Pi_{i=1}^{m} t'_i}{\exists \sigma \in \Sigma_n^m,\; (t_{\sigma(i)} \;\mathcal{R}\; t'_i)^{i \in 1..m}}$$

Definition 2 *$\leq_{AC}$ is the largest $\leq_{AC}$-simulation.*

Theorem 1 *The relation $\leq_{AC}$ and $=_{AC} \circ \leq \circ =_{AC}$ coincide.*

The decision algorithm

Idea: to decide $t \leq_{AC} t'$, start from the full relation $R_0 = T \times T$, and propagate inconsistencies with the definition of $\leq_{AC}$.

Now, a pair $(p, q) \in R_k$ is *ordered*

(p is subtype of q, up to $AC(\times)$ at stage k).

To check validity of $(\Pi(a_1, \ldots, a_m), \Pi(b_1, \ldots, b_n))$ at stage k, we must check that, for some injection $\sigma : n \to m$, we have

$$\forall i \in 1..n, \quad (a_{\sigma(i)}, b_i) \in R_k$$

This can easily verified by looking for a maximal matching in the bipartite graph $(\{a_1, .., a_m\}, \{b_1, .., b_n\}, R_k)$, and checking that all the b_i are covered.

The decision algorithm (I)

1. **Let** $R = T \times T$ $\quad (T = subtrees(p_0))$
2. **Repeat**:

    Foreach pair p in R, **do:**

    If p is inconsistent, **then** remove p from R

    done

    until no pair is removed by the foreach loop
3. If $p_0 \notin R$, return *false*, otherwise return *true*.

Improving the decision algorithm (I)

Worst case complexity: $n^2 \cdot n'^2 \cdot d^{5/2}$

Improvement: avoid the $T \times T$ overkill!

- Pairs like $(\Pi(\cdots),\ t \to t')$ need not be considered at all!
- Perform an exploration of $T \times T$ starting from p_0 to build only the *relevant* universe U, i.e. the smallest set containing p_0 and closed under:

$$\frac{(t_1 \to t_2,\ t_1' \to t_2') \in U}{(t_1', t_1) \in U \qquad (t_2, t_2') \in U} \qquad\qquad \frac{(\Pi_{i=1}^{n} t_i, \Pi_{j=1}^{m} t_j') \in U}{((t_i, t_j') \in U)^{i \in \{1,\ldots,n\},\, j \in \{1,\ldots,m\}}}$$

  We also turn U into a directed graph: p is *parent* of q if p is a premise and q a conclusion of one of the rules.

- This can be done in time linear w.r.t. the size of U.

Improving the decision algorithm (II)

We can do better by accelerating the convergence.

- Our first algorithm, after removing the inconsistent pairs $p_1, \ldots, p_k$ from the relation R at stage i, restarts exploring blindly *all* pairs left at stage $i + 1$.
- It is enough to check *only* those pairs that are parents of the just removed pairs!

  (This idea is in Downey, Sethi and Tarjan's 1980 paper).
- and, of course, stop as soon as p_0 is no longer valid.

The worklist decision algorithm

1. **Let** $W = U$ and $S = F = \emptyset$.
2. **While** W is nonempty, **do**:
   (a) Take a pair p out of W;
   (b) **If** p is of the form $(\bot, t')$ or $(t, \top)$, **then** insert p into S;
   (c) **If** p is of the form $(t_1 \to t_2, t'_1 \to t'_2)$, then
   If $(t'_1, t_1) \notin F$ and $(t_2, t'_2) \notin F$ **then** insert p into S else invalidate p;
   (d) **If** p is of the form $(\Pi_{i=1}^{n} t_i, \Pi_{j=1}^{m} t'_j)$, **then**
   If there exists $\sigma \in \Sigma_n^m$ such that, for all $j \in \{1, \ldots, m\}$, $(t_\sigma(j), t'_j) \notin F$ holds, **then** insert p into S else invalidate p;
   (e) **If** p satisfied none of the three previous tests, **then** invalidate p.

3. **If** $p_0 \notin F$, return *true*, otherwise return *false*.

Complexity of improved algorithm

- The improved algorithm runs in time

$$size(U) \cdot d^{5/2} \quad \text{with} \quad size(U) \leq N \cdot N' \leq n^2 \cdot n'^2$$

- The worst case can be as bad as the naïve algorithm, but. . .
- In practice, it runs much better
  (typically, it is fast in rejecting folkloristic queries).

There is space for further improvement

The order in which pairs are removed from W is relevant

- look first at pairs that fail *earlier* (touch Π last)
- go bottom-up on acyclic types:

$$nodes(U) \cdot d^{5/2} \qquad \text{with} \qquad nodes(U) \leq n \cdot n'$$

- go bottom-up on strongly connected components of U:

$$\begin{array}{ccccc} nodes(U) \cdot d^{5/2} & < & c & < & size(U) \cdot d^{\,5/2} \\ \leq & & & & \leq \\ n \cdot n' & & & & N \cdot N' \\ & & & & \quad \leq n^2 \cdot n'^2 \end{array}$$

Searching for in a library

- Set the database as a whole graph.
- The algorithm is incremental: keep the algorithm structure, add new requests and continue.
- Sort the data-base along $\leq_{AC}$. (pre-compiled ordering on the data-base, so it does not cost) and start proceeds nodes top-down.
  - Gain in efficiency: no need to explore nodes below a failure.
  - Provide answers in group with their maximal element.

Conclusions

We have shown

- subtyping up to $AC(\times)$ is a natural composition of subtyping and $AC(\times)$:

$$\leq_{AC} \quad \equiv \quad =_{AC} \circ \leq \circ =_{AC}$$

- subtyping up to $AC(\times)$ is decidable,
- an efficient decision algorithm,
- an efficient coercion construction algorithm,
- a realistic basis for OO library search.

We need

- large scale experimentation on Java classes

Isomorphisms of *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.