# Transitions in Programming Models


Luca Cardelli

Microsoft Research
Cambridge UK


ICSE St. Louis, 2005-05-18

# Significant Transitions

- Programming languages (PLs)
  - They evolve slowly and occasionally, e.g.:
    - C to C++ : More robust data structures (objects)
    - C++ to Java : More robust control flows (strong typing)
  - But new *programming models* are invented routinely
    - As domain-specific libraries or API's
    - As program analysis tools
    - As language extensions

- Transitions
  - Significant transitions in programming models eventually "precipitate" into new programming languages (unpredictably)
  - We can watch out for significant transitions in programming models

# Transitions in 3 (related) areas

- We are in the middle of a radical transition in programming models (and eventually PLs)

- A new emphasis on computation on WANs
  - Wide area flows
    - Messages nor RPC, schedules not threads. *Messaging API's.*
    - Need to integrate these new flows into PL control constructs.
  - Wide area data
    - XML is "net data". *XML API's.*
    - Need to integrate this new data into PL data structures.
  - Wide area protection
    - Access control, data protection. *Security and privacy API's.*
    - Need to integrate security properties into PL assertions.

- Disruptive transitions
  - Forget RPC (and threads): the world is asynchronous.
  - Forget type systems as we know them.
  - Forget trusting anything non-local.

# Flow Integration

- Wish: Wouldn't it be nice to hide concurrency from programmers?
  - SQL does it well
  - UI packages do it fine (mostly single-threaded!)
  - RPC does it ok
  - But we are moving towards more asynchrony, I.e. towards more visible concurrency (e-commerce scripts and languages, web services, etc.)

    *You can hide all concurrency some of the time,*
    *and you can hide some concurrency all the time,*
    *but you can't hide all concurrency all the time.*

  - Asynchronous message-based concurrency does not fit easily with more traditional shared-memory synchronous concurrency control

- Goal: make concurrent flows available and checkable at the language level.

# Data Integration

- Wish: Wouldn't it be nice to "program directly against the schema" in a well-typed way?
  - PL data has traditionally been "triangular" (trees), while persistent data has traditionally been "square" (tables)
  - This has caused integration problems: the “impedance mismatch” in data base programming languages
  - Now, persistent data (XML) is triangular too!
  - However, the type systems for PL data (based on tree matching) and XML (based on tree automata) are still deeply incompatible

- Goal: make semistructured data easily available and checkable at the language level.

# Protection Integration

- Wish: Wouldn't it be nice to have automatic security?
  - It's an applet. Sits is a sandbox. End of story. (?)
  - Ok, what about *semi-automatic* security? Explicitly grant/require permissions. (Stack walking etc.)
  - Leads to "sophisticated" access models that programmers do not understand reliably.
- Security today: obscure mechanisms to prevent *something* from happening.
  - It is usually not clear what security mechanisms are meant to *achieve*.
  - Need to move towards *declarative* security and privacy interfaces and policies.
- Goal: make protection policies available and checkable at the language level.

# Language Reliability

- Whether or not we merge new programming models into PLs, we need analysis tools for these new situations
  - Flow: e.g.: behavioral type/analysis system
    - "Does the program respect the protocol?"
  - Data: e.g.: semistructured type/analysis systems
    - "Does the program output match the schema?"
  - Protection: e.g.: information-flow type/analysis system
    - "Does the program defy policy or leak secrets"

- Analysis tools are critical for software reliability
  - Getting it right without assistance is just too hard.
  - These technologies need to be developed in any case, and is better if they can be incorporated in programming languages.

# A Personal Agenda

- Flows [exploit join calculus]
  - Synchronization chords
  - Cω (f.k.a. Polyphonic C#)
- Data [exploit spatial logics as types]
  - Description logics
  - Cω (f.k.a. Xen/X#)
- Protection [exploit π-calculus-style restriction]
  - Flows: Secrecy and Group Creation
  - Data: Trees with hidden labels

# Flows

# Language Support for (WAN) Distribution

- Distribution ⇒ concurrency + latency
  ⇒ asynchrony
  ⇒ more concurrency
  - Approaches: Message-passing, event-based programming, dataflow models, etc.
  - Languages: coordination (orchestration) languages, workflow languages, etc.

- Good language support for asynchrony
  - Make invariants and intentions more apparent (part of the interface), because:
    - It's good software engineering
    - Allows the compiler much more freedom to choose different implementations
    - Also helps other tools

# Cω Flows

- An extension of the C# language with new concurrency constructs

- Based on the join calculus
  - A foundational process calculus like the π-calculus but better suited to asynchronous, distributed systems.
  - First applied to functional languages (JoCaml).
  - It adapts remarkably well to o-o classes and methods.

- A single model that works for
  - Local concurrency (multiple threads on a single machine).
  - Distributed concurrency (asynchronous messaging over LAN or WAN).
  - With no distributed consensus.

- It an unusual model. But it's also a simple extension of familiar o-o notions.
  - No threads, no locks, no fork, only *join*.

# In one slide:

- Client Side (method invocation)
  - Objects have both synchronous and *asynchronous* methods.
  - If the method is synchronous, the caller blocks until the method returns some result (as usual).
  - If the method is async, the call completes at once and returns void (as in message passing).

- Server Side (class definition)
  - A class defines a collection of *chords* (method synchronization patterns), which define what happens once a particular *set* of methods have been invoked. One method may appear in several chords.
  - When enough pending method calls match a chord pattern, the chord body runs. If there are several matches, an unspecified chord is selected.
  - Each chord can have *at most* one synchronous method (providing the *result*). A chord containing *only* asynchronous methods effectively forks a new thread.

# A simple unbounded buffer

```
class Buffer {
  String get() & async put(String s) {
      return s;
  }
}
```

# A simple unbounded buffer

```
class Buffer {
   String get() & async put(String s) {
      return s;
   }
}
```

- An ordinary (synchronous) method header with no arguments, returning a string

# A simple unbounded buffer

```
class Buffer {
  String get() & async put(String s) {
     return s;
  }
}
```

- An ordinary (synchronous) method header with no arguments, returning a string
- An asynchronous method header (hence returning no result), with a string argument

# A simple unbounded buffer

```
class Buffer {
  String get() & async put(String s) {
      return s;
  }
}
```

- An ordinary (synchronous) method header with no arguments, returning a string
- An asynchronous method header (hence returning no result), with a string argument
- Joined together in a chord with a single body

# A simple unbounded buffer

```
class Buffer {
  String get() & async put(String s) {
      return s;
  }
}
```

- Calls to put() return immediately (but are internally queued if there's no waiting get()).
- Calls to get() block until/unless there's a matching put()
- When there's a match the body runs, returning the argument of the put() to the caller of get().
- Exactly which pairs of calls are matched up is unspecified.

# A simple unbounded buffer

```
class Buffer {
  String get() & async put(String s) {
      return s;
  }
}
```

- Does this example involve spawning any threads?
  - No. Though the calls will usually come from different pre-existing threads.
- So is it thread-safe? You don't seem to have locked anything…
  - Yes. The chord compiles into code which uses locks. (And that *doesn't* mean everything is synchronized on the object.)
- Which method gets the returned result?
  - The synchronous one. And there can be at most one of those in a chord.

# Reader/Writer

…using threads and mutexes in Modula 3

An introduction to programming with threads. Andrew D. Birrell, January 1989.

```
VAR i: INTEGER;
VAR m: Thread.Mutex;
VAR c: Thread.Condition;

PROCEDURE AcquireExclusive();
BEGIN
    LOCK m DO
        WHILE i # 0 DO Thread.Wait(m,c) END;
        i := -1;
    END;
END AcquireExclusive;

PROCEDURE AcquireShared();
BEGIN
    LOCK m DO
        WHILE i < 0 DO Thread.Wait(m,c) END;
        i := i+1;
    END;
END AcquireShared;

PROCEDURE ReleaseExclusive();
BEGIN
    LOCK m DO
        i := 0; Thread.Broadcast(c);
    END;
END ReleaseExclusive;

PROCEDURE ReleaseShared();
BEGIN
    LOCK m DO
        i := i-1;
        IF i = 0 THEN Thread.Signal(c) END;
    END;
END ReleaseShared;
```

**An integer i represents the lock state:**

**-1 ←→ 0 ←→ 1 ←→ 2 ←→ 3 …**

(exclusive) (available) (shared)

# Reader/Writer in five chords

```
public class ReaderWriter {
  private async Idle();

  public void AcquireExclusive() & Idle() {}
  public void ReleaseExclusive() { Idle(); }

  public void AcquireShared() & Idle()   { S(1); }
  public void AcquireShared() & async S(int n) { S(n+1); }
  public void ReleaseShared() & async S(int n) {
    if (n == 1) Idle(); else S(n-1);
  }

  public ReaderWriter() { Idle(); }
}
```

A single private message represents the state:

***none*** **←→ `Idle()` ←→ `S(1)` ←→ `S(2)` ←→ `S(3)` …**

**(exclusive) (available) (shared)**

A pretty transparent description of a simple state machine.
Moreover, the synchronization patters are apparent in the class interface,

# Features

- A clean, simple, new model for asynchronous concurrency
  - Minimalist design – to build whatever complex synchronization behaviors you need
  - Easier to express and enforce concurrency invariants; not “buried in the code” any more
  - Much better than programming reactive state machines by hand (the complier does it for you).
  - Efficiently compiled to queues, automata, match bit-vectors, and thread pools.
  - Compatible with existing constructs, though they constrain our design somewhat

- More transparently exposes the control flows
  - Solid foundations, on which to build analysis tools.
  - And checkable notions of *contracts*.

# Future Trends

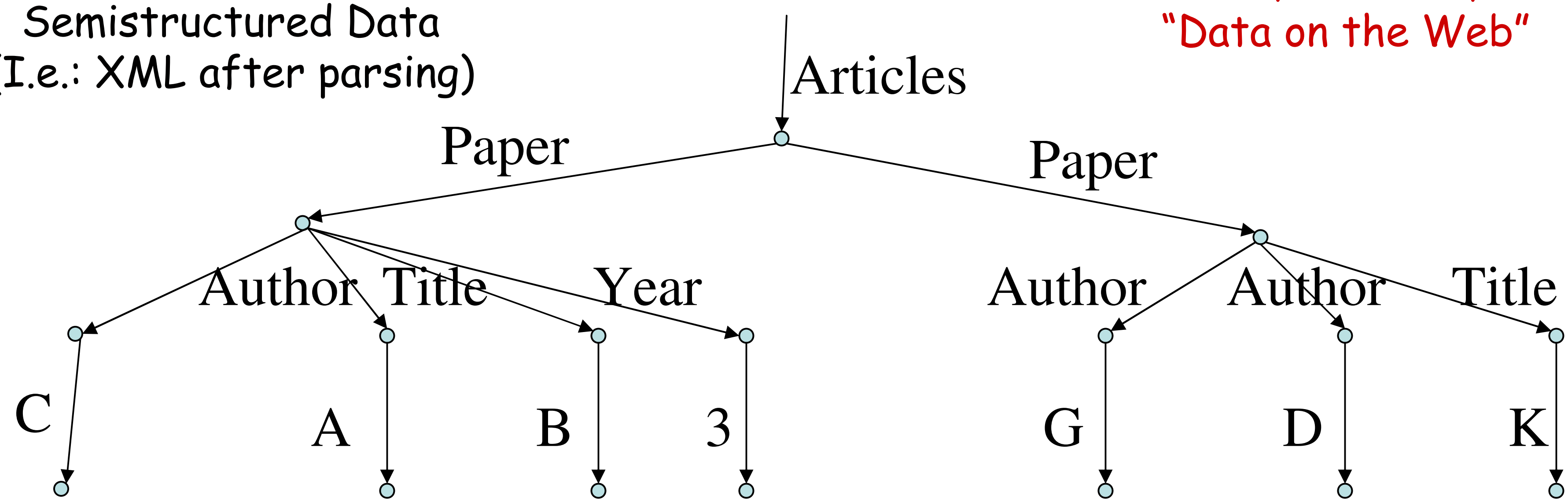
- Protocol contracts
  - Typechecking-style support for checking the interaction of concurrent protocols.
  - A.k.a behavioral type system, session types, etc.
  - Required for software reliability
  - Facilitated by explicit concurrency interfaces.

# Data

# Data

Abiteboul, Buneman, Suciu:
"Data on the Web"

Semistructured Data
(I.e.: XML after parsing)




- *A tree (or graph), unordered (or ordered). With labels on the edges.*
- Invented for "flexible" data representation, for quasi-regular data like address books and bibliographies.
- *Adopted by the DB community as a solution to the "database merge" problem: merging databases from uncoordinated (web) sources.*
- *Adopted by W3C as "web data", then by everybody else.*

# It's Unusual Data

- Not really arrays/lists:
  - Many children with the same label, instead of indexed children.
  - Mixture of repeated and non repeated labels under a node.
- Not really records:
  - Many children with the same label.
  - Missing/additional fields with no tagging information.
- Not really variants (tagged unions):
  - Labeled but untagged unions.
- Unusual data.
  - Yet, it aims to be the new universal standard for interoperability of programming languages, databases, e-commerce...

# Needs Unusual Languages

- New *flexible* types and schemas are required.
  - Based on "regular expressions over trees" reviving techniques from tree-automata theory.
- New processing languages required.
  - Xduce [Pierce, Hosoya], Cduce, …
  - Various web scripting abominations.
- New access methods/query languages required.
  - E.g. Existence of paths through the tree.

# Data Descriptions

- We want to *talk about* data
  - I.e., specify/query/constrain/typecheck the possible structure of data, for many possible reasons:
    - Typing (and typechecking): for language and database use.
    - Constraining (and checking): for policy or integrity use.
    - Querying (and searching): for semistructured database use.
    - Specifying (and verifying): for architecture or design documents.

- A *description* (*spatial formula*) is a formal way of talking about the possible structure of data.
  - We go after a general framework: a very expressive language of descriptions.
  - Combining logical and structural connectives.
  - Special classes of descriptions can be used as types, schemas, constraints, queries, and specifications.

# Example: Typing

Data

*Cambridge*[
  *Eagle*[
    *chair*[**0**] |
    *chair*[**0**]
  ]
]

In Cambridge there is (nothing but) a pub called the Eagle that contains (nothing but) two empty chairs.

data matches description

Description

*Cambridge*[
  *Eagle*[
    *chair*[**0**] |
    **T**
  ] | **T**
]

In Cambridge there is (at least) a pub called the Eagle that contains (at least) one empty chair.

# Example: Queries

With match variables $\mathcal{X}$: *Who is really sitting at the Eagle?*

*Eagle*[
  *chair*[$\neg$**0** $\wedge$ $\mathcal{X}$] |
  **T**
]

*Cambridge*[
  *Eagle*[
    *chair*[*John*[**0**]] |
    *chair*[*Mary*[**0**]] |
    *chair*[**0**]
  ]
]

Yes: $\mathcal{X}$ = *John*[**0**]

Yes: $\mathcal{X}$ = *Mary*[**0**]

With *select-from*:

*from Eagle*[...]
*match Eagle*[*chair*[$\neg$**0** $\wedge$ $\mathcal{X}$] | **T**]
*select person*[$\mathcal{X}$]

Single result:
  *person*[*John*[**0**]] |
  *person*[*Mary*[**0**]]

# Example: Policies

"Vertical" implications about nesting

"Business Policy"

*Borders*[
  *Starbucks*[…] |
  *Books*[…]
]

*Borders*[**T**] ⇒
*Borders*[*Starbucks*[**T**] | **T**]

If it's a Borders,
then it must contain a Starbucks

"Horizontal" implications about proximity

"Social Policy"

*Smoker*[…] |
*NonSmoker*[…] |
*Smoker*[…]

(*NonSmoker*[**T**] | **T**) ⇒
(*Smoker*[**T**] | **T**)

If there is a NonSmoker,
then there must be a Smoker nearby

# Example: Schemas

- Spatial formulas are a "very rich type system". We can comfortably represent various kinds of schemas.

- Ex.: Xduce-like (DTD-like) schemas:

| | | |
|---|---|---|
| $\mathbf{0}$ | the empty tree | |
| $\mathcal{A} \mid \mathcal{B}$ | an $\mathcal{A}$ next to a $\mathcal{B}$ | |
| $\mathcal{A} \vee \mathcal{B}$ | either an $\mathcal{A}$ or a $\mathcal{B}$ | |
| $n[\mathcal{A}]$ | an edge $n$ leading to an $\mathcal{A}$ | |
| $\mathcal{A}^*$ | $\triangleq \; \mu X.\mathbf{0} \vee (\mathcal{A} \mid X)$ | the merge of zero or more $\mathcal{A}$s |
| $\mathcal{A}^+$ | $\triangleq \; \mathcal{A} \mid \mathcal{A}^*$ | the merge of one or more $\mathcal{A}$s |
| $\mathcal{A}^?$ | $\triangleq \; \mathbf{0} \vee \mathcal{A}$ | zero or one $\mathcal{A}$ |

# Future Trends

- Freely mixing logical descriptions and data: spatial logics

$$a[b[A \vee B]] \Rightarrow a[C]$$

  - Fusion of type systems, query languages, policy specifications, etc.
  - Lots of open theoretical problems in this area (typing and subtyping algorithms, decidable sublogics, etc.)

# Cω Data

```
T ::= N
    |  T[]
    |  T(…, T, …)
    |  T|T
    |  T?
    |  T*
    |  […, T m, …]
```

arrays

closures

unions

nullables

streams

tuples (rows)

**Generalized member access (powerful ".")**

# Protection

# Hiding

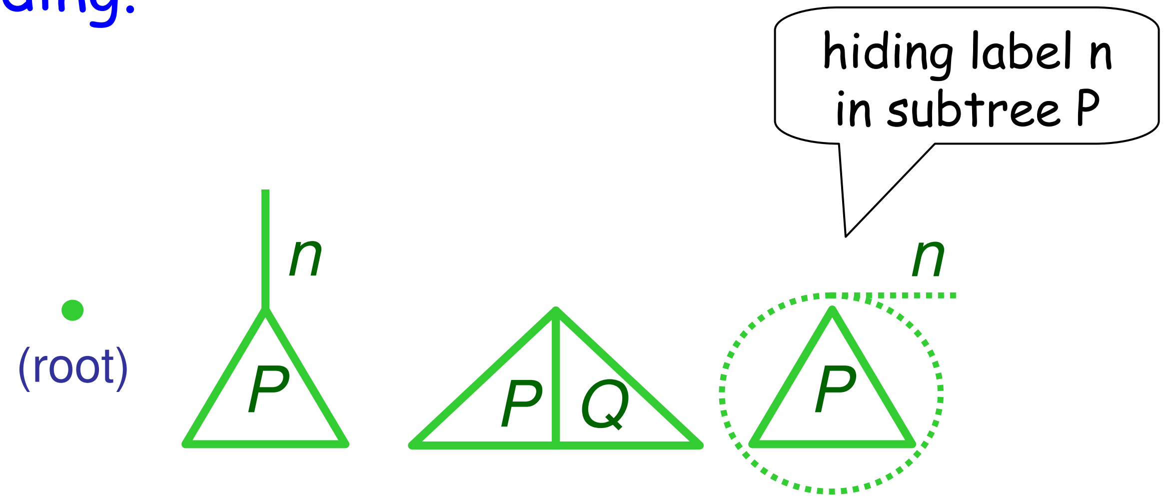
- Any kind of security/privacy issue has to do with hiding something
  - Hiding procedures by access control
  - Hiding data by encryption

- In programming languages:
  - How can we protect/hide flows? (Security)
  - How can we protect/hide data? (Privacy)
  - Exploit the mother of all hiding operators:
    $\pi$-calculus restriction
    (already widely used in crypto protocol analysis).

# Data Protection: Trees with Hidden Labels

- Let's take a fundamental data structure (trees) and add a simple notion of hiding.

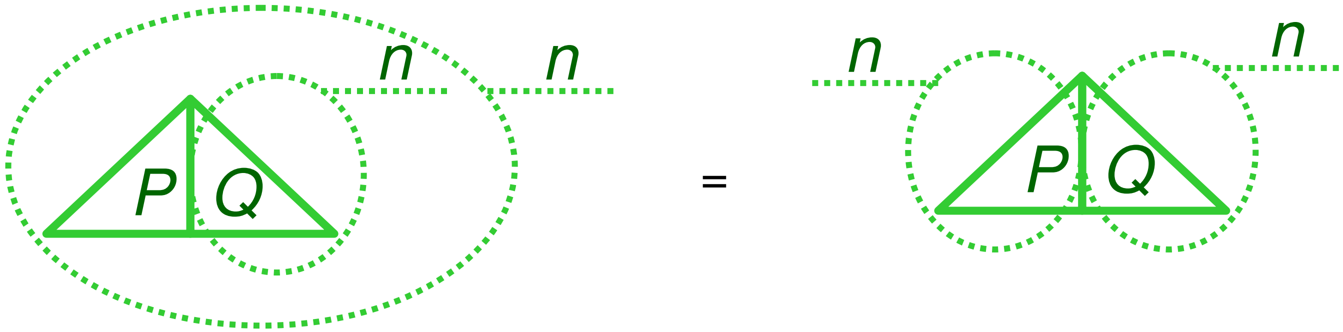
P,Q ::=
0
n[P]
P | Q
(νn)P




- E.g.: Compiler AST's with scoping of identifiers.

# Tree Equivalence (Structural Congruence)

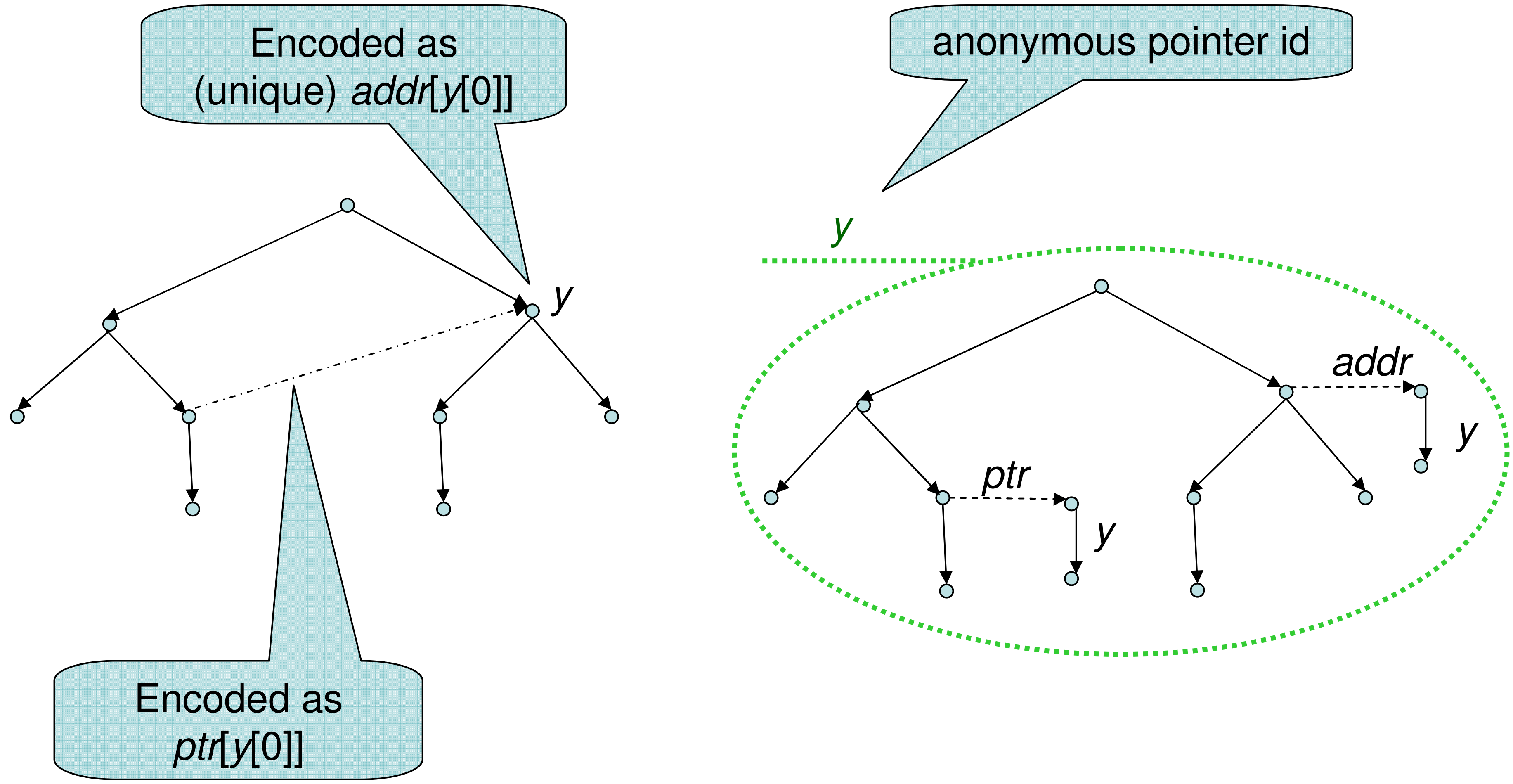
- $(\nu n)(P \mid (\nu n)Q) \equiv ((\nu n)P) \mid ((\nu n)Q)$




- $(\nu n)m[P] \equiv m[(\nu n)P]$ if $n \neq m$

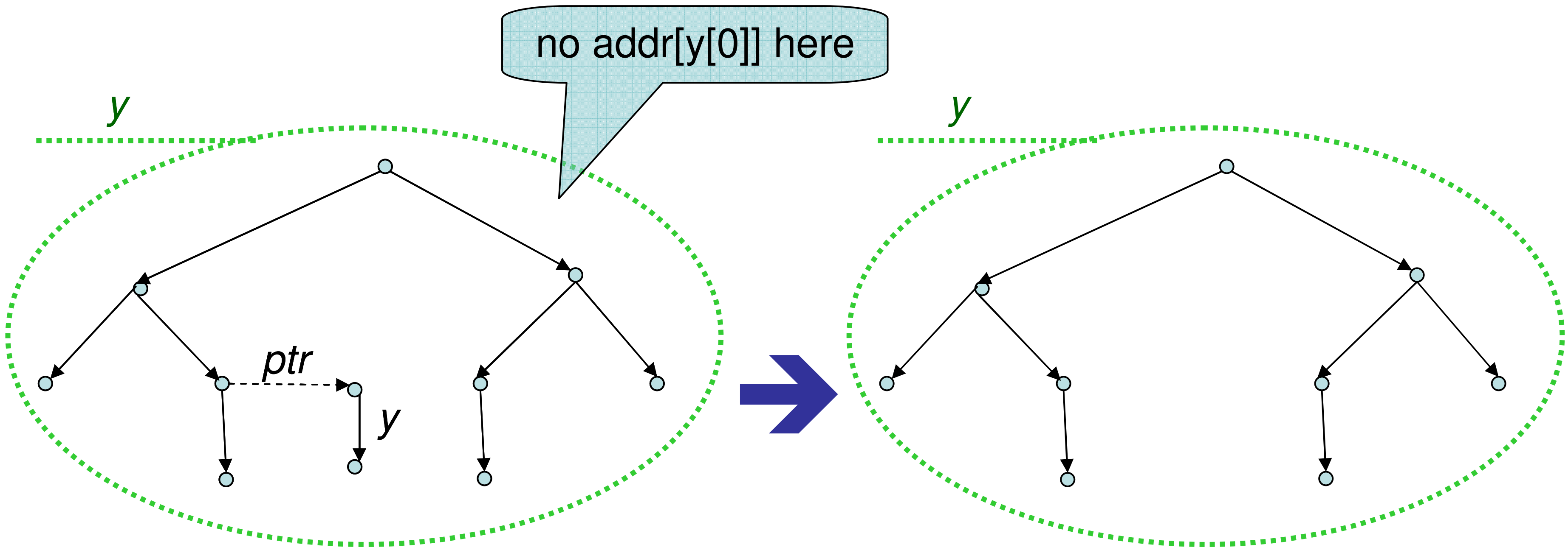
# Ex: Local Pointers

- E.g., XML IDREFs



# Data Manipulation with Hidden Labels

- E.g.: Remove all dangling pointers




- See "Manipulating Trees with Hidden Labels" (Cardelli, Gardner, Ghelli), by pattern matching on such trees.

# Type Systems for Hidden Names

- *myAccount* : Hy. … id[y] … checkbook[y] …

Hiding quantifier

- These are *name-dependent* types
  - Dependent types: traditionally very hard to handle because of computational effects.
  - But dependent only on “pure names”: no computational effects.
  - Name-dependent types are emerging as a general techniques for handling freshness, hiding, protocols (e.g. Vault), and perhaps security/privacy aspects in type systems.

- As a basis for transformation languages
  - Well-typed tree transformations that use and generate hidden labels without violating their “anonymity”.

# Future Trends

- Hiding as a data structure constructor and as a data type quantifier.

# Flow Protection: Group Creation

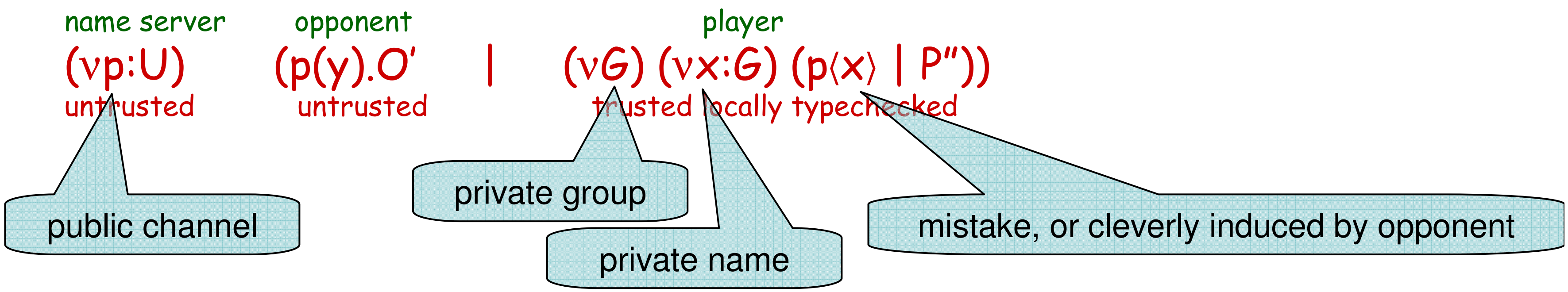
- *Group creation*
  - Create new (hidden) names, but also:
  - Partition them into separate groups, but also:
  - Hide the groups

- It is a natural extension of type systems developed for the π-calculus:
  - (νG) (νn:G) (νm:G) ...
  - create a new group (i.e., type or collection) G, and populate it with new elements n, m, ... (e.g. user-id's, cryptokeys).

- Purpose
  - A secret like n:G could escape on a public channel of type G accidentally or maliciously.
  - But if restricted by (νG), then n:G can never escape from the initial scope of G, as a simple matter of typechecking.

# Untrusted Opponents

- Problem: opponents cheat. Suppose the opponent is untyped, or not well-typed (e.g.: running on an untrusted machine):




- Will an untyped opponent, by cheating on the type of the public channel p, be able to acquire secret information?

- Fortunately, no. The fact that the player is well-typed is sufficient to ensure secrecy, even in presence of untyped opponents. Essentially because $p\langle x\rangle$ must be locally well-typed.

- We do not even need to trust the type of the public channel p, obtained from a potentially untrusted name server.

# Secrecy Guarantee

- Programmer's reference manual:

  Names of group G remain secret, forever, outside the initial scope of (νG).

- Secrecy Theorem (paraphrased)

  If (νG)(νx:…G…)P is well-typed, then P will not leak x even to an untyped (untrusted) opponent.

# Future Trends

- Programming constructs that help enforce security properties by typechecking or analysis.

# Conclusions

# WAN Flows, Data, Protection

- New languages
  - Language evolution is driven by wishes.
  - Language adoption is driven by needs.

- We now *badly need* evolution in areas related to WAN-programming for non-experts (i.e. with language support).
  - Concurrent flows.
    - Applications of Join Calculus.
  - Semistructured data.
    - Applications of Spatial Logics.
  - Flow and data protection.
    - Applications of $\pi$-calculus restriction.

- Why all these calculi/logics/blah things?
  - Formal methods are no longer a complete luxury, in a new world where we increasingly need to *guarantee* good behavior.

# References

- Flows
  - **Join Calculus**: Fournet *et al.*
  - **PolyphonicC#/** **Cω** Benton, Cardelli, Fournet.
  - **Behave!**: Larus *et al.* **Vault**: DeLine *et at.*
  - + Kobayashi, Honda, Yoshida, Vasconcelos, …
- Data
  - **Xen/Cω** Meijer, Bierman *et al.*, http://www.research.microsoft.com/Comega/
  - **TQL**: Cardelli, Ghelli *et al.*
- Protection
  - **Fresh-ML**: Pitts *et al.*
  - **Secrecy and Groups**: Cardelli, Ghelli, Gordon.
  - **Trees with Hidden Labels**: Cardelli, Gardner, Ghelli.
  - + Type systems for security (many).

www.research.microsoft.com/Comega/