

Transitions in Programming Models

Luca Cardelli

Microsoft Research

Significant Transitions

- **Programming languages (PLs)**
 - They evolve slowly and occasionally
 - 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

New Programming Models

- **We are in the middle of a transition in programming models (and eventually PLs)**
 - **More radical than C to C++**
 - Brought more robust data structures (objects)
 - **More radical than C++ to Java**
 - Brought more robust control flows (strong typing)
- **We now have a Cambrian explosion of programming models.**
 - **Lots of badly misshaped things are going to evolve before architectures settle down.**
 - **What's on the other side of the transition?**

Transitions in 3 (related) areas

- **A new emphasis on computation on WANs**
 - **Wide area data integration**
 - XML is “net data”. *XML API's.*
 - Need to integrate this new data into PL data structures.
 - **Wide area flow integration**
 - Messages nor RPC, schedules not threads. *Messaging API's.*
 - Need to integrate these new flows into PL control constructs.
 - **Wide area security integration**
 - Access control, data protection. *Security and privacy API's.*
 - Need to integrate security properties into PL assertions.
- **Impact**
 - **Disruptive transitions: not easy to convert these API's into extensions of existing PLs.**
 - **Ideal topics for research.**

Data Integration

- 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 huge integration problems, known as the "impedence mismatch" in data base programming languages
 - Now, *BIG NEWS*, persistent data (XML) is triangular too!
 - New opportunity for PL integration
 - However, the type systems for PL data (based on tree matching) and XML (based on tree automata) are still deeply incompatible

Flow Integration

- **Wouldn't it be nice to hide concurrency from programmers?**
 - SQL does it well
 - UI packages do it fine
 - RPC does it ok
 - **But we are moving towards more asynchrony, i.e. towards more visible concurrency (e-commerce scripts and languages, 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**

Security Integration

- **Wouldn't it be nice to have automatic security?**
 - It's an applet. Sits in a sandbox. End of story.
 - Ok, what about *semi-automatic* security? Explicitly grant/require permissions. (Stack walking etc.)
 - Leads to emerging “sophisticated” access models that programmers do not understand reliably.

How to Integrate Transitions

- **New programming models often require new kinds of analysis.**
 - **Domain Specific Languages: PLs equipped with specialized analysis for specific programming models**
 - **E.g. SQL (both data and concurrency optimization), security policy languages**
- **But some transitions go beyond DSL's**
 - **C++ was not just a DSL for objects, and Java was not just a DSL for type safety**
 - **Some transitions really require new “general-purpose” languages**
 - **We need more than an XML DSL, a messaging DSL, a security DSL**

Reliability

- **Whether or not we merge new programming models into PLs, we need analysis tools for these new situations**
 - **Data: e.g.: semistructured type/analysis systems**
 - “Does the program output mach the schema?”
 - **Flow: e.g.: behavioral type/analysis system**
 - “Does the program respect the protocol?”
 - **Security: e.g.: information-flow type/analysis system**
 - “Does the program defy policy or leak secrets”
- **Analysis tools are critical for software reliability**

What can we do about this?

- **Assumptions**
 - The existing situation is extremely messy
 - How many web services have you deployed lately?
 - Those 3 WAN-related transitions in programming models have a high probability of precipitating into new languages for WAN programming
- **Research plan**
 - In view of that, try to make some progress in one or more of those areas

Assorted Language-Related Advanced Activities

(Microsoft-centric)

- **Semistructured Data**
 - TQL, Spatial Data Types ...
 - **MS:<Unnamable>** (Erik Meijer, Wolfram Shulte, Herman Venter, ...)
 - Extends C# with XML-like data types and XML query expressions, integrated with real SQL queries.
- **Concurrent Flows**
 - BPEL, Polyphonic C#, Sharpie, Behavioral Types ...
 - **MS:<Unnamable>** (Greg Meredith, ...)
 - Distributed scheduling language based on π -calculus and linear logic types.
- **Security/Privacy/Protocols**
 - Samoa, Vault ...
 - **MS:Binder** (John DeTreville)
 - A Logic-Based security language

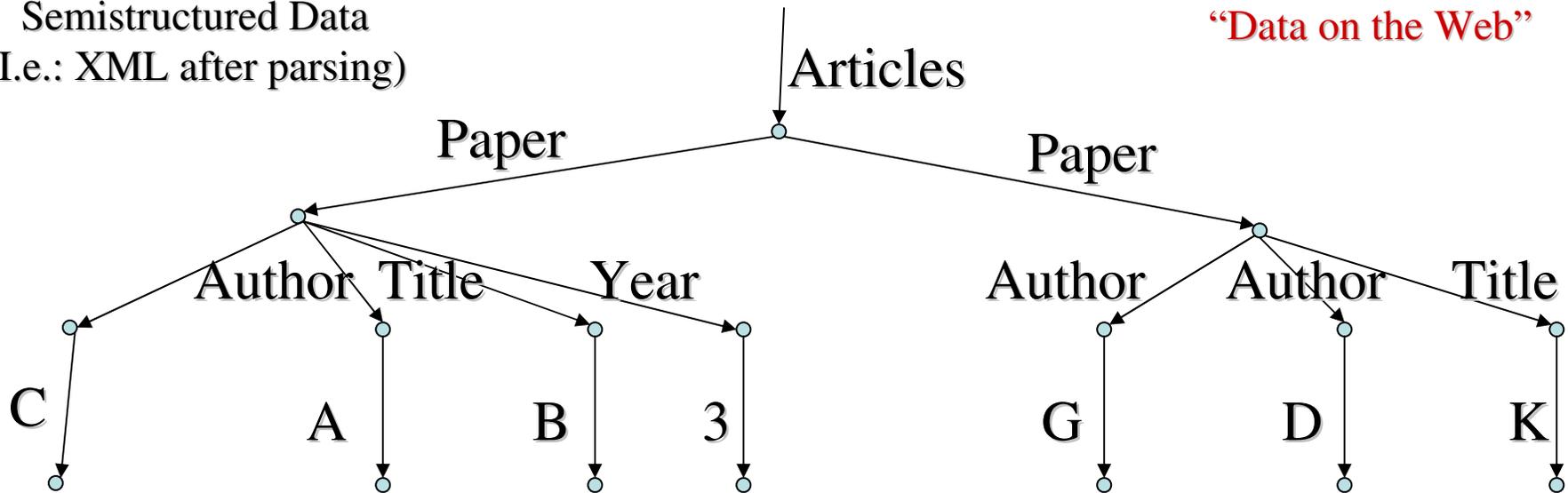
A Personal Agenda

- **Data**
 - Description logics (Spatial Logic)
 - Promising technology: Tree automata
- **Flows**
 - Polyphonic C#
 - Promising technology: Synchronization joins
- **Hiding** (a very small step towards security/privacy)
 - Trees with hidden labels
 - Promising technology: Name-dependent types

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 query languages required. Various approaches:**
 - From simple: Existence of paths through the tree.
 - To fuzzy: Is a tree “kind of similar” to another one?
 - To fancy: Is a tree produced by a tree grammar?
 - To popular: SQL for trees/graphs, for some value of “SQL”.

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* 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]  
  ]  
]
```

Description

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

data matches description

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

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[ $\neg \mathbf{0} \wedge \mathcal{X}$ ] |  
  T  
]
```

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

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

With *select-from*:

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

Single result:

```
person[John[\mathbf{0}]] |  
person[Mary[\mathbf{0}]]
```

Example: Policies

“Vertical” implications about nesting

“Business Policy”

Borders[
 Starbucks[...] |
 Books[...]
]

Borders[**T**] \Rightarrow
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**) \Rightarrow
(*Smoker*[**T**] | **T**)

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

Example: Schemas

- Descriptions are a “very rich type system”. We can comfortably represent various kinds of schemas.
- Ex.: Xduce-like schemas:

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

Current Work

- **Longer-term research:**
 - Powerful languages of data description, based on *spatial logics*. Akin to *description logics* of some time ago, but seen as type systems.
 - Special cases are regular expressions over trees (XML query, etc.)
 - Lots of open problems in this area (typing and subtyping algorithms)

Xen Type System Extensions

(structural)

```
T ::= N
    | T[] | T{ }
    | T(..., T, ...)
    | T|T | T&T
    | T! | T? | T+ | T*
    | [..., T m, ...]
```

arrays

closures

intersection,
union

streams

tuples (rows)

FLOWS

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

Language Support for Concurrency

- **Make invariants and intentions more apparent (part of the interface)**
- **Good software engineering**
- **Allows the compiler much more freedom to choose different implementations**
- **Also helps other tools**

.NET today

- **Java-style “monitors”**
- **OS shared memory primitives**
- **Clunky delegate-based asynchronous calling model**
- **Hard to understand, use and get right**
 - **Different models at different scales**
 - **Support for asynchrony all on the caller side – little help building code to *handle* messages (must be thread-safe, reactive, and deadlock-free)**

Polyphonic C#

- **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
 - It adapts remarkably well to classes and methods.
- **A single model which works both for**
 - local concurrency (multiple threads on a single machine)
 - distributed concurrency (asynchronous messaging over LAN or WAN)
- **It is different. But it's also a simple extension of familiar o-o notions.**

In one slide:

- Objects have both *synchronous* and *asynchronous* methods.
- Values are passed by ordinary method calls:
 - 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*.
- A class defines a collection of *chords* (synchronization patterns), which define what happens once a particular *set* of methods have been invoked. One method may appear in several chords.
 - When pending method calls match a pattern, its body runs.
 - If there is no match, the invocations are queued up.
 - If there are several matches, an unspecified pattern is selected.
 - If a pattern containing *only* *async* methods fires, the body runs in a new thread.

A simple buffer

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

A simple buffer

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

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

A simple buffer

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

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

A simple buffer

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

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

A simple 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 buffer

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

- Does example this 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;
```

Reader/Writer in five chords

```
public class ReaderWriter {
    public void Exclusive() & async Idle() {}
    public void ReleaseExclusive() { Idle(); }

    public void Shared() & async Idle()    { S(1); }
    public void Shared() & 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 \leftrightarrow *Idle()* \leftrightarrow *S(1)* \leftrightarrow *S(2)* \leftrightarrow *S(3)* ...
(exclusive) (available) (shared)

**A pretty transparent description of a simple state machine,
as it should be.**

Features

- **A clean, simple, new model for asynchronous concurrency in C#**
 - Declarative, local synchronization
 - Model good for both local and distributed settings
 - Efficiently compiled to queues and automata
 - Easier to express and enforce concurrency invariants
 - Compatible with existing constructs, though they constrain our design somewhat
 - Minimalist design – pieces to build whatever complex synchronization behaviours you need
 - Solid foundations
 - Works well in practice
 - Convenient - much better than programming state machines yourself

Implementation

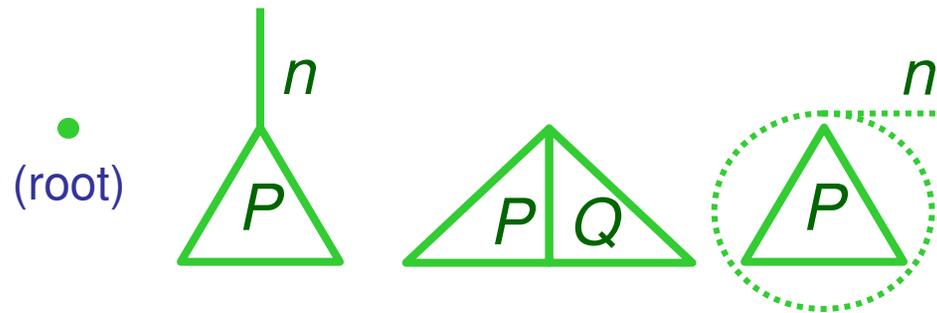
- Translate Polyphonic C# -> C#
- Introduce queues for pending calls (holding blocked threads for sync methods, arguments for asyncs)
- Efficient – bitmasks to look for matches

HIDING

- **Any kind of security/privacy issue has to do with hiding something**
 - Hiding information by encryption
 - Hiding information by access control
 - Hiding private data so it does not escape
- **Baby step:**
 - How can we support hidden data in a programming language?
 - N.B.: Hiding *pure names* (passwords/ids) not, e.g., hiding numbers

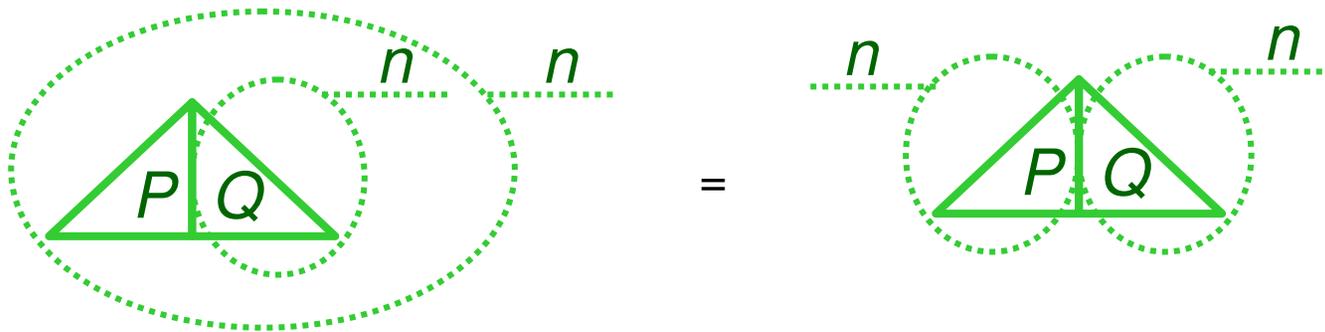
Data Model: Trees with Hidden Labels

$P, Q ::=$
 0
 $n[P]$
 $P \mid Q$
 $(\forall n)P$



Tree Equivalence (Structural Congruence)

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

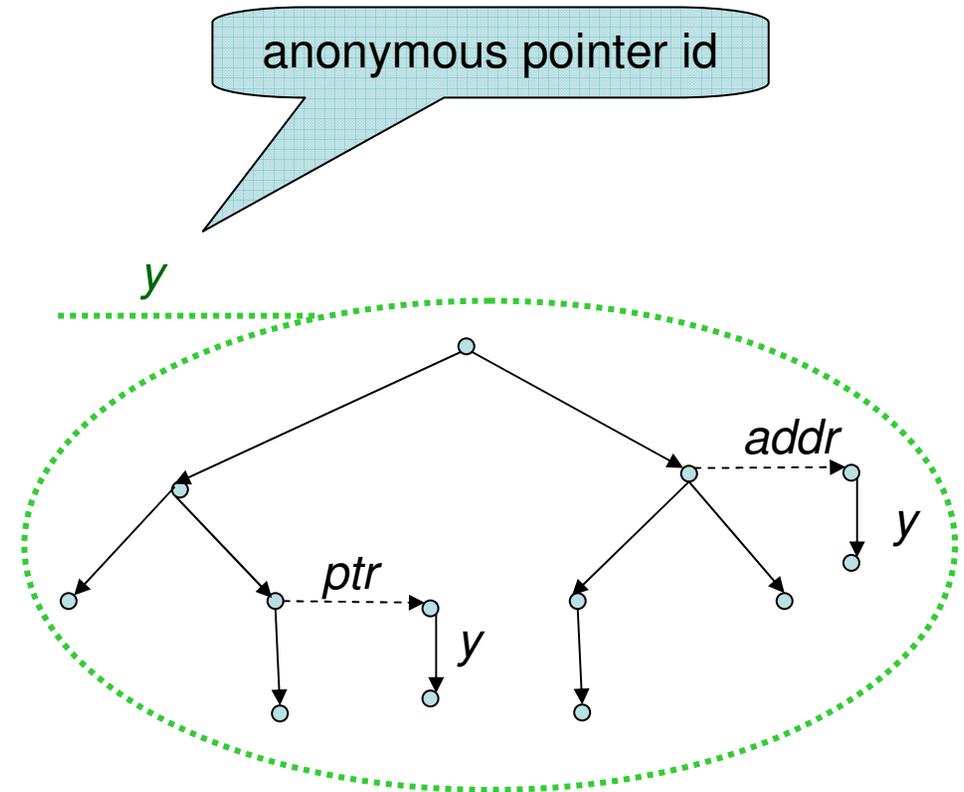
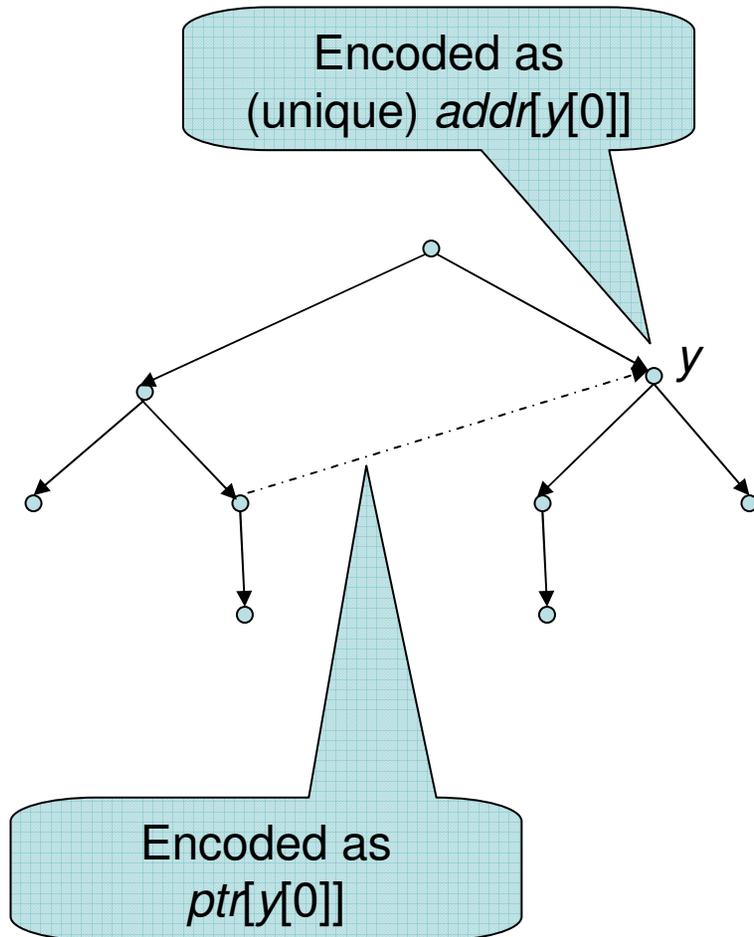


- $(\nu n)m[P] \equiv m[(\nu n)P]$ if $n \neq m$

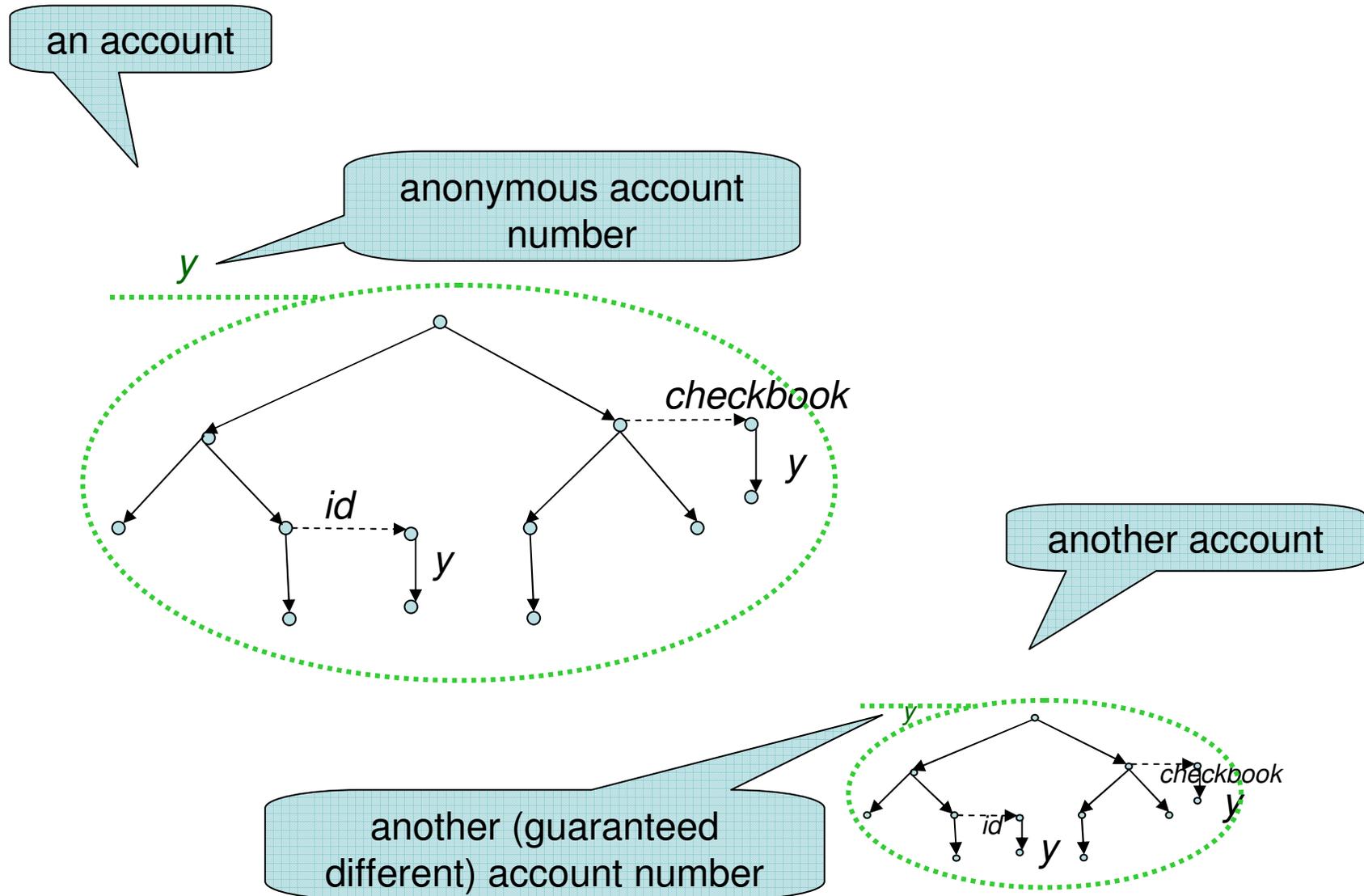


Ex: Local Pointers

- E.g., XML IDREFs



Ex: Unique and Unguessable IDs



Type Systems for Hidden Names

- *account* : Hy. ... *id*[y] ... *checkbook*[y] ...
- 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.

Conclusions

- **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.**
 - Lots of inelegant need-driven *hacks*.
 - Some interesting *designs* here and there.
 - Let's put them together into *languages* that are useful for wide-area programming!

References

- **Data**
 - Meijer *et al.*: Xen
 - Cardelli, Ghelli *et al.*: TQL
- **Flows**
 - Fournet *et al.*: Join Calculus
 - Benton, Cardelli, Fournet: Polyphonic C#
 - Larus *et al.*: Behave!
- **Hiding/Freshness**
 - Pitts *et al.*: Fresh-ML
 - Cardelli, Gardner, Ghelli: Manipulating Trees with Hidden Labels.
 - DeLine *et al.*: Vault

(See personal web pages or search engines.)