

Persistence in languages

Amer Diwan

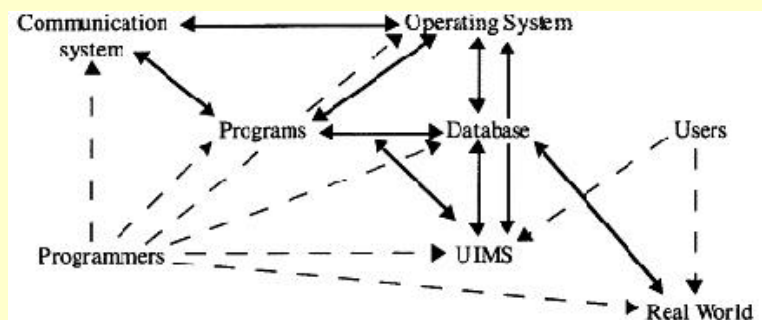
Persistent application systems

- Persistent application systems
 - Application outlives its individual components and even its implementation technology
 - e.g., large databases
- Challenge:
 - Built by “gluing” together different technologies
 - But what if the technologies are as complex as databases and programming languages?

A motivating example

- A hospital database where data “disappeared”
- Attempted solution
 - Roll back database to an earlier point in time when data was “okay”
 - But it didn’t work
- Why?
 - The problem was with fonts which did not get “rolled back”

Typical structure of a PAS



Difficulty with writing PAS

- Writing PAS is hard because the programmer needs to interact with many different subsystems
 - A checkpoint must save data in all relevant subsystems
 - Duplicate functionality between different subsystems
 - Individual subsystems may evolve independently

A solution:

Persistent programming languages

- Build a unified system that delivers a *complete computational environment*
 - Doesn't imply monolithic
- An *orthogonally persistent programming language* supports persistent data
 - attempts to integrate databases with programming languages

What is persistence w.r.t. data?

- “...*supporting data for their full lifetime however long or short it may be*”
- Traditional languages support “transient data”
 - local variables: a static scope
 - global and heap data: full execution of the program
- Database languages (e.g., SQL) support
 - data that outlives the execution of programs
 - data may be around even for many versions/instances of supporting applications

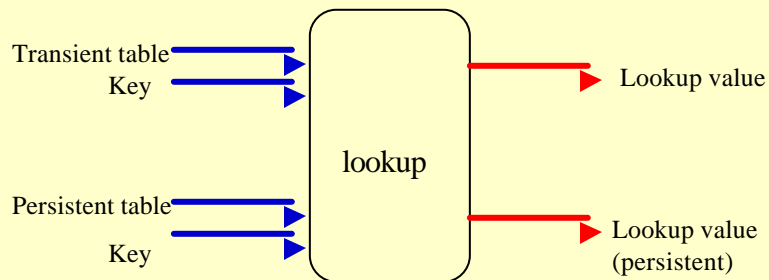
Getting persistent data in traditional programming languages

- Use files
 - Programmer needs to manage file formats, getting things from files, writing to files, ...
 - What if data in a file wants to reference data in another file?
- Make calls to a database system
 - Programmer needs to deal with two worlds of data: that with programming language types and that with database types

Orthogonal persistence principles

- **Orthogonal persistence:** any data can be persistent and the code that manipulates this data does not need to be aware of the distinction
- **Principles**
 - Persistence independence
 - Data type orthogonality
 - Persistence identification
- **Violation of any principle breaks orthogonality**

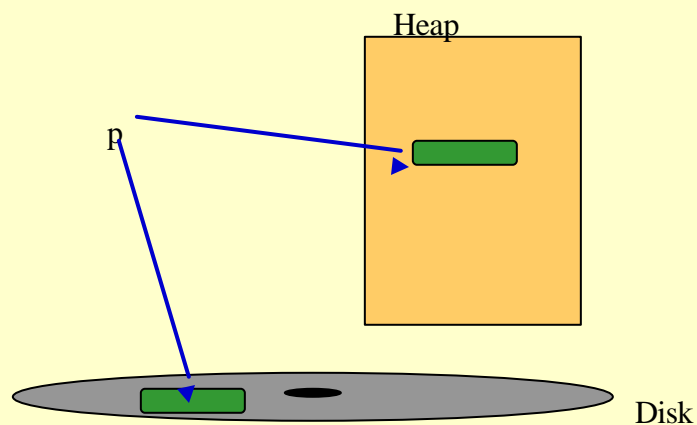
Principle of persistence independence



Advantages of Principle of persistence independence

- Code reuse
- User does not need to manage moving data between persistent and transient forms
- Simplicity in programming

Principle of data-type orthogonality

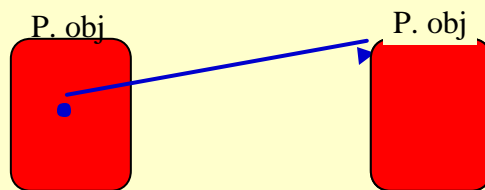


Advantages of Principle of data-type orthogonality

- “Type reuse”
- Simplicity
 - Can a transient object point to a persistent object?
 - Can a persistent object point to a transient object?

Principle of persistence identification

How does one specify whether an object is persistent or transient?



Advantages of Principle of persistence identification

- **Simplicity**
 - Doesn't need special constructs to mark what is persistent and what is not
- **Predictable**
 - No “dangling” pointers

Summary of principles

- Principles simplify a persistent programming language but complicate implementation
 - **P. Independence**: compiler/run-time system needs to automagically transfer objects
 - **D. T. Orthogonality**: compiler/run-time system needs to be prepared that any reference could access a persistent object
 - **P. Identification**: needs gc-like support

Persistent type systems

- Type checking models are quite different in traditional programming languages and database systems
 - Traditional prog. langs. try to do as much checking as possible statically
 - Most of the type-checking in database like applications is “dynamic”

Examples of type checking in databases

- type Address is record ...
var fileDescriptor: file[Address]
 - Checks at file open time
- R1 Join(SSN=ESSN) R2
 - The checking for whether or not SSN and ESSN exist in the relations may happen at run time

Avoiding the straitjacket of full “static checking”

- Use polymorphism
- ?? Key. ? Value.
(Key x Value x List[Key x Value] -> List[Key x Value])
- Polymorphism helps but it isn't enough

“Polymorphism” with relations

- R1 Join(SSN=ESSN) R2
- R1 Join(DoctorName=Name) R2
- The algorithms that are used for the two joins may be completely different
 - The best algorithm depends exactly on what R1 and R2 look like
 - What is the “type” of Join?

A persistent prog language should support dynamic typing

The “any” type in Napier88

- **type** Address **is record** (name: **string**; age: **int**; gender: **bool**)
- **let** ps = PS()
project ps **as X onto**
Address: write X (age)
...
default:
- PS returns an “any”. Need to “coerce” it into correct type before you can do anything meaningful

Partial specification of types

- **type** Address **is record** (name: **string**; age: **int**; gender: **bool**; extra: **any**)
- **let** ps = PS()
project ps **as X onto**
Address:
let this = X(extra)
type extraInfo **is record**(idNo: **int**; spouse: Address)
project this **as Y onto**
extraInfo: ...
default: ...
end
end

Dependent types

- In databases, it may be useful to have a type that depends on a value
- E.g., a search routine may be part of the “type” of a database

Along with types, code should also be “persistent data”

Binding mechanisms

- A binding associates properties with a name
- Dimensions:
 - Mutable or immutable?
 - When is the binding performed?
 - Static or dynamic scoping?
 - When is the type checking performed?

Categorization of binding

	Static R-value	Static L-value	Dynamic R-value	Dynamic L-value
Static typing Static scoping	1 (const)	5	9	13 (vars)
Static typing Dynamic scoping	2	6	10	14
Dynamic typing Static scoping	3	7	11	15 (smalltalk var)
Dynamic typing Dynamic scoping	4	8	12	16 (file names)

Non-traditional operations on bindings

- The need to break bindings
 - e.g., when shipping code
- Separation of names and values in persistent bindings

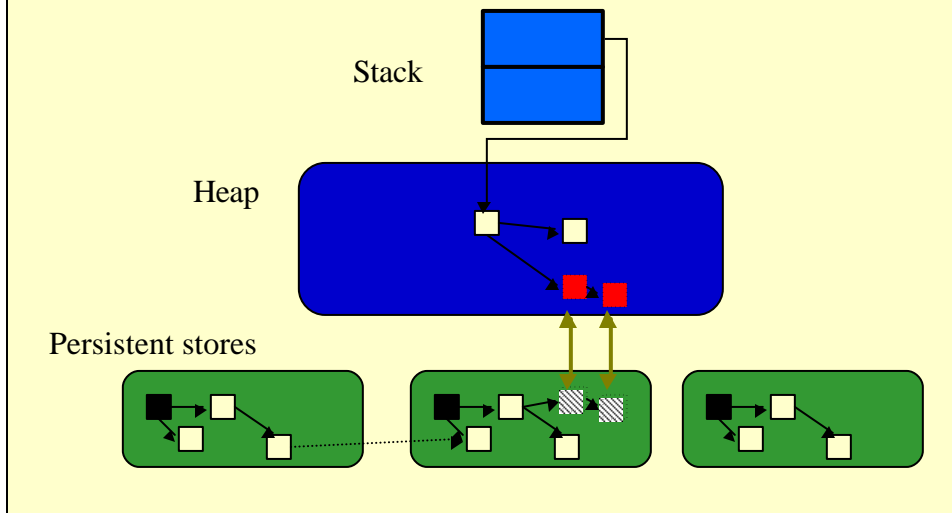
Putting it all together

- Polymorphic programming language +
 - flexible binding mechanism
 - dependent types
 - orthogonal persistence
 - concurrency mechanism
 - ...
- Traditional compiler +
 - support to fetch persistent objects and write them out if needed

Putting it all together (cont.)

- Run-time system +
 - Reachability analysis for persistence
 - Sophisticated strategies for bringing things into memory
 - Sophisticated strategies for writing things out to disk

A picture at run time



Challenges

- **Performance:** Every memory reference could possibly require an “object fault”
- **How and when to move objects to the store**
- **How to garbage collect the store**

Discussion

- Is persistence a replacement for databases?
- Is orthogonal persistence a good thing or would you rather be “in control”?

Next lecture: different models for objects

- Delegation and multiple dispatching
- Readings: Ungar and Smith (on web page)