Advanced Programming Languages in the Enterprise Datacenter

Chet Murthy
IBM Research
Two big ideas

Advanced Programming Language technology is a secret weapon in enterprise computing

Farm where the fertilizer is thickest: Enterprise Systems
Plan of Talk

• Enterprise software
• The problem and opportunity for PL research
• Applying ML and partial evaluation in enterprise software: a case study
• Summary and Future work
Enterprise software systems

- Run our *world*
- Comprise millions of lines of application code
- Written by many thousands of programmers
- Run on sometimes thousands of machines
- Cost many millions of dollars

Names have been changed to protect paying customers
FredCo Bank (2000)

One (slice of one) of the biggest banks' electronic checking system
FredCo Bank (2000)

- One out of ~10 slices of systems is shown
- All slices independently developed
- More “layers” to the left of diagram

- RPCs flow right-to-left, synchronous
- All persistent side-effects reside in DBs
Jeff's Bank (2004)

Another large bank's main client portal

- Legacy Java (00's)
- More Legacy Java
- Mainframe (80's)
- Database Vendor A
- Vendor B
- Vendor C
- Document Management
- Reporting
- Accounting
- Entitlements
- Other
- Directory Server
- TAI
- Portal Server
- Web Server TAI

Java

Java (00's)

Legacy

Mainframe (80's)

Database Vendor A

Vendor B

Vendor C
Jeff's Bank (2004)

- Layers of systems grow by accretion over time (decades)
- Only communication is RPC
Osiris Private Bank (2001) (inside the app-server)

- Request
  - Input handling
  - Data access abstraction
  - Even more object wrappers
  - Business logic
  - Updates
  - Data manipulation/reduction
  - Presentation conversion

- Response

- DB
  - Demarshalling/parsing/validation
  - Object-oriented wrappers for tables
  - Different teams, different frameworks
  - Permissions, tax, currency conversion
  - “sell GM”
  - “current profit”/ “year-to-date”
  - tables, charts, pixel-perfect rendering
Plan of Talk

• Enterprise software
• The problem and opportunity for PL research
• Applying ML and partial evaluation in enterprise software: a case study
• Summary and Future work
“Farm where the fertilizer is thickest” (1)

- Individual layers written by independent teams
- Often written at different times/decades/continents
- Lack of skill/experience results in layer after layer of framework
- Lack of business interest prevents consolidation
- Natural tendency to “wrapper” rather than extend/fix
- Strong functional interfaces separate components
- Side effects in DBs, not program variables
- Dynamic languages, static code
“Farm where the fertilizer is thickest” (2)

- Component and network interfaces are referentially transparent positions
- The “components” are externally “functional”
- Late-stage large-grain optimization is feasible
This should look familiar
And indeed it is ....

- Combinational logic is “functional”
- DIP sockets are referentially transparent positions
- State change via register update
- FP, Haskell, HOL ... for hardware
- Components are externally “functional”
- Nodes and layers are referentially transparent positions
- Transactions' side-effects all in DB
- *FP for the enterprise?*

All the reasons pure functional technology was good for describing circuitry should apply to these systems
Plan of Talk

• Enterprise software
• The problem and opportunity for PL research
• Applying ML and partial evaluation in enterprise software: a case study
• Summary and Future work
An experimental demonstration
Putting FP to work

- Find candidate “component” of an application
- Replace component with a pure functional implementation
- Show this replacement is more efficient
- Go further, replace more, make it even faster, even simpler

Subsystem is XSL
Replace with ML
The XSL language

- EXtensible Stylesheet Language
- Simple dynamically-typed functional language
  - Often dynamically compiled
- Data is all trees (XML)
  - Processors often use universal datatype (cf. LISP s-expressions)
- Usually statically typable
- Type system is remarkably ML-like
- Invariably embedded in a larger server application
- Almost all server-side uses are static code
**Example Stylesheet**

- XSL stylesheet takes in a list of (model, year, accessory), and outputs a list sorted by model, and by year, of accessories.
- Not beautiful, not useful, just a simple motivating example.

<table>
<thead>
<tr>
<th>Model</th>
<th>Year</th>
<th>Accessory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prelude</td>
<td>1998</td>
<td>Tires</td>
</tr>
<tr>
<td>Prelude</td>
<td>1998</td>
<td>Mufflers</td>
</tr>
<tr>
<td>Prelude</td>
<td>1998</td>
<td>Heater Motor</td>
</tr>
<tr>
<td>Prelude</td>
<td>1999</td>
<td>Tires</td>
</tr>
<tr>
<td>Prelude</td>
<td>1999</td>
<td>Mufflers</td>
</tr>
<tr>
<td>Accord</td>
<td>1988</td>
<td>Starter</td>
</tr>
<tr>
<td>Accord</td>
<td>1988</td>
<td>Mufflers</td>
</tr>
<tr>
<td>Accord</td>
<td>1988</td>
<td>Clutch</td>
</tr>
<tr>
<td>Accord</td>
<td>1987</td>
<td>Oil Filters</td>
</tr>
<tr>
<td>Accord</td>
<td>1987</td>
<td>Air Conditioning</td>
</tr>
<tr>
<td>Prelude</td>
<td>1998</td>
<td>Tires, Mufflers, Heater Motor</td>
</tr>
<tr>
<td>Prelude</td>
<td>1998</td>
<td>Tires, Mufflers, Heater Motor</td>
</tr>
<tr>
<td>Prelude</td>
<td>1998</td>
<td>Tires, Mufflers, Heater Motor</td>
</tr>
<tr>
<td>Prelude</td>
<td>1999</td>
<td>Tires, Mufflers</td>
</tr>
<tr>
<td>Prelude</td>
<td>1999</td>
<td>Tires, Mufflers</td>
</tr>
</tbody>
</table>
Input XML DTD and ML type

```xml
<!ELEMENT Output (Row*)>

<!ELEMENT Row (MODEL,YEAR,ACCESSORIES)>

<!ELEMENT MODEL (#PCDATA)>
<!ELEMENT YEAR (#PCDATA)>
<!ELEMENT ACCESSORIES (#PCDATA)>

module Source = struct
  type output = row list
  and row = {model: model; year: year; accessories: accessories}
  and model = string
  and year = string
  and accessories = string
end
```
module Dest = struct
  type output = model list
  and model = name * year list
  and year = date * accessories list
  and accessories = string
  and name = string
  and date = string
end
The Stylesheet

<xsl:stylesheet xmlns:xsl="http://www.w3.org/XSL/Transform/1.0"
xmlns="http://www.w3.org/TR/REC-html40" result-ns=""
indent-result="yes">
<xsl:template match="Output">
<pre><Output>
<xsl:apply-templates select="Row">
<xsl:sort select="MODEL"/>
<xsl:sort select="YEAR"/>
</xsl:apply-templates>
</Output></pre>
</xsl:template>
<xsl:template match="Row">
<xsl:variable name="model">
<xsl:value-of select="./MODEL"/>
</xsl:variable>
<xsl:variable name="year">
<xsl:value-of select="./YEAR"/>
</xsl:variable>

<MODEL name="{$model}">
<YEAR name="{$year}">
PartList>
<xsl:copy-of select="/Output/Row/MODEL[text()=$model]/..
/YEAR[text()=$year]/../ACCESSORIES"/>
</PartList>
</YEAR>
</MODEL>
</xsl:template>
</xsl:stylesheet>

(1) Sort by MODEL
(2) Sort by YEAR
(3) Get MODEL
(4) Get YEAR
(5) Output MODEL and YEAR
(6) Output all ACCESSORIES for that MODEL/YEAR
The ML Program

let transform_output (o:Source.output) =

let transform_row (r:Source.row) =
    let model = r.Source.model in
    let year = r.Source.year in
    (model,
        [(year,
          map_succeed
          (function
              ({Source.model=model';Source.year=year';} as r')
            when model=model' && year=year'
            -> r'.Source.accessories
            | _ -> failwith "caught")
          o)]) in

let sort_by_model_then_year =
    Sort.list (fun r r' -> r.Source.model <= r'.Source.model
                    or r.Source.model = r'.Source.model &&
                      r.Source.year <= r'.Source.year)
                  o in
    ((List.map transform_row sort_by_model_then_year):Dest.output)
What's better about ML?

- Datatype specialized to XML DTD
- Program specialized to types
- Standard FP technology applies
- View types eliminate serialization & parsing
  - XSL often embedded in apps (good)
  - App data translated to XML strings (bad)
  - Parsed back to generic trees (bad)
**Digression: View Types**

Is it a list or an array? Does it matter?

```ocaml
type 'a list = Nil | Cons of 'a * 'a list

module type LIST = sig
  type 'a t
  val inNil : unit -> 'a t
  val inCons : 'a -> 'a t -> 'a t

  val isNil : 'a t -> bool
  val isCons : 'a t -> bool

  val outNil : 'a t -> unit
  val outCons : 'a t -> 'a * 'a t
end
```
A Commercial Realization

(Joint work with Xylem Team)

- Xylem (what is it)
- A real application in a real customer
- What we did & how it went
- Where it's going
The Xylem Intermediate Language

- Simple polymorphic ML
- Simple module system
- Simple optimizations
  - Simplistic reduction and deforestation
  - Data-type specialization
  - View types

Full XSL → Xylem → 100% Pure Java
A real application

- **DB**: Row in DB
- **Java**: XML between middleware layers
- **XSL**: Generate HTML
- **App Server**: Glue together UI

**Diagram Description**:
- **DB** to **Java**: Data Access & business logic (in-memory Java objects)
- **Java** to **XSL**: In-memory XML tree
- **XSL** to **App Server**: In-memory XML string
- **App Server** to **Browser**: HTML page (sent to Web server tier)
The (ultimate) goal

- ~99.9% probability that you have used this app
- 80% of workload at this customer
- Validation in live production system
Xylem 1: a faster XSL

- Xylem + fast parser
- 2x faster than competitor

- Partial evaluation
- Deforestation

Smaller is better

Response time

<table>
<thead>
<tr>
<th>time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.0</td>
</tr>
<tr>
<td>3.5</td>
</tr>
<tr>
<td>3.0</td>
</tr>
<tr>
<td>2.5</td>
</tr>
<tr>
<td>2.0</td>
</tr>
<tr>
<td>1.5</td>
</tr>
<tr>
<td>1.0</td>
</tr>
<tr>
<td>0.5</td>
</tr>
</tbody>
</table>

Incumbent | Xylem 1
Xylem 2: Data structure specialization

- Xylem + fast parser
- Schema-directed datatypes, parsing/deserialization
- 2.8x faster than competitor (represents 30% improvement over Xylem 1)

- Partial evaluation
- Deforestation
- Precise ML datatypes

Response time

Smaller is better
Xylem 3: No parsing at all

- Xylem + fast parser
- Schema-directed datatypes, parsing/deserialization
- 4.3x faster than competitor (represents 44% improvement over Xylem 2)
- Not much left: 0.4ms serialization for a 7k document

- Partial evaluation
- Deforestation
- Precise ML datatypes
- **View types**
Xylem 4: Query Pushdown (future work)

- All preceding optimizations
- **Schema-directed DB access**
- How much faster can it get?

![Diagram showing data flow from DB to HTML page through various transformations](image-url)
What is of note?

- Same runtime, same app-server, same JVM
- Neil Jones: find nontrivial invariants that classical compilers cannot discover
- Immense opportunity: simpler programs, greater performance
- Business software: unique opportunity
- FP technology is the secret weapon
  - Partial evaluation
  - Deforestation
  - Type specialization
  - View types
Outcome of Experiment

- Faster
- Cheaper
- Simpler
- More “robust”

- In production today
- 40% decrease in CPU utilization for first production app

Come for the speed
Stay for the simplicity
Xylem's Future

• Query pushdown, update
• Apply technology to other parts of e-business stack
  – Presentation (portals)
  – RPC (XML-RPC, SOAP) marshallers
  – Workflow (BPEL)
  – Messaging (Java Messaging Service, pub/sub)
Plan of Talk

• Enterprise software
• The problem and opportunity for PL research
• Applying ML and partial evaluation in enterprise software: a case study
• Summary and Future work
Two big ideas

Advanced Programming Language technology is a secret weapon in enterprise computing

Farm where the fertilizer is thickest: Enterprise Systems
Future work

• Streaming, ETL (extract/transform/load)
  – Lazy languages
• Query pushdown
  – Logic programming
• Model/view/controller (MVC) UIs
  – I/O automata, reactive systems
• Code-generation to client (AJAX)
  – Attribute grammars