

Build your own XQuery processor

Mary Fernández, AT&T Labs Research

Jérôme Siméon, IBM T.J. Watson Research Center

Part I

Introduction

Why another talk on XQuery?

- ▶ What you should have learned so far:
 - ▶ What is XQuery?
 - ▶ General XQuery processing principles
 - ▶ XML storage and indexing techniques
 - ▶ XQuery optimization
 - ▶ XQuery on top of a relational system
- ▶ What is missing?
 - ▶ How to put all the pieces together...
 - ▶ ...to build a real XML query engine

Requirements & Technical Challenges

- ▶ Completeness
 - ▶ Complex implicit semantics
 - ▶ Functions & modules
 - ▶ ... many more ...
- ▶ Performance
 - ▶ Nested queries
 - ▶ Memory management
 - ▶ ... many more ...
- ▶ Extensibility
 - ▶ Variety of XML & non-XML data representations
 - ▶ Updates
 - ▶ ... many more ...

Completeness: Implicit Semantics

- ▶ User: W3C Working Group (and all Galax users!)
- ▶ Implicit XPath semantics

`$cat/book[@year > 2000]`

- ▶ Atomization
- ▶ Type promotion and casting
Presence/absence of XML Schema types
- ▶ Existential quantification
- ▶ Document order

Performance: Nested Queries

- ▶ User: IBM Clio Project
Automatic XML Schema to XML Schema Mapping
- ▶ Nested queries are hard to optimize (XMark #10):

```
for $i in distinct-values($auction/site/.../@category)
let $p :=
  for $t in $auction/site/people/person
  where $t/profile/interest/@category = $i
  return
    <personne>
      <statistiques>
        <sexe> { $t/profile/gender/text() } </sexe>
        <age> { $t/profile/age/text() } </age>
        <education> { $t/profile/education/text() } </education>
        <revenu> { fn:data($t/profile/@income) } </revenu>
      </statistiques>
      ....
    </personne>
return <categorie><id>{ $i }</id>{ $p }</categorie>
```

- ▶ Naive evaluation iterates n times over the auction document
- ▶ Recognize as group-by on category and unnest

Extensibility: Querying Virtual XML

- ▶ User: AT&T PADS (Processing Arbitrary Data Streams)
 - ▶ Declarative data-stream description language
 - ▶ Detects & recovers from non-fatal errors in data stream
- ▶ Query Native HTTP Common Log Format (CLF)

```
207.136.97.49 - - [15/Oct/1997:18:46:51 -0700] "GET /turkey/amnty1.gif HTTP/1.0" 200 3013
anx-lkf0044.deltanet.com - - [15/Oct/1997:21:13:59 -0700] "GET / HTTP/1.0" 200 3082
152.163.207.138 - - [15/Oct/1997:19:17:19 -0700] "GET /asa/china/images/world.gif HTTP/1.0" 304 -
```

- ▶ Virtual XML view

```
<http-clf>
  <host>207.136.97.49</host>
  <remoteID>-</remoteID>
  <auth>-</auth>
  <mydate>15/Oct/1997:18:46:51 -0700</mydate>
  <request><meth>GET</meth><req_uri>/turkey/amnty1.gif</req_uri><version>HTTP/1.0 ...
  <response>200</response>
  <contentLength>3013</contentLength>
</http-clf>
```

- ▶ Vet errors in data

```
fn:doc("pads:data/clf.data")/http-clf[contentLength/@pads:errCode]/@pads:loc
```

What is this course about?

- ▶ Teach you how to build a real XQuery engine
 - ▶ How are you sure you implemented the right language?
 - ▶ How to put the techniques you have learned to practice
 - ▶ Focus on how the various techniques interact
- ▶ Explain what matters in practice
 - ▶ How do you apply the “theory” in a real system
 - ▶ How do you make it run fast?
 - ▶ Focus on architecture and implementation issues
- ▶ Teach you enough of Galax’s internal
 - ▶ Learn how to use it
 - ▶ Learn how the code is organized
 - ▶ Learn how to change it

What you need

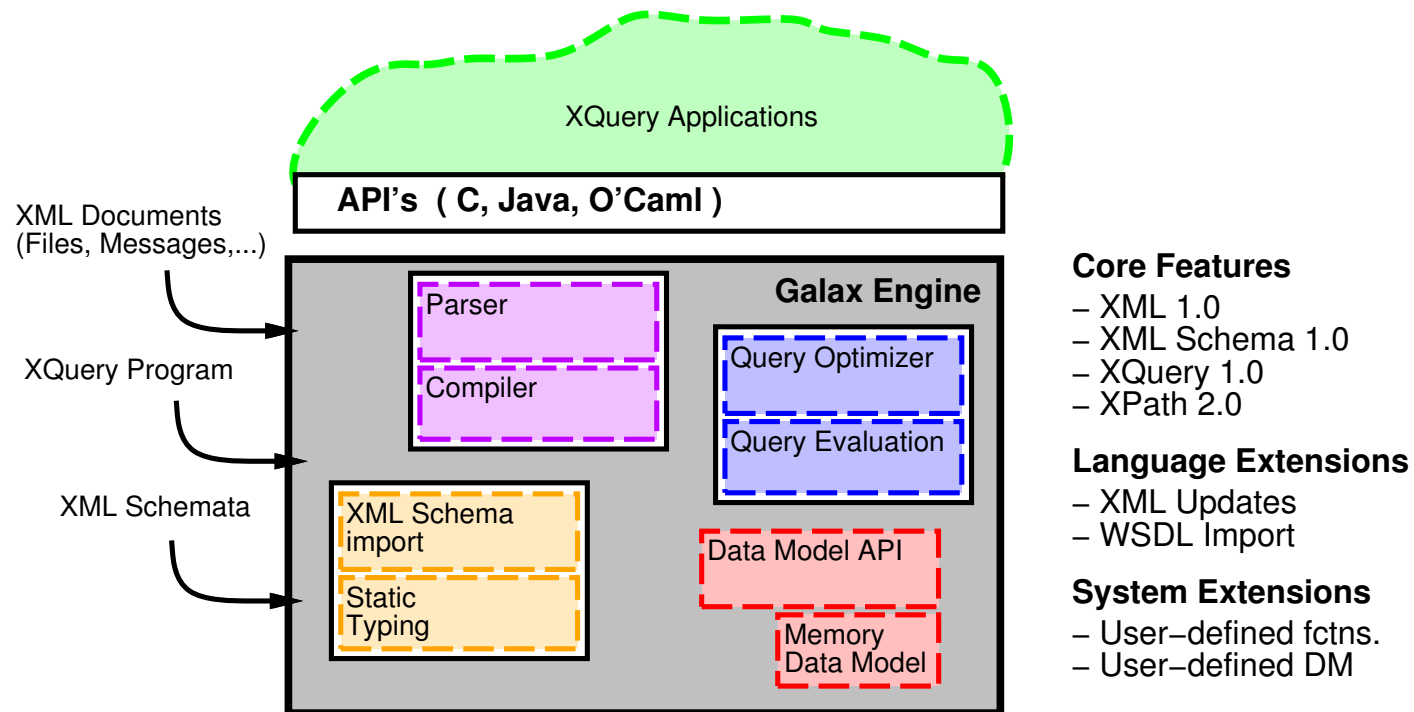
- ▶ This course assumes:
 - ▶ Some user-level knowledge of XML and XQuery
 - ▶ Some minimal programming experience
- ▶ Is also helpful, but not required:
 - ▶ Some knowledge about query processing (e.g., relational)
 - ▶ Some idea of how a typical database engine works

Part II

Preliminaries: Galax and Caml

What is Galax?

- ▶ Complete, extensible, performant XQuery 1.0 implementation
- ▶ Functional, strongly typed XML query language



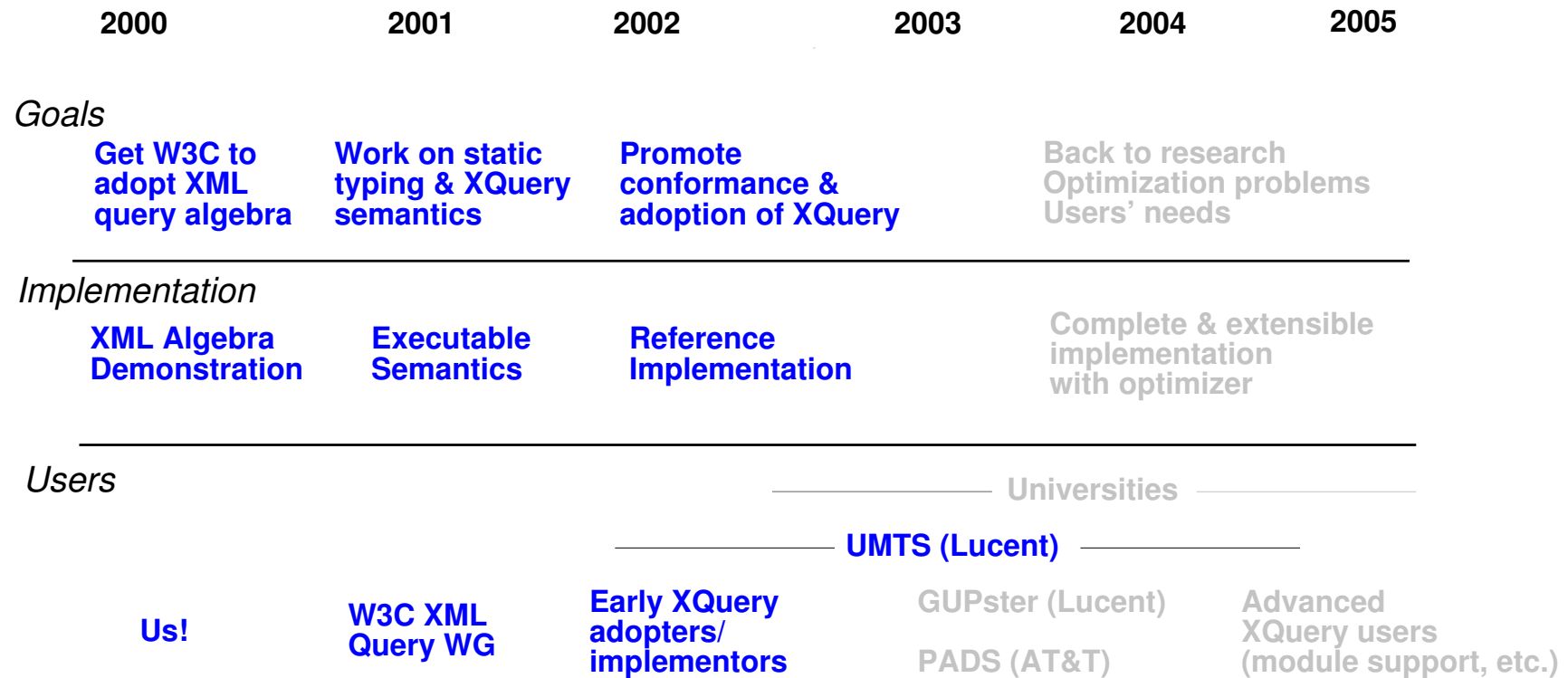
Galax History

	2000	2001	2002	2003	2004	2005
<i>Goals</i>	Get W3C to adopt XML query algebra	Work on static typing & XQuery semantics	Promote conformance & adoption of XQuery		Back to research Optimization problems Users' needs	
<i>Implementation</i>	XML Algebra Demonstration	Executable Semantics	Reference Implementation		Complete & extensible implementation with optimizer	
<i>Users</i>	Us!	W3C XML Query WG	Early XQuery adopters/ implementors	UMTS (Lucent) GUPster (Lucent) PADS (AT&T)	Universities Advanced XQuery users (module support, etc.)	

Galax History

	2000	2001	2002	2003	2004	2005
<i>Goals</i>	Get W3C to adopt XML query algebra	Work on static typing & XQuery semantics	Promote conformance & adoption of XQuery		Back to research Optimization problems Users' needs	
<i>Implementation</i>	XML Algebra Demonstration	Executable Semantics	Reference Implementation		Complete & extensible implementation with optimizer	
<i>Users</i>	Us!	W3C XML Query WG	Early XQuery adopters/ implementors	UMTS (Lucent) GUPster (Lucent) PADS (AT&T)	Universities Advanced XQuery users (module support, etc.)	

Galax History



Galax History

	2000	2001	2002	2003	2004	2005
<i>Goals</i>	Get W3C to adopt XML query algebra	Work on static typing & XQuery semantics	Promote conformance & adoption of XQuery		Back to research Optimization problems Users' needs	
<i>Implementation</i>	XML Algebra Demonstration	Executable Semantics	Reference Implementation		Complete & extensible implementation with optimizer	
<i>Users</i>	Us!	W3C XML Query WG	Early XQuery adopters/ implementors	UMTS (Lucent) GUPster (Lucent) PADS (AT&T)	Universities Advanced XQuery users (module support, etc.)	

Getting Galax

- ▶ The Galax Web site is at:

<http://www.galaxquery.org/>

- ▶ The Galax distributions are at:

<http://www.galaxquery.org/distrib.html>

- ▶ The source distribution is at:

<http://www.galaxquery.org/Downloads/download-galax-0.4.0-source.html>

Installing Galax from the source

// The source distribution

```
simeon@localhost ~/NEW > ls -la
total 1596
drwxrwxr-x    8 simeon   simeon       4096 Sep  1 06:12 .
drwx-----  94 simeon   simeon       8192 Sep  1 06:09 ..
-rw-rw-r--    1 simeon   simeon    1590932 Sep  1 06:12 galax.tar.gz
```

// Un-packing

```
simeon@localhost ~/NEW > cat galax.tar.gz | gunzip | tar xvf -
galax/
galax/Makefile
galax/.depend
galax/BUGS
galax/Changes
galax/LICENSE
.....
simeon@localhost ~/NEW > cd galax
simeon@localhost ~/NEW/galax > ls
algebra      galapi      parsing     ...
analysis     datamodel   jungledm    physicaldm  ...
```

Installing Galax from the source (2)

// Configuring

```
simeon@localhost ~/NEW/galax > cp config/Makefile.unix config/Makefile
simeon@localhost ~/NEW/galax > emacs config/Makefile
```

// Compiling

```
simeon@localhost ~/NEW/galax > make
make world
make[1]: Entering directory '/home/simeon/NEW/galax'
make prepare
make[2]: Entering directory '/home/simeon/NEW/galax'
```

```
*****
* Preparing for compilation *
*****
.....
```

// Installing

```
simeon@localhost ~/NEW/galax > make install
.....
simeon@localhost ~/NEW/galax >
```

Using Galax from the command line

// Run a query

```
simeon@localhost ~ > cat example1.xq
(: Find the 3rd author of the first book :)
doc("book.xml")//book[1]/author[3]
simeon@localhost ~ > galax-run example1.xq
<author>Dan Suciu</author>
```

// Run a query with an input document

```
simeon@localhost ~ > cat example2.xq
(: Find the 3rd author of the first book :)
//book[1]/author[3]
simeon@localhost ~ > galax-run example1.xq -context-item book.xml
<author>Dan Suciu</author>
```

Using Galax from the command line

// Compile a query

```
simeon@localhost ~ > cat example3.xq
(: Find all title of books published in 2000 :)
for $b in doc("xmpbib.xml")//book
where $b/@year = 2000
return $b/title
simeon@localhost ~ > galax-compile -verbose on example3.xq
                                     -print-normalized-expr on
Normalized Expression (XQuery Core):
-----
for $b in (
  fs:distinct-docorder((let $fs:sequence :=
...
    (data(("xmpbib.xml" as item()* )) as atomic()* ),
.....

Optimized Normalized Expression (XQuery Core):
-----
for $b in (doc("xmpbib.xml")...
.....
return
  if (
    boolean(some $fs:v4 ...
```

Using Galax to Validate Documents

// A valid document

```
simeon@localhost ~ > cat book.xml
```

```
<book>
  <title>Data on the Web</title>
  <author>Serge Abiteboul</author>
  ...
  <section>
    <title>Introduction</title>
    ...
```

```
simeon@localhost ~ > galax-parse -xmlschema book.xsd -validate book.xml
```

```
simeon@localhost ~ >
```

// An invalid document

```
simeon@localhost ~ > cat book-err.xml
```

```
<book>
  <title>Data on the Web</title>
  <author>Serge Abiteboul</author>
  ...
  <section ID="intro" difficulty="easy" >
    <title>Introduction</title>
    ...
```

```
simeon@localhost ~ > galax-parse -xmlschema book.xsd -validate book-err.xml
```

```
Validation Error: No matching declaration for attribute ID in content model
```

Using Galax from C or Java

// A main program in Java

```
import galapi.*;
public class Example {
    static void example1 (ModuleContext mc) throws GalapiException {
        ItemList r =
            Galax.evalStatementFromString (mc, "doc(book.xml)//book[1]/author[3]");
        Galax.serializeToStdout (r);
    } // example1()
    public static void main (String args[]) throws GalapiException {
        Galax.init ();
        ProcessingContext pc = Galax.defaultProcessingContext ();
        ModuleContext mc = Galax.loadStandardLibrary (pc);
        example1 (mc);
    } // main()
} // class Example
```

// Compiling it

```
simeon@localhost ~ > javac Example.java
```

// Running it

```
simeon@localhost ~ > java Example
<author>Dan Suciu</author>
```

Galax source distribution (1)

// **Documentation**

- ./Changes
- ./LICENSE
- ./doc
- ./README
- ./TODO

// **Compilation and configuration**

- ./Makefile
- ./config

// **Examples and tests**

- ./examples
- ./usecases
- ./regress

// **Web site and demo**

- ./website

Galax source distribution (3)

```
// Core Galax engine sources
./base           // some basic modules (e.g. I/O)
./fsa            // DFA/FSA library
./namespace      // XML names and namespaces
./datatypes      // XML Schema datatypes
./ast           // Main XQuery AST's
./print         // Pretty printing for the AST's
./dm            // XML data model (virtual)
./procctxt      // XQuery processing context
./rewriting     // AST generic tree walker
./lexing        // XML/XQuery lexing
./parsing       // XML/XQuery parsing
./streaming     // SAX support
./serialization // XML serialization
```

Galax source distribution (4)

```
./monitor      // Compilation monitoring support
./schema       // XML Schema support
./wsdl         // Web services support
./normalization // XQuery normalization
./projection   // Document projection
./datamodel    // Main-memory data model (DOM-like)
./stdlib       // XQuery Functions and Operators
./jungledm     // File indexes
./typing       // Static typing
./cleaning     // ‘‘syntactic’’ rewritings
./analysis     // Static analysis
./compile      // Algebraic compilation
./optimization // Algebraic optimization
./algebra      // Evaluation code
./evaluation   // Evaluation engine
./procmod      // XQuery processing model
```

Galax source distribution (5)

```
// External libraries
```

```
./tools
```

```
// APIs
```

```
./galapi
```

```
// Command-line tools
```

```
./toplevel
```

```
// Galax extensions
```

```
./extensions
```

Caml survival kit (1)

► Caml is a lot like XQuery

```
define function f($x as xs:integer) as xs:integer {  
  if ($x > 0) then $x + 2 else -$x+2  
}
```

```
(: One call to f :)  
let $a := 1 return f($a)
```

==>

```
- : xs:integer = 3
```

► In Caml:

```
# let f x = if (x > 0) then x+2 else -x+2;;  
val f : int -> int = <fun>  
# (* One call to f *)  
  let a = 1 in f(a);;  
- : int = 3
```

Caml survival kit (2)

- ▶ Caml is open source
- ▶ Caml is portable
- ▶ Caml generates efficient code
- ▶ Caml is a functional language (like XQuery)
- ▶ Caml is strongly typed (more than XQuery)
- ▶ Caml supports modules (much more than XQuery)
- ▶ Caml supports imperative feature (like Pascal)
- ▶ Caml supports object-oriented features (like Java)
- ▶ Caml has a good C interface (calling Caml from/to C)
- ▶ Caml is very well suited for program manipulation

`http://caml.inria.fr/`

Caml survival kit (3)

► Function signatures:

```
(* Function taking a boolean and returning a boolean *)
```

```
val not : bool -> bool
```

```
(* Function taking 2 integers returning one integer *)
```

```
val (+) : int -> int -> int
```

```
(* Function taking a norm_context, and expression and  
   returning a core expression *)
```

```
val normalize_expr : norm_context -> expr -> ucexpr
```

Caml survival kit (4)

► Creating types

```
// A new string type (for XML local names)

type ncname = string

// A new choice type (for namespaces prefixes)

type prefix =
  | NSDefaultPrefix
  | NSPrefix of ncname

// A new tuple type (for unresolved QNames)

type uqname = prefix * ncname

// A new record type (for XQuery main modules)

type xquery_module =
  { xquery_prolog      : prolog;
    xquery_statements : statement list }
```

Caml survival kit (5)

- ▶ Modules and compilation
 - ▶ Code organized in files
 - ▶ Each source file = a module
 - ▶ `./normalization/norm_top.ml = Module Norm_top`
 - ▶ Modules can have an Interface
 - ▶ Interfaces can export types, functions, and values
 - ▶ `./normalization/norm_top.mli = Interface for Norm_top`
 - ▶ Modules can access what is exported.

// Calling a function in a module

```
let ctxt = ... in
let expr0 = ... in
Norm_top.normalize_expr ctxt expr0
```

// Opening a whole module

```
open Norm_top
```

```
let ctxt = ... in
let expr0 = ... in
normalize_expr ctxt expr0
```

Caml survival kit (6)

► Compiling Caml code:

// Compile a module to bytecode

```
ocamlc -I INCLUDES... -c ./normalization/norm_top.mli --> norm_top.cmi
ocamlc -I INCLUDES... -c ./normalization/norm_top.ml --> norm_top.cmo
```

// Compile a module to native code

```
ocamlopt -I INCLUDES... -c ./normalization/norm_top.ml --> norm_top.cmx
                                                    + norm_top.o
```

// Compile a main program (native code)

```
ocamlopt -I INCLUDES... -o galax-run ... ./normalization/norm_top.cmx ...
```

Part III

Architecture

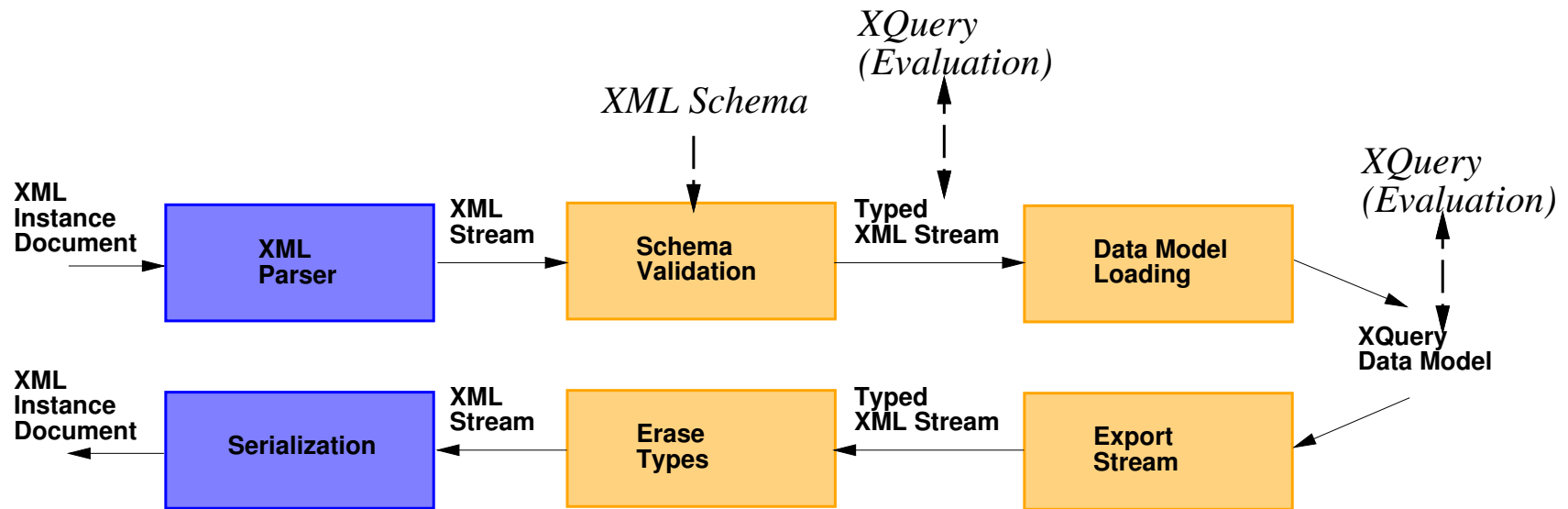
Galax's Architecture

- ▶ Architecture is composed of processing models for:
 - ▶ XML documents
 - ▶ XML schemas
 - ▶ XQuery programs
- ▶ Processing models are connected, e.g.,
 - ▶ Validation relates XML documents and their XML Schemata
 - ▶ Static typing relates queries and schemata
- ▶ **Each processing model based on formal specification**
- ▶ Interfaces between processing models well-defined & strict (e.g., strongly typed)

Architecture References

- ▶ Very little research references
- ▶ General text books:
 - ▶ Traditional DB query compiler books (e.g., Widom's)
 - ▶ Traditional PL compiler books (e.g., Appel's)
- ▶ About XQuery:
 - ▶ XQuery processing model (part of W3C spec)
 - ▶ *"Implementing XQuery by the Book"*, Fankhauser et al, SIGMOD Records.
 - ▶ *"Building an XQuery processor"*, XSym'2004

XML Processing Architecture

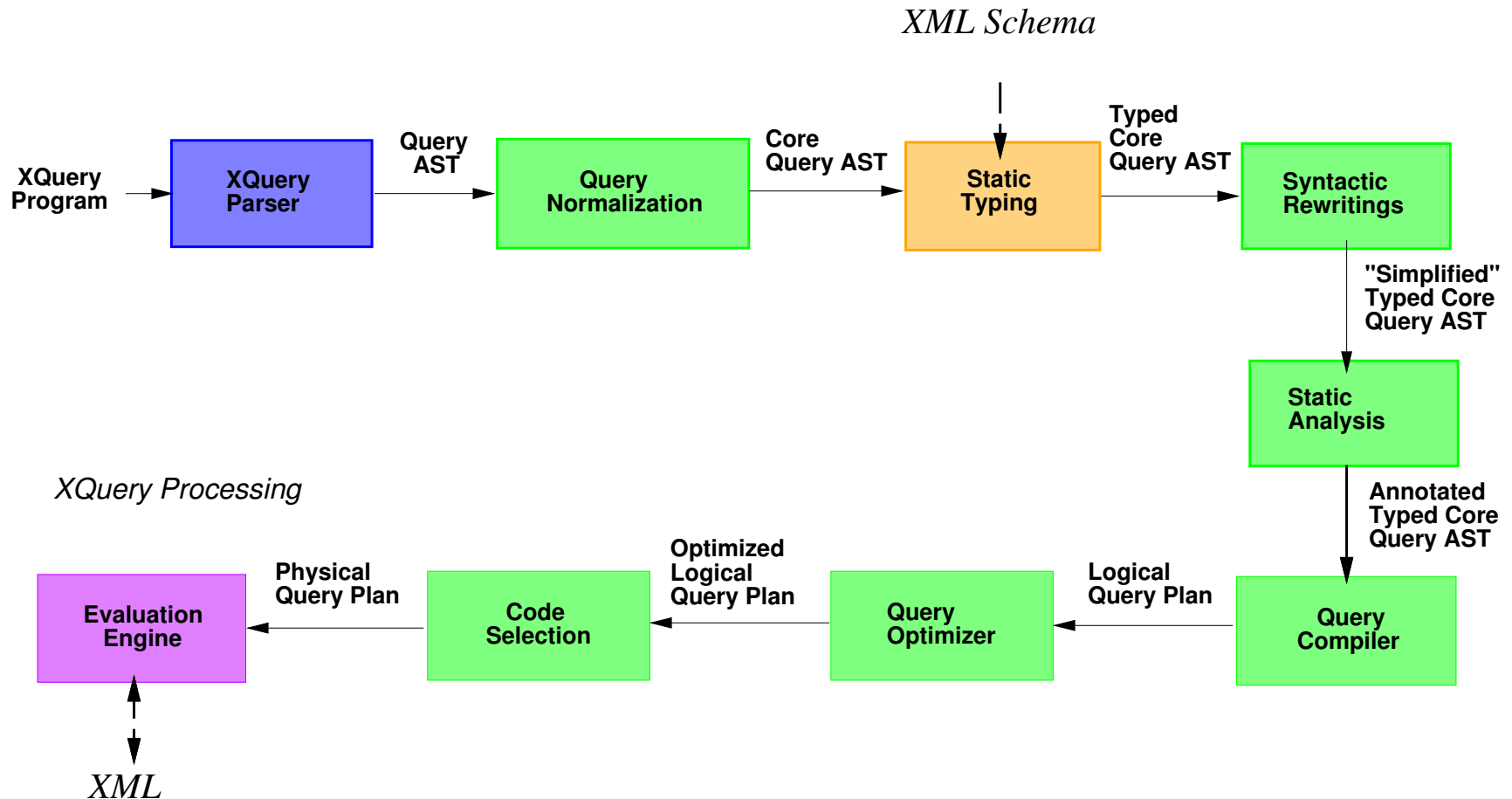


XML Processing

```
startDoc
<catalog>
  <book year="1994">
    <title>TCP/IP Illustrated</title>
    <author>Stevens</author>
  </book> ...
  startElem(catalog) element(catalog) ...
    startElem(book) element(book) ...
      startElem(title) element(title)
        [xsd:string("TCP/IP Illustrated")]
        chars("TCP/IP Illustrated")
      endElem
      startElem(author) element(author)
        [xsd:string("Stevens")]
        chars("Stevens")
      endElem ...
```

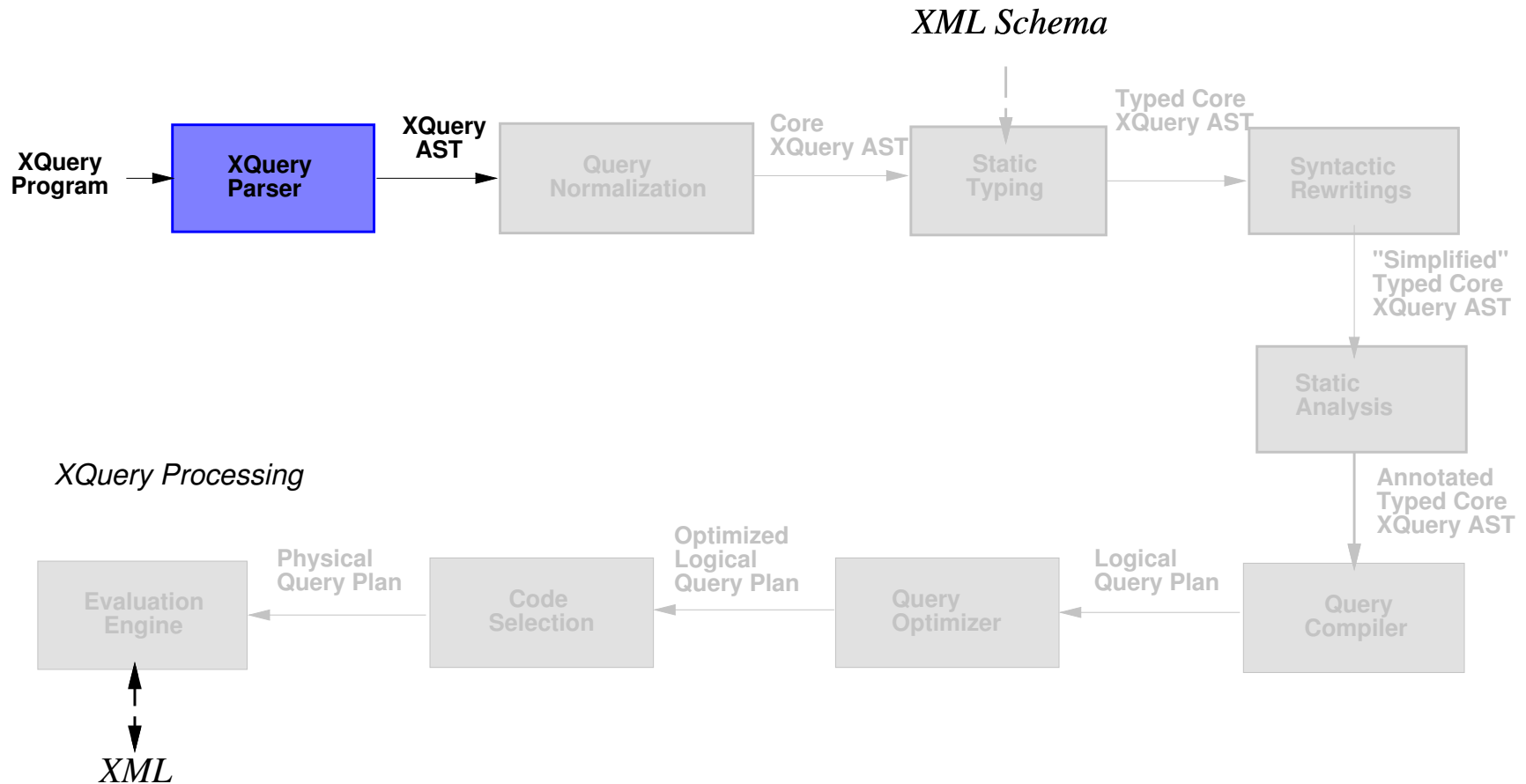
► **Reference:** XML, Infoset, XML Schema (PSVI), DM Serialization

XQuery Processing Architecture



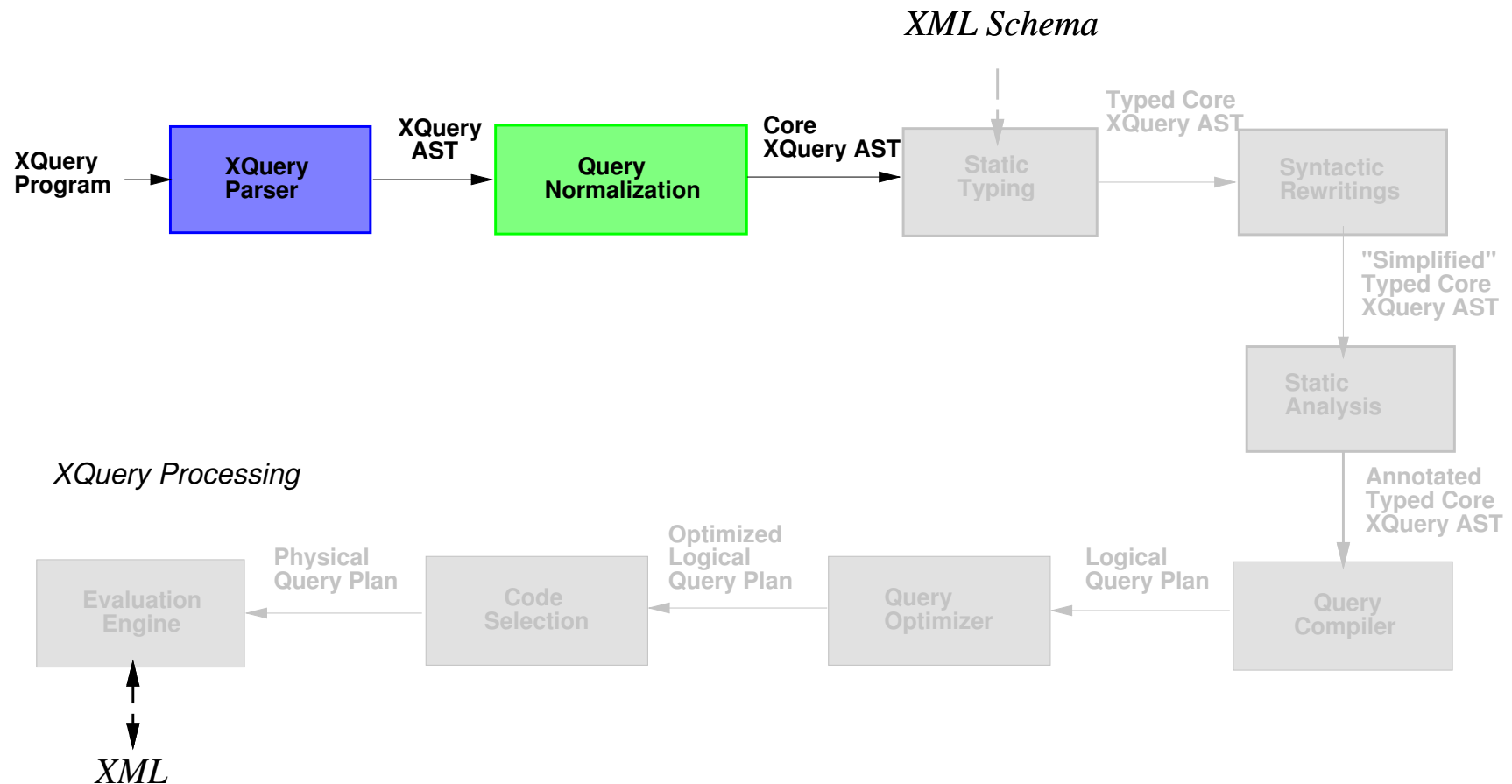
- ▶ Input: XQuery program (main module + library modules)
- ▶ Output: Instance of XQuery data model

XQuery Processing Step 1: Parsing



► **Reference:** XQuery 1.0 Working Draft

XQuery Processing Step 2: Normalization



- ▶ Rewrite query into smaller, semantically equivalent language
 - ▶ Makes surface syntax's implicit semantics explicit in core
- ▶ **Reference:** XQuery 1.0 Formal Semantics

XQuery Processing : Normalization (cont'd)

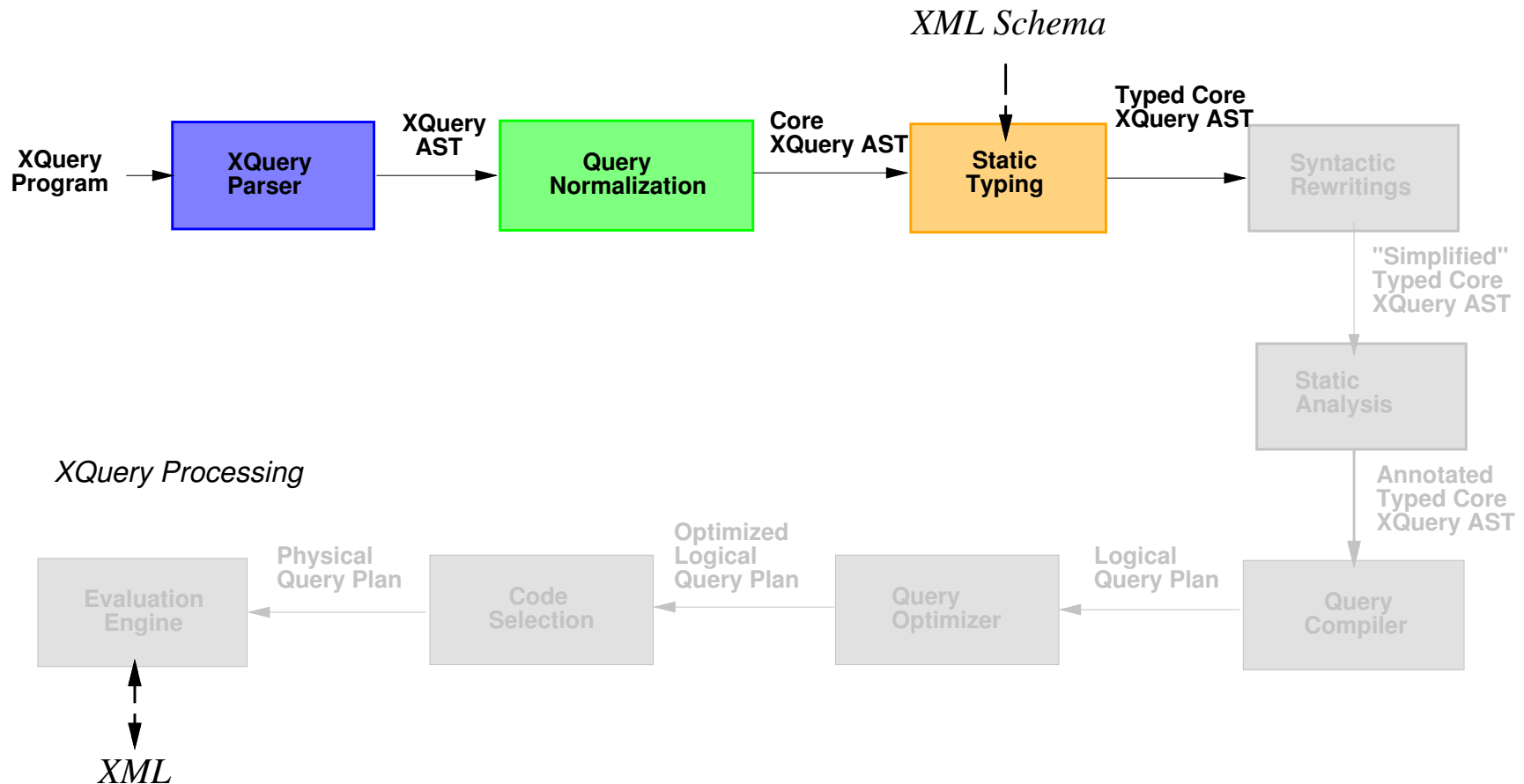
► XQuery expression:

```
$cat/book[@year >= 2000]
```

► Normalized into Core expression:

```
for $_c in $cat return
  for $_b in $_c/child::book return
    if (some $v1 in fn:data($_b/attribute::year) satisfies
        some $v2 in fn:data(2000) satisfies
          let $u1 := fs:promote-operand($v1,$v2) return
          let $u2 := fs:promote-operand($v2,$v1) return
          op:ge($u1, $u2))
    then $_b
    else ()
```

XQuery Processing Step 3: Static Typing



- ▶ Infers static type of each expression
 - ▶ Annotates each expression with type
- ▶ **Reference:** XQuery 1.0 Formal Semantics

XQuery Processing : Static Typing (cont'd)

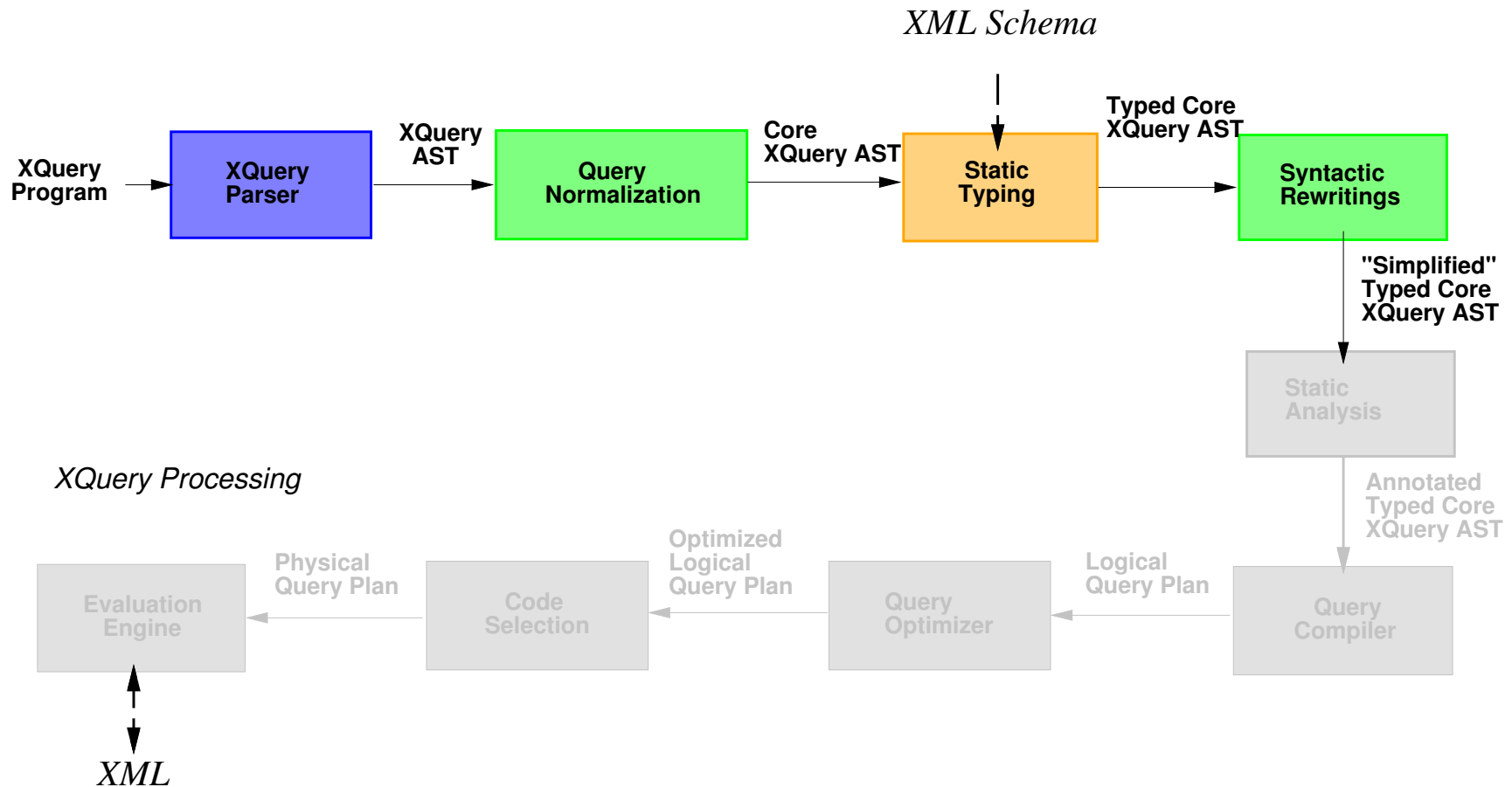
► Core expression:

```
for $_c in $cat return
  for $_b in $_c/child::book return
    if (some $v1 in fn:data($_b/attribute::year) satisfies
        some $v2 in fn:data(2000) satisfies
          let $u1 := fs:promote-operand($v1,$v2) return
          let $u2 := fs:promote-operand($v2,$v1) return
          op:ge($u1, $u2))
    then $_b
    else ()
```

► Typed Core expression:

```
for $_c [element(catalog)] in $cat [element(catalog)] return
  for $_b [element(book)] in $_c/child::book [element(book)*] return
    if (some $v1 in (fn:data($_b/attribute::year [attribute(year)]) [xs:integer])
        some $v2 in fn:data(2000) [xs:integer] satisfies
          let $u1 [xs:integer] := fs:promote-operand($v1,$v2) return
          let $u2 [xs:integer] := fs:promote-operand($v2,$v1) return
          op:ge($u1, $u2) [xs:boolean])
    then $_b [element(book)]
    else () [empty()]
[element(book)?]
```

XQuery Processing Step 4: Rewriting



- ▶ Removes redundant/unused operations, type-based simplifications; Function in-lining
- ▶ **Example:** "A Systematic Approach to Sorting and Duplicate Elimination in XQuery", Technical Report, Universiteit Antwerpen, Michiels, Hidders et al

XQuery Processing : Rewriting (cont'd)

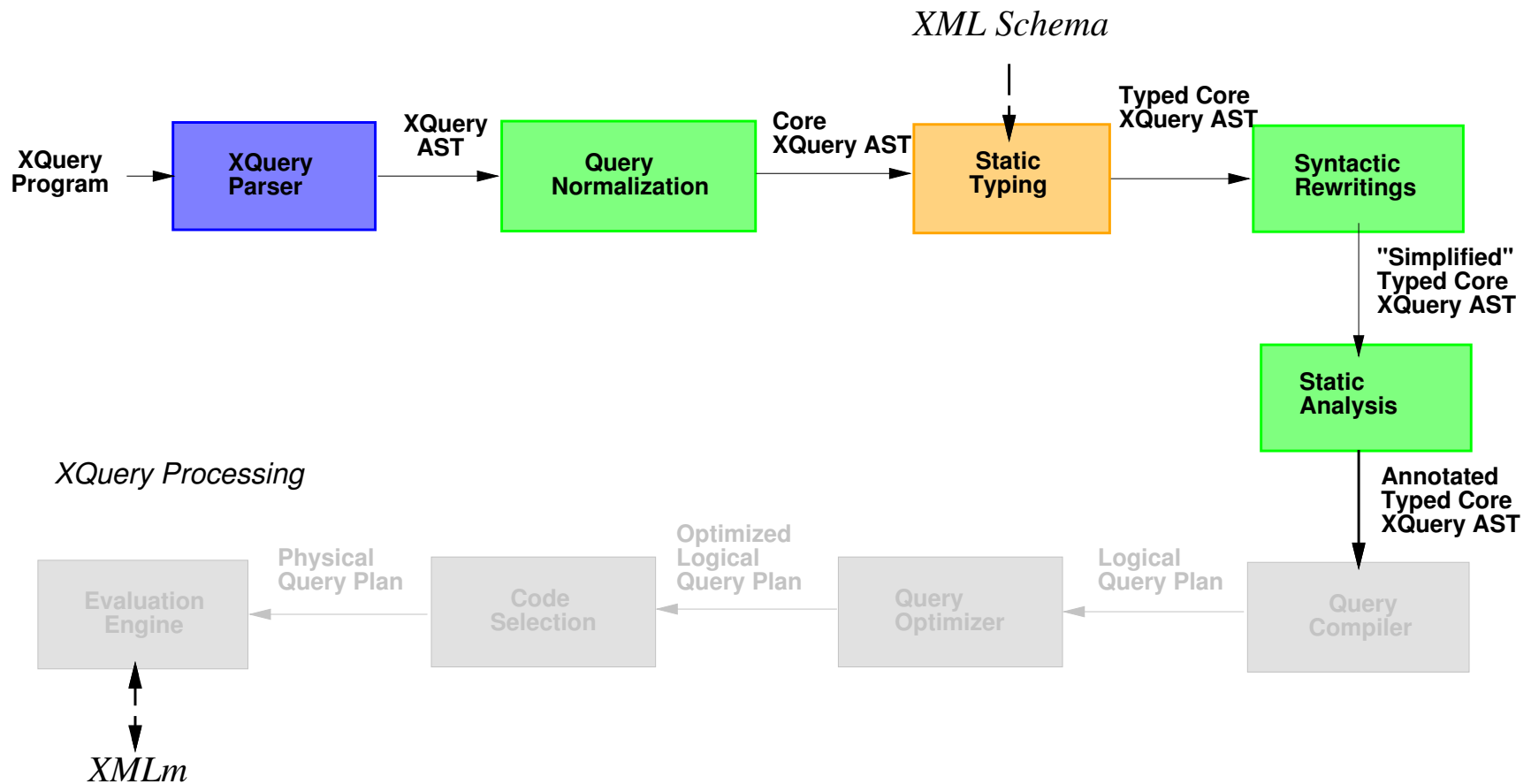
► Typed Core expression:

```
for $_c [element(catalog)] in $cat [element(catalog)] return
  for $_b [element(book)] in $_c/child::book [element(book)*] return
    if (some $v1 in (fn:data($_b/attribute::year [attribute(year)]) [xs:integer])
        some $v2 in fn:data(2000) [xs:integer] satisfies
          let $u1 [xs:integer] := fs:promote-operand($v1,$v2) return
            let $u2 [xs:integer] := fs:promote-operand($v2,$v1) return
              op:ge($u1, $u2) [xs:boolean])
    then $_b [element(book)]
    else () [empty()]
[element(book)?]
```

► Simplified typed Core expression:

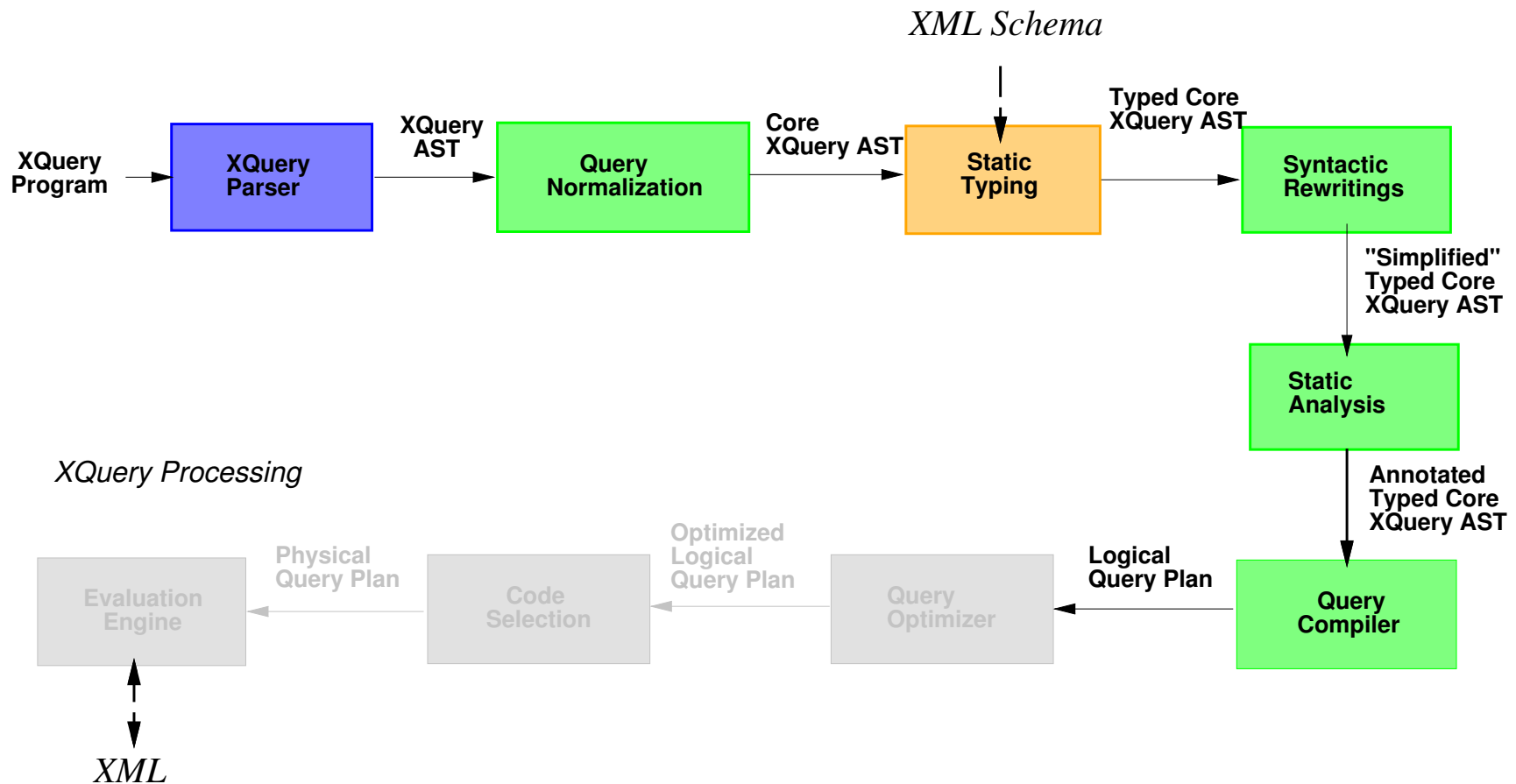
```
for $_b [element(book)] in $cat/child::book [element(book)*] return
  if (op:integer-ge(fn:data($_b/attribute::year), 2000) [xs:boolean])
  then $_b [element(book)]
  else () [empty()]
[element(book)?]
```

XQuery Processing Step 5: Static Analysis



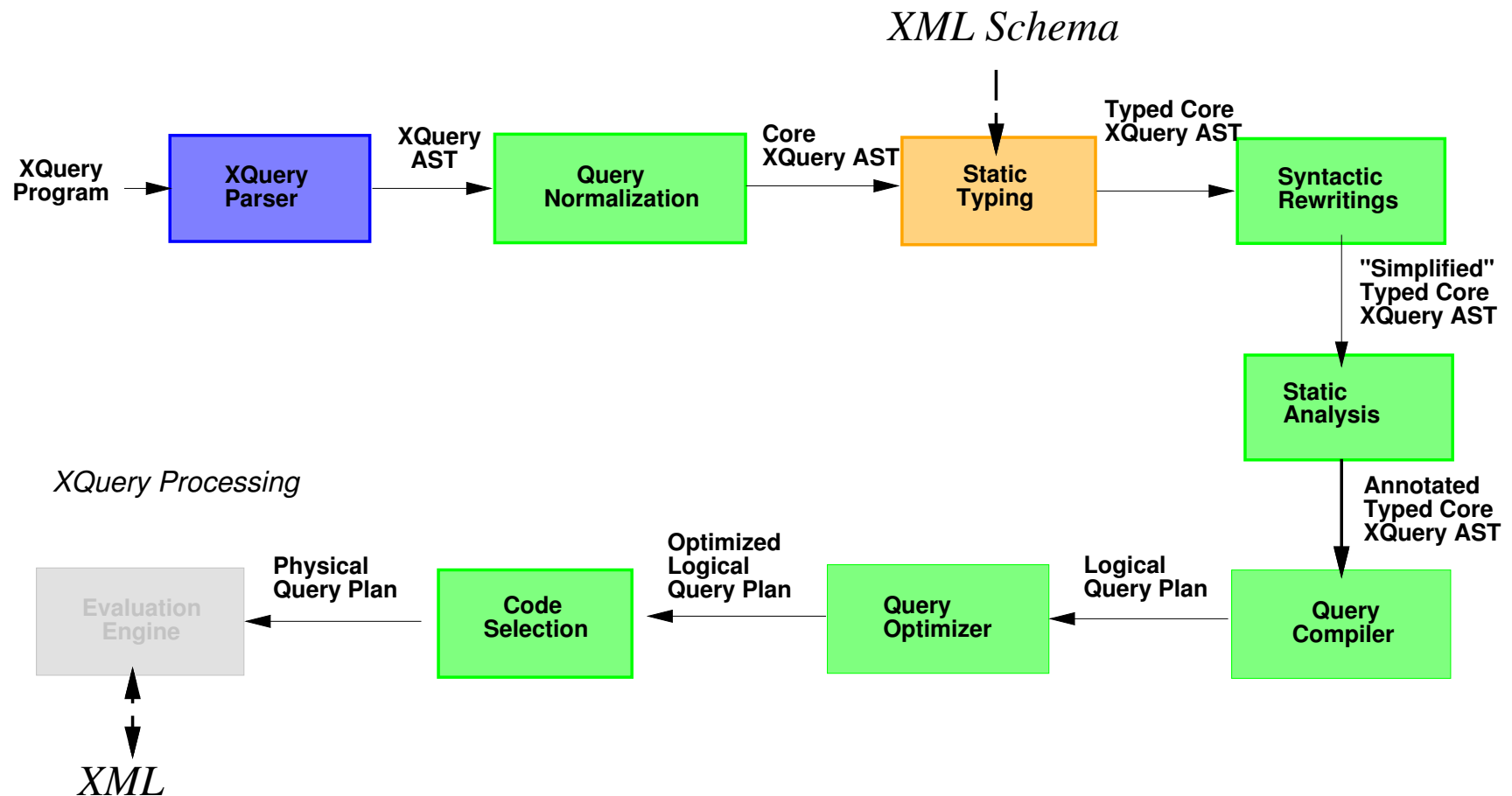
- ▶ Introduces annotations for down-stream optimizations
- ▶ **Example:** "Projecting XML Documents", VLDB 2003, Marian & Siméon

XQuery Processing Step 6: Compilation



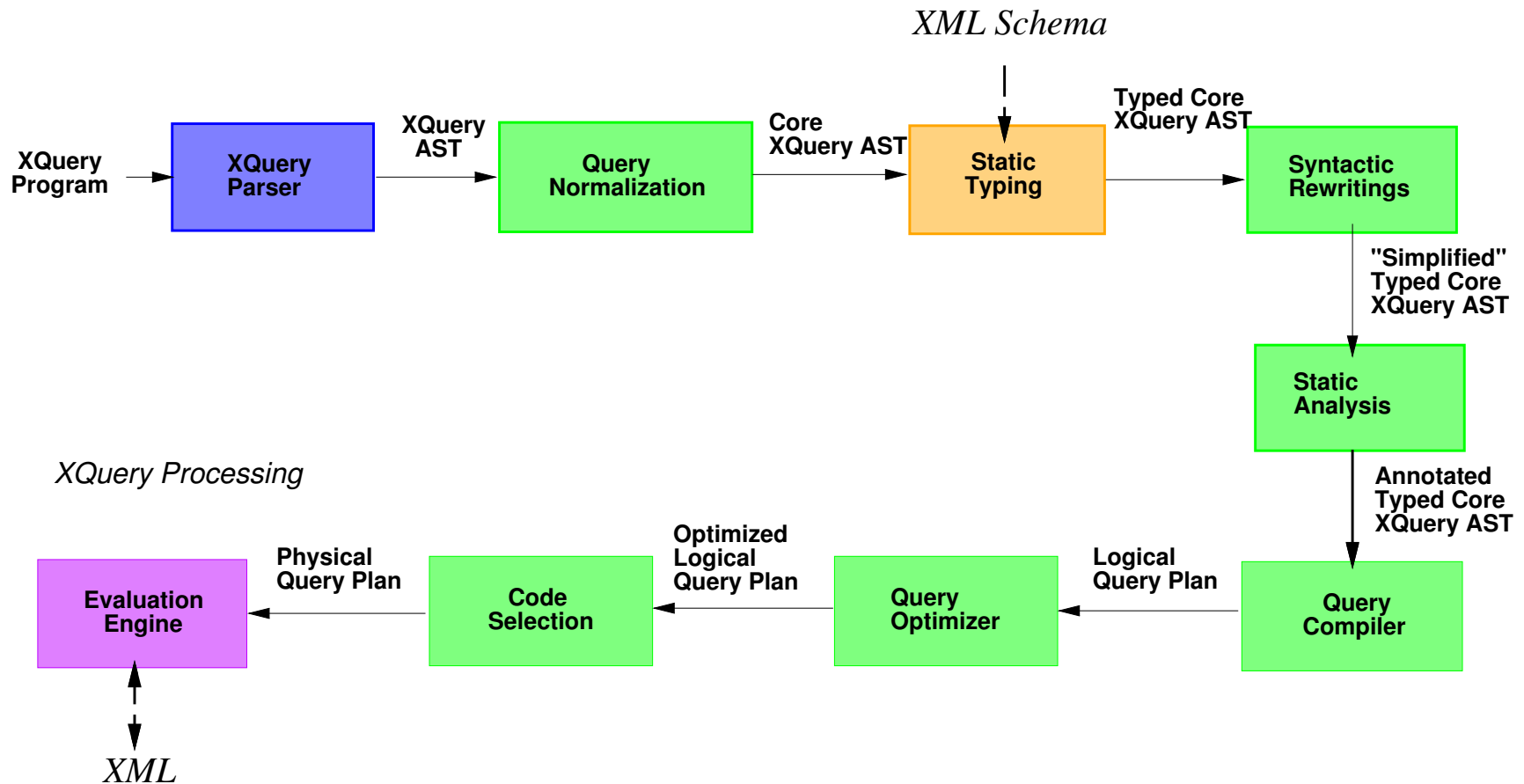
- ▶ Introduces “algebraic” operations (Join, Select, Map, etc.)
- ▶ Describes *naive* evaluation plan

XQuery Processing Steps 7 & 8



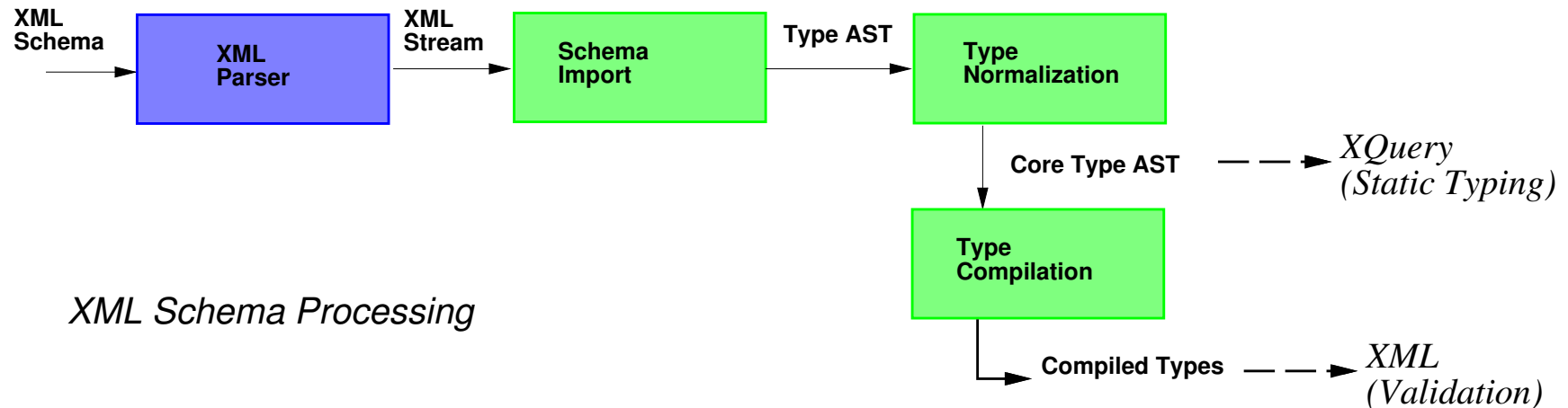
- ▶ **Step 7: Query Optimizer**
 - ▶ Describes *better* evaluation plan
- ▶ **Step 8: Code Selection**
 - ▶ Algebraic operation mapped to physical implementation(s)

XQuery Processing Step 9: Evaluation



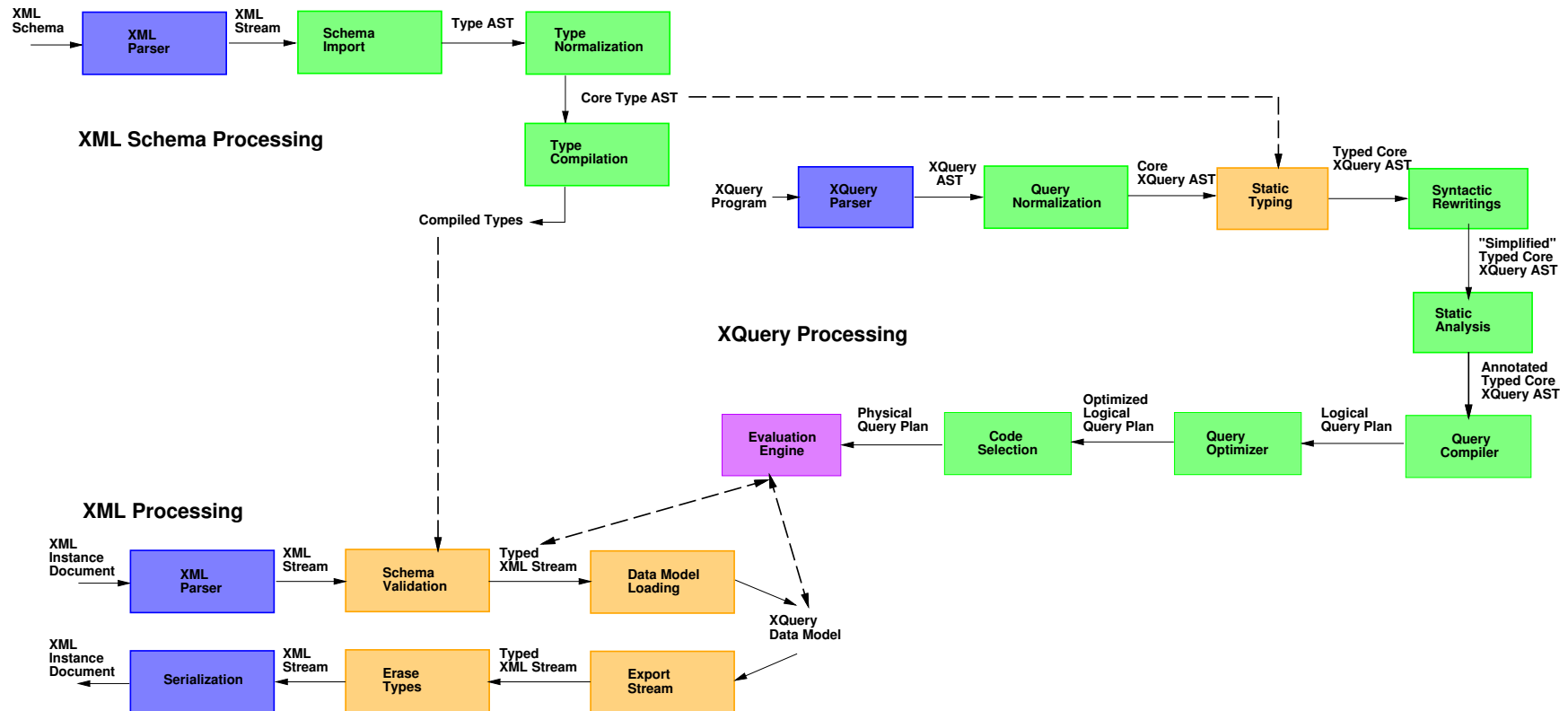
- ▶ **Output:** Instance of Galax's abstract XML data model
- ▶ Data model instance accessible via API or serialization

XML Schema Processing Architecture



- ▶ Analogous to XQuery processing model
- ▶ **References:** XQuery 1.0 Formal Semantics
"The Essence of XML", POPL 2003, Siméon & Wadler

Putting it all together

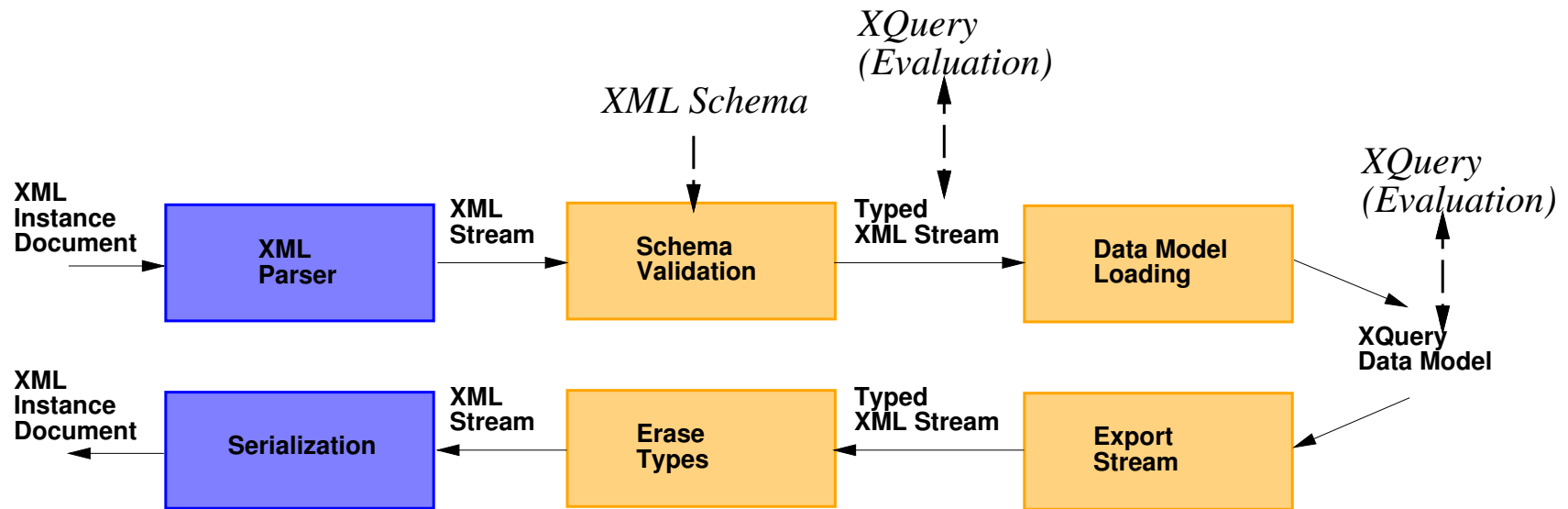


- ▶ Most code devoted to *representation transformation*
 - ▶ Of 63,000 lines O'Caml, 7600 lines (12%) for evaluation
 - ▶ O'Caml's polymorphic algebraic types support disciplined program transformation
- ▶ Coalescing phases limits extensibility & experimentation

Part IV

XML Processing

XML Processing Architecture



XML Processing

- ▶ Deals with input/output of XML
- ▶ Deals with internal XML representations
 - ▶ DOM-like (tree representation)
 - ▶ SAX-like (Stream representation)
- ▶ XML representations might not be enough
 - ▶ Need to tuples? (e.g., for relational optimization)
 - ▶ Need for index-based representations?

XML Processing References

- ▶ Very little research references
- ▶ XML 1.0 Specification
 - ▶ A lot of implementation experience
- ▶ XML Schema 1.0 Specification
 - ▶ Much less implementation experience
 - ▶ Paper at XIME-P'2004
 - ▶ Theoretical papers on tree automata
 - ▶ “Essence of XML” at POPL'2003

XQuery Data Model Representations

- ▶ Materialized XML trees:
 - ▶ DOM-like, main-memory access
 - ▶ Support for all XQuery data model operations
 - ▶ Based on PSVI (I.e., contain type information)
- ▶ XQuery Data Model additions to XML:
 - ▶ Atomic values (e.g., integer, string)
 - ▶ Sequence (e.g., (1,<a/>,'hello'))
- ▶ Streamed XML trees:
 - ▶ Stream of SAX events (open/close element, characters...)
 - ▶ Stream of **Typed** SAX events (I.e., contain type information)
 - ▶ **Pull** streams instead of SAX push streams:
 - ▶ Support open/next (cursor) interface
 - ▶ Extended to support atomic values and sequences

XML Data Model Notations

- ▶ Materialized XML values: $v_1, \dots, v_n$
- ▶ SAX events: $e_1, \dots, e_n$
 - ▶ `startElem(QName)`
 - ▶ `endElem`
 - ▶ `chars("This is text")`
- ▶ **Typed** SAX events: $te_1, \dots, te_n$
 - ▶ `startElem(QName, TypeAnnotation, Value)`
 - ▶ `endElem`
 - ▶ `chars("This is text")`
 - ▶ `atomic(xs:integer, 11)`
- ▶ SAX streams: $stream_1, \dots, stream_n$
- ▶ **Typed** SAX streams: $tstream_1, \dots, tstream_n$

Streaming operations: I/O

▶ Input / Output

▶ Parse : `channel → stream`

▶ Serialize : `stream, channel → ()`

▶ Example

```
s1 = Parse("<age>11</age>")  
[ => startElem(age) ; chars("11") ; endElem ]
```

```
Serialize(s1, stdout)  
[ => <age>11</age> ]
```

Streaming operations: Validation

▶ Typing

▶ `Validate : stream , type  $\rightarrow$  tstream`

▶ `Well-formed : stream  $\rightarrow$  tstream`

▶ `Erase : tstream  $\rightarrow$  stream`

▶ Example

```
s1 = Parse("<age>11</age>")
```

```
[ => startElem(age) ; chars("11") ; endElem ]
```

```
s2 = Validate(s1, element age of type xs:integer)
```

```
[ => startElem(age,xs:integer,11) ; chars("11") ; endElem ]
```

```
s3 = Erase(s2)
```

```
[ => startElem(age) ; chars("11") ; endElem ]
```

```
Serialize(s3,stdout)
```

```
[ => <age>11</age> ]
```

Streaming validation

- ▶ Can be done!
- ▶ Relies on XML Schema constraints:
 - ▶ One-unambiguous regular expressions
 - ▶ Transitions in Glushkov automata are deterministic
 - ▶ *or* Brozowski derivatives unambiguous
 - ▶ Same element must have same type within a content model
- ▶ Requires a stack of content models
 - ▶ Validation in a left-deep first traversal

Streaming validation

► Example:

```
declare element person of type Person;  
declare type Person { (element name, element age)+ };  
declare element name of type xs:string;  
declare element age of type xs:integer
```

```
<person><name>John</name><age>33</age></person>
```

Streaming validation

```
startElem(person) --- element person of type Person
                    push(empty)
--> startElem(person,Person)

startElem(name)    --- element name
                    push(element age,(element name, element age)*)
--> startElem(name,xs:string)

chars(''John'')    --- xs:string
                    push(empty)
--> chars(''John'',xs:string(''John''))

endElem            --- CHECK EMPTY IN TYPE empty
                    pop --> element age,(element name, element age)*
--> endElem

startElem(age)     --- element age
                    push((element name, element age)*)
--> startElem(age,xs:integer)

chars(''33'')      --- xs:integer
                    push(empty)
--> chars(''33'',xs:integer(33))
```

Streaming validation, cont'd

```
endElem      --- CHECK EMPTY IN TYPE empty
              pop --> (element name | element age)*
              --> endElem
```

```
endElem      --- CHECK EMPTY IN TYPE (element name | element age)*
              pop --> empty
endElem
```

Data Model Materialization

▶ (De)Materialization

▶ Load : `tstream` $\rightarrow$ `dm_value`

▶ Export : `dm_value` $\rightarrow$ `tstream`

```
s1 = Parse("<age>11</age>")  
[ => startElem(age) ; chars("11") ; endElem ]
```

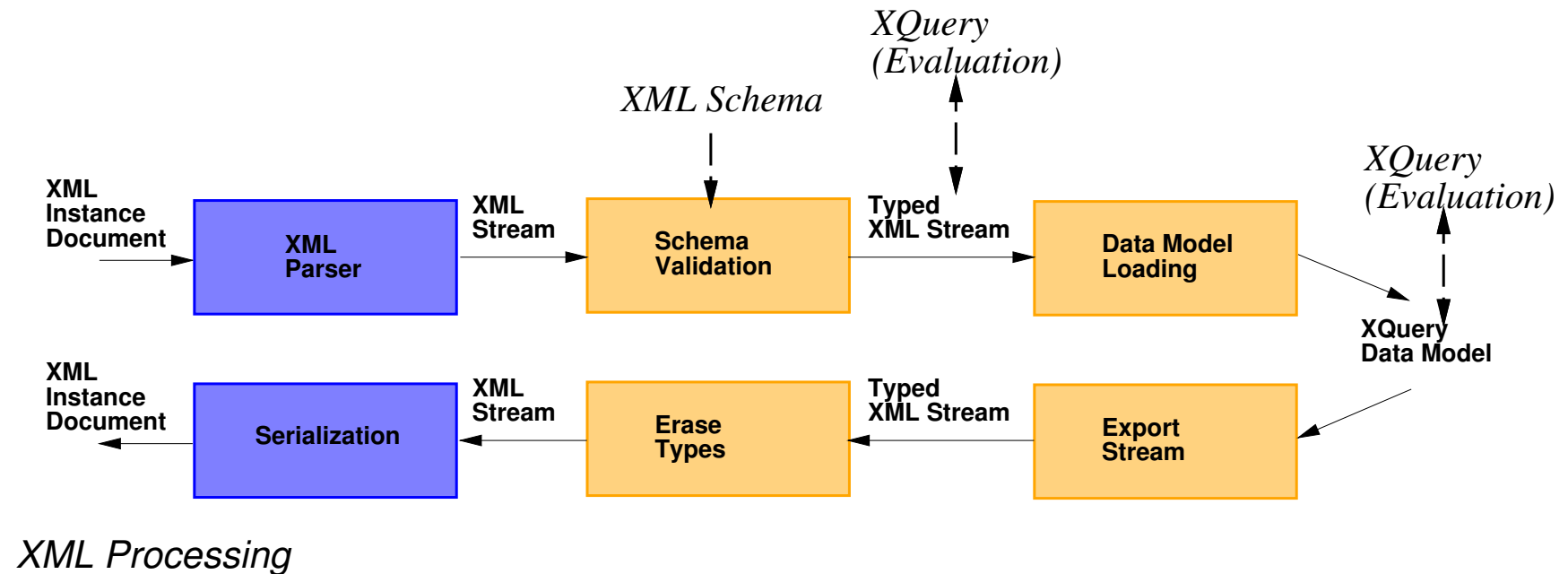
```
s2 = Validate(s1, element age of type xs:integer)  
[ => startElem(age,xs:integer,11) ; chars("11") ; endElem ]
```

```
v1 = Load(s2)  
[ => element age of type xs:integer 11 ]
```

```
s3 = export(v1)  
[ => startElem(age,xs:integer,11) ; chars("11") ; endElem ]
```

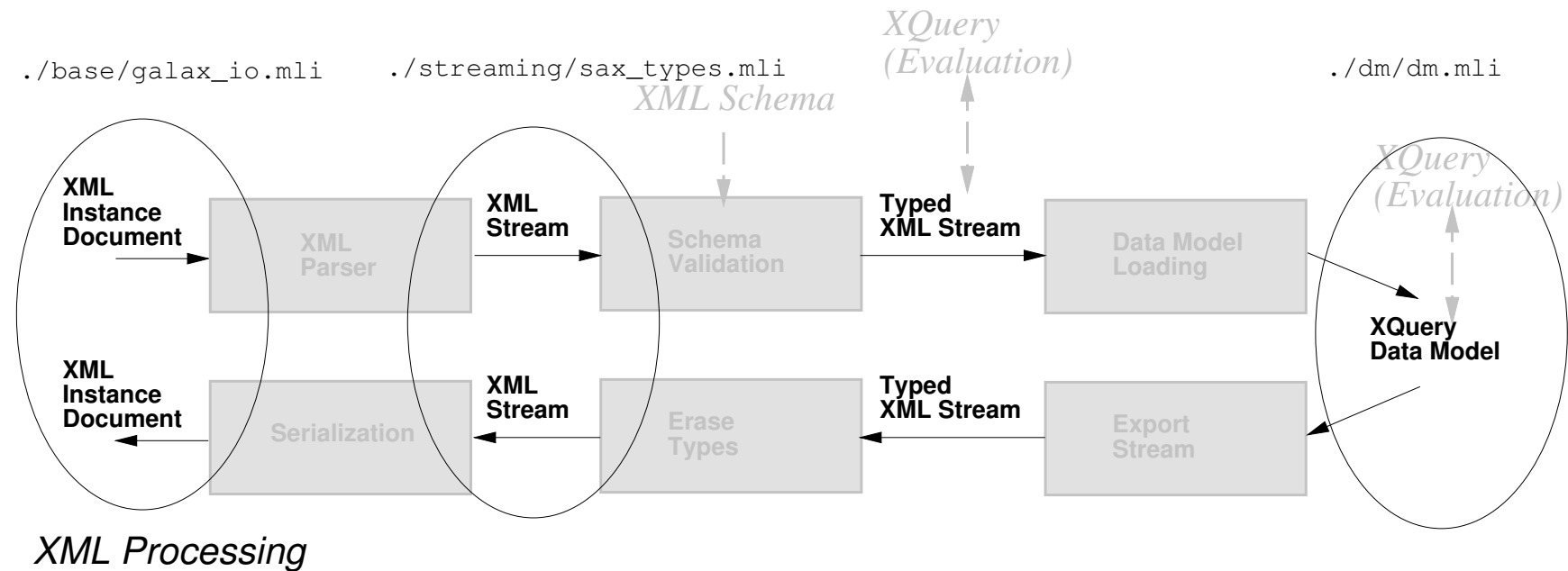
XML Processing in Galax (1)

► Abstract Syntax Trees



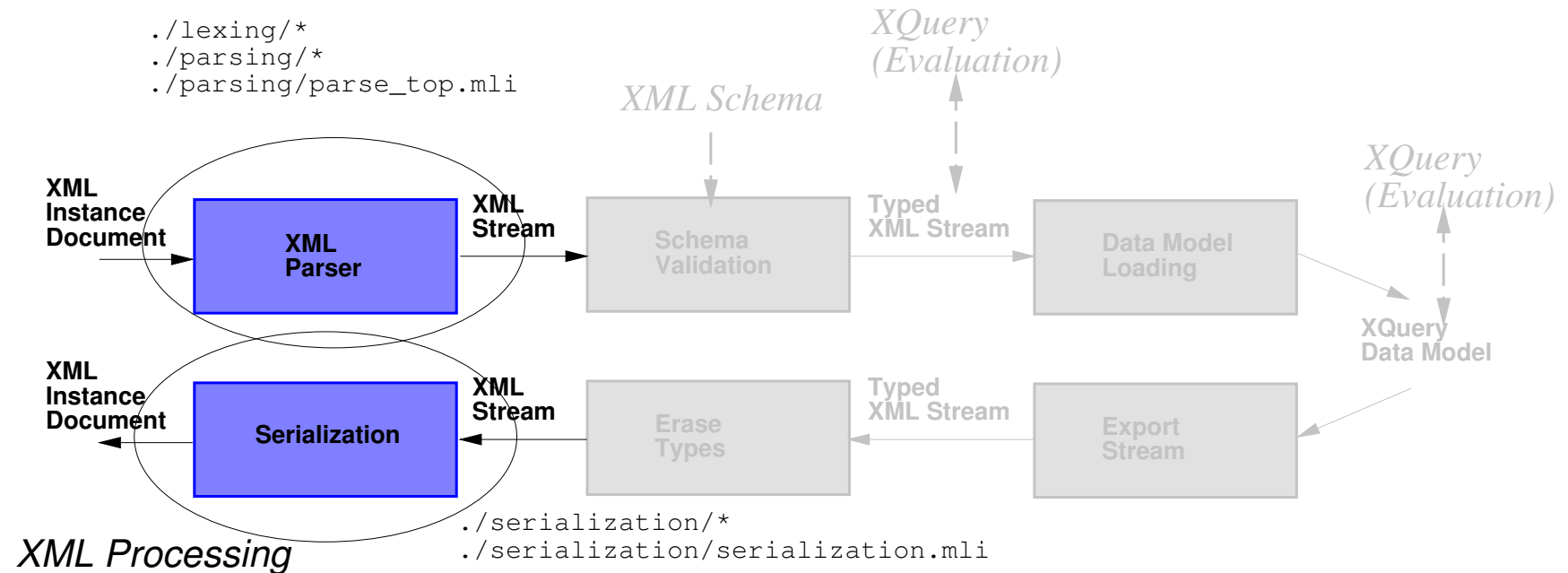
XML Processing in Galax (1)

▶ Abstract Syntax Trees



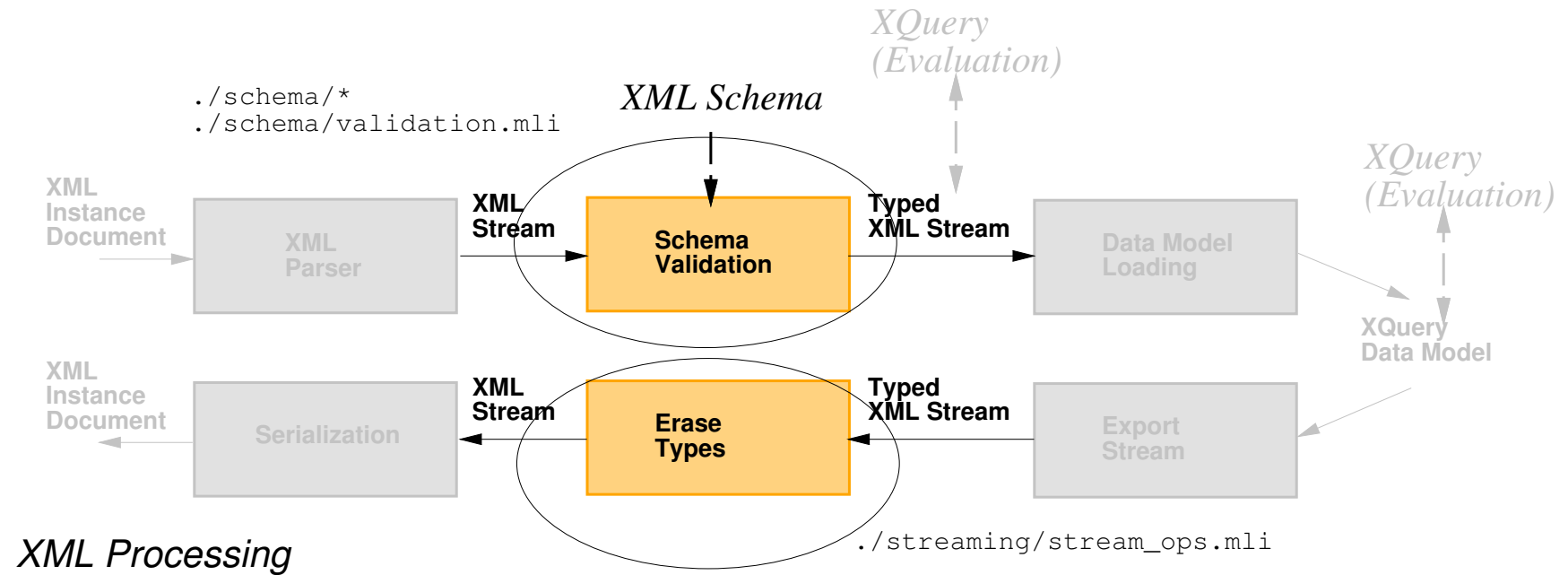
XML Processing in Galax (2)

► Parsing and serialization



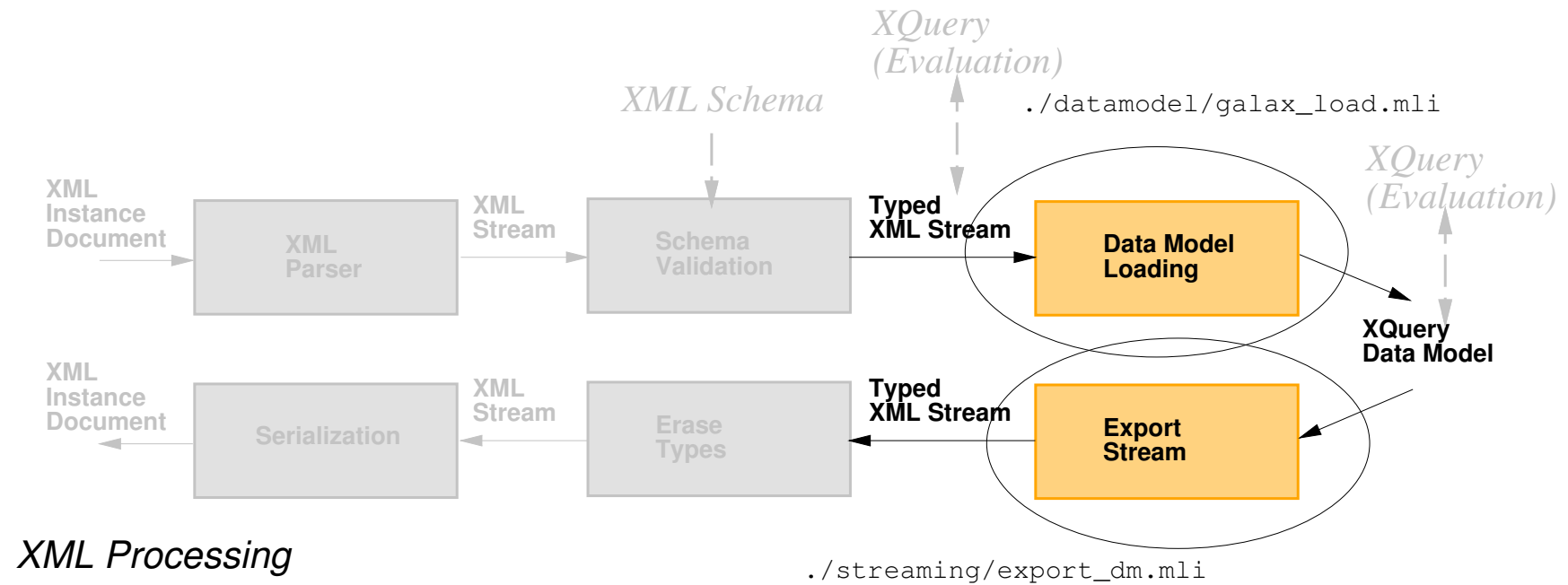
XML Processing in Galax (3)

► Stream validation and erasure



XML Processing in Galax (4)

► Document loading and export



Part V

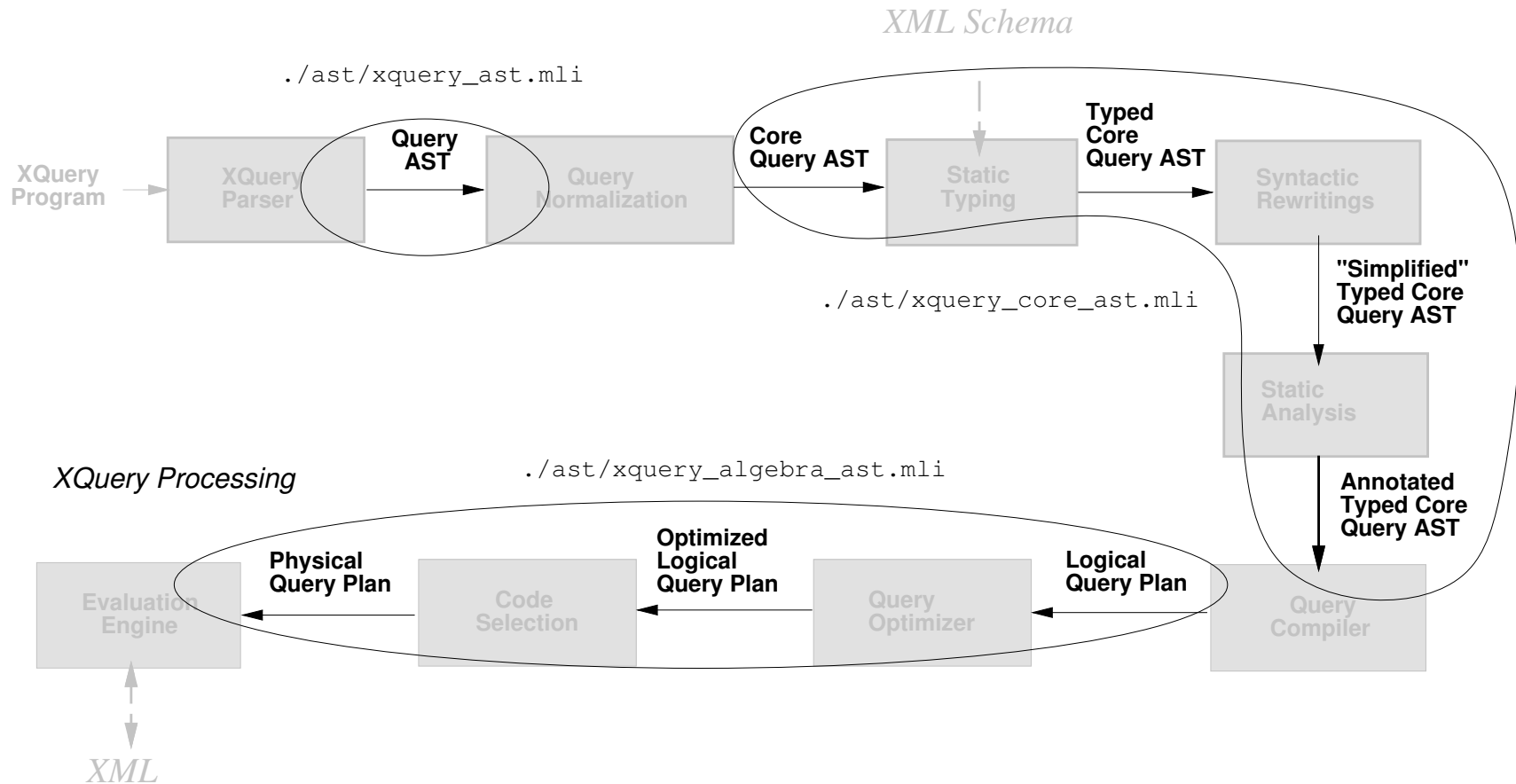
XQuery Processing

XML Query Processing References

- ▶ Huge wealth of references
- ▶ SQL references
- ▶ Programming language references
 - ▶ Function inlining
 - ▶ Tail-recursion optimization
 - ▶ etc.
- ▶ XQuery/XPath references:
 - ▶ Formal Semantics on XQuery normalization
 - ▶ XPath joins (10+ papers on twigs, staircase joins)
 - ▶ XPath streaming
 - ▶ XML algebras (TAX, etc.)
 - ▶ Indexes (Dataguides, etc)

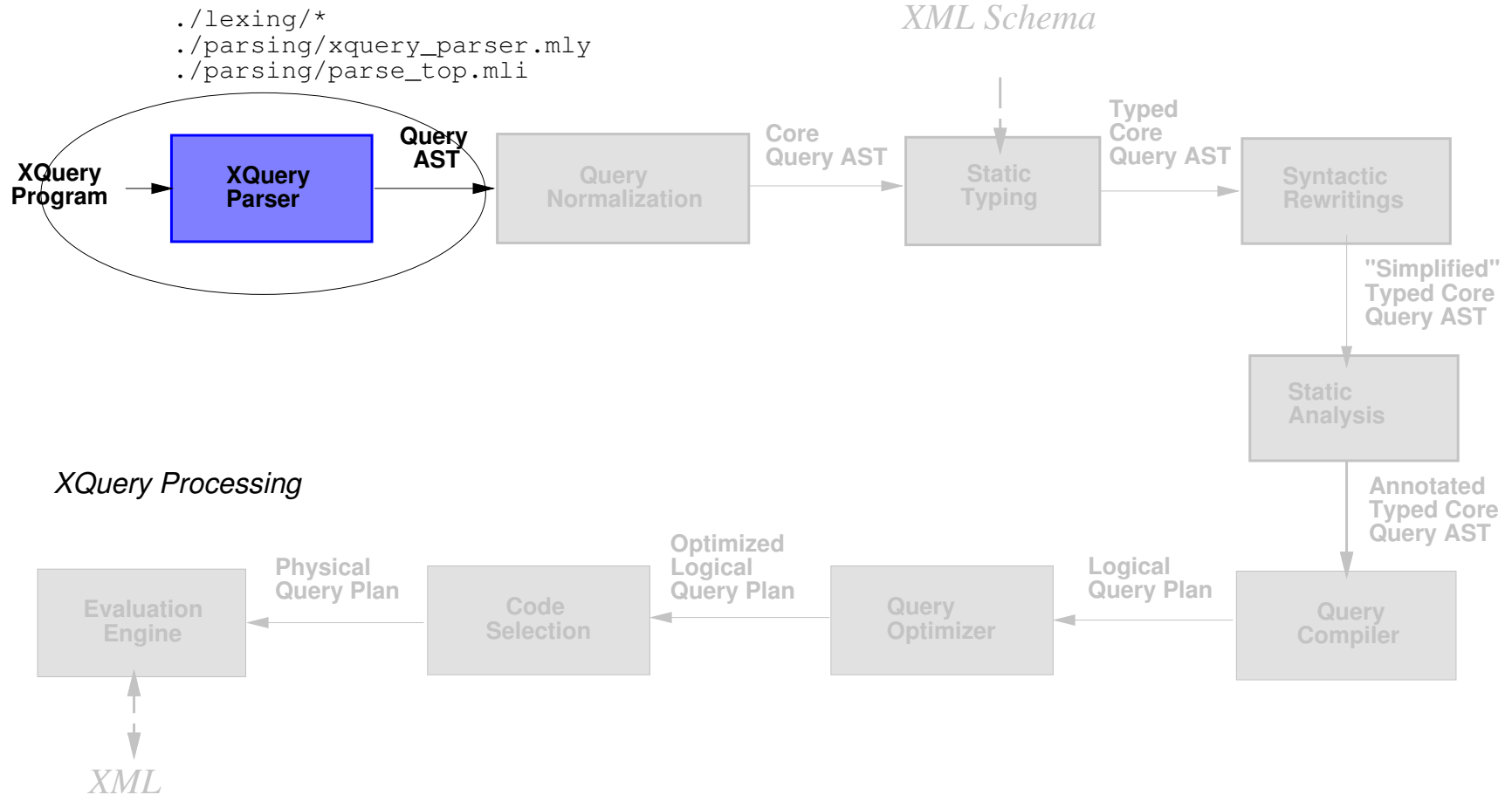
XQuery Processing in Galax (1)

► Abstract Syntax Trees



XQuery Processing in Galax (2)

► Parsing

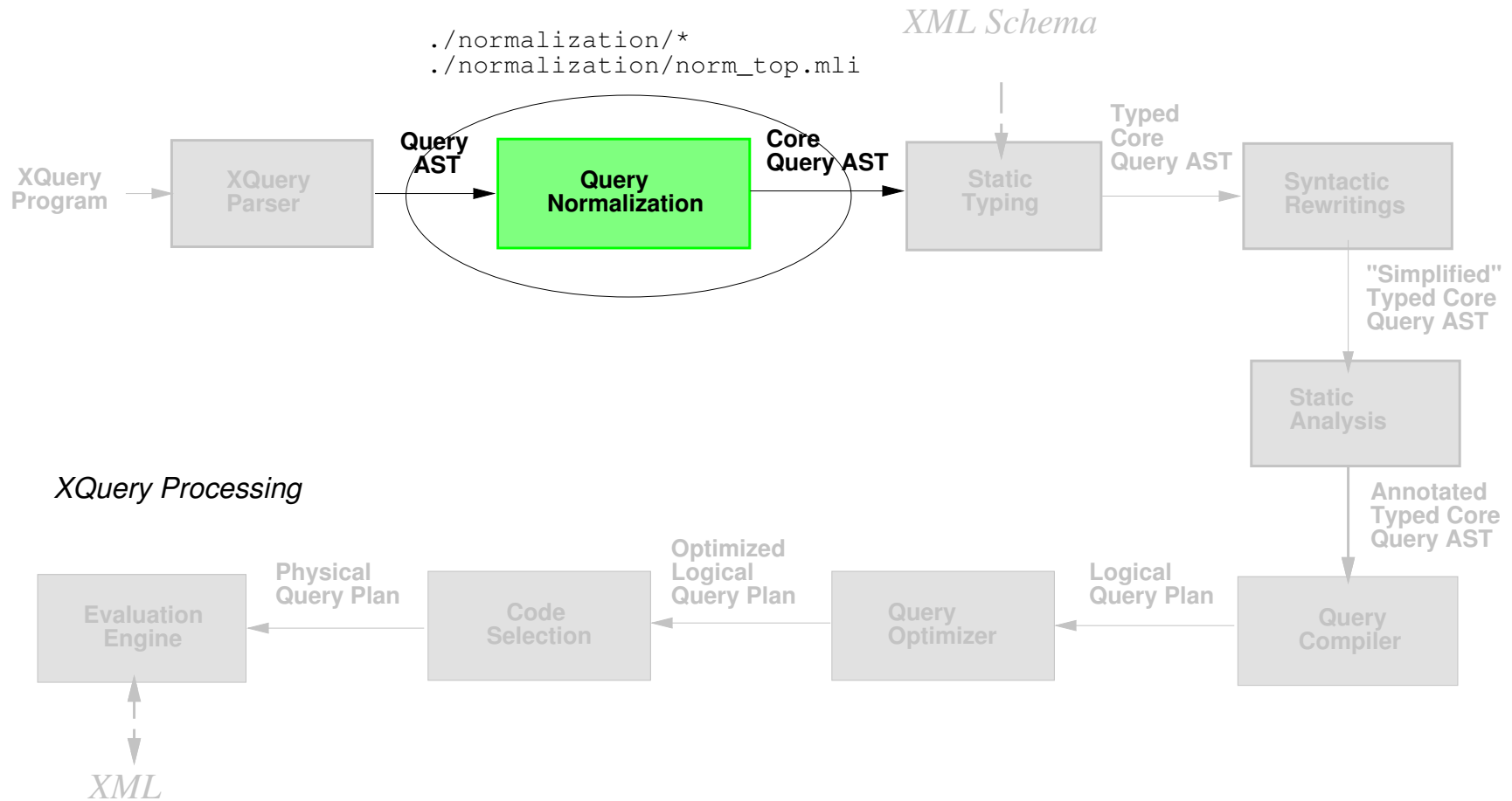


XQuery Normalization

- ▶ Fully Specified in XQuery 1.0 and XPath 2.0 Formal Semantics
- ▶ Maintained as normative by W3C XML Query working group

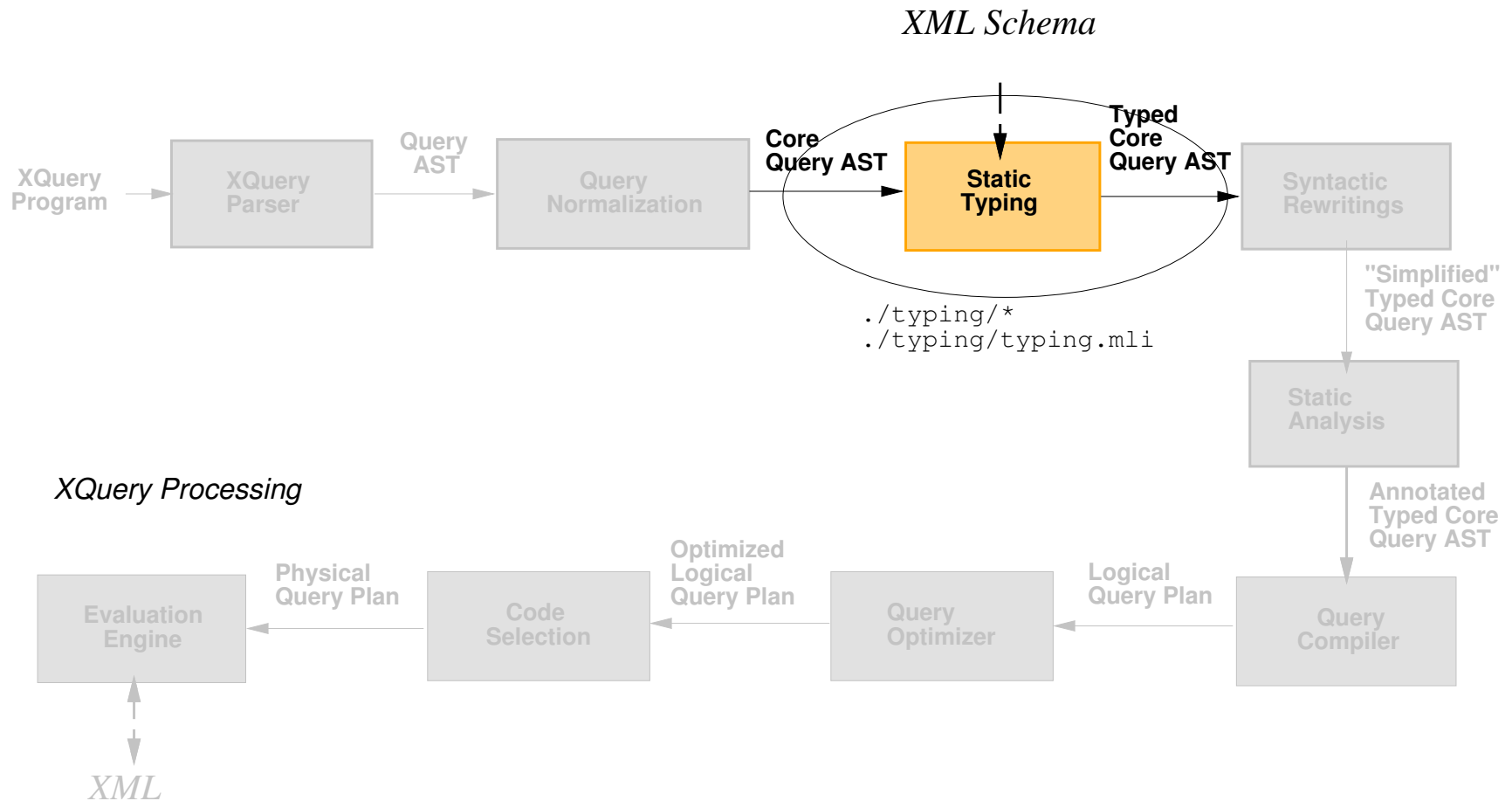
XQuery Processing in Galax (3)

► Normalization



XQuery Processing in Galax (4)

► Static Typing

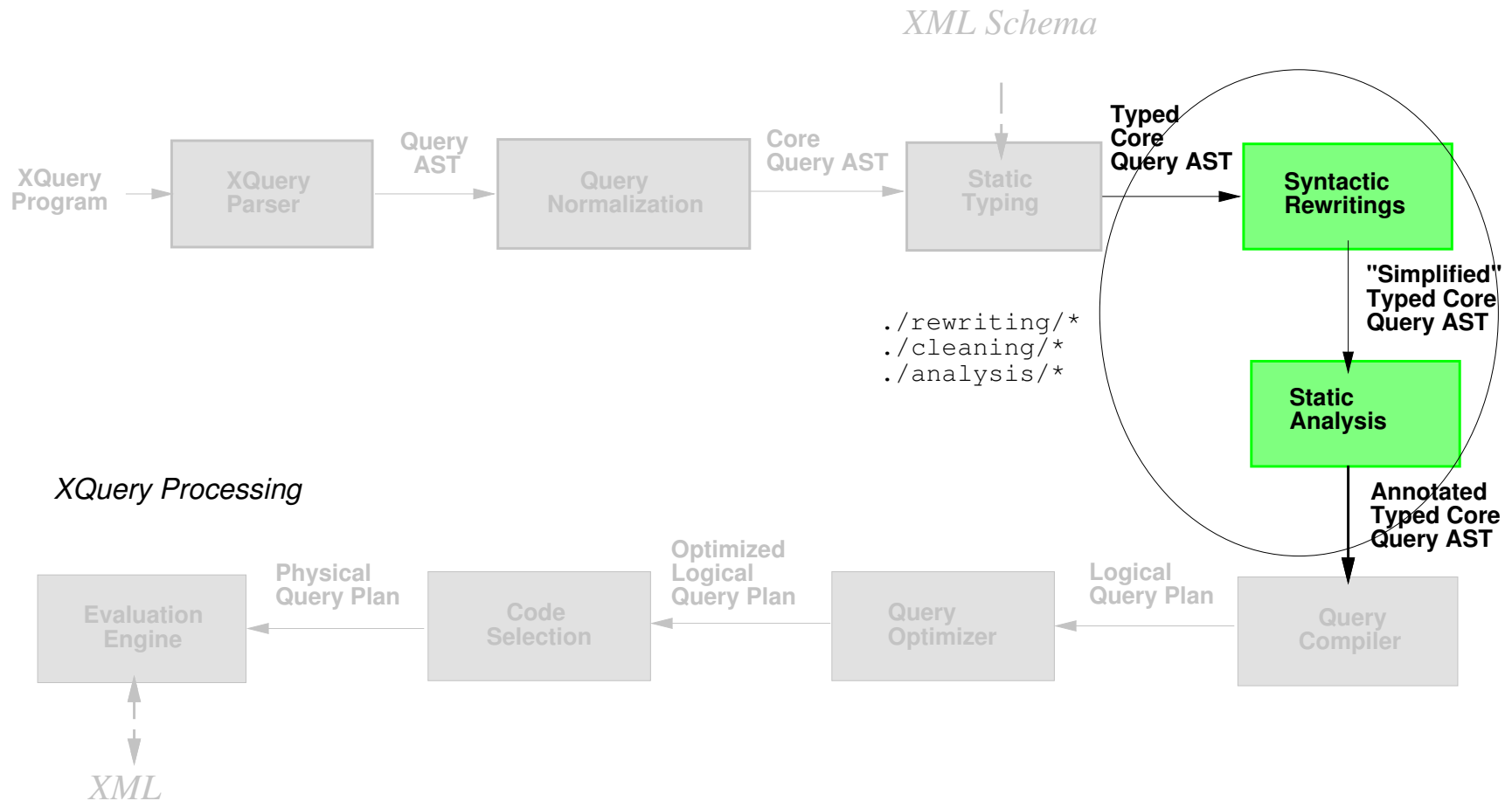


XQuery “Syntactic” Rewriting

- ▶ Only a few references
- ▶ Dana’s tutorial
- ▶ Monad laws
 - ▶ Fernandez et al “*A semi-monad for semistructured data*”, ICDT’2001
- ▶ Additional rewritings:
 - ▶ Choi et al “*The XQuery Formal Semantics: A Foundation for Implementation and Optimization*”, technical report, 2002.

XQuery Processing in Galax (5)

► Rewriting and static analysis



XML Query query processing references

- ▶ Huge wealth of references
- ▶ SQL references
- ▶ OQL references
- ▶ XQuery/XPath references:
 - ▶ XPath joins (10+ papers on twigs, staircase joins)
 - ▶ XML algebras (TAX, etc.)
 - ▶ Query decorrelation/unnesting
 - ▶ In Galax: May et al, ICDE'2004
 - ▶ Indexes (Dataguides, etc)
 - ▶ In Galax: Torsten et al *"Accelerating XPath Location Steps"*

References on streaming

- ▶ BEA's XQuery implementation:
 - ▶ “The BEA/XQRL Streaming XQuery Processor” . Florescu et al, VLDB 2003
- ▶ XPath streaming (10+ papers)
 - ▶ E.g., “Streaming XPath Processing with Forward and Backward Axes” Barton et al, ICDE'2003.
 - ▶ In Galax: Marian and Siméon *“Document projection”*
- ▶ SAX-based element construction
 - ▶ Probably similar to “Algebraic XML Construction in Natix” , WISE'2001.
 - ▶ In Galax: streaming element construction as part of the algebra (see later)
 - ▶ In Galax: streaming validation (see earlier)

Galax's Hybrid Algebra for XQuery

- ▶ Database algebra (nested relational)
 - ▶ Tuple-based processing
 - ▶ Focus on Join / Grouping / Ordering
 - ▶ Efficient over physical indexes on disk
 - ▶ Partial support for pipelining
- ▶ Extended with streaming operation
 - ▶ SAX-based processing
 - ▶ Efficient over files, and network messages
 - ▶ Direct support for pipelining
- ▶ Finally:
 - ▶ Operations to go between materialized and streaming XML

Physical Data Model Extensions

- ▶ In addition to XML data model
- ▶ Tuples:
 - ▶ Fixed-sized records with fields containing trees
 - ▶ Support access to given field
 - ▶ Either materialized
 - ▶ Tables, like in relational!
 - ▶ Or streamed
 - ▶ Support open/next (cursor) interface

Physical Data Model Notations

▶ Tuples:

- ▶ Tuple creation: [a1 : v1, ..., an : vn]
- ▶ Tuple field access: **B**#a1
- ▶ Tuple concatenation: T1 ++ T2

Basic Streaming operations

▶ Input / Output

- ▶ Parse : `channel` $\rightarrow$ `stream`
- ▶ Serialize : `stream, channel` $\rightarrow$ `()`

▶ Typing

- ▶ Validate : `stream, type` $\rightarrow$ `tstream`
- ▶ Well-formed : `stream` $\rightarrow$ `tstream`
- ▶ Erase : `tstream` $\rightarrow$ `stream`

▶ (De)Materialization

- ▶ Load : `tstream` $\rightarrow$ `dm_value`
- ▶ Export : `dm_value` $\rightarrow$ `tstream`

Advanced streaming operations

▶ StreamNestedLoop:

`StreamNestedLoop(Var, Expr, StreamExpr)`

- ▶ Evaluates input expression `Expr`
- ▶ Iterates over results, binding variable `Var`
- ▶ Processes the `StreamExpr` for each binding of variable
- ▶ Builds the output in a streamed fashion

▶ StreamXPath:

- ▶ Processes fragments of XPath in a streaming fashion
- ▶ Existing algorithms in literature
- ▶ E.g., Barton et al, ICDE'2003

▶ StreamProjection:

- ▶ Removes unnecessary parts of the stream based on the query

Streaming operations: Element construction

- ▶ Streams with *holes*

- ▶ Element constructor:

```
SmallExp ::= element QName { SmallExp }  
          | SmallExp "," SmallExp  
          | [HOLE]
```

- ▶ Creates a 'small stream' with holes:

- ▶ $\text{SmallStream} : \text{SmallExp} \rightarrow [h_1, \dots, h_n] \text{ stream}$

- ▶ Stream composition

- ▶ $\text{StreamCompose} : [h_1, \dots, h_k] \text{ stream}, [j_1, \dots, j_l] \text{ stream} \rightarrow [j_1, \dots, j_l, h_1, \dots, h_k] \text{ stream}$

Element Construction: Example

- ▶ Back to the sample query on books:

```
for $author in distinct-values($cat/book/author),  
let $books := $cat/book[@year >= 2000 and author = $author]  
return  
  <total-sales>  
    <author> { $author } </author>  
    <count> { count($books) } </count>  
  </total-sales>
```

- ▶ Compiled to the following query plan:

```
StreamNestedLoop($tu, Scan([author : ...,  
  StreamCompose  
    StreamCompose(  
      SmallStream(element total-sale {  
        element author { [HOLE] },  
        element count { [HOLE] } })),  
      Export(GETVar($tu).author)),  
    Export(count(GETVar($tu).books))))
```

Document Projection

“Document Projection”

- ▶ Similar to relational projection
 - ▶ One of key operations
 - ▶ Prunes unnecessary part of the data
 - ▶ Essential for memory management
- ▶ Specific problems related to XML
 - ▶ Projection must operate on trees
 - ▶ Requires analysis of the query
 - ▶ Need to address XQuery complexity
- ▶ Implementation may operate directly on **SAX streams**

Document Projection: The Intuition

- ▶ Given a query:

```
for $b in /site/people/person[@id="person0"]  
return $b/name
```

- ▶ Most nodes in the input document(s) not required
 - ▶ Projection operation removes unnecessary nodes
- ▶ How it works Static analysis of the query
 - ▶ Projection defined by set of paths
 - ▶ Static analysis infers set of paths used within a query
 - ▶ Example here:

```
/site/people/person  
/site/people/person/@id  
/site/people/person/name
```

Document Projection: The Intuition

```
<site>
  <regions>...</regions>
  <people>
    ...
    <person id="person120">
      <name>Wagar Bougaut</name>
      <emailaddress>mailto:Bougaut@wgt.edu</emailaddress>
    </person>
    <person id="person121">
      <name>Waheed Rando</name>
      <emailaddress>mailto:Rando@pitt.edu</emailaddress>
      <address>
        <street>32 Mallela St</street>
        <city>Tucson</city>
        <country>United States</country>
        <zipcode>37</zipcode>
      </address>
      <creditcard>7486 5185 1962 7735</creditcard>
      <profile income="59224.09">
    ...
```

- ▶ For that query, less than 2% of the original document!

Document Projection: Query Analysis

- ▶ Analyzing XQuery is difficult:
 - ▶ Deal with variables
 - ▶ Deal with complex expressions
 - ▶ Deal with compositionality
- ▶ Analysis must deal with all of XQuery
 - ▶ Performed on XQuery core (smaller instruction set)
- ▶ Idea of the analysis:
 - ▶ For an expression *Expr*, compute the paths reaching the nodes required to evaluate that expression
 - ▶ Notation:

$$Expr \Rightarrow Paths$$

Maximal Document Size

► Queries:

► **Query 3:** Navigation, single iteration with selection and element construction

► **Query 14:** Non-selective path query with contains predicate

► **Query 15:** Long, very selective path expression

<i>Configuration</i>		<i>A</i>	<i>B</i>	<i>C</i>
Query 3	NoProj	33Mb	220Mb	520Mb
	OptimProj	1Gb	1.5Gb	1.5Gb
Query 14	NoProj	20Mb	20Mb	20Mb
	OptimProj	100Mb	100Mb	100Mb
Query 15	NoProj	33Mb	220Mb	520Mb
	OptimProj	1Gb	2Gb	2Gb

► All queries operate on 100Mb or more

► Most navigation/selection queries work up to 1Gb document

► For more than 1Gb, scan of the document becomes a bottleneck

Database Algebra

▶ Standard database algebraic operators:

- ▶ Scan: Creates a sequence of tuples
- ▶ Map: iterate over a sequences of tuples
- ▶ Select: Selects a sub-sequence based on a predicate
- ▶ Join: Joins two sequences of tuples
- ▶ GroupBy: Performs re-grouping of tuples based on a criteria

```
GroupBy(Scan(T in AuthorTable),  
        T.NAME,  
        COUNT : count(PARTITION) )
```

▶ *“Regroup the tuples in the authors table by their name and count the number of tuple in each corresponding partition, putting the result in the COUNT collumn.”*

Standard Algebraic Optimization

- ▶ Pushing a selection:

```
Select(Join(Scan(A2 in AuthorTable),  
            Scan(B2 in BookTable),  
            A2.bid = B2.bid),  
       B2.year >= 2003)
```

==

```
Join(Scan(A2 in AuthorTable),  
     Select(Scan(B2 in BookTable),  
            B2.year >= 2003),  
     A2.bid = B2.bid)
```

- ▶ Removes unnecessary tuples as early as possible

Standard Algebraic Optimization, cont'd

- Unnest a query into a group-by:

```
Map(AUTHOR ;  
    distinct( Project(A1.name, Scan(A1 in AuthorTable)) ),  
    count(Select(Scan(A2 in AuthorTable),  
                AUTHOR = A2.name)))
```

==

```
GroupBy(  
    Scan(A1 in AuthorTable),  
    A1.name,  
    COUNT : count(PARTITION))
```

- Requires only one scan of the AuthorTable

DB optim adapted to XQuery

► Example with a simple join:

```
for $b in doc("bib.xml")/bib//book,  
    $a in doc("reviews.xml")//entry  
where $b/title = $a/title  
return ($b/title,$a/price,$b/price)
```

► Step 1. normalization:

```
for $b in doc("bib.xml")/bib//book return  
  for $a in doc("reviews.xml")//entry return  
    if ($b/title = $a/title) then  
      ($b/title,$a/price,$b/price)  
    else  
      ()
```

DB optim adapted to XQuery, cont'd

▶ Normalized query:

```
for $b in doc("bib.xml")/bib//book return
  for $a in doc("reviews.xml")//entry return
    if ($b/title = $a/title) then
      ($b/title,$a/price,$b/price)
    else ()
```

▶ Compiled into tuple-based algebra as:

```
for-tuple $t3 in
  (for-tuple $t2 in
    (for-tuple $t1 in
      (for-tuple $t0 in []
        return
          for $b in doc("bib.xml")/bib//book return [ b : $b ] ++ $t0 )
      return
        for $a in doc("reviews.xml")//entry return $t1 ++ [ a : $a ]))
    return
      if ($t2#b/title = $t2#a/title) then $t2 else ())
  return
    ($b/title,$a/price,$b/price)
```

- ▶ Variables turned into 'fields' in tuples
- ▶ for-tuple corresponds to Map, implemented as a nested loop.

DB optim adapted to XQuery, cont'd

► Naive algebraic plan:

```
for-tuple $t3 in
  (for-tuple $t2 in
    (for-tuple $t1 in
      (for-tuple $t0 in []
        return
          for $b in doc("bib.xml")/bib//book return [ b : $b ] ++ $t0 )
      return
        for $a in doc("reviews.xml")//entry return $t1 ++ [ a : $a ]))
    return if ($t2#b/title = $t2#a/title) then $t2 else ())
  return ($b/title,$a/price,$b/price)
```

► Can be turned onto a join:

```
for-tuple $t3 in
  (Join ($b/title = $a/title),
    $b in doc("bib.xml")/bib//book,
    $a in doc("reviews.xml")//entry,
    [ b : $b ; a : $a ])
  return
    ($b/title,$a/price,$b/price)
```

► Join and for-tuple here can be implemented through pipelining

DB optim adapted to XQuery, cont'd

▶ To relate things clearly:

▶ Operations above are some of the basic operations in the algebra proposed by Moerkotte et al.

```
X_a:E2(E1) == for-tuple $a in E1 return E2
```

▶ Selection is implemented above as a map:

```
X_a:E2(E1) == for-tuple $a in E1 return E2
```

```
Sigma_p(E) == X_a:(if p then a else ())(E)  
            == for-tuple $a in E return (if p then a else ())
```

▶ etc.

▶ Other standard algebraic operations and rewritings apply directly.

More on Nested Queries

- ▶ NRA / OQL optimizations

- ▶ “Algebraic Optimization of Object-Oriented Query Languages”, Beeri and Kornatzky, TCS 116(1&2), aug 1993.

- ▶ “Nested Queries in Object Bases”, Cluet and Moerkotte, DBPL’1993.

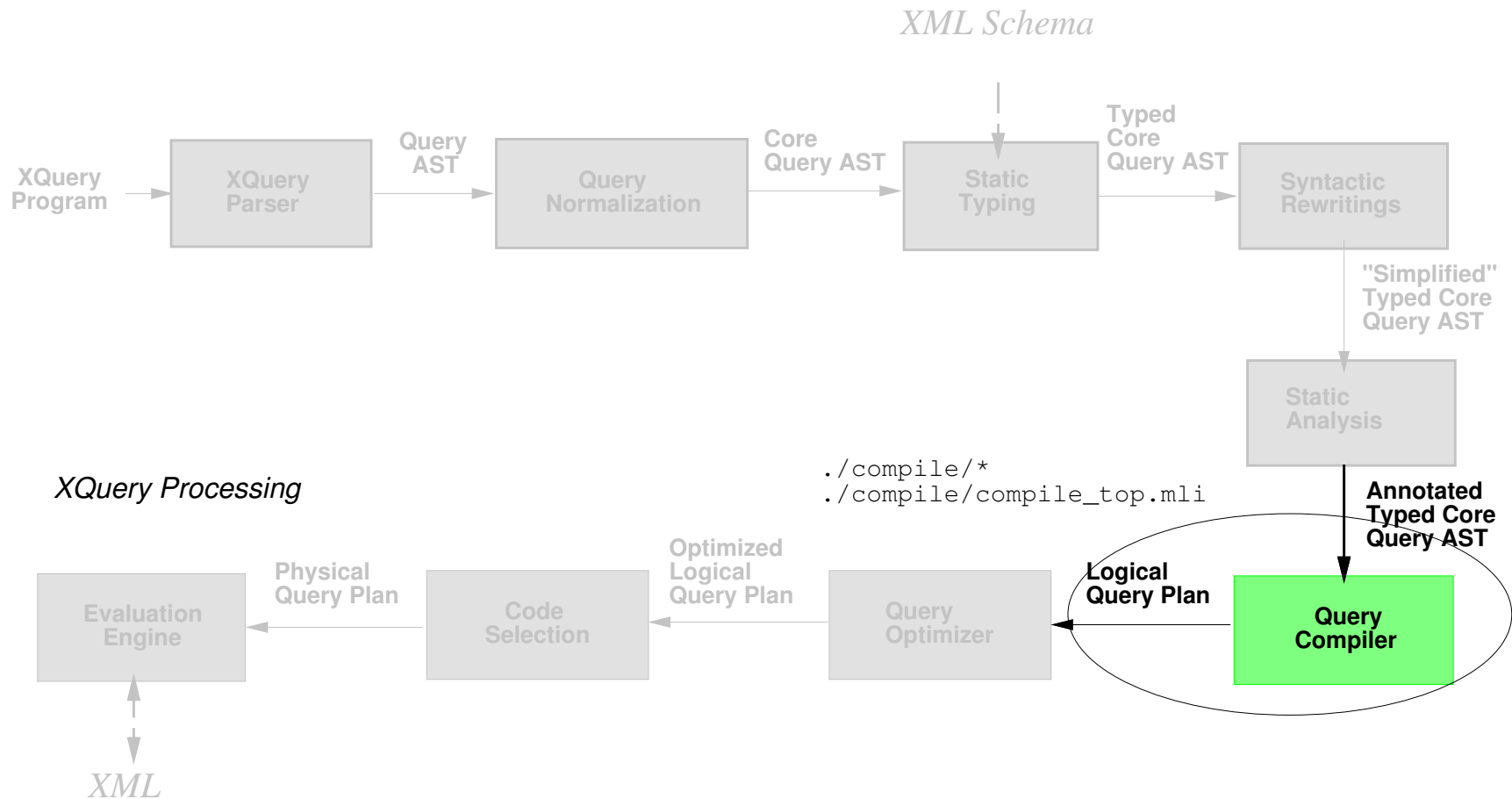
- ▶ Adapted to XML:

- ▶ “XML Queries and Algebra in the Enosys Integration Platform”, Papakonstantinou et al.

- ▶ “Three Cases for Query Decorrelation in XQuery” May et al, XSym’2003 + ICDE’2004.

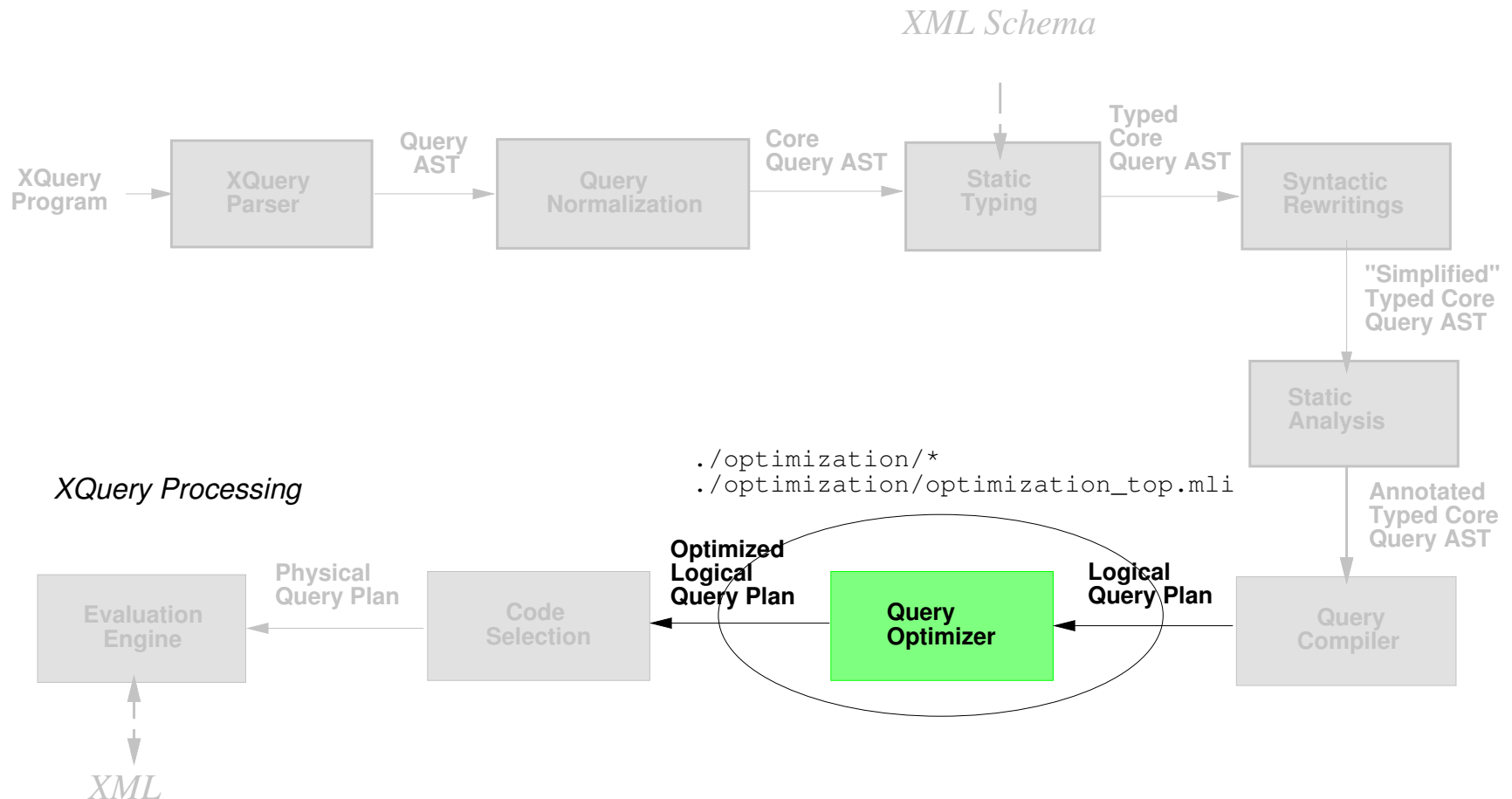
XQuery Processing in Galax (6)

► Algebraic compilation



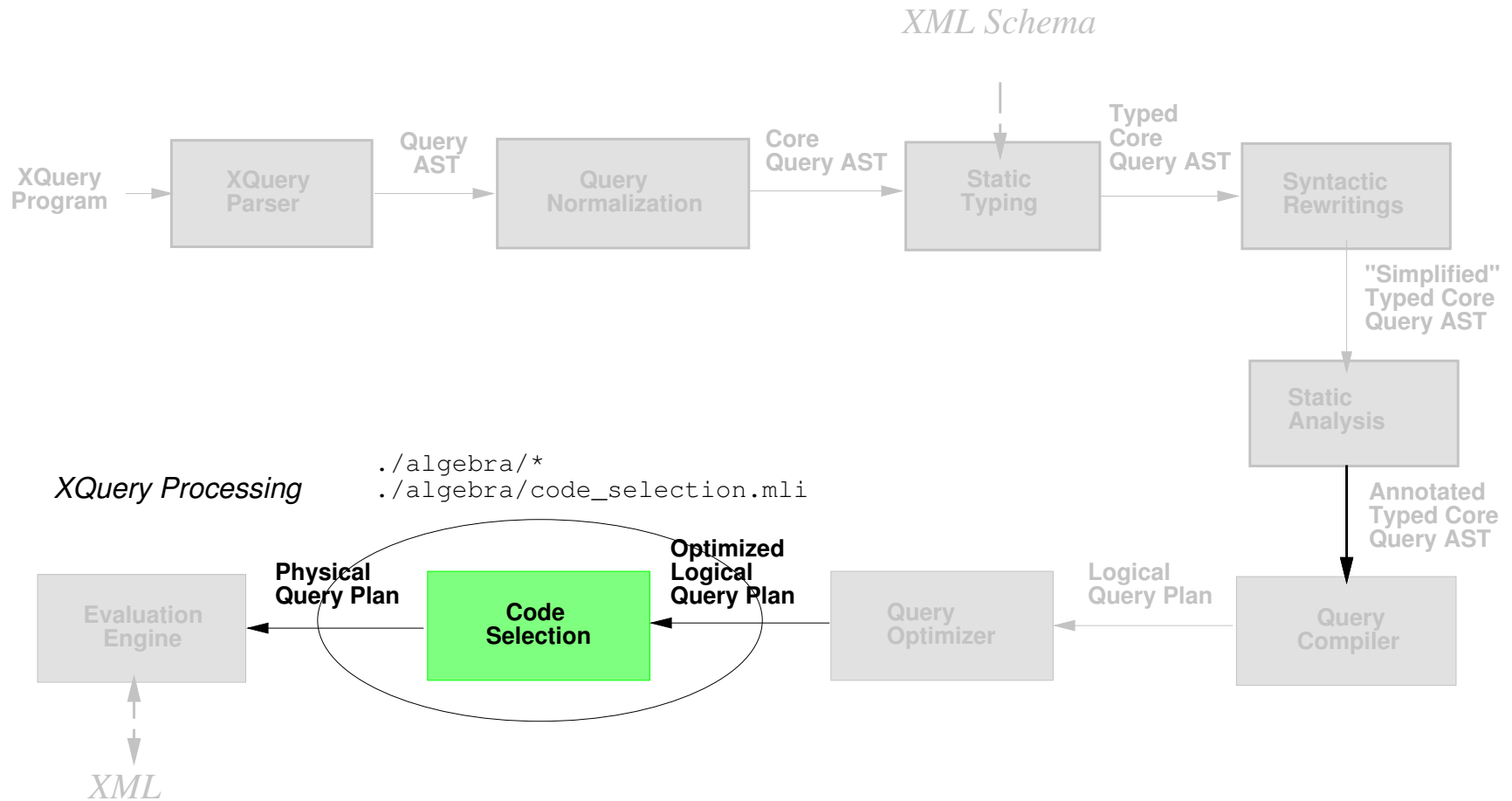
XQuery Processing in Galax (7)

► Algebraic optimization



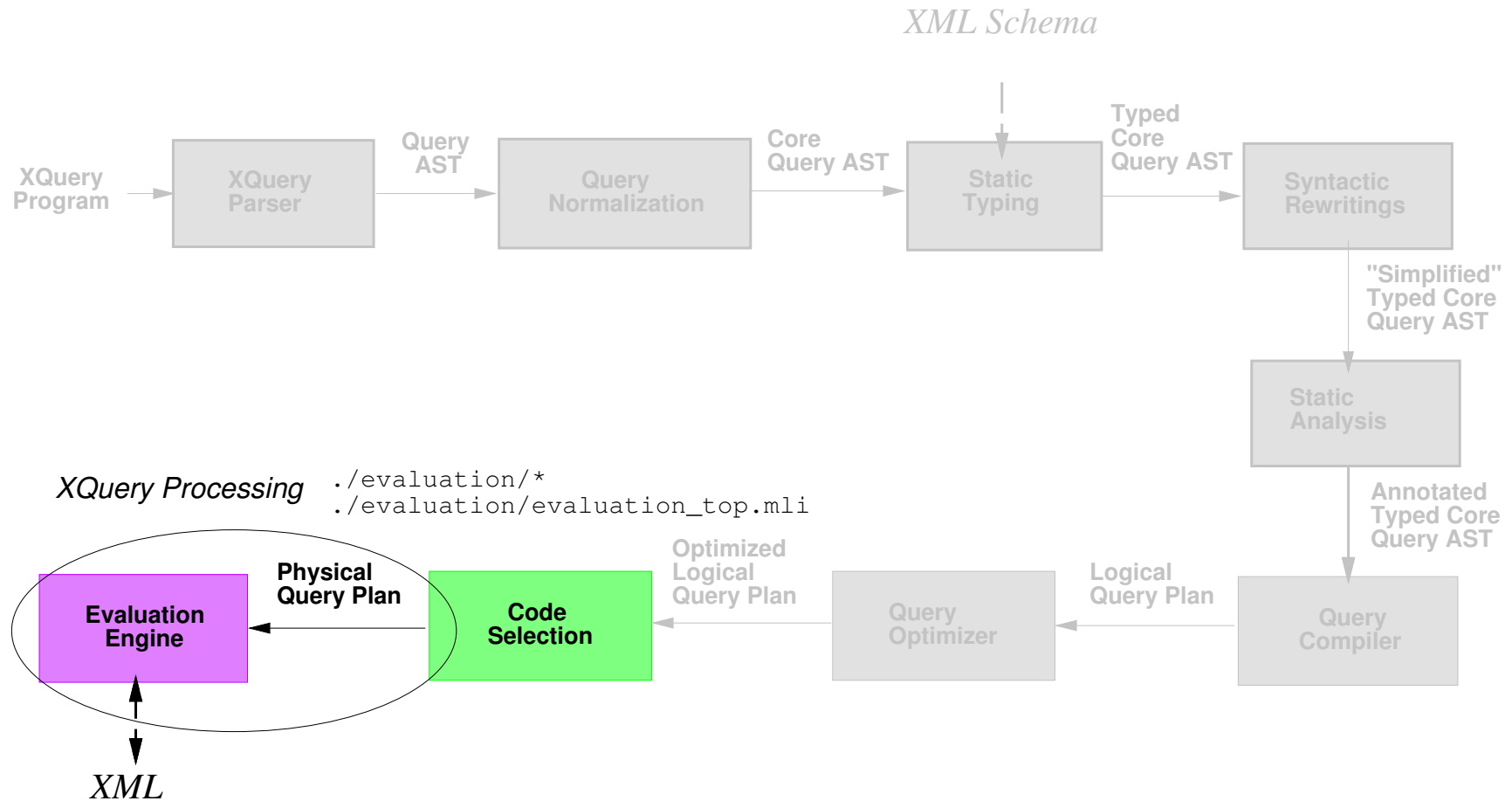
XQuery Processing in Galax (8)

► Code selection



XQuery Processing in Galax (9)

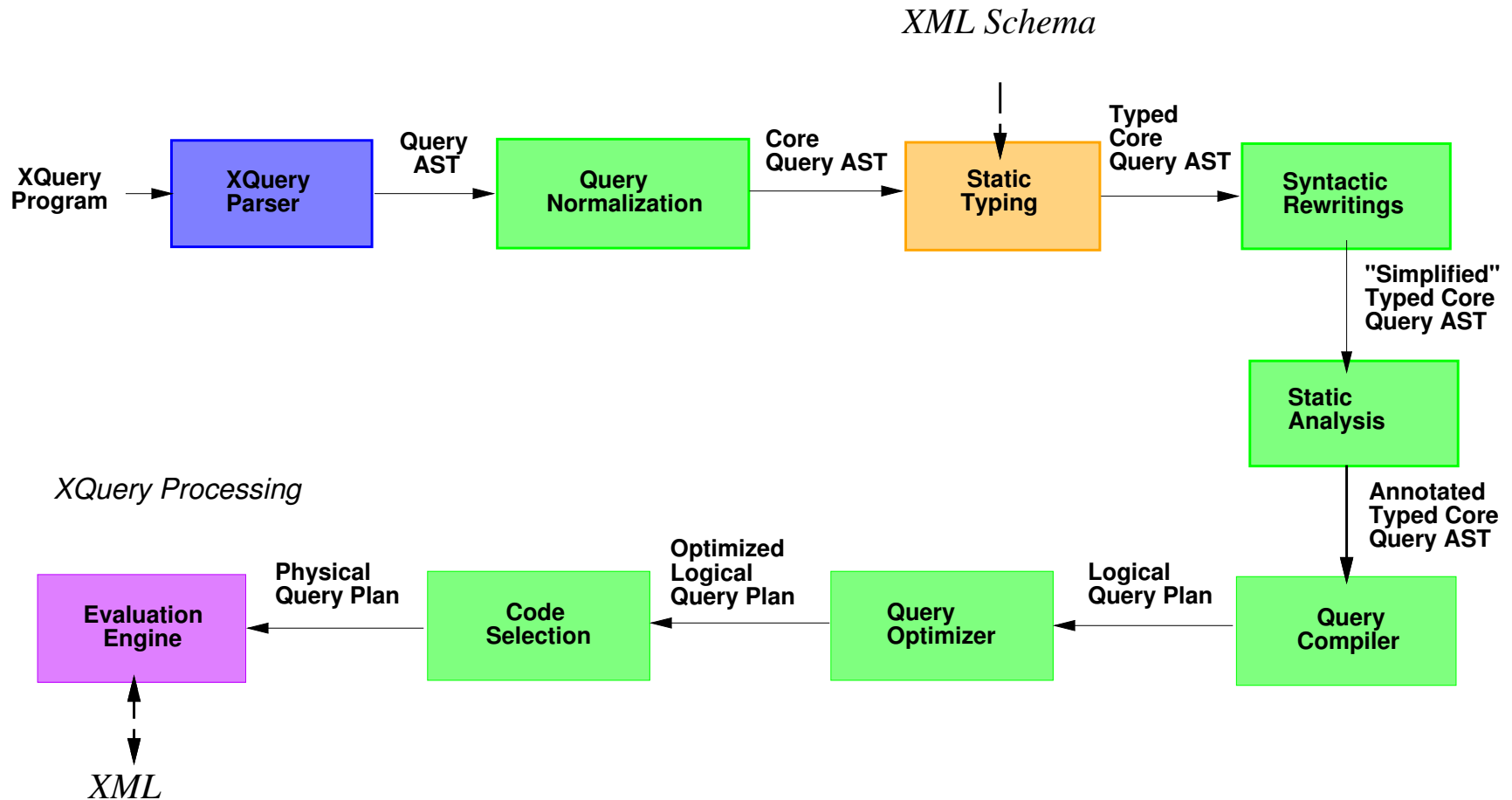
► Evaluation



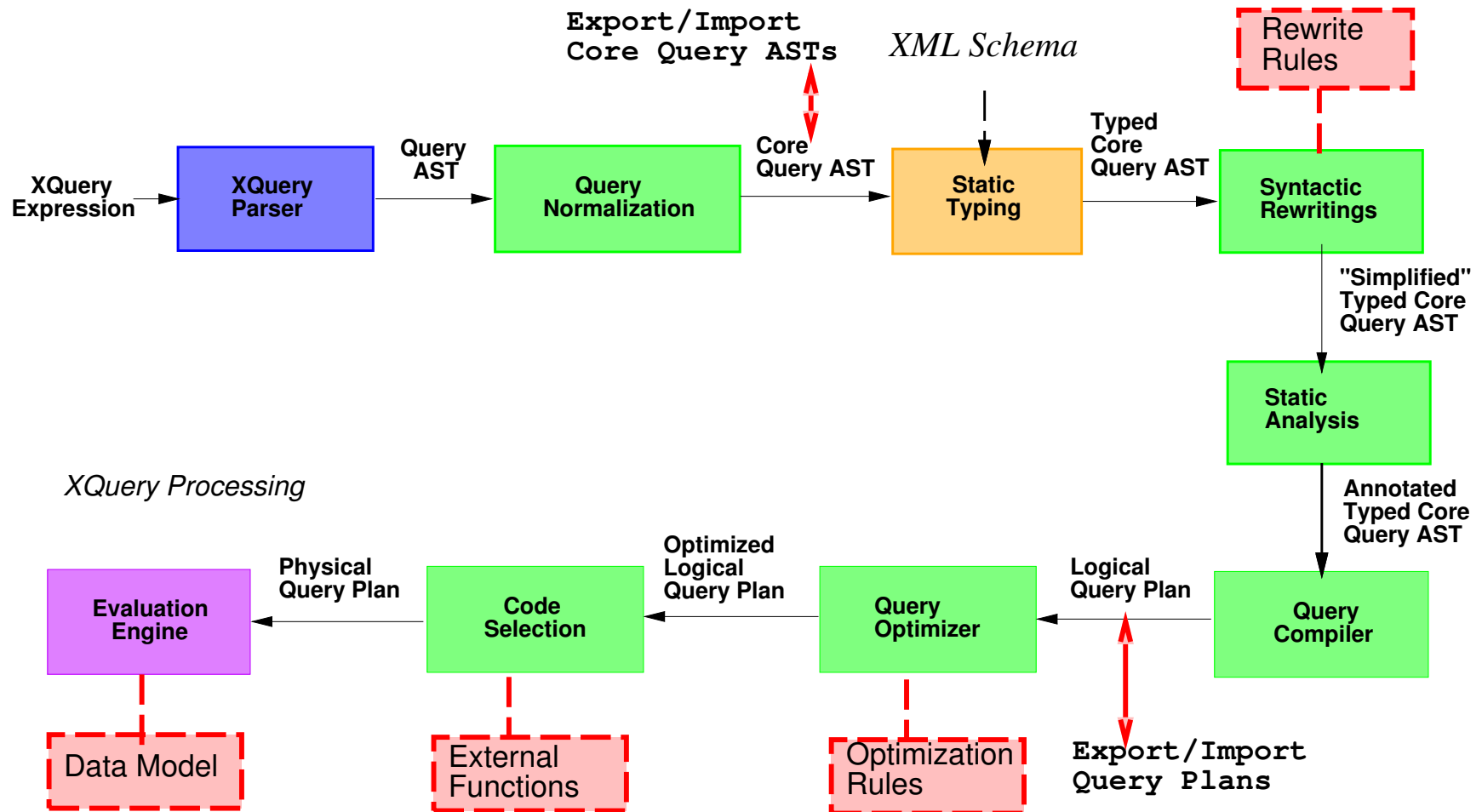
Part VI

Extensibility

Building for Extensibility



Building for Extensibility



Abstract Node Interface

```
virtual node : object inherit item
  (* Infoset accessors *)
  method virtual node_name      : unit -> atomicQName option
  method virtual base_uri      : unit -> atomicString option

  (* PSVI accessors *)
  method virtual typed_value    : unit -> atomicValue cursor
  method virtual node_type     : unit -> type_annotation option

  (* Axes *)
  method virtual parent        : unit -> node option
  method virtual children      : unit -> node cursor
  method virtual attributes    : unit -> attribute cursor
  method virtual descendant_or_self : unit -> node cursor
  method virtual descendant    : unit -> node cursor
  method virtual ancestor_or_self : unit -> node cursor
  method virtual ancestor      : unit -> node cursor

  (* Node identity *)
  method virtual nodeid        : unit -> Nodeid.nodeid
  method virtual docorder      : unit -> Nodeid.docorder
```

Implementations of Galax Data Model

- ▶ Main Memory

 - Analogous to typed Document Object Model

 - Default implementation for dynamic node construction

 - Supports update extensions

- ▶ Jungle Secondary Storage System

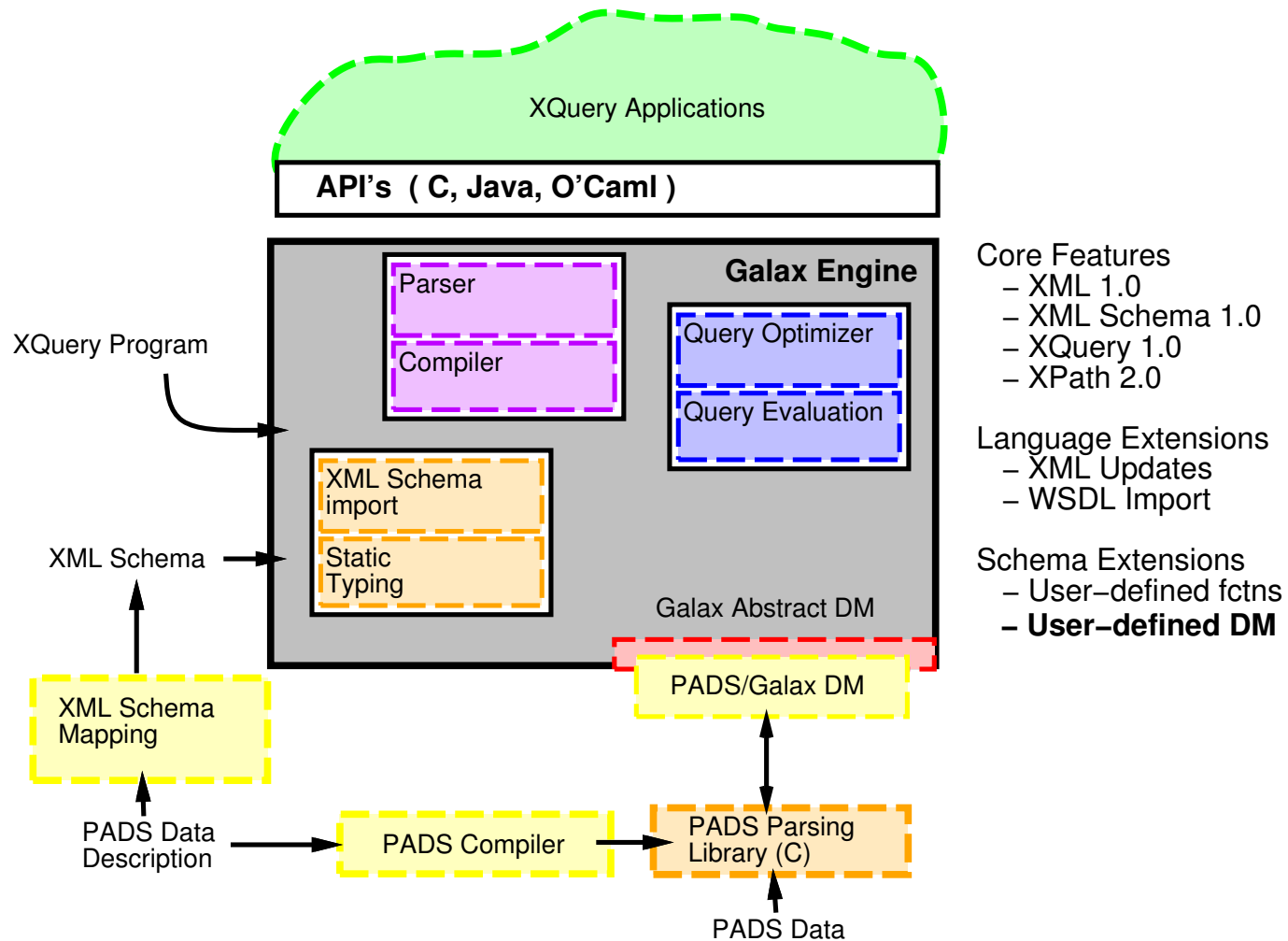
 - Supports update extensions

 - Reference:** “The Simplest XML Storage Manager Ever”, XIME-P 2004,
Fernández, Siméon & Vyas

- ▶ PADS-Galax Data Model

 - Fernández, Fisher, Gruber & Mendelbaum

PADS-Galax Architecture



Part VII

Conclusions

A few things we have left behind

- ▶ Namespaces: tricky and painful, but essential
 - ▶ See `./namespace/*`
- ▶ Built-in library of functions (a few hundreds)
 - ▶ See `./stdlib/*`
- ▶ User facing code: APIs, command-line tools
 - ▶ See `./galapi/caml/*`
 - ▶ See `./galapi/c/*`
 - ▶ See `./galapi/java/*`
- ▶ Documentation
 - ▶ See `./doc/*`
- ▶ Testing!!

A few things added to the mix

- ▶ XML updates
 - ▶ Extension to the language syntax, normalization, etc.
 - ▶ *“An XQuery-Based Language for Processing Updates in XML”*, PLAN-X'2004
- ▶ Storage and indexes
 - ▶ See previous talk by Maurice van Keulen and Torten Grust
 - ▶ (Variant) implementation available in Galax
 - ▶ See `./jungledm/`*
 - ▶ *“The Simplest XML Storage Manager Ever”*, XIME-P'2004
- ▶ Web services support
 - ▶ How to call a Web service from a Query
 - ▶ Interface with SOAP and WSDL
 - ▶ See `./wsdl/`*, `./extensions/apache/`*
 - ▶ *“XQuery at your Web Service”*, WWW'2004

Some Lessons Learned: Development

- ▶ Software-engineering principles are important!
 - ▶ Formal models are good basis for initial architectural design
 - ▶ Design, implementation, refinement are continuous
- ▶ Development infrastructure matters!
 - ▶ Choose the right tool for the job
 - ▶ O'Caml for query compiler; Java (and C) for APIs
- ▶ Team matters even more!
 - ▶ Work with people for 4 years
 - ▶ Some piece of code survives long
 - ▶ E.g., FSA code written by Byron Choi in July 2001
 - ▶ You can't make it if you don't have fun!

Some Lessons Learned: Users

- ▶ Having users is amazing
 - ▶ They are smarter than you
 - ▶ They do crazy things with your software
 - ▶ They do not complain (well sometimes...)
 - ▶ You can learn a lot from their feedback
- ▶ Examples of Galax users
 - ▶ Lucent's UMTS
 - ▶ Universities for teaching
 - ▶ Small projects (e.g., Query music in XML)
 - ▶ Ourselves (e.g., Mary in PADS, Jérôme in LegoDB)...

Some Lessons Learned: Research

- ▶ Where is research in all this?
 - ▶ 90% is **not** research
 - ▶ 10% is research
- ▶ For instance, some interesting research on Galax
 - ▶ Static typing: FST TCS'2000, ICDT'2001, POPL'2003
 - ▶ Indexing, optimization: VLDB'2003, XIME-P'2004, XSym'2004, (ICDE'2005?)
 - ▶ Updates: PLAN-X'2004
 - ▶ Web services: WWW'2004, SIGMOD'2004
- ▶ 10% is **interesting** research
 - ▶ It has very practical impact
 - ▶ You can implement it for real
 - ▶ Problems are often original
 - ▶ How to deal with sorting by document order
 - ▶ Document projection
 - ▶ etc.

Next Step with Galax? *You tell us!*

- ▶ Gold standard of open-source XQuery implementations
- ▶ Implementation of choice for experimentation
- ▶ Visit us at <http://www.galaxquery.org>
 - ▶ Version 0.4.0
 - ▶ Pre-release of source & Linux binaries
 - Compiler generates naive (top-down) evaluation plan
 - ▶ Version 0.5.0 (November 2004)
 - Incorporates bottom-up query plans

Thanks! to Galax Team, Past and Present

Byron Choi, University of Pennsylvania

Vladimir Gapeyev, University of Pennsylvania

Jan Hidders, Universiteit Antwerpen

Amélie Marian, Columbia University

Philippe Michiels, Universiteit Antwerpen

Roel Vercammen, Universiteit Antwerpen

Nicola Onose, University of California, San Diego

Douglas Petkanics, University of Pennsylvania

Christopher Ré, University of Washington

Gargi Sur, Microsoft via University of Florida

Avinash Vyas, Lucent Bell Laboratories

Philip Wadler, University of Edinburgh