Build your own XQuery processor

Mary Fernández, AT&T Labs Research

Jérôme Siméon, IBM T.J. Watson Research Center

I want to thank Gottfried for inviting me to speak to you at BTW. He assured me that I would not have to speak in German, which was a huge relief, because my German is entirely restricted to nouns and only those having to do with food, wine and opera.

Last year, I had the chance to speak about XQuery and Galax at two XML workshops and for those talks, I decided to prepare written remarks – which is a bit more formal than computer scientists are accustomed to – but it worked so well, I decided to do it.

Today, I'm going to talk to you about Galax, our implementation of XQuery 1.0. Galax is joint work with Jérôme Siméon of IBM Research and many great collaborators, who I will name at the end.

I am not going to justify XQuery's existence or give you an XQuery tutorial although I will show some sample queries and give you some references so you can learn more. I assume that you have some basic knowledge of XML and XQuery, of query processing, and of how a typical database engine works.

My goal is to convince you, in particular students, that building a software artifact that is used by someone other than the architect himself(herself) can lead to a lot of interesting research.

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 …

So during Galax's evolution, we have continuously juggled three primary requirements: completeness, performance, and extensibility. Trying to satisfy all these requirements at once has the potential pitfall of making Galax a "Jack of all trades but master of none." But we have found that by never sacrificing one requirement for an other, we have been able to build a devoted user group.

Each requirement has a related set of technical challenges and these challenges have influenced the development of the Galax architecture. For each requirement, I'd like to quickly illustrate one corresponding challenge and mention one user for whom this requirement was critical.

Completeness: Implicit Semantics

- User: Bleeding-edge XQuery Users
- Implicit XPath semantics

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

  - Atomization
  - Type promotion and casting
    Presence/absence of XML Schema types
  - Existential quantification
  - Document order
- Advanced Features
  - Schema import and typing
  - Functions and modules
  - XQuery implementation language for DSLs
    - Constraint checking on network elements
    - Semi-automatic schema mapping

The XQuery working group was the first user to require completeness, but today, all our users expect completeness.

Satisfying the completeness requirement obviously requires that Galax implement *all* of XQuery, including its sub-language XPath 2.0. Path expressions are central to XQuery, so you might think they should be pretty straightforward, but XPath 2.0 expressions have a deceivingly complex implicit semantics.

Here's an innocuous looking path expression that selects all books in a catalog that have a year attribute whose value is greater than 2000. What are its implicit semantics?

First, whenever an aritmetic or comparison operator is applied to node value, the node's atomic content is extracted, which is known as atomization.

When comparing two atomic values, implicit rules of type promotion and casting are applied. For example, when comparing a decimal and a float, the decimal is promoted to a float. XPath also distinguishes between node content that has been validated against an XML Schema type and that which has not. Unvalidated content is always cast to a target type whereas validated content is not.

All comparison operators are existentially quantified, so if more than one node or atomic value exists, then the predicate is true if any one value satisfies the predicate.

Finally, a path expression always yields its node values in document order with no duplicates.

We could argue long and loud about whether a little path expression should be so heavy weight – and trust me, I've already heard every argument in favor and against – but this is what the specification states and this is what Galax implements.

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>
```

- Naïve evaluation $O(n^2)$
- Recognize as group-by on category and unnest

You are all probably quite familiar with the problems of evaluating XQuery efficiently.

One language feature most notoriously absent from XQuery is group-by, which can be expressed in XQuery using nested FLWOR expressions.

The IBM Clio project, for example, has a semi-automatic system for generating mappings between closely related XML Schemas. They generate deeply nested FLWOR expressions, which are hard to evaluate efficiently.

The XMark 10 query is a simple example of a group-by expressed using nested FLWORS, but this query is a walk in the park compared to the queries generated by Clio.

This expression re-groups all people in an auction by the categories in which they are interested. A naïve evaluation would iterate over the input document once for each category value.

The key problem here is to recognize the nested FLWORS express a group-by and produce an unnested evaluation plan.

Until recently, Galax used a top-down, naive evaluation plan for nested FLWORS, but our most recently released version now produces un-nested evaluation plans.

Interestingly, our users have never demanded that Galax be the fastest implementation, but instead required that such queries always yield the *right* result. They wait patiently for Galax to produce faster query plans but not at the expense of completeness.

Extensibility: Querying Virtual XML

- User: AT&T PADS (Processing Ad Hoc Data Sources)
  - Declarative data-stream description language
  - Detects & recovers from non-fatal errors in ad hoc data
- 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><resolved>207.136.97.49</resolved></host>
  <remoteID><unknown/></remoteID>
  <auth><unauthorized/></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>
```

- Using XQuery to explore data
  - Hosts of records with content length greater than 2K

```
fn:doc("pads:data/clf.data")/http-clf[contentLength > 2048]/host
```

One requirement that we did not anticipate but that has become central to Galax is extensibility, in particular, support for querying virtual XML data sources.

Kathleen Fisher, my colleague at AT&T is the inventor of PADS, a language that permits data analysts to describe declaratively the format of complex legacy data sources, of which there are numerous examples in our lab. PADS can describe just about anything: fixed and variable-width records in EBCIDIC, ASCII, or UTF-8, COBOL copy books, and binary data of any kind.

Hers's a simple example of the kind of data that PADS can describe: the HTTP common log format, which has a high degree of syntactic and structural variability.

From a PADS description (not shown here), the PADS compiler generates a library of functions for parsing a data source. A key feature of PADS is that it gracefully recovers when a syntax error is detected in a data stream and can identify the cause and location of an error to the application using the parsing library.

Kathleen wanted to be able to view PADS data sources as XML so that PADS users could use XQuery to perform simple querying tasks declaratively. We were able to support this application by exposing Galax's data-model abstract in an API.

The outcome is that PADS users can view their data as virtual XML and write queries such as the following, which returns the hosts of CLF records whose content-length is greater than 2K.

So now you know a little about Galax's history and the kinds of requirements that we face as Galax evolves.

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

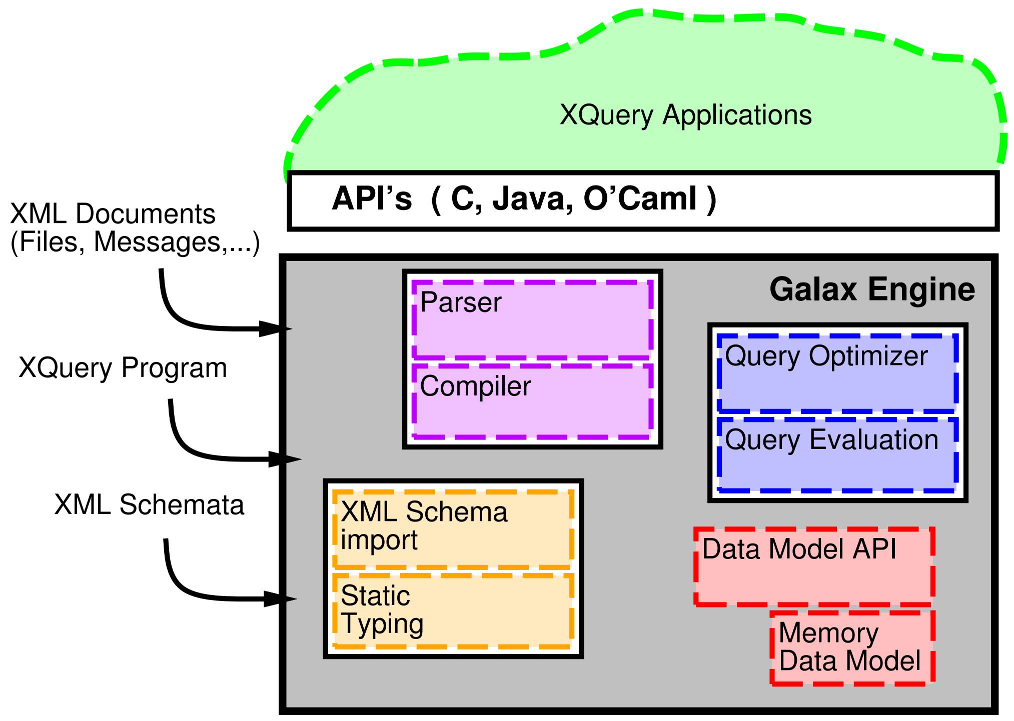
- Is also helpful, but not required:
  - Some knowledge about query processing (e.g., relational)
  - Some idea of how a typical database engine works

Part II

Preliminaries: Galax and Caml

What is Galax?

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



Core Features
– XML 1.0
– XML Schema 1.0
– XQuery 1.0
– XPath 2.0

Language Extensions
– XML Updates
– WSDL Import

System Extensions
– User–defined fctns.
– User–defined DM

Galax is a complete, extensible implementation of XQuery 1.0 that processes XML documents, their schemas, and queries over those documents.

Galax includes support for all of XML 1.0, including namespaces and for most of XML Schema 1.0, which is the foundation of the XQuery type system, and for all of XQuery 1.0, including modules.

This figure is an executive summary of Galax's current features. In addition to the core support for XML, XML Schema, and XQuery, Galax has two experimental language extensions: one for updates and one for importing and exporting WSDL-defined Web services to/from XQuery.

Galax also has two system extensions that support execution of user-defined functions, written in either C or O'Caml, and for user-defined implementations of the Galax data model, which makes it possible for Galax to evaluate queries simultaneously over non-native and native XML sources.

Like most large systems, Galax has evolved over time. I'd like to describe Galax's history as it illustrates well how different forces have impacted Galax's evolution and our own research interests.

Galax History

	2000	2001	2002	2003	2004	2005
Goals	**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	
Implementation	**XML Algebra Demonstration**	Executable Semantics	Reference Implementation		Complete & extensible implementation with optimizer	
Users				Universities		
			UMTS (Lucent)			
	Us!	W3C XML Query WG	Early XQuery adopters/ implementors	GUPster (Lucent) PADS (AT&T)	Advanced XQuery users (module support, etc.)	

Jérôme and I have both been members of the W3C's XML Query working group since its inception in 2000.

Galax's implementation closely follows the goals of our work in the XQuery group and the requirements of current and prospective users in our respective research labs.

Our first goals were to specify a simple, orthogonal query language for XML that captured much of the existing semantics of XPath as well as the features of earlier languages, such as XML-QL, YaTL and others, and to satisfy the language requirements being debated by the XQuery working group.

We called this langauge the XML query algebra. Our implementation of this little algebra became a useful demonstration tool for us within the XQuery working group, and this implementation was the first version of Galax. At this point, we were the only users of the system.

Galax History

	2000	2001	2002	2003	2004	2005
Goals	**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	
Implementation	**XML Algebra Demonstration**	**Executable Semantics**	Reference Implementation		Complete & extensible implementation with optimizer	
Users				Universities		
			UMTS (Lucent)			
	Us!	**W3C XML Query WG**	Early XQuery adopters/ implementors	GUPster (Lucent) PADS (AT&T)	Advanced XQuery users (module support, etc.)	

Early on, we along with our colleagues Phil Wadler and Peter Fankhauser championed for XQuery to be a strongly typed query language with a formal semantics.

After the first demo of the algebra, our focus shifted to specifying a complete operational semantics, both static and dynamic, for the fledgling XQuery language.

As editors of the XQuery formal semantics, we would alternate between writing XQuery's formal definition and implementing the semantics in Galax. At this point, the implementation served as an executable semantics for debugging the language, and our users included a few other members of the XQuery working group.

Galax History

	2000	2001	2002	2003	2004	2005
Goals	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	
Implementation	XML Algebra Demonstration	Executable Semantics	Reference Implementation		Complete & extensible implementation with optimizer	
Users				Universities		
			UMTS (Lucent)			
	Us!	W3C XML Query WG	Early XQuery adopters/ implementors	GUPster (Lucent) PADS (AT&T)	Advanced XQuery users (module support, etc.)	

After the XQuery working drafts became public, our goals shifted toward promoting the language to potential users, for example, by teaching XQuery to the existing XML community (mostly SGML people) and to the database-research community and by showing how XQuery meets the needs of a wide variety of classes of XML querying tasks.

During this period, Galax evolved into a reference implementation that could actually be used by early adopters of XQuery. Lucent researchers and developers were early and aggressive users of XQuery and Galax's first real users.

The UMTS system, for example, has a declarative language for expressing consistency constraints on telephone switch configurations. The language is translated into large XQuery programs, which are then run by Galax to check the constraints. Because this application runs in a production environment, we quickly had to make Galax more reliable and efficient, provide stable APIs and a basic secondary storage system for large XML documents. In general, we had to make Galax behave more like a product and less like a research prototype.

Galax History

	2000	2001	2002	2003	2004	2005
Goals	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	
Implementation	XML Algebra Demonstration	Executable Semantics	Reference Implementation		Complete & extensible implementation with optimizer	
Users				Universities		
			UMTS (Lucent)			
	Us!	W3C XML Query WG	Early XQuery adopters/ implementors	GUPster (Lucent) PADS (AT&T)	Advanced XQuery users (module support, etc.)	

During this period, we did not attempt to investigate or solve open research problems, the most significant being devising algorithms for efficient XQuery evaluation, nor did we choose to apply particular algorithms or techniques that were proposed in the research literature.

Instead, we focussed on building a complete and extensible implementation, by essentially working top down. This strategy served us well, because it helped us to attract more real users to XQuery and to Galax than we would have been able to attract if we had pursued a more bottom-up implementation strategy.

Now that we have had a chance to see how people are using XQuery and Galax, we are returning to the fundamental problems of efficient query evaluation, studying and applying optimization techniques, but all the while keeping our users' needs in mind. Our user base is getting bigger, too.

Message: Research often works bottom up but we are working top down.

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(b̈ook.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)

Galax processes XML documents, their schemas, and queries over those documents, so, not surprisingly, Galax's architecture is composed of processing models for documents, schemas, and XQuery programs.

Processing models produce various representations of documents, schemas, and queries, and their phases connect these representations. For example, validation relates XML documents to their schemas and static typing relates queries and schemata.

If you learn one thing today, I'd like it to be that the implementation of each of Galax's processing models is based on a formal specification. This design choice has made it possible for us to keep up with the rapid changes to XQuery's definition.

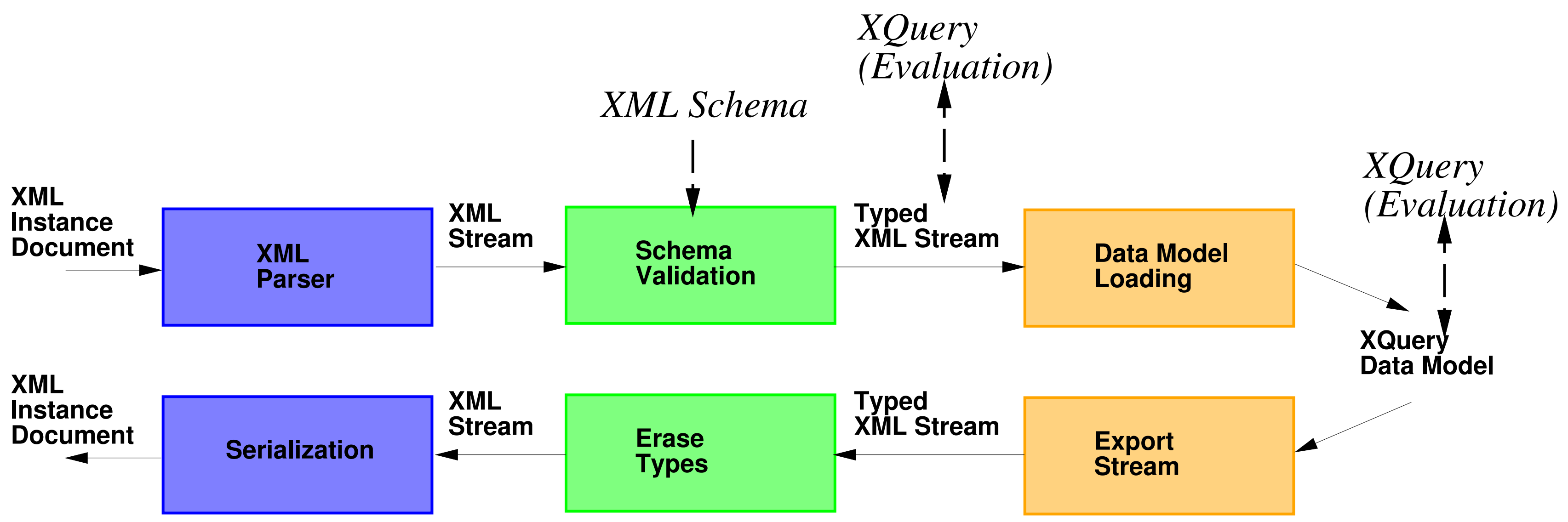
In addition, the interfaces between processing models are strictly typed, which makes it very clear how phases can be composed and ordered.

To make this more concrete, I'm going to walk you through the three processing models one step at a time. By the end, you should have a good understanding of how Galax works.

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

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

This picture illustrates Galax's XML processing model.

This processing model takes native XML documents as input and produces native XML documents as output.

Many of the phases in Galax that transform representations of XML documents take and yield streams of XML tokens. XML tokens include, for example, start-document, start-element, text nodes, etc. These tokens are similar to SAX events, however, unlike SAX events which are typically implemented as call-back functions, the streams are consumed by the next phase pulling events, on demand, from the stream source.

If you are a functional language user, you can think of XML streams as lazy lists.

So the XML parser yields a stream of XML tokens. Schema validation consumes this stream and also takes type information from the XML Schema processing model, and produces a *typed* stream of XML tokens. That is, validation adds typing information to the input stream.

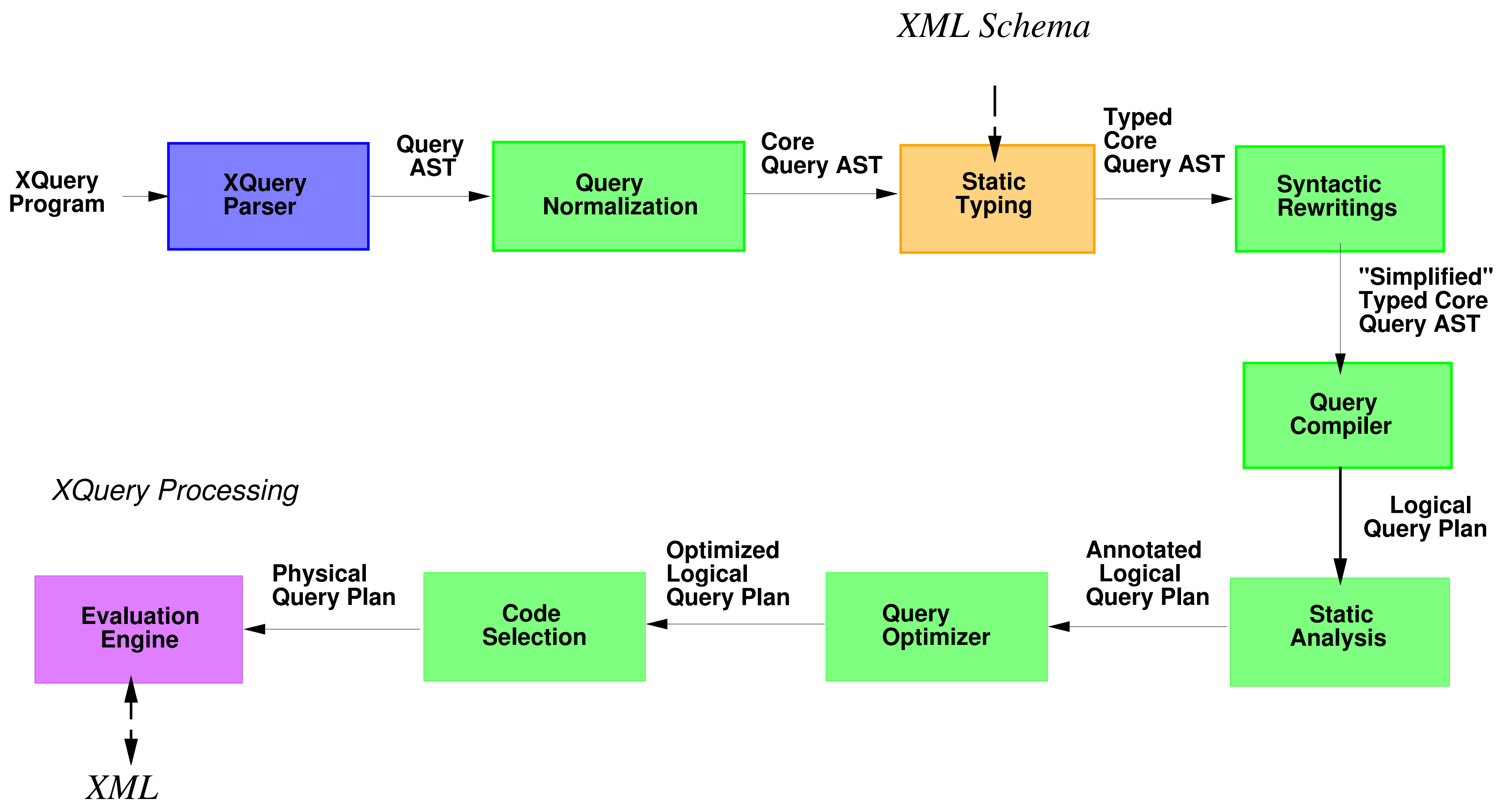
Here's the typed representation of the input document. Each node has a type annotation and some nodes are also annotated with their atomic values.

A typed XML stream is consumed directly by the XQuery processing model and/or by a data-model loader phase, which creates an instance of Galax's abstract tree data model. Currently, Galax has two built-in data model loaders: one that creates an in-memory representation of the document, and a second that stores the document in relational indices in Berkeley DB. I'll return to the data model later.

The out-bound phases are the inverses of the in-bound: export takes a data model and yields a typed XML stream; erasure takes a typed stream and yields an untyped XML stream; finally, serialization takes a typed stream and yields an XML document.

This processing model religiously implements four related specifications: XML 1.0, the XML Infoset, the post-schema validated infoset (defined in XML Schema Part 1), and the XQuery data-model serialization.

XQuery Processing Architecture



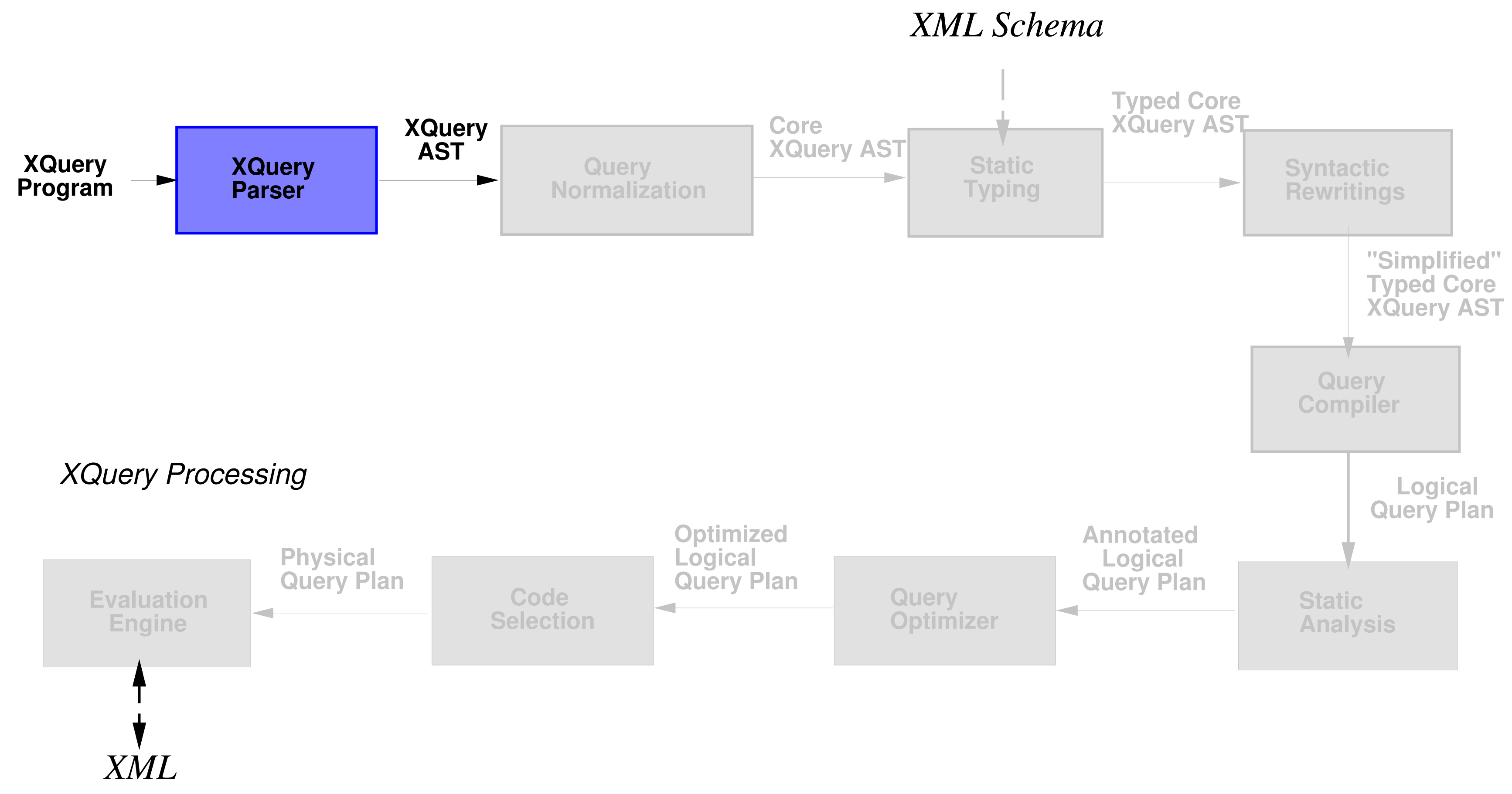
- Inputs: XQuery program (main module, library modules) + Instances of XQuery data model
- Output: Instance of XQuery data model

The heart of Galax is the XQuery processing model, which takes as input an XQuery program, which consists of one main module and zero or more library modules, and produces an instance of the XQuery data model as output.

We're going to look at each phase in this module in turn.

One remark on colors: all parsers are blue; phases that have more than one implementation are orange—for example, static typing has a default implementation in which all expressions are labeled with a default type; weak typing, which applies a very simple form of type inference; and full strong typing, which applies all of XQuery's static typing rules; all required phases that have one implementation are green.

XQuery Processing Step 1: Parsing

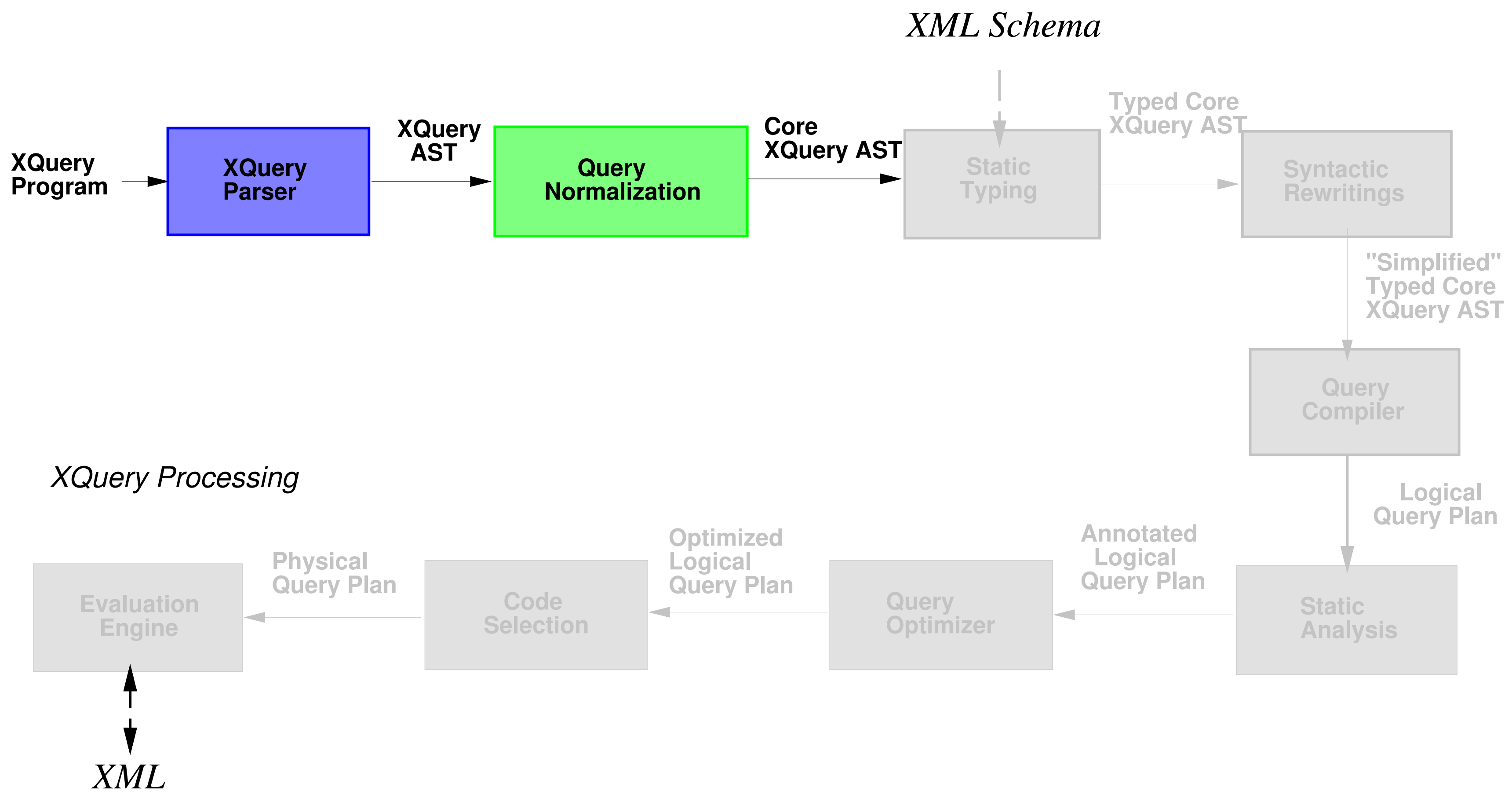


- **Reference:** XQuery 1.0 Working Draft

So XQuery parsing takes an XQuery program and produces an abstract syntax tree of the XQuery language.

Clearly, this phase is defined by the grammar rules in the XQuery spec.

XQuery Processing Step 2: Normalization



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

Query normalization takes an XQuery AST and rewrites it into a smaller, semantically equivalent language, called the XQuery Core.

The completeness requirement is met almost entirely by the normalization phase, because it makes all of the implicit semantics hidden in XQuery's surface syntax explicit in the Core language.

Normalization is defined in the XQuery Formal Semantics, and Galax's implementation is an almost literal interpretation of the formal definition.

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 ()
```

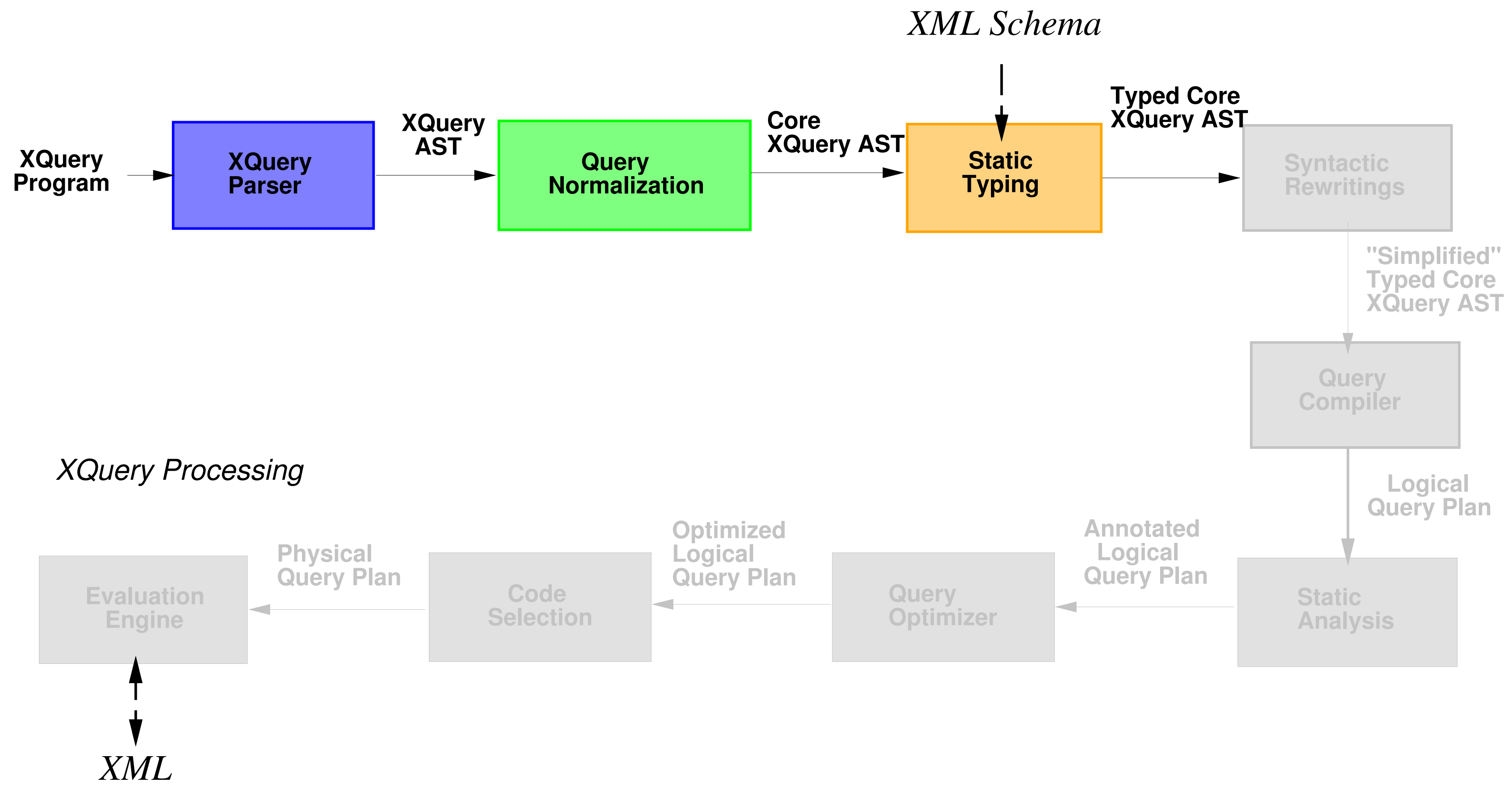
To illustrate, here's the innocuous path expression from earlier in the talk and its normalized representation in the Core language.

(I've actually already simplified the core a bit, but it's close to the normalized expression emitted by Galax.)

I've highlighted some of the Core expressions that capture some of the implicit semantics. For example, the fn:data function applies atomization to a sequence of nodes; the some-satisfies expression is existential quantification; the fs:promote-operand function applies the appropriate type promotion and casting rules; and finally, because during normalization no type information is available, we apply an overloaded, polymorphic comparison operator.

As written, this expression might be quite inefficient, but it captures the complete and correct semantics of the path expression.

XQuery Processing Step 3: Static Typing



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

Static typing follows normalization: for each expression in the Core AST, static typing infers the type of the expression and produces the corresponding typed Core AST, much in the same way as validation takes an XML stream and yields a typed XML stream.

Static typing is boot-strapped with the types of global variables and proceeds bottom-up, assigning types to leaves of the AST, then inner nodes, etc. XQuery supports functions and modules but it's typing rules do not compute the fixed point type for recursive functions, so the user has to provide them explicitly.

Like normalization, static typing is defined in the XQuery Formal Semantics, and the implementation of each typing rule is a literal interpretation of the formal definition.

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, given `$cat : element(catalog)`

```
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)?]
```

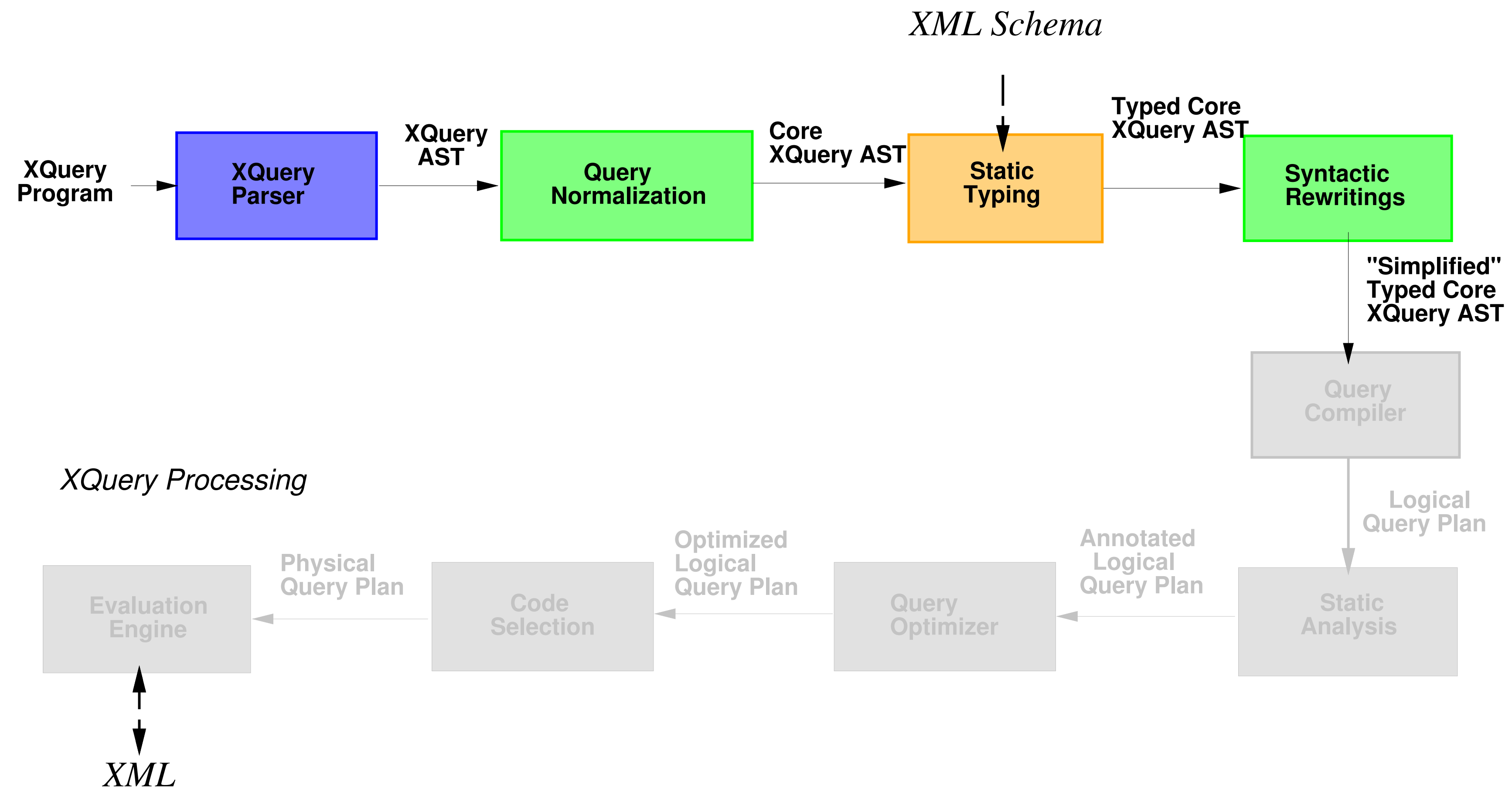
Assuming that we have an XML Schema for our book catalog example, one possible typing of this Core expression is given below.

I've added the type annotations in green – this is not valid XQuery syntax – it's just meant to illustrate the annotations associated with expressions. For example, from a query prolog, we know that the type of the $cat variable is one catalog element.

Even if strong static typing is not applied, there is always a default typing. For example, the default annotation for variable $b would be a single `element` – because the child axis always yields a sequence of elements and a for-bound variable is always bound to a single item, or element in this case.

Even without the more detailed knowledge that $b is is book element, knowing that it is always a single element can be useful.

XQuery Processing Step 4: Rewriting



- Removes redundant/unused operations,
  type-based simplifications, function in-lining
- **Example:** "Optimizing Sorting and Duplicate Elimination in XQuery Path Expressions", DEXA 2005, Michiels, Hidders et al

Rewriting applies many of the standard simplification rules applied in compilers for functional and some object-oriented languages, such are removing unused variable definitions, inlining of non-recursive functions, and applying type-aware simplifications.

One example of a Galax rewriting is described in the tech report "A Systematic Approach to Sorting and Duplicate Elimination in XQuery". The algorithm analyzes a Core expression and detects when intermediate steps in path expressions always yield nodes in document order and nodes with no duplicates, and the corresponding rewriting rule eliminates redundant operators that sort by document order and remove duplicates.

This work merits its own talk, so I will give you a simpler example of a common rewriting.

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)?]
```

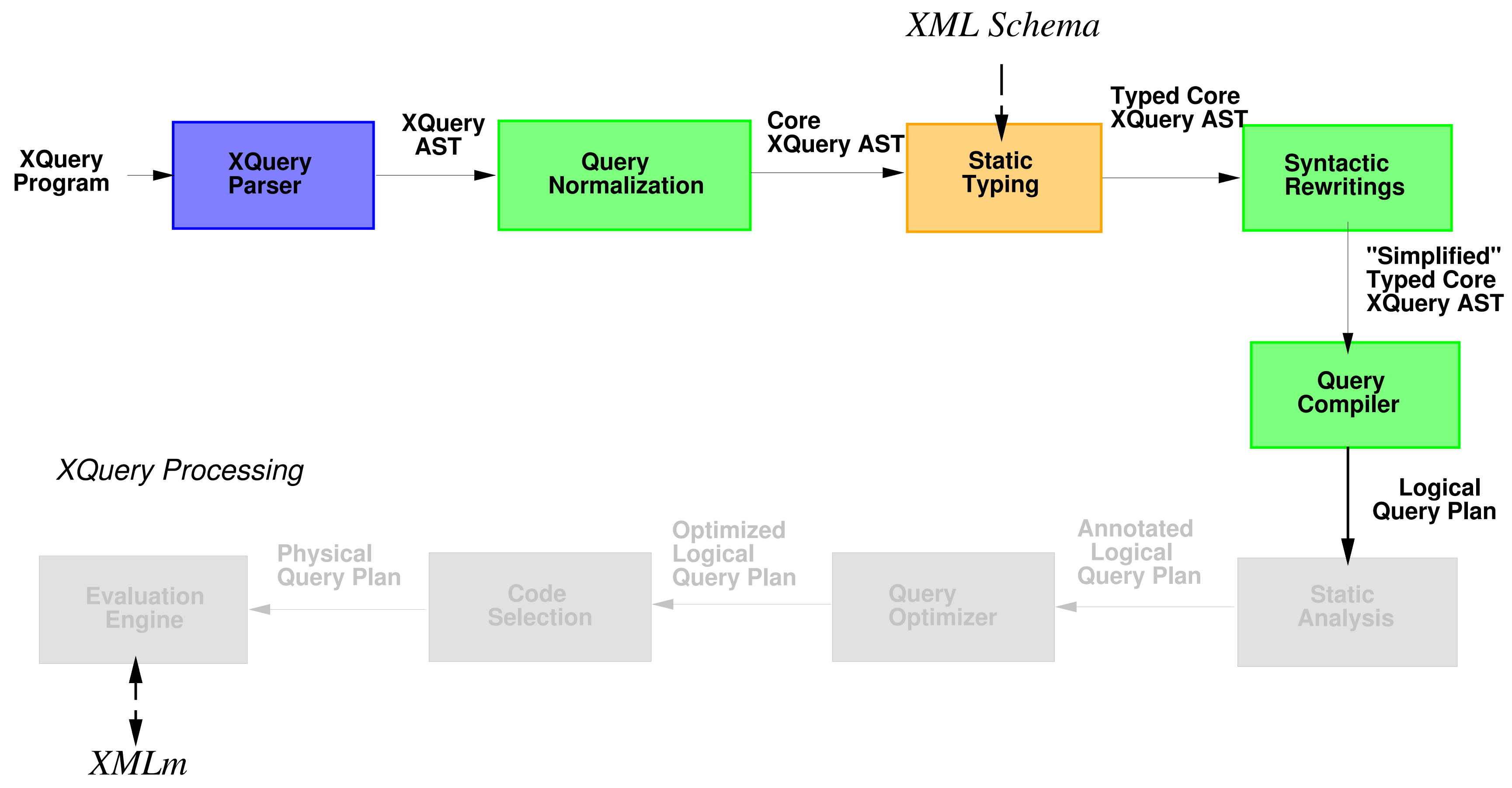
Here's our typed core expression, and the corresponding simplified expression.

We've used the typing information to make several simplifications, which are all straight-forward:

The first for expression is eliminated, because its input type is already a singleton. The existential quantifications are eliminated because a year attribute always contains an integer. The overloaded comparison operator has been replaced by an integer-comparison operator. Etc.

Note that rewriting of Core expressions is *not* the same as optimization of query plans. Our goal here is to produce the smallest Core expression that is semantically correct.

XQuery Processing Step 5: Compilation



- Introduces tuple & tree algebraic operators
  - Produces *naïve* evaluation plan

"A Complete and Efficient Algebraic Compiler for XQuery", ICDE 2006, Ré et al

So far, our program representations have been clearly related to the original query. Compilation substantially changes the program representation by converting a top-down AST into a bottom-up plan of algebraic operators.

Our algebra is a hybrid of well-known operators on tuples, such as join, select, map and of operators on XML trees, some of which are unique to Galax, such as the path-project operator. A large part of the tuple fragment is derived from May, Helmer, and Moerkotte's nested, ordered tuple algebra.

We extend it to cover the entire XQuery language, including all type operators, and we provide a more robust group-by operator.

XQuery Processing : Compilation (cont'd)

- 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)?]
```

- Algebraic plan:

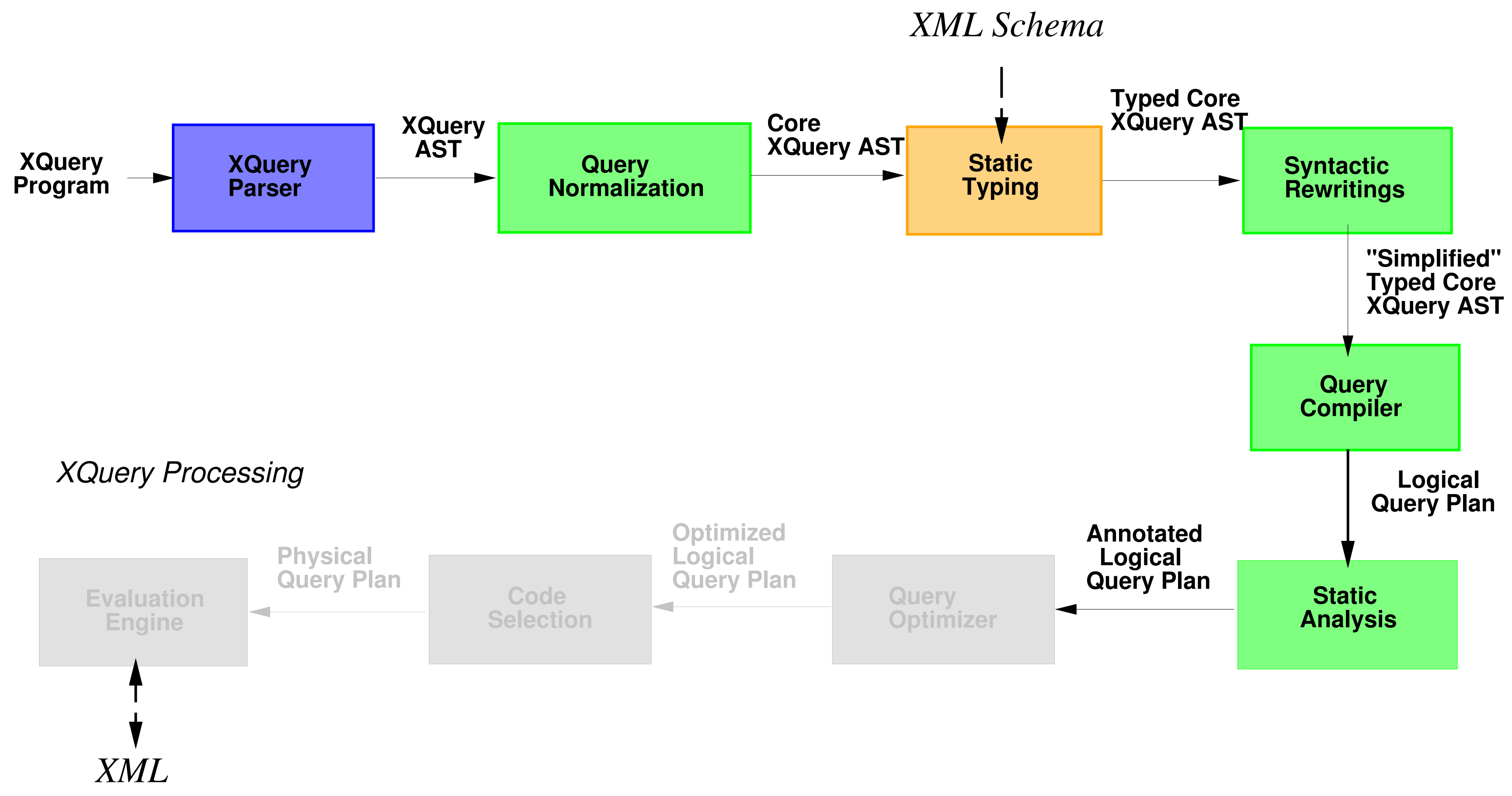
```
MapToItem
 { Input -> Input#b }
 (Materialize
   (Select
     {op:integer-ge(fn:data(Step[@year](Input#b)), 2000)}
     (MapConcat
        {MapFromItem
          {$v -> [b : $v]}
          (fs:docorder((for $fs:dot in $cat return Step[child::book]($fs:dot))) +
        ([]))))
```

Here's our query again, and here is it's representation in our algebra.

Not expecting you to parse this but what you should notice is that compilation has turned the expression tree inside out. The original expression is logically expressed top down, but the resulting plan is expressed bottom up.

Many of these are tuple operators, which are implemented using pull-based cursors, which permits for pipelined evaluation of the query.

XQuery Processing Step 6: Static Analysis



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

  "Streaming XPath Evaluation", Stark et al

Up to this point, I've essentially described the front-end of a functional language compiler – all these phases on the top-half of this dog leg.

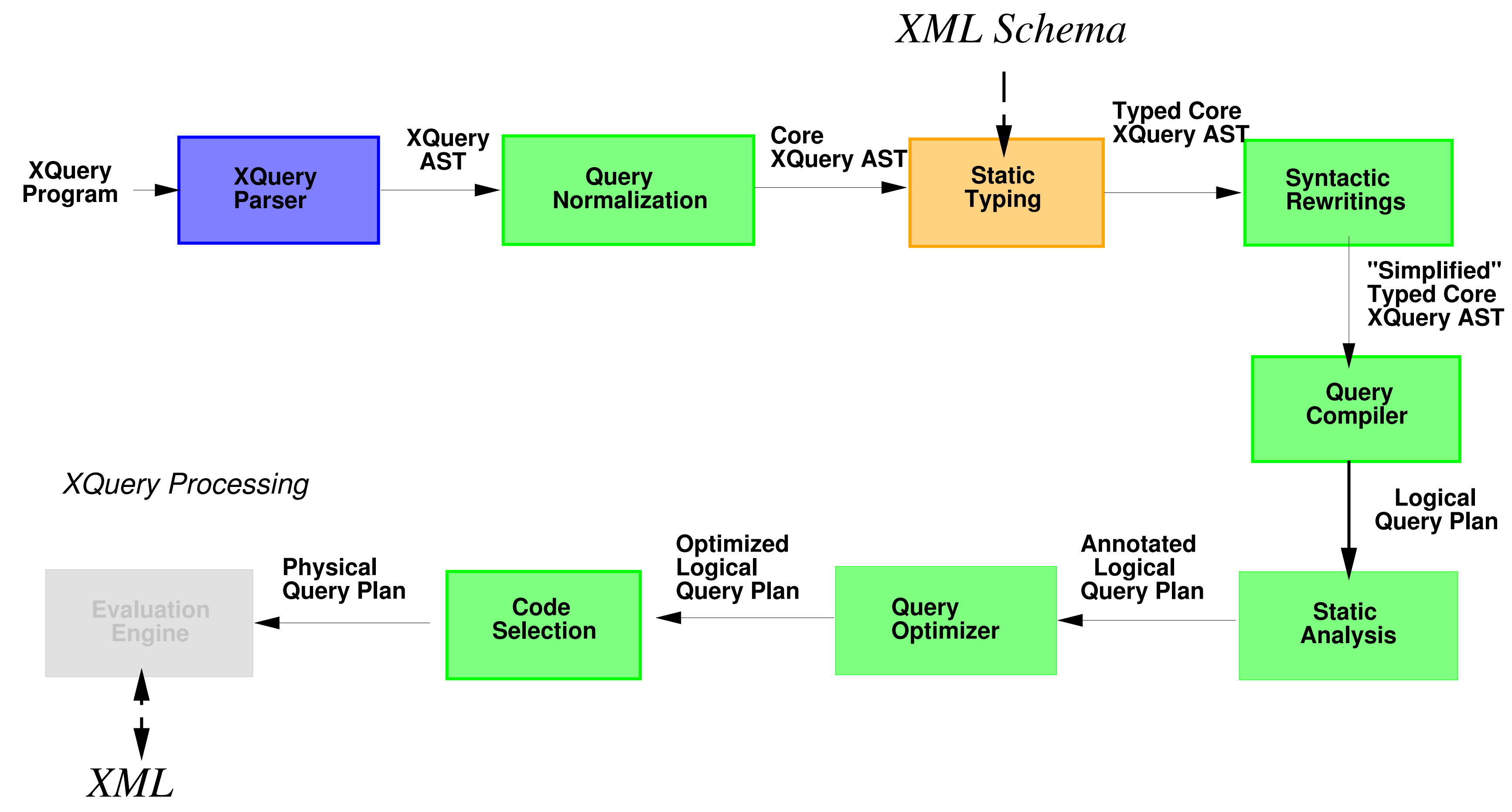
As we turn the bend in the dog leg, we move into the backend of a query-language compiler. Prior to June, the bottom half of this dog leg didn't exist and the evaluation phase simply interpreted the typed Core AST top down.

Static analysis follows rewriting. This phase annotates the typed Core AST with information that can be used in later phases, such as compilation and optimization.

Path projection is one example of a static analysis and is described in this paper by Amélie Marian and Jerome. Path projection takes a typed Core AST and determines a (possibly overlapping) set of paths that may be accessed in a source document. This analysis can be used to prune an input document after validation but before data loading and evaluation.

Another example is data-source analysis: it is sometimes useful to know whether two expressions yield nodes that are from the same document. Data-source analysis annotates expressions with the set of documents in which its results may be contained. We use data-source analysis to determine whether a query can be evaluated entirely on an input token stream, e.g., because the source is only scanned once.

XQuery Processing Steps 7: Optimization



- **Step 7:** Query Optimizer
  - Produces *better* evaluation plan
  - Query unnesting: detect joins and group-bys

Query optimization follows compilation and attempts to produce a better evaluation plan. For example, query unnesting identifies group-bys expressed by nested queries and produces a plan with unnested operators. Pushing selections early in a plan is another example.

Heuristic optimization – no cost model; simply want to avoid quadratic behavior of nested for-loops.

XQuery Processing : Compilation (cont'd)

- Algebraic plan:

```
MapToItem
 { Input -> Input#b }
 (Materialize
   (Select
     {op:integer-ge(fn:data(Step[@year](Input#b)), 2000)}
     (MapConcat
        {MapFromItem
          {$v -> [b : $v]}
          (fs:docorder((for $fs:dot in $cat return Step[child::book]($fs:dot))) +
        ([]))))
```

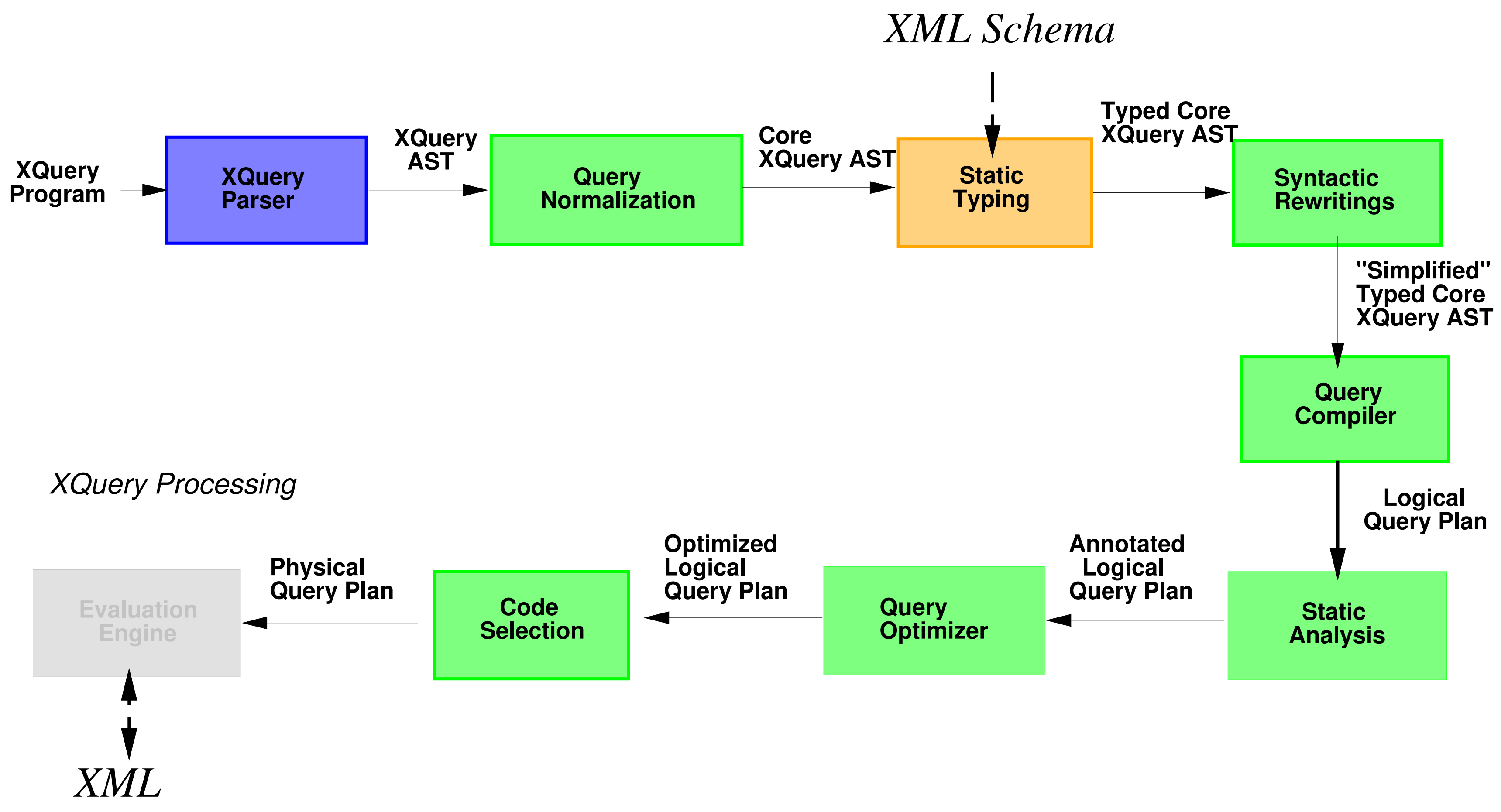
- Optimized algebraic plan:

```
MapToItem
 { Input -> Input#b }
 (MapFromItem
    {$v -> Select
             {op:integer-ge(fn:data(Step[@year](Input#b)), 2000)}
             ([b : $v])}
    (TreeJoin[child::book]($cat)))
```

This plan is pretty straightforward – multiple steps over a sequence of catalogs is converted into a TreeJoin operator, and the Select operator is pushed into the MapFromItem.

A lot more interesting when joins/group-bys are involved, but also pretty tedious to read.

XQuery Processing Step 8 : Code Selection

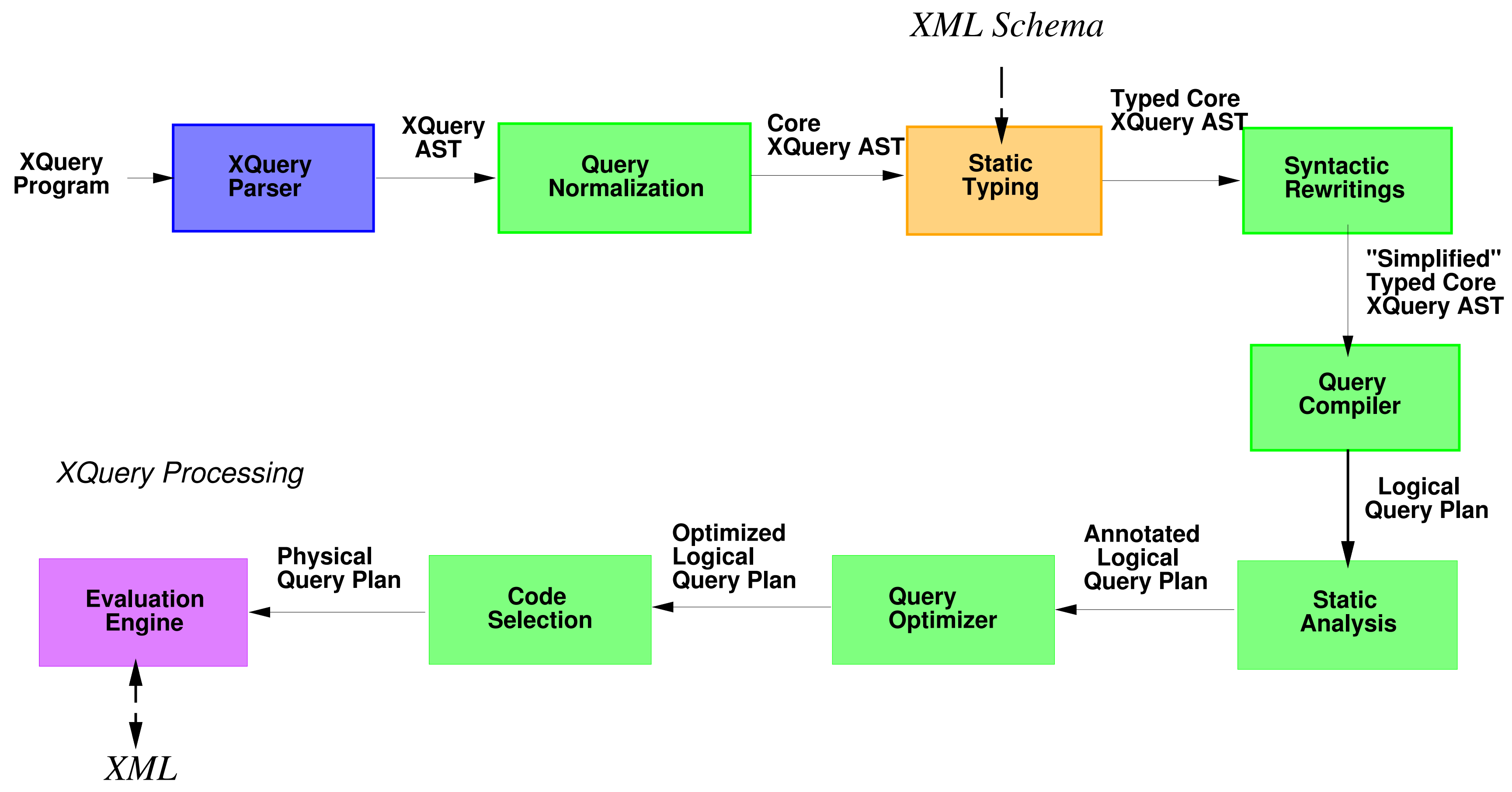


- **Step 8:** Code Selection
  - Algebraic operation mapped to physical implementation(s)

The last phase before evaluation is code selection, which takes a particular logical operator, such as a Join, and selects a particular implementation for that operator, such as SortMerge, HashJoin, etc. The result is a physical query plan in which every operator and function has been realized by a concrete implementation.

Newest part of compiler; no cost-based optimization yet. This is where we are working right now, so I don't have much to say.

XQuery Processing Step 9: Evaluation

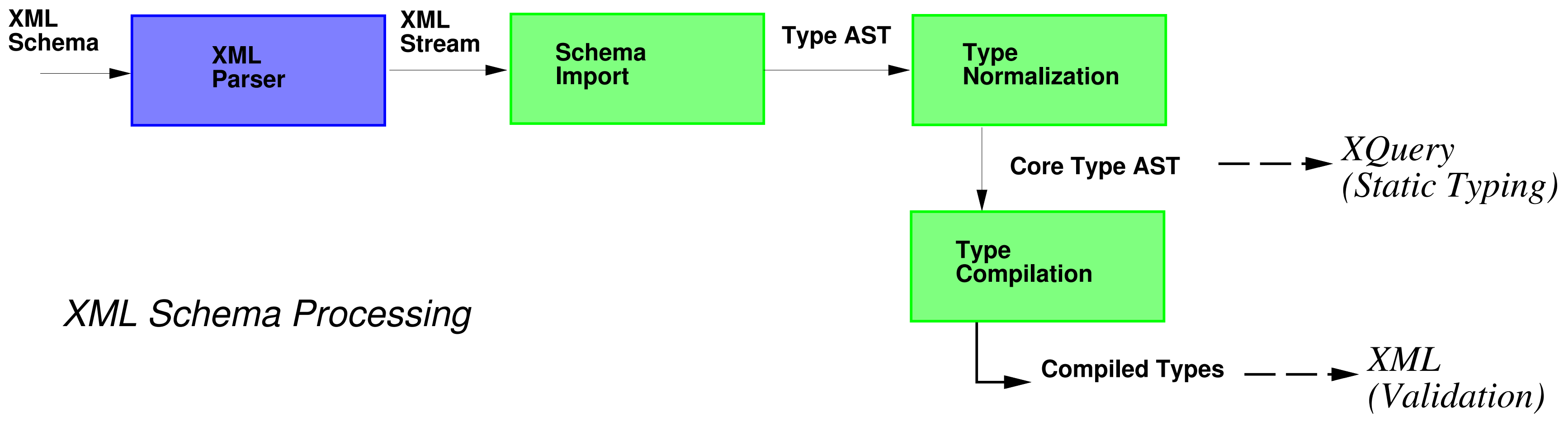


- **Output:** Instance of Galax's abstract XML data model
- Data model instance accessible via API or serialization

Evaluation takes a physical evaluation plan, and inputs, which are either typed XML streams or instances of the XQuery data model, and produces an instance of the XQuery data model.

The result can be accessed (and navigated) using the Galax data-model API from a C, Java or O'Caml program, or the result can be serialized into an XML document.

XML Schema Processing Architecture



- Analogous to XQuery processing model
- **References:** XQuery 1.0 Formal Semantics
  “The Essense of XML”, POPL 2003, Siméon & Wadler

We're almost done with the tour of Galax's architecture.

The third processing model is for XML Schemata, and is analogous to the XQuery processing model, but much simpler.

XML Schema has an XML syntax, so just as with any other XML document, it first is parsed and yields an XML token stream.

The schema-import phase takes the XML stream of the schema and produces the abstract-syntax tree of the XQuery types that corresponds to the Schema. You can think of the XQuery type language as a human-readable form of XML Schema.

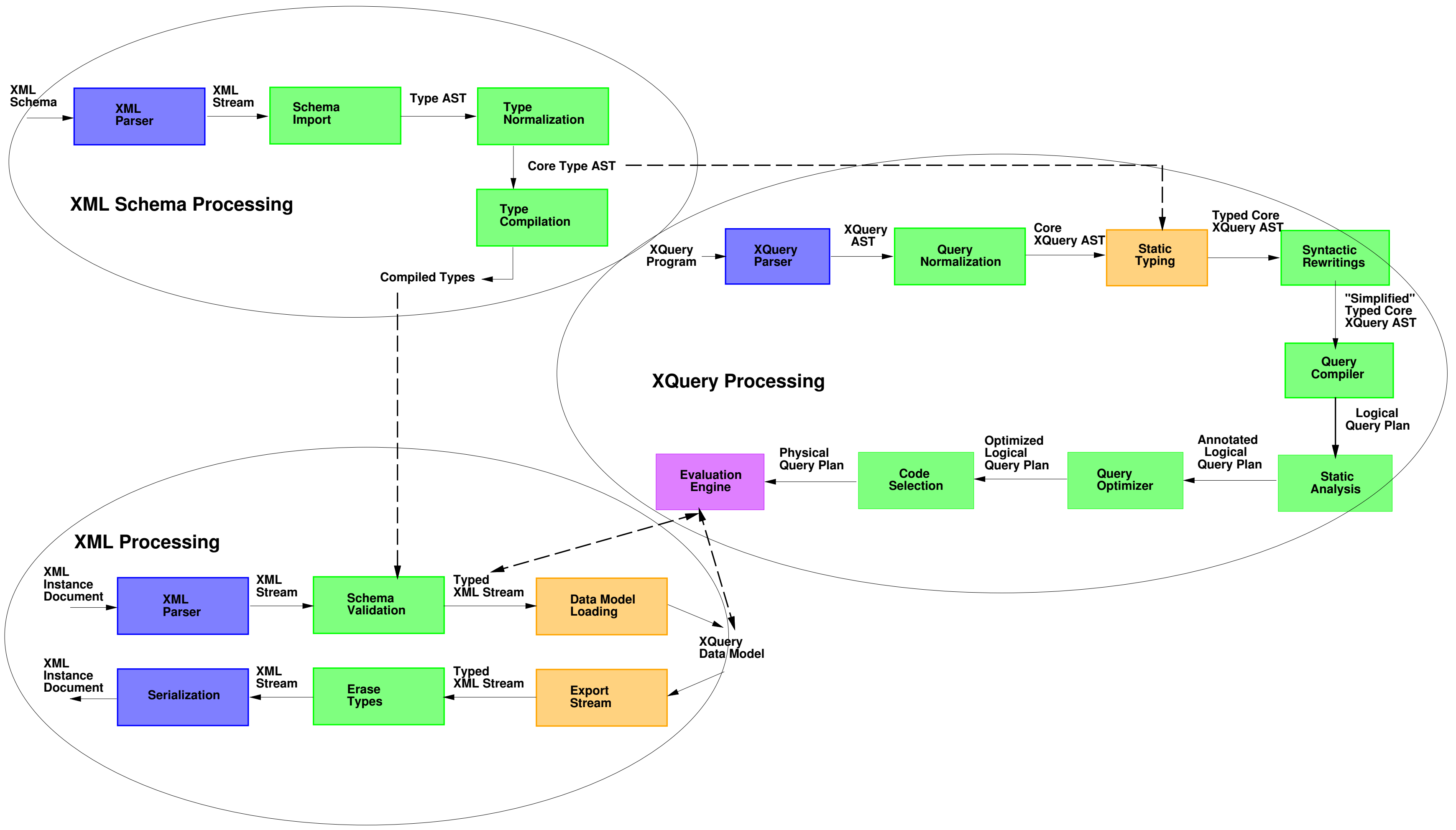
Like expressions in the XQuery language, type expressions in the XQuery type language have an implicit semantics. Type normalization takes an XQuery type and makes the implicit semantics in types explicit. For example, a generic type reference in an XQuery type is normalized into a reference to a simple or a complex type in a Core type.

The smaller, more orthogonal Core type language simplifies static typing because an expression is annotated with a unique type.

This processing model implements the formal definition of XML Schema defined in the XQuery Formal Semantics, which is based on the semantics first described in this 2003 POPL paper by Phil and Jerome.

As an aside, this paper is probably one of the most technically significant contributions to the XQuery working group.

Putting it all together



- Most code devoted to *representation transformation*
  - Of 66,000 lines O'Caml, 7600 lines (12%) for evaluation
  - O'Caml's polymorphic algebraic types support disciplined program transformation

When we put all three processing models together, you have the complete picture of Galax's architecture.

As you can probably guess from this picture, most of Galax's architecture is devoted to transforming representations of documents, schemas, and queries. In fact, only 12% of Galax's code (possibly a little more if we include the data model implementations) is devoted to query evaluation.

It is no accident that we chose O'Caml as our implementation language. O'Caml, like all other ML languages, is a meta-language, that is, a language for expressing and manipulating other languages. In particular, O'Caml's polymorphic algebraic types are used virtually everywhere in Galax and guarantee that the interfaces between phases are always well typed.

A common mistake when building a large compiler is to take short cuts and coalesce phases that are logically separate.

For example, coalesing parsing and normalization is easy to do.

Sometimes this is done because there is a perceived cost in having so many representations of one program. But the effect is that the resulting architecture is less extensibile.

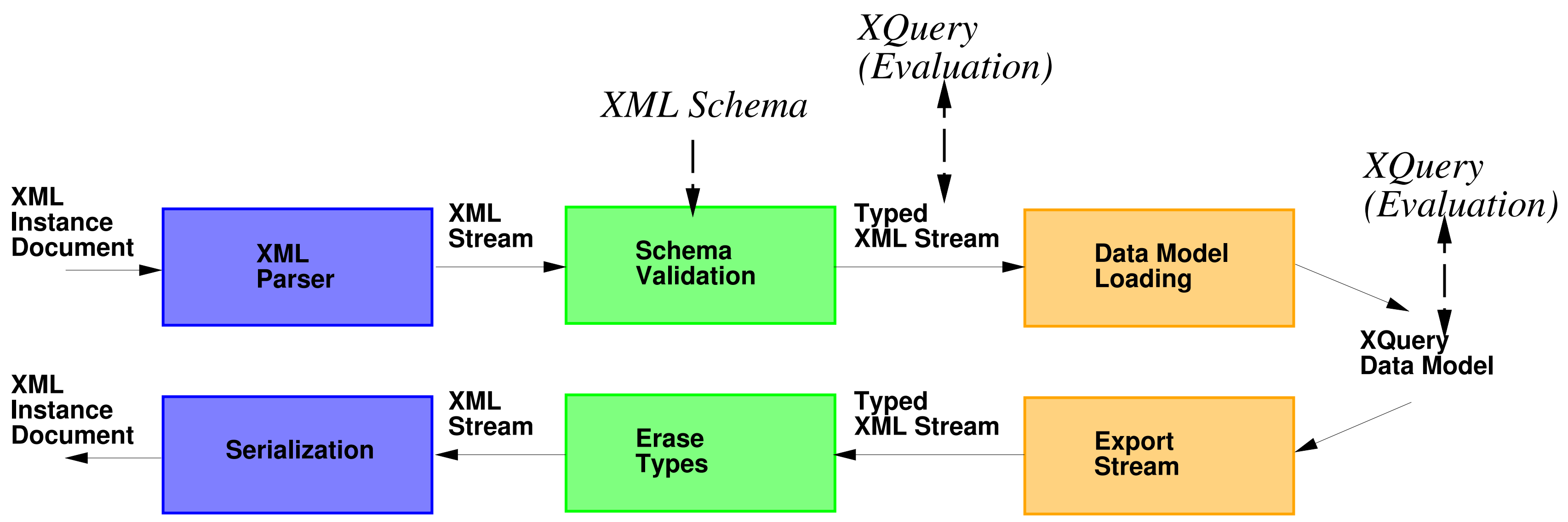
Since we went to so much trouble to build a very modular architecture, we want others to be able to contribute to its evolution and to use it for their own experimentation. Essentially partition the architecture at any point, externalize the representation at that point, and re-consume it at that point. This gives us an enormous degree of flexibility for ourselves and others to use part of the system.

66,000 includes AST .mli interfaces and all lexer .mll and source .ml files. Excludes all other interfaces. Total line count is 179012.

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: `v1, ..., vn`
- SAX events: `e1, ..., en`
  - `startElem(QName)`
  - `endElem`
  - `chars("This is text")`
- **Typed** SAX events: `te1, ..., ten`
  - `startElem(QName,`**TypeAnnotation**`,`**Value**`)`
  - `endElem`
  - `chars("This is text")`
  - **atomic(xs:integer,11)**
- SAX streams: `stream1, ..., streamn`
- **Typed** SAX streams: `tstream1, ..., tstreamn`

Streaming operations: I/O

- Input / Output
  - `Parse : channel` $\longrightarrow$ `stream`
  - `Serialize : stream, channel` $\longrightarrow$ `()`
- Example

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

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

Streaming operations: Validation

- Typing
  - `Validate : stream , type` $\longrightarrow$ `tstream`
  - `Well-formed : stream` $\longrightarrow$ `tstream`
  - `Erase : tstream` $\longrightarrow$ `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
```

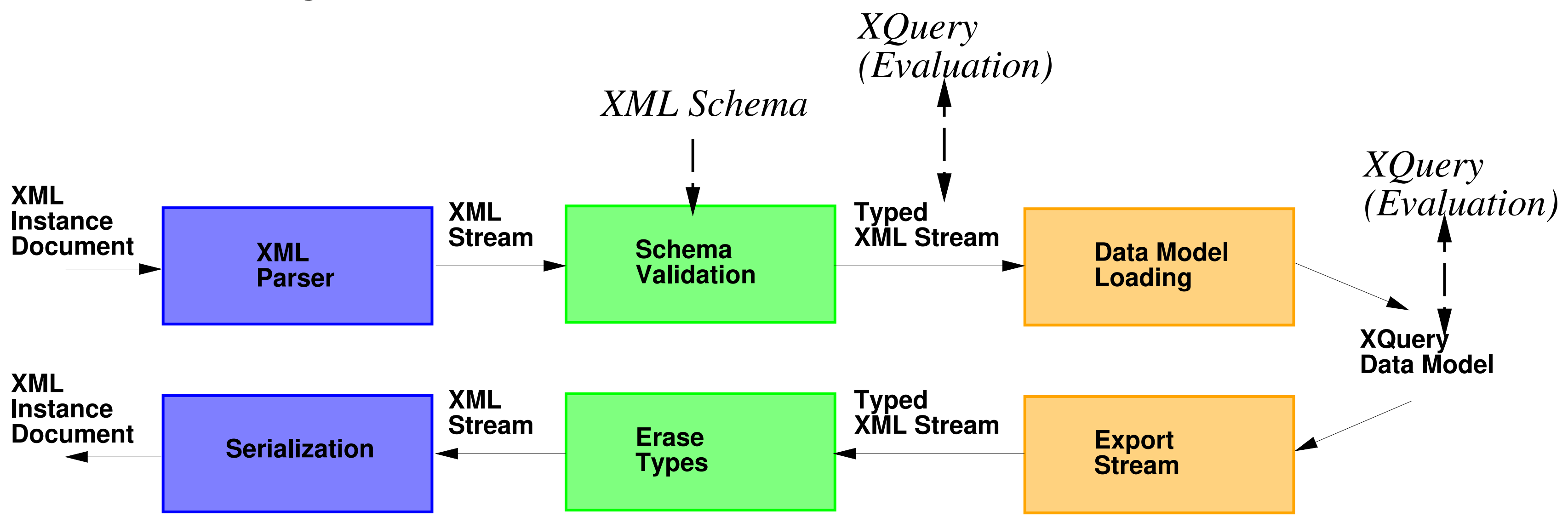
Data Model Materialization

- (De)Materialization
  - Load : tstream $\longrightarrow$ dm_value
  - Export : dm_value $\longrightarrow$ 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)

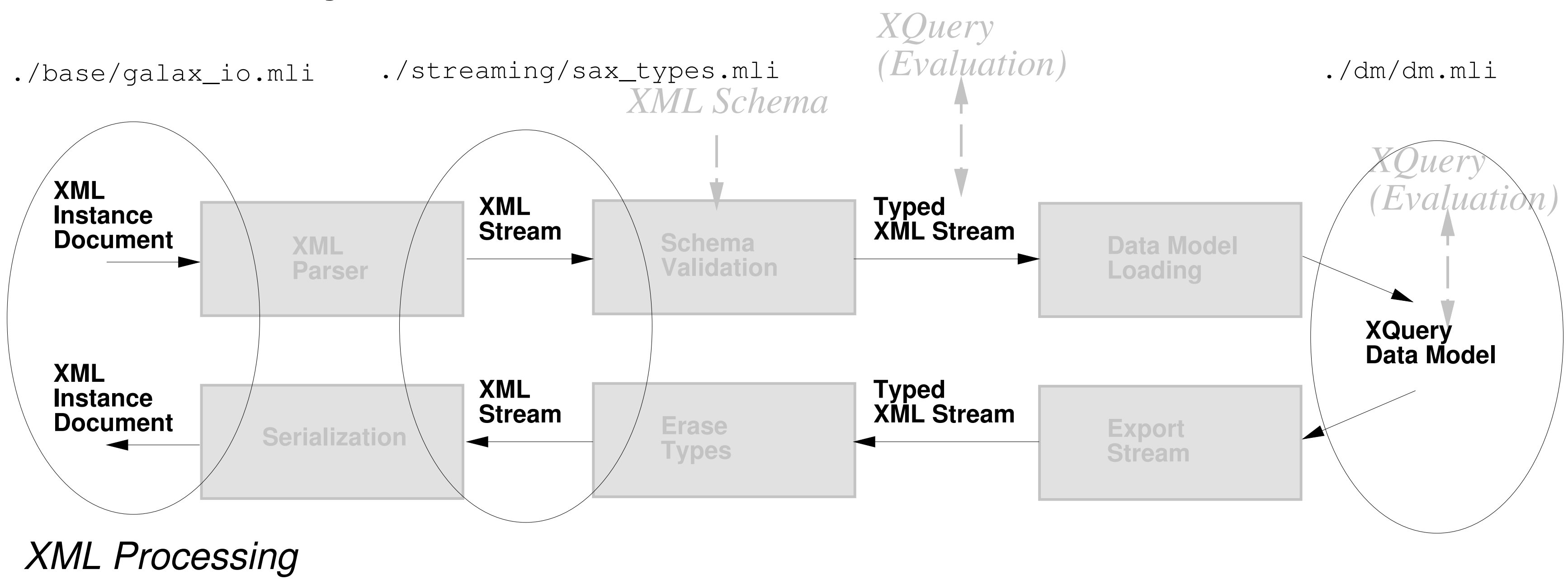
- Abstract Syntax Trees



XML Processing

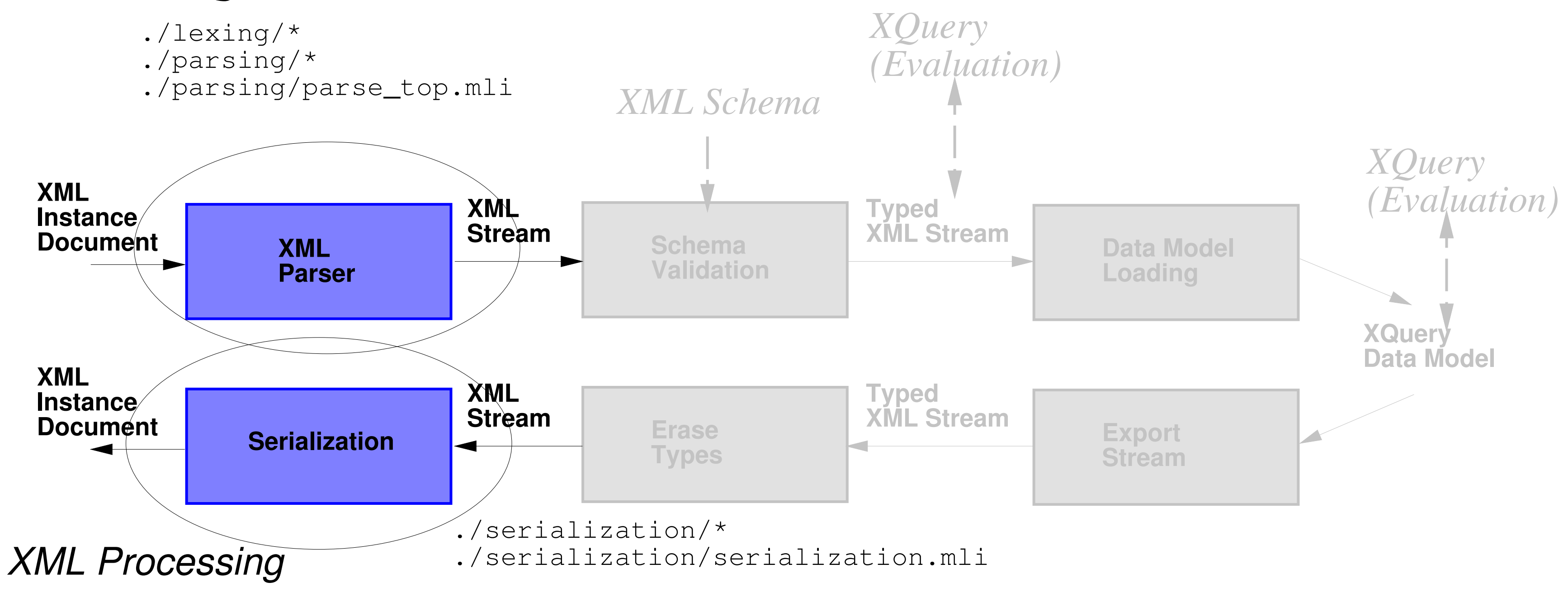
XML Processing in Galax (1)

- Abstract Syntax Trees



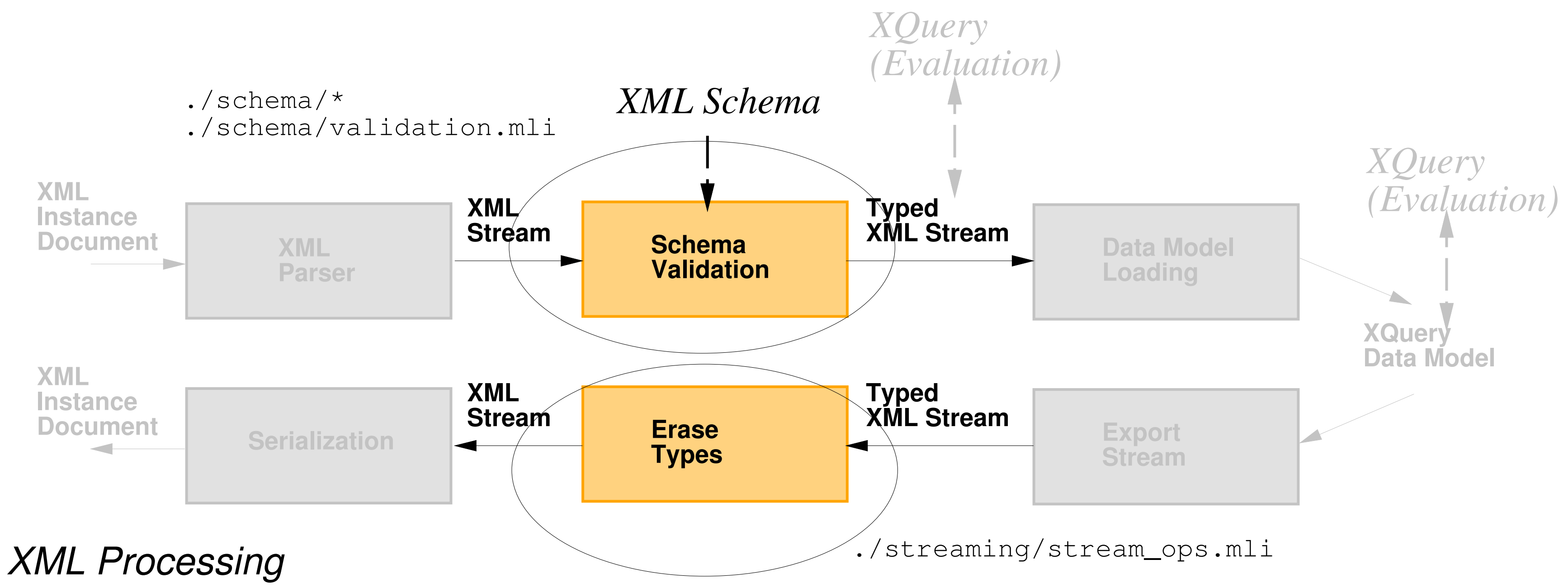
XML Processing in Galax (2)

- Parsing and serialization



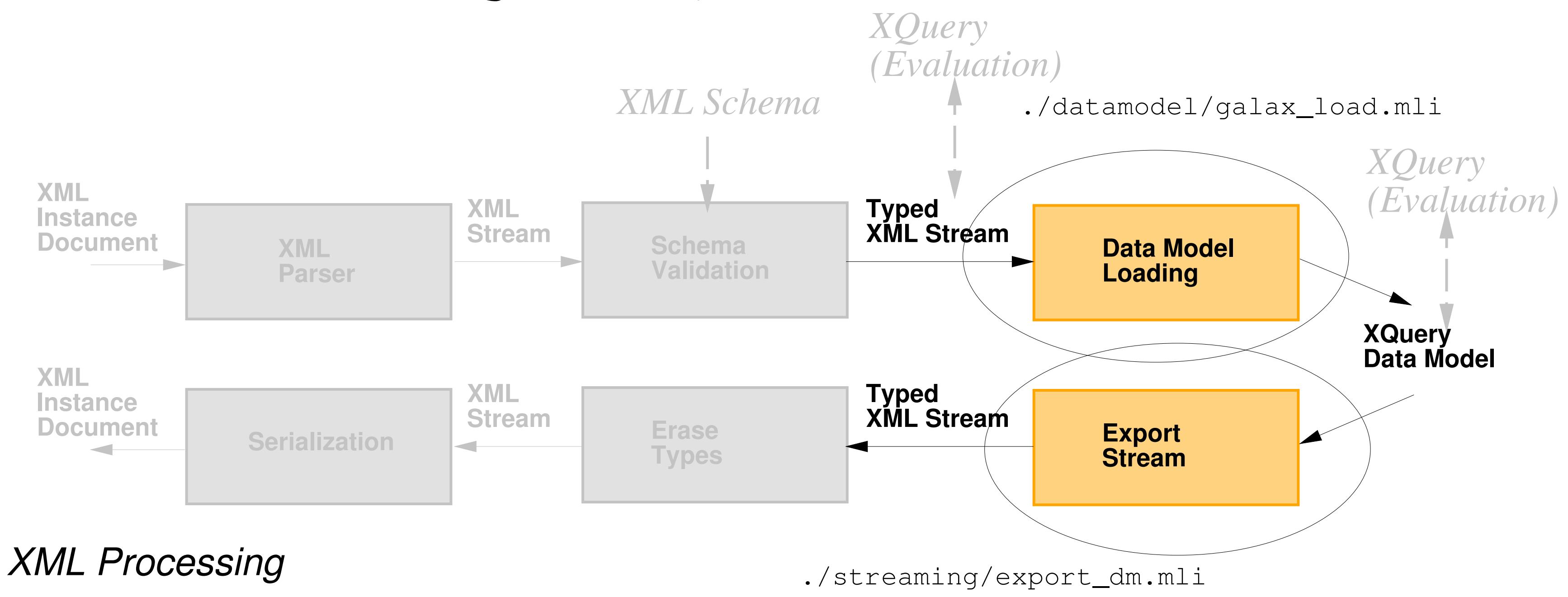
XML Processing in Galax (3)

- Stream validation and erasure



XML Processing in Galax (4)

- Document loading and export



Part V

XQuery Processing

XML Query Processing References

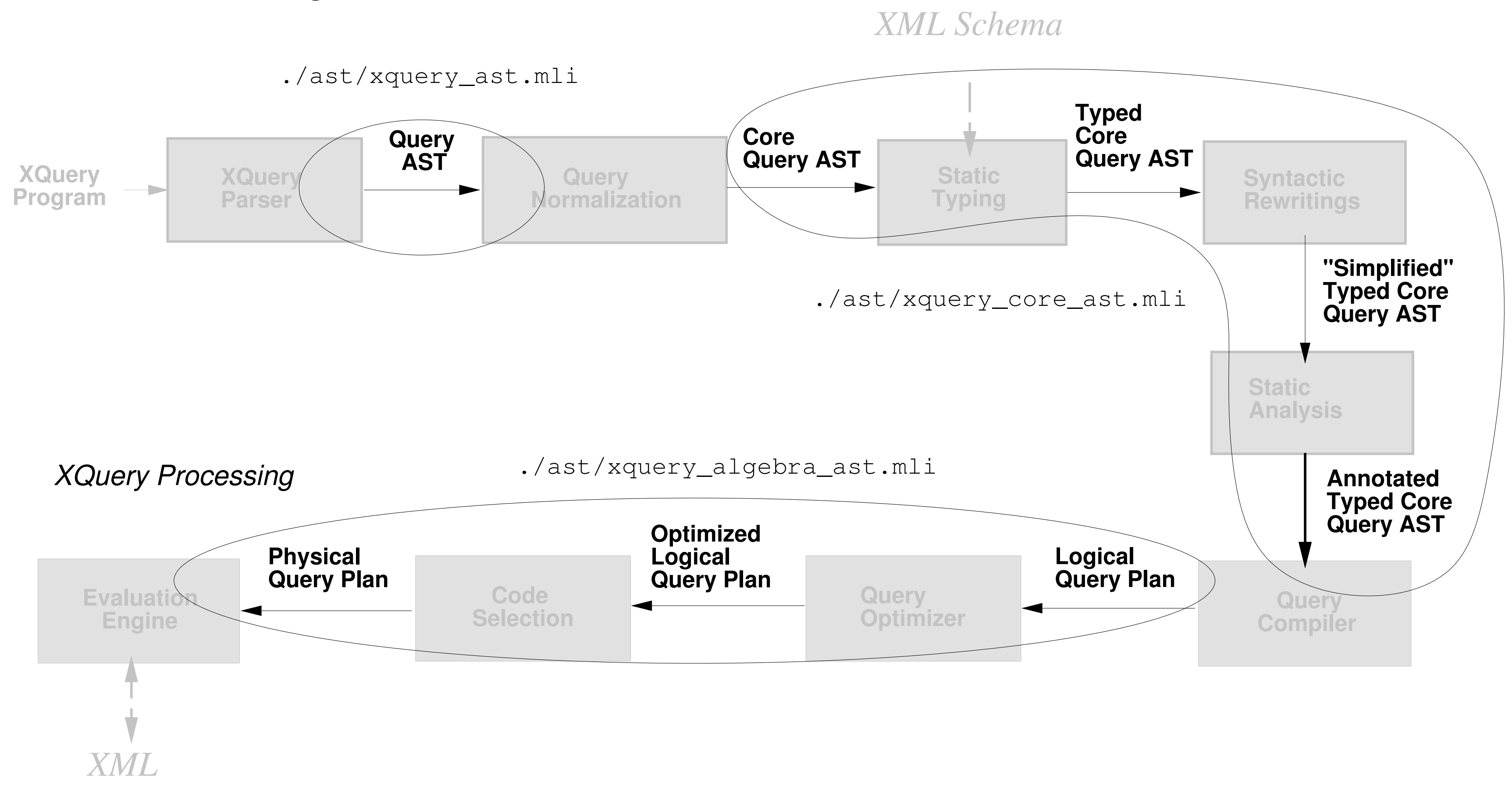
- Huge wealth of references

- 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)

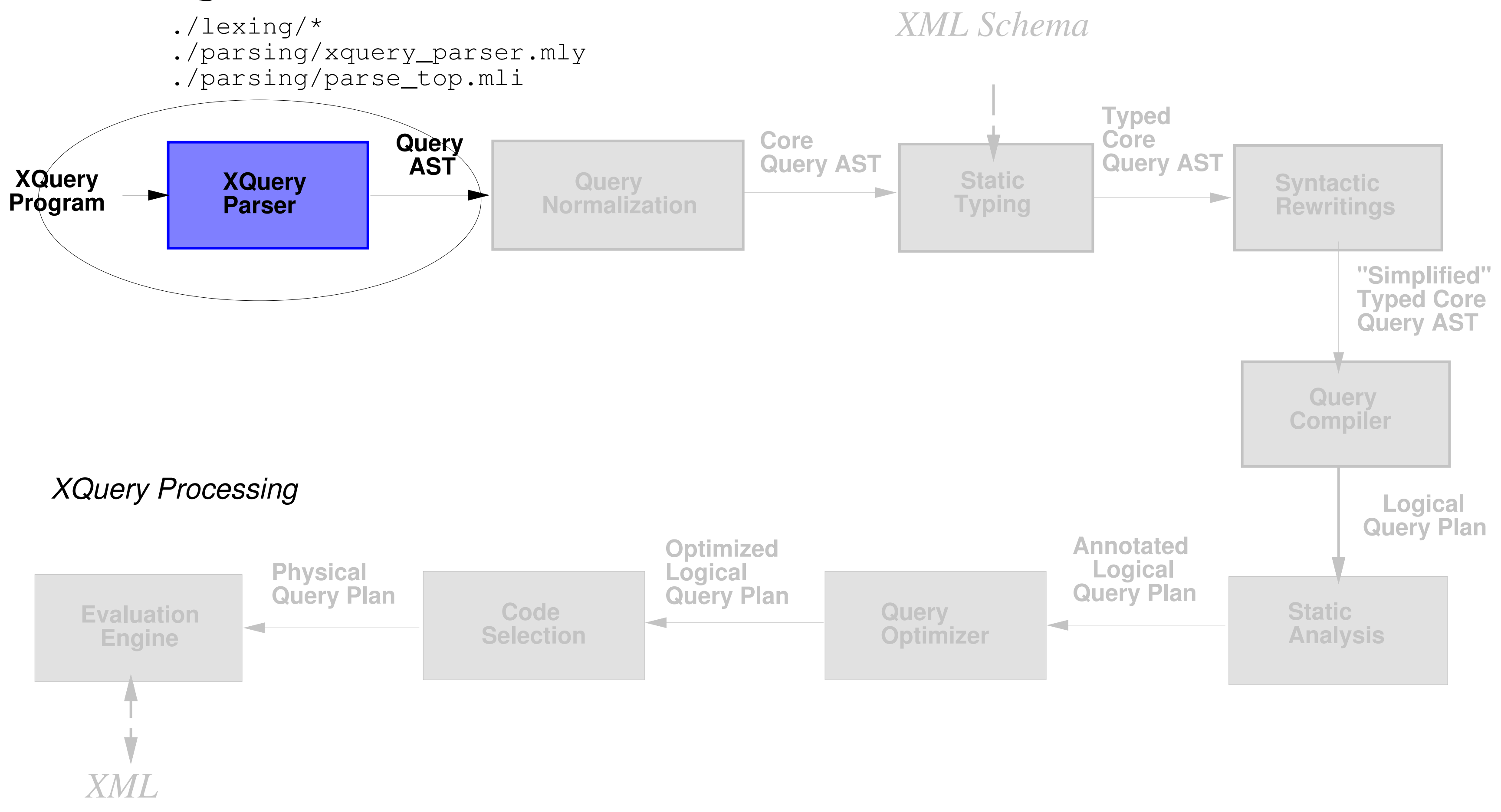
- Abstract Syntax Trees



XQuery Processing in Galax (2)

- Parsing

```
./lexing/*
./parsing/xquery_parser.mly
./parsing/parse_top.mli
```

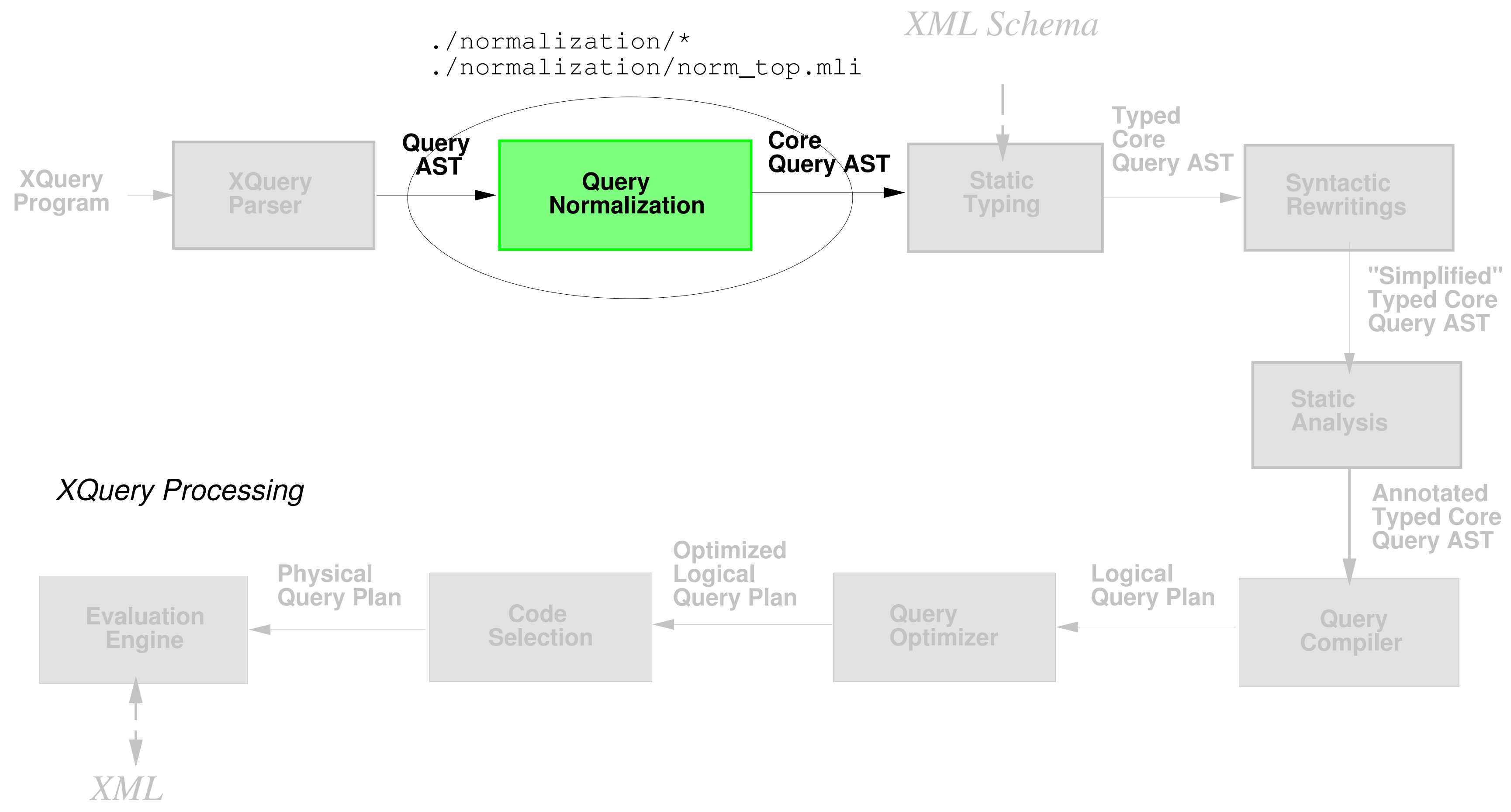


XQuery Normalization

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

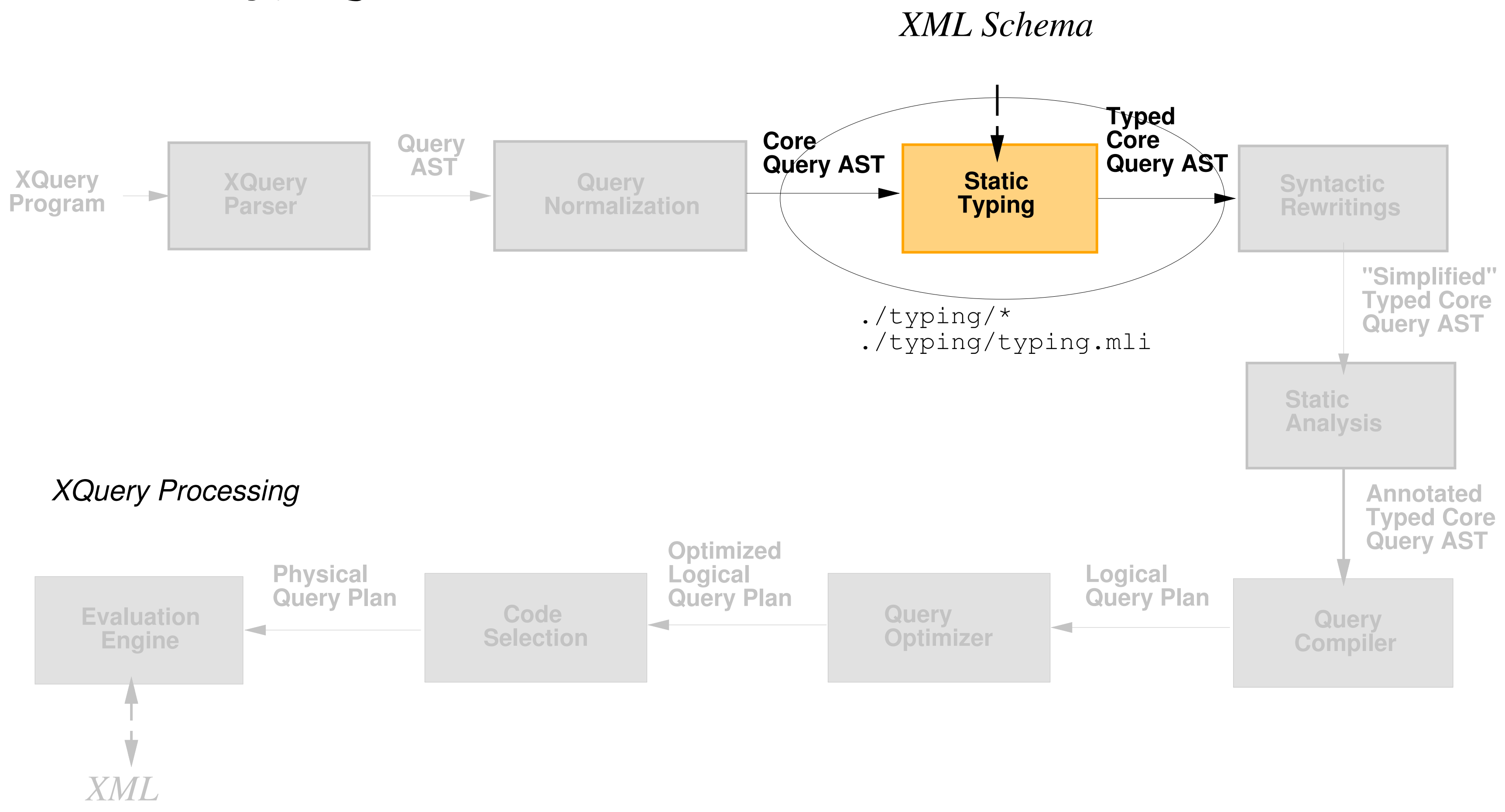
XQuery Processing in Galax (3)

- Normalization



XQuery Processing in Galax (4)

- Static Typing

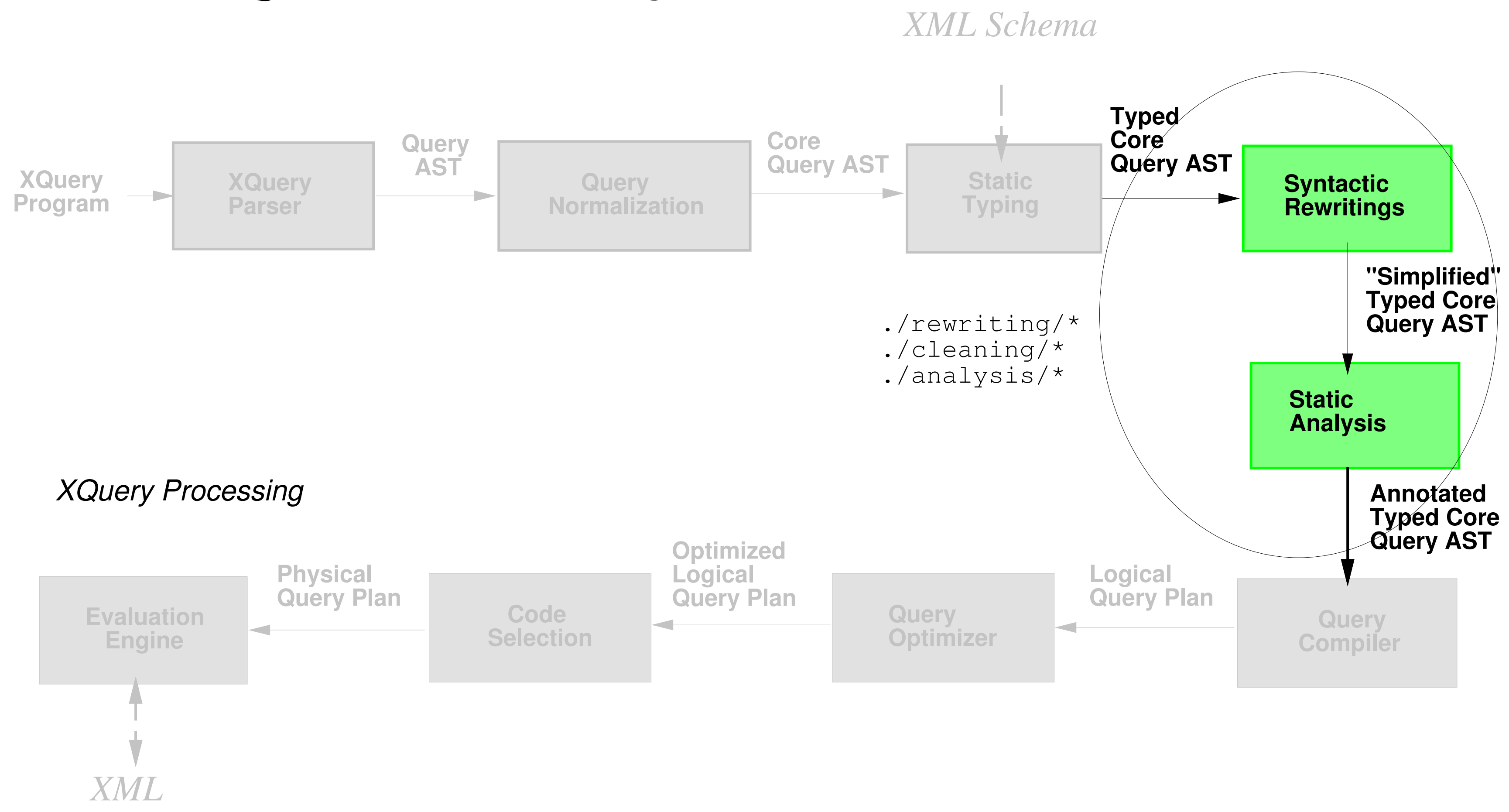


XQuery “Syntactic” Rewriting

- 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



```
./rewriting/*
./cleaning/*
./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` $\longrightarrow$ `stream`
  - `Serialize : stream, channel` $\longrightarrow$ `()`
- Typing
  - `Validate : stream , type` $\longrightarrow$ `tstream`
  - `Well-formed : stream` $\longrightarrow$ `tstream`
  - `Erase : tstream` $\longrightarrow$ `stream`
- (De)Materialization
  - `Load : tstream` $\longrightarrow$ `dm_value`
  - `Export : dm_value` $\longrightarrow$ `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:
- `SmallStream :  SmallExp` → `[h1,...,hn] stream`
- Stream composition
- `StreamCompose :  [h1,..,hk] stream , [j1,..,jl] stream`

→ `[j1,..,jl,h2..,hn] 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/perso/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

Configuration		*A*	*B*	*C*
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 bottle-neck

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 pipelin-
ing

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.

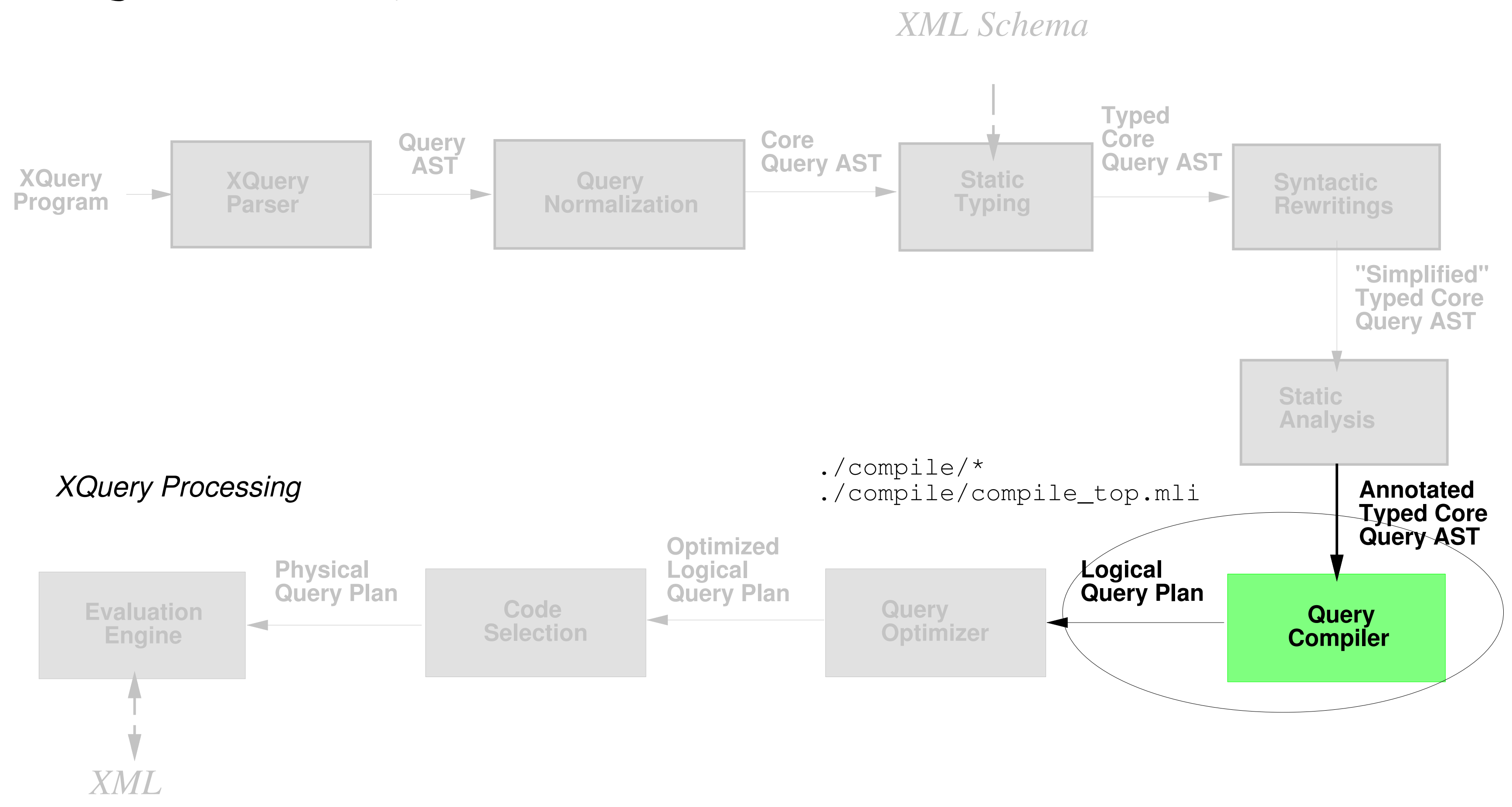
More on Nested Queries

- 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.

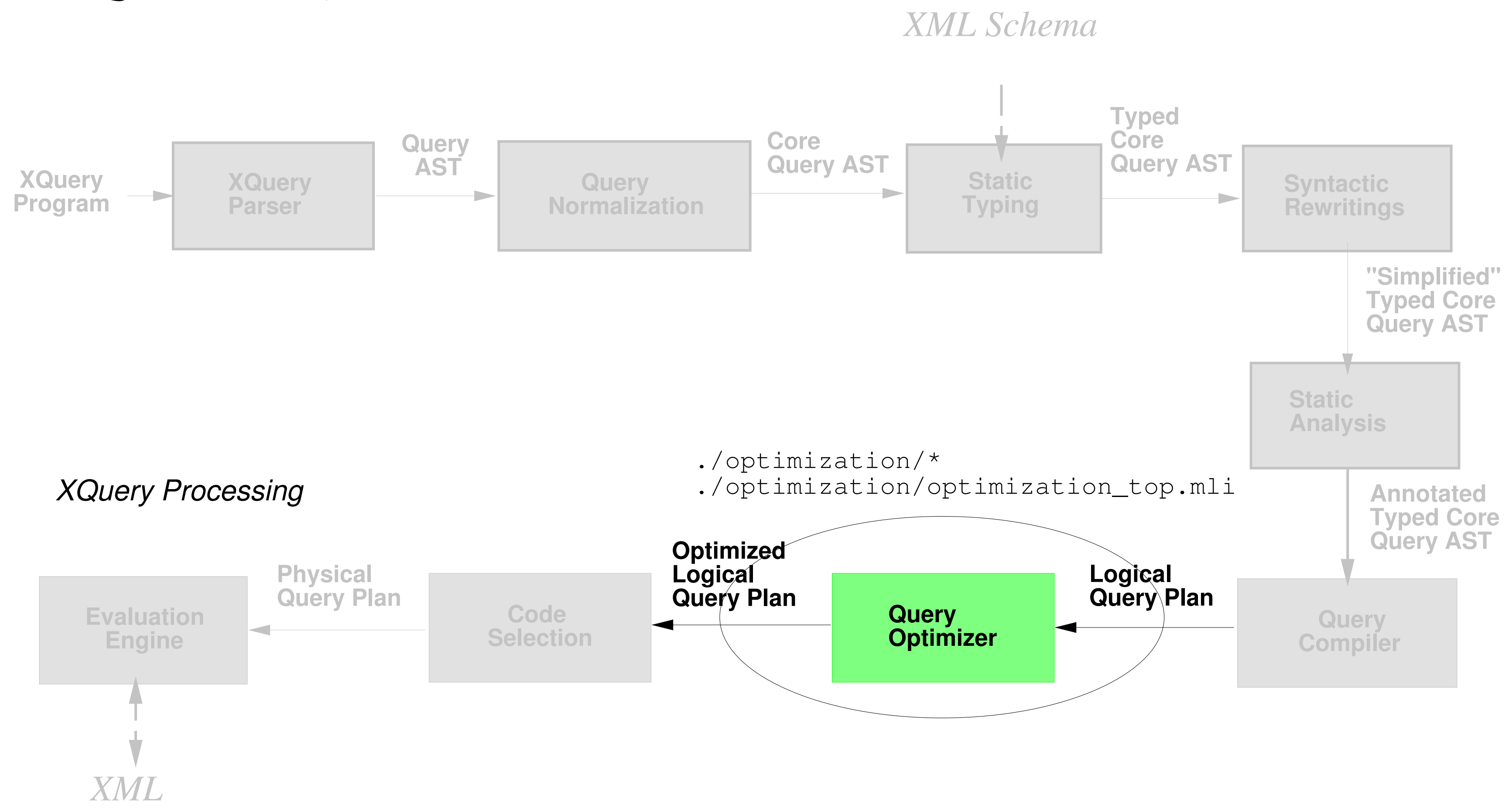
XQuery Processing in Galax (6)

- Algebraic compilation



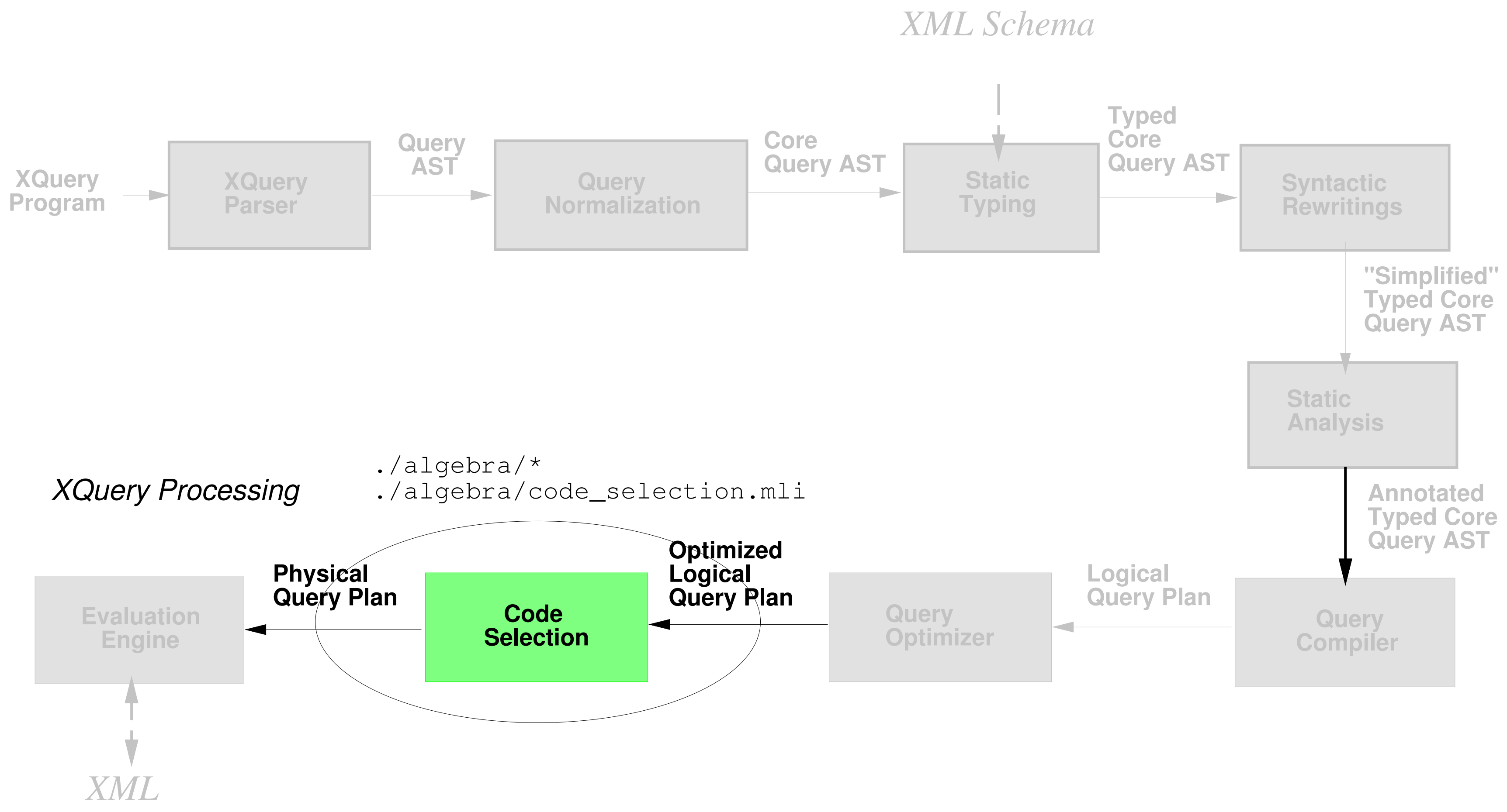
XQuery Processing in Galax (7)

- Algebraic optimization



XQuery Processing in Galax (8)

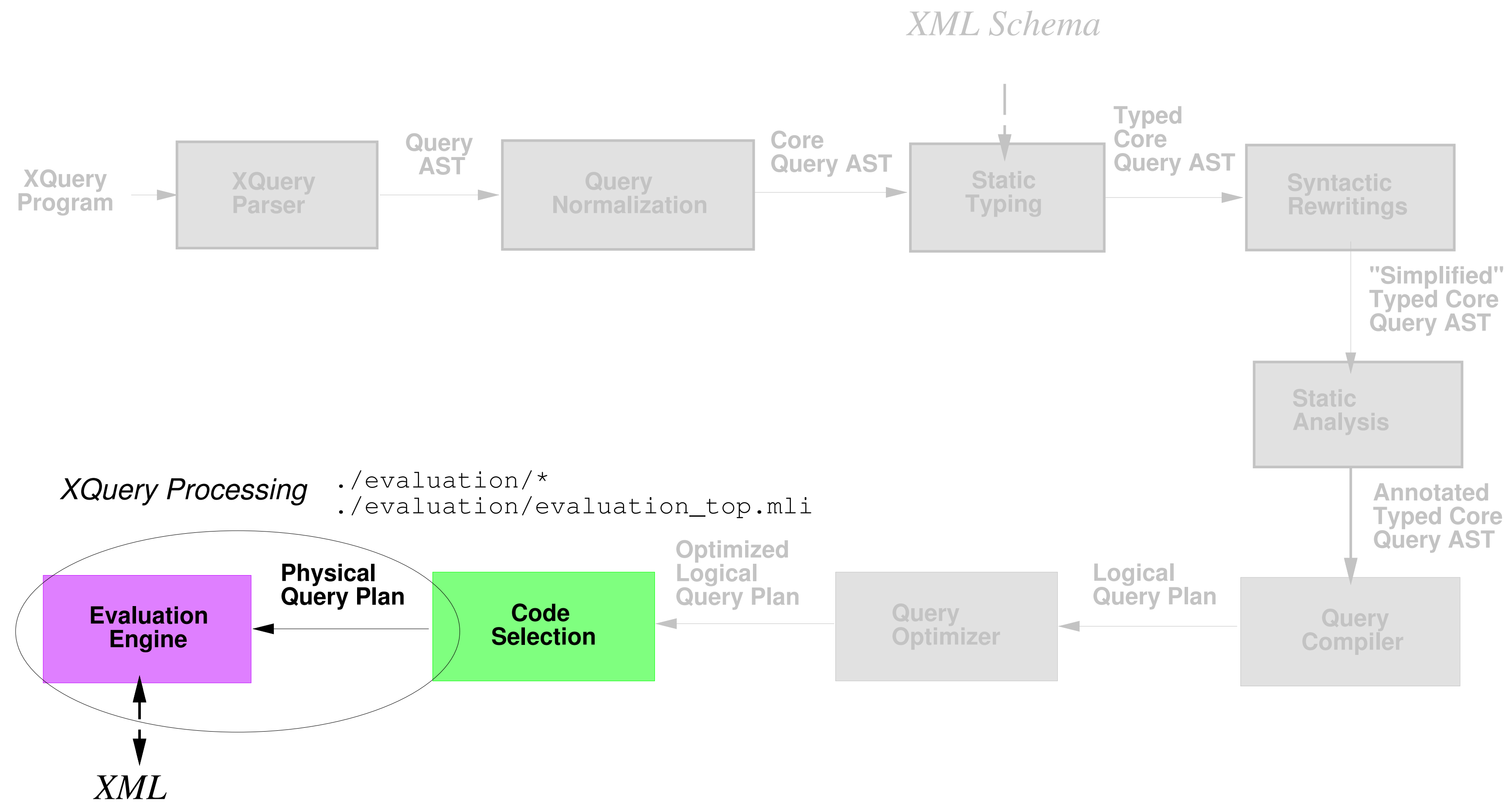
- Code selection



```
./algebra/*
./algebra/code_selection.mli
```

XQuery Processing in Galax (9)

- Evaluation



```
./evaluation/*
./evaluation/evaluation_top.mli
```

Part VI

Conclusions

Completeness—New Research & Colleagues

- Implementation language for other DSLs

- GalaTex: XQuery Full-Text Language
  - First implementation of full-text extension language
  - `http://www.galaxquery.org/galatex`

  Sihem Amer-Yahia, Emiran Curmola, Phil Brown

- Distributed XQuery
  - Trust management in peer-to-peer systems

  Grid resource management
  - Queries migrate to data

  Trevor Jim

GalaTex – implement W3C's full-text extensions to XQuery ; prototype language & features. Very exciting project. Intersection of information-retrieval and traditional databases.

Web services — explore XQuery as Web services programming language

Both these projects required the whole language – modules in particular.

Wouldn't have happened without completeness.

Extensibility—More Research & Colleagues

- XQuery!
  - Extension to language syntax, normalization, semantics

  Christopher Ré, Gargi Sur, Joachim Hammer

- Querying Ad Hoc Data Sources
  - Query-able XML views of semi-structured, ad hoc data
  - *"PADX: Query Large-scale Ad Hoc Data with XQuery", PLAN-X'06*

    `http://www.padsproj.org`

  Kathleen Fisher, Joel Gottlieb, Bob Gruber, Yitzhak Mandelbaum, David Walker

Performance—You get the idea

- Complete XQuery algebra, logical optimizations
  Unifying framework of tuple & tree operators

- Physical algorithms
  - Comprehensive comparison of algorithms for path evaluation
  Stair-case join, twig join, streaming, *et al*

  - "Streaming" physical plans
  Identify necessary conditions & integrate existing techniques
  "Projecting XML Documents", VLDB 2003, Amélie Marian

- Christopher Ré, Philippe Michiels, Michael Stark

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

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!

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)...

Lessons Learned: Research

- Where is research in all this?
  - 85% is **not** research
  - 15% is research
- Some interesting research based on Galax
  - Compilation, optimization: ICDE'06, VLDB'03
  - Static typing: POPL'03, ICDT'01
  - Indexing, storage: XIME-P'04
  - Extensibility: PLAN-X'06/04, WWW'04, SIGMOD'04
- But the 15% 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.

Where we are ... Where we want to be

- Galax Version 0.6.0 (February 2006)
  - Conformant implementation of XQuery 1.0
    8,000+ conformance test queries
  - Algebraic query plans, logical optimizations,
    & join algorithms, static typing
  - Source code & binaries for Linux, MacOS, Solaris, Windows(MinGW)
- Gold standard of open-source XQuery implementations
  - Implementation of choice for experimentation & research
- Visit us at http://www.galaxquery.org

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
Michael Stark, Technische Universität Darmstadt
Gargi Sur, University of Florida
Avinash Vyas, Lucent Bell Laboratories
Philip Wadler, University of Edinburgh