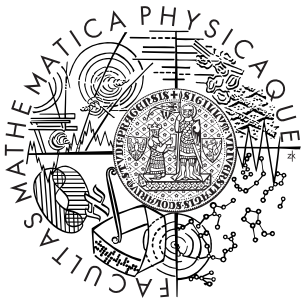
Charles University in Prague
Faculty of Mathematics and Physics

DOCTORAL THESIS



RNDr. David Bednárek

Bulk Evaluation of User-Defined Functions in XQuery

Department of Software Engineering

Supervisor: Prof. RNDr. Jaroslav Král, DrSc.

2009

I would like to thank all those who supported me in my doctoral studies and work on my thesis. In the first place I appreciate the help and advices received from my supervisor Jaroslav Král and I am grateful for corrections, comments, and ideas I have received from him and other, often anonymous reviewers. Secondly, I would like to thank the leaders of our department, Jaroslav Pokorný, František Plášil, and Peter Vojtáš, for allowing me enough time and space to complete this thesis. Last but not least, I thank Jana, Ondřej, and Eliška for their support and tolerance.

My work was partially supported by the National Programme of Research, Information Society Project number 1ET100300419.

Dedicated in memory of Jiří Demner and Jan Pavelka who influenced me much in my professional life.

Title: Bulk Evaluation of User-Defined Functions in XQuery
Author: RNDr. David Bednárek
Department: Department of Software Engineering
Faculty of Mathematics and Physics
Charles University in Prague
Supervisor: Prof. RNDr. Jaroslav Král, DrSc.
Author's e-mail address: bednarek@ksi.mff.cuni.cz
Supervisor's e-mail address: kral@ksi.mff.cuni.cz

Abstract:

XPath queries are usually translated into an algebra that combines traditional relational operators and XML-specific ones. In particular, FLWOR loops are represented using nest, unnest, join, and similar operators and their original nested-loop nature disappears, creating an opportunity for bulk evaluation and join reordering. In XQuery, two additional issues shall be handled – tree construction and the presence of user-defined functions. The recursive nature of functions pushes the problem outside of the range of relational algebra. This thesis presents a novel evaluation framework based on an expanding network of relational operators, called R-program. In this environment, functions are evaluated in bulk instead of evaluating each call separately. Besides obvious advantages of bulk evaluation, R-programs also allow rearrangement of data flow across function boundaries. A set of program transformations employing these capabilities is described; together with rule-based static interprocedural analysis algorithms used to determine the applicability of the transformations.

Keywords: XML data management, XQuery, recursion, relational algebra.

Název: Hromadné vyhodnocování uživatelských funkcí v jazyce XQuery
Author: RNDr. David Bednárek
Pracoviště: Katedra softwarového inženýrství
Matematicko-fyzikální fakulta
Univerzita Karlova v Praze
Školitel: Prof. RNDr. Jaroslav Král, DrSc.
E-mail autora: bednarek@ksi.mff.cuni.cz
E-mail školitele: kral@ksi.mff.cuni.cz

Abstrakt:

Dotazy v jazyce XPath jsou obvykle překládány do algebraického systému kombinujícího tradiční relační operátory s operátory specifickými pro XML. Konstrukce FLWOR jsou pak reprezentovány operátory nest, unnest, join a dalšími, čímž se ztrácí jejich vnořený charakter a otevírá se příležitost pro hromadné vyhodnocování a restrukturalizaci spojení. V jazyce XQuery přibývají dva další problémy – konstrukce stromů a přítomnost uživatelských funkcí. Rekurzivní charakter funkcí nedovoluje reprezentaci tohoto jazyka uvnitř relační algebry. V této práci je prezentován nový přístup k vyhodnocování založený na expandující síti relačních operátorů, nazvané R-program. V tomto prostředí jsou funkce vyhodnocovány hromadně namísto separátního vyhodnocování každého volání zvlášť. Vedle zřejmých výhod hromadného vyhodnocování R-programy dovolují restrukturalizaci toku dat přes rozhraní funkcí. V práci je uvedena sada transformací využívajících tyto schopnosti a algoritmy statické interprocedurální analýzy určující aplikovatelnost uvedených transformací.

Klíčová slova: XML, XQuery, rekurze, relační algebra.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

The recent development in the area of query languages for XML shows that the XQuery language will likely be used as one of the main application development languages in the XML world [19]. This shift from a query language towards a universal programming language will be accompanied by increased complexity of XQuery programs. In particular, intensive use of user-defined functions may be expected.

Current XQuery processing method can handle the interior of an XQuery function well, including the technique usually called *query unnesting* – a sub-query nested in a FLWOR loop is hoisted to the level of the surrounding query and the iterative character of the loop is replaced by a join operator. Subsequently, join reordering or XML-specific holistic join techniques may be applied. Experiments show that the use of such methods is the only known key to higher throughput.

Unfortunately, none of the above-mentioned methods can be extended to the inter-procedural level. Known attempts are based on procedure integration (we prefer this term known from compiler construction although it is, literally, function integration), inspired by the handling of views in SQL. However, integration techniques induce expansion of the program size and cannot be applied to recursive functions – note that recursion support is important due to the recursive nature of XML.

This work presents a method of XQuery evaluation that allows some degree of optimization across the boundaries of functions while avoiding more than linear code size expansion. The system is based on relational algebra and designed primarily for physical layouts that employ Dewey numbering [57]; however, it is also applicable to other physical storage systems.

Designing a query evaluation method is a tremendous amount of work and creating a completely new method from scratch is naturally a nonsense; on the other hand, the accent put on inter-procedural optimization may re-

quire redesigning of the architecture of the whole processing chain. In other words, inter-procedural optimization can hardly be added as an extension to an existing query processor. Instead, overall architecture must be redesigned; then, existing methods may be implemented within the new frame. Naturally, some important methods may not fit into the new architecture – for instance, traditional join-reordering methods can not be applied across functions without their integration (which is discouraged due to the cost of code expansion). In such cases, replacements for these methods shall be designed.

Obviously, a single thesis may not incorporate a complete design of a query processor in the necessary degree of detail. Therefore, the scope of this thesis is reduced to the following items:

- A mathematical model, based on relational algebra, capable to represent an optimized query, including user-defined functions.
- For important XQuery constructs, their (unoptimized) representation in the mathematical model.
- Alternative representations for important XQuery value types and concepts – Boolean values, output tree generation, unordered context etc.
- Rule-based transformations as a replacement for the most important query rewrite methods like predicate pushing and join-reordering.
- Effective static analysis algorithms for the application of the alternative representations and transformations.

All these items belong to the compilation phase of a query; the run-time phase is described only at conceptual level.

1.1 Contribution

In this thesis, we present *R-programs* as an intermediate language combining relational algebra operations with functions. While there is one-to-one correspondence between R-program functions and XQuery functions, a single call to an R-program function corresponds to multiple calls to the corresponding XQuery function. Inspired by terms like *bulk load* and *bulk operation*, we call this arrangement *bulk evaluation*.

Beyond the ability to perform more than one function call at once, R-programs also allow optimizations based on the reversal of the data flow across a function boundary. Based on this reversal, effects similar to predicate

pushing and join reordering may be achieved, even though the system does not offer general support for moving operations through function boundaries.

An R-program is conceptually a network of computing nodes, connected by pipes carrying relations. Each node performs a relational operation. In this system, functions conceptually run in parallel with their callers, exchanging their data in both directions. Since pipes are easily implemented in distributed environment, this arrangement is particularly advantageous in parallel settings.

To show the generality of R-program models, we present the *canonical* transcription of XQuery programs to R-programs. Nevertheless, the canonical mode is only a last-resort mode – in many cases, an alternate *mode* may be used to reduce the complexity of the representation. A selection of alternate modes is described in this work; we also show that these modes may be automatically reduced to *submodes*, based on the removal of relation attributes.

We also show additional transformations that may serve as an interprocedural replacement for predicate pushing or join reordering. For structural joins, the transformed network of join operators copies the behavior of a path-join algorithm in a distributed manner.

Finally, this work presents static analysis algorithms that determine which transformation is applicable or required for each parameter, variable, or subexpression in an XQuery program. While such analysis is straightforward within an expression, its extension to programs with recursive functions is an important step towards effective application of the above-mentioned transformations.

This work covers the most important aspects required in the design of the compile-time part of an XQuery engine. Although some of the presented methods are only equivalents of well-known techniques, their demonstration is important to show the viability of the approach based on the R-programs.

The most important advantage of the R-programs is their ability to support user-defined XQuery functions in a way that does not obstruct interprocedural optimization. In this sense, this is a novel technique of XQuery evaluation that may be superior to the currently used methods, especially for larger XQuery applications engineered with intensive decomposition to small functions.

1.2 Outline

The next chapter, *Related work*, presents a review of the most important techniques used in XPath and XQuery evaluation and in the areas that affect

the evaluation, from physical storage and indexing through XML-specific joins to algebraic representation of queries. A motivation example is shown in the Chapter 3 together with the principles of the evaluation method.

In the Chapter 4 *Preliminaries*, our version of relational algebra notation is summarized and the notion of *hierarchical identifiers* is introduced. The definition and the semantics of *R-programs* is shown in the Chapter 5.

The *forest model* of a query and the generic framework of *transcription* from forest models to R-programs is presented in the Chapter 6. The principles and the transcription rules for the canonical and Boolean modes are introduced in the Chapter 7 (details are given in the Appendix); the motivation, principles, examples, and transcription rules for the *reverted modes* are shown in the following chapter.

The Chapter 9 – *Static analysis* defines the task of mode selection, its phases and algorithms.

The pro's and con's of the method, optimization issues, open problems, future directions, and alternate approaches are presented in the *Conclusion*.

Chapter 2

Related Work

In this chapter, we will review the most important topics related to our goal. Of course, the related areas are quite wide; we will analyze only those aspects that have some impact on the problems we are solving.

2.1 XML data model

In the area of querying, the abstraction of a XML document is governed by the W3C definition of the *XQuery/XPath Data Model (XDM)* [29]. Note that XDM is slightly different from the *XML Information Set (Infoset)* [21] which is used to describe XML documents per se.

Two use cases are distinguished in the XDM specification: Construction from an Infoset and construction from a *Post Schema Validation Infoset (PSVI)*. While the interface presented by XDM is fixed, the behavior of the model is different in the two cases. In other words, the result of a query applied to validated input may differ from the result of the same query on the same input without validation.

A XML storage is usually stuck to one of the two use cases, depending on the schema-awareness of the approach. Moreover, the information stored in the database does not necessarily copy the original Infoset; naturally, a storage method closer to XDM would match the requirements of the querying engine better. However, storing a verbatim materialization of XDM would not be wise since XDM is redundant (see below). Consequently, XDM is only an abstraction that is not physically represented in real XML databases.

Under XDM, a XML document is an ordered unranked tree whose nodes are labeled with a set of properties. The most important properties are:

- *node-kind* – the kind of node, one of *attribute*, *comment*, *document*, *element*, *namespace*, *processing-instruction*, and *text*

- *node-name* – the qualified name of the node, if any
- *string-value* – a string value associated to the node
- *typed-value* – a sequence of typed atomic values associated to the node

The remaining *properties* and *accessors* required by the standard (*base-uri*, *document-uri*, *is-id*, *is-idrefs*, *namespace-bindings*, *namespace-nodes*, *nilled*, *type-name*, *unparsed-entity-public-id*, and *unparsed-entity-system-id*) are either deprecated or of minor importance; therefore, they are often ignored at research level. Similarly, the *comment*, *namespace*, and *processing-instruction* nodes are usually neglected.

There are two sources of redundancy in XDM:

- For a non-leaf node (i.e. a *document* or *element* node), the string value is by definition the concatenation of the string values of all its *text* node descendants in the document order.
- In schema-unaware environment, the typed value equals to the string value; in PSVI settings, the typed value corresponds to the string value under the rules of the type determined from the schema.

Since the concept of *atomization* (see the section 2.4) uses the typed value, storing the typed value in a schema-aware XML storage saves the conversion cost and makes indexing effective. The string value would be calculated from the typed value in such setting. Nevertheless, this approach may be applied only to the *attribute* nodes and the lowermost *element* nodes; for inner *element* nodes, storing the concatenated string value would cause unacceptable redundancy while not saving any conversion cost since no type conversion is done at complex-content nodes (see the section 2.3). Thus, if the string or typed value is queried for an inner *element* or *document* node, the value must be generated by concatenation. This observation suggests that the string and typed values should be stored differently at various levels of the tree, at least in schema-aware methods.

Besides the representation of complete XML documents, XDM instances (usually called *temporary trees* or *tree fragments*) are created by the XQuery *constructors*. For these trees, the XDM requirements are relaxed; consequently, a method designed to store complete documents may not be applicable to temporary trees. Therefore, separating the paths of input documents and temporary trees during the evaluation of a query may be necessary although the XQuery standard treats both the types of XDM instances equally.

2.2 XML data types

The XDM specification [29] together with XML Schema specification [79] define a hierarchy of types.

From the point of view of the XML Schema, the most important categorization is between *simple types* and *complex types*. A simple type may be assigned to an attribute and to the text content of an element node, while a complex type places a restriction on the structure of the sub-tree of an element node.

In querying, the point of view is different: An XPath or XQuery expression may evaluate to a sequence of *items*, each item being either a reference to a node or an *atomic* value. Thus, the world of item types in XQuery contains two independent spaces:

- *Atomic types*, derived from `xs:anyAtomicType`, attached to every atomic item
- Node types, usually referred to as *kind tests*, allowing to restrict the node-kind and/or the node-name properties of nodes

The hierarchy of atomic types roughly corresponds to a part of the simple type hierarchy of XML Schema; some simple types are not considered atomic since their corresponding typed value is a sequence.

Although the kind tests may contain a reference to a type from an XML schema, their expressing power is significantly lower than the power of the XML schema language. In particular, there is no way to restrict the content of a node when the corresponding document was not validated.

2.3 XML Schema

W3C XML Schema [79] is a powerful language designed to define the *schema* (i.e. the structure and type properties) of XML documents and to assist in their processing. The XML Schema language is a replacement for the older and less powerful DTD [15]. Essentially, a schema serves the following purposes:

- To restrict the structure of the document, including the node-kind and the node-name properties of nodes, using the mechanism of complex types.
- To assign simple types to attributes and (some) element nodes; this assignment restricts the syntax of their string value and defines their typed value.

- To define foreign-key relationship between elements and/or attributes.

The presence of a schema may have the following implications in querying:

- Schema-aware storage (see the section 2.5) may be applied for documents conforming to a schema. The complex types allow semantically meaningful shredding of the document while the simple types indicate the appropriate data-type to the values of attributes and simple-content elements. Nevertheless, the schema does not provide enough information to select the optimal storage strategy; therefore, the schema is often augmented with human assistance and/or statistic information [62].

- Structural conformance to a schema allows to prune the search space of various querying techniques, in particular, of twig joins (see the Sec. 2.7).

- Simple types assigned by a schema may be propagated through the query to replace dynamic typing with static type assignment. This is particularly important in the cases where value-based indexes may be applied (see the Sec. 2.6).

2.4 XML query languages

Among XML query languages, the following triplet of languages standardized by the W3C committee is currently the most important.

- *XPath 2.0* [10] – based on the *axis step* paradigm, an XPath query returns a sequence of references to input document nodes.

- *XQuery 1.0* [12] enriches XPath 2.0 with two key features – tree construction and user-defined functions – along with a number of minor improvements.

- *XSLT 2.0* [50] is a language equivalent in strength to XQuery (a method of conversion is described in [30]); however, their application areas are traditionally different – XSLT is used primarily in format conversions and transformations of complete XML documents, while XQuery is used, as its name suggests, in querying.

The XSLT language is clearly divided into the XPath layer and the proper XSLT layer. Therefore, XSLT implementations are often built upon an

XPath engine. On the other hand, this layered implementation is hardly applicable in XQuery, because specific constructs may be mixed inside XPath expressions. In XQuery engines, the core usually evaluates an XQuery expression (either the main expression or the interior of a function) while the function declarations and function calls are handled as an upper layer of the language.

It is worth noticing that the predecessors, XPath 1.0 and XSLT 1.0, missed a significant number of important features; therefore, processing methods devised for the 1.0 languages must often be significantly redesigned before upgrading to XPath 2.0 and XSLT 2.0.

2.5 Physical storage models

Serialized representation

Serialized storage models comprehend a wide range of techniques, characterised by the use of a document-ordered sequence of records and the absence of any node identifiers. The tree structure is usually represented by nesting children between a pair of markers or by explicitly storing the count of children of each node. The former case may also be regarded as a materialization of SAX events [16].

The most prominent example of serialized storage model is the canonical *textual representation* of XML. Despite its prevalence, this strategy is absolutely inappropriate for a competent XML storage due to its verbosity, the need to escape meta-symbols, no support for random access, and other issues.

These disadvantages are addressed by *binary XML* techniques. There is a number of such formats, none being generally accepted as a standard. In general, this approach reduces the size of a document by indirect representation of element and attribute names and more effective binary delimitation of string data. In the presence of a schema, further savings may be achieved using binary encoding of typed values and using references to the schema. Moreover, some binary XML formats have some level of support for random access and/or update.

Binary XML techniques are used in industrial database systems, either as stand-alone collections of documents or as XML data embedded into attributes in a relational DBMS.

Microsoft SQL Server 2005 [68] introduced a new data type capable to store complete XML documents or XML fragments using a kind of binary XML encoding. Attributes of this type may be queried using a SQL function

capable to evaluate a subset of the XQuery language. The system is capable to evaluate the query either based solely on the XML blob data or using a XML-specific index (see the section 2.6 for further discussion).

Oracle XMLDB [55] offers binary XML storage (together with indexing options) as one of three physical representation methods available, the others being object-relational storage and XML-views on relational data.

The main advantage of binary XML is its space-effectivity (being the best among non-compressing methods) and the ability to quickly recover the document (e.g. in the form of SAX events). The main factor that contributes to the low overhead is the absence of node identifiers; thus, when an index is built on a binary XML data, the space advantage disappears.

For applications where space efficiency is the primary goal, *XML compression* techniques were developed [5]. XML compression usually addresses both the document structure and the data, optionally using schema information. Although the compression methods do not necessarily follow the simple serialization pattern, they share the characteristic absence of node identifiers. Thus, from the point of view of XML storage and querying, the compression methods behave similarly to the binary XML techniques, adding space efficiency and, in most cases, jettisoning the random-access and update capabilities.

XML streams, in almost all meanings of this popular label, share the distinguishing features of serialized representation – the document-ordering and the absence of node identifiers. Of course, the usual requirement of one-pass processing creates a separated research branch of XML stream processing which is not a subject of this thesis. Nevertheless, some general query processing methods exhibit one-pass behavior for a subset of queries; thus, such methods may be adapted for stream processing. A notable example of such system is the Raindrop project [51] where algebraic techniques were successfully applied to XML streams.

XML tree

The term *XML tree*, when describing a XML storage method, refers to a graph-like representation of a XML document or its fragment. Each XML node (with the optional exclusion of attribute and text nodes) is represented by a record stored somewhere on a media, containing the required node properties. The document structure is represented by links among the node records in various arrangements:

- Links to all children, usually allowing random access to the i-th child
- A link to the first child and a link to the next sibling

- In addition, a link to the parent may be present

The links are usually implemented in the form of addresses that allow localization of the target node records; these links may be the only way to access the records. For in-memory implementations, memory addresses are usually used; for external media, structured addresses like block/offset pairs are required, sometimes with a level of indirection allowing relocation.

An XQuery engine working above a XML tree may use these addresses to implement node references produced by XPath expressions; moreover, the engine must use them in those arrangements where these references form the only access path to the records. It is important to notice that while the addresses provide a unique key for the set of nodes there is no way to retrieve document-ordering from them.

The XML *DOM* [27] is the best known case of in-memory XML tree. Although DOM is used in many widely spread XSLT and XQuery engines, it can hardly become a base for a high-performance engine due to a number of features missing in the interface [88].

The IBM DB2 engine [65] uses persistent-media XML tree as the primary physical storage, augmented with a combined path/value index (see the section 2.6).

Relational storage

A wide variety of XML storage methods is built using relational database building block, starting from the notion of a table as a set (bag) of tuples through the use of B-trees to the use of relational algebra in query representation and optimization. The level of application of the relational techniques ranges from the adoption of concepts through the reuse of implementation to tight integration in a single SQL/XQuery processing engine.

The relational principles of storage may be applied both in schema-aware and in schema-oblivious manner. Since the knowledge of schema offers a significant advantage, techniques of automatic or user-assisted schema determination were introduced [81] for the cases when there is no schema defined a priori.

In the simplest case, a XML node is encoded in one tuple. In schema-oblivious systems, all the tuples corresponding to nodes of the same kind belong to the same table (i.e. there is one table for all elements, one table for all attributes etc.); in extremely simple implementations, all node kinds may share the same table. In the presence of a schema, the nodes are distributed among tables according to their role in the schema, i.e. based on their name and (optionally) ancestors in the document.

When the XML schema employs inheritance (and also for other reasons), a node may be represented by more than one tuple (as with some object-relative techniques). Some techniques also employ collections in relational attributes, i.e. a violation of the first normal form. In some cases, a XML node may have no representation at all since its presence may be inferred from the schema.

Consequently, a document may be dispersed (*shredded*) across dozens of tables in a schema-aware storage. To recover the original XML document, the original XML schema as well as the selected mapping to the relational schema are necessary. Since the schema may change over time in most applications, adaptive/incremental techniques are required to avoid re-shredding of existing documents on each schema change.

The same set of tables may accommodate a collection of XML documents, in such a case, a document identifier is inserted into the representation of each node.

Whether schema-aware or not, the representation in relational tables assigns a key to the tuple(s) representing each node because each table has at least one unique key. Such a key (together with the identification of the table) may be used in an XQuery engine as a representation of the node.

There are two methods to encode the parent-child hierarchy:

- Inserting foreign keys to the tables representing children to link to their parents
- Using keys that encode parent-children relationship, either with *interval* or *Dewey* encoding

Schema-aware methods usually prefer the foreign key approach, especially when the node key is derived from the node data and, therefore, meaningful. For data-centric applications, the performance of the foreign-key based implementation is superior to other methods [55]. The disadvantage of the method is the necessity for transitive closure to implement recursive XPath axes (ancestor/descendant) in recursive schemata.

Encoding order is a crucial task in relational representation of XQuery. The most common solution is using node identifiers that encode the order. A prominent application of this approach is shown in [24], using the Rainbow framework.

Interval encoding

Interval encoding (also called *Dietz numbering*) [52] is used for instance in the MonetDB/XQuery system [13, 40].

Nodes are identified using a pair of pre-order and post-order ranks, usually augmented with the depth of the node: $N = \langle \mathsf{pre}, \mathsf{post}, \mathsf{depth} \rangle$. In this encoding, XPath axes are represented using a combination of comparisons, like

$$\mathsf{pre}_1 < \mathsf{pre}_2 \wedge \mathsf{post}_2 < \mathsf{post}_1$$

for the `descendant` axis.

Dewey encoding

Compared to interval encoding, Dewey encoding, also known as *dynamic level numbering* [77, 57], offers easier implementation of XML update at the cost of slightly more expensive theta-join operations. The eXist system [61] is an example of a native XML storage.

In Dewey encoding systems, each edge of a XML tree is marked with a label; the order of labels defines the order of children in the tree. A node is identified using the concatenation of the labels attached to the path from the root to the node. XPath axes are represented using prefix test or lexicographic comparison.

2.6 Indexing

In abstract words, an index attached to a XML storage may be used to localize:

- the documents that satisfy a particular condition, and/or
- the nodes of a document that satisfy a particular condition.

In document-centric applications, localizing the document may often be sufficient, while in data-centric applications, node localization is essential. In most cases, an index serves both purposes; therefore, it generates pairs of document and node identifiers. In many cases, indexes produce node identifiers in the document order; such indexes may be used to establish the document ordering among a set of nodes. Document ordering is also essential for the majority of XML-specific structural join algorithms.

The search condition for a node may be based on:

- the *name* of the node
- the *label path* of the node, i.e. the sequence of node names on the path from the root to the node

- the string or typed *value* of the node
- structural relationship to other nodes

An index may combine name/path properties with value properties; such indexes are usually called *combined* path-value indexes. In XML-specific indexing techniques, structural relationship is usually encoded in a form of interval or Dewey numbering. Thus, an index that maps paths and/or values to node numbers may provide structural information as well.

Name-based indexes are implicitly present in each schema-aware relational storage because nodes are distributed to various tables based on their name. Additionally, when a traditional relational index is attached to a column in such a storage, it may be considered a combined name-value index. Creating such an index is usually a user-assisted process; selection of an index during query processing is a matter of traditional physical query rewrite process. Note that there is usually no way to retrieve document ordering from such an index; therefore, these indexes are not well fitted for holistic approaches like twig joins.

The *DataGuide* (or *DataGuides*) [34] is a structure that implements a path-based index in schema-oblivious systems. For a given label path, the DataGuide stores a sequence of corresponding nodes, usually in the document order. The structure may also be used when a search pattern based on node names and structural relations (e.g. a twig-pattern) is given [7]. Since the nodes are usually identified using a kind of interval or Dewey numbering, the output of the DataGuide search may be piped directly to a holistic structural join algorithm (e.g. TwigStack).

A value-based index in a schema-oblivious storage usually indexes all nodes regardless of their name or context. The applicability in a structural-join algorithm requires that the nodes with the same key value are stored in the document order; consequently, the index may be directly used only in the case of exact-match search. For range searches, the set of matching nodes must be sorted before the application of a structural-join algorithm.

Because the semantics of the equality operator in XQuery is wider than the exact-match semantics due to implicit conversions and other issues, sophisticated techniques like the algebra defined in [71] are required to successfully use such an index.

Structures like the DataGuide may be augmented with a value-based index, forming together a combined path-value index. Such an index is applicable when a pattern-based query is combined with an exact-match condition on a value.

IBM DB2/XML [65] is an example of a system based on combined path-value indexes attached to XML trees.

Microsoft has chosen a different approach – their SQL/XML engine supports both path-based and value-based indexes but not combined ones [68].

In applications where schema information is not available, statistics may be used to control the creation of indexes [74].

2.7 XML-specific join methods

Besides *value-based joins* known from relational systems, XML processing requires *structural joins* to handle XPath axes. Structural joins are based on a node identification scheme that allows determining the structural relationship between two nodes solely from their identifiers. Majority of structural join techniques can accommodate both interval encoding and Dewey numbering.

At the basic level, a structural join operator consumes two sets of nodes and generates those pairs of nodes that satisfy a particular axis relationship. There are four basic XPath axes, `descendant`, `ancestor`, `preceding`, and `following`; the other axes may be derived from them (using constraints on node depth if necessary). The `descendant` axis (and its derivative `child`) is the most frequently used one.

Being based solely on the node identifiers, a structural join is formally a special case of theta-join, applying a particular condition on a pair of identifiers. For interval encoding, the condition is a conjunction of numeric comparison predicates, e.g. $a.\text{pre} \leq b.\text{pre} \wedge b.\text{post} \leq a.\text{post}$ for the `descendant` axis. For Dewey encoding, the condition is based on string prefix and/or string comparison predicates. Most relational systems are already capable of handling such conditions; therefore, the importance of special structural join techniques is due to their superior effectivity [4].

Staircase join [38] is one of the most important implementations of structural join, based on interval encoding. Originally, it was implemented as an extension to the relational *Monet* engine, serving as a minimalistic support for XML-specific operations in a relational encoding of XML. There are two versions; the basic one consumes the input relations completely, the advanced version issues *skip* commands to one of the inputs allowing to reduce the I/O cost of the operation.

Advanced structural join techniques can handle more than two sets of nodes at once. *Path join* methods are able to process a chain of `descendant` or `child` axes, i.e. XPath expressions like `$X//b[`P_1`]/c[`P_2`]//d[`P_3`]`. In this example, the path-join is a quaternary operator – besides the node set `$X`, three sets of nodes retrieved from the input document are fed to the operator, corresponding to the `//b[`P_1`]`, `//b/c[`P_2`]`, and `//b/c//d[`P_3`]` expressions.

When the predicates P_1, P_2, P_3 allow, these inputs are generated by path-value indexes; where combined indexes are not available, value indexes are used to generate `//*[`P_1`]`, `//*[`P_2`]`, and `//*[`P_3`]`.

The principle of path-join methods is fairly simple – all inputs are read in document order (since most indexes produces document-ordered sequences, no sort operation is usually required), performing a kind of merge-join. Therefore, this phase is linear with respect to the sum of sizes of the inputs. Unlike plain merge-joins, the path-join requires additional memory: Linked stacks, operated during the merging phase, are used to generate the output of the join operator – the generating phase is linear with respect to the size of the output. Note that, in the most pathological case, the size of the output may be the product of the sizes of the inputs. The maximum size of the stacks is proportional to the maximum depth of the input document.

Twig join methods [17] are a further improvement of the path-join technique, allowing to process a *twig pattern* at once. A twig pattern is essentially a tree whose nodes correspond to the inputs of the twig-join operator and whose edges represent `descendant` or `child` relationships. Twig patterns are extracted from XPath expressions like

`$X//b[c[`P_1`] and .//d[`P_2`]]/e[`P_3`]`

or

```
for $B in $X//b,
   $C in $B/c[P1],
   $D in $B//d[P2],
   $E in $B//e[P3]
return ($B, $C, $D, $E)
```

Note that these two examples produce the same twig pattern; the difference is that while the second query produces quadruples, the first query produces only the `$E` nodes and the other three variables are existentially quantified.

Most advanced twig join methods [47] are capable to handle *or* operators in predicates, like in `$X//b[c[`P_1`] or .//d[`P_2`]]/e[`P_3`]`.

An interesting extension to twig-pattern matching was described in [57]: An extended form of node identifiers, named *extended Dewey*, combines the standard Dewey number with the label path (see the Sec. 2.6). This approach allows to process the twig pattern using only the inputs corresponding to the leaves of the twig-pattern; the identifiers of the internal nodes of the pattern (like `$B` in the example above) are derived.

Most structural join algorithms act as merge joins, traversing all their inputs in the document order. Nevertheless, hash-based algorithms also exist [59]. Advanced algorithms involve skipping in the input streams [32], similar to zig-zag joins.

2.8 Query representation and rewriting

Various XML processing methods are based on different formal models. Although algebraic models prevail, formalisms like automata [28, 76], logic [11], or lambda-calculi [56] are frequently used. An interesting application of monadic Datalog to XPath can be found in the Lixto project [35].

Two causes of this diversity may be identified: First, the application area of XML is wider than of relational databases – besides data-centric XML, there are document-centric applications, stream processing, protocols etc. Second, the XML query languages are significantly more complex than SQL; thus, developing a formalism to cover all the language features is very difficult.

Algebraic systems

There is a number of systems that apply relational algebra to XML processing. In this section, we will shortly introduce a selection of them.

The *Pathfinder* system [14, 73] is a front-end attached to the Monet system (see the section 2.5). It is based on a kind of relational algebra and the *loop-lifting* technique [42]. This technique does not employ hierarchical numbering scheme – the absence of hierarchical numbering helps avoiding the safety problems encountered with unlimited domains of hierarchical numbers; on the other hand, the Pathfinder approach cannot employ the numbering scheme to shortcut a long path through FLWOR-induced joins. The effect of long join paths produced by the Pathfinder system was studied in [39].

The XAT algebra [86] used in the Rainbow system forms a long-term background for experiments with query rewrite in the area of XML data retrieval as well as in XML streams. In this environment, *query unnesting* is extensively used, i.e. rewriting the query representation so that any relational operators are performed on first-normal-form relations with atomic attributes [83]. A similar rewriting technique was presented in [60].

On the XAT algebra, an algorithm to remove unreferenced columns was demonstrated in [85]. Nevertheless, such optimization techniques based on XAT presented so far never crossed the boundary of a functions.

XML systems built by major relational system vendors are usually tightly coupled with their older relational core; therefore, based on the relational algebra. The degree of integration varies from retuning the object-relational query optimizer [55] to applying vast extensions to the relational core [68].

The VAMANA system [70] presents a number of techniques known from the relational database systems like cost-based optimization or pipelined execution; however, the system is limited to stand-alone XPath expressions.

Algebraic rewriting techniques were also used on top of the Galax XQuery engine [71].

Algebraic models are also employed in the area of incremental XML view maintenance [26].

Static rewriting

Several methods of XML query rewriting was presented without the use or relational algebra. The first branch of this area is XQuery canonization, i.e. selecting a subset of the language and rewriting the rest. A number of *canonical* or *core* forms of XQuery was already presented. Besides the W3C standard *XQuery Core* [22], probably the most complete attempt can be found in [80].

During canonization, particular problems may be solved. For instance, the elimination of intersect and except operators was addressed by the rewriting technique described in [37]. The automatic determination of unordered context in XQuery programs was tackled in [43]. Although the presented algorithm is quite efficient, it does not handle general user-defined functions.

2.9 Tree generation

Temporally existing trees were introduced in the XQuery 1.0 and XSLT 2.0 standards and many XQuery processing models can handle them. Examples may be found in the [69], based on the TLC algebra, or in the theoretical study on the tree-transformation power of XSLT [46]. However, the ignorance of user-defined functions rendered the problem simpler than it really was. In the presence of user-defined functions, more sophisticated models and algorithms are required.

The Pathfinder algebra, as well as the majority of the others, deals only with the querying part while the generative part of the XQuery language is implemented without theoretical back-up. For instance, *suspended element constructor* technique is implemented in the Sedna system [31].

The stability of output numbering, which is an important consequence of our approach, is essential in the area of XML views. The area was already extensively studied using various approaches. In [54], an XQuery view definition is transformed to a special language allowing bidirectional mapping between the input and the output document. A completely different approach is used in [36], where the view definition is transformed using the query applied atop the view.

2.10 Function handling

In contemporary XQuery processors, function boundaries usually form a barrier that optimizers cannot penetrate. The problem of inter-procedural optimization of XQuery is recognized but rarely studied [49].

Contemporary XQuery processing and optimization techniques are usually focused on querying and, in most cases, ignore the existence of user-defined functions. One of the rare exceptions may be found in the stream-processing engine described in [28]. In the era of XSLT 1.0, the implementation techniques had to recognize user-defined functions (templates) well (see for instance [36]); however, this branch of research appears discontinued as the community shifted to XQuery.

Pathfinder does support user-defined functions; however, the extent of optimization available across the function boundaries is unclear. Anyway, the cardinality forecast system based on the Pathfinder algebra described in [78] can propagate the information through function calls.

The paper [1] addresses explicitly the problem of recursive user-defined functions in XQuery, using a controlled form of recursion, limited in its applicability to *distributive* functions. This approach is related to the fixed-point systems known from relational databases.

In relational algebra, transitive closure is usually used as the only extension towards recursion. There is extensive theoretical background showing that the transitive closure, being strong enough to capture linear recursion [45] and certain cases of double recursion [84], seems to be sufficient for quite a wide class of problems.

There were infrequent attempts to study different mechanisms of recursion over relational databases, for instance the RQL language [3].

Recursive functions were successfully implemented in the streaming engine described in [53]. However, the implementation applies only to those XQuery programs that are evaluable in single pass.

2.11 Turing completeness

There are various approaches to the classification of the strength of the XQuery language. Using string or numeric expressions and the presence of recursion, one may easily prove Turing-completeness of the XQuery language. However, such plays with atomic values do not belong to typical XQuery use cases. Similarly, the unlimited power of XQuery to create temporal trees is rarely used. Therefore, it is more useful to determine the strength of a shrunk version of XQuery, stripping off functions on atomic values, ordered

sequences, and navigation over temporary trees. Such a shrunk language has its memory limited to the sets of source document nodes, therefore, it is no longer Turing-complete. Nevertheless, this language is still at least as strong as context-sensitive languages (see [66]), therefore, many static analysis problems like termination are intractable. A different approach may be found in [67] where the Turing completeness of XSLT 2.0 is proved using the tree transformation abilities of the language.

Chapter 3

Approach

In this chapter, we will present the motivation using an example and summarize the principles of the method.

3.1 Motivating example

Consider the sub-query "return all persons which were employed on a given date". In XQuery, such a sub-query may be represented by a function. The function `employed` in Fig. 3.1 is parameterized by the date `$P` and return a sequence of nodes, each representing a matching employee. The rest of the query shown there uses the function to print the number of employees for all dates listed in a document. The standard function `fn:count` is used to compute the length of the sequence of employees for each date given.

A naïve implementation calls the function `employed` for each date in the given history. This situation is depicted in Fig. 3.2 using the terminology of relational database systems, augmented with a XML-specific operator of node construction. Note that, due to the nature of the condition placed on the `employee` nodes, value indexes on `@hired` and `@fired` can not reduce the number of scanned nodes significantly.

To enlarge the opportunity to optimize, we suggest the following arrangement shown in Fig. 3.3: The function `employed` is statically transformed so that it consumes all the values of the parameter `$P` at once. Consequently, the transformed function returns all the original return values in a single batch. In other words, the transformed function is called only once, instead of repeated calling in the naïve approach.

Bulk evaluation offers the ability to use more effective join techniques than nested-loop evaluation. In our example in Fig. 3.3, a kind of *theta-join* is used to combine the set of parameter values with the set of employee nodes

```
declare function local:employed($P as xs:date) as element()*
{
    fn:doc("company")//employee[ @hired lt $P and @fired gt $P]
};

<report>{
    for $D in fn:doc("history")//@date return
        <point
            date="{$D}"
            number="{fn:count(local:employed($D))}"
        />
}</report>
```

Figure 3.1: An XQuery function returning a sequence of nodes

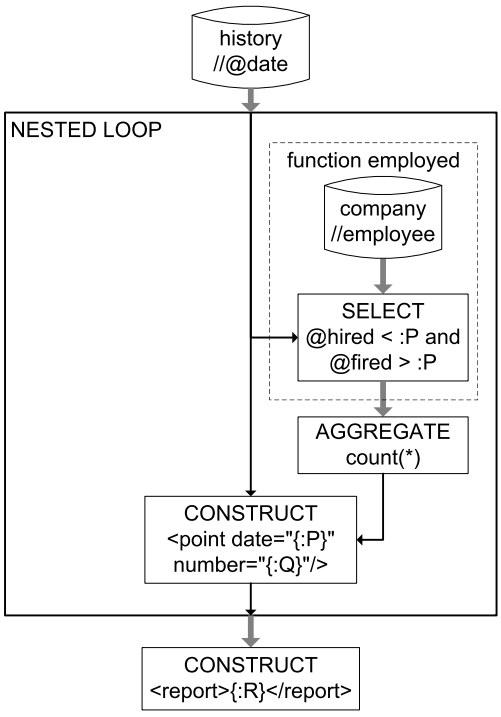


Figure 3.2: Naïve processing

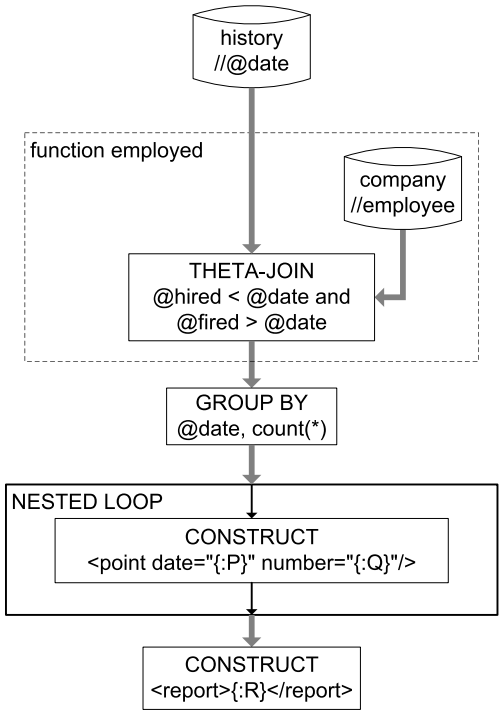


Figure 3.3: Bulk processing

retrieved from a storage. To reduce cost, this theta-join may be implemented using repeated range scans on a sorted materialization of the left operand – this arrangement would never be possible in the naïve implementation in Fig. 3.2.

Although the Fig. 3.3 might suggest that the function `employed` was integrated into the main query, this is not the case. Under bulk evaluation, the transformed query still retains its decomposition to functions and the function call operations are preserved, albeit with transformed operands. The key to correctness is that all calls to a function are transformed in the same manner consistent with the transformation of the function body.

Procedure integration known from compiler construction [72], would indeed produce similar result; however, at the cost of exponential growth of the query size. Moreover, procedure integration cannot handle recursive functions and recursion is natural in XML processing.

The concept of *view merging* [20] in relational databases is also a form of procedure integration, although unparameterized. In the merged query, subsequent transformations are unaware of the original boundary of the view, allowing aggressive optimization (like join reordering) across the hidden boundary.

Behind the ability to handle recursion, the most important advantage of bulk evaluation over procedure integration is the fact that the transformation to bulk-executed functions preserves the size of the query.

Of course, the preservation of function boundaries reduces the maneuvering space of subsequent optimization. However, we will show that some optimization is possible, using rule based transformation of the function interfaces.

Fig. 3.4 shows an improved version of the execution schema shown in Fig. 3.3. Assuming that the attribute `@date` has an ordered index, the theta-join was implemented by a repeated range scan over the index.

If the function `employed` were integrated into the surrounding query, the shift from Fig. 3.3 to Fig. 3.4 would be a relatively simple algebraic transformation. However, the bulk evaluation approach requires that the boundary of the function be still present. Therefore, such a transformation must be formalized as a transformation of the function interface.

In our example, the original interface of the function was composed of the parameter `$P` and the return value. With the shift from naïve to bulk evaluation, the singleton parameter was replaced by a sequence of dates. In Fig. 3.4, the parameter is substituted with a pair of an output parameter and an input parameter. The output parameter is essentially a sequence of intervals; for each such interval, the input parameter returns back a sequence of dates that match the interval.

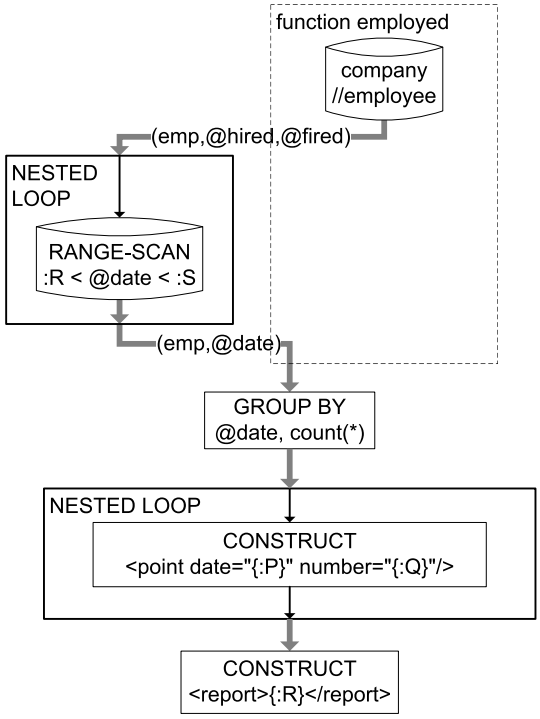


Figure 3.4: Bulk processing with reverted flow

In other words, the caller of the function is expected to perform a range-based theta-join of the original sequence of dates with the intervals generated by the function through the output parameter. The attributes not involved in the join (marked here as `emp`) are just passed around; the fact that it is an employee node is not relevant for the caller.

Although it may seem that we are pulling out all joins from the function, it is not true – only those join conditions that may be implemented with a particular physical access method are worthy of extraction. Therefore, there is only a small number of transformations that may be applied to a function parameter.

Each call to a given function must adapt to the selected interface mode, whether or not it may benefit from it. For instance, if the function `employed` were called only once (wrt. the naïve evaluation), the caller would be still required to perform the theta-join, which would be reduced to a selection operation in this case.

In our example, the input parameter transformed from `$P` is immediately returned back from the function. In general, there may be arbitrary operation applied to the input parameter, including a join or other binary operation with a copy of the data computed during the preceding evaluation of the output parameter.

Therefore, the transformed function can no longer be evaluated in a simple call-return manner. Instead, the ability to pass the control to and from the function more than once is required. Moreover, the function must be able to retain their private data during the time the control is temporarily returned to its caller.

Furthermore, pipelined execution is required to avoid unnecessary materialization of intermediate results. Therefore, a function may run effectively in parallel with its caller, returning the first data for output parameters sooner than the last data for input parameters arrive.

We should emphasize that the strength of the XQuery language brings more difficulties than those featured in Fig. 3.3:

- The original order of `@date` nodes must be recovered from the output of the transformed function `employed`. Moreover, these nodes must maintain their full identity despite the fact the original function consumed only the atomic `xs:date` values extracted from them.

- XQuery functions may be parametrized by sequences; for instance, the function `qualified` in Fig. 3.5 returns the employees which have all the skills given by the parameter `$S`. (The rest of the query in Fig. 3.5 creates a report containing a list of positions; for each position, all appropriately qualified employees are listed.) By definition, an individual

```
declare function local:qualified($S as element()*) as element()*
{
    fn:doc("company")//employee
        [ every $V in $S satisfies skill[ @name = $V/@name]]
};

<report>{
    for $P in fn:doc("positions")//position return
        <position id="{$P/@id}">{
            local:qualified( $P/skill)
        }</position>
}</report>
```

Figure 3.5: An XQuery function parameterized with a sequence of nodes

call to the function is identified by the sequence assigned to `$S`; therefore, a bulk-evaluated call would require join driven by an attribute of sequence type.

- Node construction operators must be expressed in a way compatible with relational algebra although their XQuery semantics corresponds rather to a free-store operation than an algebraic operator.
- When an XQuery function creates a node, each call to this function must, by definition, create a node with a new identity. Therefore, bulk evaluation must carefully observe the equivalence with naïve evaluation.

3.2 Principles

Our approach is based on the following ideas:

- We introduce *R-programs* as an intermediate language combining relational algebra operations with functions. While there is a one-to-one correspondence between R-program functions and XQuery functions, a single call to an R-program function corresponds to multiple calls to the corresponding XQuery function. Inspired by terms like *bulk load* and *bulk operation*, we call this arrangement *bulk evaluation*.
- A particular invocation of an XQuery function is identified by the array of values assigned to the `for` control variables on the descent to the

invocation. In the translation to an R-program function, this definition allows separate handling of function parameters, using the invocation identification as a key that binds corresponding parameter values together. Using this technique, sequence-typed parameters may be dissolved to multiple rows, avoiding nesting in a non-first-normal-form relation. Moreover, each parameter may be handled independently in further optimizations.

- The use of non-first-normal-form relations in the processing chain is reduced to the *hierarchical strings* (see the Section 4.2); no node sets or sequences are packed inside a relation attribute.

- R-program functions are evaluated in multiple passes, in a way inspired by multi-pass attribute grammars [48]. In pipelined implementations of R-programs, functions conceptually run in parallel.

- The multiple-pass nature of functions offers the ability to pass information in the reverse direction with respect to the original data flow. This rearrangement allows reducing the amount of data processed by a function using information from the calling context. This idea is similar to the principle of *Magic sets* approach used in Datalog engines [6].

Chapter 4

Preliminaries

Our mathematical model is based on relational algebra; however, additional operators are required to handle XQuery-specific tasks. Besides atomic types like numbers or strings, relation attributes will carry *identifiers* used for various purposes – Dewey numbering, order maintenance, context identification etc. Most of these identifiers are created during the query evaluation from atomic values and other identifiers; thus, they become compound hierarchical structures.

In this chapter, we will define the notation for domains and tuples. Then, the notion of *hierarchical strings* is defined as the mathematical model of the domain of identifiers. Finally, our selection of relational algebra operators is presented and explained.

4.1 Domains and tuples

Let AttrNm denote the *vocabulary of attribute names* and DomNm be the *vocabulary of domain names*.

(Relation) schema Ω is a partial function that maps a finite set of attribute names to domain names, i.e. $\Omega : \mathsf{AttrNm} \to \mathsf{DomNm}$ such that $\mathrm{dom}(\Omega)$ is finite. The universe of all relation schemata is denoted Sch.

We will show relation schemata using traditional notation like $(a_1 : D_1, a_2 : D_2, a_3 : D_3)$.

For each domain name $d \in \mathsf{DomNm}$, $\mathcal{D}(d)$ denotes the corresponding universe of values (the *domain*). These domains are assumed to be pairwise disjoint; the union of all domains is denoted $\mathcal{D} = \bigcup_{d \in \mathsf{DomNm}} \mathcal{D}(d)$

We will omit $\mathcal{D}$ and use the domain names themselves to denote the associated set of values when there is no risk of confusion.

Tuple over the schema Ω is a partial function $t : \mathrm{dom}(\Omega) \to \mathcal{D}$ such that

$t(a) \in \mathcal{D}(\Omega(a))$ for each $a \in \mathrm{dom}(\Omega)$. *Relation over the schema* Ω is any finite set of tuples over the schema Ω. The universe of all relations over the schema Ω is denoted $\mathcal{U}(\Omega)$. The universe of all relations of all schemata is denoted $\mathcal{U}$.

4.2 Hierarchical strings

We will use the traditional notation (Kleene star) Σ^* to denote the set of all finite words over an alphabet Σ. Besides the empty string (λ) and concatenation operator (.), we will need the following partial functions: $rtrim : \Sigma^* \to \Sigma^*$ and $last : \Sigma^* \to \Sigma$ such that $rtrim(x).last(x) = x$, $ltrim : \Sigma^* \to \Sigma^*$ and $first : \Sigma^* \to \Sigma$ such that $first(x).ltrim(x) = x$, and $after(x, y) = z$ such that $x = y.z$. Furthermore, we define the predicate $prefix(y, x) = (\exists z)(x = y.z)$.

Definition 1 (Hierarchical strings) *Let D be a totally ordered domain.* Hierarchical alphabet over the domain D *is an (infinite) totally ordered superset $\mathcal{H}(D) \supseteq D$ with an injection $\alpha : \mathcal{H}(D)^* \to \mathcal{H}(D)$ that is monotone with respect to the (lexicographical) ordering, $\alpha(u) < \alpha(w) \Leftrightarrow u < w$.* Hierarchical string *is a word over hierarchical alphabet, a member of $\mathcal{H}(D)^*$.*

Hierarchical alphabet may be implemented for instance by the set of unranked ordered trees whose leaves are labeled with the members of D.

On examples, we will write hierarchical strings using parenthesized notation as shown on Fig. 7.2.

In our *identifiers* (see the Chapter 7), the hierarchical strings will be built upon the domain $\mathcal{D}_L = \mathcal{D}_A \cup \mathcal{N} \cup \mathcal{D}_E \cup \mathsf{Addr}$ containing all atomic values, natural numbers, external document identifiers, and AST node addresses (in arbitrary ordering). Based on this definition, the domain of identifiers will be $\mathcal{D}_H = (\mathcal{H}(\mathcal{D}_L))^*$.

Although our notion of hierarchical strings is a bit unusual, it is backed by a theory of strings over infinite languages. Such languages were already applied in the area of XML [64], although in a different manner. The extensive use of string operations in the environment of relational algebra was already studied in [9].

4.3 Relational algebra notation

We will use the following operators borrowed from classical relational algebra (see, for instance, [33]):

- *Union* and *set difference*, $R \cup S$ and $R \setminus S$, on two relations with the same set of attributes.

- *Natural join*, $R \bowtie S$, as a Cartesian product tied by equivalence on common attributes.

- *Selection*, $\sigma[P(a_1, \ldots, a_n)](R)$, based on a predicate P.

- *Projection and duplicate elimination*, $\delta\pi[a_1, \ldots, a_n](R)$, reducing the relation R to the attributes $a_1, \ldots, a_n$. For removing attributes, we will also use the abbreviation $\delta\pi[\backslash a_1, \ldots, a_n](R)$ for $\delta\pi[A_R \backslash \{a_1, \ldots, a_n\}](R)$ where A_R is the set of attributes of R. Note that our algebraic model is defined on sets (as opposed to bags); therefore, our operator of projection is coupled with duplicate elimination. We are still using the traditional symbol δ for duplicate elimination in combination with π to emphasize this fact.

- *Rename*, $\pi[b/a](R)$, renaming the attribute a to b.

- *Function application*, $\pi[b := f(a_1, \ldots, a_n)](R)$ which adds a new attribute b to the relation R, based on the function f and the values of the attributes $a_1, \ldots, a_n$.

- *Grouping*, $\gamma[a_1, \ldots, a_n, b := g(d)](R)$ creates groups from the relation R based on the grouping attributes $a_1, \ldots, a_n$ and adds a new attribute b whose value is given by the aggregate function g applied to the bag of values of the attribute d corresponding to the given group. We will also use the abbreviation $\gamma[\backslash c, b := g(d)](R)$ for $\gamma[A_R \setminus \{c, d\}, b := g(d)](R)$ where A_R is the set of attributes of R. Traditional aggregate functions `count`, `sum`, `min`, and `max` are used.

For specific operations of XQuery, we will also use the following operators:

- *Sequential numbering*, $\nu[a_1, \ldots, a_n, b := @(c)](R)$ adds a new attribute b to the relation R, whose value is the 1-based sequential number of the row among the rows having the same values of $a_1, \ldots, a_n$ with respect to the ordering defined by the attribute c.

- *Ordered grouping*, $\gamma[a_1, \ldots, a_n, b := g(c, d)](R)$ works similarly to the plain grouping operator except that the aggregate function g is order aware, thus applied to the sequence of values of the attribute d ordered by the attribute c. We will also use the abbreviation $\gamma[\backslash, b := g(c, d)](R)$ for $\gamma[A_R \setminus \{c, d\}, b := g(c, d)](R)$ where A_R is the set of attributes of R. For XML, we will need two aggregate functions, `cat` and

`catd`; the former being plain concatenation, the latter concatenation with spaces as separators.

- *Ordered run grouping*, $\xi[a_1, \ldots, a_n, P(e_1, \ldots, e_n), b := g(c, d)](R)$ detects groups as the longest runs of rows (ordered by the attribute c) for which the predicate $P(e_1, \ldots, e_n)$ is true and the value of attributes $a_1, \ldots, a_n$ is the same. The rows having $P(e_1, \ldots, e_n)$ false are then discarded. Each group is replaced by a row with attributes $a_1, \ldots, a_n$ equal to the original ones, the attribute c set to the value of c in the first row in the run attribute, and a new attribute b computed using the ordered aggregate function g (same as for ordered grouping). The remaining attributes of R, including d are discarded.

- *Sequence generation*, $\omega[b := c \ldots d](R)$ where c and d are attributes of integer domain, replaces each row in R by a (possibly empty) set of rows with the additional attribute b whose values iterate through the range from c to d.

Note that only ordered grouping and sequence generation are true additions to relational algebra. Sequential numbering and ordered run grouping may be expressed in terms of the other operators – they are treated separately since they will be likely implemented directly. As we will see later, the exotic operators are used only in less frequent XQuery constructs like positional (`at`) variables or range (`to`) expressions. The ordered run grouping operator is dedicated to the normalization of the constructed documents and, as shown in the section 8.4, may be concentrated at the very end of the processing pipeline.

To improve readability, we will sometimes prefer textual names for infix operators $\bowtie$, $\cup$, and $\setminus$, as shown in the following definition.

Definition 2 (Relational operators) *Binary relational operator is a member of the following set:*

$$\mathsf{BinRelOp} = \{\ \mathsf{join}, \mathsf{union}, \mathsf{except}\ \}$$

Unary relational operator is a member of the following set:

$$
\begin{aligned}
\mathsf{UnRelOp} = {} & \{\ \mathsf{id}\ \} \\
& \cup \{\ \sigma[P] \mid P \textit{ is a predicate}\ \} \\
& \cup \{\ \delta\pi[A] \mid A \subseteq \mathsf{AttrNm}\ \} \\
& \cup \{\ \pi[b/a] \mid a, b \in \mathsf{AttrNm}\ \} \\
& \cup \{\ \pi[a := E] \mid a \in \mathsf{AttrNm}, E \textit{ is an expression}\ \} \\
& \cup \{\ \nu[A, b := @(c)] \mid A \subseteq \mathsf{AttrNm}, b, c \in \mathsf{AttrNm}\ \} \\
& \cup \left\{\ \gamma[A, b := g(d)] \;\middle|\; \begin{array}{l} A \subseteq \mathsf{AttrNm}, b, d \in \mathsf{AttrNm}, \\ g \textit{ is an aggregate function} \end{array}\ \right\} \\
& \cup \left\{\ \gamma[A, b := g(c, d)] \;\middle|\; \begin{array}{l} A \subseteq \mathsf{AttrNm}, b, c, d \in \mathsf{AttrNm}, \\ g \textit{ is an ordered aggregate function} \end{array}\ \right\} \\
& \cup \{\ \omega[b := c \ldots d] \mid b, c, d \in \mathsf{AttrNm}\ \}
\end{aligned}
$$

There is one constant – the empty relation – parameterized by the required schema:

$$\mathsf{NulRelOp} = \{\ \emptyset[\Omega] \mid \Omega \in \mathsf{Sch}\ \}$$

Finally, relational operator is binary, unary, or constant:

$$\mathsf{RelOp} = \mathsf{BinRelOp} \cup \mathsf{UnRelOp} \cup \mathsf{NulRelOp}$$

Chapter 5

R-programs

The recursive nature of XQuery requires an addition to the classical relational algebra. Traditionally, relational algebra is being augmented with transitive closure operator and the researchers then strive to squeeze their recursive problem to the transitive closure or a kind of fix-point operator [45, 75, 2].

However, transitive closure is a *safe* operator that cannot produce new attribute values and that always terminates. This observation shows that it is insufficient to generally model the XQuery language with its tree constructors and possibility of non-termination.

In our approach, we suggest a formalism that is more close to the original nature of the XQuery language. While such a model is less elegant than the transitive closure in mathematical sense, the behavior of the model is not skewed towards the transitive closure idiom. This in turn allows unbiased analysis and optimization based on the model.

An *R-program* consists of a set of *R-functions*. The interior of each R-function is described by a directed graph of (extended) relational algebra operators and R-function calls. Each R-function receives one or more relations as its input arguments and produces one or more relations at its output.

Since the language of R-programs does not offer any programmatic structures like conditions or loops, any recursive R-program would immediately fall into endless recursion. To give recursive R-programs their semantics, the notion of controlling argument is defined that allows to predict the output and to stop the recursion when the controlling arguments are empty. This mechanism corresponds to the case when recursion in an XQuery program is stopped by iterating over an empty set. Of course, termination of R-programs is not generally guaranteed just as the termination of XQuery programs is not.

5.1 R-nets and R-programs

The notion of R-net forms the core of our formalism, representing a directed graph of operations. Besides relational algebra operators, function calls are allowed.

A special operator trigger is introduced here which will be used later in the definition of expansion. The trigger is a unary sink, consuming one operand and producing none; the purpose of the operator is to check the validity of computation – any non-empty input to a trigger invalidates the computation.

Note that the definition of R-net does not require that the directed graph be acyclic. Acyclicity will be studied on the complete R-program, allowing a kind of cycle around a function call. Such a cycle does not necessarily paralyze the evaluation of the program; it may just require multiple entry and exit to the same function.

This approach to acyclicity is crucial; it allows the reversal of evaluation direction which is the base of the output-driven mode described later in this work.

Let ArcNm be the *vocabulary of arc names*; we assume $\{0, 1, 2\} \subset \mathsf{ArcNm}$. We will use them to bind actual arguments to formals in function calls as well as to distinguish input operands in non-commutative operators.

Definition 3 (R-net) *Let* Fncs *be a set of function names.* R-net over Fncs *is an octet*

$$N = (\mathsf{Plcs}, \mathsf{Ops}, \mathsf{In}, \mathsf{Out}, \mathsf{sch}, \mathsf{op}, \mathsf{ini}, \mathsf{fin})$$

where Plcs *is a finite set of* places, Ops *is a finite set of* operations, $\mathsf{ini}, \mathsf{fin} \in \mathsf{Ops}$ *are* initial *and* final operations.

$$\mathsf{In} : (\mathsf{Ops} \setminus \{\mathsf{ini}\}) \times \mathsf{ArcNm} \to \mathsf{Plcs}$$

$$\mathsf{Out} : (\mathsf{Ops} \setminus \{\mathsf{fin}\}) \times \mathsf{ArcNm} \to \mathsf{Plcs}$$

are finite partial mappings that define the input *and* output arcs. *The* Out *mapping must be a projection, i.e.* $\mathrm{rng}(\mathsf{Out}) = \mathsf{Plcs}$. *The mapping*

$$\mathsf{sch} : \mathsf{Plcs} \to \mathsf{Sch}$$

associates a schema to each place.

$$\mathsf{op} : (\mathsf{Ops} \setminus \{\mathsf{ini}, \mathsf{fin}\}) \to (\ \mathsf{RelOp} \cup \{\ \mathsf{trigger}\} \cup \{\ \mathsf{call}[f] \mid f \in \mathsf{Fncs}\ \}\ )$$

is a mapping that assigns operators to operations. For each $t \in \mathsf{Ops}$ *such that* $\mathsf{op}(t) \in \mathsf{RelOp}$, *the input arcs carry the arc names* 1 *and* 2 *(for binary operators) and the output arc is named* 0. *The schema* $\mathsf{sch}(\mathsf{Out}(t, 0))$ *of the*

output place must correspond to the schemata $\mathsf{sch}(\mathsf{In}(t,1))$ *and* $\mathsf{sch}(\mathsf{In}(t,2))$ *(if applicable) with respect to the properties of the operator* $\mathsf{op}(t)$*. When* $\mathsf{op}(t) = \mathsf{trigger}$*, there is exactly one input arc labeled* 1 *and no output arc.*

Note that this definition of R-net allows that a single place be the output of multiple operations. This is a kind of redundancy which is allowed by the definition of R-program semantics, provided all the operations generate the same output. This arrangement makes some forms of optimization easier; nevertheless, in the final stage, this redundancy must be removed. Thus, the final versions of R-nets will satisfy the following condition:

Definition 4 (Non-redundant R-net) *An R-net is called* non-redundant *if the mapping* Out *is an injection.*

Note that for function call operations ($\mathsf{op}(t) = \mathsf{call}[f]$), the input and output arcs are not constrained by the definition of R-net; the constraints will be described later, in the definition of R-program. Similarly, the operations ini and fin serve as placeholders for input and output arguments for an R-net and their constraints will be defined later.

We will use the notation $p \xrightarrow[M]{a} t$ as an abbreviation for $p = \mathsf{In}_M(t,a)$ and $t \xrightarrow[M]{a} p$ for $p = \mathsf{Out}_M(t,a)$.

A set of functions, whose bodies are expressed using R-nets, forms an R-program.

Definition 5 (R-program) R-program *is a tuple*

$$M = (\mathsf{Fncs}, \mathsf{Plcs}, \mathsf{Ops}, \mathsf{In}, \mathsf{Out}, \mathsf{sch}, \mathsf{op}, \mathsf{ini}, \mathsf{fin}, \mathsf{owner}^P, \mathsf{owner}^T, \mathsf{main})$$

where Fncs *is a finite set of* function names*. The mappings* $\mathsf{ini}, \mathsf{fin} : \mathsf{Fncs} \to \mathsf{Ops}$ *define the initial and final operations for each function.*

$\mathsf{owner}^P : \mathsf{Plcs} \to \mathsf{Fncs}$ *and* $\mathsf{owner}^T : \mathsf{Ops} \to \mathsf{Fncs}$ *are total functions called* ownership mappings *that induce the following partitioning of* Plcs *and* Ops*:*

$$P_f = \{\ p \in \mathsf{Plcs} \mid \mathsf{owner}^P(p) = f\ \} \qquad T_f = \{\ t \in \mathsf{Ops} \mid \mathsf{owner}^T(t) = f\ \}$$

The mappings In*,* Out*,* sch *and* op *are composed of partitions* In_f*,* Out_f*,* sch_f*, and* op_f *such that*

$$\mathsf{In} = \bigcup_{f \in \mathsf{Fncs}} \mathsf{In}_f \qquad \mathsf{Out} = \bigcup_{f \in \mathsf{Fncs}} \mathsf{Out}_f$$

$$\mathsf{sch} = \bigcup_{f \in \mathsf{Fncs}} \mathsf{sch}_f \qquad \mathsf{op} = \bigcup_{f \in \mathsf{Fncs}} \mathsf{op}_f$$

function $\mathsf{main}\ (m : (a : D, b : D))$
  return $(v : (a : D, b : D))$
begin
  $v := \mathsf{call}[f](m, m, m);$
  $r := (m\ \cup\ v);$
end;

function $f(z : (a : D, b : D),$
  $m : (a : D, b : D),$
  $x : (a : D, b : D))$
  return $(w : (a : D, b : D))$
begin
  $p := \pi[c/b](z);$
  $q := \pi[c/a](m);$
  $r := (p\ \bowtie\ q);$
  $s := \delta\pi[\backslash c](r);$
  $t := (s\ \setminus\ x);$
  $u := (s\ \cup\ x);$
  $v := \mathsf{call}[f](t, m, u);$
  $w := (t\ \cup\ v);$
end;

Figure 5.1: Example: An R-program to compute transitive closure

and, for each $f \in \mathsf{Fncs}$*, the following tuple*

$$N(f) = (P_f, T_f, \mathsf{In}_f, \mathsf{Out}_f, \mathsf{sch}_f, \mathsf{op}_f, \mathsf{ini}(f), \mathsf{fin}(f))$$

is a correct R-net over Fncs*.* $\mathsf{main} \in \mathsf{Fncs}$ *is called the* main function*. Finally, for each* $t \in \mathsf{Ops}$ *such that* $\mathsf{op}(t) = \mathsf{call}[f]$*, the following condition must be met for each* a*:*

$$(\exists p_1 \in \mathsf{Plcs})\ p_1 \xrightarrow[M]{a} t \Leftrightarrow (\exists p_2 \in \mathsf{Plcs})\ \mathsf{ini}(f) \xrightarrow[M]{a} p_2$$

$$(\exists p_3 \in \mathsf{Plcs})\ p_3 \xrightarrow[M]{a} \mathsf{fin}(f) \Leftrightarrow (\exists p_4 \in \mathsf{Plcs})\ t \xrightarrow[M]{a} p_4$$

$$\mathsf{sch}(\mathsf{In}(t, a)) = \mathsf{sch}(\mathsf{Out}(\mathsf{ini}(f), a))$$

$$\mathsf{sch}(\mathsf{In}(\mathsf{fin}(f), a)) = \mathsf{sch}(\mathsf{Out}(t, a))$$

Fig. 5.1 shows an example of R-program in a textual form; the same R-program in a graphical form is shown in Fig. 5.2. This R-program computes the transitive closure m^+ of its argument m.

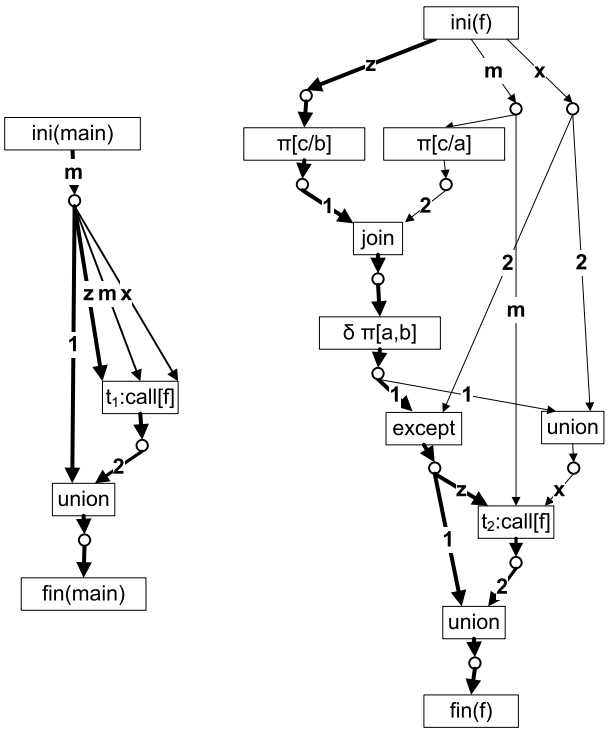


Figure 5.2: Graphical representation of the program from the Fig. 5.1

5.2 Dependency closure and acyclicity

The following definitions form the condition of acyclicity.

Definition 6 (Dependency closure) *Let d be a binary relation on the places of an R-program M such that*

$$d \subseteq \{\ \langle p_1, p_2\rangle \in \mathsf{Plcs}_M \times \mathsf{Plcs}_M \mid \mathsf{owner}^P_M(p_1) = \mathsf{owner}^P_M(p_2)\ \}$$

The dependency closure *of d is the smallest relation $\overline{d} \subseteq \mathsf{Plcs}_M \times \mathsf{Plcs}_M$ such that $d \subseteq \overline{d}$,*

$$\langle p_1, p_2\rangle \in \overline{d} \wedge \langle p_2, p_3\rangle \in \overline{d} \Rightarrow \langle p_1, p_3\rangle \in \overline{d}$$

and

$$\left(\begin{array}{c} \mathsf{op}_M(t) = \mathsf{call}[f] \wedge p_1 \xrightarrow[M]{a} t \wedge \mathsf{ini}_M(f) \xrightarrow[M]{a} p_2 \\ \wedge \langle p_2, p_3\rangle \in \overline{d} \wedge p_3 \xrightarrow[M]{b} \mathsf{fin}_M(f) \wedge t \xrightarrow[M]{b} p_4 \end{array}\right) \Rightarrow \langle p_1, p_4\rangle \in \overline{d}$$

for each $p_1, p_2, p_3, p_4 \in \mathsf{Plcs}_M$, $a, b \in \mathsf{ArcNm}$, $t \in \mathsf{Ops}_M$, and $f \in \mathsf{Fncs}_M$

Note that all conditions in the definition of dependency closure are implications with conjunctive positive premises. Therefore, the set of relations $\overline{d}$ that satisfy these conditions is closed under intersection. Consequently, a single minimum with respect to inclusion exists.

Definition 7 (Acyclic R-program) *Let D be the relation (called* primitive dependency relation*) induced by the primitive operators in an R-program M, i.e.*

$$D = \left\{\ \langle p_1, p_2\rangle \ \middle|\ \begin{array}{l} (\exists t \in \mathsf{Ops}, a_1, a_2 \in \mathsf{ArcNm}) \\ (p_1 \xrightarrow[M]{a} t \wedge t \xrightarrow[M]{b} p_2 \wedge \mathsf{op}_M(t) \in \mathsf{RelOp}) \end{array}\ \right\}$$

The R-program M is called acyclic *if the dependency closure $\overline{D}$ of the primitive dependency relation is antisymmetric and irreflexive, i.e.*

$$\langle p_1, p_2\rangle \in \overline{D} \Rightarrow \langle p_2, p_1\rangle \notin \overline{D}$$

5.3 Controllers

For acyclic R-programs, evaluation is possible. However, the evaluation may not follow the classical call-return scheme; instead, the R-program functions must be gradually instantiated and integrated into the main function.

Since we have no explicit control structures, the process of instantiation must be stopped by an implicit rule. This is based on the notion of *controller set*. A controller set is such a subset of the input arguments of an R-function that all the outputs of the function are empty whenever all the controller inputs are empty.

Since all relational algebra operators (in our model) produce empty results on empty arguments, the complete set of the input arguments of any function always forms a controller set. However, smaller controller sets will allow better optimization. The following definitions state the criteria on controller set exactly.

Definition 8 (Primitive controller relation) *Let M be an R-program.* Primitive controller relation *is any relation*

$$d_C \subseteq \{\ \langle p_1, p_2\rangle \in \mathsf{Plcs}_M \times \mathsf{Plcs}_M \mid \mathsf{owner}^P_M(p_1) = \mathsf{owner}^P_M(p_2)\ \}$$

such that

$$(\mathsf{op}_M(t) \in \mathsf{UnRelOp} \wedge p_1 \xrightarrow[M]{1} t \wedge t \xrightarrow[M]{0} p_0) \Rightarrow \langle p_1, p_0\rangle \in d_C$$

$$(\mathsf{op}_M(t) = \mathsf{join} \wedge p_1 \xrightarrow[M]{1} t \wedge p_2 \xrightarrow[M]{2} t \wedge t \xrightarrow[M]{0} p_0) \Rightarrow (\langle p_1, p_0\rangle \in d_C \vee \langle p_2, p_0\rangle \in d_C)$$

$$(\mathsf{op}_M(t) = \mathsf{union} \wedge p_1 \xrightarrow[M]{1} t \wedge p_2 \xrightarrow[M]{2} t \wedge t \xrightarrow[M]{0} p_0) \Rightarrow (\langle p_1, p_0\rangle \in d_C \wedge \langle p_2, p_0\rangle \in d_C)$$

$$(\mathsf{op}_M(t) = \mathsf{except} \wedge p_1 \xrightarrow[M]{1} t \wedge p_2 \xrightarrow[M]{2} t \wedge t \xrightarrow[M]{0} p_0) \Rightarrow \langle p_1, p_0\rangle \in d_C$$

Note that the previous definition contains a *disjunction* on the right-hand side of the condition for the join operator. It means that either input of a join may be selected in a primitive controller relation. In other words, there may be more than one primitive controller relation for an R-program; there number of such relations is exponential with respect to the number of join operators in the program. Consequently, more than one controller set may exist.

The purpose of controllers is to define where triggers shall be placed in an R-program. Instead of trying to find the best controller set automatically among the exponential number of controller sets, the primitive controller relation is defined statically during the translation of the XQuery to an R-program.

Definition 9 (Controller set) *Let d_C be a primitive controller relation and $\overline{d_C}$ be its dependency closure. For each function $f \in \mathsf{Fncs}$, the set*

$$D^C(f) = \left\{\ p_1 \in \mathsf{Plcs} \;\middle|\; \begin{array}{l}(\exists p_2 \in \mathsf{Plcs}, a, b \in \mathsf{ArcNm}) \\ (\ \mathsf{ini}_M(f) \xrightarrow[M]{a} p_1 \wedge \langle p_1, p_2\rangle \in \overline{d_C} \wedge p_2 \xrightarrow[M]{b} \mathsf{fin}_M(f)\ )\end{array} \right\}$$

is called a controller set of f.

The Fig. 5.2 shows a primitive controller relation (and portions of its dependency closure) using thick arrows.

5.4 Semantics of R-programs

The execution of an R-program is based on instantiation of R-functions and merging them into one R-net. An instantiation is identified by a *call stack* that collects the call instructions along the descent. The following definitions specify this notion explicitly.

Definition 10 (Call automaton) *Let M be an R-program. The* call automaton of M *is the following finite state machine*

$$\mathsf{CA}(M) = (Q, \Sigma, \delta, q_0, F)$$

whose set of states is $Q = \mathsf{Fncs}_M$, its initial state is $q_0 = \mathsf{main}_M$, and all states are final ($F = Q$). The alphabet is formed by the call instructions of M:

$$\Sigma = \{\ t \in \mathsf{Ops}_M \mid (\exists f \in \mathsf{Fncs}_M)\ \mathsf{op}_M(t) = \mathsf{call}[f]\ \}$$

The state-transition function is defined as:

$$\delta = \{\ \langle f_1, t, f_2\rangle \mid t \in \mathsf{Ops}_M \wedge \mathsf{owner}^T_M(t) = f_1 \wedge \mathsf{op}_M(t) = \mathsf{call}[f_2]\ \}$$

The language $\mathsf{CSL}(M) = L(\mathsf{CA}(M))$ accepted by the automaton is called the call stack language.

Definition 11 (Call tree) Call tree *of an R-program M is any finite language $c \subseteq \mathsf{CSL}(M)$ which is prefix-closed, i.e. such that $x.y \in c \Rightarrow x \in c$, and contains the empty word λ.*

A call tree corresponds to a (partial) expansion of the R-program. The expansion creates copies of the instantiated function bodies and glues them together using identity operators on their input and output arguments. On the boundary of the call tree, function calls are not expanded and operators generating empty sets are glued instead of their output values. Triggers are placed at the selected controller subset of the input arguments.

Definition 12 (Expansion) *Let* $c \subseteq \mathsf{CSL}(M)$ *be a* call tree *of an R-program*

$$\begin{array}{rcl} M & = & (\mathsf{Fncs}_M, \mathsf{Plcs}_M, \mathsf{Ops}_M, \mathsf{In}_M, \mathsf{Out}_M, \mathsf{sch}_M, \mathsf{op}_M, \\ & & \mathsf{ini}_M, \mathsf{fin}_M, \mathsf{owner}^P_M, \mathsf{owner}^T_M, \mathsf{main}_M) \end{array}$$

Let $D^C(f)$ *be a controller set for each function* $f \in \mathsf{Fncs}_M$. *The* expansion of M associated to c and D^C *is the following R-net*

$$N = (\mathsf{Plcs}_N, \mathsf{Ops}_N, \mathsf{In}_N, \mathsf{Out}_N, \mathsf{sch}_N, \mathsf{op}_N, \mathsf{ini}_N, \mathsf{fin}_N)$$

whose elements are defined using the following auxiliary definitions:

$$\begin{array}{rcl} C & = & \{\ \langle s,t,f\rangle \mid t \in \mathsf{Ops}_M \wedge f \in \mathsf{Fncs}_M \wedge s.t \in c \wedge \mathsf{op}_M(t) = \mathsf{call}[f]\ \} \\ C' & = & \left\{ \langle s,t,f\rangle \;\middle|\; \begin{array}{l} t \in \mathsf{Ops}_M \wedge f \in \mathsf{Fncs}_M \wedge s \in c \\ \quad \wedge\, s.t \notin c \wedge \mathsf{op}_M(t) = \mathsf{call}[f] \end{array} \right\} \\ K & = & \{\ \langle \lambda, \mathsf{main}_M\rangle\ \} \cup \{\ \langle s.t, f\rangle \mid \langle s,t,f\rangle \in C\ \} \end{array}$$

The expansion is composed of copies of primitive operations distinguished with the symbol R:

$$\begin{array}{rcl} \mathsf{Ops}^R & = & \left\{ \langle R,s,t\rangle \;\middle|\; \begin{array}{l} \langle R,s,f\rangle \in K \wedge t \in \mathsf{Ops}_M \\ \quad \wedge\, \mathsf{owner}^T_M(t) = f \wedge \mathsf{op}_M(t) \in \mathsf{RelOp} \end{array} \right\} \\ \mathsf{In}^R & = & \{\ \langle s,p\rangle \xrightarrow{a} \langle R,s,t\rangle \mid \langle s,p\rangle \in \mathsf{Plcs}_N \wedge \langle s,t\rangle \in \mathsf{Ops}^R \wedge p \xrightarrow[M]{a} t\ \} \\ \mathsf{Out}^R & = & \{\ \langle R,s,t\rangle \xrightarrow{a} \langle s,p\rangle \mid \langle s,t\rangle \in \mathsf{Ops}^R \wedge \langle s,p\rangle \in \mathsf{Plcs}_N \wedge t \xrightarrow[M]{a} p\ \} \end{array}$$

$$\mathsf{op}^R(\langle R,s,t\rangle) \;=\; \mathsf{op}^R(t) \quad \textit{for each } \langle R,s,t\rangle \in \mathsf{Ops}^R$$

In addition, there are identity operations labelled with I corresponding to passing input arguments to R-functions:

$$\begin{array}{rcl} P^I & = & \left\{ \langle s,t,f,p_1,p_2\rangle \;\middle|\; \begin{array}{l} \langle s,t,f\rangle \in C \wedge p_1,p_2 \in \mathsf{Plcs}_M \wedge \\ (\exists a \in \mathsf{ArcNm})(\ p_1 \xrightarrow[M]{a} t \wedge \mathsf{ini}_M(f) \xrightarrow[M]{a} p_2\ ) \end{array} \right\} \\ \mathsf{Ops}^I & = & \{\ \langle I, s.t, p_2\rangle \mid \langle s,t,f,p_1,p_2\rangle \in P^I\ \} \\ \mathsf{In}^I & = & \{\ \langle s,p_1\rangle \xrightarrow{1} \langle I,s.t,p_2\rangle \mid \langle s,t,f,p_1,p_2\rangle \in P^I\ \} \\ \mathsf{Out}^I & = & \{\ \langle I,s.t,p_2\rangle \xrightarrow{0} \langle s.t,p_2\rangle \mid \langle s,t,f,p_1,p_2\rangle \in P^I\ \} \end{array}$$

$$\mathsf{op}^I(\langle I,s.t,p_2\rangle) \;=\; \mathsf{id} \quad \textit{for each } \langle I,s.t,p_2\rangle \in \mathsf{Ops}^R$$

Furthermore, identity operations corresponding to passing output arguments, labelled with O:

$$\begin{array}{rcl} P^O & = & \left\{ \langle s,t,f,p_1,p_2\rangle \;\middle|\; \begin{array}{l} \langle s,t,f\rangle \in C \wedge p_1,p_2 \in \mathsf{Plcs}_M \wedge \\ (\exists a \in \mathsf{ArcNm})(\; p_1 \xrightarrow[M]{a} \mathsf{fin}_M(f) \wedge t \xrightarrow[M]{a} p_2 \;) \end{array} \right\} \\ \mathsf{Ops}^O & = & \{\; \langle O,s,p_2\rangle \mid \langle s,t,f,p_1,p_2\rangle \in P^O \;\} \\ \mathsf{In}^O & = & \{\; \langle s.t,p_1\rangle \xrightarrow{1} \langle O,s,p_2\rangle \mid \langle s,t,f,p_1,p_2\rangle \in P^O \;\} \\ \mathsf{Out}^O & = & \{\; \langle O,s,p_2\rangle \xrightarrow{0} \langle s,p_2\rangle \mid \langle s,t,f,p_1,p_2\rangle \in P^O \;\} \end{array}$$

$$\mathsf{op}^O(\langle O,s,p_2\rangle) \;=\; \mathsf{id} \quad \textit{for each } \langle O,s,p_2\rangle \in \mathsf{Ops}^O$$

Instead of pruned function calls at the boundaries of c, triggers are inserted at those input arguments that are in the selected controller set $D^C(f)$:

$$\begin{array}{rcl} P^T & = & \left\{ \langle s,t,f,p_1,p_2\rangle \;\middle|\; \begin{array}{l} \langle s,t,f\rangle \in C' \wedge p_1,p_2 \in \mathsf{Plcs}_M \wedge \\ (\exists a \in \mathsf{ArcNm}) \\ (\; p_1 \xrightarrow[M]{a} t \wedge \mathsf{ini}_M(f) \xrightarrow[M]{a} p_2 \wedge p_2 \in D^C(f) \;) \end{array} \right\} \\ \mathsf{Ops}^T & = & \{\; \langle T,s.t,p_2\rangle \mid \langle s,t,f,p_1,p_2\rangle \in P^T \;\} \\ \mathsf{In}^T & = & \{\; \langle s,p_1\rangle \xrightarrow{1} \langle T,s.t,p_2\rangle \mid \langle s,t,f,p_1,p_2\rangle \in P^T \;\} \end{array}$$

$$\mathsf{op}^T(\langle T,s.t,p_2\rangle) \;=\; \mathsf{trigger} \quad \textit{for each } \langle T,s.t,p_2\rangle \in \mathsf{Ops}^T$$

Operations labelled with E creating empty relations at the boundaries where function calls were pruned:

$$\begin{array}{rcl} P^E & = & \left\{ \langle s,t,f,p_2\rangle \;\middle|\; \begin{array}{l} \langle s,t,f\rangle \in C' \wedge p_2 \in \mathsf{Plcs}_M \wedge \\ (\exists a \in \mathsf{ArcNm})(\; t \xrightarrow[M]{a} p_2 \;) \end{array} \right\} \\ \mathsf{Ops}^E & = & \{\; \langle E,s,p_2\rangle \mid \langle s,t,f,p_2\rangle \in P^E \;\} \\ \mathsf{Out}^E & = & \{\; \langle E,s,p_2\rangle \xrightarrow{0} \langle s,p_2\rangle \mid \langle s,t,f,p_2\rangle \in P^E \;\} \end{array}$$

$$\mathsf{op}^E(\langle E,s,p_2\rangle) \;=\; \emptyset[\mathsf{sch}_M(p_2)] \quad \textit{for each } \langle E,s,p_2\rangle \in \mathsf{Ops}^E$$

Finally, the net is composed as follows:

$$\begin{aligned}
\mathsf{Plcs}_N &= \{\ \langle s,p\rangle \mid \langle s,f\rangle \in K \wedge p \in \mathsf{Plcs}_M \wedge \mathsf{owner}^P_M(p) = f\ \} \\
\mathsf{Ops}_N &= \{\langle R,\lambda,\mathsf{ini}_M(\mathsf{main}_M)\rangle, \langle R,\lambda,\mathsf{fin}_M(\mathsf{main}_M)\rangle\} \\
&\quad \cup \mathsf{Ops}^R \cup \mathsf{Ops}^I \cup \mathsf{Ops}^O \cup \mathsf{Ops}^T \cup \mathsf{Ops}^E \\
\mathsf{In}_N &= \mathsf{In}^R \cup \mathsf{In}^I \cup \mathsf{In}^O \cup \mathsf{In}^T \\
\mathsf{Out}_N &= \mathsf{Out}^R \cup \mathsf{Out}^I \cup \mathsf{Out}^O \cup \mathsf{Out}^E \\
\mathsf{sch}_N(\langle s,p\rangle) &= \mathsf{sch}_M(p) \quad \textit{for each } \langle s,p\rangle \in \mathsf{Plcs}_N \\
\mathsf{op}_N &= \mathsf{op}^R \cup \mathsf{op}^I \cup \mathsf{op}^O \cup \mathsf{op}^T \cup \mathsf{op}^E \\
\mathsf{ini}_N &= \mathsf{ini}_M(\mathsf{main}_M) \\
\mathsf{fin}_N &= \mathsf{fin}_M(\mathsf{main}_M)
\end{aligned}$$

The first three expansions of the R-program at the Fig. 5.2 are shown at the Fig. 5.3. Identity operations are marked with bold arrows; empty set operations are shown explicitly. The expansions correspond to the call trees $\{\lambda\}$, $\{\lambda, t_1\}$, and $\{\lambda, t_1, t_1t_2\}$, where t_1 and t_2 are the two call operations in the R-program.

The following definition defines the *computation* of an R-net with no function calls. If the computation does not fire any trigger, it means that the assumption that the output of the unexpanded calls is empty was correct. This is the base of the next definition of *controlled computation.*

Definition 13 (Computation of R-net) *Let N be an R-net.* Computation of N *is any mapping*

$$\kappa : \mathsf{Plcs}_N \to \mathcal{U}$$

that satisfies the following conditions:

$$(\forall p \in \mathsf{Plcs}_N)\ \kappa(p) \in \mathcal{U}(\mathsf{sch}_N(p))$$

and, for each $t \in \mathsf{Ops}_N$ such that $\mathsf{op}_N(t) \in \mathsf{RelOp}$

$$\kappa(p_0) = \mathsf{op}_N(t)(p_1, \ldots, p_n)$$

where $p_i \xrightarrow[N]{i} t$ for $i \in 1, \ldots, n$ and $t \xrightarrow[N]{0} p_0$.

Definition 14 (Controlled computation of R-net) *Let κ be a computation of an R-net N. κ is called* controlled *if $\kappa(p_1) = \emptyset$ for each $p_1 \in \mathsf{Plcs}_N, t \in \mathsf{Ops}_N$ such that $p_1 \xrightarrow[N]{1} t$ and $\mathsf{op}_N(t) = \mathsf{trigger}$.*

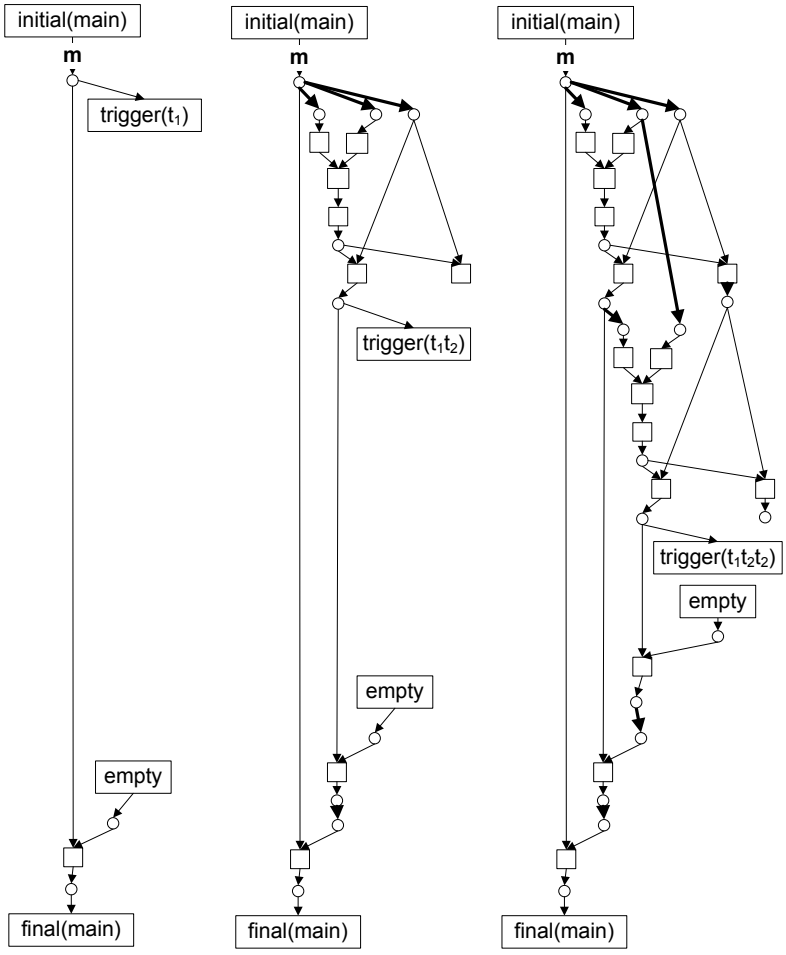


Figure 5.3: Three expansions of the R-program from Fig. 5.2

Lemma 1 (Equivalence of controlled computations)
Let M be an acyclic R-program, D_1^C and D_2^C be controller set assignments, c_1 and c_2 be call trees of M, N_1 be the expansion of M associated to c_1 and D_1^C, and N_2 be the expansion of M associated to c_2 and D_2^C. Let κ_1 be controlled computation of N_1 and κ_2 be controlled computation of N_2 such that

$$(\forall a, p_1, p_2)(\ \mathsf{ini}_{N_1} \xrightarrow[M]{a} p_1 \wedge \mathsf{ini}_{N_2} \xrightarrow[M]{a} p_2 \Rightarrow \kappa_1(p_1) = \kappa_2(p_2)\ )$$

Then

$$(\forall s)(\ s \in c_1 \cap c_2 \Rightarrow (\forall p)(\ \kappa_1(\langle s, p\rangle) = \kappa_2(\langle s, p\rangle)\ )\ )$$

The proof of this lemma is straightforward: The first difference between the computations (based on any topological sort which is ensured by the acyclicity) must be on the boundary of one of the call trees (since all operations inside are deterministic). However, the controller inputs to the unexpanded call must be empty since the computation is controlled; based on the definition of the controller set, it may be shown that the output of the expanded call in the other computation must be empty; therefore it must not be different from the empty relations injected directly at the boundary.

This lemma essentially states that, for fixed input arguments, all controlled computations produce the same results. This allows to define the semantics of the R-program using any controlled computation, if it exists.

Definition 15 (Input and output of an R-program) *Let M be an R-program, let*

$$\begin{aligned} A^I &= \{\ a \mid (\exists p)\ \mathsf{ini}_M(\mathsf{main}_M) \xrightarrow[M]{a} p\ \} \\ A^O &= \{\ a \mid (\exists p)\ p \xrightarrow[M]{a} \mathsf{fin}_M(\mathsf{main}_M)\ \} \end{aligned}$$

Any mapping $i : A^I \to \mathcal{U}$ is called an input *to M, $o : A^O \to \mathcal{U}$ is called an* output *of M.*

Definition 16 (Effect of an R-program) *Let M be an acyclic R-program and $i : A^I \to \mathcal{U}$ be an input to M. If there exists a controller set assignment D^C, a call tree c, and a controlled computation κ_c of the expansion of M associated to c and D^C such that*

$$(\forall a, p)\ \mathsf{ini}_M(\mathsf{main}_M) \xrightarrow[M]{a} p \Rightarrow \kappa_c(\langle \lambda, p\rangle) = i(a)$$

then the output of M on i is $M(i) = o$ such that $o(a) = \kappa_c(\langle \lambda, p\rangle)$ for each a, p such that $p \xrightarrow[M]{a} \mathsf{fin}_M(\mathsf{main}_M)$.

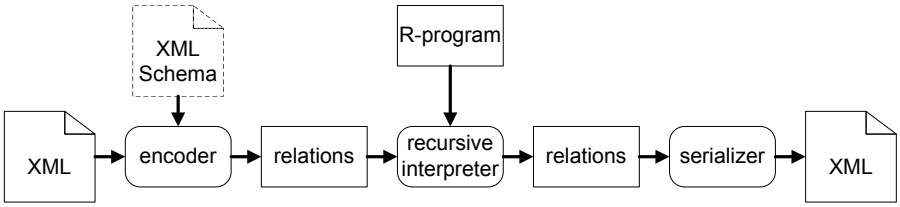


Figure 5.4: R-program run over serialized documents

Thanks to the Lemma 1, this definition is correct because all controlled computations c will compute the same values of $\kappa_c(\langle s, p\rangle)$. Nevertheless, there are acyclic programs that do not have any controlled computation (for some inputs); for such programs and inputs, the output is undefined. Naturally, this situation corresponds to unterminated recursion which may occur in an XQuery program.

The lemma 1 also allows us to simplify the notation – we can omit the call tree c from $\kappa_c(\langle s, p\rangle)$. Additionally, we will omit the angle brackets that originally corresponded to the construction of the set of places in the expansion of M corresponding to c. In other words, $\kappa(s, p)$ now means the value associated to the place p in the invocation associated to the stack s; of course, assuming a fixed R-program and its input.

5.5 Evaluation of R-programs

The Fig. 5.4 depicts the simplest situation where the R-program is run over serialized XML documents. Since the R-program paradigm is relation-oriented, an encoding phase must precede the interpretation of the program and a serializer must then recover the document from the (generally unordered) output of the interpreter.

When the source documents are retrieved from a database that employs a kind of Dewey numbering scheme, the relational data may be passed from the database without an encoder. Similarly, when the output is stored back to the database, the serializer may be omitted, if the database is able to absorb the Dewey identifiers generated by the R-program.

Simple R-programs can be evaluated in *one pass* using the call-return approach. However, advanced R-program require the pipelined architecture shown in the Fig. 5.5. The R-program is gradually expanded by the *expander* into an R-net. The expansion is controlled by *triggers* inserted into the ex-

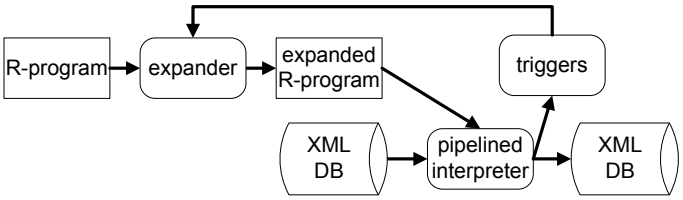


Figure 5.5: Pipelined R-program run-time

panded R-net in all places where an unexpanded R-function is called. When a trigger encounters the first tuple of data arriving in the actual arguments of the R-function, a further expansion step is invoked, consisting of integration of the R-function body into the expanded R-program.

Chapter 6

Compile-time architecture

The R-program representation is created from the XQuery program during a process called *transcription*; the whole compile-time processing chain is shown at the Fig. 6.1.

Before the transcription, a *mode selection* phase analyzes the XQuery program and determines the *evaluation mode* assigned to each variable, expression, and operation in the source program. Each mode consists of a set of relations that hold the value of a variable or expression and transcription rules for each XQuery operator or statement. Some transcription rules form a bridge between different modes.

Employing the flexibility of R-programs, transcription rules may even reverse the flow of information partially; such a reversal is an R-program analogue of *predicate pushing* and *join reordering* known from relational-algebra based query rewriting. Reversed flow forms a key component of the output-driven evaluation method.

Most of the modes are not general; therefore the main goal of the mode selection phase is determining the applicability of the modes at various places in the source program. Besides the *output-driven mode*, there are modes

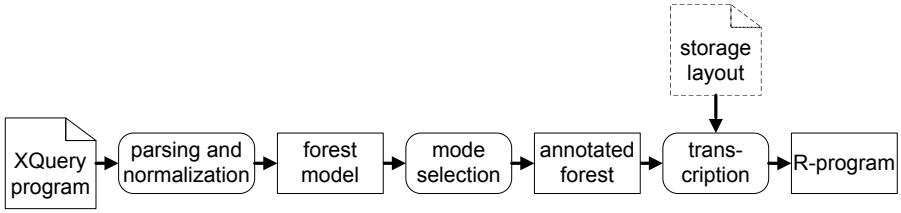


Figure 6.1: Compile-time processing

designed to handle temporal trees, Boolean values, unordered context etc. Where no alternate mode applies, the evaluation fails back to the *canonical mode* which follows the W3C definition [22] of XQuery semantics (but still employs bulk-evaluation).

6.1 XQuery normalization

There are several variants of *core* subsets of XQuery, including the *core grammar* defined in the W3C standard [22], the LixQuery framework [44], and others [30]. Since the XSLT and XQuery are related languages and the translation from XSLT to XQuery is known (see [30]), the model may be applied also to XSLT.

In our model, a selection of operators based on the W3C standard core [22] is used. Our model is deviated from the standard core in the definition of the FLWOR: to handle `order by` clauses properly, we must cope with all the variables at once, whereas in the W3C standard the FLWOR is decomposed to individual variables, leaving the definition of `order by` semantics a bit vague. With this exception, the normalization process defined in the XQuery standard may be applied.

In addition to the standard core, we must also handle those library functions defined in [58] that operate on sequences longer than one and those that access the dynamic context.

The following list summarizes the syntactic structures (as well as important library function calls) produced by the process of normalization (the numbers refer to the corresponding section of the standard [12]):

- declare function and function calls (4.1.5)
- variable references (4.1.2), context item (“`.`”, 4.1.4), `fn:last` (7.2.1), `fn:position` (7.2.2)
- FLWOR expressions (4.8)
- quantified expressions (`some`, `every`, 4.10)
- conditional expressions (`if`, 4.10)
- logical expressions (4.6)
- operations on atomic singletons – arithmetic operators (4.4), value comparisons (4.5.1), and part of library function calls
- literals (4.1.1), empty sequence (4.1.3), range expression (`to`, 4.3.1)

- sequence concatenation (“`,`”, 4.3.1)
- calls to library functions working on sequences (`fn:tokenize`, `fn:index-of`, `fn:empty`, `fn:exactly-one`, `fn:distinct-values`, `fn:insert-before`, `fn:remove`, `fn:reverse`, `fn:subsequence`, `fn:unordered`, `fn:deep-equal`, `fn:count`, `fn:avg`, `fn:max`, `fn:min`, `fn:sum`)
- node-set operators (`union`, `intersection`, `except`, 4.3.3)
- forward/reverse axis navigation (4.2.1.1), `fn:root` (4.2), `fn:id`, `fn:idref`, kind and name tests (4.2.1.2), node comparisons (4.5.3)
- statically named document references (`fn:doc`, `fn:collection`)
- computed constructors (4.7.3)
- implicit conversions – atomization (3.4.2), effective Boolean value (`fn:boolean`, 3.4.3)

In this work, we decided to ignore type-related constructs, namely:

- Type clauses (`as`) in variable, parameter, and function declarations
- `instance of`, `typeswitch`, `cast as`, `castable as`, `treat as`, `validate`
- Constructor functions (see [58])

Nevertheless, we will maintain basic type distinction between atomic values and node references, because the semantics of many operators is affected by the types of their operands in this sense. Moreover, we will use a special mode to represent singleton Boolean values because of their importance.

Similarly to the normative definition of the XQuery semantics, we use (abstract) grammar rules of the *core grammar* [22] as the base for the models. An XQuery program is formalized as a forest of abstract syntax trees (AST), one tree for each user-defined function and one for the main expression. Each node of each AST has a (program-wide) unique *address* $E \in \mathsf{Addr}$. The set $\mathsf{QFncNames} \subseteq \mathsf{Addr}$ enumerates all roots of ASTs assigned to functions, the node $\mathsf{main} \in \mathsf{Addr}$ denotes the root of the main expression.

As an example, we will use the Use Case TREE – Query 1 from the XQuery Test Suite [82], shown in Fig. 6.2. Fig. 6.3 shows the corresponding abstract syntax forest. Node labels are shown as letters left to the nodes.

```
declare function local:toc($P as element()) as element()*
{
    for $X in $P/section
    return <section> {
     $X/@* , $X/title , local:toc($X)
     } </section>
};

<toc> {
     for $S in $I/book return local:toc($S)
} </toc>
```

Figure 6.2: Query 1

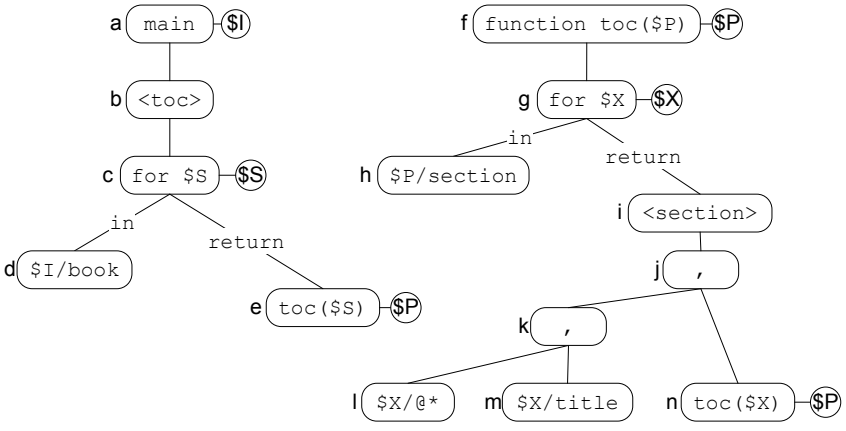


Figure 6.3: Query 1 – Forest model

An (abstract) grammar rule $G(E_0)$ is assigned to each node E_0 of the AST tree, assigning semantics to the relationship between E_0 and its children $E_1, \ldots, E_n$. Formally, the corresponding abstract grammar rule would be $X_0 \to X_1 \ldots X_n$ for some nonterminals $X_0, \ldots, X_n$. In our abstract grammar, majority of rules correspond to expressions; therefore, it is usually not necessary to distinguish between various nonterminals. Thus, we will use node address variables $E_0, \ldots, E_n$ instead of nonterminal names. Furthermore, we will add concrete syntax elements like delimiters to improve readability as in the following example:

E_0 ::= E_1 `union` E_2

(We will use the symbol ::= for grammar rules to avoid confusion with other meanings of arrows in this work.)

Some rules contain literals, XML element names, XQuery function names, or variable names. These items will not be materialized as nodes in the AST; instead, they become parameters of the node E_0. Naturally, we will include these items in the rule, as in:

E_0 ::= `for $y in` E_1 `return` E_2

Of course, the symbol $y is a placeholder for a variable name, not a concrete variable.

For each AST node E, the set $\mathsf{vars}[E]$ contains the names of *accessible variables*. In particular, when E is the root of a function AST, $\mathsf{vars}[E]$ contains the names of arguments of the function.

6.2 Transcription phase

The transcription of an XQuery program to an R-program is driven by a set of *modes* and a set of *transcription rules*. A mode defines the representation of an XQuery value, a transcription rule assigns a partial R-net to an abstract grammar rule. The system is extensible, i.e. new modes and transcription rules may be added.

Each mode is essentially a group of relational tables; in the terminology of R-programs, the tables correspond to places. To bind transcription rules together, these places are identified using the common dictionary ArcNm of arc names.

For every AST node E, the mode selection phase selects a mode $m(E)$. A mode m essentially consists of a set $P(m) \subset \mathsf{ArcNm}$ of arc names and their associated schemata. Applying union to their sets of arc names, *combined modes* modes may be produced from *basic modes*. Most transcription rules are assigned to basic modes; rules for combined modes are defined naturally by superposition of the basic mode rules.

In the transcription process, every AST node is transformed to a set of R-program places, whose identifications are merged from the arc names from $P(m(E))$ and the identification of the node E. We will use the notation $N[E]$ for the place corresponding to an arc name $N \in \mathsf{ArcNm}$ and a node $E \in \mathsf{Addr}$.

Similar process governs the transcription of variables and XQuery function arguments. Each variable or parameter `$x` is assigned a mode $m(E, \texttt{\$x}) \subset \mathsf{ArcNm}$. The mode depends on a particular AST node E since some operations like nesting into FLWOR statements change the representation of visible variables. For an arc name $N \in m(E, \texttt{\$x})$, we will use the notation $N[E, \texttt{\$x}]$ for the resulting R-program place.

For an AST node E_0 and its children $E_1, \ldots, E_n$, the corresponding transcription rule is selected among the transcription rules assigned to the grammar rule $R(E_0)$ so that it fits to the modes $m(E_0), m(E_1), \ldots, m(E_n)$. Similarly to modes, transcription rules may be combined from *basic transcription rules*. Each basic rule is associated to a grammar rule and an $(n+1)$-tuple of modes assigned to the corresponding AST nodes.

Besides the expression value which propagates bottom-up in the AST, variable values shall be propagated top-down. In some cases, their representation may change upon descent through a grammar rule; moreover, their encoding may be affected by the expression value. Thus, the propagation of variable values is governed by variable transcription rules, selected according to the grammar rule, the modes $m(E_0, \texttt{\$x}), m(E_1, \texttt{\$x}), \ldots, m(E_n, \texttt{\$x})$ assigned to the variable at the corresponding nodes, and, sometimes, the modes $m(E_0), m(E_1), \ldots, m(E_n)$ assigned to the expression values at the same AST nodes. Nevertheless, in most cases, the variable transcription rules are trivial, consisting of identity operations.

The following definition summarizes the previous paragraphs in more formal way.

Definition 17 (Transcription rule) *Let $R : X_0 \to X_1 \ldots X_n$ be an abstract grammar rule. Let $m_0^F, m_1^F, \ldots, m_n^F \subset \mathsf{ArcNm}$ and $m_0^R, m_1^R, \ldots, m_n^R \subset \mathsf{ArcNm}$ be sets of forward and reverse arc names, respectively.* Transcription rule *T is an R-net*

$$N_T = (\mathsf{Plcs}_T, \mathsf{Ops}_T, \mathsf{In}_T, \mathsf{Out}_T, \mathsf{sch}_T, \mathsf{op}_T, \mathsf{ini}_T, \mathsf{fin}_T)$$

whose initial and final arc names are pairs $\langle N, i \rangle$ where $i \in \{0, 1, \ldots, n\}$ and $N \in \mathsf{ArcNm}$ such that

$$\{\, a \mid \mathsf{ini}_T \xrightarrow[N_T]{a} p \,\} = \{\, \langle N, 0 \rangle \mid N \in m_0^R \,\} \cup \{\, \langle N, i \rangle \mid i \in \{1, \ldots, n\} \wedge N \in m_i^F \,\}$$

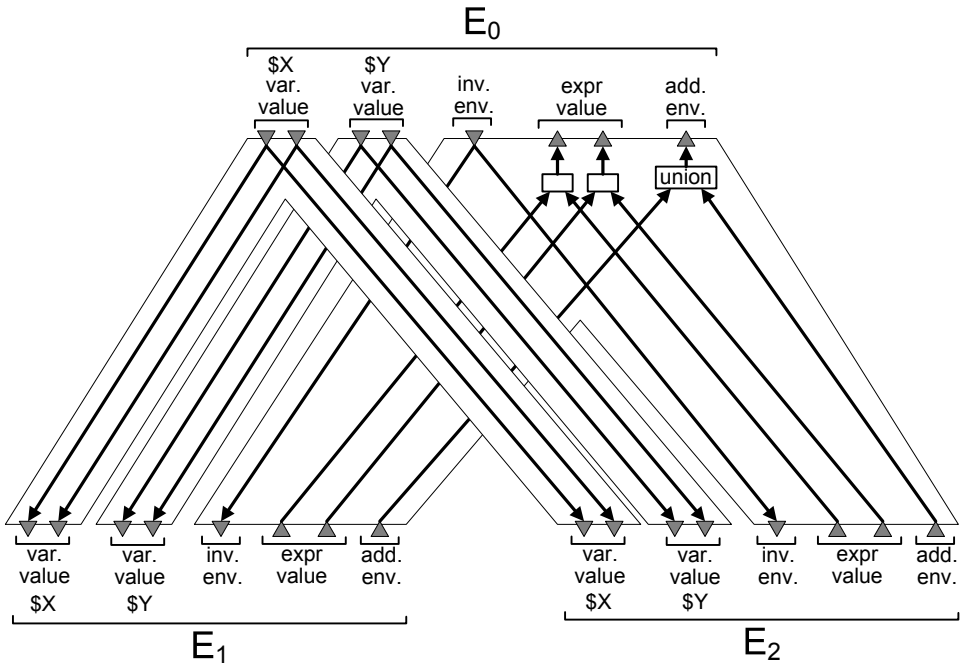


Figure 6.4: A standard-flow rule for a binary operator

$$\{\, a \mid p \xrightarrow[N_T]{a} \mathsf{fin}_T \,\} = \{\, \langle N, 0\rangle \mid N \in m_0^F \,\} \cup \{\, \langle N, i\rangle \mid i \in \{1, \ldots, n\} \wedge N \in m_i^R \,\}$$

The arc names in each mode are divided into two groups, *forward* and *reversed*. The division governs the direction of the R-program arcs in the vicinity of the associated place: The forward arcs propagate expression value in bottom-up manner with respect to the AST tree while the reversed arcs propagate values towards the leaves. For arcs representing variables and arguments, the sense is opposite. Inspired by the notation of attribute grammars [48], we will call the bottom-up propagating arcs *synthesized* and the opposite *inherited*.

The Fig. 6.4 shows the general scheme of a transcription rule for a binary operator, with forward modes only. In this case, the propagation of the expression values is independent. The rule is composed of a core part, covering the invocation environment (inherited), the expression value (synthesized), and the additional environment (synthesized). For each variable, there is an independent portion of the transcription rule (inherited).

For grammar rules that define a new variable, additional dependencies between the core part and the new variable are present as shown in the Fig. 6.5 and 6.6.

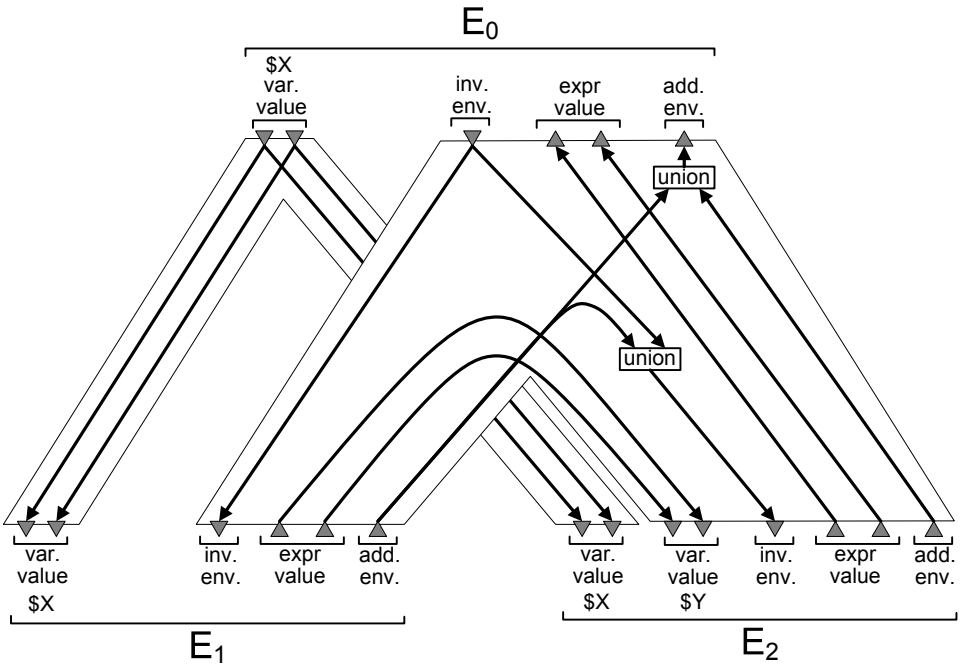


Figure 6.5: A standard-flow rule E_0 `::= let $Y :=` E_1 `return` E_2

A reverted-mode rule is shown at the Fig. 6.7.

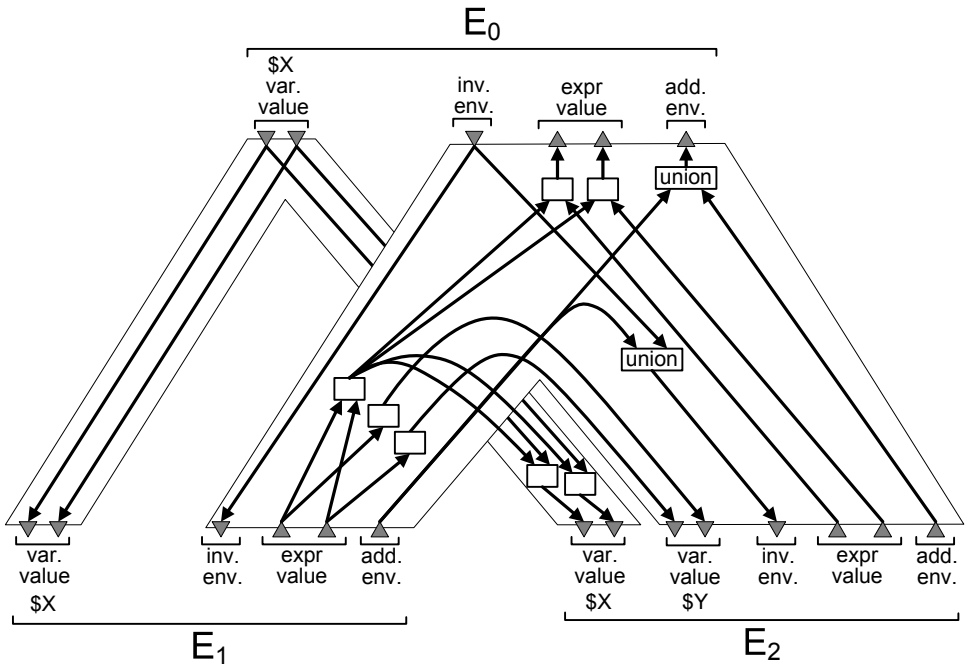


Figure 6.6: A standard-flow rule E_0 `::= for $Y in` E_1 `return` E_2

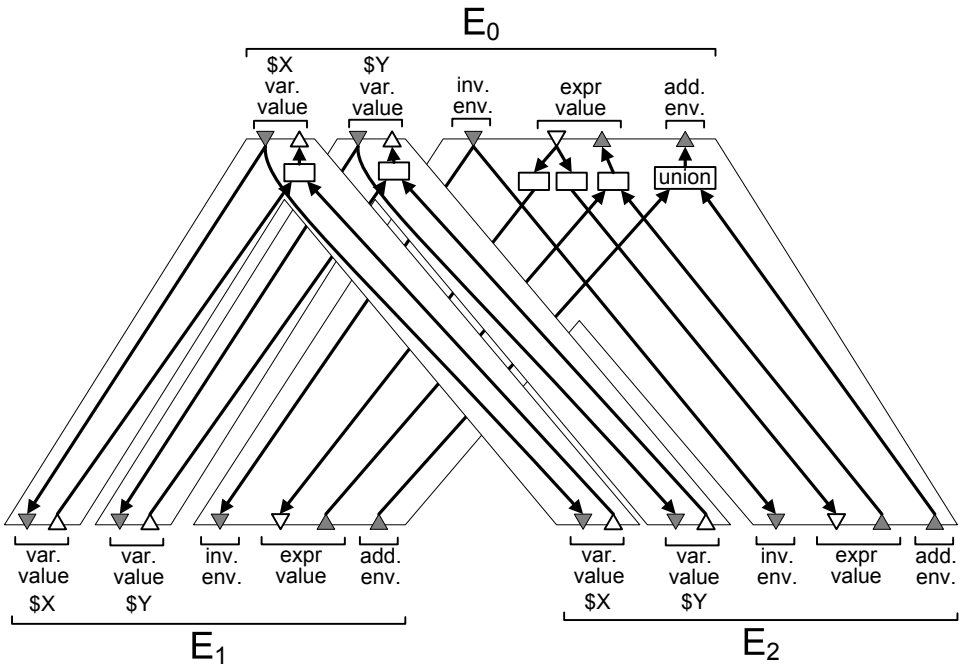


Figure 6.7: A reverted-flow rule for a binary operator

Chapter 7

Canonical mode

The canonical mode is based on the following principles:

- Nodes within a tree are identified by *node identifiers* using *Dewey ID* labeling scheme.
- A tree is encoded using a mapping of Dewey labels to *node properties*.
- A tree created during XQuery evaluation is identified by a *tree identifier* derived from the context in which the tree was constructed.
- A node is globally identified by the pair of a tree identifier and a node identifier.
- A sequence is modeled using a mapping of *sequence identifiers* to *sequence items*.
- Each *sequence* containing nodes is accompanied by a *tree environment* which contains the encoding of the trees to which the nodes of the sequence belong.
- Evaluating a `for`-expression corresponds to iteration through all sequence identifiers in the value of the `in`-clause.
- A particular context reached during XQuery evaluation is identified by the pair of a *call stack*, containing positions in the program code, and a *control variable stack*, containing sequence identifiers selected by the `for`-expressions along the call stack.
- Node identifiers, tree identifiers, sequence identifiers, and control variable stacks share the same domain of *hierarchical strings*, allowing to construct each kind of identifier from the others.

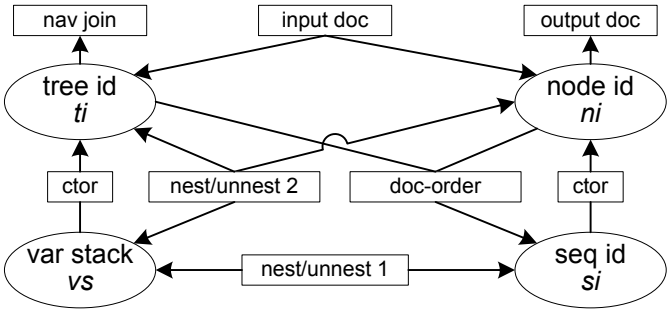


Figure 7.1: Life cycle of identifiers

In our model, various operators like FLWOR or constructors convert one kind of identifier into another. The life cycle of identifiers is shown in Fig. 7.1.

To support the life cycle, we need a common type that supports all the operations on the identifiers. This leads to the notion of *Hierarchical string* defined in the section 4.2.

The model described in this chapter is simplified for the sake of readability:

- We assume that all atomic values are represented using the same domain $\mathcal{D}_A$ and that members of this domain carry both the exact type information and the value. Consequently, all built-in operators and function are expected to act on this domain according to the type promotion and subtype substitution rules as defined in the XQuery standard [22].

- We will not describe the model of type-related constructs `instance of`, `typeswitch`, `cast as`, `castable as`, `treat as`, `validate` and type declarations (`as`) attached to function parameters, return values, and `for`, `let`, `some`, `every`, and global variable declarations.

- We will not model error detection and handling; for inputs that mandate raising an error in the standard, our model may silently produce some results not supported by the standard.

- We ignore any namespace issues, assuming that all names in the query and in the input documents are normalized to *expanded QName's* (see [22]).

7.1 Tree environment handling

XQuery expressions carry node references – the documents (generally, trees) must be represented aside the representation of the expression values. The trees are stored in a *tree environment* – a relational model of a forest of XML trees.

The trees may originate outside the query processor (as the input documents) or they may be created during the query evaluation (by constructors). In the former case, the environment formally propagates from the main expression down to AST leaves and function calls; in the latter case, the environment propagates mostly bottom-up. For an XQuery variable, trees created inside the defining expression must be forwarded down to the variable scope.

This observation leads to the definition of two relations of the same schema, one being passed top-down, one bottom-up:

$$\mathsf{invenv}[E], \mathsf{exenv}[E] : (ti : \mathcal{D}_H, ni : \mathcal{D}_H, nk : \mathcal{D}_K, nn : \mathcal{D}_N, nv : \mathcal{D}_A)$$

$\mathsf{invenv}[E]$ is the tree environment in which the expression is invoked. $\mathsf{exenv}[E]$ is the additional tree environment containing the tree fragments created by the expression E.

ti is a tree identifier. For input documents, $ti \in \mathcal{D}_E$; for trees created during the evaluation, the identifier is in the form $ti = \alpha(cs).\alpha(vs)$; its parts cs and vs correspond to the environment identification at the moment of tree creation.

ni is a node identifier; for input documents, it is the Dewey identifier of the node. For nodes constructed during the evaluation of the query, the node identifier has the properties of Dewey numbering; however, the individual labels are hierarchical, since they are constructed during the evaluation using concatenation and the α operator.

$\langle nk, nn, nv \rangle$ is a tuple of properties assigned to a node by the XQuery Data Model, containing node kind, node name, and the typed value (see the section 2.1).

7.2 Invocation stacks

In bulk evaluation, invocations of a function (or, generally, a sub-expression) are merged together. To distinguish among values corresponding to different invocations, our relational models of XQuery values carry a special attribute – the control variable stack vs. In most cases, this attribute merely propagates through operators and controls join operations. In some cases (like

with literals or constructors), the value representation must be created from scratch; therefore, a source of variable stacks is required.

This requirement is satisfied using the following relation, passed as a reversed arc from the main expression to functions and from AST roots to their leaves:

$$\mathsf{invv}[E] : (vs : \mathcal{D}_H)$$

$\mathsf{invv}[E]$ is the set of variable stacks in which E is evaluated. vs is the stack of sequence identifiers selected by the `for`-clauses throughout the descent to the examined expression. The sequence stack vs, together with a call stack $c \in \mathsf{CSL}(M)$ of the resulting R-program M, forms the identification of the dynamic context in which an expression is evaluated.

While the XQuery standard defines dynamic context as the set of variable assignments (with some negligible additions), our notion of dynamic context is based on the stack pair that determines the descent through the code to the examined expression, combining both the code path stored in c and the `for`-control variables in vs. The key to the sufficiency of this model is the observation that, under fixed outer conditions, the variable assignment is a function of the stack pair.

Constructor operations must generate globally unique tree identifiers; the uniqueness is reached using the combination of a variable stack from the set $\mathsf{invv}[E]$ with a call stack passed through the following relation:

$$\mathsf{invc}[E] : (cs : \mathcal{D}_C)$$

$\mathsf{invc}[E]$ is a single-row relation containing the call stack in which the expression E is evaluated.

7.3 Canonical mode

The canonical mode is based on the following data structures:

- $\mathsf{varseqa}[E, \texttt{\$x}]$, $\mathsf{varseqn}[E, \texttt{\$x}]$ together represent the assignment of the values of the variable $\texttt{\$x} \in \mathsf{vars}[E]$ to the contexts from $\mathsf{invv}[E]$. This representation is applied also to the implicit variables (`.`, `fn:root`).

- $\mathsf{exseqa}[E]$ and $\mathsf{exseqn}[E]$ represent the assignment of the result value of the expression E to the contexts from $\mathsf{invv}[E]$.

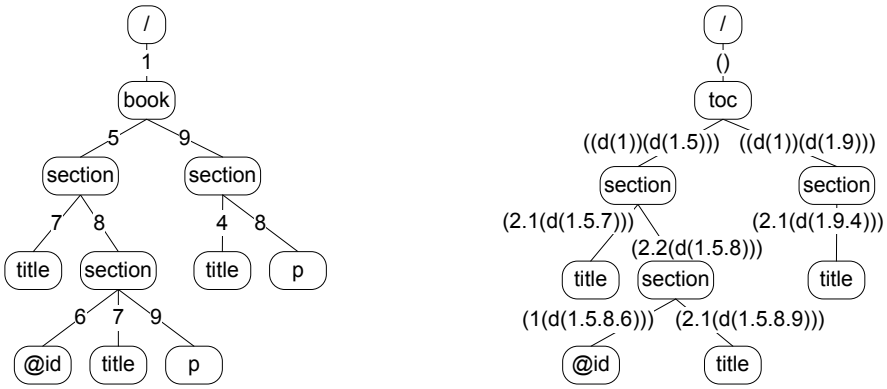


Figure 7.2: Query 1 – Sample input and output documents

The signature of the mode relations (arc names) is as follows:

$$\mathsf{varseqa}[E, \$\mathtt{x}], \mathsf{exseqa}[E] : (vs : \mathcal{D}_H, si : \mathcal{D}_H, av : \mathcal{D}_A)$$
$$\mathsf{varseqn}[E, \$\mathtt{x}], \mathsf{exseqn}[E] : (vs : \mathcal{D}_H, si : \mathcal{D}_H, ti : \mathcal{D}_H, ni : \mathcal{D}_H)$$

The prefixes var and ex are used to distinguish between arguments assigned to variables and arguments representing expressions (although there is no difference in semantics).

The $\mathsf{varseqa}[E, \$\mathtt{x}]$ and $\mathsf{exseqa}[E]$ relations describe atomic members of sequences, the corresponding relations $\mathsf{varseqn}[E, \$\mathtt{x}]$ and $\mathsf{exseqn}[E]$ contain node members of the same sequence (the two portions are interweaved using the sequence identifiers si).

Attribute si is a sequence identifier used to represent order (and duplicity) of values in a sequence. av is a value of an atomic type as defined by the XQuery standard. The attributes ti and ni together form a foreign key linked to the corresponding tree environment. For variable values $\mathsf{varseqn}[E, \$\mathtt{x}]$, the trees are stored in $\mathsf{invenv}[E]$; for expression values, each referenced tree resides either in $\mathsf{invenv}[E]$ or $\mathsf{exenv}[E]$.

7.4 Example

Fig. 7.2 shows an input document related to the example program in Fig. 6.3; note that text nodes are omitted. The input document is given a unique tree identifier d and encoded using Dewey-based relation passed as the environment $\mathsf{invenv}[\mathsf{a}]$ assigned to the root of the main function AST. The value of the global variable `$I` is a singleton reference to the document root. The

following evaluation is bound to the AST node a. (Note that the first parameter λ corresponds to the empty call stack upon the entry to the program.)

$$\kappa(\lambda, \mathsf{varseqn}[\mathsf{a}, \$\mathtt{I}]) = \left(\begin{array}{cccc} \lambda & \lambda & d & \lambda \end{array} \right)$$

$$\kappa(\lambda, \mathsf{invenv}[\mathsf{a}]) = \left(\begin{array}{ccccc} d & \lambda & \mathsf{document} & \lambda & \lambda \\ d & 1 & \mathsf{element} & \text{book} & \lambda \\ d & 1.5 & \mathsf{element} & \text{section} & \lambda \\ d & 1.5.7 & \mathsf{element} & \text{title} & \text{Introduction} \\ d & 1.5.8 & \mathsf{element} & \text{section} & \lambda \\ d & 1.5.8.6 & \mathsf{attribute} & \text{id} & 100 \\ d & 1.5.8.7 & \mathsf{element} & \text{title} & \text{Motivation} \\ d & 1.5.8.9 & \mathsf{element} & \text{p} & \text{In...} \\ d & 1.5.9 & \mathsf{element} & \text{section} & \lambda \\ d & 1.5.9.4 & \mathsf{element} & \text{title} & \text{Approach} \\ d & 1.5.9.8 & \mathsf{element} & \text{p} & \text{We...} \end{array} \right)$$

As the execution of the program proceeds, the function `toc` is called four times in the original XQuery sense. However, in the R-program the corresponding R-function is expanded only three times, due to the bulk evaluation semantics. The calls are located at AST nodes e and n, the latter being recursive. This is expressed in the values assigned to the nodes invv[f] and varseqn[f, $P] during the three R-program runs through these nodes. The former node contains the variable stacks, the latter forms the value of the variable $P. Note that the variable is always a singleton, therefore, its model contains empty sequence identifiers:

$$\begin{aligned} \kappa(\mathsf{e}, \mathsf{invv}[\mathsf{f}]) &= \left(\begin{array}{c} (d(1)) \end{array} \right) \\ \kappa(\mathsf{e.n}, \mathsf{invv}[\mathsf{f}]) &= \left(\begin{array}{c} (d(1))(d(1.5)) \\ (d(1))(d(1.9)) \end{array} \right) \end{aligned}$$

$$\begin{aligned} \kappa(\mathsf{e.n.n}, \mathsf{invv}[\mathsf{f}]) &= \left(\begin{array}{c} (d(1))(d(1.5))(d(1.5.8)) \end{array} \right) \\ \kappa(\mathsf{e}, \mathsf{varseqn}[\mathsf{f}, \$\mathtt{P}]) &= \left(\begin{array}{cccc} (d(1)) & \lambda & d & 1 \end{array} \right) \\ \kappa(\mathsf{e.n}, \mathsf{varseqn}[\mathsf{f}, \$\mathtt{P}]) &= \left(\begin{array}{cccc} (d(1))(d(1.5)) & \lambda & d & 1.5 \\ (d(1))(d(1.9)) & \lambda & d & 1.9 \end{array} \right) \\ \kappa(\mathsf{e.n.n}, \mathsf{varseqn}[\mathsf{f}, \$\mathtt{P}]) &= \left(\begin{array}{cccc} (d(1))(d(1.5))(d(1.5.8)) & \lambda & d & 1.5.8 \end{array} \right) \end{aligned}$$

Encoding of a sequence is shown at the relation exseqn[j] which corresponds to the value of the concatenation operator at node j. This operator

is evaluated three times, returning sequences of two nodes in two cases. In the R-program, the operator is bulk-evaluated twice. Input nodes and constructed nodes are mixed here, which is expressed in the use of different tree identifiers in the third column.

$$\kappa(\mathsf{e}, \mathsf{exseqn}[\mathsf{j}]) = \begin{pmatrix} (d(1))(d(1.5)) & 2.1(d(1.5.7)) & d & 1.5.7 \\ (d(1))(d(1.5)) & 2.2(d(1.5.8)) & (\mathsf{e.n.i})((d(1))(d(1.5))(d(1.5.8))) & \lambda \\ (d(1))(d(1.9)) & 2.1(d(1.9.4)) & d & 1.9.4 \end{pmatrix}$$

$$\kappa(\mathsf{e.n}, \mathsf{exseqn}[\mathsf{j}]) = \begin{pmatrix} (d(1))(d(1.5))(d(1.5.8)) & 1(d(1.5.8.6)) & d & 1.5.8.6 \\ (d(1))(d(1.5))(d(1.5.8)) & 2.1(d(1.5.8.9)) & d & 1.5.8.9 \end{pmatrix}$$

The value associated to the node j becomes the child list of a constructor; therefore, the tree identifiers *ti* are replaced to obey the required copy semantics and node identifiers *ni* are stripped during sub-tree extraction. On the other hand, the sequence identifiers *si* are used as edge labels in the output document – they may be found in the final output of the program shown in Fig. 7.2.

7.5 Document ordering

Document ordering is represented by the following operator applied to a relation x with any signature containing the attributes *ti* and *ni* (the tree and node identifiers). The operator computes the sequence identifier, using the fact that node identifiers are ordered according to the document order and adding the tree identifier to keep trees separated.

$$docorder(x) = \pi[si := \alpha(ti).\alpha(ni)](x)$$

7.6 Effective Boolean value

In the most frequent case, Boolean values are created by the implicit conversion process called *effective Boolean value* [22]. This conversion generates a singleton atomic value of type `xs:boolean`, i.e. either `true` or `false`.

The effective Boolean value may be defined using the operators of relational algebra; however, the defining expression is too complex because the definition of effective Boolean value is complex and irregular (for instance, it is order-sensitive). Fortunately, most XQuery operators do not produce

sequences of mixed atomic and node items; in these cases, the definition of the effective Boolean value is simplified.

To make use of the above-mentioned observation, the conversion must be moved against the data flow and coupled with the previous operator. This move is represented by the introduction of the *EBV mode* which stores the effective Boolean value of the original expression. This mode will be used on the data path between two operators whenever the second operator induces the conversion of its operand to the effective Boolean value.

Note that if the output of an operator is used twice, once with the EBV conversion and once without any conversion, the operator is modified so that it produces both the canonical and the EBV modes.

The EBV mode consists of the following two relations:

$\mathsf{exebvt}[E] : (vs : \mathcal{D}_H)$

$\mathsf{exebvf}[E] : (vs : \mathcal{D}_H)$

$\mathsf{exebvt}[E] = \{(vs) \mid \text{the effective Boolean value in the context } vs \text{ is } \texttt{true}\}$

$\mathsf{exebvf}[E] = \{(vs) \mid \text{the effective Boolean value in the context } vs \text{ is } \texttt{false}\}$

The Appendix A summarizes the most important transcription rules for the canonical and the EBV modes.

7.7 Attribute removal

In many cases, some attributes may be removed from the relations in the transcription. In general, there may be three reasons for the removal:

- The value of the attribute is constant.
- The value of the attribute may be derived from another relation.
- The attribute is not used in subsequent computation.

With the removal of an attribute, the model of an expression or a variable is shrunk to a *submode*. Thus, a submode is generated "automatically" from the base mode. Effective algorithms to determine removable attributes are presented in the chapter 9.

Constant attributes

Many transcription rules assign the empty string to an attribute. The most important case is the attribute *si* – singleton variables are represented in this way. Similarly, constructors generate empty strings to *ni*, representing the root of a tree. In these cases, the empty value may be traced forward and removed, until the attribute is discarded in a projection or merged in a union operator. Removing such attribute simplifies many operations and allows advanced optimizations; e.g., employing the fact that an expression is a singleton.

Derived attributes

When an XQuery expression is independent of a particular invocation (i.e., if it is constant or dependent only on the input documents), the column *vs* may be removed from its relational representation. When necessary, the column may be added using a (Cartesian product) join to the relation $\mathsf{invv}[E]$.

However, the dependency on the relation $\mathsf{invv}[E]$ was important as it kept the resulting R-program controlled (see the section 5.3). Furthermore, we have no constant (other than the empty set) in our relational algebra. Therefore, constants must be represented using a projection (function application) on a singleton relation kept in the controller set of every function. Such a relation is, formally, derived from the relation $\mathsf{invv}[E]$ by the removal of its only attribute. It means that, whenever an expression is evaluated, the reduced version of the $\mathsf{invv}[E]$ relation consists of a single row carrying no attribute. Absence of the row would mean that no evaluation is required – this is exactly the purpose of a controller argument. Since all constants will be dependent on this relation, evaluation of the R-program remains controlled.

Unused attributes

In the transcription rules, often an attribute is not used in the right-hand side of the equations. For instance, all node-set operators ignore the *si* attribute of their operands. In the terms of the relational algebra, there is a projection operator that removes the attribute. The projection operator is distributive over some of the other relational algebra operators, provided the operators do not reference attributes removed by the projection. Thus, the projection operator may be moved against the flow of computation until a barrier formed by a set difference, a natural join or a selection operator that references the removed column. The projection operator may also annihilate with a function application operator that defines the removed column.

In the special case of the attribute *si*, this removal corresponds to the evaluation in *unordered context* as defined in [22].

7.8 Relation removal

Besides relation removal, whole relations may be removed from the generated R-program whenever static analysis detects that they are always empty. This is again a kind of forward-propagating property and a similar algorithm to the detection of constant and derived attributes may be used.

Both exseqn$[E]$ and exseqa$[E]$ relations are frequent candidates for removal since, in most cases, XQuery values are either sequences of nodes or sequences of atomic values (i.e., not mixed). The same applies to the representation of variable values. Furthermore, the exenv$[E]$ relation is empty whenever the expression E (including the functions called here) does not contain constructors.

Relations may also be removed when they are not referenced – this is often the case of exebvf$[E]$ or exebvt$[E]$ because where clauses and quantified expressions use only one of them. The absence of usage is a backward-propagating property similar to the case of unused attributes.

The Fig. 7.3 shows the result of the attribute and relation removal on the canonical representation of the function `employee` from the Fig. 3.1.

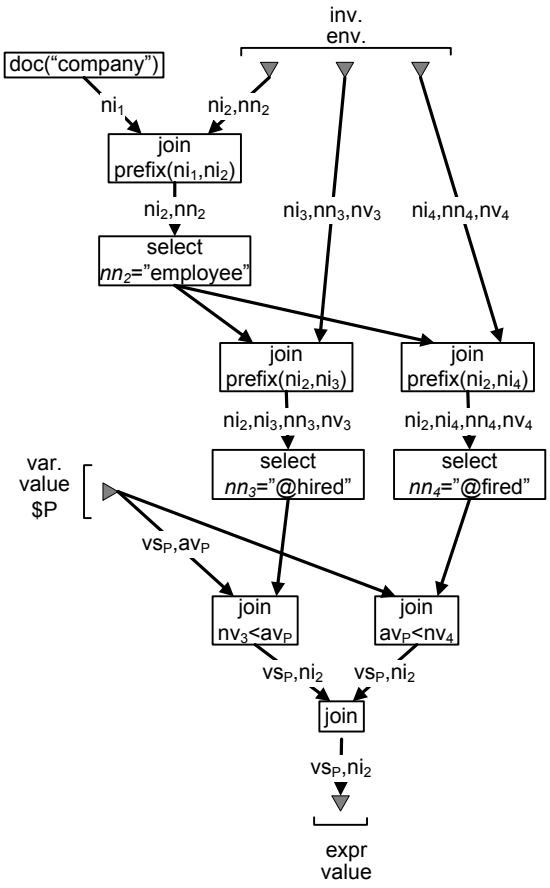


Figure 7.3: Canonical representation of the function `employee`

Chapter 8

Reverted modes

The name *reverted mode* denotes any mode that employs reverted arcs (other than the invc$[E]$, invv$[E]$, and invenv$[E]$ arcs). In this chapter, we will show a particular set of reverted modes designed to replace predicate pushing and join reordering in the context of R-programs. In addition, we will present the *output-driven* mode dedicated to output-document handling.

All the presented modes share a common feature – they are derived from a forward mode (the canonical mode) using a join operation. The general scheme is as follows: In the underlying forward mode, we have a forward arc x. In the derived mode, we are replacing the forward arc with a pair of a reverted arc y and a forward arc z with the following semantics:

$$z = \sigma[P]\ (y\ \bowtie\ x)$$

Thus, the replacement corresponds to pushing a join operation into the expression that computes x. The join may be based on equality (hidden in the natural join operator) as well as on an predicate P, or combining both conditions.

Several scenarios may fit within this general scheme:

- When the primary key of the relation y is empty, the join operator is reduced to a selection controlled by the value of y (which contains at most one tuple). This situation corresponds to pushing an independent predicate into the expression producing x. We will denote this scenario as *independent filtering*.

- When the primary key of y is a subset of the primary key of x, the join operator does not multiply the tuples from x. It may be described as *dependent filtering* by a predicate whose parameters are fed through the relation y and matched using the common part of the primary keys.

- When the primary key of y contains an attribute not present in x, the join operator produces multiple tuples for a single tuple from x. This scenario allows moving a join operation through the boundary of an expression or function: Instead of performing a join ($u \bowtie x$) at its original position, the relation u is sent through the reverted arc y to the place where x is computed. Subsequently, the join operation may be reordered with respect to any joins that produce x. This situation will be called *dependent multiplication*.

- The special case when the primary keys are disjoint will be called *independent multiplication*.

Note that any attributes of y that are not present in x (and not involved in P) will return through z unaffected. Using hierarchical strings, we are able to pack more values to a single attribute; therefore, a single additional attribute can cover all multiplication scenarios. We will use the name *ak* (*additional key*) for such an attribute.

8.1 Atomic-filtering modes

These modes are applied to the $\mathsf{exseqa}[E]$ relation of the canonical mode, in particular, to its *av* attribute. This is the attribute that carries the atomic value of an XQuery expression and XQuery predicates may be applied to it. Although there is a vast number of predicates applicable, only those predicates that may be useful in indexed access are considered. In this section, we will show modes based on two kinds of predicates – the equality $av = x$ and the bounding interval $r \leq av \leq s$.

Based on the discussion in the previous section, we can define the following eight reverted modes. Each mode consist of a reverted arc relation, a forward arc relation, and an equation that defines the forward relation based on the reverted relation and the underlying $\mathsf{exseqa}[E]$.

- *independent equality filtering*
  $\mathsf{exiefr}[E] : (av : \mathcal{D}_A)$
  $\mathsf{exiefa}[E] : (vs : \mathcal{D}_H, si : \mathcal{D}_H, av : \mathcal{D}_A)$
  $\mathsf{exiefa}[E] = (\mathsf{exiefr}[E] \bowtie \mathsf{exseqa}[E])$

- *independent interval filtering*
  $\mathsf{exiifr}[E] : (r : \mathcal{D}_A, s : \mathcal{D}_A)$
  $\mathsf{exiifa}[E] : (vs : \mathcal{D}_H, si : \mathcal{D}_H, av : \mathcal{D}_A)$
  $\mathsf{exiifa}[E] = \delta\pi[\backslash r, s]\ \sigma[r \leq av \leq s]\ (\mathsf{exiifr}[E] \bowtie \mathsf{exseqa}[E])$

- *dependent equality filtering*
  $\mathsf{exdefr}[E] : (vs : \mathcal{D}_H, av : \mathcal{D}_A)$
  $\mathsf{exdefa}[E] : (vs : \mathcal{D}_H, si : \mathcal{D}_H, av : \mathcal{D}_A)$
  $\mathsf{exdefa}[E] = (\mathsf{exdefr}[E] \bowtie \mathsf{exseqa}[E])$

- *dependent interval filtering*
  $\mathsf{exdifr}[E] : (vs : \mathcal{D}_H, r : \mathcal{D}_A, s : \mathcal{D}_A)$
  $\mathsf{exdifa}[E] : (vs : \mathcal{D}_H, si : \mathcal{D}_H, av : \mathcal{D}_A)$
  $\mathsf{exdifa}[E] = \delta\pi[\backslash r, s]\ \sigma[r \leq av \leq s]\ (\mathsf{exdifr}[E] \bowtie \mathsf{exseqa}[E])$

- *independent equality multiplication*
  $\mathsf{exiemr}[E] : (ak : \mathcal{D}_H, av : \mathcal{D}_A)$
  $\mathsf{exiema}[E] : (ak : \mathcal{D}_H, vs : \mathcal{D}_H, si : \mathcal{D}_H, av : \mathcal{D}_A)$
  $\mathsf{exiema}[E] = (\mathsf{exiemr}[E] \bowtie \mathsf{exseqa}[E])$

- *independent interval multiplication*
  $\mathsf{exiimr}[E] : (ak : \mathcal{D}_H, r : \mathcal{D}_A, s : \mathcal{D}_A)$
  $\mathsf{exiima}[E] : (ak : \mathcal{D}_H, vs : \mathcal{D}_H, si : \mathcal{D}_H, av : \mathcal{D}_A)$
  $\mathsf{exiima}[E] = \delta\pi[\backslash r, s]\ \sigma[r \leq av \leq s]\ (\mathsf{exiimr}[E] \bowtie \mathsf{exseqa}[E])$

- *dependent equality multiplication*
  $\mathsf{exdemr}[E] : (ak : \mathcal{D}_H, vs : \mathcal{D}_H, av : \mathcal{D}_A)$
  $\mathsf{exdema}[E] : (ak : \mathcal{D}_H, vs : \mathcal{D}_H, si : \mathcal{D}_H, av : \mathcal{D}_A)$
  $\mathsf{exdema}[E] = (\mathsf{exdemr}[E] \bowtie \mathsf{exseqa}[E])$

- *dependent interval multiplication*
  $\mathsf{exdimr}[E] : (ak : \mathcal{D}_H, vs : \mathcal{D}_H, r : \mathcal{D}_A, s : \mathcal{D}_A)$
  $\mathsf{exdima}[E] : (ak : \mathcal{D}_H, vs : \mathcal{D}_H, si : \mathcal{D}_H, av : \mathcal{D}_A)$
  $\mathsf{exdima}[E] = \delta\pi[\backslash r, s]\ \sigma[r \leq av \leq s]\ (\mathsf{exdimr}[E] \bowtie \mathsf{exseqa}[E])$

Other modes may be derived similarly, for instance *lower-bound* or *upper-bound* modes.

Reverted modes require special handling of variable propagation, applying union on their reverted arcs as shown in the Fig. 6.7 for the general case of a binary operator. Furthermore, rules defining a new variable contain joins as shown in the Fig. 8.1 and 8.2.

8.2 Node-filtering modes

Similarly to atomic filtering, node sequences may be filtered using the important properties of the nodes, namely the kind nk, the name nn, and the

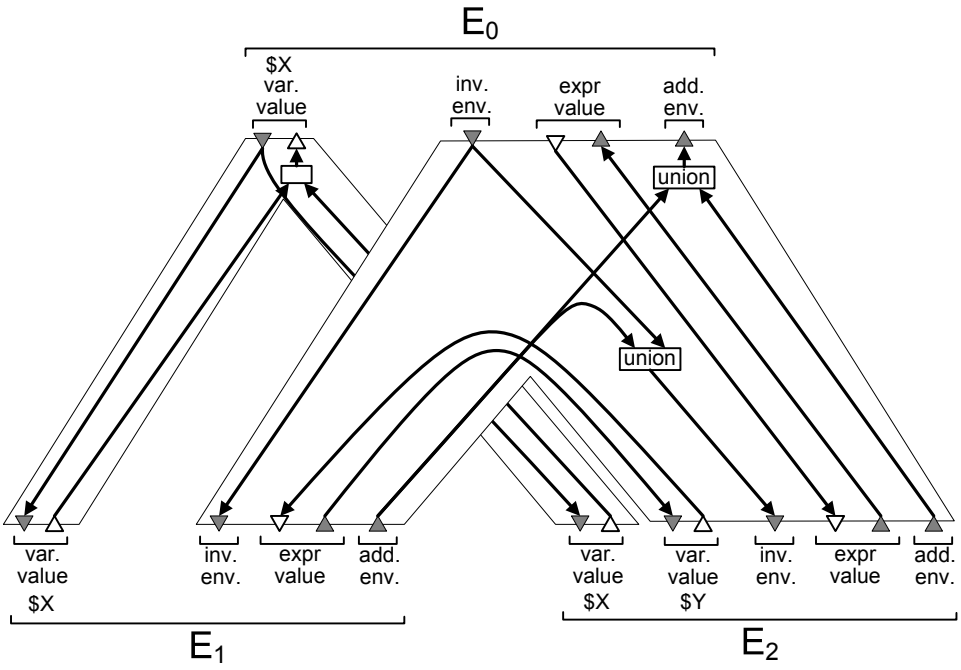


Figure 8.1: A reverted-flow rule E_0 `::= let $Y :=` E_1 `return` E_2

typed value nv. Any subset of the three attributes may be bound by equality like $nn = x$, the typed value may also be constrained by the bounding interval $r \leq nv \leq s$. These combinations generate 11 ways of filtering.

Furthermore, each filtering may be independent or dependent and may become a multiplication. Consequently, there are 44 modes applicable.

Fig. 8.3 shows the representation of the function `employee` from the Fig. 3.1, using independent node filtering on the invocation environment and independent atomic multiplication on the variable `$P`.

8.3 Structural-filtering modes

Node sequences may also be filtered with respect to the node identifier ni, using its properties as a Dewey number. The available filters are $q = rtrim(ni)$, $prefix(ni, q)$, and $prefix(q, ni)$, corresponding to the `child`, `ancestor`, and `descendant` axis navigation with respect to the node q (see the Table A.1). Based on these three types of filters, 12 reverted modes are defined, carrying the node identifier q in their reverted arc relation.

Applying structural-filtering modes, a group of structural join operators

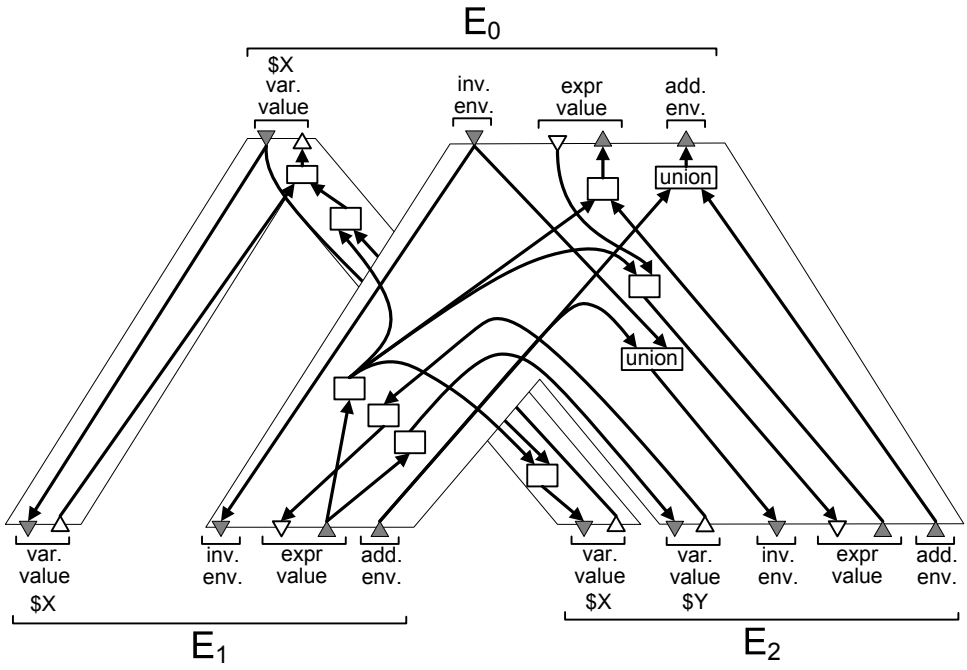


Figure 8.2: A reverted-flow rule E_0 `::= for $Y in` E_1 `return` E_2

may be interconnected so that they together emulate the behavior of a path or twig join (see the Sec. 2.7). Note that only the basic versions of joins may be emulated; the emulation of the skipping versions (see [47]) would require cycles in the R-program.

8.4 Output-driven mode

Compared to the complexity of XPath evaluation, the generation of the output tree seems straightforward. XQuery constructors create the output document in bottom-up manner; moreover, the formal definition of XQuery semantics suggests that the children of the new nodes are created from the source node list by copying (including their sub-trees). This copying is usually avoided by the use of shared pointers and reference counting [31]; however, these methods are poorly applicable in parallel environments.

To avoid the obstacles of bottom-up construction, the *output-driven mode* is devised. Using this mode, output documents are effectively constructed in top-down manner, giving each node its final position in the moment of its construction. This arrangement allows bypassing the construction of partial

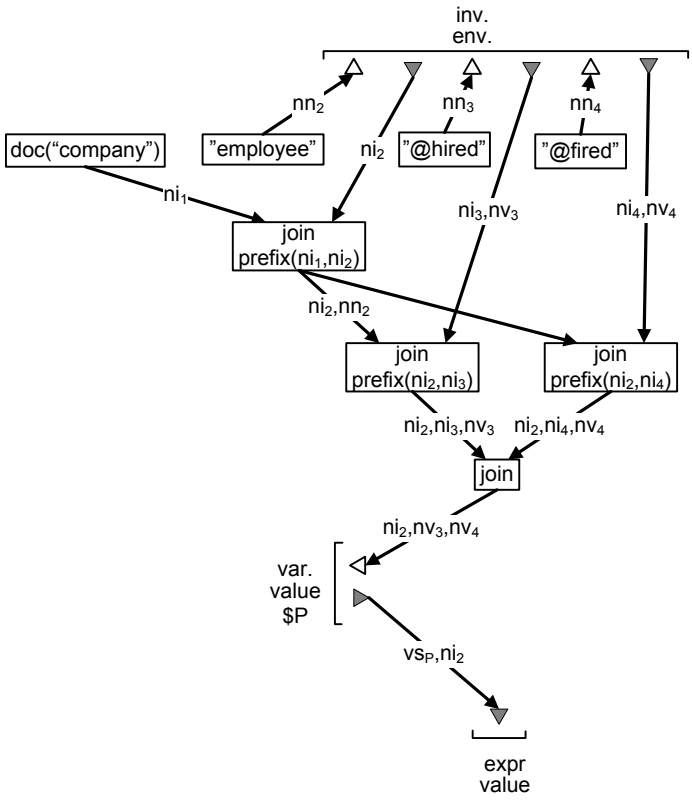


Figure 8.3: Reverted representation of the function `employee`

sub-trees and, in the case when the output of the query is stored into a database, building the output document directly in its storage location.

In most cases, constructed nodes and trees are not navigated using XPath axes or combined using node-set operations; they are just propagated to other constructors or to the final output of the program. (Note that navigation in temporary trees was prohibited in XSLT 1.0., therefore, it is still very rare practice in XSLT 2.0 and XQuery.) In such cases, the expressions forming a XQuery program may be statically divided into two classes: Expressions returning atomic values and input document nodes and expressions returning constructed nodes.

Under the assumption that output trees are not navigated, only the following operators may be applied to the constructed node sequences:

- Concatenation (`,`) operator
- `for`-expression
- `let`-expression
- Node constructor
- Input-to-output tree conversion (implicit)
- Function call

These operators can perform only a limited degree of manipulation on the sequences. As a result, each node is just placed somewhere in the output document, without any modification to its contents. Using `where` clause, nodes may be discarded; using `let`-expression or function arguments, nodes may be copied. Thus, the total effect on a constructed node may be represented by a set of *placeholders* that represent the positions in the output document where the node is placed (copied).

Our method is based on partial reversal of the direction of data flow in an XQuery program. In the synthetic part of the program that generates the output document, the canonical bottom-up evaluation is replaced by top-down distribution of placeholders. These placeholders give advice to the constructor operators on where the constructed nodes will reside in the final output document.

Output node identifiers are produced during the reversed evaluation of the XQuery program. Starting at the main expression, (*partial*) placeholders are propagated through the program; at each constructor, partial placeholders are *finalized* to produce complete output node identifiers. These identifiers,

together with the properties of the constructed node, are stored into the output database.

We assume that the XQuery was already statically analyzed to determine which expressions are used as output node sequences, in other words, where the output-driven mode shall be applied (see the Chapter 9).

Those expressions will be evaluated in two phases. In the first phase, the information is passed in reversed direction against the original data-flow, using the placeholder sets. In the second phase, the output is simply collected by (disjoint) union operators. In an implementation, each constructor may directly write its output to storage instead of union-collecting the output along function calls.

A set of descriptors is expressed using the following relation:

$$\mathsf{exdesc}[E] : (vs : \mathcal{D}_H, ni : \mathcal{D}_H, si : \mathcal{D}_H)$$

The *vs* attribute corresponds to the variable stack and determines the context of the evaluation in the same manner as in the forward modes. The remaining hierarchical identifiers form the partial placeholder $ni.\alpha(si)$.

The output of the XQuery program is collected from all constructor operators throughout the program. Every contribution is modeled as the relation:

$$\mathsf{exout}[E] : (oi : \mathcal{D}_H, nk : \mathcal{D}_K, nn : \mathcal{D}_N, nv : \mathcal{D}_A)$$

The *oi* attribute is the output node identifier, *nk*, *nn*, and *nv* attributes represent node kind, name and value as required by the XML document model.

With respect to the canonical mode, the output-driven mode relations are created so that the following equation holds:

$$\begin{aligned} &\mathit{normalize}(\mathsf{exout}[E]) = \mathit{glue}(\mathsf{exdesc}[E], \\ &\qquad \mathit{mkforest}(\mathsf{exseqn}[E], \mathsf{invenv}[E], \mathsf{exenv}[E], \mathsf{exseqa}[E])) \end{aligned}$$

$$\mathit{glue}(x, y) = \pi[oi := ni.\alpha(si.r).s]\ (x \ \bowtie\ \mathit{lunpack}(y))$$

$$\mathit{lunpack}(x) = \delta\pi[\backslash ni]\ \pi[r := \mathit{first}(ni), s := \mathit{ltrim}(ni)]\ (x)$$

See the Section 7 for the definition of the functions *normalize* and *mkforest*.

The function $\mathit{glue}(x, y)$ performs a join operation on its arguments; thus, the output-driven mode is similar to the concept of reverted mode defined by a join operation presented in the introduction to this chapter. The original join-based concept is complicated in two aspects – first, the join is applied on a function of four arcs instead of one forward arc; second, the function

normalize is applied on the left-hand-side of the equation. The latter fact corresponds to the fact that the $\mathsf{exout}[E]$ arcs carry unnormalized tree fragments and the normalization is delayed as described below.

The join operation in $glue(x, y)$ is driven by the equality of the attribute vs; thus, this is a special case of dependent multiplication as defined in the beginning of this chapter. Static analysis may determine that the value of the remaining attributes of the $\mathsf{exdesc}[E]$ relation is independent of the value of vs; in such cases, the vs attribute may be removed, resulting to an independent-multiplication scenario where $\mathsf{exdesc}[E]$ is replaced by the following relation:

$$\mathsf{exdesci}[E] : (ni : \mathcal{D}_H, si : \mathcal{D}_H)$$

We will show that the set of placeholders may be computed in the reversed direction of the constructed node flow. The computation begins at the main expression E_{main} of the program, with a single placeholder pointed at the root of the output document:

Reverted core rules

$$\mathsf{exdesc}[E_{main}] := (vs := \lambda, ni := \lambda, si := \lambda)$$

Core rules

$$\mathsf{exenv}[E_{main}] := normalize(\pi[ti := \lambda]\ \pi[ni/oi]\ \mathsf{exout}[E_{main}])$$

The *normalize* function performs the normalization required by the semantics of the constructor operation – adjacent string values are separated by spaces, concatenated, and converted to a text node; then, adjacent text nodes are merged. In the output-driven mode, the normalization is delayed to the very end of the query execution. To allow this rearrangement, an additional node kind – string node – must have been introduced and the normalization must have been generalized to trees instead of the original sequences. We have already defined the generalized *normalize* function in the definition of constructor transcription rules for the canonical mode.

An implementation that stores the output of the query in a database may decide to store the unnormalized version of output, provided it is aware of the consequences in indexing and further querying.

The reversed evaluation of the applicable XQuery operators is described by the following equations.

Concatenation

Syntax

$E_0 ::= E_1 \ , \ E_2$

Reverted core rules

$\mathsf{exdesc}[E_1] := \pi[si/u]\ \delta\pi[\backslash si]\ \pi[u := s.1]\ (\mathsf{exdesc}[E_0])$

$\mathsf{exdesc}[E_2] := \pi[si/u]\ \delta\pi[\backslash si]\ \pi[u := s.2]\ (\mathsf{exdesc}[E_0])$

Core rules

$\mathsf{exout}[E_0] := (\mathsf{exout}[E_1]\ \cup\ \mathsf{exout}[E_2])$

Node construction

$E_0 ::= \texttt{<e>}\{\ E_1\ \}\texttt{</e>}$

$$\begin{aligned}\mathsf{exdesc}[E_1] &= \delta\pi[\backslash p, u]\ \pi[si := \lambda, ni := p.\alpha(u)]\ \pi[p/ni, u/si]\ (\mathsf{exdesc}[E_0])\\ \mathsf{exout}[E_0] &= \pi[nk := \texttt{element}, nn := \mathsf{e}, nv := \lambda]\ \delta\pi[\backslash ni, si]\\ &\qquad \pi[oi := ni.\alpha(si)]\ (\mathsf{exdesc}[E_0])\\ &\qquad \cup\, \mathsf{exout}[E_1]\end{aligned}$$

For expression

$E_0 ::= \texttt{for \$y in } E_1 \texttt{ return } E_2$

$$\begin{aligned}\mathsf{exdesc}[E_2] &= \delta\pi[\backslash j, u, w, av]\ \pi[vs := j.\alpha(av), si := u.\alpha(w)]\\ &\qquad (\pi[j/vs, u/si]\ (\mathsf{exdesc}[E_0])\ \bowtie\ \pi[w/si]\ (\mathsf{exseqa}[E_1]))\\ \mathsf{exout}[E_0] &= \mathsf{exout}[E_2]\end{aligned}$$

Order-by clause

$E_0 ::= \texttt{for \$y in } E_1 \texttt{ order by } E_2 \texttt{ return } E_3$

$$\begin{aligned}\mathsf{exdesc}[E_3] = \delta\pi[\backslash j, u, w, av]\ &\pi[vs := j.\alpha(av), si := u.\alpha(y).\alpha(w)]\\ &(\pi[j/vs, u/si]\ (\mathsf{exdesc}[E_0])\ \bowtie\\ &\pi[w/si]\ (\mathsf{exseqa}[E_1])\ \bowtie\\ &\pi[j/vs, y/av]\ \delta\pi[\backslash si]\ (\mathsf{exseqa}[E_2]))\end{aligned}$$

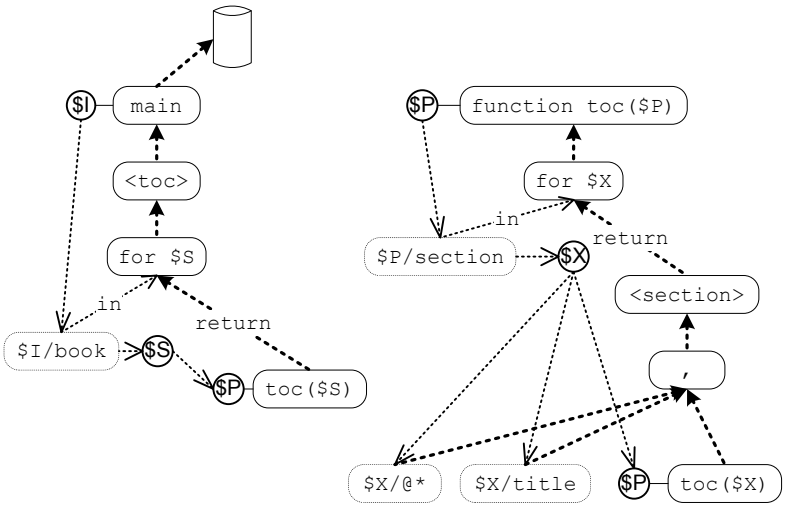


Figure 8.4: Query 1 – Standard data flow

Example

Fig. 8.4 shows the abstract syntax trees corresponding to the program in Fig. 6.2 with arrows depicting the propagation of information. The thin dashed arrows show the propagation of input tree nodes, the thick dashed arrows carry the constructed nodes. Note that the `$X/@*` and `$X/title` expressions produce input nodes, however, their corresponding sub-trees are copied into the output document. This behavior may be regarded as an implicit conversion – by definition, it shall take place just before the `<section>` constructor; however, static analysis may move the conversion operation down to the XPath expressions as shown in the figure.

The reversed flow of placeholders is shown in Fig. 8.5, using solid arcs. The side effects of constructors are shown as thick dashed arrows. In the formal model of the output-driven mode, the encodings of the constructed nodes are collected using union operators; in reality, it may be implemented by an output operation as suggested in the picture.

Implicit conversions attached to the `$X/@*` and `$X/title` expressions were transformed to side effects: Such expressions perform a Cartesian product of their new placeholder argument with the input nodes returned by their XPath expression. Each of these input nodes is (together with its sub-tree) copied into every place denoted by the placeholder set.

The computation starts with the empty *vs*-independent placeholder ($ni := \lambda, si := \lambda$). The empty placeholder reaches the `<toc>` constructor which pro-

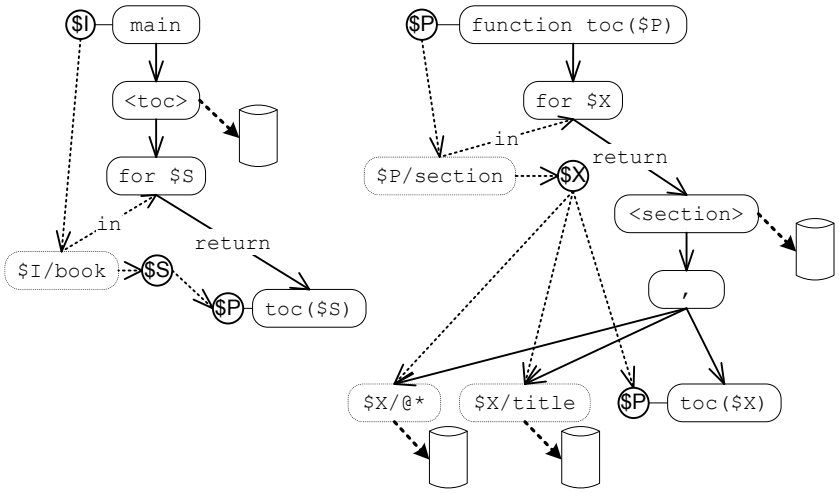


Figure 8.5: Query 1 – Reversed data flow

duces a node in the form of the tuple ($oi := \alpha(\lambda), nk :=$ `element`, $nn :=$ `toc`, $nv := \lambda$). The oi value encodes the fact that the node will be placed in the depth 1 as the only child of the document node.

The constructor operator then passes the placeholder ($ni := \alpha(\lambda), si := \lambda$) to its sub-expression, carrying the information that the nodes created by the sub-expression will become children of the node identified by $\alpha(\lambda)$.

A `for`-expression iterates through a sequence whose order is encoded in the sequence identifiers. (In our example, this is a sequence of nodes serialized in the document-order; therefore, their node identifiers determine the ordering.) In each iteration, the `for`-expression produces a new placeholder by appending the sequence identifier at the end of the si attribute of the placeholder; in the example, the `for $S` expression creates the placeholder ($vs := \alpha(si_S), ni := \alpha(\lambda), si := si_S$) for each sequence identifier si_S generated by the `in`-expression. This placeholder is passed down to the function `toc`.

The expression `for $X` iterates through another expression, generating placeholders like ($vs := \alpha(si_S).\alpha(si_X), ni := \alpha(\lambda), si := si_S.si_X$) by appending sequence identifiers si_X. Therefore, the `<section>` constructor produces a set of nodes encoded by tuples

$$(oi := \alpha(\lambda).\alpha(si_S.si_X), nk := \texttt{element}, nn := \texttt{section}, nv := \lambda)$$

The concatenation (`,`) operator is implemented using fixed labels $\{1, 2, 3\}$ assigned to its three operands. Thus, the recursive call to the function `toc`

receives a set of placeholders in the form

$$(vs := \alpha(si_S).\alpha(si_X), ni := \alpha(\lambda).\alpha(si_S.si_X), si := 3)$$

Further nested calls receive the sets like

$$(vs := \alpha(si_S).\alpha(si_X).\alpha(si_{X2}), ni := \alpha(\lambda).\alpha(si_S.si_X).\alpha(3.si_{X2}), si := 3)$$

and so on.

Applicability of the output-driven mode

The output-driven approach may be functional if the following requirements are satisfied:

- The target database uses a kind of identifiers to link the individual elements together. (Pointer-based implementations are disqualified.)
- These identifiers must be stable and independent on the presence of sibling elements. For instance, identifier schemes based on sequential (1,2,3,...) numbering of document nodes are not suitable. Fortunately, schemes designed to support updates usually satisfy this requirement.
- The target database must tolerate temporal violation of referential integrity during the process of document generation. (For instance, children representation may be generated before their parents.) Transaction-based isolation can solve this problem; however, the cost of this approach in a distributed environment is not negligible. Higher-level management of incomplete output documents may perform better than a transaction-based approach.
- The target database must offer an interface capable to accept individual document elements and identifiers from the XQuery/XSLT engine.

It seems that many contemporary XML storage systems, either relational (for instance [68]) or native, can support the first three requirements either immediately or with little changes. The main blocker is the fourth requirement, since the storage systems are usually built as black-boxes not capable to accept their internal identifiers from the outside. Nevertheless, advanced XML storage systems are often tightly coupled with an XPath engine on the retrieval side; therefore, the integration on the document-generation side may be realizable.

Materialized XML-views are one of the areas where such an approach may be used. Indeed, methods using identifier schemes satisfying the above-mentioned requirements were already presented [26, 25]; however, they are focused on view updates, therefore solving a different problem than our method described in this paper. In [36], related techniques were used in query reformulation above XSLT Views.

The main advantage of the presented method is the absence of in-memory structures representing the output document or its parts during the execution. This fact reduces the memory footprint of the execution. On the other hand, the amount of information generated during the execution is greater than in memory-based approaches, due to the fact that each generated node carries the Dewey identifier of the absolute location of the node. Therefore, the proposed approach may be advantageous under the following circumstances:

- Large output documents that would hardly fit into memory. Traditionally, streaming XQuery/XSLT processors are used in these cases; however, our technique allows to process programs that do not fit into streamability requirements associated with various streaming methods (see, for instance [23]).

- Shallow output documents. In this case, the absolute Dewey identifiers of the generated nodes are relatively short and the above-mentioned overhead connected to their generation is lower. Studies [63] show that real XML data are rather shallow.

- A distributed environment, where the maintenance of in-memory trees is costly.

- XML messaging. Our labeling scheme may also be used as an alternative to serialized XML in tightly-coupled systems (for instance XRPC [87]) where latency is more important than bandwidth.

Chapter 9

Static analysis

Before transcription, a transcription rule (or a set of rules) must be selected for each AST node of each function. These transcription rules induce mode assignment for each AST node. The selected rule/mode assignment must satisfy these requirements:

- Each selected mode must be valid in the given context. This rule applies to transcription rules applicable under a particular condition, e.g. singleton expressions, node-only sequences etc.

- The selected mode must carry enough information to the consumers. More precisely, the set of output arcs at a particular rule and node must be a super-set of the input arcs required by the selected transcription rules in the neighborhood.

- The R-program generated using the selected transcription rules must be acyclic.

More than one mode selection may fulfil these conditions; therefore, the mode selection phase uses a priority scheme to favorize the modes expected to be less costly. This prioritization forms a kind of rule-based optimization. This prioritization also ensures that the resulting R-program will be non-redundant (in the sense of the Definition 4).

Note that the determination of controller sets is also necessary for an executable R-program. However, the controller sets are determined statically and independently in each transcription rule; therefore, the mode-selection algorithms are not required to support controller sets.

9.1 Phases

The three above-mentioned conditions lead to the following three phases of the static analysis:

- *Forward propagation* determines certain properties of the values at each AST node and, thus, the applicability of transcription rules.

- *Cycle removal* phase detects cycles in the generated R-program and gradually removes them until the program is acyclic. This phase is driven by the priority rules, based on the observation that the canonical transcription (assigned the lowest priority) is always acyclic.

- *Backward propagation* determines the set of arcs (i.e. the mode) required at each AST node by the neighboring nodes that consume its outputs. This in turn removes redundancy and keeps the greatest-priority transcription rule from the available rules.

All the three phases work as if the source program (in the AST forest model) were already transcribed to an R-program. Nevertheless, the algorithms are based on traversals through the original AST forest; thus, the transcription is, at this moment, only virtual.

In the forward propagation phase, the canonical and Boolean transcription rules are virtually applied and the properties of the resulting R-program are examined. Based on the forward propagation phase, a set of applicable transcription rules is determined for each AST node. The following phases act as if all applicable transcription rules were applied at once, producing a redundant R-program. The cycle removal phase eliminates the transcription rules that would break the acyclicity condition. The redundancy is completely removed after the backward propagation phase.

Each of the three phases is composed of two steps – *dependency analysis* and a specific dependency application step. The dependency analysis step follows the same algorithm in all the three phases; however, the meaning of the word "dependency" is different in these phases.

In all phases, we are determining certain kind of dependency between the *inherited* and the *synthesized* arcs at each AST node. Inherited arcs consist of the forward arcs associated to the representation of visible variables and of the reversed arcs of the expression value; synthesized arcs are the forward arcs of the expression value and the reversed arcs of the visible variables. Arcs of all modes together are considered.

9.2 Observed dependency relations

Each of the three phases determines a different kind of dependency:

- The forward phase determines the dependency of cardinality on the scale *zero/one/more*. Thus, the output of the first phase is a set of observations like "if the inherited arc a^I has its cardinality less or equal to c^I, then the synthesized arc a^S will have a cardinality at most c^S". Such an observation bound to an AST node E will be denoted $\mathsf{carddep}_{c^I,c^S}[E, a^I, a^S]$ for $c^I, c^S \in \{0, 1\}$.

- The cycle removal phase determines dependency in the sense of the Definition 6, i.e. that the value of a synthesized arc a^S must be computed after the value of an inherited arc a^I, denoted $\mathsf{dep}[E, a^I, a^S]$. When the dependency is determined, cycles are detected and broken by the removal of a transcription rule as shown below.

- The backward phase determines the requirement for an inherited arc based on the requirement for an synthesized arc. It determines statements $\mathsf{aareq}[E, a^I, a^S]$ with the meaning "the inherited arc a^I is required by the most prioritized available transcription rule that can produce the synthesized arc a^S". At the same time, the backward phase also determines dependency at the level of relation attributes: the statement $\mathsf{ccreq}_{c^I,c^S}[E, a^I, a^S]$ means "the relational attribute c^I of the inherited arc a^I is required by the most prioritized available transcription rule that can produce the relational attribute c^S of the synthesized arc a^S". Mixed statements $\mathsf{acreq}_{c^S}[E, a^I]$ and $\mathsf{careq}_{c^I}[E, a^I]$ are also determined to show the dependency between particular attributes and whole relations in both directions.

All the above-mentioned symbols may be generalized as $\phi[E, a^I, a^S]$ where:

- $\phi \in \{\mathsf{carddep}_{0,0}, \mathsf{carddep}_{0,1}, \mathsf{carddep}_{1,0}, \mathsf{carddep}_{1,1}\}$ for the forward phase
- $\phi \in \{\mathsf{dep}\}$ for the cycle detection
- $\phi \in \{\mathsf{aareq}\} \cup \{\mathsf{acreq}_{c^S} \mid c^S \in \mathsf{AttrNm}\} \cup \{\mathsf{careq}_{c^I} \mid c^I \in \mathsf{AttrNm}\} \cup \{\mathsf{ccreq}_{c^I,c^S} \mid c^I, c^S \in \mathsf{AttrNm}\}$ for the backward phase

Thus, for each pair $\langle a^I, a^S \rangle$, $\phi[E, a^I, a^S]$ is a matrix of Boolean values. The size of the matrix is $2 \cdot 3$ for the forward phase, $1 \cdot 1$ for the cycle detection, and $(1 + |\mathsf{sch}(a^I)|) \cdot (1 + |\mathsf{sch}(a^S)|)$ for the backward phase.

Let us fix to a given vocabulary of expression arc names $\mathsf{ExArcNm}$ and variable arc names $\mathsf{VarArcNm}$. Both vocabularies are divided into forward and reverse arcs:

$$\mathsf{ExArcNm} = \mathsf{ExArcNm}^{\mathsf{F}} \cup \mathsf{ExArcNm}^{\mathsf{R}}$$

$$\mathsf{VarArcNm} = \mathsf{VarArcNm}^{\mathsf{F}} \cup \mathsf{VarArcNm}^{\mathsf{R}}$$

$$\mathsf{ExArcNm}^{\mathsf{F}} = \{e_1^F, e_2^F, \ldots, e_{n_e^F}^F\}$$

$$\mathsf{ExArcNm}^{\mathsf{R}} = \{e_1^R, e_2^R, \ldots, e_{n_e^R}^R\}$$

$$\mathsf{VarArcNm}^{\mathsf{F}} = \{v_1^F, v_2^F, \ldots, v_{n_v^F}^F\}$$

$$\mathsf{VarArcNm}^{\mathsf{R}} = \{v_1^R, v_2^R, \ldots, v_{n_v^R}^R\}$$

For the modes defined in the chapters 7 and 8, the sets are as follows:

$\mathsf{ExArcNm}^{\mathsf{F}} = \{\mathsf{exseqa}, \mathsf{exseqa}, \mathsf{exseqn}, \mathsf{exenv}, \mathsf{exout},$
$\mathsf{exiefa}, \mathsf{exiifa}, \mathsf{exdefa}, \mathsf{exdifa}, \mathsf{exiema}, \mathsf{exiima}, \mathsf{exdema}, \mathsf{exdima}\}$

$\mathsf{ExArcNm}^{\mathsf{R}} = \{\mathsf{invc}, \mathsf{invv}, \mathsf{invenv}, \mathsf{exdesc},$
$\mathsf{exiefr}, \mathsf{exiifr}, \mathsf{exdefr}, \mathsf{exdifr}, \mathsf{exiemr}, \mathsf{exiimr}, \mathsf{exdemr}, \mathsf{exdimr}\}$

$\mathsf{VarArcNm}^{\mathsf{F}} = \{\mathsf{varseqa}, \mathsf{varseqn}, \mathsf{varout},$
$\mathsf{variefa}, \mathsf{variifa}, \mathsf{vardefa}, \mathsf{vardifa}, \mathsf{variema}, \mathsf{variima}, \mathsf{vardema}, \mathsf{vardima}\}$

$\mathsf{VarArcNm}^{\mathsf{R}} = \{\mathsf{vardesc},$
$\mathsf{variefr}, \mathsf{variifr}, \mathsf{vardefr}, \mathsf{vardifr}, \mathsf{variemr}, \mathsf{variimr}, \mathsf{vardemr}, \mathsf{vardimr}\}$

9.3 AST node behavior

For an AST node E with the set $\mathsf{vars}(E)$ of visible variables, there are:

- the inherited arcs $e_i^R[E]$ for $i \in \{1, 2, \ldots, n_e^R\}$
- the inherited arcs $v_i^F[E, \mathtt{\$x}]$ for $i \in \{1, 2, \ldots, n_v^F\}$ and $\mathtt{\$x} \in \mathsf{vars}(E)$
- the synthesized arcs $e_j^F[E]$ for $j \in \{1, 2, \ldots, n_e^F\}$
- the synthesized arcs $v_j^R[E, \mathtt{\$y}]$ for $j \in \{1, 2, \ldots, n_v^R\}$ and $\mathtt{\$y} \in \mathsf{vars}(E)$

Thus, for a generalized dependency symbol ϕ, the dependency relations represented by the following set of *dependency matrices*:

$$\psi^{EE}[E] = \left(\phi[E, e_i^R[E], e_j^F[E]]\right)$$

$$\psi^{EV}[E, \$\mathtt{y}] = (\phi[E, e_i^R[E], v_j^R[E, \$\mathtt{y}]]) \text{ for each } \$\mathtt{y} \in \mathsf{vars}(E)$$

$$\psi^{VE}[E, \$\mathtt{x}] = (\phi[E, v_i^F[E, \$\mathtt{x}], e_j^F[E]]) \text{ for each } \$\mathtt{x} \in \mathsf{vars}(E)$$

$$\psi^{VV}[E, \$\mathtt{x}, \$\mathtt{y}] = (\phi[E, v_i^F[E, \$\mathtt{x}], v_j^R[E, \$\mathtt{y}]]) \text{ for each } \$\mathtt{x}, \$\mathtt{y} \in \mathsf{vars}(E)$$

Note that the symbols ϕ were already matrices, therefore, the matrix $\psi^{EE}[E]$ contains $n_e^R \cdot n_e^F \cdot k$ Boolean values where k is the size of the ϕ matrix (which is different for each phase); similarly, $n_e^R \cdot n_v^R \cdot k$ is the size of each $\psi^{EV}[E, \$\mathtt{y}]$ matrix, $n_v^F \cdot n_e^F \cdot k$ for each $\psi^{VE}[E, \$\mathtt{x}]$, and $n_v^F \cdot n_e^F \cdot k$ for each $\psi^{VV}[E, \$\mathtt{x}, \$\mathtt{y}]$.

These matrices may be arranged into a *compound* dependency matrix

$$\psi[E] = \begin{pmatrix} \psi^{EE}[E] & \psi^{EV}[E, \$\mathtt{y}_1] & \dots & \psi^{EV}[E, \$\mathtt{y}_m] \\ \psi^{VE}[E, \$\mathtt{x}_1] & \psi^{VV}[E, \$\mathtt{x}_1, \$\mathtt{y}_1] & \dots & \psi^{VV}[E, \$\mathtt{x}_1, \$\mathtt{y}_m] \\ \vdots & \vdots & & \vdots \\ \psi^{VE}[E, \$\mathtt{x}_m] & \psi^{VV}[E, \$\mathtt{x}_m, \$\mathtt{y}_1] & \dots & \psi^{VV}[E, \$\mathtt{x}_m, \$\mathtt{y}_m] \end{pmatrix}$$

The compound matrix has the total size

$$s(E) = (n_e^R + m(E) \cdot n_v^F) \cdot (n_e^F + m(E) \cdot n_v^R) \cdot k$$

where $m(E)$ is the number of visible variables for the node E.

The compound matrix completely determines the behavior of the subexpression rooted at the node E with respect to the dependency properties observed by the analysis phase.

9.4 Rule behavior

Given a transcription rule R applied at an AST node E_0 with its children $E_1, \dots, E_n$, the compound dependency matrix for the parent is a function of compound dependency matrices of the children:

$$\psi[E_0] = \Phi_R(\psi[E_1], \dots, \psi[E_n])$$

.

The function Φ_R determines the propagation of dependencies in the given rule; therefore, it is always a monotone function, i.e.

$$x_1 \leq y_1 \wedge \ldots \wedge x_n \leq y_n \Rightarrow \Phi_R(x_1, \ldots, x_n) \leq \Phi_R(y_1, \ldots, y_n)$$

where the partial ordering $\leq$ is defined on Boolean matrices as

$$x \leq y \Leftrightarrow (\forall i, j)(x_{i,j} \Rightarrow y_{i,j})$$

Note that the function Φ_R may depend on the result of the previous phase of the analysis. This dependency is expressed as a variability in the selection of the transcription rule R for a given grammar rule.

In the following text, we will also assume that the manipulation with the visible variables is already incorporated in the rule R.

Each Φ_R function is defined in terms of matrices A_R, B_R, Γ_R, and Δ_R as follows:

$$\Phi_R(\Psi) = \Delta_R + A_R \cdot \Psi \cdot (\Gamma_R \cdot \Psi)^* \cdot B_R$$

where Ψ is the matrix created from the compound matrices $\psi[E_1], \ldots, \psi[E_n]$:

$$\Psi = \begin{pmatrix} \psi[E_1] & 0 & \ldots & 0 \\ 0 & \psi[E_2] & \ldots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \ldots & \psi[E_n] \end{pmatrix}$$

The symbols $+$ and $\cdot$ denote matrix addition and multiplication with respect to the Boolean algebra (in other words, union and composition of relations), the symbol $*$ is the reflexive and transitive closure.

The matrices A_R, B_R, Γ_R, and Δ_R are constructed as follows:

$$\Delta_R = \begin{pmatrix} \delta \end{pmatrix} \quad A_R = \begin{pmatrix} \alpha_1 & \alpha_2 & \ldots & \alpha_n \end{pmatrix}$$

$$B_R = \begin{pmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_n \end{pmatrix} \quad \Gamma_R = \begin{pmatrix} \gamma_{1,1} & \gamma_{1,2} & \cdots & \gamma_{1,n} \\ \gamma_{2,1} & \gamma_{2,2} & \cdots & \gamma_{2,n} \\ \vdots & \vdots & & \vdots \\ \gamma_{n,1} & \gamma_{n,2} & \cdots & \gamma_{n,n} \end{pmatrix}$$

Here, α_p is the dependency matrix between the inherited arcs of E_0 and the inherited arcs of E_p, β_q is the dependency matrix between the synthesized arcs of E_q and the synthesized arcs of E_0, $\gamma_{p,q}$ is the dependency matrix between the synthesized arcs of E_p and the inherited arcs of E_q, and δ is the dependency matrix between the inherited and synthesized arcs of E_0.

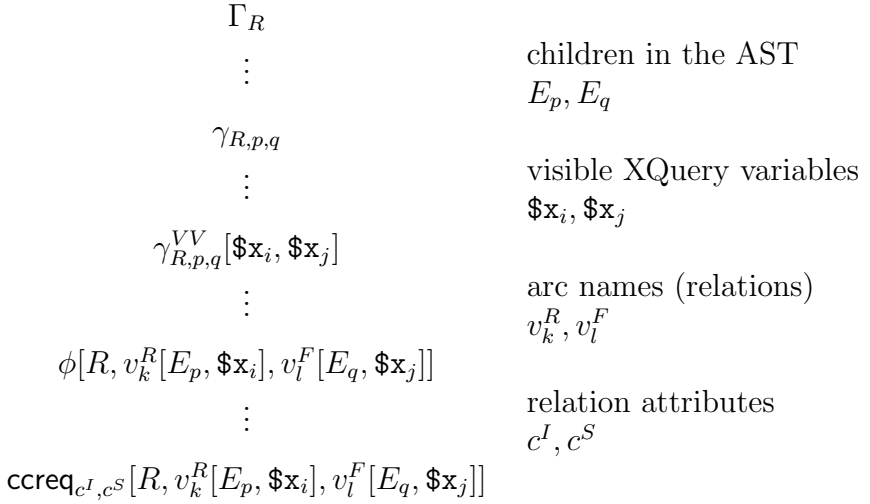


Figure 9.1: A part of the composition hierarchy of a Γ_R matrix

Similarly to the composition of the $\psi[E]$ matrices, each α_p, β_q, $\gamma_{p,q}$, or δ matrix is built from blocks corresponding to the expression and the visible variables. These blocks are divided to parts corresponding to arc names and, finally, the parts are composed of Boolean matrices according to the observed dependency (like the propagation of relation attributes). For illustration, a part of the hierarchy of a Γ_R matrix used in the backward phase is shown on the Fig. 9.1.

Function call rules

For a function call rule $R : E_0 ::= \mathtt{f}(E_1, \ldots, E_n)$, the matrices α_p, β_q, $\gamma_{p,q}$, and δ are dependent on the properties of the called function $\mathtt{f}(\$\mathtt{x}_1, \ldots, \$\mathtt{x}_n)$, i.e. on the matrix $\psi[E_f]$ where $E_F = \mathit{root}(\mathtt{f})$. The matrices are defined as follows:

$$\alpha^{EE}_{R,q} = \psi^{EV}[E_f, \$\mathtt{x}_q] \qquad \alpha^{VE}_{R,q}[\$\mathtt{x}_i] = 0$$
$$\alpha^{EV}_{R,q}[\$\mathtt{x}_j] = 0 \qquad \alpha^{VV}_{R,q}[\$\mathtt{x}_i, \$\mathtt{x}_j] = \mathbf{I} \text{ if } i = j$$
$$\alpha^{VV}_{R,q}[\$\mathtt{x}_i, \$\mathtt{x}_j] = 0 \text{ if } i \neq j$$

$$\beta^{EE}_{R,p} = \psi^{VE}[E_f, \$\mathtt{x}_p] \qquad \beta^{VE}_{R,p}[\$\mathtt{x}_i] = 0$$
$$\beta^{EV}_{R,p}[\$\mathtt{x}_j] = 0 \qquad \beta^{VV}_{R,p}[\$\mathtt{x}_i, \$\mathtt{x}_j] = \mathbf{I} \text{ if } i = j$$
$$\beta^{VV}_{R,p}[\$\mathtt{x}_i, \$\mathtt{x}_j] = 0 \text{ if } i \neq j$$

$$\gamma^{EE}_{R,p,q} = \psi^{VV}[E_f, \$\mathtt{x}_p, \$\mathtt{x}_q] \qquad \gamma^{VE}_{R,p,q}[\$\mathtt{x}_i] = 0$$
$$\gamma^{EV}_{R,p,q}[\$\mathtt{x}_j] = 0 \qquad \gamma^{VV}_{R,p,q}[\$\mathtt{x}_i, \$\mathtt{x}_j] = 0$$

$$\delta_R^{EE} = \psi^{EE}[E_f] \qquad \delta_R^{VE}[\$\mathtt{x}_i] = 0$$
$$\delta_R^{EV}[\$\mathtt{x}_j] = 0 \qquad \delta_R^{VV}[\$\mathtt{x}_i, \$\mathtt{x}_j] = 0$$

Here, the expression-to-expression dependencies α^{EE}, β^{EE}, γ^{EE}, δ^{EE} are carried through the called function, i.e. determined by its matrix $\psi[E_f]$. The variables are propagated from the node E to the children nodes – the dependency is determined by identities $\alpha_{R,q}^{VV}[\$\mathtt{x}_i, \$\mathtt{x}_i]$ and $\beta_{R,p}^{VV}[\$\mathtt{x}_i, \$\mathtt{x}_i]$. The rest of the matrices is zero since there are no other dependencies in the function call operator.

Operator rules

For grammar rules other than function calls, their matrices are fixed by the design of the corresponding transcription rules. (However, they may depend on the result of the previous phases of static analysis.)

The behavior of a grammar rule with respect to a variable $\$\mathtt{x}_i$ is identical for all the visible variables except of the variables defined or referenced by the rule; moreover, two different variables do not interfere. Therefore, the decomposition of α_p, β_q, $\gamma_{p,q}$, and δ matrices is simplified as shown in the following equations for $\gamma_{p,q}$:

$$\gamma_{R,p,q}^{EV}[\$\mathtt{x}_j] = \gamma_{R,p,q}^{EV}$$
$$\gamma_{R,p,q}^{VE}[\$\mathtt{x}_i] = \gamma_{R,p,q}^{VE}$$
$$\gamma_{R,p,q}^{VV}[\$\mathtt{x}_i, \$\mathtt{x}_j] = \gamma_{R,p,q}^{VV} \text{ if } i = j$$
$$\gamma_{R,p,q}^{VV}[\$\mathtt{x}_i, \$\mathtt{x}_j] = 0 \text{ if } i \neq j$$

The equations for α, β, and γ are similar. Thus, the behavior of a grammar rule R (having n children) is defined by the following set of matrices:

$$\alpha_{R,p}^{EE}, \alpha_{R,p}^{EV}, \alpha_{R,p}^{VE}, \alpha_{R,p}^{VV} \text{ for each } p \in \{1, \ldots, n\}$$
$$\beta_{R,q}^{EE}, \beta_{R,q}^{EV}, \beta_{R,q}^{VE}, \beta_{R,q}^{VV} \text{ for each } q \in \{1, \ldots, n\}$$
$$\gamma_{R,p,q}^{EE}, \gamma_{R,p,q}^{EV}, \gamma_{R,p,q}^{VE}, \gamma_{R,p,q}^{VV} \text{ for each } p, q \in \{1, \ldots, n\}$$
$$\delta_R^{EE}, \delta_R^{EV}, \delta_R^{VE}, \delta_R^{VV}$$

In addition, rules that define a variable $\$\mathtt{y}$ visible in its p-th child define the following dependency matrices:

$$\alpha_{R,p}^{EV}[\$\mathtt{y}], \beta_{R,q}^{VE}[\$\mathtt{y}], \gamma_{R,p,p}^{VV}[\$\mathtt{y}, \$\mathtt{y}]$$
$$\gamma_{R,q,p}^{EV}[\$\mathtt{y}], \gamma_{R,p,q}^{VE}[\$\mathtt{y}] \text{ for each } q \in \{1, \ldots, n\}$$

Conversely, a rule that references a variable `$y` must define the following additional matrices:

$$\delta_R^{EV}[\texttt{\$y}], \delta_R^{VE}[\texttt{\$y}], \delta_R^{VV}[\texttt{\$y}, \texttt{\$y}]$$

9.5 Dependency analysis algorithm

Since recursion may exist among user-defined functions, the dependency analysis algorithm shown in Alg. 1 computes all the matrices $\psi[E]$ at once, in iterative manner. During the computation, the algorithm maintains a quadruple of matrices alpha[E], beta[E], gamma[E], and delta[E] for each AST node E. Their meaning is derived from the defining equation of $\Phi_R(\Psi)$ (see the Section 9.4):

$$\text{alpha}[E] = A_R \cdot (\Psi \cdot \Gamma_R)^*$$

$$\text{beta}[E] = (\Gamma_R \cdot \Psi)^* \cdot B_R$$

$$\text{gamma}[E] = (\Gamma_R \cdot \Psi)^* \cdot \Gamma_R$$

$$\text{delta}[E] = \Delta_R + A_R \cdot \Psi \cdot (\Gamma_R \cdot \Psi)^* \cdot B_R$$

Here, Ψ denotes the previous value of delta[E_1], ..., delta[E_n] associated to the children of E, arranged diagonally into a matrix.

In the case of function call, the matrices alpha[E], beta[E], and gamma[E] are sparse, due to the special properties of the A_R, A_R, A_R, and A_R matrices (see the previous section). Therefore, the implementation stores only their parts corresponding to alpha^{EE}[E], alpha^{VE}[E], beta^{EE}[E], beta^{EV}[E] and gammaEE[E]. The rest is either zero or an identity (which is embedded in the algorithm).

The computation starts by initializing alpha[E], beta[E], gamma[E], and delta[E] from A_R, B_R, Γ_R, and Δ_R (see lines 9 to 12 of Alg. 1). This initialization corresponds to the equations with $\Psi = 0$. For function call operators, the matrices are set to zeros (see lines 4 to 7 of Alg. 1)

Then, the algorithm increases the values of alpha[E], beta[E], gamma[E], and delta[E] until the above-mentioned four equations are satisfied. In this moment, the fourth equation ensures that delta[E], substituted for $\psi[E]$, satisfies the defining equation of $\Phi_R(\Psi)$.

Each increase to a matrix delta[E] is decomposed to atomic changes corresponding to setting delta[E][i, j] from 0 to 1. Each atomic change, including the initialization (see lines 13 to 19 of Alg. 1), is logged to a stack as the triplet $\langle E, i, j \rangle$.

Algorithm 1 Dependency analysis algorithm

Input: $A_{R[E]}, B_{R[E]}, \Gamma_{R[E]}, \Delta_{R[E]}$ for every $E \in \mathsf{Addr}$

Output: alpha$[E]$, beta$[E]$, gamma$[E]$, delta$[E]$ for every $E \in \mathsf{Addr}$

```
1: stk := empty
2: for all E ∈ Addr do
3:    if E is a function call then
4:       alpha[E] := 0
5:       beta[E] := 0
6:       gamma[E] := 0
7:       delta[E] := 0
8:    else
9:       alpha[E] := A_{R[E]}
10:      beta[E] := B_{R[E]}
11:      gamma[E] := Γ_{R[E]}
12:      delta[E] := Δ_{R[E]}
13:      for all i ∈ rows(delta[E]) do
14:         for all j ∈ cols(delta[E]) do
15:            if delta[E][i, j] = 1 then
16:               stk.push(⟨E, i, j⟩)
17:            end if
18:         end for
19:      end for
20:   end if
21: end for
22: while not stk.empty do
23:   ⟨E, i, j⟩ := stk.pop
24:   if E = root(f) for some function f then
25:      for all E_0 ∈ Addr such that E_0 ::= f(E_1, ..., E_n) do
26:         propagateFnc(E,i,j)
27:      end for
28:   else
29:      E_0 := parent(E)
30:      if E_0 is a function call then
31:         propagateCall(E_0,E,i,j)
32:      else
33:         propagateOp(E_0,E,i,j)
34:      end if
35:   end if
36: end while
```

Algorithm 2 Function propagateOp

Input: $E_0 = \mathit{parent}(E)$, $i \in \mathit{rows}(\text{delta}[E])$, $j \in \mathit{cols}(\text{delta}[E])$

1: $i' := \mathit{rowPackChild}(E_0, E, i)$
2: $j' := \mathit{colPackChild}(E_0, E, j)$
3: $M_1 := \mathit{crows}(\text{alpha}[E_0], i')$
4: $M_2 := \mathit{crows}(\text{gamma}[E_0], i')$
5: $M_3 := \mathit{rcols}(\text{beta}[E_0], j')$
6: $M_4 := \mathit{rcols}(\text{gamma}[E_0], j')$
7: expand(M_1, M_2, M_3, M_4)

Algorithm 3 Function propagateCall

Input: $E_0 ::= \mathtt{f}(E_1, \ldots, E_n)$, $E_0 = \mathit{parent}(E)$,
$i \in \mathit{rows}(\text{delta}[E])$, $j \in \mathit{cols}(\text{delta}[E])$

1: **if** i belongs to a variable **then**
2: $M_1 := \{i\}$
3: $M_2 := \emptyset$
4: **else**
5: $i' := \mathit{rowPackChild}(E_0, E, i)$
6: $M_1 := \mathit{crows}(\text{alpha}[E_0], i')$
7: $M_2 := \mathit{crows}(\text{gamma}[E_0], i')$
8: **end if**
9: **if** j belongs to a variable **then**
10: $M_3 := \{j\}$
11: $M_4 := \emptyset$
12: **else**
13: $j' := \mathit{colPackChild}(E_0, E, j)$
14: $M_3 := \mathit{rcols}(\text{beta}[E_0], j')$
15: $M_4 := \mathit{rcols}(\text{gamma}[E_0], j')$
16: **end if**
17: expand(M_1, M_2, M_3, M_4)

Algorithm 4 Function propagateFnc

Input: $E_0 ::= \mathtt{f}(E_1, \ldots, E_n)$, $E = root(\mathtt{f})$,
$i \in rows(\text{delta}[E])$, $j \in cols(\text{delta}[E])$

1: **if** i belongs to the p-th argument **then**
2: $i' := colPackEx(E_p, rowUnpackVar(E, p, i))$
3: $i'' := rowPackChild(E_0, E_p, i')$
4: $M_1 := \emptyset$
5: $M_2 := \{i''\}$
6: **for all** $l \in crows(\text{delta}[E_p], i')$ **do**
7: $l' := colPackChild(E_0, E_p, l)$
8: $M_1 := M_1 \cup crows(\text{alpha}[E_0], l')$
9: $M_2 := M_2 \cup crows(\text{gamma}[E_0], l')$
10: **end for**
11: **else**
12: $i' := rowPackEx(E_0, rowUnpackEx(E, i))$
13: $M_1 := \{i'\}$
14: $M_2 := \emptyset$
15: **end if**
16: **if** j belongs to the q-th argument **then**
17: $j' := rowPackEx(E_q, colUnpackVar(E, q, j))$
18: $j'' := colPackChild(E_0, E_q, j')$
19: $M_3 := \emptyset$
20: $M_4 := \{j''\}$
21: **for all** $k \in rcols(\text{delta}[E_q], j')$ **do**
22: $k' := rowPackChild(E_0, E_p, k)$
23: $M_3 := M_3 \cup rcols(\text{beta}[E_0], k')$
24: $M_4 := M_4 \cup rcols(\text{gamma}[E_0], k')$
25: **end for**
26: **else**
27: $j' := colPackEx(E_0, colUnpackEx(E, j))$
28: $M_3 := \{j'\}$
29: $M_4 := \emptyset$
30: **end if**
31: expand(M_1, M_2, M_3, M_4)

Algorithm 5 Function expand

Input: $E_0 \in \mathsf{Addr}$, $M_1 \subseteq \mathit{rows}(\text{delta}[E_0])$, $M_2 \subseteq \mathit{rows}(\text{gamma}[E_0])$, $M_3 \subseteq \mathit{cols}(\text{delta}[E_0])$, $M_4 \subseteq \mathit{cols}(\text{gamma}[E_0])$

1: **for all** $k \in M_1$ **do**
2: **for all** $l \in M_3$ **do**
3: **if** $\text{delta}[E_0][k,l] = 0$ and $\langle k,l \rangle$ is not forbidden **then**
4: $\text{delta}[E_0][k,l] := 1$
5: $\text{stk.push}(\langle E_0,k,l \rangle)$
6: **end if**
7: **end for**
8: **for all** $l \in M_4$ **do**
9: $\text{alpha}[E_0][k,l] := 1$
10: **end for**
11: **end for**
12: **for all** $k \in M_2$ **do**
13: **for all** $l \in M_3$ **do**
14: $\text{beta}[E_0][k,l] := 1$
15: **end for**
16: **for all** $l \in M_4$ **do**
17: $\text{gamma}[E_0][k,l] := 1$
18: **end for**
19: **end for**

The stack controls the core of the algorithm (lines 22 to 36 of Alg. 1). Each triplet $\langle E, i, j\rangle$ removed from the stack induces an update to the matrix delta$[E_0]$ for the parent E_0 of the node E, using the procedure propagateOp or propagateCall. Whenever E is the root of a function AST, all nodes E_0 carrying the call to the procedure are updated using the procedure propagateFnc.

The matrices alpha$[E]$ and beta$[E]$ are used to speed-up the update of delta$[E]$; their effective maintenance requires the matrix gamma$[E]$. For a given node E, all four matrices are changed simultaneously, using the procedure expand (see Alg. 5). This procedure is controlled by two sets M_1, M_2 of row numbers and two sets M_3, M_4 of column numbers, setting the positions $M_1 \times M_4$, $M_2 \times M_3$, $M_2 \times M_4$, and $M_1 \times M_3$ in the matrices alpha$[E]$, beta$[E]$, gamma$[E]$, and delta$[E]$, respectively. In the cycle removal phase, the procedure expand does not propagate forbidden dependencies (see below); for the other two phases, the part of the test at the line 3 is omitted.

The procedure propagateOp is called whenever a child E of an AST node E_0 has set its delta$[E][i, j]$ to 1. The coordinates $\langle i, j\rangle$ are converted to the corresponding position $\langle i', j'\rangle$ in the Ψ matrix (see the Sec. 9.4), using the auxiliary functions *rowPackChild* and *colPackChild* (which consist of the addition of a constant). Then, functions *crows* and *rcols* are used to determine the sets of non-zero positions in a column or a row of a matrix, respectively.

The code of the procedure propagateOp implements the following observation: Let $\Psi' = \Psi + K$ where K is a matrix containing exactly one 1 and the rest set to 0. Then, the values alpha$[E_0]'$, beta$[E_0]'$, gamma$[E_0]'$, and delta$[E_0]'$ corresponding to Ψ' may be determined from the values alpha$[E_0]$, beta$[E_0]$, gamma$[E_0]$, and delta$[E_0]$ corresponding to Ψ as follows:

$$\text{alpha}[E_0]' = \text{alpha}[E_0] + \text{alpha}[E_0] \cdot K \cdot \text{gamma}[E_0]$$

$$\text{beta}[E_0]' = \text{beta}[E_0] + \text{gamma}[E_0] \cdot K \cdot \text{beta}[E_0]$$

$$\text{gamma}[E_0]' = \text{gamma}[E_0] + \text{gamma}[E_0] \cdot K \cdot \text{gamma}[E_0]$$

$$\text{delta}[E_0]' = \text{delta}[E_0] + \text{alpha}[E_0] \cdot K \cdot \text{beta}[E_0]$$

The lines 4 to 7 of Alg. 2 compute the sets of nonzero rows/columns in alpha$[E_0] \cdot K$, gamma$[E_0] \cdot K$, $K \cdot$ beta$[E_0]$, and $K \cdot$ gamma$[E_0]$.

The procedure propagateCall in Alg. 3 is derived from the procedure propagateOp, with respect to the special properties of A, B, Γ, and Δ matrices of function calls (see the Sec. 9.4) and the sparsity of the corresponding matrices alpha, beta, and gamma. Therefore, the behavior depends on the

position $\langle i, j \rangle$ with respect to the composition of the compound dependency matrix $\psi[E]$ – see the Sec. 9.3.

The procedure propagateFnc in Alg. 4 recalculate the matrices alpha$[E_0]$, beta$[E_0]$, gamma$[E_0]$, and delta$[E_0]$ whenever the matrix delta$[E]$ corresponding to the called function changes. In this case, the values of A_R, B_R, Γ_R, and Δ_R change as a result of the change of delta$[E]$. The effect depends on the position $\langle i, j \rangle$ of the atomic change in the compound dependency matrix delta$[E]$ with respect to its composition.

When $\langle i, j \rangle$ falls into the $\psi^{EE}[E]$ part (lines 13–15 and 28–30), the change affects the matrix δ_R; when in the $\psi^{EV}[E, \$\mathtt{x}_q]$ part (lines 13–15 and 18–21), the matrix $\alpha_R[E_q]$ is affected; $\psi^{VE}[E, \$\mathtt{x}_p]$ influences $\beta_R[E_p]$ (lines 3–6 and 28–30); finally, $\psi^{VV}[E, \$\mathtt{x}_p, \$\mathtt{x}_q]$ affects $\gamma_R[E_p][E_r]$ (lines 3–6 and 18–21).

The calculated change of A_R, B_R, Γ_R, and Δ_R propagate to the defining equations alpha$[E_0]$, beta$[E_0]$, gamma$[E_0]$, and delta$[E_0]$. The propagation depends on the values of Ψ (i.e. packed delta$[E_k]$); the total effect is calculated at the lines 6–10 and 21–25 based on the following equations derived from the four defining equations.

For an atomic change K_A in A_R:

$$\begin{aligned} \text{alpha}[E_0]' &= \text{alpha}[E_0] + K_A \cdot (1 + \Psi \cdot \text{gamma}[E_0]) \\ \text{delta}[E_0]' &= \text{delta}[E_0] + K_A \cdot \Psi \cdot \text{beta}[E_0] \end{aligned}$$

For an atomic change K_B in B_R:

$$\begin{aligned} \text{beta}[E_0]' &= \text{beta}[E_0] + (1 + \text{gamma}[E_0] \cdot \Psi) \cdot K_B \\ \text{delta}[E_0]' &= \text{delta}[E_0] + \text{alpha}[E_0] \cdot \Psi \cdot K_B \end{aligned}$$

For an atomic change K_Γ in Γ_R:

$$\begin{aligned} \text{alpha}[E_0]' &= \text{alpha}[E_0] + \text{alpha}[E_0] \cdot \Psi \cdot K_\Gamma \cdot (1 + \Psi \cdot \text{gamma}[E_0]) \\ \text{beta}[E_0]' &= \text{beta}[E_0] + (1 + \text{gamma}[E_0] \cdot \Psi) \cdot K_\Gamma \cdot \Psi \cdot \text{beta}[E_0] \\ \text{gamma}[E_0]' &= \text{gamma}[E_0] + \\ &\quad (1 + \text{gamma}[E_0] \cdot \Psi) \cdot K_\Gamma \cdot (1 + \Psi \cdot \text{gamma}[E_0]) \\ \text{delta}[E_0]' &= \text{delta}[E_0] + \text{alpha}[E_0] \cdot \Psi \cdot K_\Gamma \cdot \Psi \cdot \text{beta}[E_0] \end{aligned}$$

For an atomic change K_Δ in Δ_R:

$$\text{delta}[E_0]' = \text{delta}[E_0] + K_\Delta$$

Proper positioning of the atomic change is ensured by the functions *colUnpackEx*, *RowUnpackEx*, *colUnpackVar*, *RowUnpackVar*, *colPackEx*, *RowPackEx*, *colPackVar*, and *RowPackVar* that perform the (de-)composition of the compound matrices as defined in the Sec. 9.3.

Complexity

Let $m = max(m(E))$ be the maximal number of visible variables among all nodes of the program. Then the maximal size of a compound matrix assigned to any node is

$$s(m) = (n_e^R + m \cdot n_v^F) \cdot (n_e^F + m \cdot n_v^R) \cdot k$$

Note that $n_e^R, n_v^F, n_e^F, n_v^R$ are parameters of the system (proportional to the number of transcription modes) and k is a parameter of the analysis phase (the size of the elementary dependency matrix ϕ). Therefore, taking the parameters as constants, $s(m) = O(m^2)$.

Since the compound matrix delta$[E]$ contains at most $s(m)$ Boolean values, it may be changed at most $s(m)$ times; consequently, a given E is encountered in the main loop of the algorithm at most $s(m)$ times.

Let n be the size of the AST forest, i.e. the size of the query. The previous observation implies that the total number of calls to propagateOp or propagateCall is at most $n \cdot s(m)$. The function propagateFnc is called in a loop over all call nodes to the given XQuery function; nevertheless, since the total number of call nodes in the program is less than n, the total number of calls to propagateFnc is less than $n \cdot s(m)$.

In the interior of the Alg. 2, the function expand is called, having the time complexity proportional to the size of the greatest matrix gamma$[E]$, which is $O((1+a)^2 \cdot s(m))$ where a is the number of children of the node E. Except for the function-call node, a is a small number limited by the design of grammar rules; therefore, the expand function has the time complexity $O(s(m))$, i.e. $O(m^2)$.

In the case of function call, only reduced versions of the matrices alpha$[E]$, beta$[E]$, and gamma$[E]$ are stored. All three reduced matrices have the size $O(m^2)$; therefore, the expand function has the time complexity $O(m^2)$ also for the function call case.

Prior to the call to expand, Alg. 2, 3, and 4 compute the $M_1, \ldots, M_4$ sets. The cost of this computation is proportional to the size of the (reduced) matrices, i.e. $O(m^2)$. Therefore, each call to propagateOp, propagateCall, or propagateFnc requires the time $O(m^2)$.

Concluded, the total time complexity of the Alg. 1 is

$$O(n \cdot m^4)$$

where n is the size of the program and m is the maximal number of visible variables in the program.

For space complexity, the most important factor is the summary size of all the (reduced) gamma[E] matrices, which is

$$O(n \cdot m^2)$$

9.6 Forward propagation phase

As shown in the previous sections, the first step of the forward propagation phase computes the dependencies denoted $\mathsf{carddep}_{0,0}$, $\mathsf{carddep}_{0,1}$, $\mathsf{carddep}_{1,0}$, and $\mathsf{carddep}_{1,1}$ which determine how a sub-expression behave with respect to the cardinality of the relations involved. These relations are computed using the dependency analysis algorithm 1 and packed in the dependency matrices alpha[E], beta[E], gamma[E] and delta[E] for each node E.

In the second step, absolute cardinality information is determined. The absolute cardinality information is the pair of properties card_0 and card_1 assigned to each inherited and each synthesized arc, determining that the arc will always carry an empty relation (in the case of card_0) or a singleton value (for card_1). For an AST node E, the information is packed into two vectors, icard[E] and scard[E], for inherited and synthesized arc, respectively. The vectors are arranged in the same manner as the rows and columns of the dependency matrices. In the forward propagation phase, the transcription is limited to the canonical and Boolean rules which contain no reverted arcs; therefore, the icard[E] vector contains the cardinality flags for the visible variables and the scard[E] vector carries the cardinality flags for the expression value. In addition, the cardinality flags for the $\mathsf{invv}[E]$ relation is added to the icard[E] vector.

The vector icard[$root(main)$] associated to the main expression of the XQuery program is set to values corresponding to the start of the program; in particular, $\mathsf{card}_1(\mathsf{invv})$ is set to true to indicate that $\mathsf{invv}[root(main)]$ is a singleton.

The values of the icard[E] vectors are propagated down through the AST forest, from parents to children and from function calls to the corresponding AST roots. In the case of parent-to-children propagation, the propagation is determined by the alpha[E_0] relations computed in the dependency analysis step:

$$\text{icard}[E_i] = (\text{icard}[E_0] \cdot \text{alpha}[E_0])[i]$$

For function calls, the vectors propagate through the delta[E_1], ..., delta[E_n] relations of the children to the arguments of the called function:

$$\text{icard}[\mathit{root}(\texttt{f})] = \text{icard}[E_0] \cdot (\text{delta}[E_1], \ldots, \text{delta}[E_n])$$

Finally, the scard[E] vectors are computed from icard[E] and delta[E] as follows:

$$\text{scard}[E] = \text{icard}[E] \cdot \text{delta}[E]$$

The vectors are computed iteratively, starting from zero vectors and propagating changes using a stack. The algorithm is similar to the Alg. 1 with these two differences: Vectors are propagated instead of matrices and the propagation is top-down instead of bottom-up. Because of the simplicity and similarity, we do not show the forward propagation algorithm in code.

Complexity

Because of the similarity, the complexity analysis of the forward propagation algorithm is similar to the case of dependency propagation algorithm. The state space of the algorithm is driven by the icard vectors; their total size is $O(n \cdot m)$ so the algorithm will propagate at most $O(n \cdot m)$ changes in these vectors. Each change propagates through a multiplication with a matrix of size $O(m)$ times $O(m)$; therefore, it induces further changes in at most $O(m)$ positions. Concluded, the time complexity of the second step of the forward propagation is

$$O(n \cdot m^2)$$

9.7 Cycle removal phase

The cycle removal phase simulates the use of all transcription rules (applicable with respect to the results of the forward propagation phase) at once. During the phase, some transcription rules are removed in order to avoid cyclic dependencies. Since the rules themselves are acyclic, each cycle contains at least two rules, offering the choice to remove any of these rules. In our approach, there are two methods used to detect and remove rules in a cycle. These two methods covers the two different cases described in the following paragraphs.

Each cycle is spread over several AST nodes and, optionally, over several functions through function calls. Thus, the cycle generates a partial call tree formed of the involved functions and an AST sub-tree in each of the functions. In the root function of the partial call tree, the root of the cycle's AST sub-tree is called the *summit* of the cycle – this is the place where all

the dependencies in the cycle arise from the children of the summit node and are bent back down to the children. Formally, there is a cycle in the graph of the relation $\Psi \cdot \Gamma_R$ (see the Sec. 9.4).

Two cases may arise in the cycle within the summit node rule:

- An inherited arc of child E_i depends on a synthesized arc of the same child E_i. Formally, $\psi[E_i] \cdot \gamma_{R,i,i}$ is cyclic.
- An inherited arc of child E_j depends on a synthesized arc of another child E_i, $i \neq j$.

More than one part of the cycle may be stitched through the summit rule; therefore, the above-mentioned cases are handled independently in various parts of the same cycle.

The cycle removal phase is constituted of two steps: Dependency analysis algorithm and local cycle removal. As described in the previous sections, the dependency analysis algorithm proceeds in bottom-up manner. Therefore, for each cycle, the summit rule is the last rule encountered during the run of the algorithm. Thus, the summit rule is the place where each cycle may be detected. Unfortunately, the transcription rule that forms the part of the dependency cycle at the summit is not always removable. Therefore, the removal of a transcription rule under a child of the summit node may be required.

Within the frame of bottom-up evaluation, the removal under a child requires a kind of prediction to determine whether a cycle may be closed at the level of the parent node. Such prediction is possible in the first of the two cases mentioned above. The prediction is based on the notion of *forbidden* dependencies: There is a statically determined set of dependencies that must never occur in the $\psi[E]$ matrix for any AST node E. The set of *forbidden* dependencies is chosen so that it avoids the first case of dependency, i.e. when $\psi[E]$ contains no forbidden dependency, then $\psi[E] \cdot \gamma_{R,i,i}$ is acyclic for any rule R and any child position i.

It depends on the properties of the $\gamma_{R,i,i}$ matrices whether such a set of forbidden dependencies exists. This leads to the notion of *ordered rule set*: A set of transcription rules is called *ordered* if there exists a total strict order P on the set of arc names used by the transcription rules (see the Sec. 6.2) such that all dependencies in all $\gamma_{R,i,i}$ matrices respect the order P, i.e. the following implications hold:

$$\langle a[E_i], b[E_i] \rangle \in \gamma_{R,i,i} \Rightarrow \langle a, b \rangle \in P$$

$$\langle a[E_i], b[E_i, \$\mathtt{x}_k] \rangle \in \gamma_{R,i,i} \Rightarrow \langle a, b \rangle \in P$$

$$\langle a[E_i, \$\mathtt{x}_j], b[E_i] \rangle \in \gamma_{R,i,i} \Rightarrow \langle a, b \rangle \in P$$
$$\langle a[E_i, \$\mathtt{x}_j], b[E_i, \$\mathtt{x}_k] \rangle \in \gamma_{R,i,i} \Rightarrow \langle a, b \rangle \in P$$

Note that the requirement does not apply to $\gamma_{R,i,j}$ for $i \neq j$.

For the transcription rules presented in the chapters 7 and 8.4, such an ordering really exists: The arc names are sorted in the following order:

- Reverted arcs of expressions (like exiefr or exdesc - see the Sec. 8.4), including invc, invv and invenv
- Reverted arcs of variables (like variefr or vardesc)
- Forward arcs of variables (varseqa, varseqn, varebvt, varebvf, varout, variefr etc.)
- Forward arcs of expressions (exseqa, exseqn, exenv, exebvt, exebvf, exout, exiefr etc.)

Inside each group, any ordering may be chosen.

For a rule set ordered by P, the forbidden dependencies are defined by the transposed matrix P^T. Because P is a total ordering, any non-forbidden $\langle a[\ldots], b[\ldots] \rangle$ pair in any $\psi[E]$ matrix satisfies either $a = b$ or $\langle a, b \rangle \in P$. Thus, any $\langle a[\ldots], b[\ldots] \rangle$ pair in $\psi[E_i] \cdot \gamma_{R,i,i}$ must satisfy $\langle a, b \rangle \in P$; therefore, acyclicity is ensured by the presence of the total ordering P.

The dependency analysis step of the cycle removal phase constructs the $\psi[E]$ matrices incrementally in the delta$[E]$ matrices. The condition at the line 3 of function expand shown in the Alg. 5 ensures that forbidden dependencies are not propagated to the delta$[E]$ matrices.

The second step of the cycle removal ensures that the selected set of transcription rules R satisfies the following requirements for each node E_0:

- The matrix $\psi[E_0]$ contains no forbidden dependencies.
- The graph of the relation $\Psi \cdot \Gamma_R$ is acyclic.

Note that the matrix Ψ is a composition of the delta$[E_1]$, ..., delta$[E_n]$ matrices computed for the children in the first step. The matrix $\psi[E_0]$ must be recalculated from Ψ using the matrices Δ_R, A_R, Γ_R, and B_R which depend on the selected set of rules R.

In the second step, each node E_0 is inspected independently and its transcription rules are reexamined, starting with the highest priority (i.e. lowest cost) rules. If any of the two above-mentioned requirements is violated, the highest priority rule which contributes to the violation is replaced by a lower

priority rule that computes the same output at expectedly higher cost. This replacement is repeated until a satisfying set of rules is found.

The canonical transcription rules always satisfy the requirements; furthermore, any non-canonical representation may be derived from the relations of the canonical mode. Therefore, for each mode, a last-resort transcription rule is defined using canonical-mode input, the canonical rule R-net, and an additional R-net to convert the outputs to the required mode. The last-resort rules are assigned the lowest priority to ensure that the algorithm of the second step always terminates – in the worst case, all rules are replaced with the last-resort rules.

Complexity

The filtering of the forbidden dependencies in the first step has no impact on its asymptotic complexity. In the second step, AST nodes are examined independently; for each node, a transcription rule for each outgoing arc must be determined. The number of outgoing arcs is $O(m)$, i.e. linear with respect to the number of visible variables. The number of candidate rules for each arc is fixed by the system design. Application of a transcription rule requires recalculation of the $\psi[E_0]$ matrix – when done incrementally, it costs the time $O(m^2)$ for each rule. The acyclicity check for the $\Psi \cdot \Gamma_R$ matrix may also be done in the time $O(m^2)$. Concluded, each AST node requires the time $O(m^3)$; consequently, the whole second step of cycle removal has the time complexity

$$O(n \cdot m^3)$$

$$O(n \cdot m^2)$$

9.8 Backward propagation phase

The backward propagation phase follows the same scheme as the forward propagation with the difference that the sense of dependency is reverted, from the output to the input. It means that beta$[E_0]$ relations are used to propagate the flag vectors.

The requirement information is a pair of vectors ireq$[E]$ and sreq$[E]$ for each AST node E, containing Boolean flags that determine the requirement for particular inherited and synthesized arcs (and their relation attributes).

The vector sreq$[root(main)]$ associated to the main expression of the XQuery program is set according to the values required at the end of the program; it means that the corresponding flag is set to true to indicate that the output-driven mode arc $\mathsf{exout}[root(main)]$ is required.

The values of the sreq[E] vectors are propagated down through the AST forest, from parents to children and from function calls to the corresponding AST roots. In the case of parent-to-children propagation, the propagation is determined by the beta[E_0] relations computed in the dependency analysis step:

$$\text{sreq}[E_i] = (\text{beta}[E_0] \cdot \text{sreq}[E_0])[i]$$

For function calls, the vectors propagate through the delta[E_1], ..., delta[E_n] relations of the children to the arguments of the called function. At the same time, the requirements propagate also through the return value of the function:

$$\text{sreq}[\mathit{root}(\texttt{f})] = (\text{sreq}^E[E_0], \text{delta}[E_1] \cdot \text{sreq}^V[E_0], \ldots, \text{delta}[E_n] \cdot \text{sreq}^V[E_0])$$

Here, sreq^E and sreq^V denote the expression and variable portions of the sreq vector.

Finally, the ireq[E] vectors are computed from sreq[E] and delta[E] as follows:

$$\text{ireq}[E] = \text{delta}[E] \cdot \text{sreq}[E]$$

Complexity

The algorithm is complementary to the forward propagation algorithm. Therefore, its time complexity is the same, namely

$$O(n \cdot m^2)$$

9.9 Complexity of the static analysis

In all three phases of the static analysis, the first step dominates the time complexity. Since the second steps reuse the data of the first steps, their space complexity is equivalent. All the three first steps have the same asymptotic complexity $O(n \cdot m^4)$ (although their multiplicative constants are different because of different numbers of dependencies observed in different phases). Concluded, the total time complexity of the static analysis is

$$O(n \cdot m^4)$$

while the space complexity is

$$O(n \cdot m^2)$$

where n is the size of the XQuery program and m is the maximal number of local variables visible at any place in the program. It is important to notice that m is a local property of the program and, for programs decomposed to functions of limited size, the complexity is linear with respect to the size n of the program.

Chapter 10

Conclusion

In this work, we have presented a novel architecture aimed at effective evaluation of user-defined functions in XQuery. The architecture is based on relational representation of XML documents and XQuery values. The use of relational algebra in XML processing has already become almost a standard; however, the incorporation of functions requires significant alterations to contemporary designs. Therefore, to show the viability of the presented architecture, this work must present quite large amount of particular problems and their solution.

R-programs

We have presented a framework of R-programs, designed to handle recursion together with relational algebra operations. The R-programs allow bulk-evaluation of function calls, i.e. the ability to evaluate all calls to an XQuery function in a single call to the corresponding R-program function. The bulk-evaluation offers important advantages in the XQuery evaluation, starting with the reduction of call overhead and ending in inter-procedural optimization.

The novelty of this approach is in the absence of explicit control structures in the R-program language. Instead of stopping recursion by explicit conditions, an R-program evaluator may (or may not) decide to skip a function call based on an implicit condition. This gives the evaluator the freedom of choice to speculatively execute code before the decision is ready.

R-programs allow reusing sub-expressions; in such environment, common sub-expression elimination is applicable to reduce the size of the program and the cost of its evaluation. On the other hand, the reuse makes query rewrite more difficult, since an operator may be evaluated in more than one context. Moreover, pipelined evaluation of relational algebra operators in the

presence of expression reuse imposes additional synchronizing constraints; to satisfy these constraints, materialization of intermediate results may be required. Nevertheless, reevaluation is usually considered more costly than materialization.

One of the important advantages of the R-program model is the ability to distribute the evaluation over several computing nodes. The R-program operations (or chunks of them) may be computed at different nodes, while the R-program arcs are naturally implemented as pipes carrying relations among nodes. This way, R-programs may be easily implemented on cluster or grid architectures.

Modes and transcription

The XQuery language formally uses only one data-type (a sequence of items) to carry all kinds of values. Strict application of this approach in an implementation would degrade the performance significantly. Therefore, alternate representations of XQuery values are used, depending on the context.

Such an alternative representation for a value of an expression is called a *mode* of evaluation. In R-program representation, a XQuery value is carried by one or more relations, therefore, a mode is defined by the names and the signatures of these relations. The canonical mode is the most general mode corresponding most directly to the normative semantics of the language. Other modes are applied in special contexts; a mode is described by a mapping between the mode and the canonical mode. The alternative modes allow to reduce the amount of information relayed and to adapt the encoding to the subsequent operations.

For each operator (grammar rule) of the XQuery language, there is a set of transcription rules; each rule represents the evaluation of the XQuery operator with the operands and the result encoded in a particular mode.

Besides the selection among modes, there is also an option of "automatic" reduction whenever the consumers of an expression do not use all attributes of the model. In this way, submodes are automatically created based on static analysis. One of the submodes corresponds to the *unordered* context as defined by the XQuery standard; therefore, the static analysis algorithm described in this paper may also be used for the automatic determination of ordered and unordered contexts in a XQuery program. Methods of ordered/unordered context determination were already described; among them, *column dependency analysis* described in [41] is similar to our approach but based on a different mathematical model that does not handle user-defined functions.

The system is open – new modes may be added under the same notation and corresponding algorithms can be reused.

Algebra

In each mode, the expression or variable value is represented by a set of relations whose attributes are atomic values and strings in a hierarchical alphabet, acting similarly to Dewey identifiers. The use of hierarchical strings may be considered a violation of the *first normal form* – similar non-first-normal-form relations may be found in every relational model of XQuery. This is a natural consequence of the fact that a call to an XQuery function may have arguments carrying sequences. While most methods deal with this problem by allowing sequences in an attribute, our approach is different – hierarchical strings are used to identify the context in which a function is invoked. While sequences may easily be as large as input documents, the length of the identifiers is usually proportional to the depth of the documents; therefore, our approach generates smaller tuples than traditional models.

When designing our algebraic system, we tried to avoid introducing new operators. Thus, majority of XQuery operators is expressed using operators known from relational query engines. Although there are exotic operators like ordered grouping in our set of operators, their use is limited to infrequent XQuery constructs. Using the output-driven mode, normalization of the output document is moved to the very end of the query, leaving the core of the query populated with standard relational operators.

The models based on relational algebra may also offer new modes of optimization. For instance, the concatenation operator is rewritten using the union operator which is commutative; thus, transformations based on reordering of the operands become available.

Canonical and Boolean mode

The canonical mode described in the work is used to reflect the definition of the XQuery semantics as closely as possible. It would be natural to prove the equivalence between our canonical mode and the normative XQuery semantics; unfortunately, some of the most interesting parts of XQuery semantics, namely the effect of the order-by clause and the identity of constructed nodes, are described only informally, although under the title "Formal Semantics" [22]. Although not attempted in this work, the notation of R-programs may allow defining the XQuery semantics more exactly than the system used in the W3C specification.

Besides the canonical mode, we introduced the effective Boolean value mode in order to avoid the complex conversion rules of effective Boolean value whenever possible.

Reverted modes

The most important advantage of R-programs is the ability to pass information in the direction opposite to the original data flow. This reversal is used to simulate several variants of predicate pushing and join reordering known from pure relational algebra systems. Although the presented abilities are not as general as the traditional query rewrite techniques, we have presented that our R-program approach may be a competitive replacement of traditional systems. The loss of generality is balanced by the ability of R-programs to express recursive functions which is not possible in "single-expression" systems.

In XQuery, the most important operators are structural joins, in particular, their multiple-input (holistic) versions (see the Sec. *sec:joins*). Reverted modes offer bidirectional communication between atomic computing elements of an R-program; therefore, implementation of multiple-input joins using a network of communicating binary elements is possible. This arrangement allows to simulate some structural-join algorithms even in those cases where the join operation is spread across several functions in the XQuery source.

Output document handling

Although XQuery uses the same operators to handle input, temporary, and output data, it may be wise to use different implementations in these cases. We have shown an evaluation mode that reverses the standard flow of evaluation on the output, synthetic portion of the program. This approach allows generating elements of the output document directly at their final locations in the representation of the output document, bypassing the creation and merging of temporal trees required in the standard evaluation order. The method is developed for systems where the output of the transformation is stored in a database, bypassing any serialization.

This method of generating output is particularly advantageous in distributed environments, since it removes the bottleneck formed by the use of shared memory that would be otherwise required to store the temporary tree fragments.

Static analysis

Finally, a polynomial algorithm of static analysis is presented which evaluates the behavior of XQuery code with respect to the information required at any node of its abstract syntax tree. The algorithm is applicable to any group of modes based on the relational algebra. The process of static analysis replaces the stage of query rewriting known from relational databases. The static analysis is based on heuristic selection of modes and, thus, not necessarily optimal; on the other hand, adapting query rewriting methods to the recursive XQuery environment would be difficult. The area of compiler construction shows that inter-procedural optimization and, especially, code motion is expensive; therefore, it is important to have an effective algorithm of heuristic mode selection.

10.1 Future work

We have presented an idea that, in some aspects, differs significantly from traditional techniques of query evaluation. Therefore, its application within an existing engine is difficult or even impossible. Subsequently, before we can materialize our ideas in an implementation, many collateral problems shall be solved and many well-known solutions have to be adapted or redesigned. Some of the problems are mentioned in this section.

Besides the path to an implementation, there are also alternative (or more advanced) versions of the approach presented so far. One of them, based on a Datalog-like model, was already published in [8]; since we consider the Datalog version worthy of further development, an overview of this idea is presented in this section.

Schema awareness

If the input documents to the transformation are stored in schema-aware storage, the transcription phase may use the schema of the documents to adapt the R-program to the storage. It corresponds to replacing the single $\mathsf{invenv}[E]$ relation with a set of relations containing the shredded representation of the input document(s).

Cost-based optimization

Following the transcription, intra-procedural optimization (denoted as *static rewriting*) may be applied. Since the body of an R-function consists only of

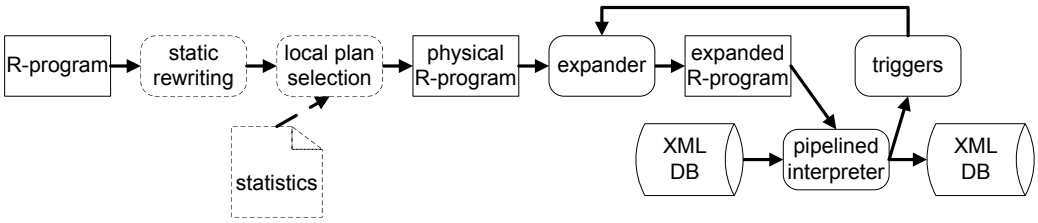


Figure 10.1: Static rewriting

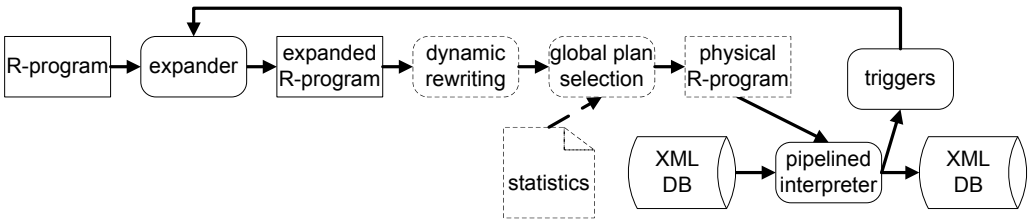


Figure 10.2: Dynamic rewriting

relational operators and function calls, almost any rewriting approach known from relational database systems may be applied.

Cost-based optimization may be required before the interpretation phase; in this setting, only intra-procedural optimization (denoted as *local plan selection*) is available. The physical plan is again in the form of a dag of algebra operators and function calls; thus it is again an R-program, albeit using a different set of operators. This architecture is shown in Fig. 10.1.

Note, however, that the effectivity of intra-procedural cost-based optimization is limited because the cost of function calls and cardinality of their outputs is not known. This weakness may be addressed with the architecture depicted at the Fig. 10.2.

In each cycle, the expanded R-program may be optimized by rewriting and transformed using cost-based plan selection to a physical R-program. Since the original R-functions were integrated into a single function, the optimization is in fact inter-procedural. Of course, this phase may alter only the newly appended R-function body because the previously integrated code is already being executed. On the other hand, the plan selection may make use of cost and cardinality estimation computed throughout the whole integrated program. Therefore, it may produce better plans than in the case

of recursive interpretation with local plan selection.

Datalog-like models

Relational systems with recursion are often examined using *Datalog* [18] or one of its derivatives; R-programs may be also rewritten using a notation inspired by Datalog. In Datalog-like representation, original function boundaries are completely dissolved – this fact may offer analysis and transformation options unavailable within the system of R-programs.

A Datalog-like representation may be created from an R-program as follows: Each place is represented by a *predicate* whose arguments correspond to the attributes of the relation associated to the place. In addition, an argument storing the call stack (see the Sec. 5.4) is added to simulate the semantics of R-program expansion. Each R-program operation is then transformed to a set of Horn clauses, using function symbols, negation, or aggregation.

Because of the Turing-completeness of XQuery (see the Sec. 2.11), such a system must be *unsafe*, allowing to create new values not listed in the extensional facts. This may lead to non-termination of bottom-up evaluation, a fact that may have been expected due to the existence of non-terminating XQuery programs.

Besides the issue of safety and termination, there are problems with semantics of the system: Some XQuery programs (including terminating ones) will be transformed to a non-stratifiable set of rules. Therefore we cannot rely on *minimal-model semantics*. Although the theory behind logical programming offers a number of more sophisticated definitions of semantics like *stable models* or *perfect models* [18], none of these definitions lead to results that exactly match the effect of the underlying XQuery program.

These issues open a new area of research – besides searching for the right definition of semantics, there is an interesting opportunity to classify XQuery programs based on their stratifiability or the number of the strata in their models. Interestingly, the stratifiability does not remove Turing-completeness from the language – even a program without negation may simulate a Turing machine using function symbols.

Bibliography

[1] Loredana Afanasiev, Torsten Grust, Maarten Marx, Jan Rittinger, and Jens Teubner. An inflationary fixed point operator in XQuery. In *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, México*, pages 1504–1506. IEEE, 2008.

[2] Rakesh Agrawal. Alpha: An extension of relational algebra to express a class of recursive queries. *IEEE Trans. Softw. Eng.*, 14(7):879–885, 1988.

[3] Rafiul Ahad and Bing Yao. RQL: A recursive query language. *IEEE Trans. on Knowl. and Data Eng.*, 5(3):451–461, 1993.

[4] Shurug Al-Khalifa, H. V. Jagadish, Nick Koudas, Jignesh M. Patel, Divesh Srivastava, and Yuqing Wu. Structural joins: a primitive for efficient XML query pattern matching. In *In ICDE*, pages 141–152, 2002.

[5] Christopher J. Augeri, Dursun A. Bulutoglu, Barry E. Mullins, Rusty O. Baldwin, and Leemon C. Baird, III. An analysis of XML compression efficiency. In *ExpCS '07: Proceedings of the 2007 workshop on Experimental computer science*, page 7, New York, NY, USA, 2007. ACM.

[6] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D Ullman. Magic sets and other strange ways to implement logic programs (extended abstract). In *PODS '86: Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 1–15, New York, NY, USA, 1986. ACM.

[7] Radim Bača, Michal Krátký, and Václav Snášel. On the efficient search of an XML twig query in large DataGuide trees. In *IDEAS '08: Proceedings of the 2008 international symposium on Database engineering and applications*, pages 149–158, New York, NY, USA, 2008. ACM.

[8] David Bednárek. Extending Datalog to cover XQuery. In Peter Vojtáš, editor, *Information Technologies - Applications and Theory*, pages 1–6, Jesenná 5, 040 01 Košice, Slovakia, 2008. PONT s.r.o., Seňa, Slovakia.

[9] Michael Benedikt, Leonid Libkin, Thomas Schwentick, and Luc Segoufin. Definable relations and first-order query languages over strings. *J. ACM*, 50(5):694–751, 2003.

[10] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, and Jérôme Siméon. *XML Path Language (XPath) 2.0*. W3C, January 2007.

[11] R. Bloem and J. Engelfriet. A comparison of tree transductions defined by monadic second order logic and by attribute grammars. *J. Comput. Syst. Sci.*, 61(1):1–50, August 2000.

[12] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. *XQuery 1.0: An XML Query Language*. W3C, January 2007.

[13] P. A. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. MonetDB/XQuery: A fast XQuery processor powered by a relational engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Chicago, IL, USA, June 2006.

[14] Peter Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. Pathfinder: XQuery—the relational way. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 1322–1325. VLDB Endowment, 2005.

[15] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, cois Yergeau Fran and John Cowan. *Extensible Markup Language (XML) 1.1 (Second Edition)*. W3C, August 2006.

[16] David Brownell. *SAX2*. O'Reilly Media, Inc., 2002.

[17] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: optimal xml pattern matching. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 310–321, New York, NY, USA, 2002. ACM.

[18] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about Datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, 1989.

[19] Donald D. Chamberlin. XQuery: Where do we go from here? In *Proceedings of the 3rd International Workshop on XQuery Implementation, Experience and Perspectives, in cooperation with ACM SIGMOD, June 30, 2006, Chicago, USA*, 2006.

[20] Surajit Chaudhuri. An overview of query optimization in relational systems. In *PODS '98: Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 34–43, New York, NY, USA, 1998. ACM.

[21] John Cowan and Richard Tobin. *XML Information Set (Second Edition)*. W3C, February 2004.

[22] Denise Draper, Peter Fankhauser, Mary Fernández, Ashok Malhotra, Kristoffer Rose, Michael Rys, Jérôme Siméon, and Philip Wadler. *XQuery 1.0 and XPath 2.0 Formal Semantics*. W3C, January 2007.

[23] Jana Dvořáková and Filip Zavoral. Xord: An implementation framework for efficient XSLT processing. In Costin Badica, Giuseppe Mangioni, Vincenza Carchiolo, and Dumitru Burdescu, editors, *2nd International Symposium on Intelligent Distributed Computing*, volume 162 of *Studies in Computational Intelligence*, Catania, Italy, 2008. Springer-Verlag.

[24] Maged El-Sayed, Katica Dimitrova, and Elke A. Rundensteiner. Efficiently supporting order in XML query processing. In *WIDM '03: Proceedings of the 5th ACM international workshop on Web information and data management*, pages 147–154, New York, NY, USA, 2003. ACM.

[25] Maged El-Sayed, Elke A. Rundensteiner, and Murali Mani. Incremental fusion of XML fragments through semantic identifiers. In *IDEAS '05: Proceedings of the 9th International Database Engineering & Application Symposium*, pages 369–378, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[26] Maged EL-Sayed, Ling Wang, Luping Ding, and Elke A. Rundensteiner. An algebraic approach for incremental maintenance of materialized XQuery views. In *WIDM '02: Proceedings of the 4th international workshop on Web information and data management*, pages 88–91, New York, NY, USA, 2002. ACM.

[27] Vidur Apparao et al. *Document Object Model (DOM) Level 1 Specification*. W3C, October 1998.

[28] Leonidas Fegaras, Ranjan K. Dash, and YingHui Wang. A fully pipelined XQuery processor. In *XIMEP 2006, 3rd International Workshop on XQuery Implementation, Experiences and Perspectives*. ACM, 2006.

[29] Mary Fernández, Ashok Malhotra, Jonathan Marsh, Marton Nagy, and Norman Walsh. *XQuery 1.0 and XPath 2.0 Data Model (XDM)*. W3C, January 2007.

[30] Achille Fokoue, Kristoffer Rose, Jérôme Siméon, and Lionel Villard. Compiling XSLT 2.0 into XQuery 1.0. In Allan Ellis and Tatsuya Hagino, editors, *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 682–691, New York, NY, USA, 2005. ACM.

[31] Andrey Fomichev, Maxim Grinev, and Sergey Kuznetsov. Sedna: A native XML DBMS. pages 272–281. 2006.

[32] Marcus Fontoura, Vanja Josifovski, Eugene Shekita, and Beverly Yang. Optimizing cursor movement in holistic twig joins. In *CIKM '05: Proceedings of the 14th ACM international conference on Information and knowledge management*, pages 784–791, New York, NY, USA, 2005. ACM.

[33] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall, first edition, October 2001.

[34] Roy Goldman and Jennifer Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB '97: Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 436–445, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[35] Georg Gottlob, Christoph Koch, Robert Baumgartner, Marcus Herzog, and Sergio Flesca. The Lixto data extraction project: back and forth between theory and practice. In *PODS '04: Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–12, New York, NY, USA, 2004. ACM.

[36] Sven Groppe, Stefan Böttcher, Georg Birkenheuer, and André Höing. Reformulating XPath queries and XSLT queries on XSLT views. *Data Knowl. Eng.*, 57(1):64–110, 2006.

[37] Sven Groppe, Jinghua Groppe, and Stefan Böttcher. Simplifying XPath queries for optimization with regard to the elimination of intersect and except operators. *Data Knowl. Eng.*, 65(2):198–222, 2008.

[38] Torsten Grust, Maurice Keulen, and Jens Teubner. Staircase join: Teach a relational DBMS to watch its (axis) steps. In *In Proc. of the 29th Intl Conference on Very Large Databases (VLDB*, pages 524–535, 2003.

[39] Torsten Grust, Manuel Mayr, and Jan Rittinger. XQuery join graph isolation. *CoRR*, abs/0810.4809, 2008.

[40] Torsten Grust and Jan Rittinger. Jump through hoops to grok the loops Pathfinder's purely relational account of XQuery-style iteration semantics. In *Proceedings of the ACM SIGMOD/PODS 5th International Workshop on XQuery Implementation, Experience and Perspectives (XIME-P 2008)*, June 2008.

[41] Torsten Grust, Jan Rittinger, and Jens Teubner. eXrQuy: Order Indifference in XQuery. In *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, April 15-20, Istanbul, Turkey*, pages 226–235. IEEE, 2007.

[42] Torsten Grust and Jens Teubner. Relational Algebra: Mother Tongue - XQuery: Fluent. In Vojkan Mihajlovic and Djoerd Hiemstra, editors, *First Twente Data Management Workshop (TDM 2004) on XML Databases and Information Retrieval, Enschede, The Netherlands, June 21, 2004*, CTIT Workshop Proceedings Series, pages 9–16. Centre for Telematics and Information Technology (CTIT), University of Twente, Enschede, The Netherlands, 2004.

[43] Jan Hidders and Philippe Michiels. Avoiding unnecessary ordering operations in XPath. In Georg Lausen and Dan Suciu, editors, *Database Programming Languages, 9th International Workshop, DBPL 2003, Potsdam, Germany, September 6-8, 2003, Revised Papers*, volume 2921 of *Lecture Notes in Computer Science*, pages 54–70. Springer, 2003.

[44] Jan Hidders, Philippe Michiels, Jan Paredaens, and Roel Vercammen. LiXQuery: a formal foundation for XQuery research. *SIGMOD Rec.*, 34(4):21–26, 2005.

[45] H. V. Jagadish, Rakesh Agrawal, and Linda Ness. A study of transitive closure as a recursion mechanism. *SIGMOD Rec.*, 16(3):331–344, 1987.

[46] Wim Janssen, Alexandr Korlyukov, and Jan Van den Bussche. On the tree-transformation power of XSLT. Technical report, University of Hasselt, 2006.

[47] Haifeng Jiang, Hongjun Lu, and Wei Wang. Efficient processing of XML twig queries with OR-predicates. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 59–70, New York, NY, USA, 2004. ACM.

[48] Thomas Johnsson. Attribute grammars as a functional programming paradigm. In *Functional Programming Languages and Computer Architecture*, pages 154–173. Springer-Verlag, 1987.

[49] R. Abdel Kader and M. van Keulen. Overview of query optimization in xml database systems. Technical Report TR-CTIT-07-39, Enschede, November 2007.

[50] Michael Kay. *XSL Transformations (XSLT) Version 2.0*. W3C, January 2007.

[51] Ming Li, Murali Mani, and Elke A. Rundensteiner. Efficiently loading and processing XML streams. In *IDEAS '08: Proceedings of the 2008 international symposium on Database engineering and applications*, pages 59–67, New York, NY, USA, 2008. ACM.

[52] Quanzhong Li and Bongki Moon. Indexing and querying XML data for regular path expressions. In *VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases*, pages 361–370, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[53] Xiaogang Li and Gagan Agrawal. Efficient evaluation of XQuery over streaming data. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 265–276. VLDB Endowment, 2005.

[54] Dongxi Liu, Zhenjiang Hu, and Masato Takeichi. Bidirectional interpretation of XQuery. In *PEPM '07: Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 21–30, New York, NY, USA, 2007. ACM.

[55] Zhen Hua Liu, Muralidhar Krishnaprasad, and Vikas Arora. Native XQuery processing in Oracle XMLDB. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 828–833, New York, NY, USA, 2005. ACM.

[56] Pavel Loupal. Evaluation of XQuery queries using lambda calculi. In Paolo Atzeni, Albertas Caplinskas, and Hannu Jaakkola, editors, *Advances in Databases and Information Systems, Proceedings of the 12th East European Conference, ADBIS 2008, September 5-9, 2008, Pori, Finland*, pages 180–183. Tampere University of Technology. Pori. Publication, 2008.

[57] Jiaheng Lu, Tok Wang Ling, Chee-Yong Chan, and Ting Chen. From region encoding to extended Dewey: on efficient processing of XML twig pattern matching. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 193–204. VLDB Endowment, 2005.

[58] Ashok Malhotra, Jim Melton, and Norman Walsh. *XQuery 1.0 and XPath 2.0 Functions and Operators*. W3C, January 2007.

[59] Christian Mathis and Theo Härder. Hash-based structural join algorithms. In Torsten Grust, Hagen Höpfner, Arantza Illarramendi, Stefan Jablonski, Marco Mesiti, Sascha Müller, Paula-Lavinia Patranjan, Kai-Uwe Sattler, Myra Spiliopoulou, and Jef Wijsen, editors, *Current Trends in Database Technology - EDBT 2006, EDBT 2006 Workshops PhD, DataX, IIDB, IIHA, ICSNW, QLQP, PIM, PaRMA, and Reactivity on the Web, Munich, Germany, March 26-31, 2006, Revised Selected Papers*, volume 4254 of *Lecture Notes in Computer Science*, pages 136–149. Springer, 2006.

[60] Norman May, Sven Helmer, and Guido Moerkotte. Strategies for query unnesting in XML databases. *ACM Trans. Database Syst.*, 31(3):968–1013, 2006.

[61] Wolfgang Meier. eXist: An open source native XML database. In Akmal B. Chaudhri, Mario Jeckle, Erhard Rahm, and Rainer Unland, editors, *Web, Web-Services, and Database Systems, NODe 2002 Web and Database-Related Workshops, Erfurt, Germany*, volume 2593 of *Lecture Notes in Computer Science*, pages 169–183. Springer, 2003.

[62] Irena Mlýnková and Jaroslav Pokorný. UserMap: an adaptive enhancing of user-driven XML-to-relational mapping strategies. In Alan Fekete and Xuemin Lin, editors, *Database Technologies 2008. Proceedings of the Nineteenth Australasian Database Conference, ADC 2008, January 22-25, 2008, Wollongang, NSW, Australia*, volume 75 of *CRPIT*, pages 165–174. Australian Computer Society, 2008.

[63] Irena Mlýnková, Kamil Toman, and Jaroslav Pokorný. Statistical Analysis of Real XML Data Collections. In *COMAD'06*, pages 20–31, New Delhi, India, 2006. Tata McGraw-Hill Publishing.

[64] Frank Neven, Thomas Schwentick, and Victor Vianu. Finite state machines for strings over infinite alphabets. *ACM Trans. Comput. Log*, 15:403–435, 2004.

[65] Matthias Nicola and Bert van der Linden. Native XML support in DB2 universal database. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 1164–1174. VLDB Endowment, 2005.

[66] Gundula Niemann and Friedrich Otto. The church-rosser languages are the deterministic variants of the growing context-sensitive languages. In *Proc. Foundations of software science and computation structures; Lecture notes in Computer Science*, pages 243–257. Springer-Verlag, 1998.

[67] Ruhsan Onder and Zeki Bayram. Xslt version 2.0 is turing-complete: A purely transformation based proof. In Oscar H. Ibarra and Hsu-Chun Yen, editors, *CIAA*, volume 4094 of *Lecture Notes in Computer Science*, pages 275–276. Springer, 2006.

[68] Shankar Pal, Istvan Cseri, Oliver Seeliger, Michael Rys, Gideon Schaller, Wei Yu, Dragan Tomic, Adrian Baras, Brandon Berg, Denis Churin, and Eugene Kogan. XQuery implementation in a relational database system. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 1175–1186. VLDB Endowment, 2005.

[69] Stelios Paparizos, Yuqing Wu, Laks V. S. Lakshmanan, and H. V. Jagadish. Tree logical classes for efficient evaluation of XQuery. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 71–82, New York, NY, USA, 2004. ACM.

[70] Venkatesh Raghavan, Kurt Deschler, and Elke A. Rundensteiner. VAMANA - a scalable cost-driven XPath engine. In *ICDEW '05: Proceedings of the 21st International Conference on Data Engineering Workshops*, page 1278, Washington, DC, USA, 2005. IEEE Computer Society.

[71] Christopher Re, Jerome Simeon, and Mary Fernandez. A complete and efficient algebraic compiler for xquery. *Data Engineering, International Conference on*, 0:14, 2006.

[72] Stephen Richardson and Mahadevan Ganapathi. Interprocedural analysis vs. procedure integration. *Inf. Process. Lett.*, 32(3):137–142, 1989.

[73] Jan Rittinger, Jens Teubner, and Torsten Grust. Pathfinder: A relational query optimizer explores XQuery terrain. In Alfons Kemper, Harald Schöning, Thomas Rose, Matthias Jarke, Thomas Seidl, Christoph Quix, and Christoph Brochhaus, editors, *Datenbanksysteme in Business, Technologie und Web (BTW 2007), 12. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), Proceedings, 7.-9. März 2007, Aachen, Germany*, volume 103 of *LNI*, pages 617–620. GI, 2007.

[74] Karsten Schmidt and Theo Härder. Usage-driven storage structures for native xml databases. In *IDEAS '08: Proceedings of the 2008 international symposium on Database engineering & applications*, pages 169–178, New York, NY, USA, 2008. ACM.

[75] Ming-Chien Shan and Marie-Anne Neimat. Optimization of relational algebra expressions containing recursion operators. In *CSC '91: Proceedings of the 19th annual conference on Computer Science*, pages 332–341, New York, NY, USA, 1991. ACM.

[76] Hong Su, Elke A. Rundensteiner, and Murali Mani. Automaton in or out: run-time plan optimization for XML stream processing. In *SSPS '08: Proceedings of the 2nd international workshop on Scalable stream processing system*, pages 38–47, New York, NY, USA, 2008. ACM.

[77] Igor Tatarinov, Stratis D. Viglas, Kevin Beyer, Jayavel Shanmugasundaram, Eugene Shekita, and Chun Zhang. Storing and querying ordered XML using a relational database system. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 204–215, New York, NY, USA, 2002. ACM.

[78] Jens Teubner, Torsten Grust, Sebastian Maneth, and Sherif Sakr. Dependable cardinality forecasts for XQuery. *PVLDB*, 1(1):463–477, 2008.

[79] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. *XML Schema Part 1: Structures Second Edition.* W3C, October 2004.

[80] Nicolas Travers, Tuyet-Tram Dang-Ngoc, and Tianxiao Liu. Untyped XQuery canonization. In Kevin Chen-Chuan Chang, Wei Wang, Lei Chen, Clarence A. Ellis, Ching-Hsien Hsu, Ah Chung Tsoi, and Haixun

Wang, editors, *Advances in Web and Network Technologies, and Information Management, APWeb/WAIM 2007 International Workshops: DBMAN 2007, WebETrends 2007, PAIS 2007 and ASWAN 2007, Huang Shan, China, June 16-18, 2007, Proceedings*, volume 4537 of *Lecture Notes in Computer Science*, pages 358–371. Springer, 2007.

[81] Ondřej Vošta, Irena Mlýnková, and Jaroslav Pokorný. Even an ant can create an XSD. In Jayant R. Haritsa, Kotagiri Ramamohanarao, and Vikram Pudi, editors, *Database Systems for Advanced Applications, 13th International Conference, DASFAA 2008, New Delhi, India, March 19-21, 2008. Proceedings*, volume 4947 of *Lecture Notes in Computer Science*, pages 35–50. Springer, 2008.

[82] W3C. *XML Query Test Suite*, November 2006.

[83] Song Wang, E.A. Rundensteiner, and M. Mani. Optimization of nested xquery expressions with orderby clauses. *Data Engineering Workshops, 2005. 21st International Conference on*, pages 1277–1277, April 2005.

[84] Weining Zhang, Clement T. Yu, and Daniel Troy. Necessary and sufficient conditions to linearize doubly recursive programs in logic databases. *ACM Trans. Database Syst.*, 15(3):459–482, 1990.

[85] Xin Zhang, Bradford Pielech, and Elke A. Rundesnteiner. Honey, I shrunk the XQuery!: an XML algebra optimization approach. In *WIDM '02: Proceedings of the 4th international workshop on Web information and data management*, pages 15–22, New York, NY, USA, 2002. ACM.

[86] Xin Zhang, Xin Zhang, Elke A. Rundensteiner, and Elke A. Rundensteiner. XAT: XML algebra for the Rainbow system. Technical report, 2002.

[87] Ying Zhang and Peter Boncz. XRPC: interoperable and efficient distributed XQuery. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 99–110. VLDB Endowment, 2007.

[88] Kristopher William Zyp. Ajax performance analysis. Technical report, IBM, 2008.

Index of symbols

Appendix A

Transcription rules

This section summarizes the most important transcription rules for the canonical and the EBV modes whose principles were explained in the Chapter 7.

A.1 Unary operators

Syntax

$E_0 ::= \ op \ \ E_1$

All unary operators share the following rules:

Invocation rules

$\mathsf{invc}[E_1] := \mathsf{invc}[E_0]$

$\mathsf{invv}[E_1] := \mathsf{invv}[E_0]$

Environment rules

$\mathsf{invenv}[E_1] := \mathsf{invenv}[E_0]$

$\mathsf{exenv}[E_0] := \mathsf{exenv}[E_1]$

Variable rules

$\mathsf{varseqn}[E_1, \$\mathtt{x}] := \mathsf{varseqn}[E_0, \$\mathtt{x}]$

$\mathsf{varseqa}[E_1, \$\mathtt{x}] := \mathsf{varseqa}[E_0, \$\mathtt{x}]$

Boolean negation

Syntax

E_0 ::= `fn:not(` E_1 `)`

Core rules

$\mathsf{exebvt}[E_0] := \mathsf{exebvf}[E_1]$

$\mathsf{exebvf}[E_0] := \mathsf{exebvt}[E_1]$

A.2 Binary operators

Syntax

$E_0 ::= E_1 \quad op \quad E_2$

All binary operators share the following rules:

Invocation rules

$\mathsf{invc}[E_1] := \mathsf{invc}[E_0]$

$\mathsf{invc}[E_2] := \mathsf{invc}[E_0]$

$\mathsf{invv}[E_1] := \mathsf{invv}[E_0]$

$\mathsf{invv}[E_2] := \mathsf{invv}[E_0]$

Environment rules

$\mathsf{invenv}[E_1] := \mathsf{invenv}[E_0]$

$\mathsf{invenv}[E_2] := \mathsf{invenv}[E_0]$

$\mathsf{exenv}[E_0] := (\mathsf{exenv}[E_1] \ \cup \ \mathsf{exenv}[E_2])$

Variable rules

$\mathsf{varseqn}[E_1, \$\mathtt{x}] := \mathsf{varseqn}[E_0, \$\mathtt{x}]$

$\mathsf{varseqa}[E_1, \$\mathtt{x}] := \mathsf{varseqa}[E_0, \$\mathtt{x}]$

$\mathsf{varseqn}[E_2, \$\mathtt{x}] := \mathsf{varseqn}[E_0, \$\mathtt{x}]$

$\mathsf{varseqa}[E_2, \$\mathtt{x}] := \mathsf{varseqa}[E_0, \$\mathtt{x}]$

Note that whenever the two environments $\mathsf{exenv}[E_1]$ and $\mathsf{exenv}[E_2]$ contain the same tree identifier ti, the corresponding tree information is merged via set-union. Since the tree identifier exactly determines the context in which the tree was created, trees having the same identifier must be identical; therefore, applying set-union to tree environments do not alter them anyway.

Concatenation

Syntax

$E_0 ::= E_1 \ \mathtt{,} \ E_2$

Core rules

$\mathsf{exseqa}[E_0] := \pi[si/r](\pi[r := 1.si]\ (\mathsf{exseqa}[E_1])\ \cup\ \pi[r := 2.si]\ (\mathsf{exseqa}[E_2]))$

$\mathsf{exseqn}[E_0] := \pi[si/r](\pi[r := 1.si]\ (\mathsf{exseqn}[E_1])\ \cup\ \pi[r := 2.si]\ (\mathsf{exseqn}[E_2]))$

Node-set union

Syntax

$E_0 ::= E_1\ \mathtt{union}\ E_2$

Core rules

$\mathsf{exseqa}[E_0] := \emptyset$

$\mathsf{exseqn}[E_0] := \mathit{docorder}\ (\delta\pi[\backslash si]\ (\mathsf{exseqn}[E_1])\ \cup\ \delta\pi[\backslash si]\ (\mathsf{exseqn}[E_2]))$

Node-set intersection

Syntax

$E_0 ::= E_1\ \mathtt{union}\ E_2$

Core rules

$\mathsf{exseqa}[E_0] := \emptyset$

$\mathsf{exseqn}[E_0] := \mathit{docorder}\ (\delta\pi[\backslash si]\ (\mathsf{exseqn}[E_1])\ \bowtie\ \delta\pi[\backslash si]\ (\mathsf{exseqn}[E_2]))$

Node-set difference

Syntax

$E_0 ::= E_1\ \texttt{except}\ E_2$

Core rules

$\mathsf{exseqa}[E_0] := \emptyset$

$\mathsf{exseqn}[E_0] := \mathit{docorder}\ (\delta\pi[\backslash si]\ (\mathsf{exseqn}[E_1])\ \setminus\ \delta\pi[\backslash si]\ (\mathsf{exseqn}[E_2]))$

Value comparisons

Syntax

$E_0 ::= E_1\ \mathit{op}\ E_2$

$\mathit{op} \in \{\texttt{eq}, \texttt{ne}, \texttt{lt}, \texttt{le}, \texttt{gt}, \texttt{ge}\}$

Core rules

$\mathsf{tmp}_1 := \delta\pi[\backslash si]\ \pi[a/av]\ \mathsf{exseqa}[E_1]$

$\mathsf{tmp}_2 := \delta\pi[\backslash si]\ \pi[b/av]\ \mathsf{exseqa}[E_2]$

$\mathsf{tmp}_3 := \delta\pi[\backslash a, b]\ \sigma[P(a,b)]\ (\mathsf{tmp}_1\ \bowtie\ \mathsf{tmp}_2)$

$\mathsf{exebvt}[E_0] := \mathsf{tmp}_3$

$\mathsf{exebvf}[E_0] := (\mathsf{invv}[E_0]\ \setminus\ \mathsf{tmp}_3)$

Here P is the binary predicate corresponding to the operator *op*.

Boolean `and`

Syntax

$E_0 ::= E_1\ \texttt{and}\ E_2$

Core rules

$\mathsf{exebvt}[E_0] := (\mathsf{exebvt}[E_1] \ \bowtie \ \mathsf{exebvt}[E_2])$

$\mathsf{exebvf}[E_0] := (\mathsf{exebvf}[E_1] \ \cup \ \mathsf{exebvf}[E_2])$

Boolean `or`

Syntax

$E_0 ::= E_1$ `or` E_2

Core rules

$\mathsf{exebvt}[E_0] := (\mathsf{exebvt}[E_1] \ \cup \ \mathsf{exebvt}[E_2])$

$\mathsf{exebvf}[E_0] := (\mathsf{exebvf}[E_1] \ \bowtie \ \mathsf{exebvf}[E_2])$

A.3 Node construction

Syntax

$E_0 ::=$ `<e>{` E_1 `}</e>`

Core rules

$\mathsf{exseqn}[E_0] := \pi[si := \lambda, ti := \alpha(cs).\alpha(vs), ni := \lambda](\mathsf{invv}[E_0])$

Environment rules

$\mathsf{tmp}_1 := \pi[ni := \lambda, nk := \texttt{element}, nn := \mathsf{e}, nv := \lambda](\mathsf{invv}[E_0])$

$\mathsf{tmp}_2 := \mathit{normalize}(\mathit{mkforest}(\mathsf{exseqn}[E_1], \mathsf{invenv}[E_1], \mathsf{exenv}[E_1], \mathsf{exseqa}[E_1]))$

$\mathsf{exenv}[E_0] := \pi[ti := \alpha(cs).\alpha(vs)] \ ((\mathsf{invc}[E_0]) \ \bowtie \ (\mathsf{tmp}_1 \ \cup \ \mathsf{tmp}_2))$

Functions

$\mathit{mkforest}(x, y, z, u) = (\mathit{envcut}(\mathit{envfilter}(x, (y \ \cup \ z))) \ \cup \ \mathit{envstring}(u))$

$\mathit{envfilter}(x, y) = \sigma[\mathit{prefix}(p, q)] \ (\ \pi[p/ni] \ (x) \ \bowtie \ \pi[q/ni] \ (y))$

$\mathit{envcut}(x) = \delta\pi[\backslash p, q, si, ti] \ \pi[ni := \alpha(si).\mathit{after}(q, p)] \ (x)$

$$envstring(x) = \delta\pi[\backslash si, av]\ \pi[ni := \alpha(si), nk := \texttt{string}, nv := av]\ (x)$$

$$normalize(x) = normalizeT(normalizeS(x))$$

$$normalizeS(x) = (\pi[nk := \texttt{text}]\ rpack\ groupS\ runpack(x)\ \cup \\ \sigma[nk \neq \texttt{string}](x))$$

$$normalizeT(x) = (\pi[nk := \texttt{text}]\ rpack\ groupT\ runpack(x)\ \cup \\ \sigma[nk \neq \texttt{text}](x))$$

$$runpack(x) = \delta\pi[\backslash ni]\ \pi[r := rtrim(ni), s := last(ni)]\ (x)$$

$$rpack(x) = \delta\pi[\backslash r, s]\ \pi[ni := r.s]\ (x)$$

$$groupS(x) = \xi[vs, r, nk = \texttt{string}, nv := \texttt{catd}(s, nv)]\ (x)$$

$$groupT(x) = \xi[vs, r, nk = \texttt{text}, nv := \texttt{cat}(s, nv)]\ (x)$$

A.4 Navigation

Syntax

$$E_0 ::= E_1\ \texttt{/}\ axis\texttt{::*}$$

Core rules

$$\mathsf{tmp}_1 := (\mathsf{invenv}[E_1]\ \cup\ \mathsf{exenv}[E_1])$$

$$\mathsf{tmp}_2 := \sigma[P(q, ni, nk)](\mathsf{tmp}_1\ \bowtie\ \pi[q/ni]\ (\mathsf{exseqn}[E_1]))$$

$$\mathsf{exseqn}[E_0] = docorder\ \delta\pi[\backslash nk, nn, nv, q, si]\ \mathsf{tmp}_2$$

Environment rules

$$\mathsf{invenv}[E_1] = \mathsf{invenv}[E_0]$$

$$\mathsf{exenv}[E_0] = \mathsf{exenv}[E_1]$$

The selection operator is driven by a predicate P applied to node identifiers q (a node from the sequence) and ni (a node from its tree environment); additionally, the node kind nk may be constrained. The predicate is selected according to the *axis* used in the navigation operator as shown in the table A.1. (See the Sec. 4.2 for the definitions of the operators used here.)

axis	P
`ancestor`	$prefix(ni, q) \wedge q \neq ni$
`ancestor-or-self`	$prefix(ni, q)$
`attribute`	$q = rtrim(ni) \wedge nk = \texttt{attribute}$
`child`	$q = rtrim(ni) \wedge nk \neq \texttt{attribute}$
`descendant`	$prefix(q, ni) \wedge q \neq ni \wedge nk \neq \texttt{attribute}$
`descendant-or-self`	$prefix(q, ni) \wedge nk \neq \texttt{attribute}$
`following`	$ni > q \wedge \neg prefix(q, ni) \wedge nk \neq \texttt{attribute}$
`following-sibling`	$rtrim(ni) = rtrim(q) \wedge ni > q \wedge nk \neq \texttt{attribute}$
`parent`	$ni = rtrim(q)$
`preceding`	$ni < q \wedge \neg prefix(ni, q) \wedge nk \neq \texttt{attribute}$
`preceding-sibling`	$rtrim(ni) = rtrim(q) \wedge ni < q \wedge nk \neq \texttt{attribute}$
`self`	$ni = q$

Table A.1: XPath axes and their corresponding predicates

A.5 FLWOR expression

Let clause

Syntax

$E_0 ::= \texttt{for \$y in } E_1 \texttt{ return } E_2$

Invocation rules

$\mathsf{invc}[E_1] := \mathsf{invc}[E_0]$

$\mathsf{invc}[E_2] := \mathsf{invc}[E_0]$

$\mathsf{invv}[E_1] := \mathsf{invv}[E_0]$

$\mathsf{invv}[E_2] := \mathsf{invv}[E_0]$

Core rules

$\mathsf{varseqn}[E_2, \$\mathsf{y}] := \mathsf{exseqn}[E_1]$

$\mathsf{varseqa}[E_2, \$\mathsf{y}] := \mathsf{exseqa}[E_1]$

$\mathsf{exseqn}[E_0] := \mathsf{exseqn}[E_2]$

$\mathsf{exseqa}[E_0] := \mathsf{exseqa}[E_2]$

Environment rules

$\mathsf{invenv}[E_1] := \mathsf{invenv}[E_0]$

$\mathsf{invenv}[E_2] := \mathsf{invenv}[E_0] \cup \mathsf{exenv}[E_1]$

$\mathsf{exenv}[E_0] := \mathsf{exenv}[E_1] \cup \mathsf{exenv}[E_2]$

Variable rules

$\mathsf{varseqn}[E_1, \texttt{\$x}] := \mathsf{varseqn}[E_0, \texttt{\$x}]$

$\mathsf{varseqa}[E_1, \texttt{\$x}] := \mathsf{varseqa}[E_0, \texttt{\$x}]$

$\mathsf{varseqn}[E_2, \texttt{\$x}] := \mathsf{varseqn}[E_0, \texttt{\$x}]$

$\mathsf{varseqa}[E_2, \texttt{\$x}] := \mathsf{varseqa}[E_0, \texttt{\$x}]$

For clause

Syntax

$E_0 ::= \texttt{for \$y in } E_1 \texttt{ return } E_2$

Invocation rules

$\mathsf{invc}[E_1] := \mathsf{invc}[E_0]$

$\mathsf{invc}[E_2] := \mathsf{invc}[E_0]$

$\mathsf{invv}[E_1] := \mathsf{invv}[E_0]$

$\mathsf{tmp}_1 := \pi[vs_I := vs.\alpha(si_O)]\ \pi[si_O/si]\ \mathsf{exseqn}[E_1]$

$\mathsf{tmp}_2 := \pi[vs_I := vs.\alpha(si_O)]\ \pi[si_O/si]\ \mathsf{exseqa}[E_1]$

$\mathsf{tmp}_3 := (\delta\pi[\backslash ti, ni]\ \mathsf{tmp}_1\ \cup\ \delta\pi[\backslash av]\ \mathsf{tmp}_2)$

$\mathsf{invv}[E_2] := \pi[vs/vs_I]\ \delta\pi[\backslash vs, si_O]\ \mathsf{tmp}_3$

Core rules

$\mathsf{varseqn}[E_2, \$\mathtt{y}] := \pi[si := \lambda]\ \pi[vs/vs_I]\ \delta\pi[\backslash vs, si_O]\ \mathsf{tmp}_1$

$\mathsf{varseqa}[E_2, \$\mathtt{y}] := \pi[si := \lambda]\ \pi[vs/vs_I]\ \delta\pi[\backslash vs, si_O]\ \mathsf{tmp}_2$

$\mathsf{tmp}_4 := \pi[si_I/si, vs_I/vs]\ \mathsf{exseqn}[E_2]$

$\mathsf{tmp}_5 := \pi[si_I/si, vs_I/vs]\ \mathsf{exseqa}[E_2]$

$\mathsf{exseqn}[E_0] := \delta\pi[\backslash vs_I, si_O, si_I]\ \pi[si := si_O.si_I]\ (\mathsf{tmp}_4 \bowtie \mathsf{tmp}_3)$

$\mathsf{exseqa}[E_0] := \delta\pi[\backslash vs_I, si_O, si_I]\ \pi[si := si_O.si_I]\ (\mathsf{tmp}_5 \bowtie \mathsf{tmp}_3)$

Environment rules

$\mathsf{invenv}[E_1] := \mathsf{invenv}[E_0]$

$\mathsf{invenv}[E_2] := \mathsf{invenv}[E_0] \cup \mathsf{exenv}[E_1]$

$\mathsf{exenv}[E_0] := \mathsf{exenv}[E_1] \cup \mathsf{exenv}[E_2]$

Variable rules

$\mathsf{varseqn}[E_1, \$\mathtt{x}] := \mathsf{varseqn}[E_0, \$\mathtt{x}]$

$\mathsf{varseqa}[E_1, \$\mathtt{x}] := \mathsf{varseqa}[E_0, \$\mathtt{x}]$

$\mathsf{varseqn}[E_2, \$\mathtt{x}] := \pi[vs/vs_I]\ \delta\pi[\backslash vs]((\mathsf{varseqn}[E_0, \$\mathtt{x}]) \bowtie \delta\pi[\backslash si_O]\ \mathsf{tmp}_3)$

$\mathsf{varseqa}[E_2, \$\mathtt{x}] := \pi[vs/vs_I]\ \delta\pi[\backslash vs]((\mathsf{varseqa}[E_0, \$\mathtt{x}]) \bowtie \delta\pi[\backslash si_O]\ \mathsf{tmp}_3)$

Order-by clause

Syntax

$E_0 ::= \mathtt{for\ \$y\ in}\ E_1\ \mathtt{order\ by}\ E_3\ \mathtt{return}\ E_2$

Core rules

$\mathsf{tmp}_6 := (\delta\pi[\backslash si]\ \pi[vs_I/vs]\ \mathsf{exseqa}[E_3] \bowtie \mathsf{tmp}_3)$

$\mathsf{exseqn}[E_0] := \delta\pi[\backslash vs_I, si_O, si_I]\ \pi[si := \alpha(av).si_O.si_I]\ \ (\mathsf{tmp}_4 \bowtie \mathsf{tmp}_6)$

$\mathsf{exseqa}[E_0] := \delta\pi[\backslash vs_I, si_O, si_I]\ \pi[si := \alpha(av).si_O.si_I]\ \ (\mathsf{tmp}_5 \bowtie \mathsf{tmp}_6)$

The other equations are identical to the equations of the plain for-expression. Note that in the case of multi-variable for-expression with order-by clause, the $\mathsf{exseqn}[E_0]$ equation shall handle all the control variables at once.

Where clause

Syntax

E_0 ::= `where` E_1 `return` E_2

Invocation rules

$\mathsf{invc}[E_1] := \mathsf{invc}[E_0]$

$\mathsf{invc}[E_2] := \mathsf{invc}[E_0]$

$\mathsf{invv}[E_1] := \mathsf{invv}[E_0]$

$\mathsf{invv}[E_2] := \mathsf{exebvt}[E_1]$

Environment rules

$\mathsf{invenv}[E_1] := \mathsf{invenv}[E_0]$

$\mathsf{invenv}[E_2] := \mathsf{invenv}[E_0]$

$\mathsf{exenv}[E_0] := (\mathsf{exenv}[E_1] \ \cup \ \mathsf{exenv}[E_2])$

Variable rules

$\mathsf{varseqn}[E_1, \$\mathtt{x}] := \mathsf{varseqn}[E_0, \$\mathtt{x}]$

$\mathsf{varseqa}[E_1, \$\mathtt{x}] := \mathsf{varseqa}[E_0, \$\mathtt{x}]$

$\mathsf{varseqn}[E_2, \$\mathtt{x}] := (\mathsf{varseqn}[E_0, \$\mathtt{x}] \ \bowtie \ \mathsf{exebvt}[E_1])$

$\mathsf{varseqa}[E_2, \$\mathtt{x}] := (\mathsf{varseqa}[E_0, \$\mathtt{x}] \ \bowtie \ \mathsf{exebvt}[E_1])$

A.6 Quantified expressions

Syntax

E_0 ::= `[some|every] $y in` E_1 `satisfies` E_2

The quantified expressions share the following rules:

Invocation rules

$\mathsf{invc}[E_1] := \mathsf{invc}[E_0]$

$\mathsf{invc}[E_2] := \mathsf{invc}[E_0]$

$\mathsf{invv}[E_1] := \mathsf{invv}[E_0]$

$\mathsf{tmp}_1 := \pi[vs_I := vs.\alpha(si_O)]\ \pi[si_O/si]\ \mathsf{exseqn}[E_1]$

$\mathsf{tmp}_2 := \pi[vs_I := vs.\alpha(si_O)]\ \pi[si_O/si]\ \mathsf{exseqa}[E_1]$

$\mathsf{tmp}_3 := (\delta\pi[\backslash ti, ni]\ \mathsf{tmp}_1\ \cup\ \delta\pi[\backslash av]\ \mathsf{tmp}_2)$

$\mathsf{invv}[E_2] := \pi[vs/vs_I]\ \delta\pi[\backslash vs, si_O]\ \mathsf{tmp}_3$

Core rules

$\mathsf{varseqn}[E_2, \$\mathsf{y}] := \pi[si := \lambda]\ \pi[vs/vs_I]\ \delta\pi[\backslash vs, si_O]\ \mathsf{tmp}_1$

$\mathsf{varseqa}[E_2, \$\mathsf{y}] := \pi[si := \lambda]\ \pi[vs/vs_I]\ \delta\pi[\backslash vs, si_O]\ \mathsf{tmp}_2$

Environment rules

$\mathsf{invenv}[E_1] := \mathsf{invenv}[E_0]$

$\mathsf{invenv}[E_2] := \mathsf{invenv}[E_0] \cup \mathsf{exenv}[E_1]$

$\mathsf{exenv}[E_0] := \emptyset$

Variable rules

$\mathsf{varseqn}[E_1, \$\mathtt{x}] := \mathsf{varseqn}[E_0, \$\mathtt{x}]$

$\mathsf{varseqa}[E_1, \$\mathtt{x}] := \mathsf{varseqa}[E_0, \$\mathtt{x}]$

$\mathsf{varseqn}[E_2, \$\mathtt{x}] := \pi[vs/vs_I]\ \delta\pi[\backslash vs]((\mathsf{varseqn}[E_0, \$\mathtt{x}])\ \bowtie\ \delta\pi[\backslash si_O]\ \mathsf{tmp}_3)$

$\mathsf{varseqa}[E_2, \$\mathtt{x}] := \pi[vs/vs_I]\ \delta\pi[\backslash vs]((\mathsf{varseqa}[E_0, \$\mathtt{x}])\ \bowtie\ \delta\pi[\backslash si_O]\ \mathsf{tmp}_3)$

some-expression

Syntax

$E_0 ::= \mathtt{some}\ \$\mathtt{y}\ \mathtt{in}\ E_1\ \mathtt{satisfies}\ E_2$

Core rules

$\mathsf{tmp}_4 = \delta\pi[\backslash vs_I]\ (\pi[vs_I/vs]\mathsf{exebvt}[E_2]\ \bowtie\ \delta\pi[\backslash si]\ \mathsf{tmp}_3)$

$\mathsf{exebvt}[E_0] = \mathsf{tmp}_4$

$\mathsf{exebvf}[E_0] = (\mathsf{invv}[E_0]\ \setminus\ \mathsf{tmp}_4)$

`every`-expression

Syntax

$E_0 ::= \texttt{every \$y in}\ E_1\ \texttt{satisfies}\ E_2$

Core rules

$\mathsf{tmp}_4 = \delta\pi[\backslash vs_I]\ (\pi[vs_I/vs]\mathsf{exebvf}[E_2]\ \bowtie\ \delta\pi[\backslash si]\ \mathsf{tmp}_3)$

$\mathsf{exebvt}[E_0] = (\mathsf{invv}[E_0]\ \setminus\ \mathsf{tmp}_4)$

$\mathsf{exebvf}[E_0] = \mathsf{tmp}_4$