Chapter 1

Overview of OXPATH

This chapter is provided as a standalone detailed summary of the OXPath formalism. Those interested in further details are referred to [2].

1.1 Web Scale Extraction with OXPATH

OXPATH is a formalism for automating web extraction tasks, capable of scaling to the size of the web. A properly specified OXPATH expression outputs *structured web objects*.

Extracting and aggregating web information is not a new challenge. Previous approaches, in the overwhelming majority, either (1) require service providers to deliver their data in a structured fashion (e.g. the Semantic Web); or, (2) "wrap" unstructured information sources to extract and aggregate relevant data. The first case levies requirements that service providers have little incentive to adopt, which leaves us with wrapping. Wrapping a website, however, is often tedious, since many AJAX-enabled web applications reveal the relevant data only through user interactions. Previous work does not adequately address web page scripting. Even when scripting is considered, the simulation of user actions is neither declarative nor succinct, but rather relies on imperative scripts.

1.2 Core Language

OXPATH extends XPath 1.0 [1] with four conceptual extensions: Actions to navigate the user interface of web applications, exposure to rendered visual information, extraction markers to specify data to extract, and the Kleene star to facilitate iteration over a set of pages with an unknown extent. With these extensions, we have a language suitable for extracting data from modern web applications.

Actions For simulating user actions such as clicks or mouse-overs, OXPATH introduces *contextual*, as in `{click}`, and *absolute action steps* with a trailing slash, as in `{click /}`. Since actions may modify or replace the DOM, we assume that they always return a new DOM. Absolute actions return DOM roots, contextual actions return the nodes in the new DOM matched by the *action-free prefix* of the performed action, which is obtained from the segment starting at the previous absolute action by dropping all intermediate contextual actions and extraction markers.

Style Axis and Visible Field Access We introduce two extensions for lightweight visual navigation: a new axis for accessing CSS DOM node properties and a new node test for selecting only visible form fields. The `style` axis navigates the actual CSS properties as returned by the DOM *style* object. For example, it is possible to select nodes based on their (rendered) color or font size.

To ease field navigation, OXPATH introduces the node-test `field()`, which relies on the `style` axis to access the computed CSS style to exclude fields that are not visible, e.g., `/descendant::field()[1]` selects the first visible field in document order.

Extraction Marker In OXPATH, we introduce a new kind of qualifier, the *extraction marker*, to identify nodes as representatives for records as well as to form attributes for these records. For example,

```
doc("http://news.google.com")//div[contains(@class,'story')]
    [.//h2]:<story>[.//h2:<title=string(.)>]
    [.//span[contains(@class,'source')]:<source=string(.)>]
```

extracts a `story` element for each current story on Google News, along with its title and sources (as strings), producing:

```
<story><title >Tax cuts ...</title>
    <source>Washington Post</source>
    <source>Wall Street Journal</source> ...  </story>
```

The nesting in the result mirrors the structure of the OXPATH expression: extraction markers in a predicate (`title`, `source`) represent attributes to the last marker outside the predicate (`story`).

Kleene Star Finally, we add the Kleene star, as in [4]. For example, the following expression queries Google for "Oxford", traverses all accessible result pages and extracts all links.

```
doc("google.com")/descendant::field()[1]/{"Oxford"}
                 /following::field()[1]/{click /}
      /( /descendant::a:<Link=(@href)>[.#="Next"]/{click /} )*
```

To limit the range of the Kleene star, one can specify upper and lower bounds on the multiplicity, e.g., `(...)*{3,8}`.

1.3 OX LATIN: A Host Language for OXPATH

We have deliberately kept the features in the core language minimal so that we can make strict guarantees of time and memory requirements, which we discuss in Section 1.5. Essential features of a web data extraction formalism also include variable binding so that a single wrapper can specify multiple input sets for common paths as well as support for aggregation, query composition, and integration based on the output of an OXPath expression. These capabilities are delivered by embedding OXPATH into host languages. Like many successful web frameworks such as HTML and CSS, OXPATH is designed to be a supporting part of other languages in order to serve the needs of both current and future development communities.

We have designed and implemented two OXPATH host languages. The first host language is straightforward and comes in the form of Java libraries and an API supporting OXPATH expression evaluation. This API provides the means for instantiating input variables in OXPath expressions and evaluating them with several supported browsers. The API provides exposure to the expression's output, either as a stream or a block, so that a Java program can read and manipulate the output for aggregation, integration with the output of other expressions, or use the output to compose other OXPATH expressions.

We also provide a declarative host language for OXPATH, designed specifically to evaluate expressions in an elastic computing environment (cloud): OX LATIN.

OX LATIN is an extraction language and platform that combines OXPATH and Pig Latin [5] to enable large-scale, multi-domain web extraction. **(1)** As a wrapper language, it extends OXPATH to capture all common extraction scenarios such as aggregation, query composition, and integration. Thanks to OXPATH's browser-backed engine, OX LATIN can deal even with heavily scripted web sites. **(2)** As an extraction platform, it employs Pig Latin to automatically distribute a set of wrappers. Using Pig Latin allows it to be easily deployed on any off-the-shelf Hadoop cluster. **(3)** OX LATIN decomposes complex wrappers at multi-way navigation points into ready-to-distribute wraplets by extending OXPATH with continuations for (navigation) actions.

OX LATIN integrates OXPATH and Pig Latin by wrapping OXPATH and adding primitives for manipulating labeled, unordered trees to match the hierarchical data model underlying the extraction semantics of OXPATH. The syntax of OX LATIN contains all original language constructs of Pig Latin as introduced in [5].

Expanding Pig Latin, we provide two additional expression types: **(1)** *oxpath expressions* take a bag of tuples, where each tuple describes a single OXPATH evaluation with bindings for the variables occurring in the OXPATH expression. As a result, a bag with the individual evaluation outcomes is returned. **(2)** OX LATIN allows *path expressions* for navigating tree structures within predicates generated by OXPATH expressions. These path expressions allow specification of regular patterns over tree paths, to obtain the bag with all tuples reachable via a path that matches the regular expression. The Kleene star is only applicable to individual steps but not to general path snippets (to avoid, for example, even queries). For example $t._*.room$ obtains all rooms which are hierarchically contained in t, or $t.pair.\$0$ obtains all first elements of a bag of pairs.

Without loss of generality, we normalize OX LATIN programs and assume that **`LOAD`** and **`STORE`** statements always feature explicit **`USING`** clauses, and that tuple attributes are only accessed with an explicit reference to the tuple itself, i.e., by a full path expression. For example, we rewrite **`FOREACH`** `tuples` **`GENERATE`** `attribute` as **`FOREACH`** `tuples` **`GENERATE`** `tuples.attribute`. Furthermore, we allow schemata in all assignments to replace derived tuple attribute names with more suitable names and to assert (redundant) type assumptions.

This normalization is necessary for the completion of the path expressions.

We formally define the data model, semantics, and complexity of program evaluation for OX LATIN. As an extension of Pig Latin, we are thus contributing the first formal specification of Pig Latin to the best of our knowledge.

1.4 Example Expressions

The majority of OXPATH notation is familiar to XPath users. In the previous section, we carefully extend XPath to achieve desired automation and extraction features. Consider now a full expression, shown in Figure 1.1. In this example, we define an OXPATH expression that extracts prices of properties from `rightmove.co.uk`. We begin in line 1 by retrieving the first HTML page via the `doc(url)` function, which OXPATH borrows from XQuery. We continue through line 3 by *simulating* user action for many different form input fields, spread over multiple HTML pages. Note here that OXPATH allows the use of the CSS selectors `#` and `.`, which select nodes based on their `id` and `class` attributes, respectively. Line 4 uses the Kleene star to specify extraction from all possible result pages,

which are traversed by clicking on hyperlinks containing the word "next". Finally, line 5 identifies all relevant properties and extracts their corresponding prices. This example could be extended to incapsulate all attributes relevant to each found web object, which in this example are all rentable properties from this site that satisfy our criteria.

We can then rapidly parallelize this wrapper using OX LATIN. Suppose instead of explicitly binding the values of our area and the number of bedrooms in Figure 1.1 to "Oxford" and "2", respectively, let them be input variables `$area` and `$beds`. We can then use the following OX LATIN program to run all instantiations of these variables from a given file, called `inputs.txt`, in parallel. The resulting OX LATIN program, omitting **`REGISTER`** commands for brevity, can be succinctly written as:

```
DEFINE RightMoveOXPath
    uk.ac.ox.comlab.diadem.oxpath.oxlatin.OXPath('rightmove.oxp');
input = LOAD 'inputs.txt' AS (area,beds);
results = FOREACH input GENERATE RightMoveOXPath(area,beds);
```

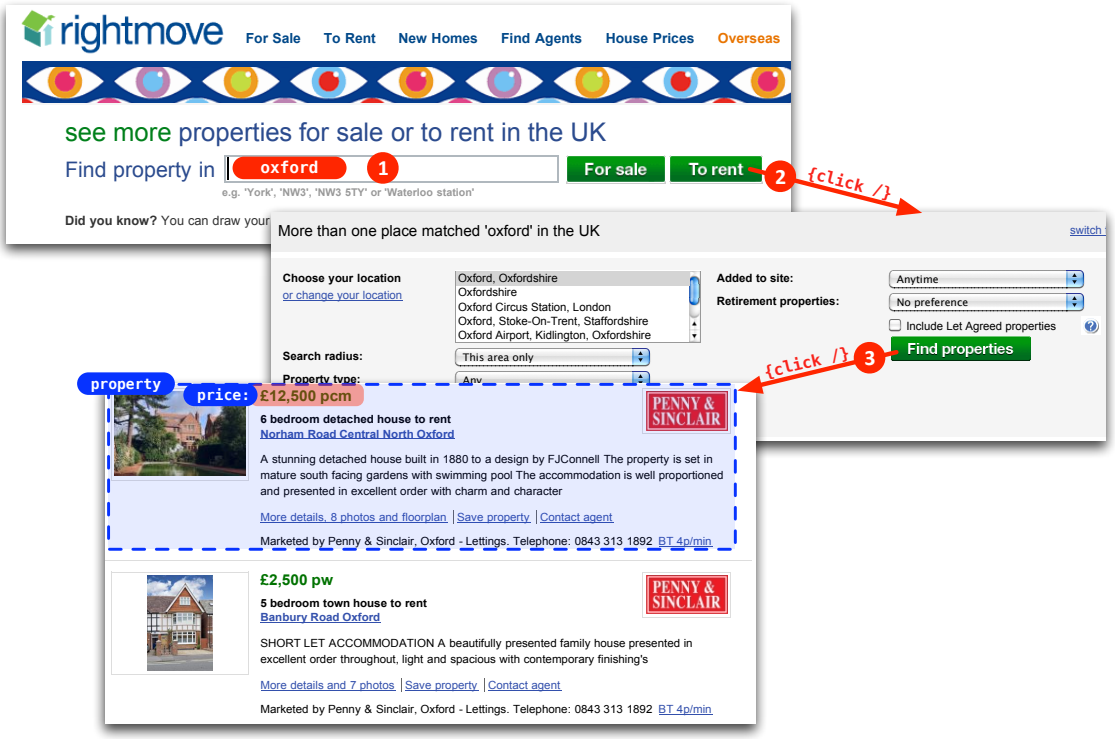
In this program, we load our expression using a Pig Latin `LOAD` function. We use this expression to instantiate an OXPATH User Defined Function (UDF) in line 1. We continue in line 2 by loading input values for the expression's variables. Finally in line 3, we execute all wrapper programs in parallel as controlled by Pig running on a Hadoop cluster. The language further supports decomposition, such as representing the OXPATH expression segment up until the form submission as a continuation encoded as an HTTP `GET` request. Further, the variable `properties` contains our structured output and could be used for further query composition or otherwise stored in a database.

1.5 Formal Aspects

In this section, we briefly discuss formal aspects of OXPATH by giving the intuition behind OXPATH's innovative PAAT algorithm which guarantees polynomial evaluation time and memory use independent of the number of visited pages. Further, we discuss the complexity of evaluating expressions of multiple OXPATH language fragments.

1.5.1 Page-at-a-Time (PAAT) Algorithm

The evaluation algorithm of OXPath is dominated by two issues: **(1)** How to minimize the number of page buffers without reloading pages, and **(2)** how to guarantee an efficient, polynomial time evaluation of individual pages. To avoid reloading pages, we need to visit each page exactly once, and hence we have to retain each page until all its descendants have been processed. To address **(1)**, our *page-at-a-time* algorithm traverses the page tree




```
doc("rightmove.co.uk")/descendant::field()[1]/{"Oxford"}
/following::input#rent/{click/}//select#minBedrooms/{"2"/}
//select#maxPrice/{"1,750 PCM"/}//input#submit/{click/}
///a[contains(.,"next")]/{click/}*
//ol#summaries/li:<property>[.//p.price:<price=string(.)>]
```

Figure 1.1: OXPATH for Rental Properties in Oxford

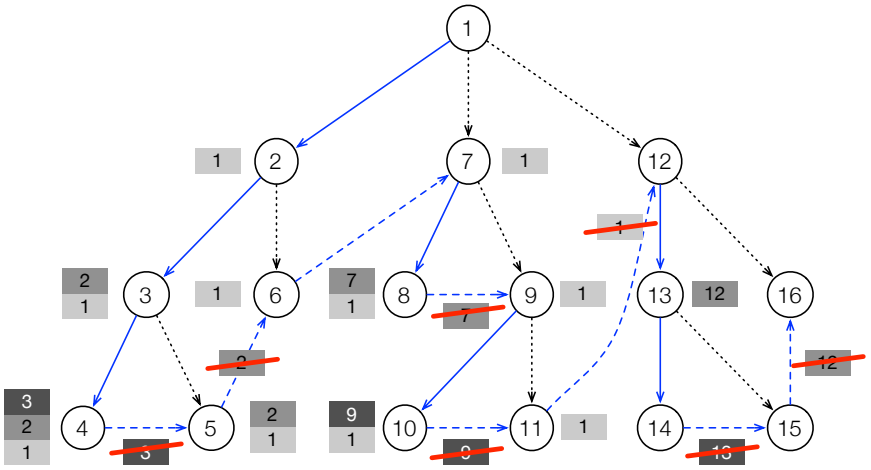


Figure 1.2: Illustration of PAAT Evaluation across Multiple Pages

in a depth-first manner without retaining information on formerly visited pages. However, a naive depth-first evaluation within individual pages would cause a worst-case exponential runtime and violate **(2)**, necessitating memoization of intermediate results. As a solution to both requirements, we employ two mutually recursive evaluation procedures that efficiently evaluate individual pages while assuring memory optimality. Individual page evaluation uses memoization to evaluate pages similar to the top-down evaluation technique introduced in [3].

An example of an abstraction of OXPath expression visiting multiple pages with PAAT is shown in Figure 1.2. In this example, the graph vertices are pages visited. Blue directed edges indicate page visits resulting from actions on page represented by the start vertex. The blue dashed edges visit the next visited pages, but with action on a previously-buffered page (black dashed edge).

Beside these vertices are stacks encoding the buffered pages. The stack frame numbers correspond to the vertex numbers. Pages are not buffered if they contain no additional context to evaluate, and buffered pages are discarded when they are no longer needed.

This algorithm is the basis for the OXPATH language complexity results described in the next section.

1.5.2 Complexity of OXPath Expression Evaluation

OXPATH has a combined complexity of PTIME-hard and data complexity of NLOGSPACE. We summarize the combined complexity of several OXPATH language fragments as well as OX LATIN in the following table:

Language	Time	Space
XPath	$O(n^4 \cdot q^2)$	$O(n^3 \cdot q^2)$
OXPATH$_{\mathsf{basic}}$	$O(n^6 \cdot q^2)$	$O(n^5 \cdot q^2)$
OXPATH$_{-\mathsf{Kleene}}$	$O((p \cdot n)^6 \cdot q^3)$	$O(n^5 \cdot q^3)$
OXPATH$_{-\mathsf{unboundedKleene}}$	$O((p \cdot n)^6 \cdot q^3)$	$O(n^5 \cdot (q+k)^3)$
OXPATH (full)	$O((p \cdot n)^6 \cdot q^3)$	$O(n^5 \cdot (q+d)^3)$
OXPATH$_{\mathsf{norm}}$	$O((p \cdot n)^5 \cdot q^3)$	$O(n^4 \cdot (q+d)^3)$

In the above table, OXPATH$_{\mathsf{basic}}$ is OXPATH without actions or Kleene stars, OXPATH$_{-\mathsf{Kleene}}$ is OXPATH without Kleene stars, OXPATH$_{-\mathsf{unboundedKleene}}$ is OXPATH without unbounded Kleene stars, and OXPATH$_{\mathsf{norm}}$ is a static query rewriting of full OXPATH we call Normalized OXPATH that results in favorable complexity bounds. In the table, n refers to document size, q is the query size, p is the number of pages visited, d is the depth of the page tree, and k is the value of d or the upper limit of the Kleene star unfoldings, whichever is greater. For languages with Kleene stars, these results hold for queries that terminate.

OXPATH generally requires n^2 more time than XPath due to extra information that must be added to the context set in order to construct the extracted output. This penalty is improved by a factor of n for OXPATH$_{\mathsf{norm}}$, a rewriting for full OXPATH we introduce that improves the complexity while (possibly) lengthening the query.

We add a factor of p with fragments with actions because we are considering multiple pages. Actions also increase complexity by a factor of q because contextual actions require computing action-free prefixes. Additionally, Kleene stars and actions increase the number of pages we must buffer: q pages must be buffered in the presence of actions and the maximum depth of the page navigation tree, d, pages may be buffered for unbounded Kleene stars.

1.6 System

OXPATH uses a three layer architecture as shown in Figure 1.3:

(1) The **Web Access Layer** enables programmatic access to a page's DOM as rendered by a modern web browser. This is required to evaluate OXPATH expressions which interact with web applications or access computed CSS style information. The web layer is implemented as a facade which promotes interchangeability of the underlying browser engine. Mozilla, Webkit and HTMLUnit are currently supported.

(2) The **Engine Layer** evaluates OXPATH expressions by PAAT, including value memoization and buffering as described in the previous section.Axis navigation is handled by the browser's XPath API.

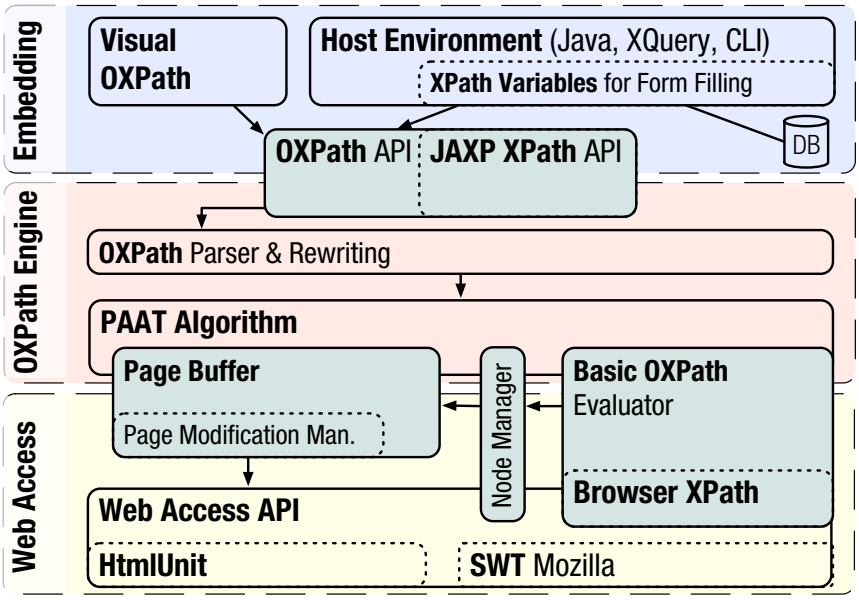


Figure 1.3: OXPATH System Architecture

(3) The **Embedding Layer** facilitates the integration of OXPATH within other systems and provides a host environment to instantiate OXPATH expressions. The host environment provides variable bindings from databases, files, or even other OXPATH expressions for use within OXPATH. To facilitate OXPATH integration, we slightly extend the JAXP API to provide an output stream for extracted data. Two such host environments were presented in Section 1.3.

OXPATH is complemented by a visual user interface, a tool that records mouse clicks, keystrokes, and other actions in sequence to construct an equivalent OXPATH expression based on the Eclipse framework (`eclipse.org`). It also allows the selection and annotation of nodes used to construct a generalized extraction expression. We are actively improving the visual interface and developing a visual debugger for OXPATH.

1.7 Summary

We have limited the scope of the discussion here to fundamental aspects of the OXPATH formalism. Further details are contained in the remainder of this report. In particular, this work introduces the PAAT (page-at-a-time) algorithm that evaluates OXPATH expressions with intelligent page caching without sacrificing the efficiency of regular XPath. In this way, PAAT guarantees polynomial evaluation time and memory use independent of the number of visited pages. Further, this work highlights experimental results of the current prototype. These experimental results validate our strong theoretical time and memory

guarantees. OXPATH performs faster than comparable systems by at least an order of magnitude in experiments where a constant memory footprint for OXPATH can be empirically observed. No observed competitor managed memory as intelligently as PAAT: either all target pages were cached (requiring linear memory w.r.t. pages visited) or a fixed number of pages were cached (requiring pages to be revisited in the general case).

Bibliography

[1] James Clark and Steve DeRose. XML Path Language (XPath) Version 1.0. Recommendation, W3C, 1999. `http://www.w3.org/TR/xpath/`, checked 2009/10/14.

[2] Tim Furche, Georg Gottlob, Giovanni Grasso, Christian Schallhart, and Andrew Sellers. Oxpath: A language for scalable, memory-efficient data extraction from web applications. In *Proceedings of Very Large Databases (VLDB) Endowment*, 2011.

[3] Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient Algorithms for Processing XPath Queries. *ACM Transactions on Database Systems*, 2005.

[4] Maarten Marx. Conditional XPath. *ACM Transactions on Database Systems*, 30(4):929–959, 2005.

[5] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.