OXPath: A Language for Scalable, Memory-efficient Data Extraction from Web Applications

Tim Furche, Georg Gottlob, Giovanni Grasso, Christian Schallhart, Andrew Sellers
Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD
firstname.lastname@comlab.ox.ac.uk

ABSTRACT

The evolution of the web has outpaced itself: The growing wealth of information and the increasing sophistication of interfaces necessitate automated processing. Web automation and extraction technologies have been overwhelmed by this very growth.

To address this trend, we identify four key requirements of web extraction: (1) Interact with sophisticated web application interfaces, (2) Precisely capture the relevant data for most web extraction tasks, (3) Scale with the number of visited pages, and (4) Readily embed into existing web technologies.

We introduce OXPath, an extension of XPath for interacting with web applications and for extracting information thus revealed. It addresses all the above requirements. OXPath's page-at-a-time evaluation guarantees memory use independent of the number of visited pages, yet remains polynomial in time. We validate experimentally the theoretical complexity and demonstrate that its evaluation is dominated by the page rendering of the underlying browser.

Our experiments show that OXPath outperforms existing commercial and academic data extraction tools by a wide margin. OXPath is available under an open source license.

Categories and Subject Descriptors

H.3.5 [**Information Storage and Retrieval**]: Online Information Services—*Web-based services*

General Terms

Languages, Algorithms

Keywords

Web extraction, web automation, XPath, AJAX

1. INTRODUCTION

The dream that the wealth of information on the web is easily accessible to everyone is at the heart of the current evolution of the web. Due to the web's rapid growth, humans can no longer find all relevant data without automation. Indeed, many invaluable web services (e.g., Amazon, Facebook, Pandora) already offer limited automation focusing on filtering or recommendation. But in many cases, we cannot expect data providers to comply with yet another interface suitable for automatic processing. Neither can we afford to wait another decade for publishers to implement these interfaces. Rather, data should be extracted from existing, human-oriented user interfaces. This lessens the burden for providers, yet allows automated processing of everything accessible to human users, not just arbitrary fragments exposed by providers. This approach complements initiatives, such as Linked Open Data, which push providers towards publishing in open, interlinked formats.

To enable automation, data accessible to humans through existing web interfaces needs to be transformed into structured data, e.g., a gray `span` with class `source` on Google News should be recognized as a news source. These observations lead us to call for a *new generation of web extraction tools*, which **(1)** can *interact* with rich interfaces of (scripted) web applications by simulating user actions, **(2)** provide extraction capabilities sufficiently *expressive* and *precise* to specify the data for extraction, **(3)** *scale* well even if the number of relevant web sites is very large, and **(4)** are *embeddable* in existing programming environments for servers and clients.

Why a new generation? Previous approaches to web extraction [11, 16] do not adequately address web page scripting. Where scripting is addressed [2, 17], the simulation of user actions is neither declarative, nor succinct, but rather relies on action scripts and standalone, heavy-weight extraction interfaces. Though Web automation tools such as [4, 8] are able to deal with scripted web applications, they are tailored to automate single action sequences and prove to be inconvenient and inefficient in large-scale data extraction tasks which require multi-way navigation (Section 5).

In this paper, we introduce **OXPath**, a careful, declarative extension of XPath for interacting with web applications to extract information revealed by such interactions. It extends XPath with a few concise extensions, yet addresses all the above desiderata:

1—Interaction. OXPath allows the *simulation of user actions* to interact with the scripted multi-page interfaces of web applications: **(I)** Actions are specified *declaratively* through action types and context elements, such as the links to click or the form field to fill. **(II)** In contrast to most previous web extraction and automation tools, actions have a *formal semantics* (Section 2.2) based on a **(III)** novel multi-page *data model for web applications* that captures both page navigation and modifications to a page (Section 2.1).

2—Expressive and precise. OXPath inherits the *precise* selection capabilities of XPath (rather than heuristics for element selection as in [4]) and extends them: **(I)** OXPath allows *selection based on visual features*. Specifically, all *CSS properties* are exposed via a new axis. **(II)** OXPath deals with navigation through page sequences, including *multi-way navigation*, e.g., following multiple links from the same page, and *unbounded navigation sequences*, e.g., following next links on a result page until there is no

Proceedings of the VLDB Endowment, Vol. 4, No. 11


```
doc("amazon.co.uk")//field()[@title='Search in']/{"Books"}❶
  /following::field()[@title='Search for']/{"Seattle" /}❷
  //field()[@alt='Go']/{click /}❸
  //a[*.refinementLink[.#='History']]/{click /}❹
  //*.result:<book>[.//a.title:<title=(.)>/{click /}❺
    //b[.#='Publisher']/following-sibling::*:<publisher=(.)>]
    [.//span.price[1]:<price=(.)>]
```

Figure 1: Finding an OXPath through Amazon.

further. **(III)** OXPath enables the identification of *data for extraction*, which can be assembled into (hierarchical) records, regardless of its original structure. **(IV)** Based on the formal semantics of OXPath (Section 2.2), we show that its extensions considerably increase the language's expressiveness (Section 2.3).

3—Scale. OXPath *scales* well both in time and memory: **(I)** We show that OXPath's *memory requirements are independent* of the number of pages visited (Section 4). To the best of our knowledge, OXPath is the first web extraction tool with such a guarantee, as confirmed by a comparison with five commercial and academic web extraction tools. **(II)** We show that the combined complexity of evaluating OXPath remains polynomial (Section 2.3) and is only slightly higher than that of XPath (Section 4). **(III)** We also show that OXPath is highly parallelizable (Section 2.3). **(IV)** We verify these theoretical results in an *extensive experimental evaluation* (Section 5), showing that OXPath outperforms existing extraction tools on large scale experiments by at least one order of magnitude.

4—Embeddable, standard API. OXPath is designed to integrate with other technologies, e.g., Java, XQuery, or Javascript. Following the spirit of XPath, we provide an API and host language to facilitate OXPath's interoperation with other systems (Section 3).

Bonus: Open Source. On `http://diadem-project.info/oxpath`, we provide under the new BSD license the OXPath prototype and API along with some illustrative examples.

1.1 Scenario: History Books on Seattle

To extract history books on Seattle currently offered on `amazon.co.uk`, a user has to perform the following sequence of actions to retrieve the page listing these books (see Figure 1): **(1)** Select "Books" from the "Search in" select box, **(2)** enter "Seattle" into the "Search for" text field, and **(3)** press the "Go" button to start the search. On the returned page, **(4)** refine the search to only "History" books. Figure 1 shows an OXPath expression that realizes this navigation in Lines 1–4 (the red numbers in Figure 1 refer to the user actions above). To select the two input fields, we use OXPath's `field()` node-test (matching only visible form elements) and each node's title attribute (`@title`). A contextual action (enclosed in `{}`) selects "Books" from the select box and continues the navigation from that field. The other actions are absolute (with an added `/` before the closing brace) where navigation continues at the root of the page retrieved by the action. To select the "History" link, we adopt the . notation from CSS for selecting elements with a class attribute `refinementLink` and use OXPath's `#=` shorthand for XPath's `contains()` function for identifying the "History" text.

Once on that page, we want to extract the book title, price, and publisher. Lines 5–7 of Figure 1 show how to achieve this extraction: We identify the record boundaries as the element with class `result` and instruct OXPath to label these records with `book` using the record *extraction marker* `:<book>`. From there, we navigate to the contained title links, extract their value as a `title` attribute, and click on the link to get to the page for the individual book, where we find and extract the publisher. Finally, we extract the price. Note that it is on the previous page, but the user does not need to care of the order pages are visited in or if they need to be buffered. OXPath buffers pages when necessary, yet guarantees that the number of buffered pages is independent of the number of visited pages.

In Appendix A and on `diadem-project.info/oxpath` we show further scenarios for using OXPath: searching for a flight on `kayak.co.uk` (Figure 9) where interaction with a visual, scripted interface is required to find non-stop flights to Seattle, finding relevant academic papers and their citations from Google Scholar (Figure 11), and extracting stock quotes from Yahoo Finance (Figure 10).

1.2 OXPath by Example

OXPath extends XPath with four concepts: Actions to navigate the interface of web applications, means for interacting with highly visual pages, extraction markers to specify data to extract, and the Kleene star for extraction from a set of pages with unknown extent.

Actions. For simulating user actions such as clicks or mouseovers, 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* (Section 2.2) 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. For lightweight visual navigation we expose the computed style of rendered HTML pages with 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 of the DOM `style` object, e.g., it is possible to select nodes by their (rendered) color or font size. To ease fields navigation, we introduce the node-test `field()`, that relies on the **`style`** axis to access the computed CSS style to exclude not visible fields, 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 and to form attributes from extracted data. For example,

```
doc("news.google.com")//div[@class~="story"]:<story>
      [.//h2:<title=string(.)>]
      [.//span[style::color="#767676"]:<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 [12]. 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 Related Work

We briefly summarise the state-of-the art in web automation and extraction and contrast it to OXPath. In particular, we consider **(1)** filling and submitting a web form, **(2)** multi-way navigation, e.g., following multiple links from the same page, **(3)** memory management for large scale extraction **(4)** and availability of an API as concrete criteria for the requirements discussed above. The relevant tools can be divided into three classes:

Full-fledged, stand-alone extraction tools (as Lixto [2]) are at least as expressive as OXPath but are outclassed by OXPath both in extraction speed and memory by a wide margin (Section 5). Lixto, Visual Web Ripper (`visualwebripper.com`), and Web Content Extractor (`newprosoft.com/web-content-extractor.htm`) are moving towards interactive wrapper generator frameworks, recording user actions in a browser and replaying these actions for extracting data. None of these systems addresses memory management and our experimental evaluation (Section 5) demonstrates that such systems indeed require memory linear in the number of accessed pages—in contrast to OXPath.

Extraction languages [13, 1, 14, 11, 16, 15] use a declarative approach, much like OXPath; however, they often do not adequately facilitate deep web interaction such as form submissions mainly due to their age. Also, they do not provide native constructs for page navigation, apart from retrieving a page given a URI.

The BODED extraction language (in BODE [17] system) is able to deal with modern web applications. Similar to OXPath, it is browser-based but its scripts are imperative and XML based; memory management is not considered and it employs browser replication for multi-way navigation that imposes a performance penalty, making BODE unsuitable for large-scale data extraction.

In the evaluation, we compare also with Web Harvest (`web-harvest.sourceforge.net`), a recent, open source example of an extraction language. Extraction tasks are specified as imperative scripts (in XML files). It does not deal with web applications (form filling) and does not give access to the rendered page, but rather to a cleaned XML view of HTML documents.

Web automation tools are mainly focused on single navigation sequences through web applications, but do not consider large-scale web extraction and mostly do not support multi-way navigation. Coscripter [8] and iMacros (`iopus.com/iMacros`) are examples of such tools: They do not allow multi-way navigation due to limited support for iterative and conditional programming. Vegemite [9] is a CoScripter extension that facilitates some extraction capability such as population of tables. However, as its authors note, this interactivity comes at a price to performance as the same page may be reloaded many times. Also, page state is not preserved and thus some web applications may not behave as expected.

The same applies to Chickenfoot [4], a language for web automation that allows users to program scripts that run in Firefox. It enables interaction with forms as well as loading and navigating pages. Multi-way navigation is feasible, but only by explicitly using "back" instructions which command the browser to return to the previous page. Thus, page buffering is unnecessary, but page state is not preserved and pages need to be rendered again.

In contrast to supervised tools, unsupervised web extraction tools (see survey in [5]) require little or no input from the user. They focus on automated analysis rather than extraction and are not directly comparable to OXPath. Furthermore, publicly available systems (such as [6]) do not deal with scripted web applications.

⟨*expr*⟩ ::= ⟨*funct*⟩ '(' (⟨*expr*⟩ (',' ⟨*expr*⟩)*)? ')'
| '(' ⟨*expr*⟩ ')' | ⟨*xpath-value*⟩ | ⟨*expr*⟩ ⟨*binop*⟩ ⟨*expr*⟩
| ⟨*xpath-unop*⟩ ⟨*expr*⟩ | ⟨*path*⟩ ('|' ⟨*path*⟩)*
⟨*path*⟩ ::= ⟨*estep*⟩ ('/' ⟨*path*⟩)?
⟨*estep*⟩ ::= ⟨*step*⟩ | ⟨*action*⟩ | ⟨*kleene*⟩
⟨*kleene*⟩ ::= '(' ⟨*path*⟩ ')*' ('{' ⟨*number*⟩ ',' ⟨*number*⟩ '}')?
⟨*step*⟩ ::= ⟨*axis*⟩ '::' ⟨*node-test*⟩ | ⟨*step*⟩ ⟨*qualifier*⟩ | ⟨*step*⟩ ⟨*marker*⟩
⟨*funct*⟩ ::= ⟨*xpath-funct*⟩ | '*doc*'
⟨*axis*⟩ ::= ⟨*xpath-axis*⟩ | '*style*'
⟨*node-test*⟩ ::= (⟨*xpath-nt*⟩ | '*field()*') (('.' | '#') ⟨*ident*⟩)?
⟨*qualifier*⟩ ::= '[' ⟨*expr*⟩ ']' | '[?' ⟨*expr*⟩ ']'
⟨*marker*⟩ ::= ':<' ⟨*name*⟩ ('=' ⟨*expr*⟩)? '>'
⟨*action*⟩ ::= '{' (⟨*ident*⟩ | ⟨*xpath-value*⟩) '/' ? '}'
⟨*binop*⟩ ::= ⟨*xpath-binop*⟩ | '~=' | '#='

Figure 2: OXPath Syntax.

2. LANGUAGE

The syntax of OXPath is defined in Figure 2 (XPath literals as in [7, 3]). We impose additional restrictions on OXPath expressions to avoid interactions of functions and sorting operators with OXPath's actions and extraction expressions (see Appendix B).

We choose XPath rather than XQuery as underlying language, since XPath provides sufficient expressiveness for most tasks if extended with data extraction features not found in either XPath or XQuery, yet allows strong guarantees on time and memory bounds.

2.1 Data Model

OXPath's data model extends the data model of XPath to multiple pages (i.e., HTML documents), interconnected by actions. OXPath expressions are evaluated on relational structures, called *page trees*, with schema $\left((R_v)_{v\in\mathsf{NodeSets}}, R_{\mathsf{child}}, R_{\mathsf{next\text{-}sibl}}, (R_\alpha)_{\alpha\in\mathsf{Acts}}\right)$ where $(R_v)_{v\in\mathsf{NodeSets}}$ is the set of (unary) node-set relations (type and label relations), R_{child} the parent-child and $R_{\mathsf{next\text{-}sibl}}$ the direct sibling relation. The remaining XPath axes (such as descendant) are derived from the basic relations as usual [3]. We add a set of binary relations $(R_\alpha)_{\alpha\in\mathsf{Acts}}$, one for each OXPath action. A tuple $(x,y)\in R_\alpha$ indicates that the action α triggered on node x yields the page rooted at y. $(R_\alpha)_{\alpha\in\mathsf{Acts}}$ together with R_{child} form a tree: Each page has a unique parent node connected by a single action edge. For instance, if two links point to the same URI this yields two different pages in the page tree.

An OXPath expression E returns an unordered XPath tree, i.e., a set of tuples over the schema $\left((R_v)_{v\in\mathsf{NodeSets}}, R_{\mathsf{child}}\right)$. This allows the extraction of (possibly nested) records with multiple values. We refer to nodes in this tree (in the page tree) as output (input) nodes.

Figure 3 illustrates both the data model and the result of an OXPath expression. The *page tree* consists of the nodes of the considered pages, connected by child, next-sibl, and (labeled) action edges (in the figure we use only `{click/}`). Each distinct path of actions and nodes leads to a distinct page. OXPath traverses action edges only in the direction of the edge (no reverse navigation), and only directly (no `descendant` over action edges). The part of the page

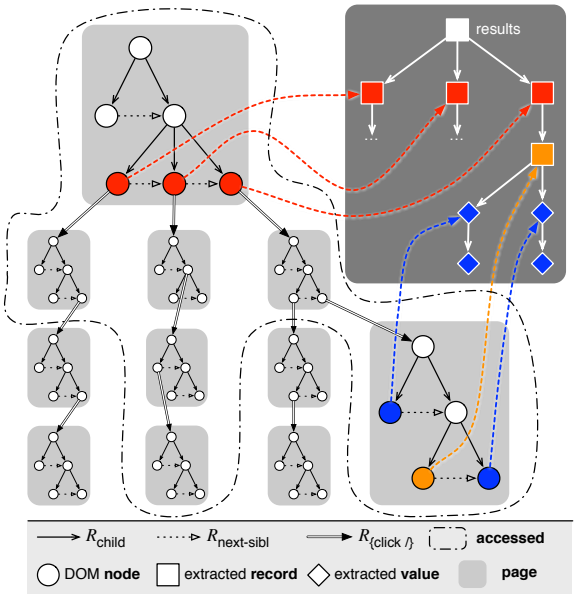


Figure 3: OXPath page tree.

tree (the pages and nodes) accessed by a given OXPath expression is called the *accessed page tree* (*accessed pages and nodes*).

Figure 3 also shows the result of an OXPath expression: For a node matching a record extraction markers (such as `:<R>`), a new node in the output tree is created and labeled `R`. For nodes matching value extraction markers (such as `:<R=string(.)>`), two new nodes in the output tree are created, one is labeled `R`, the other is a text node containing the extracted value and is created as a child of the first node. The structure of the output tree reflects the structure of the extraction markers in the OXPath expression (but not that of the input tree): For instance `//a:<R>[parent::b:<S>]` returns a tree where the output nodes for `S` are children of the output nodes for `R`.

2.2 Semantics

The semantics of OXPath is defined in terms of its extraction semantics ($[\![expr]\!]_{\mathsf{E}}(c)$). The extraction semantics specify the construction of the result tree and uses OXPath's value semantics ($[\![expr]\!]_{\mathsf{V}}(c)$) for matching nodes from the page tree. The value semantics loosely follows the XPath semantics in [3]: Parametrized by an OXPath expression *expr*, it takes a node n, supplemented into a context tuple c, and computes the value reached via *expr*. This value is either a set of context tuples, a Boolean, integer, or string value—as in XPath.

Each *context tuple*—and here the OXPath semantics differs from XPath—$c = \langle n, p, l\rangle$ consists of an input node n, the parent output node p, and the last sibling output node l, also accessed as $c.n$, $c.p$, $c.l$. We maintain both the parent and sibling match to organize the extracted nodes hierarchically: Subsequent extraction matches outside predicates yield nodes that are descendants of p and thus siblings of l. On entering a predicate, l replaces p as parent output node such that extraction matches yield nodes that are descendants of l (nested records) rather than further siblings.

We *start the evaluation* of an OXPath expression with the initial context node $c = \langle \bot, \langle \top, \texttt{results}\rangle, \langle \top, \texttt{results}\rangle\rangle$, where $\bot$ is an arbitrary context node and $\langle \top, \texttt{results}\rangle$ is the root of all subsequent extraction matches. The context node is arbitrary as an expression always starts with `doc(`*uri*`)` which returns the root of the page with the given *uri* for any context node.

The complete semantics is given in Appendix C. The extraction semantics is fairly straightforward. Thus, we focus here on the

V1	$[\![path]\!]_{\mathsf{V}}(c)$	$= [\![path]\!]_{\mathsf{N}}(c)$
N1	$[\![estep/path]\!]_{\mathsf{N}}(c)$	$= \{c'' \mid c' \in [\![estep]\!]_{\mathsf{N}}(c) \wedge c'' \in [\![path]\!]_{\mathsf{N}}(c')\}$
N2	$[\![axis::nodes]\!]_{\mathsf{N}}(c)$	$= \{\langle n', c.p, c.l\rangle \mid R_{axis}(c.n, n') \wedge n' \in R_{nodes}\}$
N3	$[\![step[q]]\!]_{\mathsf{N}}(c)$	$= \{c' \in [\![step]\!]_{\mathsf{N}}(c) \mid [\![q]\!]_{\mathsf{B}}(\langle c'.n, c'.l, c'.l\rangle)\}$
N4	$[\![step_{\pm}[qp]]\!]_{\mathsf{N}}(c)$	$= \{c' \in [\![step_{\pm}]\!]_{\mathsf{N}}(c) \mid C = [\![step_{\pm}]\!]_{\mathsf{N}}(c) \wedge$ $[\![\text{REWRITE}_{\pm}(qp, C, c')]\!]_{\mathsf{B}}(\langle c'.n, c'.l, c'.l\rangle)\}$
N5	$[\![\{action\ /\}]\!]_{\mathsf{N}}(c)$	$= \{\langle n', c.p, c.l\rangle\}$ with n' such that $R_{action}(c.n, n')$
N6	$[\![\{action\}]\!]_{\mathsf{N}}(c)$	$= [\![\text{AFP}(action, c.n)]\!]_{\mathsf{N}}([\![\{action\ /\}]\!]_{\mathsf{N}}(c))$
N7	$[\![step : \langle M[= v]\rangle]\!]_{\mathsf{N}}(c)$	$= \{\langle c'.n, c'.p, \text{OUT}(c'.n, M)\rangle \mid \exists c' \in [\![step]\!]_{\mathsf{N}}(c)\}$
N8	$[\![(path)*]\!]_{\mathsf{N}}(c)$	$= \{c_r \mid \exists r \geq 0\ \forall 0 \leq s \leq r :$ $c_{s+1} \in [\![path]\!]_{\mathsf{N}}(c_s) \wedge c_0 = c\}$
N9	$[\![(path)*\{v, w\}]\!]_{\mathsf{N}}(c)$	$= \{c_r \mid \exists v \leq r \leq w\ \forall 0 \leq s \leq r :$ $c_{s+1} \in [\![path]\!]_{\mathsf{N}}(c_s) \wedge c_0 = c\}$

Table 1: Value Semantics of OXPath.

value semantics. For an OXPath expression that is also an XPath expression, it computes the exact same result as standard XPath. In Table 1 we highlight only the major differences to the XPath semantics necessitated by OXPath extensions.

$[\![path]\!]_{\mathsf{V}}$ delegates the evaluation of a *path* to $[\![path]\!]_{\mathsf{N}}$ (V1), which handles expressions computing a node sets. The rules for other values are omitted here (see Appendix C). Paths are evaluated using $[\![path]\!]_{\mathsf{N}}$. A path is composed into OXPath steps (N1) and each type of step is treated in N2–N9:

N2–N4: *Axes, node-tests, and predicates.* OXPath axes, node-tests, and predicates are treated as in standard XPath. When entering predicates, the last sibling output node is copied to the parent output node. Expressions in predicates are cast to Booleans by means of $[\![expr]\!]_{\mathsf{B}}$ as in XPath (see [3] and Appendix C). For positional predicates (N4) we replace, using $\text{REWRITE}_{\pm}$, in *qp* each non-nested occurrence of `last()` with $|C|$ and of `position()` with the position of c' within C. The order depends on the axis in *step*.

N5–N6: *Actions.* Actions map the context node $c.n$ to a node in a different page. *Absolute actions* (N5) map $c.n$ to the root n' of some other page with $(R_\alpha)_{\alpha \in \mathsf{Acts}}$ (by the data model there can be only a single such node). For *contextual actions* in rule N6, we first evaluate the absolute action using N5, but then attempt to move to a node in the new page that corresponds closely to $c.n$ in that it is selected by the same OXPath expression used to select $c.n$ in the old page. The AFP of *action* and $c.n$ in an OXPath expression E returns that OXPath expression:[1] Let *base* be the subexpression of E between *action* and its last preceding absolute action, stripped of all contextual actions and extraction markers. Then $\text{AFP}(action, c.n) = (base)[i]$ where i is the positon of c in document order among all nodes in the current page matching *base*. In the current page, this expression uniquely identifies $c.n$. When evaluated from the root returned by the absolute action on $c.n$, it selects a node reached by the same path and in the same relative position.

N7: *Extraction markers.* For extraction markers, the context set of the corresponding step is modified by replacing the last sibling marker with $\text{OUT}(c'.n, M)$ where OUT is an injective function that maps an input node and extraction marker to an output node.

N8–N9: *Kleene star.* For unbounded and bounded Kleene stars, the Kleene-repeated path is matched multiple times; if necessary, bounds on the number of repetitions are enforced.

2.3 Complexity

Considering the complexity of OXPath, we note that expressions containing Kleene star repeated actions may require access to an unbounded number of pages. In particular, when we evaluate such

[1] For sake of brevity, we omit E where clear from context.

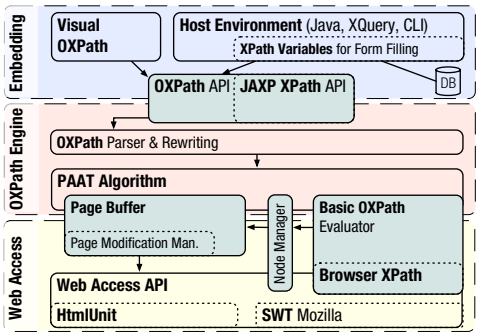


Figure 4: OXPath System Architecture.

an expression, we do not know whether the evaluation terminates and how many pages are accessed during evaluation. Thus, when we discuss the complexity of evaluating OXPath, we only consider expressions whose evaluations *terminate* and consider *all accessed pages* as input. Furthermore, we assume that traversing an action takes constant time. There are few web pages where actions (triggering a Javascript execution) are not executed quickly.

THEOREM 1 (COMPLEXITY). *OXPath query evaluation without string concatenation and multiplication is in* NLOGSPACE *for* data complexity. *OXPath query evaluation is* PTIME*-complete for* combined complexity.

We give the proof in Appendix D along with a discussion of other language properties of OXPath, viz. that no two nodes from different pages need to be sorted and that no output buffer is required.

3. SYSTEM ARCHITECTURE

OXPath uses a three layer architecture as shown in Figure 4, promoting modularity and allowing substitution of web browsers and host environments:

(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 (Firefox and HTMLUnit).

(2) The **Engine Layer** consists of two main components: the OXPath parser/rewriter and the PAAT algorithm (Section 4). Basic OXPath steps, i.e., subexpressions without actions, extraction markers and Kleene stars, are directly handled by the browser's XPath engine. The PAAT algorithm buffers pages as well as detects and manages page modifications. When a page is to be removed from the buffer, the corresponding nodes are also dropped by the node manager. The buffer manager retrieves new pages and, when necessary, makes page copies before modification by actions.

(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.

Though OXPath can be used in any number of host languages such as Java, XSLT, or Ruby, we designed a lightweight host language for large-scale data extraction. It is a SQL-like relational query language with OXPath subqueries, grouping, and aggregation. We separate these tasks, as well as the provision of variable bindings, from the core language to preserve the declarative nature of OXPath and to guarantee efficient evaluation of the core features.

OXPath is complemented by a visual user interface, a Firefox add-on that records mouse clicks, keystrokes, and other actions in sequence to construct an equivalent OXPath expression. It also allows the selection and annotation of nodes used to construct a generalised extraction expression. We are actively improving the visual interface and developing a visual debugger for OXPath.

```
1  Function eval_(Expr, c, prot):
2  if Expr ≡ ε then return {c};
3  if Lookup[Expr, c] ≠ null then return Lookup[Expr, c];
4  Expr ≡ et where e matches one of the following cases;
5  Result ← ∅;
6  if e ≡ axis :: nodes then
7      Ctx ← {⟨n′, c.p, c.l⟩ | R_axis(c.n, n′) and n′ ∈ R_nodes};
8      for c′ ∈ Ctx do
9          prot′ ← isLastIn(c′, Ctx) ? prot : true;
10         Result ← Result ∪ eval_(t, c′, prot′);
11 else if e ≡ [q] then
12     if eval(q, ⟨c.n, c.l, c.l⟩, (t ≡ ε) ? prot : true) ≠ ∅ then
13         Result ← eval_(t, c, prot)
14 else if e ≡ :⟨M⟩ or e ≡ :⟨M = v⟩ then
15     output R_M(OUT(c.n, M)), R_child(OUT(c.n, M), c.p);
16     if e ≡ :⟨M = v⟩ then
17         val ← eval(v, c, (t ≡ ε) ? prot : true);
18         output R_val(c′), R_child(c′, OUT(c.n, M)) | c′ new output node;
19     Result ← eval_(t, ⟨c.n, c.p, OUT(c.n, M)⟩, prot);
20 Lookup[Expr, c] ← Result;
21 return Result;
```

Algorithm 1: PAAT Simple Evaluation.

4. PAGE-AT-A-TIME EVALUATION

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 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 eval_(Expr, c, prot) and eval(Expr, Ctx, prot), shown in Algorithms 1 and 2. Both take an expression Expr, either a single context tuple c or a set Ctx of such tuples referring to the same page, and a Boolean flag prot, indicating whether the page referred to by the context is needed after the current invocation—and is therefore protected. We evaluate a general OXPath expression Expr with eval(Expr, $\{\langle\bot, \langle\top, \texttt{results}\rangle, \langle\top, \texttt{results}\rangle\rangle\}$, **false**).

4.1 PAAT Simple Evaluation

The first procedure, eval_ (Algorithm 1), applies memoization to evaluate *simple expressions* which may contain actions or Kleene stars only in predicates but not in the main path. For brevity, we only show the evaluation of the most important expressions, i.e., axis navigation with node-tests, predicates, and extraction markers.

In line 2 we check for empty expressions as *base case* and return the input context tuple c as result $\{c\}$ of the evaluation. Next, in line 3, we check whether the result of applying Expr to c has been memoized, and if so, return this result. Otherwise, we evaluate Expr and store the result at the end of the algorithm in line 20.

The main part of the algorithm is organized in a *depth-first manner:* The algorithm splits the input expression Expr into prefix e and remainder t (line 4) to evaluate e directly and t recursively.

```
1  Function eval(Expr, Ctx, prot):
2  if Expr is simple then
3      return ⋃_{c∈Ctx} eval_(Expr, c, isLastIn(c, Ctx)?prot : true);
4  Expr ≡ het where h is simple, e matches one of the following;
5  Ctx ← ⋃_{c∈Ctx} eval_(h, c, true);
6  if Ctx = ∅ then return ∅;
7  Result ← ∅;
8  if e ≡ {action/} ∨ {action} then
9      for c ∈ Ctx do
10         prot′ ← isLastIn(c, Ctx) ? prot : true;
11         c′ ← getPage(c, action, prot′);
12         if e ≡ {action} then
13             if isPageUnmodified() then c′ ← c;
14             else c′ ← eval(AFP(action, c.n), c′, true);
15         Result ← Result ∪ eval(t, {c′}, false);
16         FreeMem(pageof(c′));
17 else if e ≡ (path) ∗ {v, w} then
18     if containsAction(path) then
19         if w = 0 then Result ← Result ∪ eval(t, Ctx, prot);
20         else
21             if v ≤ 0 then Result ← Result ∪ eval(t, Ctx, true);
22             if w > 1 then Expr′ ← path path ∗ {v − 1, w − 1} t;
23             else Expr′ ← path t;
24             Result ← Result ∪ eval(Expr′, Ctx, prot);
25     else
26         Ctx′ ← Ctx;
27         for i ← 0 to w − 1 do
28             Ctx ← eval(path, Ctx, (t ≡ ε ∧ i = w − 1) ? prot : true);
29             if i ≥ v then Ctx ← Ctx \ Ctx′; Ctx′ ← Ctx′ ∪ Ctx;
30             if Ctx = ∅ then break;
31         Result ← eval(t, Ctx′, prot);
32 return Result;
```

Algorithm 2: PAAT Full Evaluation.

Thereby, e is either an axis with node test, a predicate, or an extraction marker, leading to the following case distinction: **(1)** For *axis navigation,* starting at line 6, we obtain a new context set Ctx with R_{axis} and R_{nodes} and evaluate t recursively on each $c' \in$ Ctx. If prot is set or c' is not the last tuple in the iteration, the current page must be protected since it is needed subsequently (line 9). **(2)** In line 11, we deal with *predicate expressions* $[q]$. We evaluate q with `eval`, since q may contain actions or Kleene stars. Since extracted records are structured according to the nesting of extraction markers in predicates, the last sibling match $c.l$ becomes the new parent match. **(3)** In line 14, for *extraction markers*, we output the marked node, and—if the marker computes a value—output the evaluation of v of the marked node. After outputing the match $\text{OUT}(c.n, M)$, we set the last sibling in the context accordingly.

We denote the sub-language of OXPath which contains no actions or Kleene stars at all as *OXPath*$_{\text{basic}}$. For OXPath$_{\text{basic}}$ expressions, the only recursive call to `eval`, can be replaced with calls to `eval_` to obtain a self-contained algorithm for OXPath$_{\text{basic}}$.

4.2 PAAT Full Evaluation

In Algorithm 2, we show the procedure `eval` for full OXPath. Essentially, `eval` deals with three cases: Actions, Kleene stars containing actions in the main path, and other Kleene expressions. In the first two cases, `eval` performs a depth-first traversal in the page tree, the latter is solved with a set-based approach, as all resulting nodes are on the same page. As invariance throughout the recursion, the input set Ctx always contains nodes from a single page.

In line 2, we delegate simple Expr to `eval_`. Next, we split Expr into three parts: h is the maximum prefix which is simple, e is either an action or a Kleene star, and t is the remaining expression. The prefix h is evaluated with `eval_`, and the result is returned if empty.

(1) The first main case deals with *actions,* starting at line 8. Roughly, we iterate over all c in Ctx, obtain c' via the action, and evaluate t on c'. This is not exponential, since actions cannot be traversed twice from the same node (no reverse traversal). More specifically, we protect the page to perform the action upon, either if the input flag prot is set, or if c is not last in the iteration (line 10). If the page is protected, `getPage` opens a new buffer for the page accessed through the action and returns the tuple c' referring to the root of the new page. If not, the page is replaced and all memoization information of `eval_` for the old page is freed. If the action is contextual and did not change the page, we stay at c and avoid evaluating the action free prefix $\text{AFP}(action, c.n)$. Otherwise, if the page has been modified, we apply $\text{AFP}(action, c.n)$ to obtain c'. Either way, we evaluate t recursively on c', descending one step further in the depth-first traversal of the page tree (line 15). We set the protection to **false**, since we free the page and all memoized information on this page after the invocation in any case (line 16). **(2)** Below line 18, we handle *Kleene star expressions with actions on the main path*. If the upper iteration bound w has reached 0, we evaluate t and exit (line 19). Otherwise, we also evaluate t, but only if the lower bound v has reached 0 (line 21), then instantiate one copy of the Kleene repeated *path*, and decrement the bounds (line 22) to evaluate the resulting expression Expr′ recursively. **(3)** Finally we deal with the *remaining Kleene star expressions* (line 25). They do not leave the current page; thus, exponentially many paths may reach the same node. We apply *path* repeatedly until Ctx becomes empty or the upper bound w is reached. To visit tuples only once, we store in Ctx′ all visited tuples and keep in Ctx only the new ones (line 29). Finally we apply t to Ctx′ (line 31).

4.3 Analysis of PAAT

The proofs of the following results are given in Appendix E.

PROPOSITION 2. *Evaluating an OXPath*$_{basic}$ *expression on a context tuple c using* `eval_` *takes $O(n^6 \cdot q^2)$ time and $O(n^5 \cdot q^2)$ space where q is the size of the expression and n the number of nodes in the page of c.*

Evaluating OXPath expressions containing unbounded Kleene stars and actions may cause non-termination, if there are infinitely many paths in the page tree matched by the Kleene star expression. Such cases occur **(1)** if there are cycles in the web graph matched by the expression; **(2)** if web sites serve (different) answers for an infinite number of URIs, e.g., by including its page impressions.

We investigate the complexity of OXPath$_{\text{−Kleene}}$ which contains no Kleene stars, OXPath$_{\text{bounded}}$ with only bounded Kleene stars, and full OXPath over page trees of bounded depth.

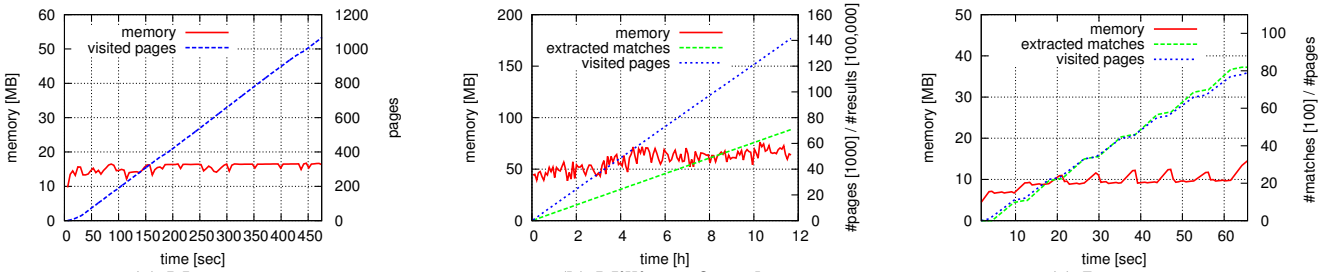
PROPOSITION 3. *Evaluating an OXPath*$_{-Kleene}$ *expression using* `eval` *takes $O((p \cdot n)^6 \cdot q^3)$ time and $O(n^5 \cdot q^3)$ space with q as above, n the maximum number of nodes in an accessed page, and p the number of such pages.*

THEOREM 4 (FULL OXPATH). *Evaluating a full OXPath expression using* `eval` *takes $O((p \cdot n)^6 \cdot q^3)$ time and $O(n^5 \cdot (q+d)^3)$ space where q, p, and n are as above, and d the depth of the accessed page tree.*

For OXPath$_{\text{bounded}}$, we note that d is bounded by the product of all Kleene star upper bounds and obtain the following result:

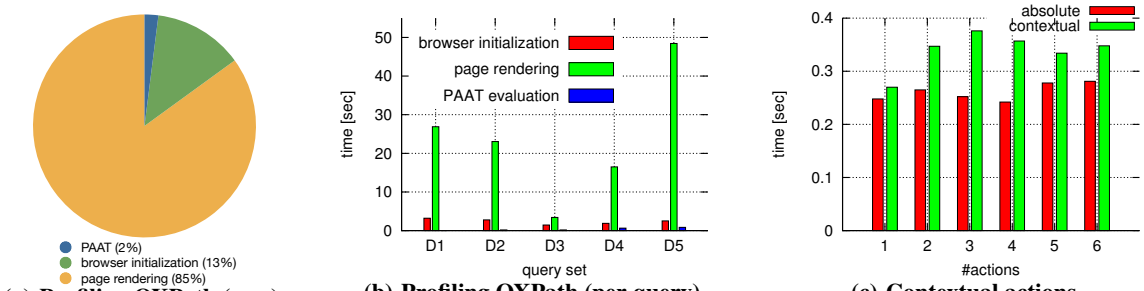
COROLLARY 5. *Evaluating a fixed OXPath expression without unbounded Kleene stars takes only* constantly many page buffers.

THEOREM 6 (MEMORY OPTIMALITY). *There exists an OXPath expression and a page tree for which every algorithm that computes $[\![\cdot]\!]_V$ without prior knowledge of the page tree requires at least as many page buffers as PAAT—if no page is loaded twice.*



(a) Many pages. **(b) Millions of results.** **(c) Large pages.**

Figure 5: Scaling OXPath: Memory, visited pages, and output size vs. time.



(a) Profiling OXPath (avg.). **(b) Profiling OXPath (per query).** **(c) Contextual actions.**

Figure 6: Profiling OXPath's components.

5. EVALUATION

In this section, we confirm the scaling behaviour of OXPath:

(1) The theoretical complexity bounds from Section 4.3 are confirmed in several large scale extraction experiments in diverse settings, in particular the constant memory use even if extracting millions of records from hundreds of thousands of web pages.

(2) We illustrate that OXPath's evaluation is dominated by the browser rendering time, even for complex queries on small web pages. None of the extensions of OXPath (Section 1.2) significantly affects the scaling behaviour or the dominance of page rendering.

(3) In an extensive comparison with commercial or academic web extraction tools, we show that OXPath outperforms previous approaches by at least an order of magnitude, although they are more limited. Where OXPath requires memory independent of the number of accessed pages, most others use linear memory.

Scaling: Millions of Results at Constant Memory. We validate the complexity bounds for OXPath's PAAT algorithm and illustrate its scaling behaviour by evaluating three classes of queries that require complex page buffering. Figure 5(a) shows the results of the first class, which searches for papers on "Seattle" in Google Scholar and repeatedly clicks on the "Cited by" links of all results using the Kleene star operator. The query descends to a depth of 3 into the citation graph and is evaluated in under 9 minutes (93 pages/min). Pages retrieved are linear w.r.t. time, but memory size remains constant even as the number of pages accessed increase. The jitter in memory use is due to the repeated ascends and descends of the citation hierarchy. Figure 5(b) shows the same test simulating Google Scholar pages on our web server (to avoid overly taxing Google's servers). The number in brackets indicate how we scale the axes. OXPath extracts over 7 million pieces of data from 140,000 pages in 12 hours (194 pages/min) with constant memory.

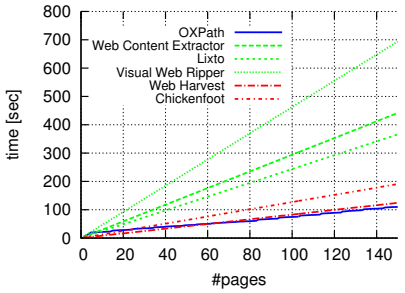
We conduct similar tests (repeatedly clicking on all links) for tasks with different characteristics: one on very large pages (Wikipedia) and one on pages with many results (product listings on Google) from different web shops. Figures 5(c) and 12 (in Appendix E) again show that memory is constant w.r.t. to pages visited even for the much larger pages on Wikipedia and the often quite visually-involved pages reached by the Google product search.

Profiling: Page Rendering is Dominant. We profile each stage of OXPath's evaluation performing five sets of queries on the following web sites: `apple.com` (D1), `diadem-project.info` (D2), `bing.com` (D3), `www.vldb.org/2011/` (D4), and the Seattle page on Wikipedia (D5). On each, we click on all links and extract the `html` tag of each result page. Figures 6(a) and 6(b) show the total average and the individual averages for each site, respectively. For `bing.com`, the page rendering time and the number of links is very low, and thus also the overall evaluation time. Wikipedia pages, on the other hand, are comparatively large and contain many links, thus the overall evaluation time is high.

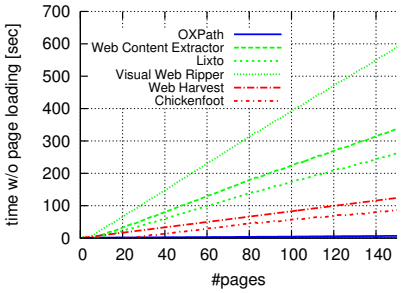
The second experiment analyses the effect of OXPath's actions on query evaluation, comparing time performance of contextual and absolute actions. Our test performs actions on pages that do not result in new page retrievals. Figure 6(c) shows the results with queries containing one to six actions on Google's advanced product search form. Contextual actions suffer a small but insignificant penalty to evaluation time compared to their absolute equivalents.

Comparison: Order-of-Magnitude Improvement. OXPath is compared to four commercial web extraction tools, Web Content Extractor (WCE), Lixto, Visual Web Ripper (VWR), the academic web automation and extraction system Chickenfoot and the open source extraction toolkit Web Harvest. Where the first three can express at least the same extraction tasks as OXPath (and, e.g., Lixto goes considerably beyond), Chickenfoot and Web Harvest require scripted iteration and manual memory management for many extraction tasks, in particular where multi-way navigation is needed. We do not consider tools such as CoScripter and iMacros as they focus on automation only and offer no iterative constructs as required for extraction tasks. We also do not consider tools such as RoadRunner [6] or XWRAP [10] as they work on single pages and lack the ability to traverse to new web pages.

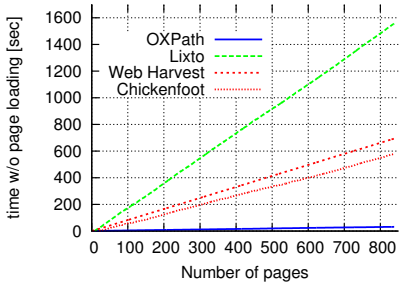
In contrast to OXPath, many of these tools cannot process scripted websites easily. Thus, we compare using an extraction task on Google Scholar, which does not require scripted actions. On heavily scripted pages, the performance advantage of OXPath is even more pronounced. With each system, we navigate the cita-



(a) Evaluation time, <150 p.



(b) Norm. evaluation time, <150 p.



(c) Norm. evaluation time, <850 p.

Figure 7: Comparison.

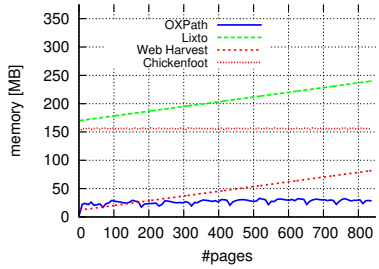


Figure 8: Comparison: Memory (<850 p.).

tion graph to a depth of 3 for papers on "Seattle" (see Appendix F).

We record evaluation time and memory for each system. In Figure 7(a), we show the (averaged) evaluation time for each system up to 150 pages. Though Chickenfoot and Web Harvest do not manage page and browser state or do not render pages at all, OXPath still outperforms them. Systems that manage state similar to OXPath are between two and four times slower than OXPath even on this small number of pages. Figure 7(b) show the evaluation time discounting the page loading, cleaning, and rendering time. This allows for a more balanced comparison as the different browser engines or web cleaning approaches used in the systems affect the overall runtime considerably. Figure 7(c) shows the same evaluation time up to 850 pages, but omits WCE and VWR as they were not able to run these tests. Again both figures show a considerable advantage for OXPath (at least one order of magnitude). Finally, Figure 8 illustrates the memory use of these systems. Again WCE and VWR are excluded, but they show a clear linear trend in memory usage in the tests we were able to run. Among the systems in Figure 8, only Web Harvest comes close to the memory usage of OXPath, which is not surprising as it does not render pages. Yet, even Web Harvest shows a clear linear trend. Chickenfoot exhibits a constant memory use just as OXPath, though it uses about ten times more memory in absolute terms. The constant memory is due to Chickenfoot's lack of support for multi-way navigation that we compensate by using the browser's history. This forces reloading when a page is no longer cached and does not preserve page state, but requires only a single active DOM instance at any time. We also tried to simulate multi-way navigation in Chickenfoot, but the resulting program was too slow for the tests shown here.

6. CONCLUSION AND FUTURE WORK

To the best of our knowledge, OXPath is the first web extraction system with strict memory guarantees, which reflect strongly in our experimental evaluation. We believe that it can become an important part of the toolset of developers interacting with the web.

We are committed to building a strong set of tools around OXPath. We provide a visual generator for OXPath expressions and a Java API based on JAXP. Some of the issues raised by OXPath that we plan to address in future work are: **(1)** OXPath is amenable to significant optimization and a good target for automated generation of web extraction programs. **(2)** Further, OXPath is perfectly suited for highly parallel execution: Different bindings for the same variable can be filled into forms in parallel. The effective parallel execution of actions on context sets with many nodes is an open issue. **(3)** We plan to further investigate language features, such as more expressive visual features and multi-property axes.

7. REFERENCES

[1] G. O. Arocena and A. O. Mendelzon. WebOQL: Restructuring documents, databases, and webs. In *ICDE*, 24–33, 1998.
[2] R. Baumgartner, S. Flesca, and G. Gottlob. Visual web information extraction with Lixto. In *VLDB*, 119–128, 2001.
[3] M. Benedikt and C. Koch. XPath Leashed. *CSUR*, 41(1):3:1–3:54, 2007.
[4] M. Bolin, M. Webber, P. Rha, T. Wilson, and R. C. Miller. Automation and customization of rendered web pages. In *UIST*, 163–172, 2005.
[5] C.-H. Chang, M. Kayed, M. R. Girgis, and K. F. Shaalan. A survey of web information extraction systems. *TKDE*, 1411–1428, 2006.
[6] V. Crescenzi and G. Mecca. Automatic information extraction from large websites. *JACM*, 51(5):731âĂŞ779, 2004.
[7] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. *TODS*, 30(2):444âĂŞ491, 2005.
[8] G. Leshed, E. M. Haber, T. Matthews, and T. Lau. CoScripter: automating & sharing how-to knowledge in the enterprise. In *CHI*, 1719–1728, 2008.
[9] J. Lin, J. Wong, J. Nichols, A. Cypher, and T. A. Lau. End-user programming of mashups with Vegemite. In *IUI*, 97–106, 2009.
[10] L. Liu, C. Pu, and W. Han. XWRAP: An XML-enabled wrapper construction system for web information sources. In *ICDE*, 611–621, 2000.
[11] M. Liu and T. W. Ling. A rule-based query language for HTML. In *DASFAA*, 6–13, 2001.
[12] M. Marx. Conditional XPath. *TODS*, 30(4):929–959, 2005.
[13] A. O. Mendelzon, G. A. Mihaila, and T. Milo. Querying the World Wide Web. In *DIS*, 80–91, 1996.
[14] A. Sahuguet and F. Azavant. Building light-weight wrappers for legacy web data-sources using W4F. In *VLDB*, 738–741, 1999.
[15] N. Sawa, A. Morishima, S. Sugimoto, and H. Kitagawa. Wraplet: Wrapping your web contents with a lightweight language. In *SITIS*, 387–394, 2007.
[16] W. Shen, A. Doan, J. F. Naughton, and R. Ramakrishnan. Declarative information extraction using datalog with embedded extraction predicates. In *VLDB*, 1033–1044, 2007.
[17] J.-Y. Su, D.-J. Sun, I.-C. Wu, and L.-P. Chen. On design of browser-oriented data extraction system and plug-ins. *JMST*, 18(2):189–200, 2010.

APPENDIX

A. FURTHER EXAMPLES

WWW Papers with their citations. We might want to extract the most relevant papers of a scientific field together with other work citing them. Figure 11 shows the necessary user actions: **(1)** Enter "world wide web" in the search field and **(2)** press "search". From the result page we extract the title and authors of a paper and **(3)** click on its "cited by" link. Having extracted the citing papers for that paper we do the same for all papers on the current result page and **(4)** click on the "Next" to retrieve the next result page, where we continue with (3). For sake of space, we extract only the citing papers from the first page, but it is easy to extract all citing papers.

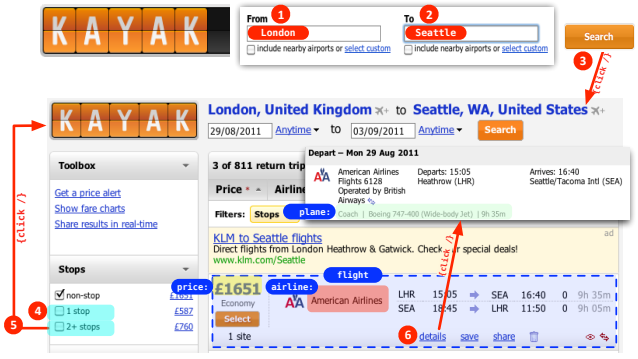
Figure 11 shows an OXPath solution for this extraction task: Lines 1–2 fill and submit the search form. Line 3 realizes the iteration over the set of result pages by repeatedly clicking the "Next" link. It uses the Kleene star borrowed from Regular XPath [12] to express this navigation. Lines 4–5 identify a result record and its author and title, lines 6–8 navigate to the cited-by page and extract the papers. The expression yields nested records of the following shape:

```
<paper><title>The diameter of the world wide web</title>
  <authors>R Albert, H Jeong, AL Barabasi</authors>
  <cited_by><title>Emergence of scaling in random ... </title>
    <authors>AL Barabasi...</authors></cited_by>...</paper>
```

Stock Quotes from Yahoo Finance. Figure 10 illustrates an OXPath expression extracting stock quotes from Yahoo Finance. In particular, note the use of optional predicates (`[? ]`) for conditional extraction: If the change is formatted in red, it is prefixed with a minus, otherwise with a plus.

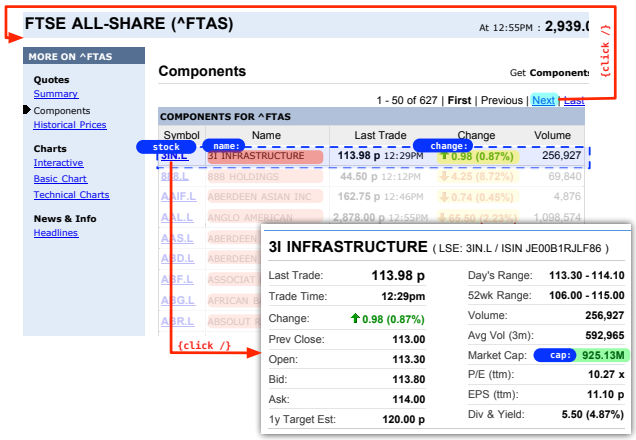
B. SYNTACTIC RESTRICTIONS

As extraction markers and actions have side effects, the grammar in Figure 2 omits a few restrictions necessary to avoid undesirable interplay of expression markers and actions with functions, predicates and sorting operators: **(1)** Actions and extraction markers may not occur in extraction marker values, arguments of functions and comparison operators, or in bracketed expressions. **(2)** Extraction markers may only occur in a predicate, if there is a marker on




```
doc("kayak.co.uk")//input#origin/{"London"}
 /following::input#destination/{"Seattle"}
 /following::input[@name='Search']/{click /}
 //*#stops1/{click }/following::*#stops2/{click /}
 //tbody.resultrow:<flight>[.//a.results_price:<price=(.)>]
   [.//a.resultdetaillink/{click /}
    //*.flight_detailsExtra:<plane=(substring-before(.,'|'))>]
```

Figure 9: OXPath for Flights to Seattle




```
doc("uk.finance.yahoo.com/q/cp?s=%5EFTAS")
  //table[tbody/tr/td[.='Name']]
  //tr[not(position() = 1)]:<stock>[td[2]:<name=string(.)>]
    [td[4][? .[img/@alt='Up']:<change=concat('+',.)>]
           [? .[img/@alt='Down']:<change=concat('-',.)>]]
```

Figure 10: OXPath for stock quotes

the path leading to the predicate and the last such marker does not extract a value. **(3)** Extraction markers that extract a value may not occur outside a predicate.

In addition, due to OXPath operating over multiple HTML pages rather than a single XML document, we also enforce the following: **(4)** The `id` function is replaced with the CSS-style `#` node-test. **(5)** OXPath does not allow sorting of node sets with nodes from different pages. To that end, bracketed expressions such as `(//a | //b)` are not allowed. **(6)** OXPath expressions always start with an explicit document access `doc`(uri).

C. FULL OXPATH SEMANTICS

In Section 2.2 we highlight where the OXPath semantics differs from the one for XPath. Table 2 and 3 give the full OXPath value and extraction semantics, though we omit for space reasons those rules in the extraction semantics that recursively decompose the expression. F_f, B_o and U_o are the semantic functions on the nodes corresponding to the functions and operators of XPath, extended by the OXPath `.` and `#` node-tests and `#=` and `.=` operators. For simplicity, we disallow positional functions outside qualifiers.

The extraction semantics $[\![expr]\!]_{\mathsf{E}}(c)$ for OXPath in Table 3, takes context tuple c and extracts an output tree from the input page tree. The semantics is straightforward except for extraction markers discussed below: For expressions with sub-expressions using a different context, we retrieve the new context using the node semantics $[\![\,]\!]_{\mathsf{N}}$ and return the markers extracted by all sub-expressions. E1 and E3 show this case exemplary for paths and predicates. For all other expressions not shown here, we just collect all extraction markers returned by their subexpressions (if there are any), regardless of the (value) semantics of the involved expressions.

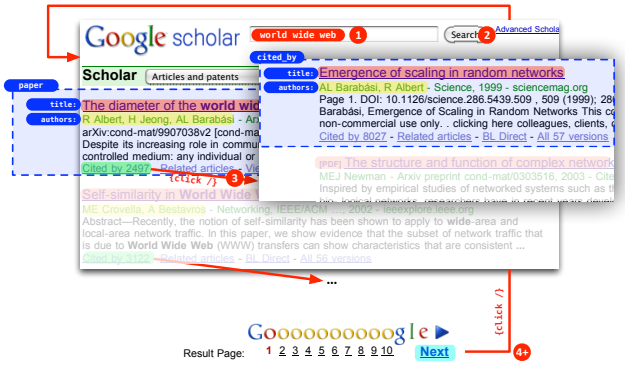
Extraction markers are treated in rule E7.1 and E7.2: In rule E7.1 for *markers without extracted values*, a tuple $R_M(\mathrm{OUT}(c'.n, M))$ is extracted for each reached tuple c' in $C = [\![step]\!]_{\mathsf{N}}(c)$ and related to its parent match via $R_{\mathsf{child}}(\mathrm{OUT}(c'.n, M), c'.p)$. For *markers with values,* we evaluate additionally in rule E7.2 the value expression v for each reached tuple c': We take the value returned by $[\![v]\!]_{\mathsf{E}}(c')$ (which is a string or other scalar value) and add a child text-node to $\mathrm{OUT}(c'.n, M)$ with content $[\![v]\!]_{\mathsf{E}}(c')$ (i.e., is added to $R_{[\![v]\!]_{\mathsf{E}}(c')}$.

V1	$[\![path]\!]_{\mathsf{V}}(c)$	$= [\![path]\!]_{\mathsf{N}}(c)$
V2	$[\![fct(e_1,\ldots,e_k)]\!]_{\mathsf{V}}(c)$	$= F_{fct}([\![e_1]\!]_{\mathsf{V}}(c),\ldots,[\![e_k]\!]_{\mathsf{V}}(c))$
V3	$[\![value]\!]_{\mathsf{V}}(c)$	$= value$
V4	$[\![e_1\ op\ e_2]\!]_{\mathsf{V}}(c)$	$= B_{op}([\![e_1]\!]_{\mathsf{V}},[\![e_2]\!]_{\mathsf{V}})$
V5	$[\![op\ e]\!]_{\mathsf{V}}(c)$	$= U_{op}([\![e]\!]_{\mathsf{V}})$
N1	$[\![estep/path]\!]_{\mathsf{N}}(c)$	$= \{c'' \mid c' \in [\![estep]\!]_{\mathsf{N}}(c) \wedge c'' \in [\![path]\!]_{\mathsf{N}}(c')\}$
N2	$[\![axis::nodes]\!]_{\mathsf{N}}(c)$	$= \{\langle n', c.p, c.l\rangle \mid R_{axis}(c.n,n') \wedge n' \in R_{nodes}\}$
N3	$[\![step[q]]\!]_{\mathsf{N}}(c)$	$= \{c' \in [\![step]\!]_{\mathsf{N}}(c) \mid [\![q]\!]_{\mathsf{B}}(\langle c'.n, c'.l, c'.l\rangle)\}$
N4	$[\![step_{\pm}[qp]]\!]_{\mathsf{N}}(c)$	$= \{c' \in [\![step_{\pm}]\!]_{\mathsf{N}}(c) \mid C = [\![step_{\pm}]\!]_{\mathsf{N}}(c) \wedge$ $[\![\text{REWRITE}_{\pm}(qp,C,c')]\!]_{\mathsf{B}}(\langle c'.n, c'.l, c'.l\rangle)\}$
N5	$[\![\{action\ /\}]\!]_{\mathsf{N}}(c)$	$= \{\langle n', c.p, c.l\rangle\}$ with n' such that $R_{action}(c.n,n')$
N6	$[\![\{action\}]\!]_{\mathsf{N}}(c)$	$= [\![\text{AFP}(action,c.n)]\!]_{\mathsf{N}}([\![\{action\ /\}]\!]_{\mathsf{N}}(c))$
N7	$[\![step : \langle M[= v]\rangle]\!]_{\mathsf{N}}(c)$	$= \{\langle c'.n, c'.p, \text{OUT}(c'.n,M)\rangle \mid \exists c' \in [\![step]\!]_{\mathsf{N}}(c)\}$
N8	$[\![(path)*]\!]_{\mathsf{N}}(c)$	$= \{c_r \mid \exists r \geq 0\ \forall 0 \leq s \leq r :$ $c_{s+1} \in [\![path]\!]_{\mathsf{N}}(c_s) \wedge c_0 = c\}$
N9	$[\![(path)*\{v,w\}]\!]_{\mathsf{N}}(c)$	$= \{c_r \mid \exists v \leq r \leq w\ \forall 0 \leq s \leq r :$ $c_{s+1} \in [\![path]\!]_{\mathsf{N}}(c_s) \wedge c_0 = c\}$
N10	$[\![(expr)[qp]]\!]_{\mathsf{N}}(c)$	$= \{c' \mid c' \in C \wedge C = [\![expr]\!]_{\mathsf{N}}(c) \wedge$ $[\![\text{REWRITE}_{+}(qp,C,c')]\!]_{\mathsf{B}}(\langle c'.n, c'.l, c'.l\rangle)\}$
N11	$[\![doc(uri)]\!]_{\mathsf{N}}(c)$	$= F_{doc}(uri)$
B1	$[\![expr]\!]_{\mathsf{B}}(c)$	$= U_{Boolean}([\![expr]\!]_{\mathsf{V}}(c))$

Table 2: Value Semantics of OXPath

E1	$[\![estep/path]\!]_{\mathsf{E}}(c)$	$= [\![estep]\!]_{\mathsf{E}}(c) \cup \bigcup_{c' \in [\![estep]\!]_{\mathsf{N}}} [\![path]\!]_{\mathsf{E}}(c')$
E3	$[\![step[q]]\!]_{\mathsf{E}}(c)$	$= [\![step]\!]_{\mathsf{E}}(c) \cup$ $\bigcup_{c' \in [\![step]\!]_{\mathsf{N}}(c)} [\![q]\!]_{\mathsf{E}}(\langle c'.n, c'.l, c'.l\rangle)$
E7.1	$[\![step : \langle M\rangle]\!]_{\mathsf{E}}(c)$	$= [\![step]\!]_{\mathsf{E}}(c) \cup \{R_M(\text{OUT}(c'.n,M)),$ $R_{\text{child}}(\text{OUT}(c'n,M),c'.p) \mid c' \in [\![step]\!]_{\mathsf{N}}(c)$
E7.2	$[\![step : \langle M = v\rangle]\!]_{\mathsf{E}}(c)$	$= [\![step]\!]_{\mathsf{E}}(c) \cup \bigcup_{c' \in C} [\![v]\!]_{\mathsf{E}}(c') \cup$ $R_M(\text{OUT}(c'.n,M)), R_{\text{child}}(\text{OUT}(c'.n,M),c'.p),$ $R_{[\![v]\!]_{\mathsf{N}}(c')}(c'')), R_{\text{child}}(c'', \text{OUT}(c'.n,M)) \mid$ $c' \in [\![step]\!]_{\mathsf{N}}(c) \wedge c''$ new output node

Table 3: Extraction Semantics of OXPath (partial)




```
doc("scholar.google.com")/descendant::field()[1]/{"world..."}❶
    /following::field()[1]/{click/}❷
    /(//a[contains(string(.),'Next')]/{click/})*❹
    //div.gs_r:<paper>[.//h3:<title=string(.)>]
      [.//*.gs_a:<authors=substring-before(.,' - ')>]
      [.//a[.#='Cited by']/{click /}❸
       //div.gs_r:<cited_by>[.//h3:<title=.>]
          [.//*.gs_a:<authors=substring-before(.,' - ')>]]
```

Figure 11: Finding an OXPath through Google Scholar

D. OXPATH PROPERTIES

No sorting across Pages. OXPath avoids sorting context sets which contain nodes from different pages, since it is unclear how to order nodes from different pages, without first retrieving (and thus buffering) those pages.

PROPOSITION 7 (NO NODE SORTING ACROSS PAGES). *The evaluation of an OXPath expression never requires sorting context sets which contain nodes from different pages.*

PROOF. OXPath requires sorting only for positional qualifiers in Rule N4 and bracketed expressions in Rule N11 (Appendix C). In both cases, the function $\text{REWRITE}_{\pm}(q_{\pm},C,c')$ sorts the tuples in the context set C and determines the position of c' within C. Thus, it suffices to show, that C in Rule N4 and N11 never contains nodes from different pages.

This holds in N4, since $C = [\![step]\!]_{\mathsf{V}}(c)$ is computed from a single axis navigation $axis :: nodes$ (N2) and a sequence of (positional) qualifiers (N3 and N4) and markers (N7). Since N4 always results in a context set with nodes from the same page, and since N2-N47 can only remove nodes from the context set, C must contain nodes from a single page only.

This holds in N11 as bracketed expressions may not contain actions by Appendix B and neither N2–N4 nor N7–N10 return nodes from a different page than the context node, if no nested actions occur in the expressions. □

Streaming Extraction Tuples. OXPath's semantics does not require any further processing on result tuples, but allows them to be streamed out as they are extracted. Extracted tuples are never modified, deleted, or re-accessed again.

PROPOSITION 8 (NO OUTPUT BUFFER). *The evaluation of OXPath expressions requires no output buffer.*

PROOF. Only rules E7.1 and E7.2 in Table 3 create output tuples. Each created tuple is unique, as OUT is injective. Furthermore, when the tuples are created the parent output nodes are known by construction and thus no buffering is needed to create the proper tuples in R_{child}.

All other rules in the extraction semantics only collect the tuples returned by their sub-expressions. Since no duplicate tuples are created this collection does require no buffering. □

Intuitively, this holds as the structure of the output tree reflects the structure of the OXPath expression and thus parent nodes are always created before their children nodes.

Consider the OXPath expression $expr[p_1][p_2]$. If p_1 contains extraction expressions, the extracted tuples are returned whether p_2 matches or not.

Requiring that tuples extracted by p_1 are returned only if p_2 matches, would require an unbounded buffer, as the visited pages and extracted results for p_1 are both unbounded. Furthermore, we can achieve the same effect in the existing OXPath semantics at the cost of an increased query size (and thus evaluation time): The OXPath expression $expr[p_2'][p_1][p_2]$ where p_2' is obtained from p_2 by removing all extraction markers. p_2' matches if and only if p_2 matches, as extraction markers do not affect matching, and tuples extracted by p_1 are only returned if p_2 matches.

Complexity. We offer a proof for the theorem stated in Section 2.3.

PROOF OF THEOREM 1. *Data Complexity:* From all extensions to XPath, only the Kleene star causes an increase in complexity: Actions are assumed to take constant time, extraction markers do

not require additional memory as they are streamed out, and the additional axis does not introduce further complexity.

XPath 1.0 without string concatenation and multiplication has data complexity LOGSPACE [3]. Each Kleene star expression can be realized as transitive, reflexive closure of the Kleene star repeated expression, therefore we arrive at NLOGSPACE data complexity for OXPath without string concatenation and multiplication.

Combined Complexity: PTIME-hardness follows immediately from the PTIME-hardness for XPath query evaluation [3].

To evaluate an OXPath query, we process query left to right. Since evaluating each subexpression requires at most polynomial time, our overall evaluation runs in polynomial time as well.

If a subexpression is an XPath expression which makes no use of our extensions except for one of the additional axes, we rely on one of the known polynomial time algorithms for XPath [7]. If the expression is a marker, we stream out the extracted matching, which is in polynomial time, too, and actions are assumed to take constant time.

The only remaining case is the Kleene star: If the Kleene star repeated expression contains a non-nested action, we know that each iteration of the repeated expression leads to a new page. Consequently, there are at most input size many iterations. If the Kleene star does not contain an action it is an ordinary Regular XPath expression which can be evaluated in polynomial time [12]. □

E. PROOFS FOR PAAT ALGORITHM

PROOF OF PROPOSITION 2. We call an output node o *generated* by a marker M and input node n if $\text{OUT}(n,M) = o$. n and M uniquely identify o among all output nodes as OUT is injective. $\top$ is generated by any marker and input node. A sub-expression e of Expr has *nesting level* i in Expr, if it is nested inside i predicates in Expr.

In OXPath$_{\text{basic}}$, `eval_` is called for a sub-expression e only with context tuples (i,p,s) for which it holds that there is a marker P and a marker S, such that all the parent output nodes p are generated by P and all the sibling output nodes s by S. P and S can be statically determined for each e: P is the first extraction marker with a lower nesting level on the path from e to the start of the expression. S is the first extraction marker with the same nesting level on that path.

Therefore, each context node i occurs with at most n different parent output nodes and n different sibling nodes, and for each subexpression the number of context tuples is bounded by $O(n^3)$. The size l of the lookup table Lookup is thus bounded by $O(q \cdot n^4)$.

As already remarked, for OXPath$_{\text{basic}}$, `eval` just passes the evaluation through to `eval_` and does not affect the complexity.

The overall complexity of OXPath$_{\text{basic}}$ is bounded by $O(l \cdot o \cdot q)$ where o is the time for evaluating OXPath functions, extraction markers, and node-tests (see F, B, and U in Appendix C). o is bound by $O(n^2 \cdot q)$ for all operations and functions in XPath, see Theorem 6.6 in [7], and extraction markers do not affect this complexity. This includes the cost for sorting needed in evaluating a positional predicate (which is avoided in [7] by extending the context tuple) at $O(n \cdot \log n)$.

Thus the overall time is bound by $O(n^6 \cdot q^2)$.

For space, we have a bound of $O(l \cdot v \cdot q)$ where l is as above and v is the maximum size of a value. Again from Theorem 6.6 [7] we have a bound of $O(n \cdot q)$ for v and thus an overall space bound of $O(n^5 \cdot q^2)$.

It is worth noting that this is an increase of a multiplicative factor of n^2 over full XPath in both cases. That is due to the fact that in our case there is no functional dependency from the context tuple to the result value and that our context tuples are increased by the marker information. We can, however, omit the parent node (as we allow access to positional information only in positional predicates). □

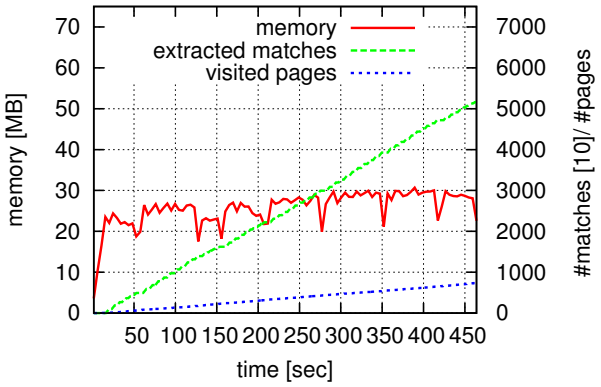


Figure 12: Scaling OXPath: many results

PROOF FOR PROPOSITION 3. OXPath$_{-\text{Kleene}}$ includes actions and thus Expr may access more than one page. However, the page tree (see Section 2.2) is navigated one page at a time by `eval`: A new page is only reached by an action in Line 11. For each such page the remaining expression is evaluated and then all lookup entries for that page are deleted (Line 16).

This buffer management is implemented in `eval` by means of the prot flag and the three page management functions `doc()`, `getPage`, and `FreeMem`: `FreeMem`($expr$) deletes all buffers associated with $expr$. `doc`(uri,prot) returns the root node of the page with the given uri. If prot is **false**, it replaces the previous page (in the underlying browser) and deletes memorized results for the previous page. `getPage`($node$,$action$,prot) executes $action$ on the given $node$. If prot is **true**, it first clones the page in the browser before executing the action, otherwise it uses the same browser window and deletes all memoized results for the old page.

Thus, we traverse the page tree in a depth-first fashion. As there is no repeated traversal of actions in a Kleene star free expression, we can traverse at most to a depth of $O(q)$ (if each step in Expr is an action) into the page tree.

Therefore, the lookup table in `eval_` is bounded by $O(n^4 \cdot q \cdot \min(q,d)) = l$.

Overall in the evaluation up to $O((p \cdot n)^4 \cdot q) = t$ context tuples are created and the complexity of OXPath$_{\text{basic}}$ is bounded by $O(t \cdot o_a \cdot q)$ where o is the time for evaluating OXPath functions, extraction markers, and node-tests, but now also actions. If only absolute actions occur o_a is bounded by $O(n^2 \cdot q)$ (see proof of Proposition 2) as actions do not affect the complexity as for each context node a single page root is returned in constant time. This assumes that action execution is constant. Though action execution may require the browser engine to evaluate an arbitrary, potentially non-terminating Javascript program, in practice action execution is almost always near instant. Thus for OXPath$_{-\text{Kleene}}$ without contextual actions we obtain $O((p \cdot n)^6 \cdot q^2)$ as time and $O(n^5 \cdot q^3)$ as space limit in analogy to the case for basic OXPath. For the space bound it is worth noting that the size of values remains the same as in basic OXPath (and XPath) as actions may not occur in parameters of functions.

For contextual actions, the same observation as for absolute actions applies except that we also reevaluate the action-free prefix on the retrieved page, if it has been modified. In the worst-case, every step in Expr carries a contextual action and the action-free pre-

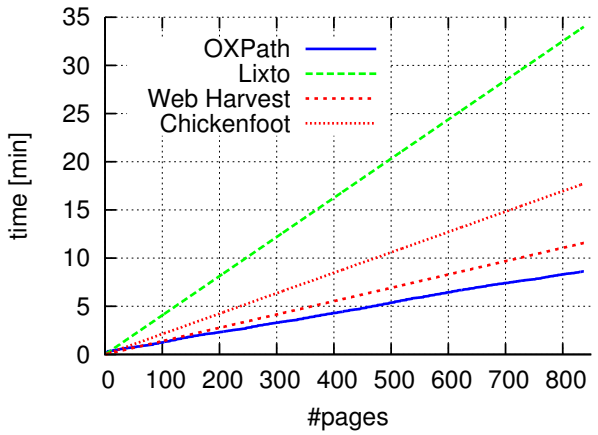


Figure 13: Comparison: Absolute evaluation time, <850 p.

fix is always $O(q)$. Then we evaluate the action-free prefix $O(q)$ times and the overall time spent evaluating action-free prefixes is bounded by $O(n^6 \cdot q^3)$ and space by $O(n^5 \cdot q^2)$.

Thus the overall time is bound by $O((p \cdot n)^6 \cdot q^2 + n^6 \cdot q^3) \leq O((p \cdot n)^6 \cdot q^3)$ and the space bound remains the same as for absolute actions. □

PROOF OF THEOREM 4. If we also consider the bounded Kleene star, we have to observe that the number of pages buffered at one time is no longer necessarily bound by the size of the expression, as we do not know in advance how often a Kleene star that contains an expression that navigates to a new page matches. Rather the number of buffered pages is bound by the maximum level d of any accessed page in the page tree for Expr. If $d < \infty$, the above complexity applies: The time complexity remains unchanged from $\text{OXPath}_{-\text{Kleene}}$, but the space complexity increases, as the number of memorization tables depends on the query size *and* the maximum number of successful expansion steps for a Kleene star expression, which is bounded by d. Again the size of values remains the same as actions are not allowed in parameters of functions and operators and Kleene stars without contained actions yield at most n nodes and thus, with Theorem 6.6 [7], values of at most $O(n \cdot q)$ size. □

PROOF OF COROLLARY 5. For expressions with unbounded Kleene stars the length of the paths in the navigation tree from the root that can be reached by an OXPath expression is bounded by the expression. Thus, the corollary follows immediately from Theorem 3. □

PROOF OF THEOREM 6. Consider the series of expressions $e_d = \texttt{doc(w)}r_d$ with $r_d = \texttt{//*/\{action\}}r_{d-1}$ for $d > 1$ and $r_1 = \texttt{self::*}$. Assume further that in the page tree of the expressions e_d each page has at least two nodes with an action that leads again to another page of this form. e_d executes `{action}` on all nodes of page w, and continues recursively from all pages thus reached. It returns the roots of the pages finally reached.

When we evaluate e_d with PAAT, we access the page tree up to a depth of d and use exactly d page buffers. This holds, since the accessed page tree has at least two branches at each page.

Any other algorithm A must load the leaves of the accessed page tree of PAAT as these nodes are the result of evaluating $[\![e_d]\!]_V$. To visit such a leaf node l of the accessed page tree, we have to load its parent p first, because without prior knowledge all children of p are only accessible by performing `{action}` on the respective node in p. Thus, A must have loaded all $d-1$ ancestors of l to finally access l. Assume that l is the first leaf reached by A. Then, A must buffer all $d-1$ ancestors in addition to l, because for each ancestor of l there are further children to be visited. □

F. COMPARATIVE EVALUATION

For the comparative evaluation we implement the following OXPath expression in the other systems:

```
doc("scholar.google.com")/descendant::field()[1]/{"Seattle"}
      /following::field()[1]/{click /}/
           (//a[string(.)#="Cited by"]/{click/})*{0,3}
```

An equivalent Web Harvest program uses 54 lines (see `diademe-project.info/oxpath`, a Chickenfoot script uses 27 (see below). The other tools use visual interfaces.

Chickenfoot Script for Google Scholar

```
go("http://scholar.google.co.uk/")
// For the first two fields, Chickenfoot's label heuristics fail and
      thus
// we need to select the fields using XPath
click(new XPath("/HTML[1]/BODY[1]/CENTER[1]/FORM[1]
                /TABLE[1]/TBODY[1]/TR[1]/TD[2]/INPUT[1]"))
enter(new XPath("/HTML[1]/BODY[1]/CENTER[1]/FORM[1]/
                TABLE[1]/TBODY[1]/TR[1]/TD[2]/INPUT[1]"),"world wide
                     web")
click("Search button")

m=find("Cited by");
for (i=1; i<=m.count;i++) {
  click(new XPath("/descendant::a[contains(.,\"Cited by\")][" + i
        +"]"));
  n=find("Cited by");
  v=find(new XPath("//h3"));
  output(v);
  for (j=1; j<=n.count;j++) {
    click(new XPath("/descendant::a[contains(.,\"Cited by\")][" + j
          +"]"));
    z=find(new XPath("//h3"));
    output(z);
    o=find("Cited by");
    for (k=1; k<=o.count;k++) {
      click(new XPath("/descendant::a[contains(.,\"Cited by\")][" + j
            +"]"));
      zz=find(new XPath("//h3"));
      output(zz);
      back();
    }
    back();
  }
  back();
}
```


Acknowledgements

The research leading to these results has received funding from the European Research Council under the European Community's Seventh Framework Programme (FP7/2007–2013) / ERC grant agreement no. 246858. This work was carried out in the wider context of the networking programme FoX – Foundations of XML, FET-Open grant agreement number FP7-ICT-233599, of which Oxford University is a partner. The views expressed in this article are solely those of the authors.