VLDB Journal manuscript No.
(will be inserted by the editor)

OXPATH: A Language for Scalable Data Extraction, Automation, and Crawling on the Deep Web

Tim Furche · Georg Gottlob · Giovanni Grasso · Christian Schallhart · Andrew Sellers

10 Feb 2012

Abstract The evolution of the web has outpaced itself: A growing wealth of information and increasingly sophisticated interfaces necessitate automated processing, yet existing automation and data extraction technologies have been overwhelmed by this very growth.

To address this trend, we identify four key requirements for web data extraction, automation, and (focused) web crawling: **(1)** Interact with sophisticated web application interfaces, **(2)** precisely capture the relevant data to be extracted, **(3)** scale with the number of visited pages, and **(4)** readily embed into existing web technologies.

We introduce OXPATH as an extension of XPATH for interacting with web applications and extracting data thus revealed – matching 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 experimentally validate the theoretical complexity and demonstrate that OXPATH's resource consumption is dominated by page rendering in the underlying browser. With an extensive study of sub-languages and properties of OXPATH, we pinpoint the effect of specific features on evaluation performance. Our experiments show that OXPATH outperforms existing commercial and academic data extraction tools by a wide margin.

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, such as Amazon, Facebook, or 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 designed 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.

For automation, data accessible to humans through existing interfaces must be transformed into structured data, e.g., each gray `span` with class `source` on Google News should be recognized as news source. These observations 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.

Previous approaches to web extraction languages [34, 46] use a declarative approach akin to OXPATH; however, mainly due to their age, they often do not adequately facilitate deep web interaction, e.g. form submissions. Also, they do not provide native constructs for page navigation, apart from retrieving pages for given URIs. Where scripting is addressed [12,47], the simulation of user actions is neither declarative nor succinct, but rather relies on imperative scripts and standalone, heavy-weight extraction interfaces. Lixto [12], Web Content Extractor [2], and Visual Web Ripper [3] are moving towards interactive wrapper generator frameworks, recording user actions in a browser and replaying these actions for extracting data. As large com-

Oxford University Department of Computer Science
Wolfson Building, Parks Road, Oxford OX1 3QD
E-mail: firstname.lastname@cs.ox.ac.uk

mercial extraction environments, their feature set goes beyond the scope of OXPATH. They all emphasize the visual aspect of wrapper generation that ease the design of extraction tasks by specifying only few examples, mainly selecting layout elements on the rendered page. Further, Lixto adopts tree generalization techniques to produce more robust wrappers, whereas Web Content Extractor allows to write complex user-defined text manipulation scripts. However, despite their feature richness, none of these systems addresses memory management and our experimental evaluation (Section 6) demonstrates that such systems indeed take memory linear in the number of accessed pages.

OXPATH restricts its focus to data extraction in the context of deep web crawling. This sets OXPATH apart from information extraction (IE) systems, that aim at extracting structured data (entities and relations) from unstructured text. Systems like [11,23] and [19] extract factual information from textual description (mainly by lexico-syntactic patterns) and HTML tables on the web, respectively. Though, OXPATH could be extended to support libraries of user defined functions to refine extraction from text (along the lines of procedural predicates in [46]), this is out of the scope of this paper. Whereas OXPATH takes advantage of the structures on the web (possibly revealed through simulating user interaction) to extract, e.g., infoboxes on Wikipedia or products on Amazon along with all their reviews, the task of extracting, e.g., named entities from these reviews is not addressed by OXPATH, but delegated to post-processing of the extracted information, e.g., using a IE systems.

As far as web automation tools are concerned, though some of them [17,31] can deal with scripted web applications, they are tailored to single action sequences and prove to be inconvenient and inefficient in large-scale extraction tasks requiring multi-way navigation (Section 6).

It is against this backdrop that we introduce OXPATH, a careful, declarative extension of XPATH for interacting with web applications to extract information revealed during 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* with action types and context elements, such as the links to click on, or the form field to fill. **(ii)** In contrast to most previous web extraction and automation tools, actions have a *formal semantics* (Section 4.4) based on a **(iii)** novel multi-page *data model for web applications* that captures both page navigation and modifications to a page (Section 4.3).

2—Expressive and precise. OXPATH inherits the *precise* selection capabilities of XPATH (rather than heuristics for element selection as in [17]) and extends them: **(i)** OXPATH allows *selection based on visual features* by exposing all *CSS properties* 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 further such link. **(iii)** OXPATH provides *intensional axes* to relate nodes through multiple conditions, e.g., to select all nodes which are at the same vertical position *and* have the same color as the current node. **(iv)** OXPATH enables the identification of *data for extraction,* which can be assembled into (hierarchical) records, regardless of its original structure. **(v)** Based on the formal semantics of OXPATH (Section 4.4), we show that its extensions considerably increase the language's expressiveness (Section 4.10).

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 5). 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 4.10) and is only slightly higher than that of XPATH (Section 5). **(iii)** We also show that OXPATH is highly parallelizable (Section 4.10). **(iv)** We provide a *normal form* which reduces the size of the memoization tables during evaluation and *rewriting rules* to normalize arbitrary expressions (Section 4.9). **(v)** We verify these theoretical results in an *extensive experimental evaluation* (Section 6), 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, such as Java, XQUERY, or Javascript. Following the spirit of XPATH, we provide an API and host language to facilitate OXPATH's interoperation with other systems.

Bonus: Open Source. We provide our OXPATH implementation and API at `http://diadem.cs.ox.ac.uk/oxpath`, for distribution under the new BSD license.

OXPATH has been employed within DIADEM [24], a domain-driven, large-scale data extraction framework developed at Oxford University, proving to be a practically viable tool for **(1)** succinctly describing web interaction and extraction tasks on sophisticated web interfaces, for **(2)** generating and processing such task descriptions, and for **(3)** efficiently executing these wrappers on the cloud.

This paper extends [25] in three main aspects:

(1) It clarifies the design of OXPATH and introduces *intensional axes* (Section 4.7) as a further instrument for extracting data from rich web applications. With intensional axes, the OXPATH user can on-the-fly specify new types of relations between elements of a web page,


```
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=(.)>]
```

Fig. 1 Finding an OXPATH through Amazon.

e.g., to select all paragraphs with the same color and dimension.

(2) The page-at-a-time evaluation algorithm (Section 5) has been further refined to cater to these additions and to improve the complexity bounds from [25]: By splitting and specializing the memoization table, we achieve a reduction by up to a factor of n in time and memory. An additional factor of n reduction (at a slight increase in expression size) can be achieved by applying a new normalization rewriting (Section 4.9).

(3) An extensive study (Section 5.7) of the impact of the main features of OXPATH, extraction markers, actions, and Kleene-star iteration, on evaluation performance is used to define a normal form for OXPATH (Section 4.9) together with a sound and complete rewriting.

1.1 A Gentle Introduction to OXPATH

OXPATH extends XPATH with five concepts: Actions to navigate the interface of web applications, means for interacting with highly visual pages, intensional axes to identify nodes by multiple relations, extraction markers to specify data to extract, and the Kleene star to extract from a set of pages with unknown extent.

Actions. For simulating user actions such as clicks or mouseovers, OXPATH introduces **(i)** *contextual*, as in **`{click}`**, and **(ii)** *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 continue at DOM roots, contextual actions continue at those nodes in the new DOM matched by the *action-free prefix* (Section 4.4) of the performed action. This prefix 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 **(i)** a new axis for accessing CSS DOM node properties and **(ii)** 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 field navigation, we introduce the node-test `field()`, that relies on the **`style`** axis to access the computed CSS style to exclude non visible fields, e.g., **`/descendant`**`::field()[1]` selects the first visible field in document order.

Intensional Axis. XPATH is not able to express queries where nodes from two node sets are related by more than one relation. Also, the set of relations that can be used is fixed. To overcome this limitation, OXPATH introduces intensional axes: Users can identify nodes by intentionally defining relations between node pairs, as needed. For example, exploiting the **`style`** axis, the following expression selects all nodes with the same font and color as a hyperlink:

```
doc("www.google.com")//a[@href]/
   [$lhs/style::color = $rhs/style::color and
    $lhs/style::font-size = $rhs/style::font-size]::*
```

Intensional axes allow to compare node sets by more than one relation with the use of the reserved variables `$lhs` and `$rhs`.

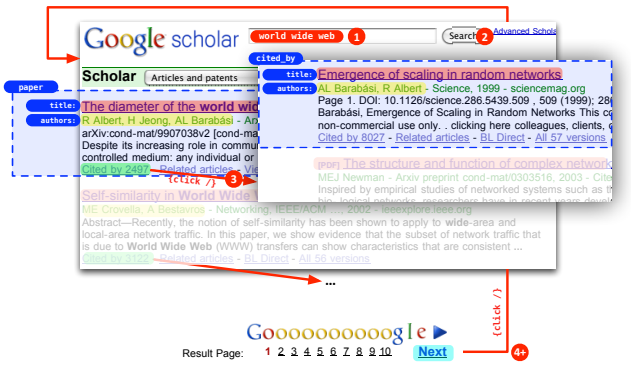
Extraction Marker. In OXPATH, we introduce a new kind of qualifier, the *extraction marker*, to identify nodes as representatives for records and to form attributes. For example,

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

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

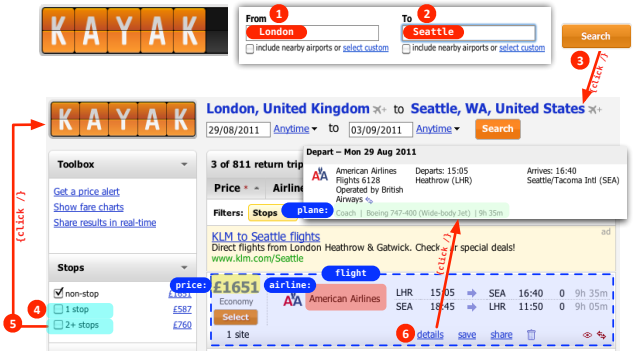
```
<story><title >Tax cuts ...</title>
       <source>Washington Post</source>
       <source>Wall Street Journal</source></story>
```

The nesting in the result mirrors the structure of the OXPATH expression: extraction markers in a particular predicate, such as `title` and `source`, yield attributes associated with the last marker outside this predicate, in our example the `story` marker.




```
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(.,' - ')>]]
```

Fig. 2 Finding an OXPATH through Google Scholar




```
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(.,'|'))>]
```

Fig. 3 OXPATH for Flights to Seattle

Kleene Star. We follow [35] in adding the Kleene star. The following expression, e.g., 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}`.

2 Application Scenario

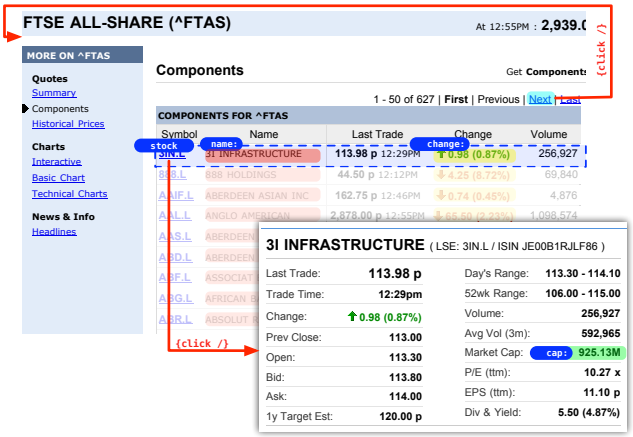
This section showcases some typical applications of OXPATH: interacting with a visual, scripted interface to find flights on Kayak, extracting books on Amazon, information on relevant academic papers and their citations on Google Scholar, and stock quotes from Yahoo Finance.

History Books on Seattle. To extract data about history books on Seattle as offered on `amazon.co.uk`, a user has to perform the following sequence of actions to retrieve the page listing these books (see Fig. 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, **(5)** and open the details page for retrieving further details. Fig. 1 shows an OXPATH expression that realizes this extraction (each action is numbered according to the involved step). Lines 1–5 implement the above steps: 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 not contextual but absolute (with an added `/` before the closing brace) to continue 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 to match the "History" text.

For the obtained books, we extract their title, price, and publisher in step **(5)**, as shown in Lines 6–8 of Fig. 1: The element with class `result` serves as indicator of `book` records, denoted by 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 obtain the page for the individual book, where we find and extract the publisher. Finally, we extract the price from the previous page – without caring for the order in which the pages are visited during extraction. OXPATH buffers pages when necessary, yet guarantees that the number of buffered pages is independent of the number of visited pages.

WWW Papers with their citations. We might want to extract the most relevant papers of a scientific field together with other works citing them. Fig. 2 shows the sequence of necessary user actions (each one numbered accordingly): **(1)** Enter "world wide web" in the search field and **(2)** press "search". From the result page we extract, for each paper, its title and authors paper and **(3)** click on its "cited by" link. To retrieve the next result page, we **(4)** click on "Next" and continue with **(3)**.

Fig. 2 shows an OXPATH solution for this extraction task: Lines 1–3, navigate to Google Scholar, fill and submit




```
doc("http://uk.finance.yahoo.com/q?s=%5EFTAS")
//table[tbody/tr/td[.='Name']]
//tr[not(position() = 1)]:<stock>
[td[2]:<name=string(.)>]
[td[4][? .[img/@alt='Up']:<changeUp=concat('+',.)>]
[? .[img/@alt='Down']:<changeDown=concat('-',.)>]]
       /{click /}//tr[.~'Market Cap']
           [.//span:<marketCap=string(.)>]
```

Fig. 4 OXPATH for stock quotes

the search form. Line 4 realizes the iteration over the set of result pages by repeatedly clicking the "Next" link, denoted with a Kleene star. Lines 5–6 identify a result record and its author and title, lines 7–9 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..</title>
 <authors>R Albert, H Jeong, ...</authors>
 <cited_by>
    <title>Emergence of scaling in ... </title>
    <authors>AL Barabasi...</authors></cited_by>
 <cited_by> .....</paper>
```

Flights from Kayak. Our next example (Fig. 3) demonstrates using OXPATH for finding non-stop flights to Hyderabad. This example illustrates the interaction with a heavily scripted page for refining the search results from any stops to only non-stop flights. Here search results are only exposed through serial user action input, which is allowed via well-formed OXPATH expressions. After submitting the form, we wait for the first result page to load and refine our results by selecting only non-stop flights and flights with a single stop. We can then extract flights, along with features such as price, airline, and plane. In this example, we use the # selector from CSS, analogous to the . selector used in this previous example, that filters elements based upon their `id` attributes.

Stock Quotes from Yahoo Finance. Fig. 4 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.

$\mathsf{parent} = \mathsf{child}^{-1}$
$\mathsf{descendant} = \mathsf{child}^{+}$
$\mathsf{ancestor} = \mathsf{parent}^{+}$
$\mathsf{descendant\text{-}or\text{-}self} = \mathsf{descendant} \cup \mathsf{self}$
$\mathsf{ancestor\text{-}or\text{-}self} = \mathsf{ancestor} \cup \mathsf{self}$
$\mathsf{following} = \mathsf{descendant\text{-}or\text{-}self} \circ \mathsf{nextsibling}^{+} \circ \mathsf{ancestor\text{-}or\text{-}self}$
$\mathsf{preceding} = \mathsf{descendant\text{-}or\text{-}self} \circ (\mathsf{nextsibling}^{-1})^{+} \circ \mathsf{ancestor\text{-}or\text{-}self}$
$\mathsf{following\text{-}sibling} = \mathsf{nextsibling}^{+}$
$\mathsf{preceding\text{-}sibling} = (\mathsf{nextsibling}^{-1})^{+}$

Fig. 5 XPATH axes composed in terms of "primitive" tree relations child and nextsibling, their inverses, and the identity relation self.

3 Preliminaries: XPATH Data Model

XPATH is used to query XML documents modeled as unranked and ordered trees of nodes. The set of nodes within a document are given as *DOM,* with nodes of seven types, namely root, element, text, comment, attribute, namespace, and processing instruction. Documents begin at a unique root node with elements as the most common non-terminal. All nodes except comment and text have an associated name (or label). Within a document, nodes x and y are ordered by *document order*, which is defined as the binary relation $x <_{doc} y$, iff the opening tag of x occurs before the opening tag of y in the well-formed XML document. Node types and labels are formally represented in this paper through a set of unary relations, $(\mathsf{unary}_v)_{v \in \mathsf{Unary}}$ with, e.g., $\mathsf{text}, \mathsf{element}, \mathsf{a} \in \mathsf{Unary}$ (all text nodes, elements, and a-labeled nodes).

An XPATH query result has one out of four possbible data types, which is either **(1)** an unordered collection of distinct document nodes, called *node sets*, or a scalar value of type **(2)** *Boolean*, **(3)** *string*, or **(4)** *number*.

Briefly, XPATH queries are composed of functions and operators. Most importantly, node sets are specified by path expressions of concatenated steps: Each step evaluates the context nodes resulting from the previous step and consists of an axis (or child, if omitted), node test, and optional predicate expressions. XPATH axes are binary relations, relating a context node to another node in the document according to the definitions in Fig. 5 (self is given as $\{\mathsf{self}(x,x) | x \in dom\}$). In addition to self and the axes in Fig. 5, XPATH has specific axes to access attribute and namespace nodes.

Axis navigation is further refined by node tests. In addition to a wildcard node test (`node()`) covering all document nodes, XPATH defines node tests to filter by each of the unnamed node types (`text()`, `processing-instruction()`, and `comment()`). Node tests can also filter elements by name or by $*$, which is a wildcard for all elements.

Finally, steps can be filtered further with an arbitrary number of predicate expressions. Predicates contain a single input expression, and return each node from the context that evaluates to true for its input expression.

We leave further details on XPATH to Section 4, where we discuss XPATH implicitly as OXPATH's sublanguage.

4 Language

4.1 Design Principles

The design of OXPATH is guided by the following design goals derived from two core principles:

(1) Spirit of XPATH. OXPATH maintains, where possible, the principles upon which XPATH is built, in particular the use of a single, navigational expression, polynomial time evaluation, and concise syntax. Where we extend XPATH, we do so using existing web standards such as CSS or DOM events. **(I) Single Expression.** OXPATH expressions are path expressions just like plain XPATH. We choose *not* to extend OXPATH with a separate "construct" clause specifying the result of the expression (as in SPARQL, SQL, or XQuery), but rather to embed the specification of the result of the extraction into the path expression through extraction markers. This requires the shape of the expression to mirror the shape of the extracted result, a limitation, however, that is insignificant due to XPATH's flexible axes.[1] **(II) Tree result.** The result of an OXPATH expression is a tree constructed from matches for the record extraction markers and their attributes. This poses only a small limitation, as data on the web is usually presented in a hierarchical way. **(III) Polynomial time.** We design OXPATH to remain polynomial and in two-variable logic for both selection and construction (see Section 4.10). Though there are cases, where full first order-logic is necessary for extraction (e.g., to refer back to values encountered on other pages), we believe that OXPATH presents a more useful trade-off in most cases (see Section 2).

(2) Low memory. The second core principle is that OXPATH should not require an unbounded buffer for pages, but rather be able to extract from hundreds of thousands of pages with very little memory use. This is necessary for large-scale data extraction. **(I) No page identity.** OXPATH does not manage page "identity": If two links lead to the same URL, OXPATH considers the pages reached by clicking on those links as distinct. This avoids issues with server state where the same URL returns different results at different times or points in an interaction. It also avoids the need to maintain pages in OXPATH in case they are later encountered again. **(II) No back.** OXPATH does not allow the reverse (or "back") navigation over pages. Once we have moved from page A to B, there is no way back to A. This is a limitation as it (together with the lack of variables) prohibits a class of wrappers that refer back to values encountered on earlier pages. However, it is essential to maintain the low memory profile of OXPATH.

[1] However, classical results [41] on rewriting reverse axes such as `ancestor` in XPATH do not extend to OXPATH.

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

Fig. 6 OXPATH Syntax.

4.2 Syntax

The syntax of OXPATH is defined in Fig. 6 atop XPATH's syntax as defined in [26, 13]. We add OXPATH specific constructs to XPATH non-terminals and highlight the newly introduced non-terminals, i.e., the ***action*** and ***kleene***-star expressions added to step, and the ***marker*** expression added to step-suffix. Moreover, we allow certain CSS selectors [5] to occur in OXPATH expressions and maintain their normal semantics. These include '`#`' and '`.`', which filter by the id and the class attribute, respectively. For instance, `a#myid` matches a elements with *id* attribute equal to `myid` whereas `a.result` selects an a node if it contains `result` as a word in its *class* attribute. The containment operator '`~=`' returns **true** iff the right operand is contained in the left operand as a word (cf. word containment in CSS [5]). Furthermore, we add five abbreviations: For `contains(`a`,`b`)` we write a `~` b (string containment), for `count(`a `|` b`) = count(`b`)` we abbreviate with a **`subset`** b (subset), for `[`a `|` **`true`**`]` we write `[? `a`]`, for `(`a`)*{1,1}` we write `(`a`)`, and for **`style`**`::`*`color* we write `^`*`color*.

Even with the added selection capabilities of CSS, OXPATH's text extraction capabilities are still rather weak compared to full IE system. We are currently extending OXPATH with rich text processing operators (e.g., regular expression matching) inspired by XPATH 2.0. However, proper entity or relation extraction is beyond the scope of OXPATH and, if necessary, can be performed in a post-processing step.

Finally, we define the *main path* to a step s in an expression e as the sequence of steps occurring on the path to from the root of e's expression tree to the step s. For example, given $e =$`a[b[c]/d]/e`, we obtain for $s =$`d` and $s =$`e` respectively the main paths `a/b/d` and `a/e`.

The grammar in Figure 6 omits a few restrictions necessary to avoid an undesirable interplay of our new language features with functions, predicates and sorting operators as well as to avoid unintuitive extraction: **(R1)** Actions and extraction markers may not occur in other extraction markers, function or operator arguments. Further, extraction markers may not occur inside intensional axes (see Section 4.7). This limitation is mainly for simplicity, although actions in operator arguments can also affect the low memory principle of OXPATH (see Section 4.1). **(R2)** We disallow `position()` on the node set obtained from (Kleene-starred) bracketed expressions having actions or extraction markers in their main paths, to avoid sorting nodes which originate from different documents or relate to different extraction markers. This restriction is necessary to maintain the low memory goal of OXPATH, as it would require storing all these nodes from different pages if allowed. Expressions with extractions inside Kleene stars are rewritable into expressions satisfying this restriction, as shown in Theorem 2. **(R3)** Extraction markers may only occur in a predicate if there is a marker on the path leading to the predicate and the last such marker does not extract a value. Extraction markers extracting a value may not occur outside a predicate. The value of an extraction marker must yield a scalar value. These three restrictions ensure a sane use of extraction markers such that the expressions always produces a proper output tree ("tree result" principle). **(R4)** All predicates which are followed by a contextual action with no absolute action between must be free of actions. This ensures that the action-free prefix (see Section 4.4) of a contextual action does not contain any actions in predicates and thus can safely be executed multiple times on different pages without violating the "no page identity" principle.

4.3 Data Model

OXPATH's data model extends the data model of XPATH to multiple HTML pages, interconnected by actions. We evaluate OXPATH expressions on relational structures, called *page trees*, with an input schema extending the model introduced in Section 3:

$$\big((\mathsf{unary}_v)_{v\in\mathsf{Unary}}, \mathsf{child}, \mathsf{self}, \mathsf{next\text{-}sibl}, \mathsf{attribute}, \mathsf{style}, (\mathsf{action}_\alpha)_{\alpha\in\mathsf{Action}} \big)$$

Therein, $(\mathsf{unary}_v)_{v\in\mathsf{Unary}}$ is the set of unary relations, indicating type and label sets, child is the parent-child relation, self is the identity relation, next-sibl is the direct sibling relation, attribute relates nodes with their attribute nodes, and analogously, style relates nodes with their style nodes, that is, the dynamically computed style values of their parent nodes. For example, the node types `box-top`, `box-bottom`, `box-left`, `box-right`, `box-width`, and `box-height` refer to the dimensions of the bounding box of node at hand.

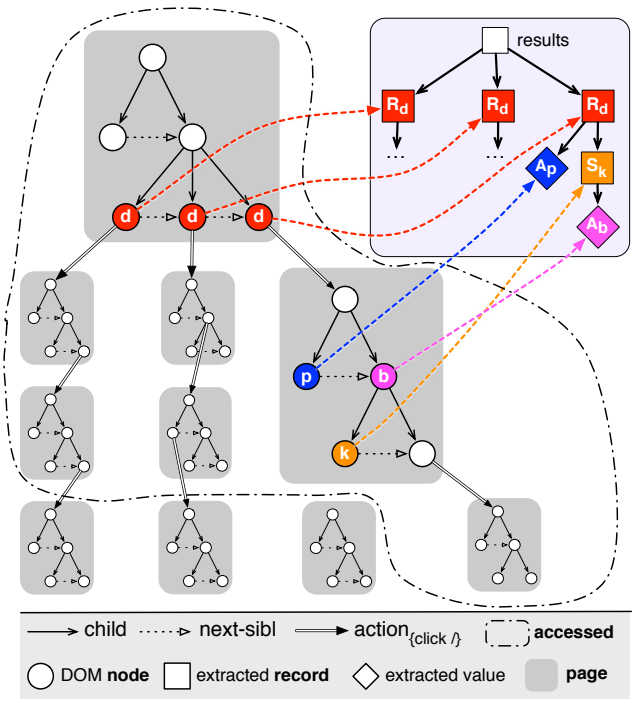


Fig. 7 OXPATH page tree.

The remaining XPATH axes, such as descendant, are derived from the basic relations, as in Section 3. We also add for each OXPATH action α a binary relation $(\mathsf{action}_\alpha)_{\alpha\in\mathsf{Action}}$. Herein, $\mathsf{action}_\alpha(x,y)$ indicates that action α triggered on node x yields the page rooted at y. The actions $(\mathsf{action}_\alpha)_{\alpha\in\mathsf{Action}}$ and the child axis together form a tree: Each page has a unique parent node connected by a single action edge, i.e., if two links point to the same URI, they yield two different pages in the page tree.

An OXPATH expression E returns an unordered tree of extracted nodes represented as a single relation O such that $\langle o,p,t,v\rangle \in O$ indicates that there is an output node o with parent output node p, tag t, and a value v, which is possibly null. A Record extraction marker R yields tuples of the form $\langle r,p,R,\mathsf{null}\rangle$, (typically) containing tuples of the form $\langle a,r,A,v\rangle$, generated from an attribute extraction marker A and a non-null value v. 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.

Fig. 7 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 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 tree accessed by a given OXPATH expression is called the *accessed page tree* (dot-

ted area in Fig. 7), consisting of *accessed pages and nodes.* Fig. 7 also shows the result of an OXPATH expression

```
//d:<R>/{click/}/.[//p:<Tp=string(.)>]
        [//k:<S>/parent::b:<Tb=string(.)>]
```

producing (square) output nodes $\langle R_d, \mathsf{results}, R, \mathsf{null}\rangle$ for matching record extraction markers `//r:<R>`, and $\langle S_k, R_d, S, \mathsf{null}\rangle$ for `//s:<S>` (nested records). It produces (diamond) nodes $\langle A_p, R_d, T_p, v_p\rangle$ and $\langle A_b, S_k, T_b, v_b\rangle$ for value extraction markers `//p:<`T_p`=string(.)>` and `parent::b:<`T_b`=string(.)`. Note that, the structure of the output tree reflects the structure of the extraction markers in the OXPATH expression, but not necessarily the structure of the input tree. In our example, the fragment `//k:<S>/parent::b:<`T_b`=string(.)>` produces output nodes A_b children of output nodes S_k, although the `b`'s are parents of the `k`'s in the input tree.

4.4 Semantics

The semantics of OXPATH is defined with its *extraction semantics* $[\![expr]\!]^{\beta}_{\mathsf{E}}(c)$, specifying the result tree for expression *expr*, context tuple c, and variable assignment β. Each *context tuple* $c = \langle n, p, l\rangle$ consists of an input node n, the parent output node p, and the last sibling output node l, accessed through the notation $c.n$, $c.p$, $c.l$. We define the extraction semantics atop the *value semantics* $[\![expr]\!]^{\beta}_{\mathsf{V}}(c)$ which matches nodes from the page tree, akin to XPATH's semantics in [13]. The latter computes the value reached via *expr*, which is either a set of context tuples, a Boolean, integer, or string value. For lucidity's sake, we first develop the semantics for the OXPATH language without the intensional axes, which we add to the semantics in Section 4.7.

The OXPATH semantics deviates from XPATH in the contents of its context tuples. We maintain both, the preceding parent and sibling match, to organize the extracted nodes hierarchically: At context $c = \langle n, p, l\rangle$, an extraction match outside predicates **(1)** yields an output tuple $\langle o, c.p, M, v\rangle$, with a fresh output node o, becoming descendant of $c.p$ and sibling of $c.l$, and **(2)** changes the context to $c' = \langle c.n, c.p, o\rangle$. On entering a predicate, $c.l$ replaces $c.p$ as parent output node, such that further extraction matches yield nodes that are nested as descendants of $c.l$ instead of $c.p$.

We *start the evaluation* of an OXPATH expression with the initial context node $c = \langle \bot, \mathsf{results}, \mathsf{results}\rangle$, where $\bot$ is an arbitrary context node and results is the root of all subsequent extraction matches. The context node is arbitrary since all expressions start with `doc(`*uri*`)` which returns the root of the page at *uri* regardless of the supplied context node.

For an OXPATH expression that is also an XPATH expression, the value semantics computes the exact same result as standard XPATH. Table 1 and 2 give the full OXPATH value and extraction semantics, respectively, though we omit those rules in the extraction semantics that only

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	$[\![e_1\ op\ e_2]\!]_{\mathsf{V}}(c)$	$= B_{op}([\![e_1]\!]_{\mathsf{V}}(c),[\![e_2]\!]_{\mathsf{V}}(c))$
V4	$[\![op\ e]\!]_{\mathsf{V}}(c)$	$= U_{op}([\![e]\!]_{\mathsf{V}}(c))$
V5	$[\![value]\!]_{\mathsf{V}}(c)$	$= value$
V6	$[\![\$var]\!]^{\beta}_{\mathsf{V}}(c)$	$= \beta[var]$
N1	$[\![(i)step/path]\!]_{\mathsf{N}}(c)$	$= \{c'' \mid c' \in [\![(i)step]\!]_{\mathsf{N}}(c) \wedge c'' \in [\![path]\!]_{\mathsf{N}}(c')\}$
N2	$[\![axis]\!]_{\mathsf{N}}(c)$	$= \{\langle n', c.p, c.l\rangle \mid axis(c.n, n')\}$
N3	$[\![axis::nt]\!]_{\mathsf{N}}(c)$	$= \{c' \in [\![axis]\!]_{\mathsf{N}}(c) \mid c'.n \in \mathsf{unary}_{nt}\}$
N4	$[\![(i)step[q]]\!]_{\mathsf{N}}(c)$	$= \{c' \in [\![(i)step]\!]_{\mathsf{N}}(c) \mid [\![q]\!]_{\mathsf{B}}(\langle c'.n, c'.l, c'.l\rangle)\}$
N5	$[\![(i)step_{\pm}[qp]]\!]_{\mathsf{N}}(c)$	$= \{c' \in C \mid C = [\![(i)step_{\pm}]\!]_{\mathsf{N}}(c) \wedge$ $[\![\mathrm{REWRITE}_{\pm}(qp, C, c')]\!]_{\mathsf{B}}(\langle c'.n, c'.l, c'.l\rangle)\}$
N6	$[\![\{action\,/\}]\!]_{\mathsf{N}}(c)$	$= \{\langle n', c.p, c.l\rangle\}$ with $\mathsf{action}_{action}(c.n, n')$
N7	$[\![\{action\}]\!]_{\mathsf{N}}(c)$	$= [\![\mathrm{AFP}(action, c.n)]\!]_{\mathsf{N}}([\![\{action\,/\}]\!]_{\mathsf{N}}(c))$
N8	$[\![:\langle M[=v]\rangle]\!]_{\mathsf{N}}(c)$	$= \{\langle c.n, c.p, \mathrm{OUT}(c.n, M)\rangle\}$
N9	$[\![(path)*]\!]_{\mathsf{N}}(c)$	$= \{c_r \mid \exists r \geqslant 0\ \forall 0 \leqslant s < r :$ $c_{s+1} \in [\![path]\!]_{\mathsf{N}}(c_s) \wedge c_0 = c\}$
N10	$[\![(path)*\{v,w\}]\!]_{\mathsf{N}}(c)$	$= \{c_r \mid \exists v \leqslant r \leqslant w\ \forall 0 \leqslant s < r :$ $c_{s+1} \in [\![path]\!]_{\mathsf{N}}(c_s) \wedge c_0 = c\}$
N11	$[\![]\!]_{\mathsf{N}}(c)$	$= \{c\}$
B1	$[\![expr]\!]_{\mathsf{B}}(c)$	$= U_{Boolean}([\![expr]\!]_{\mathsf{V}}(c))$

Table 1 Value Semantics of OXPATH

decompose the expression. F_f, B_o and U_o are the semantic functions on the nodes corresponding to XPATH's functions, binary, and unary operators, extended by the OXPATH `.` and `#` node-tests and `~` and `~=` operators. As the variable assignment β is handed thorough unchanged throughout the semantics, we show β only in Rule **V6**, where we evaluate variables , and drop it otherwise. For simplicity, we disallow positional functions outside qualifiers.

4.5 Value Semantics

In Table 1, Rule **V1**, $[\![path]\!]_{\mathsf{V}}$ delegates the evaluation of a *path* to $[\![path]\!]_{\mathsf{N}}$ which handles expressions computing *node sets.* The Rules **V2–V4** deal with functions and operators and apply the corresponding semantic functions on the evaluated subexpressions and operands. Rule **V5** handles literals and Rule **V6** maps a variable $\$var$ to its value $\beta(\$var)$.

The major part of the value semantics, consisting of Rules **N1-N10**, deals with *path* expressions. At first, Rule **N1** decomposes a given *path* into its first element, which is either a *step* or an *istep*, and the tail expression *path*. Then the semantics evaluates the $(i)step$ with Rules **N2–N10**, and evaluates *path* on the resulting node set recursively.

N2–N5: *Axes, node-tests, and predicates.* In Rules **N2** and **N3**, we handle OXPATH axes and node-tests as in standard XPATH. The case of predicates in Rules **N4** and **N5**

follows XPATH as well, but requires additional provisions: To manage the nesting of the extracted data, upon entering a predicate, OXPATH takes the last sibling output node as new parent output node, both in Rule **N4** and **N5**. Expressions in predicates are cast to Booleans by means of $[\![expr]\!]_{\mathsf{B}}$ with Rule **B1**, as in XPATH. Rule **N5**, for positional predicates, relies on two new functions, REWRITE_+ and REWRITE_-, of the form $expr \times C \times c \to expr'$, where $expr$ is an input expression, C is a context set, $c \in C$, and $expr'$ is a rewritten expression as follows: Both functions replace in the input expression each non-nested occurrence of `last()` with $|C|$ and of `position()` with the position of c within C according to document order. This order is well-defined within all possible context sets at this point, since all such context sets contain only tuples with nodes from the same page with identical parent and sibling matches: If $(i)step$ starts with an axis/node navigation or an action, this property holds. Otherwise $(i)step$ must a be bracketed (Kleene star) expression, and in this case, the expression is not allowed to contain actions or extraction markers (Restriction **(R2)** on page 7), preventing any changes in the underlying page or context tuples. We write $\text{REWRITE}_\pm$ for conciseness, where the specific function applied depends on the $(i)step$: For axis navigation, we choose REWRITE_+ for forward and REWRITE_- for reverse axes. Otherwise, for (Kleene-stared) bracketed or action expressions, we always select REWRITE_+.[2]

N6–N7: *Actions.* Actions map the context node $c.n$ to a node in a different page. *Absolute actions* in Rule **N6** map $c.n$ to the root n' of some other page with $(\mathsf{action}_\alpha)_{\alpha \in \mathsf{Action}}$. By our data model, we assume that the node n' is different from any other node previously reached. For *contextual actions*, Rule **N7** first also evaluates the absolute action to obtain the root of the new page, but then attempts to move to a node that corresponds closely to the node $c.n$ on the original page. We select the corresponding node by applying the same OXPATH expression to the new page root as we used to select $c.n$ in the original page. The *action-free prefix* $\text{AFP}(action, c.n)$ of *action* and $c.n$ in OXPATH expression *expr* returns the following OXPATH expression: Let *base* be the subexpression of *expr* between *action* and its last preceding absolute action, stripped of all extraction markers and all contextual actions occurring in the main path of *expr*. Then $\text{AFP}(action, c.n) = (base)[i]$ where i is the position of $c.n$ in document order among all nodes in the current page matching *base*. In the original page, this expression uniquely identifies $c.n$. When evaluated from the root returned by the absolute action on $c.n$, it selects a unique node reached by the same path and in the same relative position.

N8: *Extraction markers.* For extraction markers, the context set of the corresponding step is modified by replacing the last sibling output node $c.l$ with the one generated from this marker M and current node $c.n$. OXPATH computes the new node with $\text{OUT}(c.n, M)$ where OUT injectively maps an input node $c.n$ and extraction marker M to an output node.

N9–N10: *Kleene star.* Unbounded and bounded Kleene stared expressions match the Kleene-repeated path multiple times, enforcing optional iteration bounds.

N11: *Empty expressions.* Return the context node.

[2] Thus, $(path)*[\mathtt{qp}] = (\bigcup_{i=0}^{\infty} path^i)[\mathtt{qp}]$ always holds, but $(path)*[\mathtt{qp}] = \bigcup_{i=0}^{\infty} path^i[\mathtt{qp}]$ does *not* hold necessarily, since `[qp]` is applied to each of the i-th copy of *path*.

E1	$[\![(i)step/path]\!]_{\mathsf{E}}(c)$	$= [\![(i)step]\!]_{\mathsf{E}}(c) \cup \bigcup_{c' \in [\![(i)step]\!]_{\mathsf{V}}} [\![path]\!]_{\mathsf{E}}(c')$
E2	$[\![(i)step[q]]\!]_{\mathsf{E}}(c)$	$= [\![(i)step]\!]_{\mathsf{E}}(c) \cup \bigcup_{c' \in [\![(i)step]\!]_{\mathsf{V}}(c)} [\![q]\!]_{\mathsf{E}}(\langle c'.n, c'.l, c'.l\rangle)$
E3	$[\![:\langle M\rangle]\!]_{\mathsf{E}}(c)$	$= \{\langle \text{OUT}(c.n, M), c.p, M, \mathsf{null}\rangle\}$
E4	$[\![:\langle M = v\rangle]\!]_{\mathsf{E}}(c)$	$= \{\langle \text{OUT}(c.n, M), c.p, M, [\![v]\!]_{\mathsf{V}}(c)\rangle\}$

Table 2 Extraction Semantics of OXPATH (partial)

4.6 Extraction Semantics

The extraction semantics $[\![expr]\!]_{\mathsf{E}}(c)$ for OXPATH in Table 2 takes context tuple c and extracts an output tree from the input page tree. For sake of brevity, we omit those rules that recursively decompose the expression but have no other effect. Except for extraction markers, the extraction semantics is straightforward: For expressions with subexpressions, we compute the extraction semantics for each subexpression and take the union of all extracted results. To compute the extraction semantics for a subexpression *expr*, we first compute with the value semantics $[\![\,]\!]_{\mathsf{V}}$ the context set to apply *expr* on, and then apply $[\![expr]\!]_{\mathsf{E}}$ recursively on each obtained context node. Rule **E1** and **E2** exemplify this case for paths and predicates. For all other expressions not shown here, we collect all extraction markers returned by their subexpressions (if any), regardless of the (value) semantics of the involved expressions.

Extraction markers are treated in Rule **E3** and **E4**: For *markers without extracted values* (Rule **E3**), OXPATH extracts the tuple $\langle \text{OUT}(c.n, M), c.p, M, \mathsf{null}\rangle$. The resulting node is thus a child of the parent match $c.p$. For *markers with values* (Rule **E4**), we evaluate additionally the value expression v: We take the value returned by $[\![v]\!]_{\mathsf{V}}(c)$ (a string or other scalar) and output $\langle \text{OUT}(c.n, M), c.p, M, [\![v]\!]_{\mathsf{V}}(c)\rangle$.

4.7 Intensional Axis

In XPATH, axes relate nodes through a fixed set of relations such as child or following. Together with functions and operators these are the only means in XPATH for relating two nodes. Unfortunately, these means are rather limited, e.g., we cannot identify all nodes that follow the current node in

document order *and* are displayed in the same font size as the current node. In general, XPATH is not able to express queries where nodes from two node sets are related by more than one relation.

Theorem 1 *Let $\mathscr{I}$ be the set of first order queries of the form $Q(x,y) \Leftarrow \phi(x,y) \wedge \psi(x,y)$ where $\phi(x,y)$ and $\psi(x,y)$ are non-empty first order formulas expressible in* XPATH. *Then there are queries in $\mathscr{I}$ that cannot be expressed in* XPATH.

Proof (sketch) From [36] and [13], this follows for navigational XPATH, which expresses exactly all **XPNF queries**. An XPNF query is a FO^2 query over page trees (without action relations) built from relations between two node sets which are limited to disjunctions of binary atomic formulas.

For full XPATH, we need to show that neither relational operators, functions, aggregation, or positional arithmetic allow us to express multiple relations between two node-sets. All queries in $\mathscr{I}$ relate two nodes, and thus we can ignore boolean and other value queries.

First, *outside predicates,* `id()` is the only functional operator allowed in such queries. As `id()` returns the same result for any context node, it does not relate multiple nodes.

Second, *inside predicates*, XPATH allow conjunctions of functions, relational operators, and aggregations. However, the only "shared variable" between such conjuncts is the context node (see V1–V6 in Table 1). Though it is possible to build up several node sets originating from the same context node, once constructed, its individual elements are only accessible via quantification. For example, `[.//a=.//b and .//a=.//c]` does *not* require the existence of a single *a* node having the same value as some *b* and some *c*: In contrast, the predicate is already satisfied if there exist two *a* nodes which match respectively the values of some *b* and *c*.

Though the same context set can be matched by multiple conjuncts (by using two equivalent sub-expressions), the individual nodes in these node-sets cannot be related. This even holds for `count()` which can be used to relate entire node-sets, but not individual nodes. □

To address this limitation, we introduce **intensional axes** to support OXPATH users in intensionally defining relations between node pairs, as needed. We denote intensional axes like predicates, but place them in lieu of an axis, i.e., before a `::`. For example, if we evaluate on a context c,

```
[$rhs subset $lhs/following::* and
   $lhs/style::font-size=$rhs/style::font-size]::*
```

we obtain the context set C containing all nodes c' such that the expression inside the brackets evaluates to **true** for the variable assignment $\beta' = \beta[\$lhs \leftarrow c.n, \$rhs \leftarrow c'.n]$. We bind `$lhs` to the current node $c.n$, try for `$rhs` every node in the current page, and return those for which the axis expression is satisfied. Thus, this expression solves the query from the beginning of this section: It returns all nodes that follow the current node in document order *and* are displayed in the same font size. The expression uses OXPATH's `subset` operator to test if `$rhs` is among the nodes following `$lhs` (see Section 4.2).

⟨*axis*⟩ ::= ⟨*xpath-axis*⟩ | '`style`' | ⟨*intensional-axis*⟩
⟨*intensional-axis*⟩ ::= '`[`' ⟨*expr*⟩ '`]`'

Fig. 8 OXPATH intensional axis.

N2	$[\![axis::nodes]\!]_N(c)$	$= \{\langle n', c.p, c.l\rangle \mid n' \in [\![axis]\!]_N(c) \wedge \mathsf{nodes}(n')\}$
N12	$[\![[expr]]\!]_N(c)$	$= \{\langle n', c.p, c.l\rangle \mid n' \in \mathsf{nodes} \wedge [\![expr]\!]_B^{\beta'}(c) \wedge \beta' = \beta[\$lhs \leftarrow c.n, \$rhs \leftarrow n']\}$

Table 3 Value semantics for intensional axes

Definition 1 An *intensional step* `[`ϕ`]::`$n\psi$ consists of an *intensional axis* of `[`ϕ`]`, a node-test n, an arbitrary number of predicates ψ, and an arbitrary OXPATH expression ϕ that may use the reserved variables `$lhs` and `$rhs`. An intensional axis `[`ϕ`]` returns, for a context node c, all nodes m such that $[\![\phi]\!]_B$ is **true** if `$lhs` is bound to c and `$rhs` to m.

Note, that an intensional axis that does not refer to `$lhs` or the context node relates all context nodes to the same bindings for `$rhs` (since $[\![\phi]\!]_B$ does not depend on `$lhs`. If it does not refer to `$rhs` it acts as a filter to the context nodes, but each context node that matches the filter is related to all nodes in the DOM. If neither is referenced, it is either $\varnothing$ or the set of all pairs of DOM nodes, depending on whether $[\![\phi]\!]_B$ holds (which is absolute and independent of the context node).

Table 3 show the necessary additions to the semantics of OXPATH. The existing Rule **N2** already suffices to cover intensional steps. Rule **N12** (and only this rule) modifies the set of variable bindings β such that `$lhs` is now bound to the context node and `$rhs` is successively bound to all nodes n in the document. With this updated variable binding, *expr* is evaluated under Boolean semantics, and if $[\![expr]\!]_B$ evaluates to **true**, n is added to the resulting context set. Consequently, in case of nested intensional axes, `$lhs` and `$rhs` are always bound according to the last axis.

Examples. Table 4 lists five applications of intensional axes in OXPATH. Expression **(1)** selects all nodes that have the same font color and size as an `a`. In **(2)** we select all `div`s that are rendered north-west of an `a`. Expression **(3)** selects all `book`s having a common author with another `book` that is cited by the latter one, i.e., it selects `book`s containing self citations. Example **(4)** shows a nested intensional axis: It selects all `em` children of elements that have the same font family as a `div` and that are to the north of an `a` of that `div` with the same value. This case requires a nested relational

(1) *Same font color and size:*
```
//a/[$lhs/^color = $rhs/^color and
      $lhs/^font-size = $rhs/^font-size]::*
```

(2) *To the north-west:*
```
//a/[$rhs/^box-right < $lhs/^box-left and
      $rhs/^box-bottom < $lhs/^box-top]::div
```

(3) *Same author and cites:*
```
//book/[$lhs/author = $rhs/author and
      $lhs/@id = $rhs/cites/@idref]::book
```

(4) *Same font family and to the north and same value as a descendant:*
```
//div/[$lhs/^font-family = $rhs/^font-family and
      $rhs subset $lhs//a/[$lhs = $rhs and
            $lhs/^box-bottom <= $rhs/^box-top]::*]::*/em
```

(5) *Visually contained:*
```
//div/[$lhs/^box-left <= $rhs/^box-left and
      $lhs/^box-right >= $rhs/^box-right and
      $lhs/^box-top <= $rhs/^box-top and
      $lhs/^box-bottom >= $rhs/^box-bottom]::*
```

Table 4 Intensional axis examples

axis, as the `div` is related to the `em` by more than on relation, but so is the `a` to the `em`'s parent. Expression **(5)** selects all elements that are visually contained in some `div`.

4.8 OXPATH Properties

As discussed in Section 4.1, OXPATH is designed to avoid buffering many pages or result tuples at the same time. This design goal is expressed in two formal properties:

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 1 (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 **N5**, where the function $\text{REWRITE}_{\pm}(q,C,c')$ 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 **N5** never contains nodes from different pages.

This holds true, since in Rule **N5**, $C = [\![(i)step_{\pm}]\!]_{\mathsf{V}}(c)$ is computed from a single axis navigation *axis* :: *nodes* (**N3**), followed by a sequence of (positional) qualifiers (**N4** and **N5**) and markers (**N8**). Our language restriction **(4)** from Section 4 ensures that actions cannot occur in $(i)step_{\pm}$, as they are disallowed in positionally qualified steps. Since Rule **N3** always results in a context set with nodes from the same page, and since **N4-5,8** can only remove nodes from the context set, C must contain nodes from a single page only. □

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

Proposition 2 (No Output Buffer) *The evaluation of* OXPATH *expressions requires no output buffer.*

Proof Only Rules **E3** and **E4** in Table 2 create output tuples. We visit each input node $c.n$ at most once with extraction marker M, and thus, each created output tuple is unique (and is not overridden or altered in any anymore). The output tuples are immediately added to the output relation O, regardless of whether the current expression evaluates eventually to true or not. Also, when the output tuples are created, the parent output nodes are known by construction and thus no buffering is needed to obtain all tuple values. All other rules in the extraction semantics (as in **E1** and **E2**) only collect the tuples returned by their sub-expressions. Since no duplicate tuples are created, this requires no buffering. □

More intuitively, this holds as the structure of the output tree reflects the structure of the OXPATH expression and parent nodes are therefore always created before their children nodes. Also, tuples are extracted immediately upon creation, regardless of whether the current subexpression matches or not. For example, consider $expr[p_1][p_2]$. If p_1 contains extraction markers, 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: We can rewrite the above expression into $expr[p_2'][p_1][p_2]$ where p_2' is obtained from p_2 by removing all extraction markers. Then 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.

4.9 Normalizing OXPATH

We can reduce the size of the necessary memoization tables by a factor of n, without restricting the language, at potential expense of longer queries by rewriting general OXPATH into *normalized* OXPATH, a fragment denoted $\text{OXPATH}_{\text{norm}}$.

We introduce two normalization properties for OXPATH expressions: Property **(A)** does not allow any extraction markers within Kleene-star expressions, and Property **(B)** disallows any two extraction markers on the same expression branch. The latter property means that all extraction markers after any given marker must be nested within predicates. For example, Property **(B)** is violated by a`:<R>`b`:<S>` but satisfied by a`:<R>[`b`:<S>]`. In this section, we do not distinguish record markers, such as `:<R>`, and attribute extraction markers of the form `:<R=...>`.

To normalize general OXPATH expressions, we apply two rewriting steps. The first rewriting, shown in Theorem 2,

produces expressions which meet Property **(A)**. When we cannot apply Theorem 2 anymore, we rewrite the obtained expression following Theorem 3, again until inapplicable, to meet Property **(B)** as well.

The proof of Theorems 2 and 3 relies on the loose coupling of value and extraction semantics in OXPATH: The extraction semantics does not influence the value semantics at all, as stated in Fact 3. On the other hand, the extraction semantics depends on the value semantics, as the value semantics determines which nodes to extract. But extractions take place immediately, independently of whether the tailing expression matches any values or not, as stated in Fact 4.

Fact 3 (Extraction agnostic value semantics) *The introduction or removal of extraction markers does not affect the value semantics of an* OXPATH *expression.*

Fact 4 (Tail agnostic extraction) *When extraction marker* `<R>` *in* `a:<R>b` *is evaluated after having matched a, the corresponding pairs* $\langle n, R\rangle$ *are extracted immediately, regardless of whether b matches subsequently.*

4.9.1 Extraction-Free Kleene Star

For the next theorem, we rely on OXPATH not being short-circuited, e.g., the evaluation of `[`a `|` b`]` cannot be aborted once a is evaluated to **`true`** since b may contain extraction markers which must be matched, even if they do not affect the value semantics.

Theorem 2 (Extraction-Free Kleene Star) *Let e be an* OXPATH *expression* $hp{*}t$*, with h, p, and t as arbitrary subexpressions. Then the expression e is rewritable with*

$$hp{*}t \equiv ht \texttt{ | } hq{*}pt \equiv h \texttt{ [? } q{*}p\texttt{] } q{*}t \texttt{ ,}$$

where q denotes the expression derived from p by removing all extraction markers.

Proof We show the theorem by proving

$$p{*} \equiv \textbf{self} \texttt{ | } q{*}p \equiv \textbf{self} \texttt{ [? } q{*}p\texttt{] } q{*} \texttt{ ,}$$

where we abbreviate **`self`**`:node()` with **`self`**. From this claim, the theorem follows, by adding h and t to the start and end of the three subexpressions in claim.

First, we show the claim for the *value semantics* and subsequently for the extraction semantics. Using $\equiv_V$ to denote the equivalence with respect to value semantics, we obtain $p{*} \equiv_V q{*} \equiv_V$ **`self`** `[? `x`]` $q{*} \equiv_V$ **`self`** `[? `$q{*}p$`]` $q{*}$. Therein, Step **(1)** holds for Fact 3, **(2)** holds for every expression x, since optional predicates are not required to evaluate to true (in fact, they have been introduced for conditional extraction), and **(3)** instantiates x with $q{*}p$. Similarly, we have $p{*} \equiv_V$ **`self`** `|` $p{*}p \equiv_V$ **`self`** `|` $q{*}p$, where Step **(1)** unrolls the Kleene star expression, and **(2)** holds again for Fact 3. Taken together, these identities show the claim for value semantics, i.e., for $\equiv_V$.

Second, for *extraction semantics,* we show

$$\begin{aligned} p{*} &\equiv_E \textbf{self} \texttt{ | } p{*}p \equiv_E \textbf{self} \texttt{ | } q{*}p \\ &\equiv_E \textbf{self} \texttt{ [? } q{*}p \texttt{] } y \equiv_E \textbf{self} \texttt{ [? } q{*}p \texttt{] } q{*} \texttt{ ,} \end{aligned}$$

yielding the sought for equivalences after Steps **(2)** and **(4)**. Step **(1)** unrolls the first iteration of the Kleene star. In **(2)** we replace an instance of p with q, and hence we need to show that every pair extracted by an instance of p in $p{*}p$ is also extracted by $q{*}p$. But this is the case: If $p{*}p$ extracts some pair, then there must exist a minimal $i \geqslant 0$ such that $p^i p$ extracts this pair. Because of Fact 4, we only consider the prefix leading to the extraction, while we ignore the subsequent expressions to be matched. Since i is minimal, the extraction does not occur within p^i but in the tailing p, and therefore, $q^i p$ produces the same pair. Hence, $q{*}p$ does so as well, proving the soundness of this step. Step **(3)** holds for any y without extraction markers: All pairs extracted by $q{*}p$ are also extracted by the conditional predicate `self [? `$q{*}p$`]`, regardless of the tailing y. On the other hand, since y does not contain extraction markers, `self [? `$q{*}p$`]` y cannot extract more than $q{*}p$. Step **(4)** instantiates y with $q{*}$, which is valid since q contains no extraction markers. □

Both rewriting options in Theorem 2 have exponential upper bounds: If we rewrite $hp{*}t$ with ht `|` $hq{*}pt$, we need to duplicate the entire expression, with additional occurrences of h, t, and p (in terms of q). On the other hand, if we use h `[? `$q{*}p$`]` $q{*}t$, we triplicate p with two additional copies of q. In our experience, if extraction markers occur in Kleene star expressions, then the Kleene-stared expressions are rather short, i.e., the second option is usually the better choice. Finally, if the Kleene star must match at least once (e.g., when using the Kleene `+` instead of $*$), then we can use an even more efficient rewriting:

Corollary 1 (Rewriting for Kleene +) *Let e be an* OXPATH *expression* $hp{+}t$*, with h, p, and t arbitrary. Then the expression e is rewritable with* $hp{+}t \equiv hq{*}pt$ *where q denotes the expression derived from p by removing all extraction markers.*

In general all these rewritings are exponential in the expression size. However, the overhead introduced by rewriting Kleene star expressions with bounded Kleene nesting depth is polynomial. This bound is practically relevant, as we never encountered natural OXPATH expressions with a nesting depth larger than 2.

Moreover, the rewriting of Theorem 2 is naturally extended to the bounded case with (for $a \dot{-} b = max(a-b, 0)$)

$$\begin{aligned} hp{*}\textbf{\{n,m\}}t \equiv\ & ht \texttt{ | } hq{*}\textbf{\{n}\dot{-}\textbf{1,m}\dot{-}\textbf{1\}}pt \\ & \equiv h \texttt{ [? } q{*}\textbf{\{n}\dot{-}\textbf{1,m}\dot{-}\textbf{1\}}p\texttt{] } q{*}\textbf{\{n,m\}}t \texttt{ .} \end{aligned}$$

4.9.2 Sibling-Free Extraction

Theorem 3 (Sibling-Free Extraction Markers) *Let e be an* OXPATH *expression* `a:<R>b:<S>c`*, with a, b, and c as arbitrary subexpressions. Then e is rewritable with*

a:<R>b:<S>$c \equiv a$ [**self**:<R>] b:<S>c

Proof We prove Theorem 3 first for *value semantics,* then for extraction semantics. With $\equiv_V$ for equivalence according to value semantics, we obtain

a:<R>b:<S>$c \equiv_V ab$:<S>$c \equiv_V a$ [**self**:<R>] b:<S>c

where Step **(1)** holds because of Fact 3, and Step **(2)** holds since [**self**:<R>] evaluates to true under value semantics for every node.

For *extraction semantics,* a:<R> and a [**self**:<R>] must extract the same pairs, since <R> is applied to the same node sets, as a and a/**self** select same nodes. Again, because of Fact 4, the respective tail expressions are irrelevant. □

4.10 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 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, as most pages execute their actions quickly.

Theorem 4 (Complexity) OXPATH *evaluation without multiplication and string concatenation is in* NLOGSPACE *for* data complexity. OXPATH *evaluation is* PTIME*-complete for* combined complexity.

Proof We show the theorem statements separately, starting with *data complexity:* From all extensions over 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 [13]. 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 [13]. To evaluate an OXPATH query, we process the query left to right, and decompose it recursively. Since we show that evaluating each subexpression requires at most polynomial time, the overall evaluation runs in polynomial time as well.

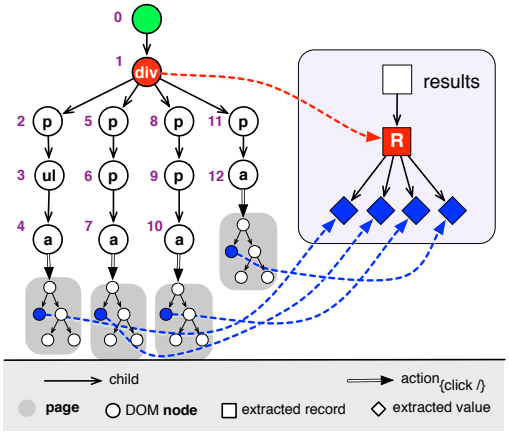


Fig. 9 Page and Result Tree Example.

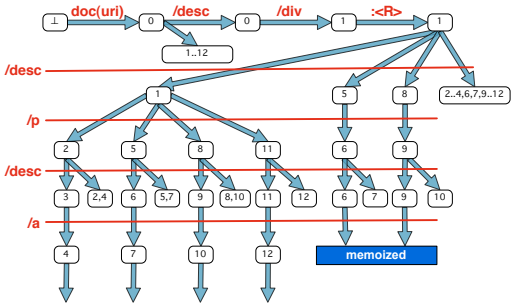


Fig. 10 Call Graph Example.

For XPATH subexpressions, we rely on one of the known polynomial time algorithms for XPATH [26], which can be easily extended with **style**, intensional axes, and the other selection only features added in OXPATH. If the expression is an extraction marker, we stream out the extracted tuple, which is in polynomial time, too, while 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 non-nested actions, we evaluate it like an ordinary Regular XPATH, leading to a polynomial time algorithm [35]: If a predicate within the expression contains an action, we can evaluate this predicate in polynomial time, and since we need to evaluate this predicate at most for a polynomial number of context tuples, the theorem statement follows. □

5 Page-At-A-Time Evaluation

As shown in the previous section, OXPATH remains close to the favorable complexity results of XPATH. In this section, we show OXPATH's practical performance to be close

to XPath's performance, only slightly increasing the upper bounds. For example, consider evaluating the expression

```
doc(u)//div:<R>//p//a[{click /}//title:<t=string(.)>]
```

on the (simplified) DOM fragment in Fig. 9. The expression navigates to all descendant links of `p` nodes, which are themselves descendants of `div` nodes. It extracts an `<R>` record for the `div` node and then clicks on all found links to extract the `title` of each reached page as `<t>` attribute. The evaluation first expands the abbreviated syntax to obtain

```
Expr =doc(u)/descendant-or-self::node()/child::div:<R>
        /descendant-or-self::node()/child::p
        /descendant-or-self::node()/child::a
           [{click /}/descendant-or-self::node()
            /child::title:<t=string(.)>]
```

and then proceeds by processing the individual steps of the expanded expression. Fig. 10 shows the call graph during the evaluation of Expr, starting with the initial context tuple $\langle \bot, \mathsf{results}, \mathsf{results} \rangle$. For simplicity, the nodes in the call graph do not show the full context tuples $\langle n, p, l \rangle$ but only the respective DOM nodes n. In the first step, `doc(uri)` is applied to $\bot$ to yield 0, where the latter is the DOM root of `uri`. The next step reaches via `descendant-or-self:node()` all nodes 0...12. Since the nodes 1...12 have no `child::div` successor, as required in the third step, we summarize those nodes into a single node, and continue from node 0, which is the only node with a `div` child. In the fourth step, the *extraction marker* `:<R>` creates an `<R>` record for node 1, linked to the results record which contains all extraction results. Subsequently, when we process the remaining steps in this fashion, we need to make sure, that we do not perform redundant computations. For example, there are two ways to reach nodes 7 and 9, respectively, such that all computations below those two nodes would be replicated by a naive algorithm. In such cases, we avoid recomputing the same results by *memoizing* the afore-computed results. We complete the processing by clicking on all found links (nodes 4, 7, 10, 12) and by extracting the title of the reached pages (not shown in the figure). In general, we have to assume that loading a page a second timeyields two different pages. Hence, to minimize the needed resources, we buffer the current page and load each of the four linked pages one after another into a second page buffer.

5.1 Algorithmic Design Goals

Starting with a standard XPath evaluation with memoization [26], only two of OXPath extensions demand significant additional treatment, leading to the following three design goals: **(1)** *Actions* visit different pages, and multiple actions on the same page yield branches in the page tree (see Section 4.3). Unfortunately, if the same page is fetched multiple times, we may obtain different results, e.g., if the underlying data has changed or the page contains time sensitive information. Thus, we need to buffer such pages, but at the same time, need to *minimize the number of necessary page buffers* without reloading pages. **(2)** With *extraction markers*, we can return multiple, possibly related data items, requiring the evaluation to collect these items. To scale well with large scale extraction tasks, we *need to efficiently propagate matches* on extraction markers and their relations. Aside these two goals, we also need to **(3)** *maintain the polynomial evaluation* of XPath, catering the other extensions of OXPath efficiently.

To address **(1)**, our *page-at-a-time* algorithm traverses the page tree in a depth-first manner without retaining information on pages not visited again. However, a naive depth-first traversal of the DOM nodes within individual pages would cause an exponential worst-case runtime in violation of **(3)**, necessitating memoization of intermediate results. To address **(2)**, our algorithms stream out extraction matches, requiring no buffering at all.

Mutual Recursive Evaluation. As a solution to these design goals, we employ two mutually recursive evaluation procedures `eval_` and `eval`. Thereby, `eval_` evaluates *simple* OXPath *expressions* without actions or extraction markers. As such, they are regular XPath expressions (as in [35]), extended with the style axis and additional operators discussed at the beginning of Section 4. In our example `//div`, `//p//a`, and `//title` are simple subexpressions to be evaluated with `eval_`. Given `eval_`, `eval` decomposes *full* OXPath *expressions* into chunks of simple expressions for delegation to `eval_`, and directly handles those steps which contain actions and markers. Each chunk spans from one extraction marker, action, or predicate, containing markers or actions to the next such step. Both algorithms implement a recursive top-down evaluation, in the spirit of the semantics in Section 4.4, as shown in Algorithms 2 and 3. To compute the value semantics $[\![\,]\!]_{\mathsf{V}}$ and extraction semantics $[\![\,]\!]_{\mathsf{E}}$ simultaneously, `eval_` and `eval` return the result according to $[\![\,]\!]_{\mathsf{V}}$, while outputting the tuples following $[\![\,]\!]_{\mathsf{E}}$. For clarity, we factor the evaluation of individual tuples from `eval_` into `evalT_` in Algorithm 1.

Memoization. As essential design goal, we need to prevent multiple evaluations of the same expression with the same context tuple. For example, while evaluating `//p//a[...]` in our example, there are two `a` nodes (7 and 10) that are descendants of multiple `p` nodes. While a naive implementation processes such `a` nodes multiple times, we avoid this overhead by inserting memoization at two strategic positions: We **(1)** encapsulate the evaluation of simple expressions into `eval_` extending the memoization-based XPath evaluation from [26], and **(2)** additionally memoize the outcome of non-simple predicates of `eval`). Only recursion branches starting at these points can possibly process the same node and expression more than once. Thus it suffices to memoize at these points (see Section 5.7 for details).

Page Management. To keep the resource consumption of PAAT low, PAAT minimizes the number of simultaneously retained pages and frees a page as soon as possible, either explicitly or by implicitly replacing it with a new page. To decide whether we can replace a page with a new one, `eval` maintains recursively a flag Free which is set to **true** if a page is not required anymore by the caller – and thus can be overridden or freed. In our example, we visit for nodes 4, 7, 10, and 12 a new page recursively. In the first three of recursive calls, Free is set to **false**, since we still need the current page to follow the last link to another page. Only in case of the last link, Free is set to **true**. When a page is removed from memory, both the browser DOM and the corresponding entries in the lookup tables are freed.

5.2 Context Tuples

`eval` and `eval_` take as input a (full or simple) OXPATH expression, and two sets ICtx and Ctx of context tuples, both of the same type (out of two possibilities): **(1)** XPATH *context tuples* $c_x = \langle c_x.n_x, c_x.p_x \rangle$ depend on their context set to evaluate `position` and `size` and consist of the context node and its parent context node, see [26]. **(2)** *Extraction context tuples* $c_e = \langle c_e.c_x, c_e.p, c_e.l \rangle$ consist of one XPATH context tuple and the ids of the last parent and sibling extraction match – reminiscent of the context tuples in the semantics, allowing subsequent extraction matches to be nested according to the predicate nesting (Section 4.4).

As remarked above, XPATH context tuples are always relative to some context set, which determines the position of the tuple within the set. Since we want to evaluate only part of the Ctx, we maintain those tuples in ICtx $\subseteq$ Ctx, and use Ctx only for determining the position of a tuple. For efficiency, we maintain several context sets in the same program variable. Hence, we select all tuples with same parent context node (and same parent and sibling extraction matches) to obtain the restricted tuple set $\mathsf{Ctx}_{|c_x}$ ($\mathsf{Ctx}_{|c_e}$) containing only tuples from the (proper) context of c_x. Furthermore, we write $\mathsf{Ctx}_{|c_e,c'_e} = \{\langle c''_e.c_x, c'_e.p, c'_e.l \rangle \mid c''_e \in \mathsf{Ctx}_{|c_e}\}$ to adapt parent and sibling match in a restricted context set.

5.3 PAAT Simple Evaluation

The first procedure, `evalT_` (Algorithm 1), evaluates a *simple expression* Expr on a context tuple c_x, belonging to a context set Ctx. As simple expression, Expr is free of actions or extraction markers, but may use other OXPATH features such as Kleene stars. For brevity, Algorithm 1 only deals with the most important expressions, i.e., axis navigation, Kleene stars, and predicates. The omitted parts (mostly functions and operators) do not affect the algorithm design and can be added analogously to predicates. In our design of

```
1  Function evalT_(Expr, c_x, Ctx):
2  if Expr ≡ ε then return {c_x};
3  if Lookup[Expr, c_x] ≠ null then return Lookup[Expr, c_x];
4  Expr ≡ e t where e matches one of the following cases;
5  ICtx ← ∅;Ctx' ← ∅;
6  if e ≡ axis :: nodes then
7      ICtx ← {⟨n'_x, c_x.n_x⟩ | axis(c_x.n_x, n'_x) ∧ n'_x ∈ unary_nodes⟩};
8      Ctx' ← ICtx;
9  else if e ≡ (path) ∗ {v, w} then
10     ICtx ← {c_x}; OCtx ← ∅;
11     for i ← 0 to w − 1 do
12         if i ⩾ v then
13             ICtx ← ICtx\OCtx; OCtx ← OCtx ∪ ICtx;
14         if ICtx = ∅ then break;
15         ICtx ← eval_(path, ICtx, ICtx);
16     ICtx ← {⟨c'_x.n_x, c_x.p_x⟩ | c'_x ∈ ICtx ∪ OCtx};
17     Ctx' ← ICtx;
18 else if e ≡ [q] then
19     if evalT_(q, c_x, Ctx) ≠ ∅ then ICtx←{c_x};
20     Ctx' ← {c_x ∈ Ctx | evalT_(q, c_x, Ctx) ≠ ∅};
21 Result ← eval_(t, ICtx, Ctx');
22 Lookup[Expr, c_x] ← Result;
23 return Result;
```

Algorithm 1: Evaluation of simple OXPATH on tuples

```
1  Function eval_(Expr, ICtx, Ctx):
2  Result ← ∅;
3  if Ctx is extraction context set then
4      for c_e ∈ ICtx do
5          for c_x ∈ evalT_(Expr, c_e.c_x, Ctx) do
6              Result ← Result ∪ {⟨c_x, c_e.p, c_e.l⟩};
7  else for c_x ∈ ICtx do
8      Result ← Result ∪ {evalT_(Expr, c_x, Ctx);
9  return Result;
```

Algorithm 2: Evaluation of simple OXPATH on tuple sets

`evalT_`, we are inspired by polynomial time XPATH evaluation algorithms from [26]: `evalT_` implements a dynamic programming approach in maintaining a memoization table Lookup which maps subexpressions and context tuples to intermediate results.

First we check for empty expressions as *base case* and return the input tuple c_x as result $\{c_x\}$ (Line 2). Next, we check whether the result of applying Expr to c_x has been memoized, and if so, return this result (Line 3). Otherwise, Algorithm 1 proceeds in a *depth-first manner:* It splits Expr into prefix e and remainder t (Line 4) to evaluate e directly and t recursively. In its main part (Lines 5 to 20), Algorithm 1 evaluates e on c_x to obtain the context sets ICtx and Ctx' (see Section 5.2), where e is either an axis with node test, a Kleene star, or a predicate. Finally, the evaluation of the tailing expression t on ICtx and Ctx' yields the result to memoize and return (Lines 21 to 23).

(1) For *axis navigation* (Line 6), we obtain $\mathsf{ICtx} = \mathsf{Ctx}'$ via axis and $\mathit{unary}_{\mathsf{nodes}}$, adjusting the parent node to $c_x.n_x$.

```
1  Function eval(Expr, ICtx, Ctx, Free):
2  if Expr is simple then return eval_(Expr, ICtx, Ctx);
3  Expr ≡ het where h is simple, e is not;
4  ICtx ← eval_(h, ICtx, Ctx, false);
5  if ICtx = ∅ then return ∅ ;
6  Result ← ∅;
7  if e ≡ {action/} ∨ {action} then
8      for c_e ∈ ICtx do
9          f ← Free ∧ isLast(c, ICtx);
10         ICtx'←{⟨getPage(action, c_e.c_x.n, f), c_e⟩, c_e.p, c_e.l⟩};
11         if e ≡ {action} then
12             if isPageUnmodified() then ICtx' ← {c_e};
13             else ICtx' ← eval_(AFP(action, c_e.c_x.n), ICtx', ICtx');
14         Result ← Result ∪ eval(t, ICtx', ICtx', true);
15         if Free then FreeMem(pageof(ICtx'));
16 else if e ≡ ⟨M[= v]⟩ then
17     ICtx' ← ∅ ;
18     for c_e ∈ ICtx do
19         val ← evalT_(v, c_e.c_x, Ctx|c_e);
20         output ⟨OUT(c_e.c_x.n_x, M), c_e.p_x, M, val⟩;
21         ICtx' ← ICtx' ∪ {⟨c_e.c_x, c_e.p, OUT(c_e.c_x.n_x, M)⟩};
22     Result ← eval(t, ICtx', ICtx', Free);
23 else if e ≡ (path) ∗ {v, w} then
24     if path contains an action on the main path then
25         if w > 0 then
26             if v = 0 then Result←Result∪eval(t, ICtx, ICtx, false);
27             Expr' ← path path ∗ {max(0, v − 1), w − 1} t;
28             Result ← Result ∪ eval(Expr', ICtx, ICtx, Free);
29         else Result ← Result ∪ eval(t, ICtx, ICtx, Free);
30     else
31         N ← ∅;
32         for c_e ∈ ICtx do
33             ICtx' ← {c_e};OCtx ← ∅;
34             for i ← 0 to w − 1 do
35                 if i⩾v then ICtx←ICtx'\OCtx;OCtx←OCtx∪ICtx';
36                 if ICtx' = ∅ then break;
37                 ICtx'←eval(path, ICtx', ICtx', Free∧(i=w−1)∧(t≡ε));
38             N←N∪{⟨⟨c'_e.n_x, c_e.p_x⟩, c'_e.p_e, c'_e.l_e⟩ | c'_e∈ICtx'∪OCtx};
39         Result ← eval(t, N, N, Free);
40 else if e ≡ [q] then
41     ICtx' ← ∅ ;
42     for c_e ∈ ICtx do
43         Free' ← Free ∧ (t ≡ ε) ∧ isLast(c_e, ICtx);
44         c'_e ← ⟨c_e.c_x, c_e.l_e, c_e.l_e⟩;
45         if Lookup∃[e, c_e] = false then continue;
46         else if Lookup∃[e, c_e] = true or
47                 eval(q, {c'_e}, Ctx|c_e,c'_e, Free') ≠ ∅ then
48             ICtx'← ICtx'∪{c'_e};Lookup∃[e, c_e] ←true;
49         else Lookup∃[e, c_e] ← false;
50     Result ← eval(t, ICtx', ICtx', Free);
51 return Result;
```

Algorithm 3: Evaluation of full OXPATH on tuple sets

(2) In case of *Kleene star expressions* without actions or extraction markers (Line 9), each successive iteration might return nodes already reached through prior iterations. Thus, in each of the w application of *path*, we avoid traversing paths starting at already analyzed context tuples. More specifically, we collect in OCtx the tuples reached by the compound Kleene star expression, and maintain in ICtx the tuples to be explored. Inside the loop, if we have reached the lower bound v, we remove from ICtx all tuples already collected in OCtx (as they would be redundant), and take all new nodes into OCtx (Line 12). If there are no new tuples left, we break the loop (Line 14), otherwise we evaluate *path* once (Line 15) to complete the iteration. Finally we add OCtx to ICtx and set in all resulting tuples $c_x.n_x$ as parent context (Line 16). The latter groups the tuples in the context ICtx, as ICtx might become part of a larger context set subsequently: This ensures, e.g., that in an expression ψ`/(`ϕ`){2,3}[`i`]`, for *each* node n matching ψ, the i-th node reached via ϕ from n is returned, instead of the *single* i-th node among all nodes reached via ψ`/(`ϕ`){2,3}`.

(3) We deal with *predicate expressions* $[q]$ (Line 18), by recursively evaluating q with `evalT_`, since q must be simple itself. If the evaluation returns a non-empty set, we keep c_x in ICtx, and likewise, determine Ctx' as the subset of Ctx whose nodes satisfy q (Line 20).

Finally, in all cases, we evaluate tail t of Expr with the new context sets ICtx and Ctx' to return the memoized result.

Algorithm 2 iterates over a set of context tuples ICtx and calls `evalT_` for each tuple. It is split into two parts: The first part (Lines 3 to 6) covers the case that ICtx contains extraction context tuples, the other part the case of XPATH context tuples. In the former case, we strip the extraction matches from the tuples to reduce the space for Lookup in `evalT_`, see Section 5.7. Either way, we call `evalT_` for each tuple in ICtx, providing the restriction $\text{Ctx}_{|c_x}$ (or $\text{Ctx}_{|c_e}$) of Ctx to the proper context set of c_x (or c_e). In case of extraction tuples, the algorithm reattaches the parent and sibling matches to the returned nodes in Line 6.

5.4 PAAT Full Evaluation

In Algorithm 3, we show `eval` for handling full OXPATH. Building upon `eval_`, `eval` deals with actions and extraction markers, and expressions that contain actions and extraction markers as subexpressions, i.e., Kleene stars and predicates. Next to exploiting the memoization of `eval_`, `eval` also memorizes the outcome of predicate evaluations in its own lookup table $\text{Lookup}_\exists$. As in `evalT_`, we omit functions and operators for clarity; they are handled analog to predicates.

Performing the *actions* in the expression, `eval` traverses the page tree in a depth-first manner. For Kleene stars without actions on the main path, `eval` works similar to `evalT_`,

since all resulting nodes are on the same page, even if actions within the predicates of the Kleene star expressions may navigate to other pages. While the input context set Ctx only contains nodes from a single page, the result set Result, however, may contain nodes from many pages.

If Expr is simple, we delegate its evaluation to `eval_` and return the result (Line 2). Otherwise, we split Expr into three parts (Line 3): h is the maximum prefix which is simple, e is either an action, an extraction marker, or a Kleene star or predicate containing an action or extraction marker, and t is the remaining expression. The prefix h is evaluated with `eval_` (Line 4), and the result is returned if empty (Line 5).

(1) The first main case deals with *actions* (Line 7), covering both, absolute $\{action\}$ and contextual actions $\{action/\}$. Roughly, we iterate over all c_e in ICtx, obtaining per c_e a new context set ICtx′ with a single tuple. That tuple is either the root of the page returned by the *action* applied to c_e or the result of evaluating the action free prefix on that root. Either way, the parent and sibling extraction matches are copied from c_e. In `getPage`, we free the page to perform the action upon, if the input flag Free is set and c_e is last in the iteration (Line 9). Upon freeing a page, all memoization information in Lookup and Lookup$_\exists$ related to this page is freed, too. If the action is contextual (Line 11) and did not alter the page, we stay at c_e to avoid evaluating the action free prefix $\text{AFP}(action, c_e.c_x)$ (Line 12), as done otherwise. Either way, we evaluate t recursively on ICtx′, descending one step further in the depth-first traversal of the page tree (Line 14). We set Free to **true**, since the page and all related memoized information is freed after the invocation in any case (Line 15).

(2) For *extraction markers* (Line 16), first the value to be extracted is evaluated with `evalT_`, as extraction values are always computed from simple expressions (Line 19), and the obtained result tuple written to the output (Line 20). Finally, we add a tuple to ICtx′ that is identical to c_e up to the sibling match which is set to $\text{OUT}(c_e.c_x.n_x, m)$. The tail is then recursively evaluated with the new context set ICtx′.

(3) *Kleene stars containing actions* are treated in two cases: (i) If a Kleene star contains an action on its main path, we expand the expression (Line 27) and recursively evaluate the expanded expression (Line 28). Once the expansion has reached the lower bound, i.e., v became 0, we also collect results by evaluating the tail t (Line 26). The results for the final iteration are collected in separately (Line 29). By evaluating the tail t at each individual recursion step, we avoid context sets with nodes from different pages and nevertheless evaluate the same expression only once for the same context, as each iteration yields nodes from different pages. (ii) Otherwise, without actions on the main path (Line 30), different iterations can never re-reach a node already processed, and hence, we use similar strategy as in `evalT_`.

(4) It remains to address *predicates containing actions or extraction markers* (Line 40). Here, we need to evaluate the contained expression q for every c_e in ICtx to test if it yields $\varnothing$. Since q contains an action or extraction marker, we need to use `eval`. Doing so without memoization would lead to an exponential runtime, due to expressions such as `//a[.//b[{click/}][.//c[{click/}][...]]]` – requiring another lookup table Lookup$_\exists$. Here, we need to memoize an entry per extraction context tuple and expression, however, as result we only store **true** or **false**. We construct the extraction context tuple c'_e for evaluating the predicate by taking the sibling extraction match as new parent match (Line 44). Then q is evaluated over $\{c'_e\}$ with $\text{Ctx}_{|c_e,c'_e}$ (see Section 5.2) as new context set (Line 46). It remains to evaluate tail t on the filtered context ICtx′ (Line 50).

5.5 PAAT Example

With the algorithms at hand, we now revisit the example shown in Fig. 9 and 10 to discuss its processing in detail.

1—Navigation. PAAT starts with `eval`(Expr, Ctx, Ctx, **true**) where $\text{Ctx} = \{\langle\langle\bot,\bot\rangle, \text{results}, \text{results}\rangle\}$. Expr is split into $h = \varepsilon$, $e =$`doc(uri)`, and $t =$`descendant-or-self::node()...`, see Line 4 of Algorithm 3. As h is empty, the call to `eval_` in Line 3 returns the unchanged context tuples, continuing with $e =$`doc(`u`)`, an absolute action to load the new page (Line 10). Thereby, we $\text{Free}' = \text{Free} =$ **true**, since there is only a single tuple in the context set. On loading u, `getPage` returns 0, the root of the page tree in Fig. 9. The context tuple $\langle\langle 0,\bot\rangle, \text{results}, \text{results}\rangle$ is used for processing the tail recursively (Line 14).

In the recursive invocation on 0 (yielding the second box in Fig. 10), the former tail t becomes Expr, is split into $h =$ `/descendant-or-self::node()/child::div`, $e =$ `:<R>`, and the rest t. First, `eval` evaluates h with `eval_` which calls `evalT_` for each single context node. Since `evalT_` has no memoized data available for h (Line 3), it splits Expr into `descendant-or-self::node()` and `child::div` (Line 4). Evaluating the first expression, the context is expanded in Line 7 to all nodes in the page and `child::div` is evaluated on these nodes with `eval_` which calls `evalT_` once for each node. In Fig. 10, we summarize these calls with three boxes: Starting at 0, `descendant-or-self::node()` yields a summary box for the DOM nodes 1...12 (albeit, there is one call for each node), and a box for 0. Finally, `child::div` leads from 0 to 1.

Thus `eval_` returns $\langle\langle 1,\bot\rangle, \text{results}, \text{results}\rangle$ to `eval` to continue with the evaluation of `:<R>` (Line 16). For that context, $\langle\text{OUT}(1,R), \text{results}, R, \text{null}\rangle$ is written to the output by the evaluation of `:<R>` (Line 20). In Fig. 9 the new tuple is shown as a (square) node R.

`eval` continues recursively with the rest of the expression using the context tuple $\langle\langle 1, \bot\rangle, \mathsf{results}, \mathrm{OUT}(1,R)\rangle$, carrying the fresh output node $\mathrm{OUT}(1,R)$ as new sibling marker.

The rest of the expression is split again, this time into the simple expression h of four XPath steps between `:<R>` and the predicate, the predicate (as e) and an empty tail. The evaluation of the simple expression is again delegated to `evalT_` (via `eval_`): 1 has all nodes except $\bot$ as descendants, but only the only nodes with `p` are 1, 5, and 8, all others will return subsequently empty results. The evaluation continues in a depth-first manner, evaluating the left most branches first. Fig. 10 shows how we first find all `a` descendants of `p` descendants of 1, memoizing the results at every step. When we later search for such nodes for 5 and 8, we find that all matching nodes have been already evaluated and the corresponding results memoized.

2—Predicate Evaluation. The evaluation of the predicate in `eval` starts with the context set containing one context tuple for 4, 7, 10, and 12. For example, with 4, we get the tuple $\langle\langle 4,2\rangle, \mathsf{results}, \mathrm{OUT}(1,R)\rangle$. To evaluate $e =$ `[{click/}/....]`, `eval` changes the tuple to $\langle\langle 4,2\rangle, \mathrm{OUT}(1,R), \mathrm{OUT}(1,R)\rangle$, such that the last sibling match $\mathrm{OUT}(\mathtt{div}_1,R)$ becomes the parent in the recursive call to `eval` (Line 46). Thus, all tuples extracted during the predicate evaluation (i.e., within this branch of the call tree) are descendants of $\mathrm{OUT}(1,R)$.

The recursive calls to `eval` split Expr into $e =$ `{click /}` and $t =$ `/desc-or-self::node()/title:<t=string(.)>`. In the first three of these four calls, Free is set to **false**, since the page is still needed, but in the last invocation, Free is set to **true**, such that the page buffer of the current page can be reused. Accordingly, the action in e opens the linked page with `getPage` (Line 10) either into a new or the current page buffer, overwriting the old page. In any case, the context now refers to the root node of the new page. `eval` evaluates t recursively (Line 14), setting Free to **true** in any case, the newly loaded page will be freed after this invocation anyway (Line 15).

The subsequent recursive invocations on the new page navigate to the title node for value extraction. Hence, with i ranging over the reschaed title nodes, `evalT_` outputs the tuple $\langle \mathrm{OUT}(i,t), \mathrm{OUT}(1,R), t, \mathsf{val}_i\rangle$ (diamond in Fig. 9) as child of $\mathrm{OUT}(1,R)$, associated with its textual content val_i (Line 20).

5.6 Intensional Axes

So far, we did not consider the evaluation of intensional axes. As we will show in Theorem 8, we could rather assume that they are precomputed on page load. In practice, however, it is usually preferable to delay that computation and to use memoization, requiring a small modification to the PAAT algorithm: First, we must explicitly manage the environment containing the variable bindings. While technically necessary for plain XPath, we omitted the variable bindings in the algorithms for OXPath without intensional axes for clarity, as these bindings are handed through all recursive invocations unaltered. Second, in Line 7 of `evalT_`, we can no longer assume that the axis is precomputed, but rather need to determine all nodes related to the current context node by calling `eval` with the expression defining the intensional axis and an updated environment. If the result is not empty, the node is related to the context node and added to Ctx'. To avoid multiple evaluations of the same intensional axis, we guard this evaluation with a memoization table similar to $\mathsf{Lookup}_\exists$, but with an additional entry for the related node. In Section 6, we show that, implemented in this way, intensional axes have little impact on the practical performance of OXPath.

5.7 Analysis of PAAT

In Table 5, we summarize the complexity of OXPath and some sublanguages. We denote the size of the query with q, with n the (maximum) number of nodes in a document, with p the number of pages in the page tree, and with d its depth tree. On the left of the table we show which features of OXPath are allowed, **X**, or not allowed, —. (**X**) indicates that the complexity is not affected by removing or adding the particular feature. The considered features are extraction markers (normalized only or any), actions (absolute or contextual), Kleene stars (bounded only or any). We also give the number of page buffers PAAT maintains at most for each class. In the following, we give proofs for the most important cases. Intensional axes are considered separately in Theorem 8.

Theorem 5 *Evaluating a simple* OXPath *expression with* `evalT_` *takes at most* $O(q^2 \cdot n^4)$ *time and* $O(q^2 \cdot n^3)$ *space where q is the size of the expression and n the number of nodes in all documents reached by the evaluation.*

In case of simple OXPath[3], n is the number of nodes in the start document.

Proof The proof follows closely the proof of Theorem 6.6 in [26], since simple OXPath is roughly comparable to XPath, adding only Kleene stars, the **`style`** axis and a few operators. Due to the memoization, the Kleene star does not affect the complexity (it does, however, impact practical performance, as it generates on average far more Lookup entries than other expressions).

PAAT is a top-down, recursive implementation of the context value table principle from [26].

We first show that evaluating a simple OXPath expression using `evalT_` takes $O(l_- \cdot T_{\mathsf{op}})$ time and $O(l_- \cdot S_{\mathsf{val}})$

[3] Simple OXPath is the restriction of OXPath to simple OXPath expression, but we allow a `doc()` action at the start of the expression to set the document to be queried.

extraction		actions		Kleene		Time	Space	Page buffers	
norm.	any	abs.	context.	bounded	any				
—	—	—	—	—	—	$O(q^2 \cdot n^4)$	$O(q^2 \cdot n^3)$	$O(1)$	XPATH + **style**, ...
—	—	—	—	X	(X)	$O(q^2 \cdot n^4)$	$O(q^2 \cdot n^3)$	$O(1)$	simple
X	—	—	—	—	—	$O(q^2 \cdot n^4)$	$O(q^2 \cdot n^3)$	$O(1)$	
X	X	—	—	—	—	$O(q^2 \cdot n^4)$	$O(q^2 \cdot n^4)$	$O(1)$	
—	—	X	(X)	—	—	$O(q^2 \cdot (p \cdot n)^4)$	$O(q^2 \cdot (\min(q,d) \cdot n)^3)$	$O(\min(q,d))$	
—	—	X	(X)	X	X	$O(q^2 \cdot d \cdot p^2 \cdot n^4)$	$O(q^2 \cdot d^3 \cdot n^3)$	$O(\min(q,d))$	Extraction-free
X	—	X	(X)	—	—	$O(q^3 \cdot p^4 \cdot n^4)$	$O(q^6 \cdot n^3)$	$O(\min(q,d))$	
X	X	X	(X)	—	—	$O(q^3 \cdot p^4 \cdot n^4)$	$O(q^7 \cdot n^4)$	$O(\min(q,d))$	Kleene-free
X	—	X	(X)	X	X	$O(q^2 \cdot d \cdot p^3 \cdot n^4)$	$O(q^2 \cdot d^4 \cdot n^3)$	$O(d)$	normalized
X	X	X	(X)	X	X	$O(q^3 \cdot d \cdot p^4 \cdot n^4)$	$O(q^3 \cdot d^5 \cdot n^4)$	$O(d)$	full

Table 5 Complexity of OXPATH family

space where $l_{\_}$ is the maximum number of entries, S_{val} the maximum size of an entry in the lookup table Lookup in `evalT_`, and T_{op} the maximum cost for evaluating a function or operator. This holds as the body of `evalT_` runs at most once per entry in Lookup. For each entry, the time for executing `evalT_` is bounded by the maximum time for evaluating a function or operator, as this time dominates the other cases in `evalT_` (all bounded by $O(n)$, whereas the function or operator evaluation is $\geqslant n$). For size, observe that other than the size of the lookup table, we only manage the three contexts Ctx, Ctx′, and Ctx″ and the Result, the first three bounded by $O(n^2)$, the second by $O(n)$, and thus dominated by the size of Lookup.

Second, we show that **(1)** $l_{\_} \in O(q \cdot n^2)$. This follows immediately from the signature of Lookup. **(2)** $S_{\mathsf{val}} \in O(q \cdot n)$. `concat()` and multiplication are the operations that yield the largest increase in value size and the resulting values are bounded by $O(q \cdot n)$, see [26]. **(3)** $T_{\mathsf{op}} \in O(q \cdot n^2)$. Again following Theorem 6.6 in [26] we observe that the most expensive operation is = which compares two nodesets of up to $O(n)$ size. Together with the bound on value size we obtain a bound of $O(q \cdot n^2)$ for T_{op}. The added axes or comparison operators in OXPATH do not affect this result if we assume a pre-computed table for ~ and ~= as for =. We also deviate in how we compute `position()` and `size()` by projecting the context set to tuples with the same parent and sorting the result. However this is done in $O(n \log n)$ and thus dominated by the time for =. □

Proposition 5 *Evaluating an* OXPATH *expression with* `eval` *takes* $O(N_{\mathsf{Expr}} \cdot N_{c_e} + N_{\mathsf{Expr}} \cdot N_{c_x} \cdot T_{op})$ *time and* $O(N_{\mathsf{Expr}} \cdot S_{\mathsf{Ctx}} + l_{\exists} + l_{\_} \cdot S_{val})$ *where* N_{Expr} *the number of subexpressions evaluated by recursive calls in the evaluation,* N_{c_e} *the number of extraction context tuples,* N_{c_x} *the number of* XPATH *context tuples from all reached pages,* S_{Ctx} *the maximum size of a context set in* `eval`, $l_{\exists}$ *the maximum size of* $\mathsf{Lookup}_{\exists}$, $l_{\_}$ *the maximum size of* Lookup, *and* S_{val}, T_{op} *as in Theorem 5.*

Proof We first consider space complexity: `eval` uses $\mathsf{Lookup}_{\exists}$ and (indirectly) the lookup table Lookup for `evalT_`. Additionally, we have to account for the various context sets and the result set Result. The latter can be streamed out rather than stored, as it is never processed further. The context sets are accounted for by the $S_{\mathsf{Ctx}} \cdot N_{\mathsf{Expr}}$, as each call to `eval` uses a constant number of such sets and the depth of the call graph (and thus the number of simultaneously stored context sets) is bounded by N_{Expr}. It suffices to consider $l_{\_} \cdot S_{\mathsf{val}}$ for the impact of `evalT_`, as Theorem 5 shows that this expression dominates its space complexity.

For time complexity, observe that `eval` is called at most $N_{\mathsf{Expr}} \cdot N_{c_e}$ times. This holds since, `eval` is never called twice on the same expression for the same context tuple other than in Line 46 where two calls may use the same context tuple c'_e (originating from context tuples with different parent matches). However, in that case the total number of calls is still bounded by the original ICtx set.

In each such call, `eval` may delegate the evaluation of the expression or some subexpression to `evalT_`. Due to the memoization in `evalT_` the total time for all these calls is, however, bounded by $N_{\mathsf{Expr}} \cdot N_{c_x} \cdot T_{\mathsf{op}}$ as any repeated calls immediately return the memoized result and the memoization tables are kept until all nodes from the page are processed.

Other than those calls and recursive calls to itself on a proper subexpression, `eval` only requires constant time per node if we assume that action execution is constant.

Theorem 6 *Evaluating a full* OXPATH *expression requires* $O(q^3 \cdot d \cdot p^4 \cdot n^4)$ *time and* $O(q^3 \cdot d^5 \cdot n^4)$ *space.*

Proof For this proof, we first observe the invariant on `eval`: each context set contains only context tuples with context nodes from one page.

For full OXPATH the following bounds hold (q query size, d depth of the page tree reached by the evaluation, n maximum number of nodes on a page, p number of pages reached by the evaluation) **(1)** N_{Expr} is bounded by $O(q \cdot d)$ not $O(q)$ as the expansion of Kleene stars in Line 27 introduces new expressions. However, there can be at most $d+1$ such expansions on the path to the root from any leaf expression, as each expansion includes at most one action and, af-

ter d expansions any additional expansion yields an expression with $d+1$ actions on a single path and thus an empty result as the page tree is bounded by d. In this case the expansion is stopped and we obtain the bound of $O(q \cdot d)$. For the evaluation of action-free prefixes, we at most double this number (if each step contains a contextual action). **(2)** N_{c_e} is bounded by $O(q^2 \cdot p^4 \cdot n^4)$ since there are at most $p \cdot n$ (parent or actual) context nodes, each such pair combined with at most $q \cdot p \cdot n$ different parent and sibling matches, since they must originate from an extraction marker on the path to the root and such a path has at most q distinct such expressions. Kleene star expansion may cause the same extraction marker to occur in several positions, but matches for all occurrences are indistinct. **(3)** N_{c_x} is similarly bounded by $O((p \cdot n)^2)$. **(4)** T_{op} and S_{val} are as in Theorem 5, as we do not allow actions in operands of functions and operators and thus an operand is limited to nodes from a single page. The value size is also not affected by Kleene star expansions. **(5)** $l_{\_}$ is bounded by $O(N_{\mathsf{Expr}} \cdot (d \cdot n)^2)$, since only the lookup entries from at most d pages are active at a time. **(6)** $l_{\exists}$ is similarly bounded by $O(N_{\mathsf{Expr}} \cdot q^2 \cdot d^4 \cdot n^4)$ since there are only tuple from at most d pages with at most n nodes stored in $\mathsf{Lookup}_{\exists}$ and each of those may be combined with $q \cdot d \cdot n$ parent and the same number of sibling markers. **(7)** S_{Ctx} is bounded by $O(n^2 \cdot q^2 \cdot d^2 \cdot n_m^2)$ as each context set contains only extraction context tuples for context nodes from one page (but extraction matches may originate from any page on the current branch of the page tree). With this, the complexity follows from Theorem 5. □

Theorem 7 *Evaluating a normalized* OXPATH *expression takes $O(q^2 \cdot d \cdot p^3 \cdot n^4)$ time and $O(q^2 \cdot d^4 \cdot n^3)$.*

Proof For normalized OXPATH we can drop the sibling extraction markers from the extraction context tuples in `eval` and `eval_`.

The complexity remains as for full OXPATH except for **(1)** N_{c_e} is bounded by $O(q \cdot p^3 \cdot n^3)$ since we do not need to maintain sibling extraction matches. **(2)** $l_{\exists}$ is bounded by $N_{\mathsf{Expr}} \cdot (q \cdot d^3 \cdot n^3)$ for the same reason. **(3)** S_{Ctx} is bounded by $O(q \cdot d \cdot n^3)$ as each context set contains only extraction context tuples for context nodes from one page (but parent extraction matches may originate from any page on the current branch of the page tree). With this, the complexity follows from Theorem 5. □

Theorem 8 *Intensional axes increase the complexity of* OXPATH *and any sub-language including* XPATH *by at most a factor of $O(n^2)$ where n is the total number of nodes in the page tree.*

Proof In general, intensional axes can be evaluated as follows: First, we materialize all intensional axes in an expression bottom-up. Then, the expression is evaluated over the intensional axis as usual. The actual evaluation complexity is not affected as the two necessary operations, testing if a pair of nodes is in an axis and iterating over all nodes that are related to a given context node by an axis, retain their (constant, resp. linear) complexity.

For the materialization of the intensional axes, we first note that there are at most q such axes. For each axis, we need to store at most $O(n^2)$ tuples. To compute the materialization, we need to evaluate the expression inside the intensional axis at most once for each pair of nodes, binding each successively to `$lhs` and `$rhs`. □

Theorem 9 (Memory minimality) *Let L be an* OXPATH *expression without actions in predicates. Then there exists 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.*

Proof An expression from L has the shape $e_d = \texttt{doc}(w)r_d$ with $r_d = \texttt{/}\phi_d\texttt{/\{action\}}r_{d-1}$ for $d > 1$ and $r_1 = \varepsilon$. 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 a page w that match the corresponding ϕ_i, 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.

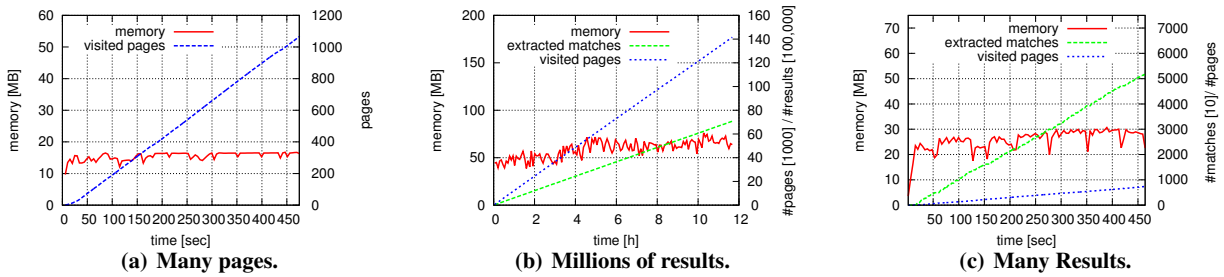
Any other algorithm A must load the leaves of the accessed page tree of PAAT as these nodes are the result of evaluating $[\![e_d]\!]_V$. To visit such a leaf node l of the accessed page tree, we have to load its parent p first, because without prior knowledge all children of p are only accessible by 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. □

6 Evaluation

We confirm OXPATH's scaling behavior as follows:

(1) The theoretical complexity bounds from Section 5.7 are confirmed in several large scale extraction experiments in diverse settings, in particular the constant memory use even for 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 1.1) significantly affects the scaling behavior or the dominance of page rendering.



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

Fig. 11 Scaling OXPATH: Memory, visited pages, and output size vs. time.

(3) In an extensive comparison with commercial and academic web extraction tools, we show that OXPATH outperforms previous approaches by at least an order of magnitude. Where OXPATH stays within constant memory bounds independently of the number of accessed pages, most other tools require 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. Fig. 11(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 (130 pages/min). The number of retrieved pages grows linear in time, while the memory size remains constant throughout. The jitter in memory use is due to the repeated ascends and descends of the citation hierarchy. Fig. 11(b) shows the same test mirroring 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, i.e., repeatedly clicking on all links on certain pages, chosen from sites with different characteristics: In one experiment we took very large pages from Wikipedia, and in another one we took pages from Google Products to reach different web shops with their typically visually elaborate pages. Fig. 11(c) demonstrate OXPATH's constant memory usage in the face of an increasing number of visited pages. For large pages the results are very similar.

Profiling: Page Rendering is Dominant. We profile each stage of OXPATH's evaluation performing five sets of queries on the following 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 all links and extract the `html` tag of the resulting pages. Figures 12(a) and 12(b) show the total and page-wise averages, respectively. For `bing.com`, the page rendering time and

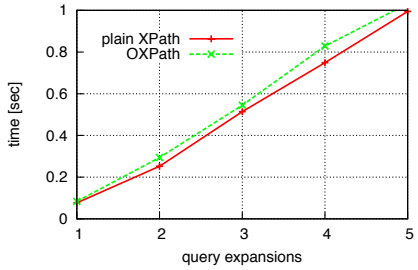
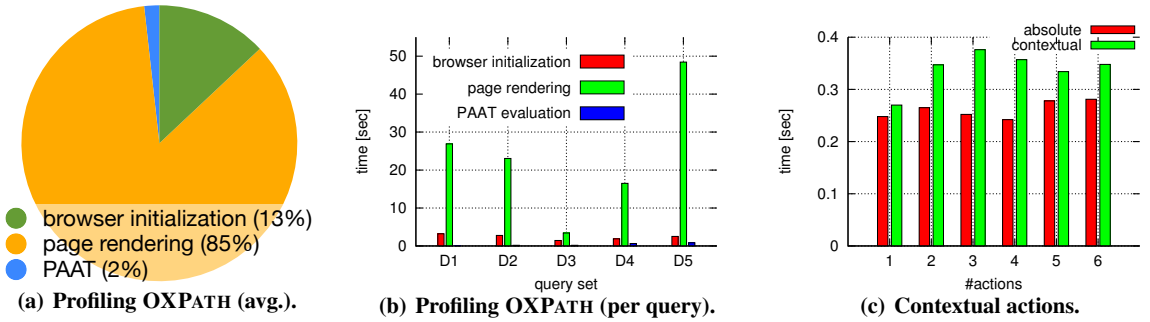


Fig. 13 OXPATH vs. Plain XPATH

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 analyzes 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. Fig. 12(c) shows the results with queries containing 1 to 6 actions on Google's advanced product search form. Contextual actions suffer an insignificantly small penalty in their evaluation time as compared with their absolute equivalents.

Actions (as well as extraction markers and Kleene star expressions) affect the evaluation notably, as expected. This contrasts sharply with the added selection capabilities: Neither `style`, `field()`, nor the added selectors significantly impact evaluation performance. The same holds for intensional axes. Fig. 13 summarizes this observation showing the evaluation time for a series of expansions of a simple expression. It compares the case where the expression is pure XPATH (`/descendant-or-self::*[self::*]` repeated 1 to 5 times) with the case where the expression uses a `style` axis (instead of `self::*`). The results are nearly identical for intensional axis or any of the other added features, if expression size is properly accounted for. We show results for up to 5 repetitions, but observe that the roughly 10% overhead holds also for much larger expressions. Note that these results are affected by our use of the XPATH engine provided by the browser, which does not perform any optimization of XPATH expressions.



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

Fig. 12 Profiling OXPATH's components.

Order-of-Magnitude Improvement. We benchmark OXPATH against four commercial web extraction tools, namely, Web Content Extractor [2] (WCE), Lixto [12], Visual Web Ripper [3] (VWR), the academic web automation and extraction system Chickenfoot [17], and the open source extraction toolkit Web Harvest [4]. 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, particularly for realizing multi-way navigation. 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 disregard tools such as RoadRunner [22] or XWRAP [33], since 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 choose an extraction task on Google Scholar as benchmark example, since it does not require scripted actions. On heavily scripted pages, the performance advantage of OXPATH is even more pronounced. With each system, we navigate the citation graph to a depth of 3 for papers on "Seattle".

Specifically, we implement the following OXPATH expression in the other systems for our experiments:

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

An equivalent Web Harvest program takes 54 lines, whereas an equivalent Chickenfoot script takes 27, and the other tools use visual interfaces.

We record evaluation time and memory consumption for each system. We measure the normalized evaluation time, in which we discount the time for page loading, cleaning, and rendering. This allows for a more balanced comparison as the differences in the employed browser or web cleaning engines affect the overall runtime considerably. Fig. 14(a) shows the results for each system up to 150 pages. Though Chickenfoot and Web Harvest do not render pages at all or do not manage page and browser state, OXPATH still outperforms them. The systems that manage state similar to OXPATH are between two and four times slower than OXPATH even on this small number of pages.

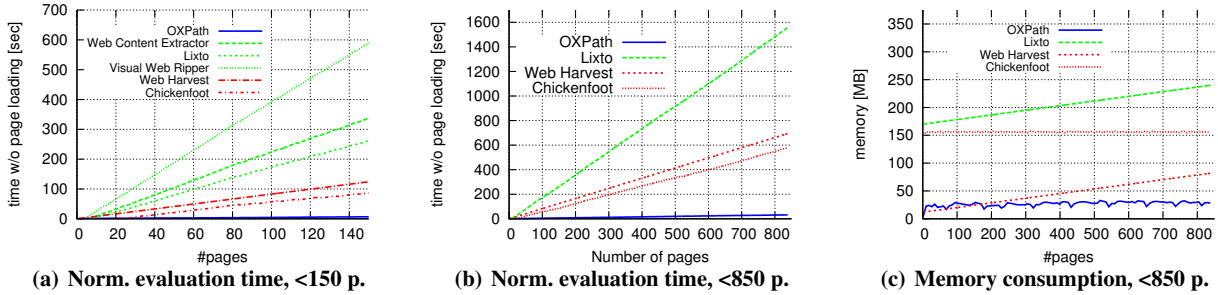
Fig.14(b) illustrates the normalized evaluation time up to 850 pages. In this case, we omits WCE and VWR as they were not able to run these tests. Figures 14(a) and 14(b) show a considerable advantage for OXPATH, amounting at least one order of magnitude.

Fig. 14(c) illustrates the memory use of these systems. WCE and VWR are again excluded, but they show a clearly linear memory usage in those tests we were still able to run. Among the systems in Fig. 14(c), 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 consumption just as OXPATH, though it takes 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 manually using the browser's history whenever possible. This forces reloading when a page is no longer cached, 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.

7 Related Work

The automatic extraction and aggregation of web information is not a new challenge. Almost all previous approaches require either **(1)** service providers to deliver their data in a structured fashion, as in the Semantic Web, or **(2)** clients to wrap unstructured information sources to extract and aggregate relevant data. The first case levies requirements service providers have little incentive to adopt, rendering client-side wrapping as only realistic choice.

As recognized in [6], wrapping web sites has become even more involved with the advent of AJAX-enabled web applications which reveal the relevant data only during user interactions. Previous approaches to web extraction [34,46]



(a) Norm. evaluation time, <150 p. **(b) Norm. evaluation time, <850 p.** **(c) Memory consumption, <850 p.**

Fig. 14 Comparison.

do not adequately address web page scripting. Where scripting is addressed [12,42,15,47], the simulation of user actions is neither declarative, nor succinct, but rather relies on imperative action scripts and standalone, heavy-weight extraction interfaces. Web automation tools such as [17,39] are increasingly able to deal with scripted web applications, but are tailored to automate a single sequence of user actions. Hence they are neither convenient nor efficient for large-scale data extraction with their inherent multi-way navigation, necessary to reach all relevant information pieces by following multiple links on the same page.

Thus, in the following, we particularly consider **(1)** filling and submitting (scripted) web forms, **(2)** multi-way navigation, and **(3)** memory management for large scale extraction. We focus on supervised extraction tools, categorized into web crawlers, web extraction IDEs, extraction languages, and web automation tools. Thus, we exclude unsupervised web extraction tools (see [21] for a survey), as they focus on automated analysis rather than extraction, rendering them rather incomparable to OXPATH.

Web Crawlers. Most commonly employed by search engines for indexing web pages, such crawlers store the relevant information on any found web pages and move on to new pages by traversing all present hyperlinks. Due to the commercial relevance of web crawlers, rather little published research exists, as compared to their prominence in industrial applications. Nevertheless, some work has been published, e.g., most famously on the Google crawler [18]. Acting only on static page representations, such crawlers are unable to handle dynamic, scripted content.

Thus, these web crawlers, in their current state, are incapable of extracting information from content reachable only via some user interaction. [14] first recognized that such amount of content exceeds the quantity of data accessible by hyperlink traversal by far, and is assumed to grow in importance ever since [28] OXPATH expressions can specify the crawling of scripted websites by following web links, submitting forms, etc.

Web Extraction IDEs. Web extraction IDEs, such as [12], have a wider scope than OXPATH, as they provide an entire development environment (e.g., extraction cluster on Amazon Cloud EC2, full support of XPATH 2.0). But in terms of extraction speed and memory consumption, OXPATH outclasses these systems by a wide margin (see Section 6). Lixto, Visual Web Ripper [3], and Web Content Extractor [2] are interactive wrapper generator frameworks, recording user actions in browsers to replay these actions for extracting data. As our experimental evaluation (Section 6) demonstrates, the memory footprint of these systems grows linear in the number of accessed pages – in contrast to OXPATH's constant memory requirements.

Deep web extraction tools such as [15] increasingly deal with scripted, highly visual web sites, but infer the extraction scripts automatically from user-provided examples. Though allowing for easy wrapper generation, such an approach lacks the precision necessary for many fully automated tasks. An another example, BODE [47] is a browser-based extraction tool, whose imperative extraction language BODED is not portable and hard to optimize. Without minimizing the memory requirements, BODE replicates complete browser instances for multi-way navigation, imposing a significant performance penalty, rendering BODE unsuitable for large-scale data extraction.

Web Extraction Languages. Most extraction languages follow a declarative approach [37,9,44,34,46,45], much like OXPATH. However, they do not adequately facilitate deep web interaction, such as form submissions, often due to their age. Also, they do not provide native constructs for page navigation, apart from retrieving a page from a given URL. As an exception, the BODED extraction language [47] deals with modern web applications, but is unsuitable for large-scale extraction tasks, as discussed in the previous section.

In our evaluation, we compare with Web Harvest [4], a recent, open source extraction language. Extraction tasks are specified as imperative scripts, formulated in XML. Web Harvest does not deal with interactive web applications and does not give access to the rendered page, but rather to a cleaned XML view of HTML documents.

Another strand of research employed XML technologies, e.g., XPATH, for information extraction. As a notable example, ANDES [40] is capable of navigating modern web

interfaces, but only by generating URLs from naively filled forms and feeding these URLs back to the underlying crawler. In contrast, OXPATH embeds extraction and navigation into a single seamless process, handling more complicated web interfaces in a more intuitive manner.

Otherwise similar to ANDES, the approach in [7] is limited to generalizing tree traversal patterns. A third example are L-wrappers [10], albeit limited to scraping data from result pages returned on query submissions. In [20], the authors reported on an XPATH-based interface for web forms, but did not release their work so far. As a final example, Web-Prospector [38] processes the Deep Web within the science domain, but appears to be limited to this domain.

XLog [46] extends the ideas from Elog (the datalog-based extraction formalism underlying Lixto [12]) for information extraction by embedding (procedural) extraction predicates. It is optimized for large-scale information extraction tasks, but does not address any kind of web interaction such as form filling and page navigation. Earlier work also explores declarative languages for specifying extraction [44, 37, 9], but does not sufficiently support interaction or page scripting.

W4F [44] offers wysiwyg support for wrapper specification, whereas extraction rules are specified using HEL (HTML Extraction Language), an SQL-like language for HTML elements selection in the spirit of WebSQL [37] and WebOQL [9]. However, all these languages do not adequately address web interaction.

Web Automation Tools. Web automation tools mainly focus on single navigation sequences to automate a single task, but do not consider large-scale web extraction with their need for low overhead and multi-way navigation. Coscripter [31] and iMacros [1] are examples of such tools, not supporting multi-way navigation due to their limited iterative and conditional programming constructs. Vegemite [32] is a CoScripter extension that introduces some extraction capabilities, such as querying some value for a number of inputs. However, as its authors note, such navigation patterns are expensive, since the same page might be reloaded many times. Furthermore, as the page state is not preserved, some web applications may not behave as expected.

The same applies to Chickenfoot [17], a language for web automation running its scripts in Firefox. Chickenfoot scripts are essentially imperative Javascript programs which contain loops and iterations, enabling interaction with forms as well as loading and navigating pages. Multi-way navigation is possible, but only by explicit "back" instructions commanding the browser to return to previous pages. Thus, page buffering is unnecessary, but for a high price: Page states are lost, and thus, pages must be rendered anew for each branch leaving a page during a multi-way navigation.

There are some other systems relying on recorded user actions, e.g., WebVCR [8] and WebMacros [43], or more recently [39]. All these tools suffer from limitations on modern web pages and consider only single action sequences rather than scalable multi-path data extraction tasks.

More recent work [48] addresses the issue of filling web forms automatically. This work, however, does not offer declarative scripting and makes several simplifying assumptions we do not take – for example, they consider drop down lists as the only dynamic content.

8 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.

9 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 (DIADEM). 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. The views expressed in this article are solely those of the authors.

References

1. www.iopus.com/iMacros.
2. www.newprosoft.com/web-content-extractor.htm.
3. www.visualwebripper.com.
4. www.web-harvest.sourceforge.net.
5. http://www.w3.org/TR/CSS2/selector.html.
6. A. Alba, V. Bhagwan, and T. Grandison. Accessing the deep web: when good ideas go bad. In *OOPSLA*, 2008.
7. T. Anton. XPATH-wrapper induction by generalizing tree traversal patterns. In *LWA*, 2005.
8. V. Anupam, J. Freire, B. Kumar, and D. Lieuwen. Automating web navigation with the webvcr. In *WWW*, 2000.

9. G. O. Arocena and A. O. Mendelzon. Weboql: Restructuring documents, databases, and webs. In *ICDE*, 1998.
10. C. Badica, A. Badica, E. Popescu, and A. Abraham. L-wrappers: concepts, properties and construction: A declarative approach to data extraction from web sources. *Soft Comput.*, 11(8):753–772, 2007.
11. M. Banko, M. J. Cafarella, S. Soderland, M. Broadhead, and O. Etzioni. Open Information Extraction from the Web. In *IJCAI*, 2007.
12. R. Baumgartner, S. Flesca, and G. Gottlob. Visual web information extraction with Lixto. In *VLDB*, 2001.
13. M. Benedikt and C. Koch. Xpath leashed. *CSUR*, 41(1):3:1–3:54, 2009.
14. M. K. Bergman. The deep web: Surfacing hidden value. *J. Electronic Publishing*, 7(1):1–17, 2001.
15. J. P. Bigham, A. C. Cavender, R. S. Kaminsky, C. M. Prince, and T. S. Robison. Transcendence: enabling a personal view of the deep web. In *IUI*, 2008.
16. P. Boldi, B. Codenotti, M. Santini, and S. Vigna. Ubicrawler: a scalable fully distributed web crawler. *Software: Practice and Experience,*, 34(711–726), 2004.
17. M. Bolin, M. Webber, P. Rha, T. Wilson, and R. C. Miller. Automation and customization of rendered web pages. In *UIST*, 2005.
18. S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1-7):107 – 117, 1998.
19. M. J. Cafarella, A. Y. Halevy, D. Z. Wang, E. Wy, and Y. Zhang. WebTables: exploring the power of tables on the web. *PVLDB*, 1(1):538 – 549, 2008.
20. V. L. Centeno, C. D. Kloos, L. S. Fernández, and N. F. García. Intelligent automated navigation through the deep web. In *Advances in Web Intelligence*, 2004.
21. C.-H. Chang, M. Kayed, M. R. Girgis, and K. F. Shaalan. A survey of web information extraction systems. *TKDE*, 18(10):1411–1428, 2006.
22. V. Crescenzi, G. Mecca, and P. Merialdo. Roadrunner: automatic data extraction from data-intensive web sites. In *SIGMOD*, 2002.
23. M. J. Cafarella, D. Downey, A.-M. Popescu, T. Shaked, S. Soderland, D. S. Weld, and A. Yates. Unsupervised named-entity extraction from the Web: An experimental study. *Artificial Intelligence*, 165(1):91–134, 2005.
24. T. Furche, G. Gottlob, G. Grasso, O. Gunes, X. Guo, A. Kravchenko, G. Orsi, C. Schallhart, A. Sellers, and C. Wang. DIADEM: Domain-centric, Intelligent, Automated Data Extraction Methodology. In *WWW*, 2012.
25. T. Furche, G. Gottlob, G. Grasso, C. Schallhart, and A. Sellers. Oxpath: A language for scalable, memory-efficient data extraction from web applications. In *PVLDB*, 4(11):1016-1027, 2011 2011.
26. G. Gottlob, C. Koch, and R. Pichler. Efficient Algorithms for Processing XPATH Queries. *TODS*, 2005.
27. D. Gruhl, L. Chavet, D. Gibson, J. Meyer, P. Pattanayak, A. Tomkins, and J. Zien. How to build a webfountain: An architecture for very large-scale text analytics. *IBM Syst. J.*, 43:64–77, January 2004.
28. B. He, M. Patel, Z. Zhang, and K. C.-C. Chang. Accessing the deep web. *Commun. ACM*, 50(5):94–101, 2007.
29. A. Heydon and M. Najork. Mercator: A scalable, extensible web crawler. *World Wide Web*, 2(4):219–229, 1999.
30. J. Kranzdorf, A. Sellers, G. Grasso, C. Schallhart, and T. Furche. Spotting the tracks on the oxpath. In *WWW*, 2012.
31. G. Leshed, E. M. Haber, T. Matthews, and T. Lau. Coscripter: automating & sharing how-to knowledge in the enterprise. In *CHI*, 2008.
32. J. Lin, J. Wong, J. Nichols, A. Cypher, and T. A. Lau. End-user programming of mashups with vegemite. In *IUI*, 2009.
33. L. Liu, C. Pu, and W. Han. Xwrap: An xml-enabled wrapper construction system for web information sources. In *ICDE*, 2000.
34. M. Liu and T. W. Ling. A rule-based query language for html. In *DASFAA*, 2001.
35. M. Marx. Conditional XPath. *ACM Trans. Database Syst.*, 30(4):929–959, 2005.
36. M. Marx and M. de Rijke. Semantic Characterizations of Navigational XPATH. *ACM SIGMOD Record*, 2005.
37. A. O. Mendelzon, G. A. Mihaila, and T. Milo. Querying the world wide web. *Int. J. on Digital Libraries*, 1(1):54–67, 1997.
38. S. Mir, S. Staab, and I. Rojas. Web-Prospector – An Automatic, Site-Wide Wrapper Induction Approach for Scientific Deep-Web Databases. In *BTW*, 2009.
39. P. Montoto, A. Pan, J. Raposo, F. Bellas, and J. López. Automating navigation sequences in ajax websites. In *ICWE*, 2009.
40. J. Myllymaki. Effective web data extraction with standard xml technologies. *Computer Networks*, 39(5):635 – 644, 2002.
41. D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking Forward. In *EDBT-XML-Based Data Management*, LNCS 2490, 2002.
42. J. Raposo, A. Pan, M. Álvarez, J. Hidalgo, and A. Viña. The wargo system: Semi-automatic wrapper generation in presence of complex data access modes. In *DEXA*, 2002.
43. A. Safonov. Web macros by example: users managing the www of applications. In *CHI*, pages 71–72, 1999. ACM.
44. A. Sahuguet and F. Azavant. Building light-weight wrappers for legacy web data-sources using w4f. In *VLDB*, pages 738–741, 1999.
45. N. Sawa, A. Morishima, S. Sugimoto, and H. Kitagawa. Wraplet: Wrapping your web contents with a lightweight language. In *SITIS*, pages 387–394, 2007.
46. W. Shen, A. Doan, J. F. Naughton, and R. Ramakrishnan. Declarative information extraction using datalog with embedded extraction predicates. In *VLDB*, 2007.
47. J.-Y. Su, D.-J. Sun, I.-C. Wu, and L.-P. Chen. On design of browser-oriented data extraction system and plug-ins. *J. of Marine Science and Tech.*, 18(2):189–200, 2010.
48. Y. Wang and T. Hornung. Deep web navigation by example. *Scalable Computing: Practice and Experience*, 9:281–292, 2008.