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

The Ontological Key: Automatically Understanding and Integrating Forms to Access the Deep Web

Tim Furche · Georg Gottlob · Giovanni Grasso · Xiaonan Guo · Giorgio Orsi · Christian Schallhart

25 Sep 2012

Abstract Forms are our gates to the web. They enable us to access the deep content of web sites. Automatic form understanding provides applications, ranging from crawlers over meta-search engines to service integrators, with a key to this content. Yet, it has received little attention other than as component in specific applications such as crawlers or meta-search engines. No comprehensive approach to form understanding exists, let alone one that produces rich models for semantic services or integration with linked open data.

In this paper, we present OPAL, the first comprehensive approach to form understanding and integration. We identify form labeling and form interpretation as the two main tasks involved in form understanding. On both problems OPAL advances the state of the art: For form labeling, it combines features from the text, structure, and visual rendering of a web page. In extensive experiments on the ICQ and TEL-8 benchmarks and a set of 200 modern web forms OPAL outperforms previous approaches for form labeling by a significant margin. For form interpretation, OPAL uses a schema (or ontology) of forms in a given domain. Thanks to this domain schema, it is able to produce nearly perfect ($> 97\%$ accuracy in the evaluation domains) form interpretations. Yet, the effort to produce a domain schema is very low, as we provide a Datalog-based template language that eases the specification of such schemata and a methodology for deriving a domain schema largely automatically from an existing domain ontology. We demonstrate the value of OPAL's form interpretations through a light-weight form integration system that successfully translates and distributes master queries to hundreds of forms with no error, yet is implemented with only a handful translation rules.

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

1 Introduction

Unlocking the vast amount of data in the deep web for automatic processing has been a central part of "web as a database" visions in the past. The web offers unprecedented choice and variety of products, but we lack tools to make this wealth of offers easily manageable. Say you are looking for a flat. *Aren't you tired of filling registration forms with your search criteria on the websites of hundreds of local agencies? You fear to miss the site with the very best offer? Wouldn't you wish to automatize these tiresome tasks?* To unlock this data for automatic processing requires two keys: a key that allows us through the human-centric, scripted form interfaces of the web and a key to identify offers among all the other data on the web. In this paper, we focus on the former: A key to web forms, the gates to the deep web. Since these gates are designed for human admission, they pose a plethora of challenges for automatic processing: Even web forms within a single domain denote search criteria differently, e.g., "address", "city", "town", and "neighborhood" all refer to locations, while other terms denote different criteria ambiguously, e.g., "tenure" might refer to the choice either between "freehold" vs. "leasehold" or between "buy" vs. "rent". Moreover, web forms present their criteria in different manners, e.g., for a choice among several options, a form may contain either a drop-down lists or a set of check boxes. Automatically understanding these variants to pass through forms is needed by a broad range of applications: crawling and surfacing the deep web [28, 21, 8], interface and service integration [36], matching interfaces across domains [7, 33], classifying the domain of web databases [4] for web site classification, sampling the contents of web databases [22, 2], ontology enrichment and knowledge-base construction [26], question answering for the deep web [20]. In web engineering, automated form understanding contributes, e.g., to web accessibility and us-

ability [17], web source integration [11], automated testing on form-related web applications.

The form understanding problem has attracted a number of approaches [36,33,11,24,18], for a recent survey see [19, 12]. These approaches turn observations on common features of web forms (in general, across domains) into specifically tailored algorithms and heuristics, but generally suffer from three major limitations:

(1) Most approaches are *domain independent* and thus limited to observations that hold for forms across all domains. This limitation is acknowledged in [36,24,18], but addressed only through domain specific training data, if at all. Our evaluation supports [18] in that a set of generic design rules underlies all domains, but that specific domains parameterise or adapt these in unique ways.

(2) Most approaches are limited in the *classes of features* they use in their heuristics and often based on a single sophisticated heuristics using one class of features, e.g., only visual features [11] or textual and field type features [18].

(3) Heuristics are translated into monolithic algorithms limiting maintainability and adaptability. E.g., [33] and [24] encode assumptions on the spatial distance and alignment of fields and labels, [18] employs hard-coded token classes for certain concepts such as "min", "from" vs. "max", "to".

To overcome these limitations, we present OPAL (**o**ntology based web **p**attern **a**nalysis with **l**ogic), a domain-aware form understanding system that does not limit its scope to one class of features, but rather combines visual, textual, and structural features with a thin layer of domain knowledge. The visual, textual, and structural features are combined in a domain-independent multi-scope analysis to produce a highly accurate form labeling. However, for most applications a model of the form is needed, where all the fields are typed consistently with types from a (reference) schema of the given domain.

In OPAL, this schema is not only used to classify the fields and segments of the form model, but also to improve the form model based on a set of constraints that describe typical fields and their arrangement in forms of the domain, e.g., how price ranges are presented in forms. To ease the development of these domain ontologies, OPAL extends Datalog with templates to enable reuse of common form patterns in forms, e.g., how ranges (of any type) are presented in forms. With this approach, OPAL achieves nearly perfect analysis results ($>97\%$ accuracy). The combination of these ontologies with the use of datalog rules throughout OPAL also ease maintenance and adaptation to new domains or changing patterns in the web.

In contrast to previous approaches, OPAL produces rich form models, typed to the given schema: The models contain not only types and (individual) constraints for form fields, but also group those fields into semantic segments, possibly with inter-field constraints. These rich models ease the development of applications that interact with these forms. To demonstrate this, we have developed a light-weight form integration system on top of OPAL that fully automatically translates queries to the reference schema into queries to the concrete forms.

1.1 Contributions

OPAL's main contributions are:

(1) *Multi-scope domain-independent analysis* (Section 4) that combines structural, textual, and visual features to associate labels with fields into a form labeling using three sequential "scopes" increasing the size of the neighbourhood from a subtree to everything visually to the left and top of a field. We exploit (i) at *field* scope the structure of the page between fields and labels; (ii) at *segment* scope observations on fields in groups of similar fields, and (iii) at *layout* scope the relative position of fields and texts in the visual rendering of the page. We impose a strict preference on these scopes to disambiguate competing labelings and to reduce the number of fields considered in later scopes.

(2) *Domain awareness.* (Section 5) OPAL is domain-aware while being as domain-independent as possible without sacrificing accuracy. This is based on the observation that generic rules contribute significantly to form understanding, but nearly perfect accuracy is only achievable through an additional layer of domain knowledge. To this end, we add an optional, domain-dependent classification and form model repair stage after the domain-independent analysis. Driven by a domain schema, OPAL classifies form fields based on textual annotations of their labels and values assigned in the domain-independent form labeling, as well as the structure of that form labeling. This classification is often imperfect due to missing or misunderstood labels. OPAL addresses this in a repair step, where segment constraints are used to disambiguate and complete the classification and reshape the form segmentation.

(3) *Template Language* OPAL-TL. (Section 5.1) To specify a domain schema, we introduce OPAL-TL. It extends Datalog to express common patterns as parameterizable templates, e.g., describing a group consisting of a minimum and maximum field for some domain type. Together with some convenience features for querying the field labeling and its annotations, OPAL-TL allows for very compact, declarative specification of domain schemata. We also provide a template library of common phenomena, such that the adaption to new domains often requires only instantiating these templates with domain specific types. OPAL-TL preserves the complexity of Datalog.

(4) *Methodology for Deriving Domain Schemata.* (Section 6) To ease the derivation of an OPAL domain schema, we present a simple, step-by-step methodology how to de-

(a) Full search form (highlighting added) (b) Labeling 1: Field labels (c) Interpretation

Fig. 1: *Colin Mason* with OPAL (see Figure 2 for segment scope)

rive such a schema from a standard domain ontology (hierarchy of types and properties and a part-of relations), external knowledge bases for instances of the types, and observations on typical form configurations in the domain. OPAL's methodology exploits the fact that properties (such as price or mileage of a car) in the domain ontology often determine the configuration of the corresponding form fields.

(5) *Light-weight Form Integration.* (Section 7) To demonstrate the value of OPAL's rich form models, we implement a form integration system on top of OPAL that automatically translates a master query to concrete forms. As shown in the evaluation on 200 forms in two domains, even rather simple translation rules achieve accurate form filling.

(6) *Extensive Evaluation.* (Section 8) In an evaluation on over 700 forms of four different datasets, we show that OPAL achieves highly accurate ($> 95\%$) form labeling and, with a suitable domain schema, near perfect accuracy in form classification ($> 97\%$). To compare with existing approaches (which only report form labeling), we show that OPAL's domain-independent analysis achieves 94–100% accuracy on the ICQ benchmark and 92–97% on TEL-8. Thus, even without domain knowledge OPAL outperforms existing approaches by at least 5%. We also show that the form integration system developed on top of OPAL is able to fill forms correctly in nearly all cases ($> 93\%$)

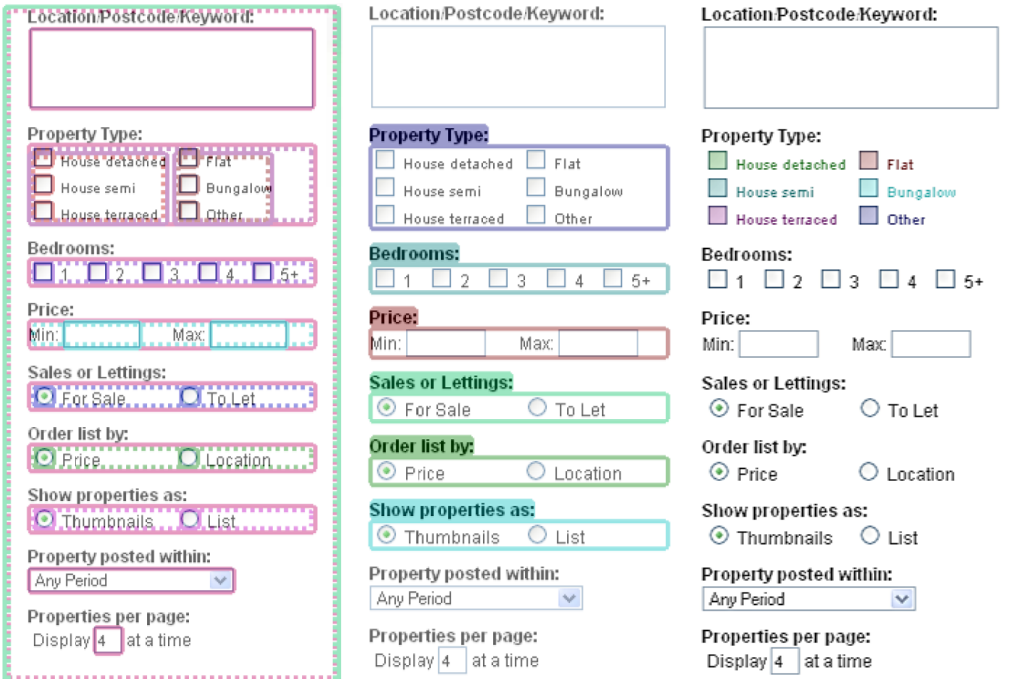
We believe that OPAL offers a comprehensive solution to form understanding for most applications, but also discuss, in Section 10, the two major remaining challenges for OPAL (and form understanding, in general): highly scripted, interactive forms, possibly including customised form widgets, as well as richer integrity constraints and access restrictions, in particular for applications that aim to extract all of the data behind a form.

This paper is based on [13], but has been significantly extended in every part.

1.2 OPAL: A Walkthrough

We present the OPAL approach to form understanding using the form from the UK real estate agency Colin Mason (`cmea.co.uk/properties.asp`). Figure 1a shows the web page with its simplified CSS box model. The page contains two forms (center and left): one for detailed search and the other for quick search. OPAL is able to identify, separate, label, and classify both forms correctly yielding two (real-estate) form models. The following discussion focuses on the search form in the center of Figure 1a, in which each of the components (1)-(10), each of the fields (3)-(7) and the two columns of checkboxes in (2) are enclosed in a `table`, `tr`, or `td` element. Labels for each of the components such as "Bedrooms:" appear in separate `tr`'s.

OPAL's form understanding operates in two parts: Form labeling and form interpretation. In form labeling fields and groups of fields (called segments) are arranged in a tree and assigned text labels. These nodes directly correspond to fields or other nodes in the DOM tree. In the form interpretation phase the text labels are used to classify the fields



(a) Segmentation (b) Labeling 2a: Segments (c) Labeling 2b: Fields by Segment

Fig. 2: *Colin Mason* segment scope

and segments on the page, eventually verifying and repairing the label assignment and producing a form model in line with the given domain schema. Form labeling itself is split into field, segment, and layout scope, each assigning successively labels to more fields and segments of a form.

Field scope. (Section 4.1) OPAL starts by analysing individual fields assigning labels in two ways: First, we add labels that explicit reference the field (using the `for` attribute). Second, we assign text nodes as labels of a field if the common ancestor of the text nodes and the field in the DOM tree has no other fields as descendant. In our example from Figure 1a, no explicit references occur, but the second approach correctly labels all fields except the checkboxes in (2). In Figure 1b we show this initial form labeling using same color for fields and their labels.

Segment scope. (Section 4.2) In segment scope, we increase the scope of the analysis from form fields to groups of similar fields (called *segments*). OPAL constructs these segments from the HTML structure, but eliminates segments that likely have no semantic relevance and are only introduced, e.g., for formatting reasons. This elimination is primarily based on semantic similarity between contained fields approximated via semantic attributes such as `class` and visual similarity. In our example, components (2)-(7) become segments, with (2) further divided into two segments for each of the vertical checkbox groups, as shown in Figure 2a. This rough, approximate segmentation may later be corrected in the form interpretation.

For each *segment as a whole*, OPAL associates text nodes to create segment labels. Segment labels can be useful to repair the form model and to classify fields that have no labels otherwise. In this example, OPAL assigns the text in bold face appearing atop each segment as the label, e.g., “Price:” becomes the label for (4), see Figure 2b. Furthermore, *within* each segment, OPAL identifies repeated groups of interleaving fields and texts. In the example, each check box in (2) is labeled with the text appearing after it, cf. Figure 2c.

Layout scope. (Section 4.3) In the layout scope, OPAL further enlarges the scope of the analysis to all nodes visually to the left and above a field. The primary challenge in this scope is “overshadowing”, i.e., if other fields appear in the quadrants to the left and above a field. In this example the layout scope is not needed.

The result of the layout scope is a form labeling. Notice, that this form labeling is entirely domain independent.

Domain scope. If a form model is required, the final step in OPAL produces a *form model* that is consistent with a given domain schema. How to derive such a domain schema and the necessary annotators is discussed in Section 6. It uses domain knowledge to classify and repair the labeling and segmentation from the form labeling. In the classification step, OPAL annotates fields and segments with types based on annotations of the text labels. The verification step

repairs and verifies the domain model if needed. For both steps, OPAL uses constraints specified in OPAL-TL. These constraints model typical representations of types in a domain. E.g., the first field in (4) is classified as MIN_PRICE as we recognise this segment as an instance of a price range template. These constraints also disambiguate between multiple, conflicting annotations, e.g., the first field in (6) is annotated with *order_by* and *price*, but the *price* annotation is disregarded due to the group label. Even without the group label, *price* would be disregarded as the domain schema gives precedence to *order_by* over *price* due to the observation that if they occur together, the field is likely about "order by price" and not about actual prices. Finally, a single repair is performed in this case: We collapse the two checkbox segments in (2) as they are the only children of their parent segment and both of the same type. Figure 1c shows the final field classification as produced by OPAL.

Form integration and filling. Using the form interpretation constructed in the preceding stages, OPAL is able to map a master query formulated on the domain schema into both of the concrete forms on this page (see Figure 1a). For location, the values are typed in directly. For price, the range in the master query can also be directly entered, as the concrete forms use text inputs for prices and OPAL's form interpretation identifies the min and max price field successfully. For the bedroom number, the value from the master query is compared with the members of the check box list and the most similar is selected.

2 Problem Definition

Form understanding constructs a model of a form consistent with a *domain schema.* A domain schema describes how forms in a given conceptual domain, such as the UK real estate domain, are structured. Form understanding can be divided into *form labeling* and *form interpretation.* The form labeling identifies forms and their fields, arranges the fields into a tree, and labels the found fields, segments, and forms with text nodes from the page. The form interpretation aligns a form labeling with the given domain schema and thereby classifies the form fields based on their labels.

Finally, we define the *form integration* problem where a query on the domain schema is translated into queries against the individual forms using the above form interpretation as basis for the translation.

Form Labeling. In general, it is impossible to define which form labeling is *suitable* for a given page, as even humans cannot always unambiguously group form elements into segments and associate them with labels. Hence, the suitability of a form labeling F for a given page P needs to be evaluated by human annotators (which our approach aims to simulate). Our evaluation (Section 8) shows that OPAL produces form labelings that match the gold standard in nearly all cases ($> 95\%$ without using any domain knowledge). We define the form labeling problem to produce a labeling which is *consistent with the page structure and a human provided gold standard.*

A **web page** $P = \big((U)_{U\in\mathsf{Unary}}, \mathsf{child}, \mathsf{next\text{-}sibl}, \mathsf{attribute}\big)$ is a DOM tree where $(U)_{U\in\mathsf{Unary}}$ are unary type and label relations, child is the parent-child, next-sibl the direct next sibling, and attribute the attribute relation. Further XPath relations (such as descendant) are derived from these basic relations as usual [6]. U contains relations for types as in XPath (element, text, attribute, etc.) and three kinds of label relations, namely tag^t for tags of elements and attributes, text^l for text nodes containing string l, and box^b for elements with bounding box b in a canonical rendering of the page. For consistency with elements, we represent the value of an attribute as text child node of the attribute.

A node can be labeled with no, one, or many labels via $\mathfrak{La}$. The form labeling contains a representative (via $\mathfrak{Re}$) for each form. A representative contains all fields (and segments) of that form. This allows OPAL to distinguish multiple forms on a single page, even if no form element is present or multiple forms occur in a single form element.

Definition 1 A **form labeling** of a web page P is a tree F with functions $\mathfrak{Re}$ (representative) and $\mathfrak{La}$ (label). $\mathfrak{Re}$ is an injective function that maps leafs (FIELDS_F) in F to *form fields* and inner nodes (SEGMENTS_F) to *form segments*, each grouping a set of fields or subsegments. Every node n in F is also mapped to a set $\mathfrak{La}(n)$ of text nodes, the *labels* of n.

We use `child` (`descendant`, resp.) for the child (descendant) relation in F and extend document and sibling order from P to F: $\texttt{next}(X,Y)$ for $X,Y \in F$, if $\mathsf{following}(\mathfrak{Re}(X),\mathfrak{Re}(Y))$ and no other node in F occurs between X and Y in document order; $\texttt{adjacent}(X,Y)$, if $\mathsf{next\text{-}sibl}(\mathfrak{Re}(X),\mathfrak{Re}(Y))$ or vice versa. Finally, we abbreviate $\mathsf{text}^l(\mathfrak{Re}(X))$ and $\mathsf{tag}^t(\mathfrak{Re}(X))$ as $\texttt{"}l\texttt{"}(X)$ and $t(X)$.

Definition 2 For a web page P, the **form labeling problem** asks for a form labeling F where for each form f in P

(1) there is a node $r \in F$ such that $\mathfrak{Re}(r)$ is a suitable representative of f and

(2) for each field e in f, there exists a leaf node $n_e \in F$ with $\mathfrak{Re}(n_e) = e$ such that n_e is a descendant of r and $\mathfrak{La}(n_e)$ is a suitable label set for e.

(3) for each segment n_s in F, $\mathfrak{La}(n_s)$ is a suitable set of labels for the set of fields contained in n_s.

Form Interpretation. We formalize the notion of domain schema and introduce a form model as a form labeling extended with type information consistent with a given domain schema. A domain schema provides types for form elements and segments and imposes constraints on the assigned types.

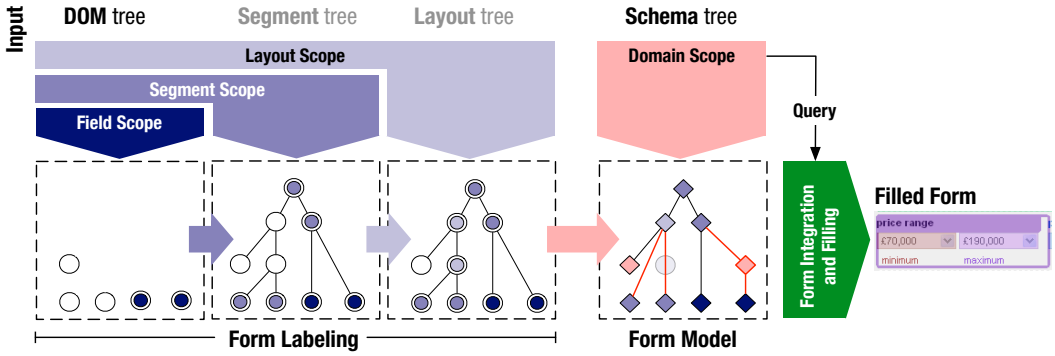


Fig. 3: OPAL Overview

Definition 3 A **domain schema** $\Sigma = (\mathcal{S}, \mathcal{F}, \multimap, \mathcal{C}_\mathcal{S}, \mathcal{C}_\mathcal{F})$ defines sets of segment and field types with (transitive, reflexive) part-of relation $\multimap : \mathcal{S} \cup \mathcal{F} \to \mathcal{S}$, and a mapping $\mathcal{C}_\mathcal{S}$ ($\mathcal{C}_\mathcal{F}$) to the segment (field) constraints for each type $t \in \mathcal{S}$ ($t \in \mathcal{F}$).

For example, the segment constraint $\mathcal{C}_\mathcal{S}(\text{PRICE-RANGE})$ for a PRICE-RANGE segment n requires n to contain a MIN-PRICE and MAX-PRICE field or a PRICE-RANGE field. We write $n \models c$ for a constraint c, if c holds for n under a fixed τ. For a *set* of constraints C, $n \models C$ if there exists a $c \in C$ such that $n \models c$. In particular, $n \models c \in \mathcal{C}_\mathcal{S}$ implies that n is a segment, $n \models c \in \mathcal{C}_\mathcal{F}$ that n is a field (no node can satisfy both a segment and a field constraint). Finally, we abbreviate $n \models c$ for all $c \in \mathcal{C}_\mathcal{S}(t) \cup \mathcal{C}_\mathcal{F}(t)$ of a type t, as $n \models t$. $\multimap$ plays an important role in the definition of the constraints, as it prescribes the structure of the types in the domain. For details on constraints and how to define them, see Section 5. Now, a *form interpretation* over domain schema Σ consists of a form labeling F with a partial type-of relation τ, relating nodes in F to types in $\mathcal{S} \cup \mathcal{F}$. If for all nodes in a form interpretation F the constraints for the types are satisfied and τ is total, then F is a form model.

Definition 4 A **form model** (F, τ) over a domain schema $\Sigma = (\mathcal{S}, \mathcal{F}, \multimap, \mathcal{C}_\mathcal{S}, \mathcal{C}_\mathcal{F})$ consists of a form labeling F and a total mapping τ from F to $2^{\mathcal{S} \cup \mathcal{F}}$ such that for all nodes $n \in F$ and all $t \in \tau(n)$, $n \models t$.

For example, Figure 1c is a form interpretation that is also a form model under the real-estate domain schema used in OPAL. It maps all fields to types (indicated by the red labels), such that all fields satisfy all constraints of the corresponding types. The complete form model additionally contains segments, e.g., for the price range.

To interpret a labeling F, we want to type all form representatives, fields, and labels in our interpretation. On the other hand, it is sometimes necessary to adapt the segmentation structure of F to fit the constraints induced by the typing. Similar to the case of form labelings, the *suitability* of the types in a form model cannot be defined formally, as some forms defy even human efforts to type them correctly. Hence, we refer to human judgment for assessing suitability assuming that suitability implies logical consistency (t suitable for n implies $n \models t$).

Definition 5 For a domain schema Σ and web page P, the **form interpretation problem** asks for a form model (F, τ) of P under Σ, such that **(1)** F is a solution for the form labeling problem on P, **(2)** (F, τ) is complete: for each node $n \in F$, $\mathfrak{La}(n)$ is the set of all suitable labels and $\tau(n)$ the set of all suitable types.

Form Integration and Filling. In web interface integration a query against a global domain schema is translated and executed on concrete forms. The data returned from the concrete site is translated into the domain schema and returned. We focus here on the first part of the integration problem, the query translation or *form integration* problem, and more specifically on its optimistic variant: Let Σ be a domain schema. Then a query Q on Σ is a set of unary constraints on $\mathcal{S} \cup \mathcal{F}$, the domain types in Σ. We consider three types of constraints: **(1)** *Equality constraints* such as POSTCODE $=$ OX1; **(2)** *range constraints* such as PRICE $\in [700, 1250]$; **(3)** *inclusion constraints* such as COLOUR $\in \{$red, green, black$\}$.

Definition 6 Given a domain schema Σ, a query Q on Σ, and a concrete form F, the **form integration problem** is the problem to translate Q into a (single) query Q' on F such that **(1)** Q' returns all results that match Q and can be retrieved by F and that **(2)** there is no other query on F with that property that returns less results.

We do not require Q' to return only results that match Q, but that the result set is minimal among all queries on F returning all matches for Q, since there may be no query on F exactly expressing Q.

3 OPAL Architecture

OPAL is divided into three parts. Of those, the first two represent OPAL's form understanding: a domain-independent

part to address the form labeling problem and a domain-dependent part for form interpretation according to a domain schema. The remaining part is devoted to form integration and translates queries to a domain schema into queries to concrete forms.

OPAL produces form labelings in a novel multi-scope approach that incrementally constructs a form labeling combining textual, structural, and visual features (Figure 3). Each of the three labeling scopes considers features not considered in prior scopes:

(1) In *field scope*, we consider only fields and their immediate neighbourhood using the DOM tree as only input.

(2) In *segment scope*, we arrange form segments into a segment tree to interleave text nodes and fields.

(3) In *layout scope*, we search for labels in the layout tree, i.e., the visual rendering of the page, and assign text nodes to fields, given a strong visual relation.

Each scope builds on the partial form labeling of the previous scope and uses the information from the additional features to find labels for previously unlabeled fields (or segments). Only the segment scope adds nodes, namely form segments, whereas field and layout scope only add labels.

Finally, in the **(4)** *form interpretation* (Section 5) we turn the form labeling produced by the first three scopes into a form model consistent with a given domain schema. (i) The labeling model is extended with (domain-specific) annotations on the textual content of proper labels and values. (ii) Fields and segments of the form labeling are classified according to classification constraints in the domain schema. (iii) Finally, violations of structural schema constraints are repaired in a top-down fashion.

Types and constraints of the domain schema are specified using OPAL-TL, an extension of Datalog that combines easy querying of the form labeling and of annotations with a rich template system. Datalog rules already ease the reuse of common types and their constraints, but the template extension enables the formulation of generic templates for such types and constraints. These are then instantiated for concrete types of a domain. An example of a type template is the range template, that describes typical ways for specifying range values in forms. In the real estate domain it is instantiated, e.g., for price and various room ranges. In the used car domain, we also find ranges for engine size, mileage, etc. Thus, creating a domain schema is in many cases as easy as importing common types and instantiating templates, see Section 6.

The form understanding part of OPAL is complemented with a form integration to translate a given query on the domain schema into queries on concrete forms. To do so, we construct an OPAL form model to map the constraints of the given query to fields and fillings on the concrete form. If a constraint cannot be mapped precisely, we find with standard similarity techniques the closest, inclusive option (in case of numerical types) or just the closest option (in case of categorial types), see Section 7.

Algorithm 1: `FieldScopeLabelling`(*DOM P*)

```
1  foreach field f in P do
2      n ← f;
3      while n has a parent do
4          if n is already coloured then colour n red; break;
5          colour n orange;
6          n ← parent of n;
7  F ← empty form labeling ;
8  foreach field f in P do
9      n ← new leaf node in F;
10     ℜe(n) ← f;
11     if ∃l ∈ P with for attribute referencing f then
12         assign all text node descendants of l as labels to n ;
13     p ← parent of f;
14     while p not coloured red do
15         f ← p; p ← parent of f;
16     assign all text node descendants of f as labels to n ;
```

4 Form Labeling

In OPAL, form labeling is split into three scopes. Each scope is focused on a particular class of features (e.g., visual, structural, textual). The form labeling scopes, *field*, *segment*, and *layout* scope, use **domain-independent** labeling techniques to associate form fields or segments with textual labels, building a form labeling F. If a domain schema is available, the form labeling is extended to a form model in the domain-dependent analysis (Section 5).

The form labeling F is constructed bottom-up, applying each scope's technique in sequence to yet unlabelled fields. Whenever a field is labelled at a certain scope level, further scopes do not consider this field again. This precedence order reflects higher confidence in earlier scopes and addresses competing label assignments.

4.1 Field Scope

Based on the DOM tree of the input page, the **field scope** assigns text nodes in a unique structural relation to individual fields as labels to these fields (see Algorithm 1). It relies on the observation that, if a text node shares a sub-tree of the DOM with a single field only, then that text node is most likely related to that field. This simple observation produces a significant portion of form labels, as shown in Section 8, and is designed to produce nearly no false positives, as also verified in Section 8 (Table 1).

Specifically, Algorithm 1 **(1)** colours (lines 1–6) all nodes in P that are ancestors of a field and do not have other form fields as descendants in orange. The least ancestor that violates that condition is coloured red. **(2)** It identifies (line 7–10) all form fields and initialises the form labeling F with

Algorithm 2: `SegmentTree`$(DOM\ P)$

```
1  P' ← P;
2  while ∃n ∈ SEGMENTS'_P : (∄ field d : descendant(d,n) ∈ P') do
3      delete n and all incident edges from P';
4  while ∃n ∈ P' : |{c ∈ P' : child(c,n) ∈ P'}| = 1 do
5      delete n from P' and move its child to the parent of n;
6  foreach inner node n in P' in bottom-up order do
7      C ← {f : child(f,n) ∈ P' ∧ f is a field};
8      C ← C ∪ {Representative(n') : child(n',n) ∈ P'} ;
9      choose r ∈ C arbitrarily ;
10     if ∀r' ∈ C : r style-equivalent to r' then
11         Representative(n) ← r;
12         delete n's segment children moving their children to n;
13     else Representative(n) ← ⊥ ;
14 return P';
```

one leaf node for each such field. **(3)** It considers (lines 11–12) explicit HTML `label` elements with *direct reference* to a form field. **(4)** It labels (lines 13–16) each field f with all text nodes t whose *least common ancestor* with f has no other form field as descendant. This includes all text nodes t in the content of f such as its values (in case of `select`, `input`, or `textarea` elements), since the least common ancestor of t and f is f itself.

4.2 Segment Scope

At **segment scope**, the labeling analysis expands from individual fields to form segments, i.e., groups of consecutive fields with a common parent, forming the segment tree (Algorithm 2). These segments are then used to distribute text nodes to unlabeled fields in that segment (Algorithm 3). At this scope, we approximate form segments through the DOM structure and the style of the contained fields. This segmentation is later adjusted to yield only form segments with a clear semantic. It is worth noting, that on many forms only very few adjustments are necessary, supporting the veracity of the approximation of semantic segments through structure and style.

Segmentation tree. We observe that the DOM is often a fair, but noisy approximation of the semantic form structure, as it reflects the way the form author grouped fields into segments. Therefore, we start from the DOM structure to find the form segments, but we eliminate all nodes that can be safely identified as superfluous: nodes without field descendants, nodes with only one child, and nodes n where all fields in n are style-equivalent to the fields in the siblings of n. Two fields are **style-equivalent** if they carry the same `class` attribute (indicating a formatting or semantic class) or the same `type` attribute and CSS `style` information.

If all field descendants of the parent of an inner node n are style-equivalent, then n should be eliminated from the segment tree, as it artificially breaks up the sequence of style-equivalent fields and is thus *equivalence breaking*.

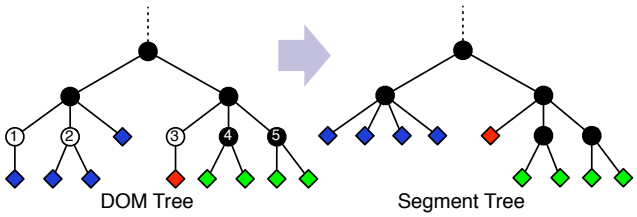


Fig. 4: Example DOM and Segment Tree

Definition 7 The **segment tree** P' of a form page P is the maximal DOM tree included in P (i.e., obtained by collapsing nodes) such that the leaves of P' are all fields and, for all its inner nodes n,
(1) $\left|\{c \in P' : \mathsf{child}(c,n)\}\right| > 1$,
(2) n is not equivalence breaking.

As an example, consider the DOM tree on the left of Figure 4, where diamonds represent fields and style-equivalent fields carry the same colour. On the right hand side, we show OPAL's segment tree for that DOM. Nodes 1 and 3 from the original DOM are eliminated as they have only one child, and node 2 as it is equivalence breaking. Nodes 4 and 5 are retained due to the red field.

Algorithm 2 computes the segment tree P' for any DOM tree P. Its leafs are fields (as any non field leafs are eliminated in line 2–3) and any inner node must have more than one child (due to line 4–5), a field descendant (due to line 2–3), and not be equivalence breaking (due to lines 6–13). In lines 6–13, we compute a Representative, bearing the style prevalent among the inner node's fields, for each inner node in a bottom-up fashion: If all field children (line 7) and the representatives of all inner children (line 8) are style-equivalent (line 9–10), we choose an arbitrary representative and collapse all inner children of that node. Note, that it suffices to compare any of the representatives with the fields in C as style-equivalence is transitive. Otherwise, we assign $\perp$ as representative, which is style-equivalent neither to any node nor to itself. Thus it prevents this node (and its ancestors) from ever being collapsed. By construction, these nodes respect (1) and (2) and this property is retained in all later steps, as their subtrees are never touched again.

P' is maximal: Any tree P'' that includes P' but is included in P must contain at least one node from P that has been deleted by one of the above conditions. Such a node, however, violates at least one of the conditions for a segment tree and thus P'' is not a segment tree. This holds because the order of the node deletions does not affect the nodes deleted.

Segment Labeling. We extend the existing form labeling F of the field scope with form segments according to the structure of the segment tree and distribute labels in regular groups, see Algorithm 3. First (lines 2–5), we create a form segment node s in the form labeling for each inner

Algorithm 3: SegmentLabeling(*DOM P, Form Labeling F*)

```
1  S ← SegmentTree(P) ;
2  foreach inner node s in S in bottom-up order do
3      create a new segment n_s in F;
4      ℜe(n_s) ← s;
5      create an edge (n_s, c_s) in F for every ℜe(c_s) child of s;
6  foreach segment n in F do
7      Nodes, Labels ← new List();
8      textG ← ∅ ;
9      foreach c : R_descendant(c, ℜe(n)) ∈ P in document order do
10         if ∃f ∈ F : ℜe(f) = c ∧ 𝔏a(f) = ∅ then
11             if textG ≠ ∅ then Labels.add(textG); textG ← ∅;
12             Nodes.add(c);
13             skip all descendants of c in the iteration ;
14         else if c is a text node ∧ ∄d ∈ F : c ∈ 𝔏a(d) then
15             textG ← textG ∪ {c};
16     if textG ≠ ∅ then Labels.add(textG); textG ← ∅;
17     if Labels.size() = Nodes.size() + 1 then
18         add Labels[0] to 𝔏a(n);
19         delete Labels[0] from Labels;
20     if Labels.size() = Nodes.size() then
21         foreach i do add Labels[i] to 𝔏a(Nodes[i]);
```

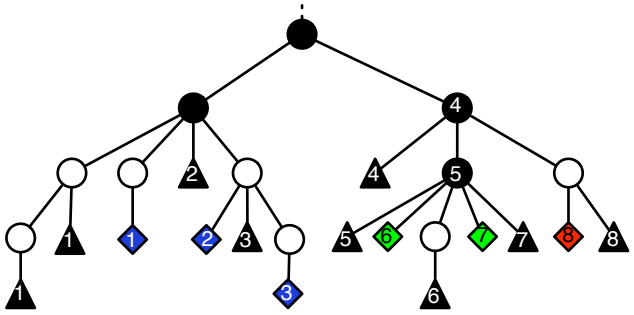


Fig. 5: Example for Segment Scope Labeling

node n_s in the segment tree and choose n_s as representative for s ($\mathfrak{Re}(s) = n_s$). For each segment with regular interleaving of text nodes with field or segment nodes, we use those text nodes as labels for these nodes, preserving any already assigned labels and fields (from field scope). In detail, we iterate over all descendants c of each segment in document order, skipping any nodes that are descendants of another segment or field itself contained in n (line 13). In the iteration, we collect all field or segment nodes in Nodes, and all sets of text nodes between field or segment nodes in Labels, except those already assigned in field scope (line 14), as we assume that these are outliers in the regular structure of the segment. We assign the i-th text node group to the i-th field, if the two lists have the same size (possibly using the first group as labels of the segment, line 17–19).

Figure 5 illustrates the segment scope labeling with triangles standing for text nodes, diamonds for fields, black circles for segments, and white circles for DOM nodes not in the segment tree. The numbers indicate which text nodes are assigned as labels to which segments or fields. E.g., for the left hand segment, we observe a regular structure of (text node+, field)+ and thus we assign the i-th group of text nodes to the i-th field. For the right hand segment (4), we find a subsegment (5) and field 8 that is already labeled with text node 8 in the field scope. Thus 8 is ignored and only one text node remains directly in 4, which becomes the segment label. In 5, we find one more text node group than fields and thus consider the first text node group as a segment label. The remaining nodes have a regular structure (field, text node+)+ and get assigned accordingly.

4.3 Layout Scope

At **layout scope**, we further refine the form labeling for each form field not yet labelled in field or segment scope, by exploring the visible text nodes in the west, north-west, or north quadrant, if they are not overshadowed by any other field. To avoid false positives, we limit this search to the boundaries of the enclosing form. First, OPAL constructs a layout tree from the CSS box labels of the DOM nodes:

Definition 8 The **layout tree** of a given DOM P is a tuple $(N_P, \lhd, \mathsf{w}, \mathsf{nw}, \mathsf{n}, \mathsf{ne}, \mathsf{e}, \mathsf{se}, \mathsf{s}, \mathsf{sw}, \mathsf{aligned})$ where N_P is the set of DOM nodes from P, $\lhd, \mathsf{w}, \mathsf{nw}, \mathsf{n}, \ldots$ the "belongs to" (containment), west, north-west, north, ... relations from RCR [23], and $\mathsf{aligned}(x,y)$ holds if x and y have the same height and are horizontally aligned.

We call $\mathsf{w}, \mathsf{nw}, \ldots$ the neighbour relations. For convenience, we write, e.g., w-nw-n to denote the union of the relations w, nw, and n.

In cultures with left-to-right reading direction, we observe a strong preference for placing labels in the w-nw-n region from a field. However, forms often have many fields interspersed with field labels and segment labels. Thus we have to carefully consider overshadowing. Intuitively, for a field f, a visible text node t is overshadowed by another field f' if t is above f' or also visible from, but closer to f'. In the particular case of aligned fields, the former would prevent any labeling for these fields and thus we relax the condition.

Definition 9 For a given text node t, a field f' **overshadows** another field f if

(1) f and f' are unaligned, $\mathsf{w\text{-}nw\text{-}n}(f', f)$, and $\mathsf{w\text{-}nw\text{-}n\text{-}ne\text{-}e}(t, f')$ or

(2) f and f' are aligned and (i) $\mathsf{w}(t, f')$ or (ii) $\mathsf{nw\text{-}n}(t, f')$ and there is a text node t' not overshadowed by another field with $\mathsf{ne\text{-}e}(t', f')$ and $\mathsf{w\text{-}nw\text{-}n}(t', f)$.

To illustrate this overshadowing, consider the example in Figure 6. For field F_1, T_2 and T_4 are overshadowed by F_2 and T_3 by F_3, only T_1 is not overshadowed, as there is no other text node that is west, north-west, or north from F_1 and not overshadowed by another field.

The layout scope labeling is then produced as follows: For each field f, we collect all text nodes t with $\mathsf{w\text{-}nw\text{-}n}(t, f)$

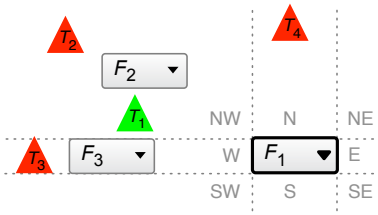


Fig. 6: Layout Scope Labeling

and add them as labels to f if they are not overshadowed by another field and not contained in a segment that is no ancestor of f. The latter prevents assignment of labels from unrelated form segments.

4.4 Form labeling complexity

The complexity of the form labelling task is the maximum among the complexities of field-scope, segment-scope and layout-scope labelling tasks.

Theorem 1 *Given a page P, a form-labeling F for P can be computed in polynomial time in the size of P.*

Proof Given a DOM P with n nodes of depth d, we prove the complexity of the three domain independent scopes individually: **(1)** *Field scope:* Algorithm 1 runs in time $\mathcal{O}(n^2)$, as both loops in Algorithm 1 iterate over the nodes of P and the loop bodies are linear in n. **(2)** *Segment scope:* In Algorithm 3, OPAL computes the segment tree in Line 1 and expands the labeling with the found segments in Lines 2–5, all running in time $\mathcal{O}(n \times d)$. Then, the propagation of segment labels (Lines 6–21) iterates over the descendants of each segment s (potentially all the remaining segments but s and its ancestors), thus leading to the final segment labelling after $\mathcal{O}(n^2)$ steps, as there are at most n segments. OPAL computes the segment tree with Algorithm 2: Lines 2–3 are in $\mathcal{O}(n)$. Lines 4–5 and Lines 6–13 are both in $\mathcal{O}(n \times d)$ as they are dominated by the collapsing of the nodes. At most, we collapse $d-2$ inner nodes and move $\mathcal{O}(n)$ leaves $d-2$ times. **(3)** *Layout scope:* The remaining labels can be computed in $\mathcal{O}(n^2 + t \times f^2) = \mathcal{O}(n^3)$ with t and f the number of text nodes and fields respectively in P. OPAL first computes all neighborhood relations and the layout tree in time $\mathcal{O}(n^2)$. Second, OPAL computes the overshadowing relation by inspecting the text nodes in P from west to east, each time checking the conditions of Definition 9 for each pair of form fields, yielding $\mathcal{O}(t \times f^2)$ in complexity.

5 Form Interpretation

There is no straightforward relationship between domain concepts, such as location or price, and the structure of their implementation on a form. Even seemingly domain-independent concepts, such as price, often exhibit domain specific peculiarities, such as "guide price", "current offers in excess", or payment periods in real estate. OPAL's domain schemata allow for covering these specifics. Recall from Section 2 that a form model (F', τ) for schema Σ is derived from a labeling F by extending F with types and restructuring it to satisfy the segment constraints of Σ.

OPAL performs form interpretation of a form labeling F in two steps: **(1)** the *classification* of fields and segments in F according to the domain types $\mathcal{F}$ and $\mathcal{S}$ to obtain a (partial) typing τ_P. This step relies on an *annotation schema* and its typing of fields in F based on the constraints in $\mathcal{C}_\mathcal{F}$; **(2)** the *model repair* where the segmentation structure derived in the segmentation scope (Section 4.2) is aligned with the segment constraints $\mathcal{C}_\mathcal{S}$ of Σ to complete the typing.

The effort for creating an OPAL domain schema may, at the first glance, appear considerable. However, not only do we provide OPAL-TL (Section 5.1) to ease the specification of a domain schema, we also discuss in Section 6 how all the artefacts needed by OPAL for a new domain can be nearly automatically derived from a standard ontology of a domain (including concept labels) and a set of entity recognisers (or annotators) for instances of the concepts. We illustrate this methodology for *domain instantiation* along the example of the used car domain.

Annotations. To bridge the gap from labeling to interpretation, OPAL annotates values (*"£200"*) and labels (*"price"*).

Definition 10 An **annotation schema** $\Lambda = \big(\mathcal{A}, \sqsubset, \prec, (\mathsf{isLabel}_a, \mathsf{isValue}_a : a \in \mathcal{A})\big)$ defines a set $\mathcal{A}$ of annotation types, a transitive, reflexive subclass relation $\sqsubset$, a transitive, irreflexive, antisymmetric precedence relation $\prec$, and two characteristic functions $\mathsf{isLabel}_a$ and $\mathsf{isValue}_a$ on text nodes for each $a \in \mathcal{A}$.

For example, for type *price* $\in \mathcal{A}$, $\mathsf{isLabel}_{price}$(*"Price:"*) and $\mathsf{isValue}_{price}$(*"more than £500"*) holds. The $\sqsubset$ relation describes subtypes, e.g., *postcode* $\sqsubset$ *location*, and the $\prec$ relation defines precedence on annotation types for disambiguating competing annotations. A select box offering *"Choose sorting order"*, *"By price"*, and *"By postcode"* may be annotated with *order-by*, *price*, and *postcode*. If *order-by* $\prec$ *price* and *order-by* $\prec$ *postcode*, OPAL picks *order-by*.

5.1 Schema Design: OPAL-TL

OPAL provides a **t**emplate **l**anguage, OPAL-TL, for easily specifying domain schemata reusing common concepts and their constraints as well as concept templates. To implement a new domain, we only need to provide **(1)** for each annotation type a an annotator implementing $\mathsf{isLabel}_a$ and $\mathsf{isValue}_a$ and **(2)** an OPAL-TL specification of the field and segment

types through their field and segment constraints. The latter can be derived almost mechanically from the domain types as discussed in Section 6.

Annotation types and their queries. Annotations (instances of annotation types) are characterised by an external specification of the characteristic functions $\mathsf{isLabel}_a$ and $\mathsf{isValue}_a$ for each $a \in \mathcal{A}$. In the current version of OPAL, these functions are either implemented with GATE (`gate.ac.uk`) gazetteers and transducers, that are provided by human domain experts, or realised by access to external annotators and knowledge bases such as DBPedia and Freebase. Together they provide annotators for common domain types such as price, location, or date. Additional entity recognisers or annotators can be added easily, as described in Section 6.

Annotation queries select fields based on annotations which are associated with their own labels or with labels on their enclosing segments. Intuitively, an annotation query $X@A$ returns all fields labeled with a text node that is annotated with A. If the modifier d (direct) is *not* present, we also consider immediate segment parents, otherwise only *direct* labels are considered (this defines the allowed labels for modifier μ, $\mathsf{Allowed}_\mu$ in Definition 11). If the modifier p (*proper*) is present, only $\mathsf{isLabel}_A$ is used, otherwise also $\mathsf{isValue}_A$. Together with $\mathsf{Allowed}_\mu$, this establishes the full set of matching labels M_μ. If the modifier e (*exclusive*) is present, a node that fullfils all other conditions is still not returned, if there are more labels with annotations of a type that has precedence over A. If the modifier m (*maximal*) is present, no other type, regardless of precedence, may have more labels with annotations at the node. Since m excludes strictly more nodes than e, a query with both m and e returns the same nodes as that query without e. Nodes where labels for A do not match these conditions for modifier μ are collected in Block_μ.

Definition 11 For a form labeling F on a DOM P and an annotation schema Λ with annotation types $\mathcal{A}$, an **OPAL-TL annotation query** is an expression of the form $X@A\{d,p,e,m\}$ where X is a first-order variable, $A \in \mathcal{A}$, and d, p, e, and m are annotation *modifiers*. An annotation query $X@A\mu$ with $\mu \subseteq \{d,p,e,m\}$ holds for $X \in [\![A\mu]\!]$ with

$$[\![A\mu]\!] = \{n \in \text{FIELDS}_F : \mathsf{M}_\mu(A,n) \neq \emptyset\} \setminus \mathsf{Block}_\mu(A)$$

$$\mathsf{M}_\mu(A,n) = \begin{cases} \mathsf{Allowed}_\mu(n) \cap \bigcup_{A' \sqsubset^* A} \mathsf{isLabel}_{A'} & \text{if } p \in \mu \\ \mathsf{Allowed}_\mu(n) \cap \bigcup_{A' \sqsubset^* A} (\mathsf{isLabel}_{A'} \cup \mathsf{isValue}_{A'}) & \text{otherwise} \end{cases}$$

$$\mathsf{Block}_\mu(A) = \begin{cases} \{n : \exists A' \neq A : |\mathsf{M}_\mu(A,n)| < |\mathsf{M}_\mu(A',n)|\} & \text{if } m \in \mu \\ \{n : \exists A' \prec A : |\mathsf{M}_\mu(A,n)| < |\mathsf{M}_\mu(A',n)|\} & \text{if } e \in \mu \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathsf{Allowed}_\mu(n) = \begin{cases} \mathfrak{La}(n) & \text{if } d \in \mu \\ \mathfrak{La}(n) \cup \mathfrak{La}(\text{parent of } n) & \text{otherwise} \end{cases}$$

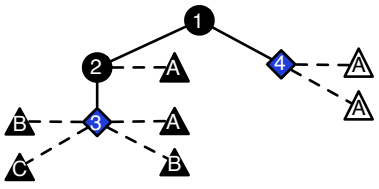


Fig. 7: Example Form Labeling

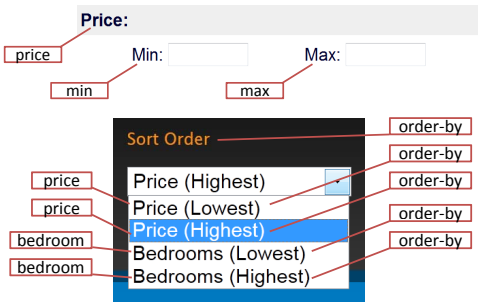


Fig. 8: Label Annotation Examples

Consider the form labeling of Figure 7 under a schema with $B \prec A$. Labels are denoted with triangles, fields with diamonds, segments with circles. Labels are further annotated with matching annotation types (here always only one), with value labels drawn as outlines only. Then, $X@A\{\}$ matches 3,4; $X@A\{e,d\}$ matches 4, but not 3 as 3 has more labels of B than of A and the exclusive modifier e is present; $X@A\{e,p\}$ matches 3, but not 4 as the proper modifier p prevents the value labels in white to be considered. The latter matches 3 despite the presence of e, as we consider also the labels of the parent of 3 (since the direct modifier d is absent) and thus there are two A labels.

Figure 8 shows two real-life example with the annotations produced by a typical set of annotators. In the upper part, there are two text inputs for min and max price. However, the two labels "min" and "max" are the only directly associated text boxes and do not carry any information that indicates that these fields are about prices. This is available only when considering the segment label "Price:". Thus, $X@\mathit{price}\{d\}$ returns the emptyset, but $X@\mathit{price}\{\}$ returns the two fields. In the lower part, the drop-down menu for result ordering receives two price annotations, two bedroom annotations, and five order-by annotations. With $\mathit{order\text{-}by} \prec \mathit{price}$, $X@\mathit{price}\{e\}$ returns the emptyset, as the price annotations are "blocked" by the order-by annotations.

OPAL-TL templates. OPAL-TL extends Datalog with templates and predefined predicates for querying annotations and DOM nodes. Relations from F and P (as introduced in Section 2) are made accessible to OPAL-TL in the obvious way. Examples of templates are basic classification rules defining a field type from a conjunction of annotation

queries, or templates capturing the relationship among multiple fields with related annotations, e.g., min-max ranges. In general, there are two types of such templates, one for field constraints, one for segment constraints. The former specify relationships between field and annotation types, the latter the structure of field and segment types.

Definition 12 An **OPAL-TL template** is an expression of the form: `TEMPLATE` $N\langle T_1,\ldots,T_k\rangle$ { $p_1 \Leftarrow expr_1.$ … } where N is the template name, $T_1,\ldots,T_k$ are template variables, p_1 is a template atom, $expr_1$ a boolean formula over template atoms and annotation queries. A *template atom* $p\langle\mathbf{t}\rangle(\mathbf{s})$ consists of a first-order predicate p, a sequence of terms $= t_1,\ldots,t_n\}$ (where t_i is either a constant or a template variable), and a sequence of terms $\mathbf{s} = s_1,\ldots,s_n$ where each s_i is either a constant or a first-order variable. Template and first-order variables constitute two disjoint sets. If $\mathbf{t}$ is empty, then a template atom is a normal first-order atom, and if all terms $\mathbf{t}$ are constants, the atom is called *template-ground*.

Multiple rules with the same head express disjunction of their bodies. For convenience, we use $\vee$ and $\neg$ over conjunctions, which are translated to Datalog$^\neg$ as usual.

As an example, the following template defines a family of constraints that associate the field type C to a field N whenever N is labeled by an exclusive direct and proper annotation of type A.

```
TEMPLATE basic_field<C,A>{ field<C>(N)⇐N@A{d,e,p} }
```

An *instantiation* of a template *tpl* produces a set of rules where the template variables $C_1,\ldots,C_k$ are assigned to values $v_1^i,\ldots,v_k^i$ defined by a *template instantiation* expression of the form:

```
INSTANTIATE tpl<T1,…,Tk> using {<v1^1,…,vk^1> … <v1^n,…,vk^n>}
```

For example, the following expression instantiates `basic_field` replacing C with type `RADIUS` and A with annotation type *radius*

```
INSTANTIATE basic_field<C,A> using {<RADIUS, radius>}
```

and produces the following instantiated rule:

```
field<RADIUS>(N)⇐N@radius{d,e,p}
```

The full syntax of OPAL-TL is given in Figure 9, with $\langle string\rangle$, $\langle id\rangle$, and $\langle var\rangle$ as in Datalog and $\langle tvar\rangle$, $\langle type\text{-}id\rangle$, $\langle annot\text{-}id\rangle$, $\langle tag\rangle$ template variables, domain types, annotation types, and HTML tags, respectively.

The semantics of OPAL-TL is given by rewriting a set of templates Σ_T into Datalog$^\neg$ programs, assigning template variables to constants as specified by the instantiation rules to consider every template-ground predicate name as new first-order predicate. Due to possible occurrences of `INSTANTIATE` within templates, the instantiation must be repeated until there are no applicable `INSTANTIATE` rules. To ensure termination of the instantiation procedure, we do not allow recursive template instantiations. Properties such as safety can be easily extended from Datalog$^\neg$ to OPAL-TL:

```
⟨program⟩  ::= (⟨template⟩ | ⟨inst⟩ | ⟨trule⟩ )+
⟨template⟩ ::= 'TEMPLATE' ⟨id⟩ '<' ⟨tvar⟩+ '>' '{' ⟨trule⟩+ '}'
⟨inst⟩     ::= 'INSTANTIATE' ⟨id⟩ '<' ⟨tvar⟩+ '>'
                 'using' '{' ('<' ⟨const⟩+ '>')+ '}'
⟨trule⟩    ::= ⟨tatom⟩ '←' ⟨tbody⟩ | ⟨inst⟩
⟨tbody⟩    ::= ⟨texpr⟩ (',' ⟨texpr⟩)*
⟨texpr⟩    ::= ⟨atom⟩ | ⟨annot⟩ | ⟨tatom⟩ | ⟨neg⟩ | ⟨disj⟩
⟨annot⟩    ::= ⟨var⟩'@' '{' ('d' | 'e' | 'p' | 'm')* '}'
⟨tatom⟩    ::= ⟨id⟩ '<' (⟨tvar⟩ | ⟨const⟩)+ '>' '(' ⟨par⟩* ')'
             | '<' ⟨tvar⟩ '>' '(' ⟨par⟩* ')'
⟨par⟩      ::= ⟨var⟩ | ⟨tvar⟩ | ⟨const⟩
⟨const⟩    ::= ⟨type-id⟩ | ⟨annot-id⟩ | ⟨tag⟩ | ⟨string⟩ | ⟨id⟩
⟨neg⟩      ::= '¬' '(' ⟨tbody⟩ ')'
⟨disj⟩     ::= '(' ⟨tbody⟩ '∨' ⟨tbody⟩ ')'
```

Fig. 9: OPAL-TL syntax

Definition 13 A OPAL-TL template is **safe**, if every template variable that occurs in the body also occurs in the head of the template and every rule is safe, i.e., all first-order variables that occur in the head or in a negative atom in the body, also occur in a positive atom in the body.

In contrast to safety, stratification depends on the instantiation and is defined over the expanded program as usual.

Proposition 1 *Let Σ_T be a set of safe* OPAL-TL *templates, and let $\mathcal{I}$ be a set of* OPAL-TL *instantiation rules, then any instantiation $\langle\Sigma_T,\mathcal{I}\rangle$ is a safe Datalog$^\neg$ program.*

Proof Let T be a safe OPAL-TL template with template variables $v_1,\ldots,v_k$. Recall, that any instantiation of T must bind all variables in the head of T to constants. Since T is safe, all template variables in the body also occur in the head and the instantiation yields rules, where all template variables are replaced by constants and thus are safe. Since T is safe, all first order variables are safe by definition and so is $\langle\Sigma_T,\mathcal{I}\rangle$.

5.2 Field Classification

Field classification is based on the field constraints $\mathcal{C}_\mathcal{F}$ of the domain schema Σ. These constraints are specified in OPAL-TL to enable reuse of field types and templates. For instance, in the real estate and used car domains, we identify four templates that suffice to describe nearly all field constraints. These templates effectively capture very common semantic entities in forms and are parametrized using domain knowledge. The building blocks are a field type C and an annotation type A that is used to define a field constraint for C. None of these templates uses more than one annotation type as template parameter, though many query additional (but fixed) annotation types in their bodies.

```
1  TEMPLATE field_by_proper<C,A> {field<C>(N)⇐N@A{d,e,p}}
2
3  TEMPLATE field_by_segment<C,A>{field<C>(N)⇐N@A{e,p}}
4
5  TEMPLATE field_by_value<C,A>  {field<C>(N)⇐N@A{m},
6        ¬(A1 ≠ A, N@A1{d,e,p}∨N@A1{e,p}) }
7
8  TEMPLATE field_minmax<C,CM,A> {
9   field<CM>(N1)⇐child(N1,G),child(N2,G),adjacent(N1,N2),
10    N1@A{e,d},(field<C>(N2)∨N2@A{e,d})
11  field<CM>(N2)⇐child(N1,G),child(N2,G),next(N2,N1),
12    field<C>(N1),N2@range_connector{e,d},¬(A1≺A,N2@A1{d})
13  field<CM>(N1)⇐child(N1,G),child(N2,G),adjacent(N1,N2),
14    N1@A{e,p},N2@A{e,p},((N1@min{e,p},N2@max{e,p})
15      ∨(N1@max{e,p},N2@min{e,p}))
```

Fig. 10: OPAL-TL field classification templates

```
1  TEMPLATE segment<C>{
2   segment<C>(G)⇐lone<C>(G),child(N1,G),¬(child(N2,G),
3     ¬(C1 ⇝ C, field<C1>(N2)∨segment<C1>(N2))) }
4
5  TEMPLATE segment_range<C,CM> {
6   segment<C>(G)⇐lone<C>(G),field<CM>(N1),
7     field<CM>(N2),N1 ≠ N2,child(N1,G),child(N2,G) }
8
9  TEMPLATE segment_with_unique<C,U> {
10  segment<C>(G)⇐lone<C>(G),child(N1,G),field<U>(N1),
11    ¬(C1 ⇝ C, child(N2,G), N1 ≠ N2,
12                  ¬(field<C1>(N2)∨segment<C1>(N2))).}
13
14 TEMPLATE lone<C>{
15  lone<C>(G)⇐child(N,G),(segment<C>(N)∨field<C>(N)),
16                ¬(adjacent(G, G′), segment<C>(G′)). }
```

Fig. 11: OPAL-TL segment constraints

Figure 10 shows the field classification templates for real-estate and used car: **(1)** *Field by proper label.* The first template captures direct classification of a field N with type C, if N matches `X@A{d,e,p}`, i.e., has more proper labels of type A than of any other type A' with $A' \prec A$. This template is used by far most frequently, primarily for types with unambiguous proper labels. **(2)** *Field by segment label.* The second template relaxes the requirement by considering also indirect labels (i.e., labels of the parent segment). In the real estate and used car domains, this template is instantiated primarily for control fields such as ORDER_BY or DISPLAY_METHOD (grid, list, map) where the possible values of the field are often misleading (e.g., an ORDER_BY field may contain *"price"*, *"location"*, etc. as values). **(3)** *Field by value label.* The third template also considers value labels, but only if neither the first nor the second template can match. In that case, we infer that a field has type C, if the majority of its direct or indirect, value or proper labels are annotated with A. **(4)** *Minmax field.* Web forms often show pairs of fields representing min-max values for a feature (e.g., the number of bedrooms of a property). We specify this template with three simple rules (Line 5–12), that describe three configurations of segments with fields associated with value labels only (proper labels are captured by the first two templates). It is the only template with two template parameters, C and C_M where $C_M \sqsubset C$ is the "minmax" variant of C. The first locates, adjacent pairs of such nodes or a single such node and one that is already classified as C. The second rule locates nodes where the second follows directly the first (already classified with C), has a *range_connector* (e.g., "from" or "to"), and is not annotated with an annotation type with precedence over A. The last rule also locates adjacent pairs of such nodes and classifies them with C_M if they carry a combination of *min* and *max* annotations.

In addition to these templates, there is also a small number of specific rules. In the real estate domain, e.g., we use the following rule to describe forms that use links (`a` elements) for submission (rather than submit buttons). Identifying such a link (without probing and analysis of Javascript event handlers) is performed based on an annotation type for typical content, `title` (i.e., tooltip), or `alt` attribute of contained images. This is mostly, but not entirely domain independent (e.g., in real estate a there is a "rent" link).

```
field<LINK_BUTTON>(N1)⇐form(F),descendant(N1,F),link(N1),
  N1@LINK_BUTTON{d},¬(descendant(N2,F),
    (field<BUTTON>(N2)∨next(N1,N2)))
```

5.3 Segment Classification

As for field constraints, we use OPAL-TL to specify the segment constraints. The segment constraints and templates in the real estate and used car domains are shown in Figure 11 (omitting only the instantiation as in the field case). All segment templates require that the segment has at least one C child and is the lone C segment among its siblings (see `lone<C>(G)`). **(1)** *Basic segment.* A segment is a C segment, if its children are only other segments or fields typed with C. This is the dominant segmentation rules, used, e.g., for ROOM, PRICE, or PROPERTY_TYPE in the real estate domain. **(2)** *Minmax segment.* A segment is a C segment, if it has at least two field children typed with C_M where $C_M \sqsubset C$ is the minmax type for C. This is used, e.g., for PRICE and BEDROOM range segments. **(3)** *Segment with mandatory unique.* A segment is a C segment, if its children are only segments or fields typed with C except for one (mandatory) field child typed with U where $U \not\sqsubset C$. This is used, e.g., for GEOGRAPHY segments where only one RADIUS may occur.

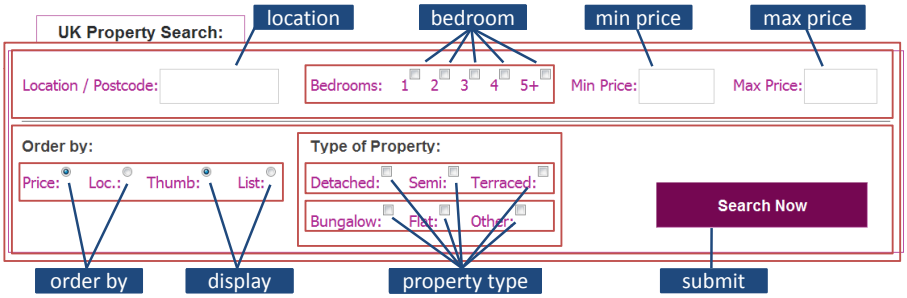


Fig. 12: *Farlowestates* before model repair

5.4 Model Repair

The classification yields a form interpretation F, that is not necessarily a model under Σ, and may contain violations of segment constraints, e.g., due to missing or superfluous segment nodes that resulting from a badly structured DOM tree. Recall (from Section 2), that we aim for a form model that is complete w.r.t. the suitable labels and types. Since "suitability" is up to human judgement, we can only hope to approximate it in the model repair by introducing types where they are "obviously" missing and removing them where they are "obviously" superfluous. To approximate suitability, we focus on *single-type defects*, i.e., defects where a repair for defects of a single type suffices to obtain a configuration for n that satisfies $\tau(n)$. For example, if a segment s has a defect A for type t and a defect B for type t', then we assume that repairing A or B alone yields an s' with $s' \models \tau(s')$. Other defects are treated by simply dropping the violating types. As demonstrated in Section 8, this suffices to obtain most suitable form types while avoiding unsuitable types introduced by complex repair sequences.

Let s be a segment with $s \not\models \tau(s)$. Then there is a type t such that $s \not\models t$ and we distinguish five types of defects for t:

(1) *Under Classification:* If there are untyped children $c_1, \ldots, c_k$ of s and corresponding types $t_1, \ldots, t_k$, such that $s \models \tau(s)$ would hold if each c_i is typed with t_i and $c_i \models \tau(c_i) \cup \{t_i\}$, then we call s under classified.

(2) *Over Classification:* If there are children $c_1, \ldots, c_k$ of s typed with $t_1, \ldots, t_k$ and $s \models \tau(s)$ would hold, if each c_i is not typed as t_i, and each c_i is also typed with a type different from t_i, then we call s over classified.

(3) *Under Segmentation:* If there is a segment s and a partition $P_1, \ldots, P_k, P_{\text{rest}}$ of its children, such that each P_i – wrapped in a new segment – satisfies $C_S(t_i)$, and s with additional segment children of type $\{t_1, \ldots, t_k\}$ satisfies $\tau(s)$, then s is said to be under segmented.

(4) *Over Segmentation:* If there are children $c_1, \ldots, c_k$ of s such that $s \models \tau(s)$ would hold after deleting all c_i and moving their children to s, then we call s over segmented.

(5) *Miss Classification:* Otherwise, s is miss classified.

Algorithm 4: `ModelRepair`(*Form interpretation* F)

```
1  for k ← height(F) down to 0 do
2      foreach segment s at depth k do
3          F_orig ← F;
4          F_max = F with all types for s removed;
5          foreach t ∈ τ(s) do
6              if under classified then type c_i with t_i;
7              else if over classified then remove t_i from c_i;
8              else if over segmented then
9                  delete each c_i moving the children to s;
10             else if under segmented then
11                 wrap P_i in segment c_i typed t_i;
12             if score(F) > score(F_max) then
13                 F_max ← F;
14             F ← F_orig;
15         F ← F_max;
16         if τ(s) = ∅ then delete s moving children up;
```

As stated above, we assume only simple repairs and thus at most one defect for a single type at each node. This still allows single-type defects for different types. To choose between the possible repairs, we use a simple scoring of form interpretations which favours more over less types and less over more segments: $\texttt{score}(F) = \sum_{n \in F} |\tau(n)| - |\text{SEGMENTS}_F|$. Algorithm 4 shows OPAL's approach to repair single-type defects. It iterates over all segments in a bottom-up fashion (Lines 1–2). For each segment s it considers single-type defects for each type (Lines 6–11) and determines which repair yields the highest scoring form interpretation (Lines 12–13). It also compares those with the case where all types for s are removed (Lines 4). The highest scoring repair is finally applied (Line 15) and segments that are untyped after the repair are deleted (Line 16).

Figure 12 shows the segmentation and classification OPAL obtains for a fairly complex real-estate form before model repair with several problems:

(1) The `min_price` and `max_price` fields are not arranged into a range segment as no such node is present in the DOM. This is a case of under segmentation. Following the `segment_range` constraint, OPAL introduces a price range segment to include both fields.

(2) The four radio buttons under "order by" are of different domain types, i.e., ORDER_BY for the first two and DISPLAY for the last two. Due to `field_by_segment` from Figure 10 and the segment label "order by", the last two would also be classified as ORDER_BY, if not for DISPLAY $\prec$ ORDER_BY. This is an example of under segmentation, where OPAL needs to split the existing segment as it is not supported by a segment constraint, but there are subsequences of children that can form valid segments.

(3) Having the four radio buttons grouped together, the last two buttons are also typed as ORDER_BY in addition to DISPLAY. OPAL resolves this over classification by removing ORDER_BY following the restructuring of the segment.

(4) The PROPERTY_TYPE segment is subdivided into two segments in the original segmentation, since OPAL identifies no style-equivalence among the six check boxes due to lack of similarity. However, two segments of PROPERTY_TYPE can not be contained in a single parent segment, see Figure 11. Thus, the two segments are removed, placing their children in the surrounding segment. This is an example of over segmentation.

(5) The original segmentation preserves the two DOM nodes representing the two form rows. However, in the domain schema, these nodes do not carry meaning, and thus are treated as over segmentation and removed.

5.5 Form interpretation complexity

The complexity of form interpretation is determined by the complexity of *fact inference* in safe OPAL-TL: Given safe OPAL-TL templates Σ_T, their instantiations $\mathcal{I}$, a set of atoms D and a single atom a, decide whether $D \cup \langle \Sigma_T, \mathcal{I} \rangle \models a$ holds, where $\langle \Sigma_T, \mathcal{I} \rangle$ denotes the Datalog$^\neg$ program obtained by instantiating Σ_T with $\mathcal{I}$. Hence, it is natural to ask whether OPAL-TL increases the complexity of fact-inference wrt. Datalog$^\neg$. The following results show that this is not the case.

Proposition 2 *Fact inference in* OPAL-TL *is* PTIME*-complete in data complexity (when Σ_T and $\mathcal{I}$ are fixed) and* EXPTIME*-complete in combined complexity.*

Proof As each instantiation $\mathcal{I}$ yields a Datalog$^\neg$ program $\langle \Sigma_T, \mathcal{I} \rangle$, the data complexity is PTIME-complete as for Datalog$^\neg$. Regarding the combined complexity, recall that fact inference for a Datalog$^\neg$ program $\langle \Sigma_T, \mathcal{I} \rangle$ and a set of atoms D is EXPTIME-complete, since the maximum number of inferrable atoms is $| \mathcal{R} | \, dom(D)^w$ where $\mathcal{R}$ is the set of predicates in $\langle \Sigma_T, \mathcal{I} \rangle$, $dom(D)$ is the domain of D and w is the maximum arity of predicates in $\mathcal{R}$. The rewriting $\langle \Sigma_T, \mathcal{I} \rangle$ generates at most $| \mathcal{R}_T | \cdot |\mathcal{I}|^k$ template-ground atoms if k is the maximum template arity. Therefore, the number of atoms that can be generated is $\mathcal{O}(2^k \cdot 2^w)$ that is still exponential. The claim follows.

Proposition 3 *Given a form interpretation where all fields are typed and satisfy their constraints,* OPAL*'s model repair yields a form model with correctly classified segments in polynomial time (when Σ_T and $\mathcal{I}$ are fixed).*

Proof We show that Algorithm 4 yields in polynomial time a form interpretation that is a form model.

First, note that OPAL-TL segment and field constraints only access siblings and children of the constrained node.

The bottom-up traversal in Algorithm 4 ensures that all children satisfy their type constraints before considering the parent segment. Furthermore, all repairs require that all involved nodes (segment and children) satisfy their type constraints after the repair, as we only consider single-type constraint violations **(1)**–**(4)**. This is explicitly ensured, except for removing all types (Lines 4) or deleting untyped segments (Line 16). However, all OPAL-TL constraints are such that they do not access the type of the parent segment (only those of the siblings and children). Thus, the constraints on the children remain satisfied.

Algorithm 4 requires at most $\mathcal{O}(|F|^2 \times |\mathcal{C}_\mathcal{S}|^2)$ time. More specifically, it processes each segment s in F at most once, performing for each segment up to $|\tau(s)| \leq |\mathcal{C}_\mathcal{S}|$ repairs and scorings, each in $\mathcal{O}(|F| \times |\mathcal{C}_\mathcal{S}|)$.

Theorem 2 *Up to suitability,* OPAL *provides a solution to the form interpretation problem with polynomial data complexity (when Σ_T and $\mathcal{I}$ are fixed).*

Proof From Theorem 1, we obtain a solution F of the form labeling problem in polynomial time. On that solution we apply the OPAL-TL field and segment classification constraints to obtain a form interpretation F' where all fields are typed and satisfy their constraints (by construction). This classification has polynomial data complexity (Proposition 2). F' may still contain unclassified or wrongly classified segments, which are repaired by Algorithm 4 in polynomial time (Proposition 3), yielding a form model.

In Section 8, we illustrate that the labels and types in form models obtained by OPAL are almost always suitable.

6 Deriving OPAL-TL Domain Schemas

In this section, we provide a methodology for deriving OPAL domain schemas, from a given description of the domain, e.g., an ontology. This is the typical way to instantiate a domain for use with OPAL.

Figure 13 shows a simple ontology for the used car domain (in the UK). Note, that most search forms are about searching for entities (double border in Figure 13) by their properties (single border) such as price or mileage of a car. Therefore, most of the types in an OPAL domain schema correspond to such properties of entities in the domain.

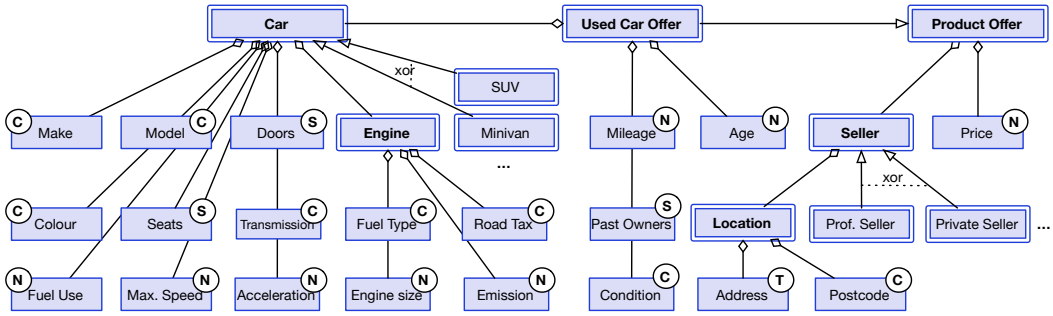


Fig. 13: Used car ontology

We observe that properties can be roughly distinguished into numerical, categorical, and free text according to their range and that these distinctions dictate to a large extent the expected form fields for searching by those properties. For a numerical property we expect, e.g., either a single text input or slider, two min-max fields for entering a range, or a set of checkboxes to select common values or ranges. Categorical properties, on the other hand, never exhibit range inputs.

These observations are codified in the derivation templates of Figure 14. These templates group typical instantiations for the above kinds of properties as well as for compound object types such as LOCATION in Figure 13:

(1) For an **object type** (ENGINE), we instantiate only the `segment<C>` template, i.e., we allow segments, but not fields of this type. Such segments typically collect multiple properties of the object type, e.g., ENGINE_SIZE and FUEL_TYPE.

(2) For a **free text type** (e.g., ADDRESS), we instantiate only the `field_by_proper<C,A>` and `field_by_value<C,A>` templates that allows fields, but not segments of that type. There is usually no need for a segment in this case, as there are rarely multiple occurrences of fields for such a type. In the rare case where that is nevertheless possible, we instantiate `segment<C>` separately.

(3) For a **categorical type** (MAKE or COLOUR), we instantiate in addition to `field_by_proper<C,A>` also `segment<C>` and the `field_by_segment<C,A>`. Categorical types are often represented as single select boxes or lists of radio buttons or check boxes. For the latter, an enclosing segment is desirable and `field_by_segment<C,A>` allows us to propagate the segment labels to the fields.

(4) For a **numerical type** (PRICE or SEATS), we also instantiate the `segment_range` and `field_minmax` templates, enabling the classification of range segments and fields.

With these templates, we can derive an OPAL annotation and domain schema very quickly from a given domain schema such as Figure 13.

First, we normalize the ontology: If a class C has subclasses without additional properties (type classes), we generate a new categorical property C_TYPE, add all labels from the sub-classes to that property, and remove the sub-classes.

```
1  TEMPLATE object_type<C> {
2    INSTANTIATE segment<C> using { <C> } }
3
4  TEMPLATE free_text_type<C,A> {
5    INSTANTIATE field_by_proper<C,A> using { <C,A> }
6    INSTANTIATE field_by_value<C,A> using { <C,A> } }
7
8  TEMPLATE categorical_type<C,A> {
9    INSTANTIATE field_by_proper<C,A> using { <C,A> }
10   INSTANTIATE field_by_segment<C,A> using { <C,A> }
11   INSTANTIATE field_by_value<C,A> using { <C,A> }
12   INSTANTIATE segment<C> using { <C> } }
13
14 TEMPLATE numeric_type<C,C_M, A> {
15   INSTANTIATE field_by_proper<C,A> using { <C,A> }
16   INSTANTIATE field_by_segment<C,A> using { <C,A> }
17   INSTANTIATE field_by_value<C,A> using { <C,A> }
18   INSTANTIATE field_minmax<C,C_M,A> using { <C,C_M,A> }
19   INSTANTIATE segment<C> using { <C> }
20   INSTANTIATE segment_range<C,C_M> using { <C,C_M> } }
```

Fig. 14: Templates for object and property types.

Second, we derive the annotation schema and, in particular, the necessary annotators as follows:

(1) For each concept or property c of the ontology, we create an annotation type c. All *labels* of c, possibly enriched with synonyms from an external knowledge base such as Wordnet, form an annotator for the proper labels of the concept ($\mathsf{isLabel}_c$).

(2) For categorical concepts or properties, we require a list of instances, a regular expression, or an external entity recogniser, again possibly provided by an external knowledge base such as DBPedia or LinkedGeoData or an external service such as OpenCalais. Numerical values are treated similarly, though these often take simply the form of number in a certain range. This provides $\mathsf{isValue}_c$.

Third, we derive the domain schema in four steps:

(1) For each **class** C, add an instantiation rule for `object_type<C>`. In our example, this yields 6 instantiations (recall, that type classes are normalised to properties above).

(2) For each **property**, add an instantiation rule of corresponding type, e.g.,

```
INSTANTIATE numeric_type<C,C_M,A> using {<PRICE,PRICE_M,price>}
```

In our example, this yields 22 instantiations (20 properties from Figure 13 and two ..._type properties).

(3) Determine which "presentational" fields and segments occur in the given domain and add them to the domain schema. A field or segment is presentational, if it determines the way the results are represented. In the used car and real estate domains, we identify two types of presentational fields: "order-by" and "pagination" which control the order in which the results are presented as well as the number of results per page. These presentational types are mostly shared between domains and can be easily reused thanks to OPAL-TL templates:

```
INSTANTIATE categorical_type<C, A> using
  { <ORDER_BY, order_by> <PAGINATION, pagination> }
```

In this step, we also add generic rules that are independent of the domain, e.g., for the form itself and domain-independent form facilities such as submit buttons or generic keyword search fields.

(4) Sometimes small manual adjustments are necessary. For example, numerical types may occur with multiple units of measure or other modifiers, e.g., prices with different currencies or locations with a search radius. Such modifier fields are usually unique in their corresponding segment and thus added using the `segment_with_unique<C,U>` template. In the used car domain, we observe this for CURRENCY and RADIUS:

```
INSTANTIATE TEMPLATE segment_with_unique<C,U> using
  { <PRICE, CURRENCY> <LOCATION, RADIUS>  }
INSTANTIATE TEMPLATE field_by_proper<C,A> using
  { <CURRENCY,currency>, <RADIUS,radius> }
INSTANTIATE TEMPLATE field_by_value<C,A> using
  { <CURRENCY,currency>, <RADIUS,radius> }
```

Some object types, in particular LOCATION, may also be entered as a whole through free text fields and accordingly instantiate the `free_text_type` template for them:

```
INSTANTIATE TEMPLATE free_text_type<C,A> using
  { <LOCATION,location }
```

Finally, we need to determine part-of, type alternatives, and precedence between types. The part-of relations and type alternatives are derived from the associations of the domain schema, e.g., ADDRESS $\multimap$ LOCATION, FUEL_TYPE $\multimap$ ENGINE, SUV $\underline{xor}$ MINIVAN for our case. Precedence requires some observation of cases where annotations for different types overlap. Typically, we want to give presentational types precedence over all domain types (as they often contain values such as "sort by price"). For the used car domain, we observe that PAGINATION $\prec$ ORDER_BY and that both have precedence over all domain types. We also observe that MILEAGE and RADIUS (in locations) can have overlapping values. Though radius is only used in `segment_with_unique<C,U>`, for LOCATION segments which disallow MILEAGE elements, we add MILEAGE $\prec$ RADIUS to express a preference for MILEAGE.

Derivation effort. For the real estate domain, our domain schema consists of a few dozen field and segment types and about 40 annotation types. Similarly, in the used car domain, there are about 30 annotation types. Creating and verifying an initial domain schema (including annotators) takes a single person familiar with the domain and basic conceptual modelling and programming skills roughly 1 week. When not available, annotators have been created by acquiring the necessary background knowledge from structured sources such as LOD and FreeBase. In our experience, the human effort required to build the domain schema has been often significantly lower than the effort required to manually annotate the test corpus for OPAL.

7 Light-weight Form Integration

OPAL's form models allow the easy implementation of many types of applications that require automatic understanding and interaction with forms, such as form integration and filling, data extraction, or web automation. As discussed in Section 2, we focus here on *form integration* (or filling), i.e., the part of a web integration system [15] that translates a query on the global schema (OPAL's domain schema) to a query against concrete forms. In this section, we introduce a light-weight form integration system that performs this task fully automatically in a given domain, only requiring an OPAL domain schema. We have instantiated this system for the real estate and used car domain, but OPAL is as easily applied to other domains, since only a very limited amount of additional customisation is needed (on type variations and, possibly, similarities).

Recall, that we focus on the optimistic, single-query variant of the form integration problem: We aim for a single-query that returns all results matching the global (or *master*) query, but allow to return also non-matching results, if there is no more specific query that returns all matching ones.

OPAL's form integration translates the master query into concrete queries through a small set of translation rules supported by a notion of similarity on property values. OPAL can perform form integration without any other information than what is provided by an OPAL domain schema and corresponding form model. However, it can be further improved by providing additional domain-specific information.

Similarity on values is represented as a real-valued function on pairs of values and is based on the property type: For free-text and categorical properties, OPAL uses a mix of Levenshtein and longest common substring distance, for numeric properties a difference-based similarity. A domain schema can be enhanced by property-specific similarity functions, e.g., to deal with different units of measure. A

small set of such functions is provided with OPAL: for price, for distance properties, and for dates.

Translation rules use these similarity functions to translate the constraints of the master query Q into queries on the concrete forms. For each form F with form model M and constraint $C \in Q$ on type T, we retrieve the fields $f_1, \ldots, f_n$ classified with T. Let $\mathsf{values}(C)$ be the (possibly infinite) set of values for which C holds.

(1) *Single field, single value:* If $n = 1$, $\mathsf{values}(C) = \{v\}$, and
- (i) f_1 is a free text input, return $f_1 = v$.
- (ii) f_1 is a select box, return $f_1 = v'$ where v' is the option of f_1 most similar to v.

(2) *Multi field:* If $n \geq 1$,
- (i) $\mathsf{values}(C) = \{v\}$, and all f_i are radio buttons (exclusive options), return $f_k =$ **true** for the f_k that is most similar to v.
- (ii) $\mathsf{values}(C) = \{v_1, \ldots, v_k\}$ and all f_i are check boxes (non-exclusive options), return $f_k =$ **true** for each f_k where a v_i exists such that the similarity of f_k and v_i is minimal among all such pairs.
- (iii) and all f_i are free-text range input fields (i.e., of type T_M, where T_M is the minmax type to T), then return $f_s = v_1$ for each f_s that is a minimum input and $f_e = v_k$ for each f_k that is a maximum input.
- (iv) and all f_i are select-box range input fields, then return $f_s = v'_1$ for each f_s that is a minimum input where v'_1 is the most similar option of f_s to v_i that is smaller or equal to v_1. Analog for f_e.

In all other cases (e.g., a select box for a set inclusion constraint), we return no constraints to avoid false negatives.

In many domains, we can observe that the same information is represented in alternative ways on different sites. E.g., the age of a car is represented by the manufacturing year on some sites. Similarly, the location of property may be given as a street address, a postcode, or even just a town, in particular for rural agencies. To treat this cases, we need to be able to translate a constraint such as "AGE $= 6$" to a constraint "YEAR $= 2006$" or "POSTCODE $=$ OX1" to "TOWN $=$ Oxford". We call AGE and YEAR *type variants* and amend the domain schema with a value mapping for each pair of type variants. Value mappings for numerical properties are typically simple conversion functions, e.g., from different units of measure. Value mappings for categorical properties are typically realised by a query to an external database or service such as DBPedia. In our example domains, we use value mappings for conversions of metric and imperial distances as well as of postcodes to towns and other locations. To treat type variants we perform the following test and translation before the aforementioned translation rules:

(0) *Type variants.* If $n = 0$ and there is a field f' with type T' such that T' is a variant type of T, we translate the values in C to T' and continue with that constraint.

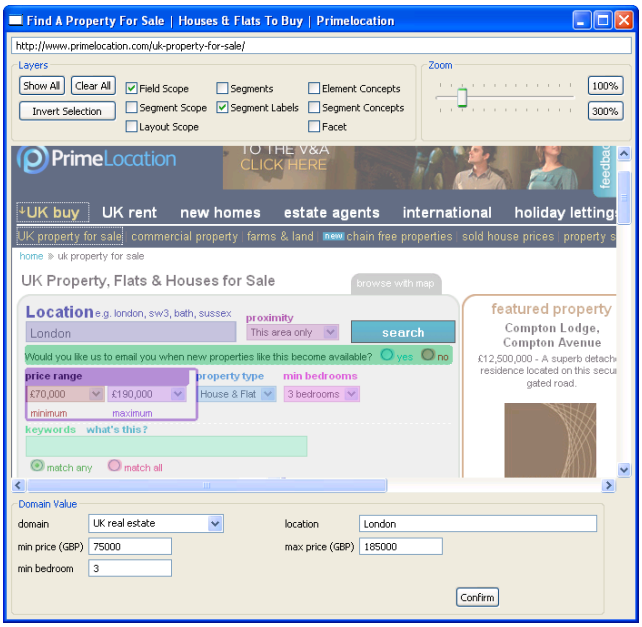


Fig. 15: OPAL Testing Tool

With those simple rules, OPAL's form integration manages to translate most constraints as shown in Section 8. There are, of course, still cases where the translation fails, e.g., if categorical values are mapped to ranges by some ordering such as road tax brackets or iPhone models (ordered according to year of introduction). But as demonstrated in Section 8, this light-weight simple form integration already provides us with a successful translation of a master query in the vast majority of cases.

To illustrate OPAL's form integration, we consider the form of `primelocation.com` as shown in the middle of Figure 15. The figure shows the OPAL testing tool that we use to test and verify the accuracy of OPAL domain schemas. It allows the user to visualize the form labels, form segments, and classifications derived by OPAL and to track down, where, e.g., there are problems with the classification constraints or the annotations. It also provides a master query in the lower third. The concrete form is automatically filled according to the values provided in the master form. This allows the user to visually verify that the query has been translated correctly. The master form is automatically generated from the domain schema, but the user can provide additional information on which fields to include. For space reasons, we have focused in Figure 15 on the types most commonly used in constraints in the UK real estate domain.

For the concrete form on `primelocation.com`, we highlight form fields and labels by colouring them with the same color (here, e.g., the "minimum" and the first price field). Form segments are shown as boxes with no filling except for their labels (a price segment with "price range" label). The figure shows the form *after* OPAL has filled it accord-

ing to the values from the master query. Notice, how for the three select boxes for minimum and maximum price, as well as bedroom number, OPAL picks the closest value to the one specified in the master form.

8 Evaluation

We perform experiments on several domains across four different datasets. Two datasets are randomly sampled from the UK real estate and UK used-car domains, respectively. We compare with existing approaches via ICQ and TEL-8, two public benchmark sets, on which we only evaluate OPAL's form labeling. This limitation is necessary to allow a comparison that is fair to existing approaches, that only report form labeling and do not use domain knowledge. Even with this limitation, however, OPAL outperforms previous approaches in most domains by at least 5%. We also perform an introspective analysis of OPAL to show **(1)** the impact of field, segment, layout, and repair in the form interpretation, **(2)** OPAL's performance and scalability with increasing page size, and **(3)** the effectiveness of the form integration in OPAL.

We evaluate the proper assignment of text nodes to form fields using standard notions of precision, recall and F-score (harmonic mean $F = F_1 = 2PR/(P+R)$ of precision and recall). For form labeling (classification), precision P is measured as the proportion of correctly labeled (classified) fields over total labeled fields, while recall R is the fraction of correctly labeled fields over total number of fields. For form filling precision and recall do not apply and we therefore report the error rate as portion of total fields that are not correctly filled (i.e., either filled but with a wrong value or not filled at all, despite a corresponding constraint in the master query). For all considered datasets, we compare the extracted result to a manually constructed gold standard. We evaluate segmentation through their impact on classification, see Figure 18a, and the improved performance on the two datasets where we perform form interpretation (UK real estate and used car) versus the ICQ and TEL-8 datasets.

Datasets. For the UK real estate domain, we build a dataset randomly selecting 100 real estate agents from the UK yellow pages (`yell.com`). Similarly, we randomly pick 100 used-car dealers from the UK largest aggregator website `autotrader.co.uk`. The forms in these two domains have significantly different characteristics than the ones in ICQ and TEL-8, mainly due to changes in web technology and web design practices. The usage of CSS stylesheets for layout and AJAX features are among the most relevant.

The ICQ and TEL-8 datasets cover several domains. ICQ presents forms from five domains: air traveling, (used) cars, books, jobs, (U.S.) real estate. There are 20 web pages for each of the domains, but two of them are no longer accessible and thus excluded from this evaluation. TEL-8, on the other hand, contains forms from eight domains: books, car rental, jobs, hotels, airlines, auto, movies and music records. The dataset amounts to 477 forms, but only 436 of them are accessible (even in the cached version).

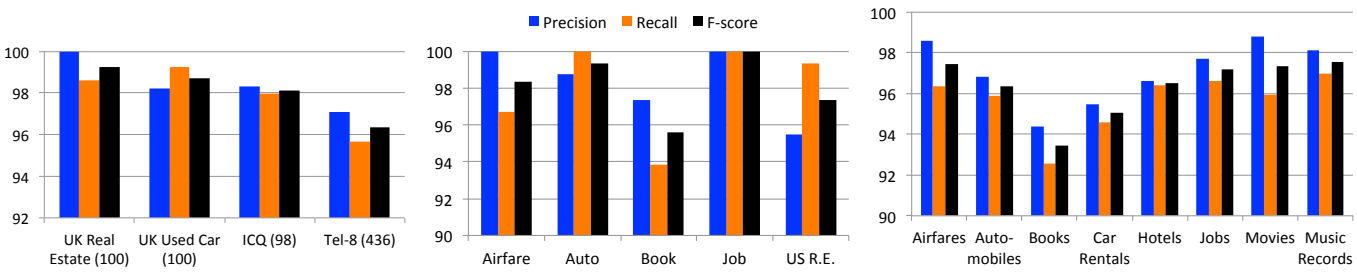
8.1 Field Labeling

In our first experiment we evaluate the accuracy of OPAL's field labeling on all four datasets, but only in the UK real estate and used car domain we employ the form interpretation to further improve the field labeling. Figure 16a shows the results. The first two bars are for the random sample datasets. For the real estate domain, OPAL classifies fields with perfect precision and 98.6% recall. Overall we obtain a remarkable 99.2% F-score. The result is similar for the used car domain, where OPAL obtain 98.2% precision and 99.2% recall, that amount to 98.7% F-score. OPAL achieves lower precision than recall in the used car domain due to the fact that web forms in this domain are more interactive: certain fields are enabled only when some other field is filled properly, yet non-field placeholders are present in the HTML to indicate that a field will appear when the other field is filled. This introduces noise to field labeling and classification.

The other two entries in Figure 16a regard field labeling on the ICQ and TEL-8 datasets. On these, OPAL applies only its domain-independent scopes (field, segment, scope) as no domain schema is available for these domains. Nonetheless, OPAL reports very high accuracy also on these forms, confirming the effectiveness of our domain-independent analysis. Not unexpected, OPAL performs better in the UK real estate and used car domain where domain knowledge is present, even though the forms in those datasets are on average more modern and contain more fields (10.4 and 9.2 fields per form in the real-estate and used-car dataset versus 6.5 and 7.9 fields per form for ICQ and Tel-8).

Cross Domain Comparison. We use ICQ and TEL-8 to compare field labeling in OPAL against existing approaches, on a wide set of domains. Figure 16b details the result of OPAL on each domain of the ICQ dataset. It shows perfect F-score values for the jobs domain (100%) as well as auto and air travelling (99.3% and 98.3%). For comparison, [11] reports 92% F-score for labeling on ICQ on average, which we outperform even in the domain most difficult for OPAL (books). [33] reports slightly better precision and recall than [11], but OPAL still outperforms it by several percents.

The results for the TEL-8 dataset are depicted in Figure 16c. Here, the overall F-score is 96.3%, again mostly affected by the performance in the books domain. Note that, especially on TEL-8, OPAL obtains very high precision compared to recall. Indeed, lower recall means OPAL is not able



(a) Overall (b) ICQ results (c) TEL-8 results

Fig. 16: OPAL labeling performance

	field	labeling segment	layout
total	761	154	72
false positives	2	3	8
= %	0.3%	1.9%	11.1%

Table 1: False positives

to assign labels to all fields, missing some of them. For comparison, [11] reports 88 – 90% overall F-score, which we outperform by a wide margin. [24] reports F-scores between 89% and 95% for four domains in the TEL-8 dataset. Though they perform slightly better on books, we significantly outperform them on the three other domains included in their results, as well as on average. It is worth noting that both [11,24] use a slightly differently sampled set of forms from TEL-8, though this should not significantly affect the presented comparison.

In Section 5, we discuss that OPAL prioritises field over segment over layout scope and we claim that this is due to the decreasing precision. Table 1 shows the total number of fields labeled in each scope, as well as the number and percentage of false positives among those labels. It illustrates that, indeed, the field scope produces almost no false positives (2 out of 762 fields labeled in this scope, i.e., 0.3%), the segment scope also produces very few (3 out of 154 labeled fields), and the layout scope produces most (8 out of 72 labeled fields).

What keeps OPAL from achieving 100% accuracy? Most of the cases are due to OPAL's assumption that form labels are separate text nodes. This is evidently the case in most forms, as demonstrated by near perfect accuracy, but there are some outliers that use image only labels or merge multiple labels into one node and use whitespace to achieve the desired result, e.g., by aligning text in a single node to different fields using ` `. While both cases are easy enough to address, they do require specific treatment and we omitted them from the version of OPAL presented here to illustrate that nearly perfect form labeling and interpretation is possible even without such specifically tailored heuristics.

8.2 Form Interpretation

The quality of OPAL's form interpretation depends on the quality of the form labeling and that of the annotators. As discussed above, for this evaluation we used background knowledge for the UK real estate and used car domains plus generic entities such as locations, numbers, and colors. The location related annotators are based on standard sources (GeoNames and LinkedGeoData) and thus have reasonable recall, but precision is fairly low, due to the high number of locations in the UK that are homonyms to common English words (e.g., the town of "Selling"). Noise in the value annotators, however, affects OPAL very little, as the values of form fields are only used if the proper labels are inconclusive and then only those of the most frequent annotation type. Noise in the label values is far more likely to lead to classification errors. However, typical annotators are small lists of 5 – 10 typical labels which are easy to create and have very low noise. E.g., for *bedroom* labels we use just "bedroom", "bed", and their plural forms, for *make*, *model*, *mileage* and many more just "make", "model", "mileage", and their plural form, resp.

With this, we achieve near perfect classification, correctly classifying most of the fields, see Table 1: Precision is 97.3% over all fields in the real estate data set (with just 24 out of 931 classified fields incorrectly classified) and recall 97.4%. This excludes 56 (or 5.5%) fields for which our domain schema does not contain a concept (usually as they appear only very rarely).

Classification errors are mostly caused by ambiguity in the used form labels. For example, consider a field with a proper label "model style" which is correctly assigned to the field in the field labeling, and field values "4x4", "City Car", etc. In the classification, we prioritise proper labels over values (as value annotators are more noisy). In most cases, this is indeed preferable, but here the proper label "model style"

is annotated with *model* and we classify the field as *model* rather than *car_type*, as "model style" is not recognised as a label for *car_type*. A probabilistic classifications that combines classifications from labels and values (with a lower weight) would allows us to choose the most likely global form classification and thus to address such outliers. However, this would also increase the effort in creating a domain schema.

8.3 Contributions of Scopes

We demonstrate the effectiveness of combining different types of analysis by measuring to what extent each of our four scopes contributes to the overall quality of form understanding. We use again the two domain datasets from the previous experiment. For both we show the results for recall, though the picture is similar for precision and F-score, cf. Figure 16a. As illustrated in Figure 18a, for the field labeling in the real-estate dataset, the field scope already contributes significantly (67%). The Segment scope increases recall by 18%, layout scope and the repair in the form interpretation add together another 13%. Note that, the contribution of the repair in the form interpretation is more significant than that of the layout scope, indicating the importance of domain knowledge to achieve very high accuracy form understanding. In the used car domain, field scope alone is even more significant 85% (as many of the websites use modern web technologies and frameworks with reasonable structure). Figure 17 further shows which individual scopes are necessary on which page (of the 100 real estate sites). A dark blue rectangle indicates that the scope is used to label at least one field, a light blue that it is not used. We also show where OPAL misses some fields (red rectangles in the last row). This shows that only for less than a third of the pages does the field scope suffice. For more than a quarter three or more scopes are necessary.

8.4 Form Integration

For the evaluation of the form integration, we determine the error rate in the query translation for all forms in the used car and real estate datasets. We use multiple master queries in both cases, using for the real estate domain combinations of location, min price, max price, and min bedroom. For the used car domain, we use combinations of location, make, model, min price, and max price. The values are generated randomly from the known Gazetteers or valid numeric ranges. Values for different properties are generated independently, except for "make" and "model" in the used car domain, which are cross-validated. We evaluate the constraints separately and consider a constraint correctly translated, if it involves the right field on the concrete form

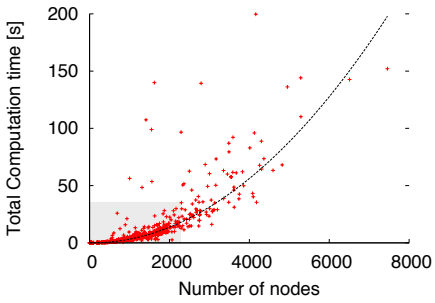


Fig. 19: Time

and uses the best matching value. Overall, OPAL generates 95.6% and 93.8% correctly translated constraints.

Figure 18b presents the number of web forms where OPAL fails to translate one or more constraints correctly. Overall, 87% of the forms were filled perfectly, and 95% of the forms have no more than one failure. Figure 18c presents the major causes for OPAL's failure in translating constraints: Most of the errors are caused by scripted forms with hidden (21%) or heavily customised form controls (24%). The remaining cases divide rather evenly between errors in the form labeling (17%), in the classification or annotation (incomplete gazetteer), and an assortment of other issues, mostly browser related (e.g., scripted popovers that block access to the form fields or fields that can only be filled in a certain order).

8.5 Scalability

As discussed in Sections 4 and 5, overall the analysis of OPAL is polynomial in the size of the form. As expected actual performance follows a quadratic curve, but with very low constants. There is a significant amount of outliers, partially due to long page rendering time and partially due to variance in the depth and sophistication of the HTML structure. Figure 19 reports OPAL performance on all 534 forms in the combined TEL-8 and ICQ datasets. The highlight area covers 80% of the forms with 2200 nodes. OPAL requires at most 30*s* for the analysis (including page rendering) of these forms. Further analysis on the effect of increasing field or form numbers confirms that these have little effect and page size is the dominant factor.

9 Related Work

Form understanding has attracted a number of approaches motivated by deep web search [21,28,29], meta-search engines and web form integration [16,11,32–34,36] and web extraction [30,31]. We focus here on differences to OPAL, for a complete survey see [19,12]. We present related work

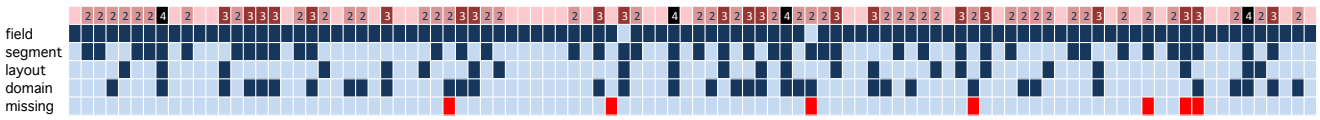
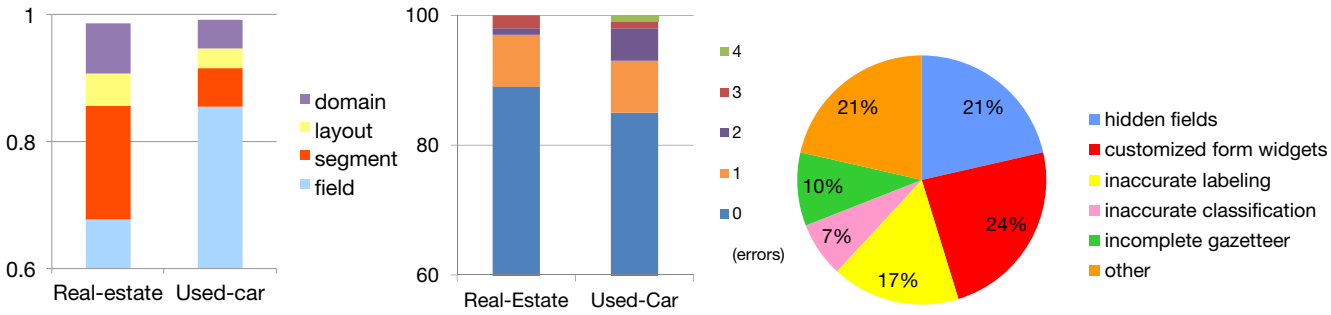


Fig. 17: Scopes by form (real-estate)



(a) Scope contribution (b) Number of integration errors (c) Type of integration error

Fig. 18: OPAL evaluation

for form understanding and form integration separately, as not all approaches consider both aspects.

9.1 Form Understanding

Form understanding approaches can be roughly categorised by the fundamental approach to the problem:

(1) The most common type encodes (mostly domain independent) observations on typical forms into implicit heuristics or explicit rules MetaQuerier [9,36], ExQ [33], SchemaTree [11], LITE [28], Wise-iExtractor [16], DEQUE [29], and CombMatch [17]. **(2)** Alternatively, some approaches LabelEx [24] and HMM [18] use machine learning from a set of example forms (possibly of a specific domain). **(3)** Form understanding is often done to surface the results hidden behind the form and approaches such as [21,32,28] exploit the extracted results for form understanding.

Aside of system design, OPAL primarily differs from these approaches in two aspects: **(1)** They mostly incorporate only one or two of OPAL's scopes (and feature classes): MetaQuerier, ExQ, and SchemaTree mostly ignore the HTML structure (and thus field and segmentation scope) and rely on visual heuristics only; CombMatch, LITE, DEQUE, and LabelEx ignore field grouping. HMM ignores visual information. [21, 32,28] use only the HTML structure, but exploit probing information, i.e., whether a submission is successful or not. **(2)** None of the approaches provides a proper form model classifying the form fields according to a given schema. Furthermore, no approach uses domain knowledge is used to improve the labeling or verify the classification, though LabelEx analyses domain specific term frequencies of label texts and HMM checks for generic terms, such as "min". As evident in our evaluation, each of the scopes in OPAL considerably affects the quality of the form labeling and classification. The fact, that each of these approaches omits at least one of the domain-independent scopes, explains the significant advantage in accuracy OPAL exhibits on Tel-8 and ICQ. Notice also that not using domain knowledge keeps these approaches out of reach of the nearly perfect field classification achieved by OPAL.

Form understanding by observation and heuristics. Most closely related in spirit to OPAL, though very different in realisation and accuracy, is MetaQuerier [36]. It is built upon the assumption that web forms follow a "hidden syntax" which is implicitly codified in common web design rules. To uncover this hidden syntax, MetaQuerier treats form understanding as a parsing problem, interpreting the page a sequence of "atomic visual elements", each coming with a number of attributes, in particular with its bounding box. In a study covering 150 forms, the authors of MetaQuerier identified 21 common design patterns. These patterns are captured by production rules in grammar with preferences. MetaQuerier is not parameterisable for a specific domain. In contrast, the domain independent part of OPAL achieves nearly perfect accuracy with only 6 generic patterns by combining visual, structural, and textual features, and a simple prioritisation of these patterns by scope. OPAL's domain dependent part allows us to adjust it for patterns specific to a domain.

ExQ [33] is similarly based primarily on visual features such as a bias for the top-left located labels comparable to OPAL, but disregards most structural clues, such as explicit `for` attributes of `label` tags and does not allow for any domain specific patterns.

Also SchemaTree [11] uses only visual features (and the `tabindex` and `for` attributes for fields and labels). It exploits nine observations on form design, e.g., that query interfaces are organised top-down and left-to-right or that fields form semantic groups. It uses a hierarchical alignment between fields and text nodes similar to OPAL's segment scope and a "schema tree" where the nine observations are observed. Again, no adaptation to a specific domain is possible.

Wise-iExtractor [16] firstly tokenizes the form to obtain a high-level visual layout description (an *interface expressions (IEXP)*), distinguishing text fragments, form fields, and delimiters, such as line breaks. It then associates texts and fields by computing the *association weight* between any given field and the texts in the same line and the two preceding lines, exploiting ending colons, similarities between the text and the field's HTML name attribute, and the text-field distance. In addition, Wise also identifies generic relationships between fields: range (e.g. from, to), part (e.g. first and last name), group (e.g. radio buttons), or constraint (e.g. exact match required). However, in contrast to OPAL their form labeling only explores limited visual and textual information relying mainly on weight computation. Moreover, their domain-independent typing shares some similarities with OPAL's templates but lacks the flexibility provided by OPAL's domain schemata that allow us to adjust these generic types to a given domain. Though these adjustments are often small, their impact is significant, as shown in Section 8.

In [35], a (manually derived) domain schema is used to guide understanding. In contrast to OPAL, it segments a form purely based on the domain schema (called schema tree). They evaluate on a fragment (around 100-150 forms) of TEL-8 using domain schemata derived from the rest of TEL-8 (about 250 forms). This yields on the considered fragment similar accuracy as OPAL achieves on the full TEL-8, yet OPAL does not use any domain schema in this case, let alone domain schemata specifically trained on TEL-8.

Form understanding by learning from example forms. Where the above approaches rely on humans to derive heuristics and rules for form understanding, the following approaches use machine learning on a set of example forms. Therefore, they can also be trivially adapted to a specific domain by using domain-specific training data. The evaluation in [18], however, shows little effect of domain-specific training data: a training set from the biological domain outperforms domain-specific training in 4 of the 5 other domains.

LabelEx [24] uses limited domain knowledge when considering the occurrence frequencies of label terms. Domain relevance of the terms occurring in a label, measured as the occurrence frequency in previous forms, is one feature used to score field-label candidates. Field-label candidates are otherwise created primarily using neighbourhood and other visual features, as well as their HTML markup. However, LabelEx does not consider field groups and thus is unable to describe segments of semantically related fields or to align fields and labels based on the group structure and does not use any domain knowledge aside of term frequency.

HMM [18] uses predefined knowledge on typical terms in forms, such as "between", "min", or "max", but does not adapt these for a specific domain. HMM employs two hidden Markov models to model an "artificial web designer". During form analysis, the HMMs are used to explain the phenomena observed on the page: The state sequences, that are most likely to produce the given web form, are considered explanations of the form. Compared to OPAL, HMM uses no visual features and no domain knowledge.

Form understanding by probing. All the above approaches conduct their analysis based purely on information available on the web forms. Alternatively, there is also an indirect route for form understanding by submitting the forms and analysing the query results, which often are much easier to classify (as there are many instances compared to a single form). The price is, however, that a certain amount of analysis of those result pages is necessary. Therefore, this is primarily used in a context where such analysis is anyway required, e.g., in crawlers or data extraction systems. Typically, these approaches use an incremental approach, identifying inputs for some fields, submitting the form, analysing the result page, and then possibly restarting the whole process, now with, e.g., an increased set of input values for the form. For example, [21] determines whether a field must be filled or is a "free" input by iterating over possible templates and selecting those that return sufficiently distinct result pages. This is driven by the desire to surface some representative, but not necessarily complete set of results from the web form. None of these approaches produces a sophisticated form model, but at best rough classifications of the fields and whether they are mandatory.

9.2 Form Filling and Integration

Form integration has been considered in many shapes, either as "meta-search" where a master query on a given global schema is translated to concrete forms as in OPAL, as "interface matching" where many concrete forms are integrated without a global schema (often using schema matching), or as "query generation" in the context of data extraction or crawling where the aim is to generate a set of queries to extract all or most of the data, but often not even full form understanding is performed.

Though some query generation and most interface matching approaches use form understanding, they are focused on different issues than OPAL's form integration

which is a type of "meta-search": How to find an optimal query set that uncovers as much deep content as possible [3], how to determine if a query will produce relevant data even if only partial information about the data is available [5], how to maximize the diversity of the extracted content [21], or how to identify semantic equivalences between fields from different forms [25].

Similar to OPAL, [1] fills web forms by connecting fields at the conceptual level, but with WordNet [27] instead of proper annotations. Furthermore, OPAL produces more structured form model that is verified against a domain schema. MetaQuerier [9], targets the integration of web sources and tackles query translation for form filling in that context. As OPAL, MetaQuerier selects values closest to the constraint in the source query (similar to our master query). They also perform type-based query translation to map a source query to a target query considering numeric and text types, but achieve only 87% accuracy. OPAL performs form filling in a similar fashion, but also considers the number of fields for each domain type in the master query and performs significantly better (93%).

10 Conclusion and Future Work

To the best of our knowledge, OPAL is the first comprehensive approach to form understanding and integration. Previous form understanding approaches have been limited mainly by generic, domain independent, monolithic algorithms relying on narrow feature sets. OPAL pushes the state of the art significantly, addressing these limitations through a very accurate domain independent form labeling, exploiting visual, textual, and structural features, by itself already outperforming existing approaches. This domain independent part is complemented with a domain dependent form field classification that significantly improves the overall quality of the form understanding and produces nearly perfect form interpretations. Accurate form interpretations enables form integration: OPAL successfully realizes a light-weight form integration system, able to translate master queries to forms of a domain with nearly no errors.

Nevertheless, there remain open issues in OPAL and form understanding in general that need to be addressed for form understanding to become a reliable tool to access web data through forms with little more effort than through APIs:

(1) Dynamic, scripted forms: OPAL is able to understand most static forms with near perfect accuracy, but performs much worse on dynamic forms. We are already working on an extension of OPAL for dealing with dynamic, heavily scripted interfaces that combines ideas from state exploration and crawling with form understanding.

(2) Customised form widgets: More and more forms use customised widgets such as tree views or sliders. Though most of these cases use hidden form fields that can be analysed by OPAL, the use of fully scripted cases increases. We plan to extend OPAL to allow the customisation of the form widgets that it can recognise. However, if these cases become more common, it may become necessary to automatically explore and learn such new widget types.

(3) Probing-based understanding: One of OPAL's virtues is that it achieves its near perfect accuracy without any probing, using only the form page. However, this also limits the information that OPAL can provide, and prevents the verification and repair of the form model through the results returned by a form submission. For applications that need to access the result pages (e.g., data extraction and surfacing), we plan to integrate OPAL with the result page analysis system AMBER [14] to further improve accuracy and to address integrity and access constraints.

(4) Integrity and access constraints. OPAL produces some integrity constraints through the domain schema and it's form segmentation, e.g., dependencies between min and max fields in a range segment. We see an increase in the use of integrity constraints in forms thanks to the availability of easy-to-use client-side validation libraries. Light-weight methods for analysing and exploiting such client side validation would allow us to extend our form models with more detailed integrity constraints. This is in addition to integrity and access constraints derived from probing.

(5) Supporting domain-schema derivation: A domain schema for a new domain can be easily obtained by means of the methodology described in Section 6. We are currently developing a support tool for semi-automatic derivation of the necessary OPAL-TL templates from ontologies specified in OWL or as UML Class Diagrams. The second challenge is the semi-automatic acquisition of the necessary background knowledge, e.g., gazetteers, for domain types and properties, whether by leveraging existing knowledge bases, Gazetteer extension techniques, or crowdsourcing [10] to partially bootstrap this process.

(6) Evolving OPAL: OPAL is based on a set of assumptions about general patterns of web forms and OPAL domain schemas encode specific patterns for a domain. If the way forms appear on the web evolves, e.g., new widget types are introduced, OPAL's labeling heuristics may require adjustment, thought the combination of simple heuristics based on different feature sets has proved fairly robust to such changes. If the way forms appear in a specific domain evolves, e.g., new attributes are introduced, the corresponding domain schema must be adapted. Given a set of evolution operations on a standard domain ontology, these can likely be translated into corresponding operations on the OPAL domain schema.

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 DIADEM, no. 246858. Giorgio Orsi has also been supported by the Oxford Martin School's grant no. LC0910-019.

References

1. S. Araujo, Q. Gao, E. Leonardi, and G.-J. Houben. Carbon: domain-independent automatic web form filling. In *Proc. Int'l. Conf. on Web Engineering (ICWE)*, pages 292–306, 2010.
2. Z. Bar-Yossef and M. Gurevich. Random sampling from a search engine's index. *J. ACM*, 55(5):24:1–24:74, 2008.
3. L. Barbosa and J. Freire. Siphoning hidden-web data through keyword-based interfaces. In *Proc. Brazilian Symp. on Database*, pages 309–321, 2004.
4. L. Barbosa and J. Freire. Combining classifiers to identify online databases. In *Proc. Int'l. World Wide Web Conf. (WWW)*, pages 431–440, 2007.
5. M. Benedikt, G. Gottlob, and P. Senellart. Determining relevance of accesses at runtime. In *Proc. Symp. on Principles of Database Systems (PODS)*, pages 211–222, 2011.
6. M. Benedikt and C. Koch. XPath leashed. *ACM Computing Surveys*, pages 3:1–3:54, 2007.
7. A. Bilke and F. Naumann. Schema matching using duplicates. In *Proc. Int'l. Conf. on Data Engineering (ICDE)*, pages 69–80, 2005.
8. M. J. Cafarella, E. Y. Chang, A. Fikes, A. Y. Halevy, W. C. Hsieh, A. Lerner, J. Madhavan, and S. Muthukrishnan. Data management projects at google. *Sigmod Records*, 37(1):34–38, 2008.
9. K. C.-C. Chang, B. He, and Z. Zhang. Mining semantics for large scale integration on the web: evidences, insights, and challenges. *SIGKDD Explor. Newsl.*, 6(2):67–76, Dec. 2004.
10. W. Crescenzi, P. Merialdo, and D. Qiu. A framework for learning web wrappers from the crowd. In *Proc. Int'l. World Wide Web Conf. (WWW)*, 2013. To appear.
11. E. C. Dragut, T. Kabisch, C. Yu, and U. Leser. A hierarchical approach to model web query interfaces for web source integration. In *Proc. Int'l. Conf. on Very Large Data Bases (VLDB)*, pages 325–336, 2009.
12. E. C. Dragut, W. Meng, and C. T. Yu. *Deep Web Query Interface Understanding and Integration*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2012.
13. T. Furche, G. Gottlob, G. Grasso, X. Guo, G. Orsi, and C. Schallhart. Opal: automated form understanding for the deep web. In *Proc. Int'l. World Wide Web Conf. (WWW)*, pages 829–838, 2012.
14. T. Furche, G. Gottlob, G. Grasso, G. Orsi, C. Schallhart, and C. Wang. Little Knowledge Rules The Web: Domain-Centric Result Page Extraction. In *Proc. Int'l. Conf. on Web Reasoning and Rule Systems (RR)*, pages 61–76, 2011.
15. B. He, Z. Zhang, and K. C.-C. Chang. Towards building a metaquerier: Extracting and matching web query interfaces. In *Proc. Int'l. Conf. on Data Engineering (ICDE)*, pages 1098–1099, 2005.
16. H. He, W. Meng, Y. Lu, C. Yu, and Z. Wu. Towards deeper understanding of the search interfaces of the deep web. *Word Wide Web*, 10:133–155, 2007.
17. O. Kaljuvee, O. Buyukkokten, H. Garcia-Molina, and A. Paepcke. Efficient web form entry on pdas. In *Proc. Int'l. World Wide Web Conf. (WWW)*, pages 663–672, 2001.
18. R. Khare and Y. An. An empirical study on using hidden markov model for search interface segmentation. In *Proc. Int'l. Conf. on Information and Knowledge Management (CIKM)*, pages 17–26, 2009.
19. R. Khare, Y. An, and I.-Y. Song. Understanding deep web search interfaces: A survey. *Sigmod Records*, 39(1):33–40, 2010.
20. J. Lehmann, T. Furche, G. Grasso, A.-C. N. Ngomo, C. Schallhart, A. Sellers, C. Unger, L. Bühmann, D. Gerber, D. L. Konrad Höffner and, and S. Auer. Deqa: Deep web extraction for question answering. In *Proc. Int'l. Semantic Web Conf. (ISWC)*, 2012.
21. J. Madhavan, D. Ko, L. Kot, V. Ganapathy, A. Rasmussen, and A. Halevy. Google's deep web crawl. In *Proc. Int'l. Conf. on Very Large Data Bases (VLDB)*, pages 1241–1252, 2008.
22. A. Maiti, A. Dasgupta, N. Zhang, and G. Das. Hdsampler: revealing data behind web form interfaces. In *Proc. Symp. on Management of Data (SIGMOD)*, pages 1131–1134, 2009.
23. I. Navarrete, A. Morales, M. Cardenas, and G. Sciavicco. Spatial reasoning with rectangular cardinal relations - the convex tractable subalgebra. In *Annals of Mathematics and Artificial Intelligence*, 2012.
24. H. Nguyen, T. Nguyen, and J. Freire. Learning to extract form labels. In *Proc. Int'l. Conf. on Very Large Data Bases (VLDB)*, pages 684–694, 2008.
25. T. H. Nguyen, H. Nguyen, and J. Freire. PruSM: a prudent schema matching approach for web forms. In *Proc. Int'l. Conf. on Information and Knowledge Management (CIKM)*, pages 1385–1388, 2010.
26. F. Niu, C. Zhang, C. Re, and J. Shavlik. DeepDive: Web-scale knowledge-base construction using statistical learning and inference. In *Proc. Very Large Data Search (VLDS)*, pages 25–28, 2012.
27. T. Pedersen, S. Patwardhan, and J. Michelizzi. Wordnet::similarity - measuring the relatedness of concepts. In *Proc. HLT-NAACL–Demonstrations*, pages 38–41, 2004.
28. S. Raghavan and H. Garcia-Molina. Crawling the hidden web. In *Proc. Int'l. Conf. on Very Large Data Bases (VLDB)*, pages 129–138, 2001.
29. D. Shestakov, S. Bhowmick, and E. Lim. Deque: querying the deep web. *Data & Knowledge Engineering (DKE)*, 52(3):273–311, 2005.
30. W. Su, J. Wang, and F. H. Lochovsky. ODE: Ontology-assisted data extraction. *ACM Transactions on Database Systems*, 34(2):12:1–12:35, 2009.
31. J. Wang and F. H. Lochovsky. Data extraction and label assignment for web databases. In *Proc. Int'l. World Wide Web Conf. (WWW)*, pages 187–196, 2003.
32. J. Wang, J.-R. Wen, F. Lochovsky, and W.-Y. Ma. Instance-based schema matching for web databases by domain-specific query probing. In *Proc. Int'l. Conf. on Very Large Data Bases (VLDB)*, pages 408–419, 2004.
33. W. Wu, A. Doan, C. Yu, and W. Meng. Modeling and extracting deep-web query interfaces. In *Advances in Information & Intelligent Systems*, pages 65–90, 2009.
34. W. Wu, C. T. Yu, A. Doan, and W. Meng. An interactive clustering-based approach to integrating source query interfaces on the deep web. In *Proc. Symp. on Management of Data (SIGMOD)*, pages 95–106, 2004.
35. X. Yuan, H. Zhang, Z. Yang, and Y. Wen. Understanding the search interfaces of the deep web based on domain model. In *Proc. Int'l Conf. on Computer and Information Science*, pages 1194–1199, 2009.
36. Z. Zhang, B. He, and K. C.-C. Chang. Understanding web query interfaces: Best-effort parsing with hidden syntax. In *Proc. Symp. on Management of Data (SIGMOD)*, 2004.