Combining Form and Function: Static Types for JQuery Programs*

Benjamin S. Lerner, Liam Elberty, Jincheng Li, and Shriram Krishnamurthi

Brown University

Abstract. The jQuery library defines a powerful *query language* for web applications' scripts to interact with Web page content. This language is exposed as jQuery's API, which is implemented to fail silently so that incorrect queries will not cause the program to halt. Since the correctness of a query depends on the structure of a page, discrepancies between the page's actual structure and what the query expects will also result in failure, but with no error traces to indicate where the mismatch occurred. This work proposes a novel type system to statically detect jQuery errors. The type system extends Typed JavaScript with *local structure* about the page and with *multiplicities* about the structure of containers. Together, these two extensions allow us to track precisely which nodes are active in a jQuery object, with minimal programmer annotation effort. We evaluate this work by applying it to sample real-world jQuery programs.

1 Introduction

Client-side web applications are written in JavaScript (JS), using a rich, but low-level, API known as the Document Object Model (DOM) to manipulate web content. Essentially an "assembly language for trees", these manipulations consist of selecting portions of the document and adding to, removing, or modifying them. Like assembly language, though, programming against this model is imperative, tedious and error-prone, so web developers have created JS libraries that abstract these low-level actions into higher-level APIs, which essentially form their own domain-specific language.

We take as a canonical example of this effort the *query portion* of the popular jQuery library, whose heavily stylized form hides the imperative DOM plumbing under a language of "sequences of tree queries and updates". This query-language abstraction is widely used: other JS libraries, such as Dojo, Yahoo!'s YUI, Ember.js and D3, either embed jQuery outright or include similar query APIs. (All these libraries also include various features and functionality not related to document querying; these features differ widely between libraries, and are not our focus here.) JQuery in particular enjoys widespread adoption, being used in over half of the top hundred thousand websites [4].

From a program-analysis standpoint, jQuery can also drastically simplify the analysis of these programs, by directly exposing their high-level structure. Indeed, "jQuery programs" are written so dramatically differently than "raw DOM

* This work is partially supported by the US National Science Foundation and Google.

programs" that in this work, we advocate understanding jQuery as a language in its own right. Such an approach lets us address two essential questions. First, what bugs are idiomatic to this high-level query-and-manipulation language, and how might we detect them? And second, if a jQuery-like language were designed with formal analyses in mind, what design choices might be made differently?

Contributions This work examines *query errors*, where the result of a query returns too many, too few, or simply undesired nodes of the document. We build a type system to capture the behavior of jQuery's document-query and manipulation APIs; our goal is to expose all query errors as type errors. (Our type system is not *jQuery*-specific, however, and could be applied to any of the query languages defined by JS libraries.) In designing this type system, we find a fundamental tension between the flexibility of jQuery's APIs and the precision of analysis, and propose a small number of new APIs to improve that loss of precision.

At a technical level, this paper extends prior work [10, 15, 17] that has built a sophisticated type system for JS, to build a domain-specific *type* language for analyzing jQuery. Our type system enhances the original with two novel features:

- *Multiplicities*, a lightweight form of indexed types [26] for approximating the sizes of container objects; and
- *Local structure*, a lightweight way to inform the type system of the "shape" of relevant portions of the page, obviating the need for a global page schema.

Combined, these two features provide a lightweight dependent type system that allows us to typecheck sophisticated uses of jQuery APIs accurately and precisely, with minimal programmer annotation. Our prototype implementation is available at `https://jswebtools.org/jQuery/`.[1]

We introduce a running example (Section 2) to explain the key ideas of our approach (Sections 3 and 4), and highlight the key challenges in modeling a language as rich as jQuery (Sections 5 and 6). We evaluate its flexibility using several real-world examples (Section 7), and describe related work and directions for future improvements (Sections 8 and 9).

2 Features and Pitfalls of the JQuery Language

JQuery's query APIs break down into three broad groups: *selecting* an initial set of nodes in the page, *navigating* to new nodes in the document relative to those nodes, and *manipulating* the nodes. These functions allow the programmer to process many similar or related nodes easily and uniformly. To illustrate most of these APIs and their potential pitfalls, and as a running example, we examine a highly simplified Twitter stream. A stream is simply a ⟨**div**/⟩ element containing tweets. Each tweet in turn is a ⟨**div**/⟩ element containing a timestamp, an author, and the tweet content:

[1] We were unable to submit this artifact for evaluation per that committee's rules, because our fourth author co-chaired that committee.

⟨**div** class="stream"⟩
  ⟨**div** class="tweet"⟩
    ⟨**p** class="time"/⟩ ⟨**p** class="author"/⟩ ⟨**p** class="content short"/⟩
  ⟨/**div**⟩
  ...
⟨/**div**⟩

Selecting Nodes The jQuery `$` function is the entry point for the entire API. This function typically takes a CSS selector string and returns a jQuery object containing *all* the elements in the document that match the provided selector:

```
1 $("*.time").css('color', 'red'); // Colors all timestamp elements
```

But the power and flexibility of this method may lead to silent errors. *How do we assure ourselves, for example, that `$("*.time")` matches any elements at all?*

Navigating Between Nodes JQuery supplies several methods for relative movement among the currently selected nodes. These methods operate uniformly on all the nodes contained in the current collection:

```
1  // Returns a collection of the children of aTweet:
2  //   a timestamp, an author, and a content node
3  $(aTweet).children();
4  // Returns all the children of all tweets
5  $(".tweet").children();
6  // Returns the authors and contents of all tweets
7  $(".tweet").children().next();
8  // Returns the empty collection
9  $(".tweet").children().next().next().next().next();
10 // Meant to colorize contents, but has no effect
11 $(".tweet").children().next().next().next().css("color", "red");
```

The first two examples are straightforward: requesting the `children()` of a node returns a collection of all its children, while calling `children()` on several nodes returns all their children. The next example (line 7) shows jQuery's silent error handling: calling `next()` on a collection of timestamps, authors and contents will return a collection of the next siblings of each node, even if some nodes (here, content nodes) have no next siblings. In fact, jQuery does not raise an error even when calling a method on an empty collection: on line 9, after the third call to `next()`, there are no remaining elements for the final call. The final example highlights why this might be a problem: The programmer intended to select the contents nodes, but called `next()` once too often. The call to `css()` then dutifully changed the styles of all the nodes it was given—i.e., none—and returned successfully. This is another important error condition: *how can we assure ourselves that we haven't "fallen off the end" of our expected content?*

Manipulating Content The purpose of obtaining a collection of nodes is to manipulate them, via methods such as `css()`. These manipulator functions all follow a common getter/setter pattern: for example, when called with one argument,

`css()` returns the requested style property of the given node. When supplied with two arguments, it sets the style property of the given nodes.

Note carefully the different pluralities of getters and setters. Getters implicitly apply only to the *first* node of the collection, while setters implicitly apply to *all* the nodes of the collection. This can easily lead to errors: while in many cases, the order of elements in a collection matches the document order, and while in many other cases, the collection only contains a single element, neither of these conditions is guaranteed. *How can we assure ourselves that the collection contains exactly the node we expect as its "first" child?*

JQuery provides a general-purpose `each()` method, that allows the programmer to map a function over every node in the collection. (This is useful when the necessary computation requires additional state that cannot be expressed as a simple sequence of jQuery API calls.) This API is one of the few places where it is possible to trigger a run-time error while a jQuery function is on the call stack, precisely because the programmer-supplied function is *not* jQuery code. *How can we assure ourselves that the function calls only those methods on the element that are known to be defined?*

Chaining JQuery's API revolves around the *chaining* pattern, where practically every method (that is not a getter) is designed to return a jQuery object: manipulation APIs return their invocant, and navigation APIs return a new collection of the relevant nodes. This allows for fluid idioms such as the following, simplified code that animates the appearance of a new tweet on Twitter (the precise arguments to these functions are not relevant; the "style" of API calls is):

```
1 $(aTweet).fadeIn(...).css(...).animate(...);
```

Each call returns its invocant object, so that subsequent calls can further manipulate it. Modeling this idiom precisely is crucial to a useful, usable type system.

Choosing a Type System Though jQuery dynamically stifles errors, in this paper we argue that query errors are latent and important flaws that should be detected and corrected—statically, whenever possible. Accordingly, we aim to develop a type system that can capture the behaviors described above, and that can catch query errors without much programmer overhead. In particular, we would like type inference to work well, as any non-inferred types must be grudgingly provided by the programmer instead.

Because the behavior of a query depends crucially on the query parameters, the navigational steps, and any manipulations made, it would seem reasonable that the type of the query must depend on these values too, leading to a dependent type system whose types include strings and numbers to encode the query and its size. Unfortunately, in general, type inference for dependent type systems is undecidable, leading to large annotation or proof burdens on the programmer.

However, as we will see below, a type system tailored for jQuery can forego much of the complexity of general-purpose dependent types. By carefully restricting our use of strings, and by approximating our use of numbers, we can regain a "lightweight" dependently typed system with sufficient precision for our purposes, and that still enjoys decidable type inference in practice.

Comparisons with other query languages Describing jQuery as "a query language" invites comparison with other tree-query languages, most notably XPath and XQuery [23, 24], and the programming languages XDuce [11] and ℂDuce [2]. We compare our approach with XDuce and ℂDuce more thoroughly in Section 8; for now, we note that unlike XML databases that conform to schemas and from which strongly-typed data can be extracted, HTML pages are essentially schema-free and fundamentally presentational. JQuery therefore adopts CSS selectors, the language used to style web pages, as the basis of its query language.

3 Multiplicities

In this and the following section, we describe the novel features of our type system, and build up the central type definition in stages. Our system is an extension of Typed JavaScript [10], which provides a rich type system including objects types respecting prototype inheritance, (ordered) intersection and (unordered) union types, equi-recursive types, type operators and bounded polymorphism. The type language, including our new constructs, is shown in Fig. 1; we will introduce features from this type system as needed in our examples below.

Each of the three phases of the jQuery API described above induces a characteristic problem: in reverse order, ensuring that

1. Callback code only calls appropriate methods on the provided elements,
2. Precisely the intended target element is the first element of the jQuery object,
3. Navigation does not overshoot the end of known content, and
4. Selectors return the intended elements from the page.

Warmup The first challenge is a traditional type-checking problem, and one that is well handled by existing Typed JavaScript. We simply must ensure that the supplied callback function has type `SomeElementType -> Undef`, provided we know which `SomeElementType` is expected by the current jQuery object. This leads to a (very simplistic) first attempt at "the jQuery type":

```
1 type jQuery = μ jq :: * => * .
2   Λ e :: *  . {
3     each : ['jq⟨'e⟩] ('e -> Undef) -> 'jq⟨'e⟩
4   }
```

In words, this defines `jQuery` to be a type constructor (of kind `* => *`, i.e., a function from types to types) that takes an element type (line 2) and returns an object with a field `each` (line 3), which is a function that must be invoked on a `jQuery` object containing elements of the appropriate type (the type in brackets) and passed a single callback argument (in parentheses) and returns the original jQuery object. So that the type of `each` can refer to the overall jQuery type, we say that `jQuery` is in fact a recursive type (line 1). The `'e` type parameter records the type of the elements currently wrapped by the jQuery object.

$$
\begin{array}{rrl}
\alpha \in & & \textit{Type and multiplicity variables} \\
\kappa \in & \textit{Kind} & ::= * \mid \texttt{M}\langle * \rangle \\
r \in & \textit{RegEx} & ::= \textit{regular expressions} \\
\tau \in & \textit{Type} & ::= \alpha \mid \texttt{Num} \mid r \mid \texttt{True} \mid \texttt{False} \mid \texttt{Undef} \mid \texttt{Null} \mid \top \mid \bot \\
 & & \quad \mid \texttt{ref}\ \tau \mid [\tau]\tau \times \cdots \to \tau \mid \tau\texttt{+}\tau \mid \tau\texttt{\&}\tau \mid \mu\alpha.\tau \mid \{\star : \tau, s : \tau, \ldots\} \\
 & & \quad \mid \forall\alpha \leq \tau.\tau \mid \forall\alpha :: \kappa.\tau \mid \Lambda\ \alpha :: \kappa.\tau \mid \tau\langle\tau\rangle \mid \tau\langle m\rangle \\
 & & \quad \mid \tau\ \texttt{@}\ \textit{CSS selector} \\
m \in & \textit{Mult} & ::= \texttt{M}\langle\tau\rangle \mid \texttt{0}\langle\texttt{m}\rangle \mid \texttt{1}\langle\texttt{m}\rangle \mid \texttt{01}\langle\texttt{m}\rangle \mid \texttt{1+}\langle\texttt{m}\rangle \mid \texttt{0+}\langle\texttt{m}\rangle \mid m_1 \mathbin{+\!\!+} m_2 \\
l \in & \textit{LS} & ::= (\langle \textit{Name} \rangle\ \texttt{:}\ \langle \textit{ElementType} \rangle \\
 & & \qquad \texttt{classes=[...] optional classes=[...] ids=[...]} \\
 & & \qquad l\texttt{...)} \\
 & & \quad \mid \texttt{<}\textit{Name}\texttt{>} \mid \texttt{<}\textit{Name}\texttt{>+} \mid \texttt{...}
\end{array}
$$

Fig. 1: Syntax of types, multiplicities and local structure

The Need for More Precision The next two challenges cannot be expressed by traditional types. We need to keep track of *how many* elements are present in the current collection, so that we know whether calling a getter function like `css()` is ambiguous, or whether calling a navigation function like `next()` has run out of elements. (We defer the challenge of knowing exactly what type comes `next()` after a given one until Section 4; here we just track quantities.)

3.1 Defining Multiplicities

JQuery's APIs distinguish between zero, one and multiple items. To encode this information, we introduce a new kind that we call *multiplicities*, written $\texttt{M}\langle * \rangle$, with the following constructors:

$$
m \in \textit{Mult} ::= \texttt{M}\langle\tau\rangle \mid \texttt{0}\langle\texttt{m}\rangle \mid \texttt{1}\langle\texttt{m}\rangle \mid \texttt{01}\langle\texttt{m}\rangle \mid \texttt{1+}\langle\texttt{m}\rangle \mid \texttt{0+}\langle\texttt{m}\rangle \mid m_1 \mathbin{+\!\!+} m_2
$$

The first of these constructors (`M`) embeds a single type into a multiplicity; for brevity, we often elide this constructor below. The next five assert the presence of zero, one, zero or one, one or more, or zero or more multiplicities. These multiplicities can be nested, but can be normalized by simple multiplication: for instance, $\texttt{01}\langle\texttt{1+}\langle\tau\rangle\rangle = \texttt{0+}\langle\tau\rangle$. (In words, zero or one groups of one or more τ is equal to zero or more τ.) The normalization table is listed in Fig. 2a.

The remaining multiplicity, the sum $m_1 \mathbin{+\!\!+} m_2$, asserts the presence of both m_1 and m_2, and is useful for concatenating two jQuery collections. Sum multiplicities do not normalize multiplicatively the way the previous ones do, but they can be approximated additively: for example, $\texttt{1}\langle\tau_1\rangle \mathbin{+\!\!+} \texttt{01}\langle\tau_2\rangle <: \texttt{1+}\langle\tau_1\texttt{+}\tau_2\rangle$. In words, if we have one τ_1 and zero or one τ_2, then we have one or more τ_1s or τ_2s (this union type is denoted by the plus symbol); the full rules are in Fig. 2b. This loses precision: the latter multiplicity also describes a collection of several

$\cdot\langle\cdot\rangle$	$\mathtt{0}\langle\tau\rangle$	$\mathtt{1}\langle\tau\rangle$	$\mathtt{01}\langle\tau\rangle$	$\mathtt{1+}\langle\tau\rangle$	$\mathtt{0+}\langle\tau\rangle$
$\mathtt{0}$	$\mathtt{0}\langle\tau\rangle$	$\mathtt{0}\langle\tau\rangle$	$\mathtt{0}\langle\tau\rangle$	$\mathtt{0}\langle\tau\rangle$	$\mathtt{0}\langle\tau\rangle$
$\mathtt{1}$	$\mathtt{0}\langle\tau\rangle$	$\mathtt{1}\langle\tau\rangle$	$\mathtt{01}\langle\tau\rangle$	$\mathtt{1+}\langle\tau\rangle$	$\mathtt{0+}\langle\tau\rangle$
$\mathtt{01}$	$\mathtt{0}\langle\tau\rangle$	$\mathtt{01}\langle\tau\rangle$	$\mathtt{01}\langle\tau\rangle$	$\mathtt{0+}\langle\tau\rangle$	$\mathtt{0+}\langle\tau\rangle$
$\mathtt{1+}$	$\mathtt{0}\langle\tau\rangle$	$\mathtt{1+}\langle\tau\rangle$	$\mathtt{0+}\langle\tau\rangle$	$\mathtt{1+}\langle\tau\rangle$	$\mathtt{0+}\langle\tau\rangle$
$\mathtt{0+}$	$\mathtt{0}\langle\tau\rangle$	$\mathtt{0+}\langle\tau\rangle$	$\mathtt{0+}\langle\tau\rangle$	$\mathtt{0+}\langle\tau\rangle$	$\mathtt{0+}\langle\tau\rangle$

(a) Normalization for simple multiplicities

$\lfloor m + n \rfloor$	$\mathtt{0}\langle\tau_1\rangle$	$\mathtt{1}\langle\tau_1\rangle$	$\mathtt{01}\langle\tau_1\rangle$	$\mathtt{1+}\langle\tau_1\rangle$	$\mathtt{0+}\langle\tau_1\rangle$
$\mathtt{0}\langle\tau_2\rangle$	$\mathtt{0}\langle\top\rangle$	$\mathtt{1}\langle\tau_1\rangle$	$\mathtt{01}\langle\tau_1\rangle$	$\mathtt{1+}\langle\tau_1\rangle$	$\mathtt{0+}\langle\tau_1\rangle$
$\mathtt{1}\langle\tau_2\rangle$	$\mathtt{1}\langle\tau_2\rangle$	$\mathtt{1+}\langle\tau_3\rangle$	$\mathtt{1+}\langle\tau_3\rangle$	$\mathtt{1+}\langle\tau_3\rangle$	$\mathtt{1+}\langle\tau_3\rangle$
$\mathtt{01}\langle\tau_2\rangle$	$\mathtt{01}\langle\tau_2\rangle$	$\mathtt{1+}\langle\tau_3\rangle$	$\mathtt{0+}\langle\tau_3\rangle$	$\mathtt{1+}\langle\tau_3\rangle$	$\mathtt{0+}\langle\tau_3\rangle$
$\mathtt{1+}\langle\tau_2\rangle$	$\mathtt{1+}\langle\tau_2\rangle$	$\mathtt{1+}\langle\tau_3\rangle$	$\mathtt{1+}\langle\tau_3\rangle$	$\mathtt{1+}\langle\tau_3\rangle$	$\mathtt{1+}\langle\tau_3\rangle$
$\mathtt{0+}\langle\tau_2\rangle$	$\mathtt{0+}\langle\tau_2\rangle$	$\mathtt{1+}\langle\tau_3\rangle$	$\mathtt{0+}\langle\tau_3\rangle$	$\mathtt{1+}\langle\tau_3\rangle$	$\mathtt{0+}\langle\tau_3\rangle$

(b) Simplification of sum multiplicities, where $\tau_3 = \tau_1 + \tau_2$, and m and n are normalized

Fig. 2: Normalization and simplification for multiplicities: simplifying sums assumes that the arguments have first been normalized

τ_2s and zero τ_1s, while the original sum does not. Our type-checking algorithm therefore avoids simplifying sums whenever possible.

Note that while types describe expressions or values, multiplicities do not directly describe anything. Instead, they are permitted solely as arguments to type functions:[2] they are a lightweight way to annotate container types with a few key pieces of size information. In particular, they are easier to use than typical dependent types with arbitrary inequality constraints, as they can be manipulated syntactically, without needing an arithmetic solver.

We can now use these multiplicities to refine our definition for `jQuery`:

```
type jQuery = μ jq :: M⟨*⟩ => * .
  Λ m :: M⟨*⟩  . {
    each : ∀ e <: Element .
           ['jq⟨0+⟨'e⟩⟩] ('e -> Undef) -> 'jq⟨'m⟩
    css : ∀ e <: Element .
          (['jq⟨1⟨'e⟩⟩] Str -> Str &
           ['jq⟨1+⟨'e⟩⟩] Str*Str -> 'jq⟨1+⟨'e⟩⟩)
  }
```

The kind for `jq` has changed to accept a multiplicity, rather than a bare type. The type for `each` has also changed to recover the type `e` describing elements, which we bound above by `Element`. Multiplicities also give us sufficient precision to describe the type of `css()`: it is an overloaded function (i.e., it can be invoked at more than one type, which we model using intersection types $\tau_1 \& \tau_2$ [18]) where the getter must be invoked on a jQuery object containing *exactly one* element, and where the setter must be invoked on a jQuery object containing *at least one* element. This completely solves the ambiguity problem: a getter cannot be ambiguous if it is invoked on exactly one target.

Additionally, it partially resolves the "falling-off" problem: with the additional machinery of Section 4, we can ensure that `$(aTweet).children().next().next().next()` will have type `jQuery⟨0⟨Element⟩⟩`, which means that attempt-

[2] This is exactly analogous to the distinction between types and ordinals in indexed types [1, 26], or between types and sequence types in XDuce [11]; see Section 8.

$m_1 <: m_2$	$\mathtt{0}\langle\tau\rangle$	$\mathtt{1}\langle\tau\rangle$	$\mathtt{01}\langle\tau\rangle$	$\mathtt{1+}\langle\tau\rangle$	$\mathtt{0+}\langle\tau\rangle$
$\mathtt{0}\langle\tau\rangle$	✓*		✓		✓
$\mathtt{1}\langle\tau\rangle$		✓	✓	✓	✓
$\mathtt{01}\langle\tau\rangle$			✓		✓
$\mathtt{1+}\langle\tau\rangle$				✓	✓
$\mathtt{0+}\langle\tau\rangle$					✓

$$\text{M-Typ}\ \frac{\tau <: \tau'}{\mathtt{M}\langle\tau\rangle <: \mathtt{M}\langle\tau'\rangle} \qquad \text{M-Sums}\ \frac{(m_1 <: m_3 \wedge m_2 <: m_4) \vee (m_1 <: m_4 \wedge m_2 <: m_3)}{m_1 +\!\!+ m_2 <: m_3 +\!\!+ m_4}$$

$$\text{M-Sum-L}\ \frac{\lfloor m_1 +\!\!+ m_2 \rfloor <: m_3}{m_1 +\!\!+ m_2 <: m_3} \qquad \text{M-Sum-R}\ \frac{m_1 <: m_2 \vee m_1 <: m_3}{m_1 <: m_2 +\!\!+ m_3}$$

Fig. 3: Subtyping rules for simple multiplicities (left) and sums (right). * This case actually allows τ to differ in m_1 and m_2

ing to call `css()` will result in a static type error: intuitively, $\mathtt{0}\langle\mathtt{Element}\rangle$ is not a subtype of $\mathtt{1}\langle\mathtt{'e}\rangle$ for any possible type `'e`.

3.2 Subtyping Multiplicities

To make our notion of multiplicities sufficiently flexible, we need to define a "sub-multiplicity" relation, analogous to subtyping, that defines when one multiplicity is more general than another. The definition is largely straightforward: most of the primitive constructors should clearly subtype covariantly, e.g., $\mathtt{1}\langle\tau\rangle <: \mathtt{1}\langle\tau'\rangle$ if $\tau <: \tau'$, and $\mathtt{1}\langle\tau\rangle <: \mathtt{01}\langle\tau\rangle$. The one exception is that $\mathtt{0}\langle\tau_1\rangle <: \mathtt{0}\langle\tau_2\rangle$ for any types, since in both cases we have nothing.

Sum multiplicities are trickier to subtype. In particular, unlike the simpler multiplicities, the size of a sum is not immediately obvious; instead both arguments must be normalized, and then the whole sum simplified. The rules for sum multiplicities are shown in Fig. 3. Our typechecker uses M-Sums instead of M-Sum-L if possible, to avoid the loss of precision in normalizing sums.

4 Local Structure

Given multiplicities from the previous section, we now must assign types to the navigation functions (e.g., `next()`) and the jQuery `$` function itself such that they produce the desired multiplicities. One heavyweight approach might be to define a full document schema, as in XDuce [11], and then the types for navigation functions are easily determined from that schema (the selection function is still non-trivial). But this is impossible for most web content, for two reasons. First and foremost, a page may include arbitrary third-party content, and thus its overall schema would not statically be known. Second, even if all HTML were well-formed, HTML markup is not schematically rich: many tags define generic presentational attributes (e.g., "list item", "paragraph", or "table cell"), rather than semantic information (e.g., "tweet author", or "timestamp").

Clearly, global page structure is too onerous a requirement. But abandoning all structure is equally extreme: certainly, developers expect the page to possess some predictable structure. In this section, we propose a lighter-weight, local

alternative to global page schemas, and explain what changes are needed in our type language to incorporate it. We then explain how to use this local structure to give precise types to the jQuery navigation APIs. Finally, we show how to obtain local structure types from the type of the `$()` method itself.

4.1 Defining Structure Locally

Local structure allows developers to define the shape of only those sections of content that they intend to access via jQuery. For instance, our running example would be defined as follows:

```
(Tweet : DivElement classes = [tweet] optional classes = [starred]
  (Time : PElement classes = [time])
  (Author : PElement classes = [author] optional classes = [starred])
  (Content : PElement classes = [content short]))
(Stream : DivElement classes = [stream]
  <Tweet>+) // One or more Tweets as defined above
```

This local structure declaration defines five types: `Tweet`, `Time`, `Author`, `Content`, and `Stream`. Moreover, these declarations imply several local invariants between code and element structure:

- Each type implies structural information about an element: for example, a `Tweet` is a `DivElement` that is required to have class `.tweet`.
- Type membership can be decided by the declared classes: for example, at runtime, *every* element in the document with class `.tweet` is in fact a `Tweet`, and *no other elements* are permitted to have class `.tweet`. *Any* class by itself suffices to assign a type to an element: for example, `Content` elements can be identified by either `.content` or `.short`.
- Classes need not be unique: `starred` identifies either `Author`s or `Tweet`s. (This weakens the previous invariant slightly; a single class now may identify a set of types, all of which include that class in their declarations.)
- "Classes" are mandatory: `Content` elements will have both `.content` and `.short` classes. (Combined with the previous invariants, for example, all elements with class `.content` are `Content`s and therefore will also have class `.short`.) "Optional classes" and "ids" may or may not be present on elements at runtime.

The full syntax of local structure declarations is given in Fig. 1. Besides the explicit structure seen above, we support two other declarations. First, for legibility and reuse, types may be referenced by name from other structures: a `Stream` consists of one or more `Tweet`s, as indicated by reference `<Tweet>` and the repetition operator.[3] Second, the `...` construction (not used in this example) indicates the presence of one or more subtrees of no interest to the program's queries (for instance, a document might contain elements for purely visual purposes, that need never be selected by a query).

[3] We do not yet support the other regular expression operators $*$ and `?`, though they pose no fundamental difficulty.

Incorporating Local Structure as Types From the perspective of our type language, `Author` is essentially a refinement of the `PElement` type: in addition to the tag name, we know more precisely what selectors might match it. To record these refinements, we add to our type language a new *CSS refinement type* constructor: `Author = PElement @ div.stream>div.tweet>p.time+p.author`, that includes both the base element type (`PElement`) as well as precise information (the `CSS` selector) indicating where `Author`s can be found relative to other known local structures.

To be useful, we must extend the subtyping relation to handle these `CSS` refinement types. First, note that the refinements may be dropped, so that τ `@` s `<:` τ. For example, `Author <: PElement`. Second, we must decide when one `CSS` refinement is a subtype of another. Lerner [12] showed that set operations on `CSS` selectors are decidable; in particular, it is decidable whether for all documents, the nodes matched by one selector are a subset of the nodes matched by another. Accordingly, τ_1 `@` s `<:` τ_2 `@` t holds whenever $\tau_1 <: \tau_2$ and $s \subseteq t$.

4.2 Using Local Structure

The `CSS` refinement types above exploit only part of the information available from local structure definitions. In particular, they do not capture the structural relations *between* elements. For instance, the Twitter stream definition above also implies that:

- A `Tweet`'s children are each of `Time`, `Author` and `Content`. A `Tweet`'s parent is a `Stream`.
- A `Stream`'s children are all `Tweet`s; a `Stream`'s parent is unknown.
- `Time`s, `Author`s and `Content`s have no children, and have a `Tweet` parent.
- A `Time` has an `Author` as its next sibling, and has no previous sibling.
- An `Author` has a `Time` and a `Content` as its previous and next siblings.
- A `Content` has an `Author` as its previous sibling, and no next sibling.

These are precisely the relationships needed to understand the navigation functions in jQuery. Accordingly, we define four primitive type functions, `@children`, `@parent`, `@next` and `@prev`, whose definitions are pieced together from the local structure: in our example,

- `@children(Tweet)` $=$ `1⟨Time⟩` $+\!\!+$ `1⟨Author⟩` $+\!\!+$ `1⟨Content⟩`, and `@parent(Tweet)` $=$ `1⟨Stream⟩`.
- `@children(Stream)` $=$ `1+⟨Tweet⟩` and `@parent(Stream)` $=$ `01⟨Element⟩`.
- `@parent(Time)` $=$ `1⟨Tweet⟩`, `@next(Time)` $=$ `1⟨Author⟩`, `@prev(Time)` $=$ `@children(Time)` $=$ `0⟨Element⟩`, and likewise for `Author` and `Content`.

These functions clearly must be primitives in our system, since they are decidedly not parametric, and inspect the structure of their argument.

We may now enhance our jQuery type with navigation functions:

```
parent :   ∀ e <: Element, ['jq⟨1+⟨'e⟩⟩] -> 'jq⟨1+⟨@parent⟨'e⟩⟩⟩,
children : ∀ e <: Element, ['jq⟨1+⟨'e⟩⟩] -> 'jq⟨1+⟨@children⟨'e⟩⟩⟩,
next :     ∀ e <: Element, ['jq⟨1+⟨'e⟩⟩] -> 'jq⟨1+⟨@next⟨'e⟩⟩⟩,
prev :     ∀ e <: Element, ['jq⟨1+⟨'e⟩⟩] -> 'jq⟨1+⟨@prev⟨'e⟩⟩⟩,
```

These types, however, are not quite precise enough: if we have a jQuery object containing a single item, asking for its parent should return at most one item, but the types here ensure that we return many items instead. Worse, we may introduce imprecisions that easily could be avoided. In particular, suppose a developer defines two local structures:

```
(A : DivElement classes = [a]
  (B : DivElement classes = [b]))
(C : DivElement classes = [c]
  (D : DivElement classes = [d]))
```

JQuery supports an `add` function, which concatenates two collections into one for further processing across their combined elements. An appropriate type for this function is

```
add : ∀ n <: 0+⟨Element⟩ . ['jq⟨'m⟩] 'jq⟨'n⟩ -> 'jq⟨'m ++ 'n⟩
```

which neatly expresses the "combination" effect of the method. Using this function and the the rules above, the following code only typechecks under a loose bound (comments beginning with a colon are type assertions):

```
var bAndD = /*:jQuery⟨1+⟨B⟩ ++ 1+⟨D⟩⟩*/$(".b").add($(".d"));
var bdParents = bAndD.parent();
```

The type for `parent()` above requires its receiver to have type `jQuery⟨1+⟨'e⟩⟩`. The typechecker must therefore normalize `1+⟨B⟩ ++ 1+⟨D⟩` to `1+⟨B+D⟩`. Then, `@parent` receives `B+D` as its argument, and returns `1+⟨A+C⟩`. The resulting multiplicity describes one or more `A`s or `C`s, but we might have done better: because `bAndD` is known to include both `B`s and `D`s, the code above returns one or more `A`s *and* one or more `C`s. Therefore, we define `@parent` (and the other primitive navigation type functions) over multiplicities directly, rather than over types:

```
parent :   ∀ n <: 1+⟨Element⟩, ['jq⟨'n⟩] -> 'jq⟨@parent⟨'n⟩⟩
children : ∀ n <: 1+⟨Element⟩, ['jq⟨'n⟩] -> 'jq⟨@children⟨'n⟩⟩
next :     ∀ n <: 1+⟨Element⟩, ['jq⟨'n⟩] -> 'jq⟨@next⟨'n⟩⟩
prev :     ∀ n <: 1+⟨Element⟩, ['jq⟨'n⟩] -> 'jq⟨@prev⟨'n⟩⟩
```

Now `@parent` can be passed the original sum multiplicity without approximation, and its returned multiplicities can be correspondingly more precise; the same holds for the other three functions.

This extra precision is particularly crucial when one of the multiplicities degenerates to zero. Consider the following short program:

```
1 var ts = /*:jQuery⟨1+⟨Time⟩⟩*/ ...;
2 var cs = /*:jQuery⟨1+⟨Content⟩⟩*/ ...;
3 var tsAndCs = /*:jQuery⟨1+⟨Time⟩⊹1+⟨Content⟩⟩*/ ts.add(cs);
4 var tsOrCs  = /*:jQuery⟨1+⟨Time+Content⟩⟩ */ tsAndCs; // looser type
5 var prevTsAndCs = tsAndCs.prev(); // Can be jQuery⟨1+⟨Author⟩⟩
6 var prevTsOrCs  = tsOrCs.prev();  // Must be jQuery⟨0+⟨Author⟩⟩
```

The combined collection `ts.add(cs)` can be given two types: one using sum-multiplicities and one using a normalized multiplicity with a union type. When calling `@prev(1+⟨Time⟩⊹1+⟨Content⟩)`, `@prev` can distribute over the sum and since we know that there exists at least one `Content` element, the result must contain at least one `Author`. But when calling `@prev(1+⟨Time+Content⟩)`, we might have a collection containing only `Time` elements and so the result collection may be empty.

4.3 Creating Local Structure with the `@selector` Function

The last remaining challenge is to ascribe a type to the `$` function itself. We start with two concrete examples, and then explain the general case. Finally, we explore some subtleties of our design.

CSS and the `@selector` Function Consider determining which local structures matches the query string `"* > .author"`. We examine the local structure for types that mention the class `.author`, finding just one in this case, namely `Author = PElement @ div.stream > div.tweet > p.time + p.author`. We must then check whether the offered selector `* > .author` and the structure's selector `div.stream > div.tweet > p.time + p.author` can ever describe the same element: this is an intersection test over selectors, and can be done easily [12]. In fact, `* > .author` does intersect with `Author`'s selector (because all `Author`s do have a parent element). Finally, recall that the local structure invariants in Section 4.2 implicitly assert that, by omission, all other local structures are asserted *not* to have class `.author`, and are therefore excluded. We can therefore conclude `@selector⟨"div > .author"⟩ = 1+⟨Author⟩`.

On the other hand, consider the selector `"div.author"`. We know that `"div.author"` does not intersect with `Author` (because an element cannot match both `"div"` and `"p"`), nor with anything else (because nothing else has class `".author"`), so `@selector⟨"div.author"⟩ = 0⟨Element⟩`.

As with the primitive navigation type functions, we can encapsulate this reasoning in another primitive function, `@selector`. Here we must pass the precise string value of the selector to the type function, so that it can be examined as needed. We exploit Typed `JS`'s refined string types: rather than grouping all strings together under type `String`, Typed `JS` uses regular expressions to define subtypes of strings [9]. (In fact, `String` is just an alias for the regular

expression `/.*/`, which matches all strings.) In particular, string literals can be given singleton regular expression types matching only that string, and so pass the string value into our type function.[4] Therefore, we might give the `$` function the general type

```
$ : ∀ s <: /.*/ . 's -> jQuery⟨@selector⟨'s⟩⟩
```

The full algorithm for `@selector`, then, parses the string argument as a `CSS` selector. When it fails, it simply returns `0⟨Element⟩`. It next searches through the local structure definitions for candidates with matching class names, and intersects the query selector with the selectors compiled from the candidates' local structure declarations (using the approach in [12]). The returned *type* of `@selector` is the union of those local structure types whose selectors intersect the query string. (But see Section 5 below for which *multiplicity* it should return.)

Note that we never expose the parsing and interpretation of strings as `CSS` queries directly, but only the result of comparing strings to the selectors induced by local structure. We also do not add type constructors to our type language that mimic `CSS` selector combinators; we instead use only the refinement types shown above. We chose this design to keep our type system loosely coupled from the precise query language being used; our only requirement is that query intersections and containment be decidable. If jQuery used another language, perhaps more expressive than `CSS`, we would only need to replace the `CSS` selector intersection and containment algorithms, and not need to change any other aspects of our type system.

Applying `@selector` to Overly-Broad Queries Defining the `@selector` function entails two crucial choices: how many items should it return when the query matches local structures, and how flexible should it be in matching selectors to local structure? The former question is quite subtle, and we address it in Section 5. The latter is more a matter of programmer expectations, and we can resolve it relatively easily.

In our Twitter stream example, what should the query `$("div > p")` return? It does not mention any local structure classes or ids, so we have three options:

1. We might return `0+⟨PElement @ div > p⟩`, because nothing in the selector matches any of the required classes or ids of the local structure.
2. We might return `1+⟨Time + Author + Content⟩`, because each of these three structure definitions match the query selector, even though none of the required classes are present in the query.
3. We might return the "compromise" union `0+⟨Time + Author + Content + (PElement @ div>p)⟩`, because the query might return either any of the declared structures, or any other ⟨**p**/⟩ elements with ⟨**div**/⟩s as parents.

[4] Note: we do not *require* the argument to the `$()` function to be a string literal; the types here admit any string expression. We attempt to parse that type as a `CSS` selector, and such precise types often only work to literals. Since in practice, the arguments we have seen usually *are* literals, this encoding most often succeeds.

Of the three options, the third is clearly the least appealing, as it is tantalizingly useful (including `Time`, `Author` and `Content`), while still not guaranteeing anything about the returned values' multiplicity or type. The second is the most precise and useful, but it is incorrect: there might well be other elements matching `div > p` that are not part of a `Tweet` structure. As a result, the only sound multiplicity we can return in this case is the first one, which has the side benefit of highlighting, by its obtuseness, the imprecision of the query selector.

Unfortunately, several of the examples we evaluated used queries of this vague and overly-broad form! An amateur jQuery programmer, upon reading these examples, might infer that query brevity is preferred over precision. To accommodate these examples, we define our selector function to return the first—sound—option above by default, but currently we provide a command-line flag to implement the second (unsound but useful) option instead. (But see Section 5 below for a non-flag-based solution.) We view the need for this option as an unfortunate symptom of web developers' current practice of writing jQuery code without co-developing a specification for that code.

5 Type-system Guarantees and Usability Trade-offs

JQuery provides four APIs for querying: `$` itself, that selects nodes in the document; `find()`, that takes a current query set and selects descendant nodes; `filter()`, that selects a subset of the current query set; and `eq()`, that selects the n^{th} element of the current query set. Each of these might return zero items at runtime, so the most natural types for them are:

```
1 $ : ∀ s <: String . 's -> jQuery⟨0+⟨@selector⟨'s⟩⟩⟩
2 filter : (∀ e <: Element, ['jq⟨0+⟨'e⟩⟩] ('e -> Bool) -> 'jq⟨0+⟨'e⟩⟩) &
3          (∀ s <: Str, ['jq⟨'m⟩] 's -> 'jq⟨0+⟨@filter⟨'m, 's⟩⟩⟩)
4 find : ∀ s <: Str, ['jq⟨'m⟩] 's -> 'jq⟨0+⟨@filter⟨@desc⟨'m⟩, 's⟩⟩⟩
5 eq : ∀ e <: Element, ['jq⟨1+⟨'e⟩⟩] Num -> 'jq⟨01⟨'e⟩⟩
```

(In these types, `@filter` is a variant of `@selector`, and `@desc` is the transitive closure of `@children`.) But these types are inconvenient: because they all include `0`, their return values cannot be chained with methods that require a non-zero multiplicity. Note that these zeroes occur *even if the query string matches local structure*. This is particularly galling for the `$` function, as after all, if a query string *can* match local structure definitions, then surely the developer can expect that it *will* actually select nodes at runtime! Worse, using this type effectively marks all jQuery chains as type errors, because the first API call after `$()` will expect a non-zero multiplicity as input. Likewise, developers may often expect that their `filter()` or `find()` calls will in fact find non-zero numbers of nodes.

Despite their precision, our types seem to have lost touch with developers' expectations. It appears the only way to make our types useful again is to change them so they no longer reflect the behavior of jQuery! What guarantee, then, does our type system provide, and how can we resolve this disparity between "expected" and actual jQuery behavior?

5.1 Type-system Guarantees

We have described our type system above as capturing jQuery's behavior "precisely", but have not formalized that claim. Recall that the traditional formal property of *soundness* asserts that a type system cannot "lie" and affirm that an expression has some type when in fact it does not, or equivalently, affirm that an expression will execute without error at runtime when in fact it will. Such soundness claims are necessarily made relative to the semantics of the underlying language. But jQuery is a language without a formal semantics, so what claims besides soundness can we make about our approach?

Our type system is based upon Typed JS [10], which has a formal soundness guarantee relative to JS semantics. We are therefore confident that our system correctly handles typical JS features. However, while our type system may be sound for JS, we might ascribe types to jQuery APIs that do not match their actual behavior. This would be a failing of our *type environment*, not of our type system. To produce plausibly correct types for jQuery's APIs, we have experimented with many examples to determine potential types, and constructed counterexamples to convince ourselves the types cause typing errors only in programs we expected to be wrong.

Of course, jQuery is in fact implemented in JS, and therefore it is conceivable that we might typecheck the implementation to confirm it has the types we have ascribed. As our experience with typechecking ADsafe [15] has shown, typechecking a library can be a complex undertaking in its own right; we leave typechecking the source of jQuery itself as future work we intend to pursue.

But as we have begun to see already in Section 4.3, even choosing appropriate types for jQuery involves aesthetic choices, trading off between strictness and static determination of errors on one hand, and flexibility and ease of development on the other. We now examine these trade-offs in more detail.

5.2 Accommodating Varied Developer Expectations

The useful output of a type system—its acceptance or rejection of a program, and the error messages that result—provides crucial feedback to developers to distinguish buggy code from incomplete code. But type checkers can only operate on the code that is actually written, and that may not always suffice to distinguish these two cases, particularly when they are syntactically identical.

Merely by writing queries, developers have codified implicit expectations about how the state of their program matches their code. In particular, they understand that because of the dynamic, stateful changes to page structure that make their application interactive, the size of a query's result may vary during execution: queries that might return several elements at one point of a program's execution might later return none, or vice versa. But buggy queries also may have unexpected sizes. Nothing syntactically distinguishes queries with anticipated size variability from ones that are buggy. Without this knowledge, we cannot reliably report type errors to developers. We explore two untenable approaches for resolving this ambiguity, and then explore two more palatable ones.

A Non-solution: Specialized Code One unsuccessful strawman approach might require programmers to write repetitive variants of code to deal with each potential query result size. This is untenable, as these variants clutter the code, make future code modifications difficult, and break from the compositional style of jQuery programs.

An Unsound "Solution": Assuming Presence Another approach might simply assert by fiat that any local structures defined by the author are always guaranteed to be present: effectively, this means removing the explicit `0+` in the output type of the query APIs. Under this assumption, the following query should always return `1⟨Tweet⟩` (recall the definition of `Tweet`s from Section 4):

```
$(".tweet").find(".starred").first() // Has type jQuery⟨1⟨Tweet⟩⟩
```

But the developer explicitly said `starred` only *might* be present! Further, it is laughably wrong in the face of page mutations:

```
$(".tweet").find(".starred").removeClass("starred");
$(".tweet").find(".starred").first() // Still has type jQuery⟨1⟨Tweet⟩⟩
```

Note that this assumption leads to incorrect types, in that it does not accurately reflect the behavior of `find()`. But this type does not break the soundness of the type system itself: it is entirely possible to write a method that always returns a non-zero number of elements—though to do so, one would have to use raw DOM APIs, and effectively make such a function a "primitive" operation with respect to the jQuery language.

On the other hand, these "incorrect" types are much more useful for the `$()` function: by allowing non-zero multiplicities, jQuery chains can be checked without immediately requiring annotation.

A Spectrum of Type Environments One seemingly simple possibility is simply to add "flags" to the type system indicating whether `find()` (above), `@selector` (as in Section 4.3), and many other functions should be strict or permissive about their multiplicities. We reject this approach. If the flags affect the core type language or the type checker itself, then any soundness or correctness claims of the type system can only be made relative to every possible combination of flags. As type checkers are already quite complicated, these flags are a poor engineering decision, especially when a much cleaner, modular alternative exists.

Rather than add these flags to the type *system*, we can compile a variety of *type environments* from local structure, that contain stricter or looser types for these functions. Unlike "flags", whose interpretation is hidden within the type system's implementation, these various environments are easily examined by the developer, who can see exactly how they differ. A developer choosing to use the looser environments is therefore making a conscious choice that certain error conditions are not important, or equivalently that he is willing for them to appear as runtime errors instead. Moreover, the developer may migrate from permissive to strict environments as his program matures and nears completion.

The Need for New APIs Our last approach requires actually enhancing the jQuery API. JQuery objects expose their size via the `length` field; we might use this to regain lost precision. We might force programmers to explicitly write if-tests on the `length` field, but this has the unwanted effect of breaking chaining. Instead, we propose new jQuery APIs:

```
assertNotEmpty : (∀ t, ['jq⟨01⟨'t⟩⟩] -> 'jq⟨1⟨'t⟩⟩) &
                 (∀ t, ['jq⟨0+⟨'t⟩⟩] -> 'jq⟨1+⟨'t⟩⟩)
ifZero : ∀ t, ['jq⟨0+⟨'t⟩⟩] (['jq⟨0⟨'t⟩⟩]->Undef) -> 'jq⟨0+⟨'t⟩⟩
ifOne  : ∀ t, ['jq⟨0+⟨'t⟩⟩] (['jq⟨1⟨'t⟩⟩]->Undef) -> 'jq⟨0+⟨'t⟩⟩
ifMany : ∀ t, ['jq⟨0+⟨'t⟩⟩] (['jq⟨1+⟨'t⟩⟩]->Undef) -> 'jq⟨0+⟨'t⟩⟩
```

The first API converts possibly-empty collections into definitely-non-empty ones, and throws a runtime error if the collection is in fact empty. Developers would use this API to indicate queries they expect should never fail. The example above would be rewritten

```
$(".tweet").find(".starred").removeClass("starred");
$(".tweet").find(".starred").assertNotEmpty().first();
```

This would typecheck successfully, and always throw an exception at runtime. By contrast, the latter three APIs take a receiver object of unknown multiplicity, and a callback that expects a argument of precise multiplicity, and calls it only if the receiver has the expected multiplicity. Developers would use these APIs to indicate computation they expect might not be needed. The example above would be rewritten

```
$(".tweet").find(".starred").removeClass("starred");
$(".tweet").find(".starred").ifMany(function() { this.first(); });
```

This again would typecheck successfully, and would not call the inner function at runtime. Unlike the previous API, these APIs never throw exceptions, but indicate to the type checker that the programmer knows that these calls—and these alone—might fail. These APIs implicitly perform a specialized form of occurrence typing [22], without needing any extra type machinery.

6 Modeling JQuery Reality

Real jQuery programs avail themselves of several additional query APIs. One of them requires revising the kind of our type, and leads to the final form of our type for jQuery. This change unfortunately makes the type more unwieldy to use, but we can resolve this with a simple type definition.

Merging and Branching We have already seen the `add()` operation, which allows combining two jQuery objects into one. Occasionally, a program might need the dual of this operation, "splitting" one jQuery into multiple subqueries. Consider the following two queries:

```
$(".tweet").find(".star").each(processStarredTweet);
$(".tweet").find(".self").each(processOwnTweet);
```

They both have the same prefix, namely `$(".tweet")`, and more generally this prefix might be an expensive computation. To avoid this, jQuery objects conceptually form a stack of their navigated states, where each navigation pushes a new state on the stack: programs may use a new API, `end()`, to pop states off this stack. The example above would be rewritten:

```
$(".tweet").find(".star").each(processStarredTweet)
           .end() // pops the find operation
           .find(".self").each(processOwnTweet);
```

Internally, jQuery objects form a linked list that implements this stack: the chainable jQuery methods that appear to modify the contents of the collection return a new jQuery object that wraps the new contents and whose tail is the old jQuery object. (The old object is sill available, and has not been mutated.) The `end()` method simply returns the tail of the list. In order to type `end()` correctly, we must encode the linked list in our types. To do so requires a systematic change in our type: we redefine the `jQuery` type constructor to take *two* type parameters, where the new second type parameter describes the tail of the stack. Our final type definition for jQuery is (only representative examples of jQuery's more than fifty query APIs are shown; the full type is available in our implementation):

```
type jQuery = μ jq :: (M⟨∗⟩, ∗) => ∗ .
  Λ m :: M⟨∗⟩, prev :: ∗  . {
    // Navigation APIs: next, prev, parent, etc. are analogous
    children : ∀ e <: 1+⟨Element⟩ .
           ['jq⟨'e, 'prev⟩] -> 'jq⟨@childrenOf⟨'e⟩, 'jq⟨'e, 'prev⟩⟩
    // Accessor APIs: offsetX, height, attr, etc. are analogous
    css : ∀ e <: Element .
          (['jq⟨1⟨'e⟩, 'prev⟩] Str -> Str &
           ['jq⟨1+⟨'e⟩, 'prev⟩] Str∗Str -> 'jq⟨1+⟨'e⟩, 'prev⟩)
    // Stack-manipulating APIs
    add : ∀ n <: 0+⟨Element⟩, 'q .
           ['jq⟨'m, 'prev⟩] 'jq⟨'n, 'q⟩ -> 'jq⟨'m++'n, 'jq⟨'m, 'prev⟩⟩
    end : ['jq⟨'m, 'prev⟩] -> 'prev
    // Collection-manipulating APIs
    each : ∀ e <: Element . ['jq⟨0+⟨'e⟩, 'prev⟩] ('e->Undef) -> 'jq⟨'m, 'prev⟩
    filter : (∀ e <: Element, ['jq⟨0+⟨'e⟩⟩] ('e->Bool) -> 'jq⟨0+⟨'e⟩⟩) &
             (∀ s <: Str, ['jq⟨'m⟩] 's -> 'jq⟨0+⟨@filter⟨'m, 's⟩⟩⟩)
    find : ∀ s <: Str, ['jq⟨'m⟩] 's -> 'jq⟨0+⟨@filter⟨@desc⟨'m⟩, 's⟩⟩⟩
    eq : ∀ e <: Element, ['jq⟨1+⟨'e⟩⟩] Num -> 'jq⟨01⟨'e⟩⟩
    // No other fields are present
    ⋆ : _
  }
```

Methods such as `add()` and `children()` nest the current type one level deeper, and `end()` simply unwraps the outermost type.

Convenience Often the suffix of the stack is not needed, so we can define a simple synonym for the supertype of all possible jQuery values:

```
type AnyJQ = ∀ p . jQuery⟨0+⟨Any⟩, 'p⟩
```

Any functions that operate over jQuery objects, and that will not use `end()` to pop off the jQuery stack any more than they push on themselves, can use `AnyJQ` to summarize the type stack easily.

7 Evaluation

To determine the flexibility of our type system, we analyzed a dozen samples from *Learning JQuery* [6] and from the Learning JQuery blog [20, 21]. Our prototype system[5] supports standard `CSS` selectors, but not pseudoselectors or the bespoke jQuery extensions to `CSS` syntax. In the examples below, we have edited the original code by replacing pseudoselectors and extensions with calls to their API equivalents and with calls to `ifMany()` (defined above). These added calls represent our understanding of how the examples should work; without them, the program's intended behavior is unclear. With these added calls, the examples below were checked *assuming presence for the `$()` function, but not for `find()` or `filter()`*. We distinguish three sets of results: those that should typecheck and do, those that should not typecheck and do not, and cases that expose weaknesses for our system.

7.1 Successfully Typechecking Type-correct Examples

Our samples consist of between 2 and 9 calls to jQuery functions in linear, branching, and nested call-patterns. These examples *each typecheck with no annotations needed on the code* besides local structure definitions that need only be written once. These local structures were *not defined* by the accompanying text; we derived them by manual inspection of the examples and their intended effects. We highlight three illustrative examples here.

Selecting All of a Row Our first example shows a straightforward selection of all ⟨**td**/⟩ tags in a table row and adding a class to them. Note that `addClass()` is called with a class name that is not declared in the local structure; our type system is not intended to restrict such code. Rather, it ensures that the call to `addClass()` is given a non-empty collection to work with. Note also that the initial query is over-broad (a better query would be `'td.title'`) and requires the command-line option for flexible query matching (of Section 4.3) to typecheck as written:

[5] Available at https://jswebtools.org/jQuery/

```
/*::
  (PlaysTable : TableElement ids = [playstable] classes = [plays]
    (PlaysRow : TRElement classes = [playsrow]
      (Title : TDElement classes = [title])
      (Genre : TDElement classes = [genre])
      (Year : TDElement classes = [year])
    )+ );
*/
$('td') // Has type t1 = jQuery⟨1+⟨Title⟩++1+⟨Genre⟩++1+⟨Year⟩, AnyJQ⟩
  .filter(':contains("henry")') // t2 = jQuery⟨0+⟨Title+Genre+Year⟩, t1⟩
  .ifMany(function() { // [jQuery⟨1+⟨Title+Genre+Year⟩, t1⟩] -> Undef
    this                      // t3 = jQuery⟨1+⟨Title+Genre+Year⟩, t1⟩
      .nextAll()              // t4 = jQuery⟨1+⟨Genre⟩ ++ 1+⟨Year⟩, t3⟩
      .andSelf()     // jQuery⟨1+⟨Genre⟩ ++ 1+⟨Year⟩ ++ 1+⟨Title⟩, t4⟩
      .addClass('highlight');       // allowed, non-empty collection
   });
```

Filtering by Ids This next example is a deliberately circuitous query, and illustrates several features both of jQuery and our type system. The call to `parents()` normally would return all the known parents of the current elements, up to and including the generic type `Element`, but because the query also filters on an id, and because that id appears in local structure, our type is much more tightly constrained. Similarly, the calls to `children()` and `find()` are constrained by the local structure. Note that if the call to `parent()` were *not* so constrained, then these subsequent calls would mostly be useless, since nothing is known about arbitrary `Element`s.

```
/*::
(SampleDiv : DivElement ids = [jqdt2] classes = [sampledivl]
  (Paragraph : PElement classes = [goofy]
    ...)          // The children of a Paragraph are irrelevant here
  (OrderedList : OLElement classes = [list]
    (LinkItem : LIElement classes = [linkitem]
      (Link : AElement classes = [link]))
    (GoofyItem : LIElement classes = [goofy]
      (StrongText : StrongElement classes = [strongtext]))
    (FunnyItem : LIElement classes = [funny])
    <LinkItem>
    <GoofyItem>));
*/
$('li.goofy')           // Has type t1 = jQuery⟨1+⟨GoofyItem⟩, AnyJQ⟩
  .parents('#jqdt2')                // t2 = jQuery⟨1+⟨SampleDiv⟩, t1⟩
  .children('p')                    // t3 = jQuery⟨1+⟨Paragraph⟩, t2⟩
  .next()                          // t4 = jQuery⟨1+⟨OrderedList⟩, t3⟩
  .find('a')                             // t5 = jQuery⟨1+⟨Link⟩, t4⟩
  .parent();                          // t6 = jQuery⟨1+⟨LinkItem⟩, t5⟩
```

Manipulating Each Element The following example demonstrates that our approach scales to higher-order functions: the callback passed to the `each()` operation (line 16) needs no annotation to be typechecked. The `ifMany()` calls could be eliminated using the unsound options described in Section 5. Note also the use of `end()` on line 14 to restore a prior query state.

```
1  /*::
2  (NewsTable : TableElement classes = [news] ids = [news]
3    ...                          // Placeholder element; never queried
4    (NewsBody : TBodyElement classes = [newsbody]
5      (YearRow : TDElement classes = [yearrow])
6      (NewsRow : TRElement classes = [newsrow] optional classes = [alt]
7        (NewsInfo : TDElement classes = [info])+)+
8      <YearRow>
9      <NewsRow>+))
10 */
11 $('#news')               // Has type t1 = jQuery⟨1⟨NewsTable⟩, AnyJQ⟩
12   .find('tr.alt')                    // t2 = jQuery⟨1+⟨NewsRow⟩, t1⟩
13   .ifMany(function() { this.removeClass('alt'); })             // t2
14   .end()                                                       // t1
15   .find('tbody')                     // t3 = jQuery⟨1⟨NewsBody⟩, t1⟩
16   .each(function() {                          // [NewsBody] -> Undef
17     $(this)                        // t4 = jQuery⟨1⟨NewsBody⟩, AnyJQ⟩
18       .children()      // t5 = jQuery⟨1+⟨YearRow⟩ ++ 1+⟨NewsRow⟩, t4⟩
19       .filter(':visible') // t6 = jQuery⟨0+⟨YearRow⟩++0+⟨NewsRow⟩, t5⟩
20       .has('td')                    // t7 = jQuery⟨0+⟨NewsRow⟩, t6⟩
21       .ifMany(function() {     // [jQuery⟨1+⟨NewsRow⟩, t6⟩] -> Undef
22         this.addClass('alt');      // allowed, non-empty collection
23       });                                                        // t7
24   });                                                          // t3
```

7.2 Successfully Flagging Type-incorrect Examples

The examples we examined from the *Learning JQuery* textbook and blog are typeable. To verify that our type-checker was not trivially passing all programs, we injected errors into these examples to ensure our system would correctly catch them, and it does. Our modifications changed queries to use the wrong element id, or have too many or too few navigational calls.

```
1  $('li.goofy')           // Has type t1 = jQuery⟨1+⟨GoofyItem⟩, AnyJQ⟩
2    .parents('#WRONG_ID') // t2 = jQuery⟨01⟨Element @ "#WRONG_ID"⟩, t1⟩
3    .children('p');
4          ⇒ ERROR: children expects 1+⟨Element⟩, got 01⟨Element⟩
5  $('#news')               // Has type t3 = jQuery⟨1⟨NewsTable⟩, AnyJQ⟩
6    .find('tr.alt')                    // t4 = jQuery⟨1+⟨NewsRow⟩, t3⟩
7    .ifMany(function() { this.removeClass('alt'); })             // t4
```

```
                                        // Note: missing call to .end()
  .find('tbody')                          // t5 = jQuery⟨0⟨Element⟩, t3⟩
  .each(...)
          ⇒ ERROR: each expects 1+⟨Element⟩, got 0⟨Element⟩
$(".tweet").children().next().next().next().css("color", "red");
          ⇒ ERROR: css expects 1+⟨Element⟩, got 0⟨Element⟩
$(".tweet").children().next().css("color");
          ⇒ ERROR: css expects 1⟨Element⟩, got 1+⟨Author + Time⟩
```

7.3 Weaknesses of Our System

Beyond the restrictions shown in the paper, and the trade-offs between soundness and utility of types, there still exist queries our system cannot handle. We construct one such instance here.

Under our running example, the expression `$(".tweet").parent()` will have the expected type `jQuery⟨1+⟨Stream⟩, AnyJQ⟩`. However, the similar expression `$(".stream").parent()` correctly has type `jQuery⟨0+⟨Element⟩, AnyJQ⟩`, because nothing is known about the parent of a `Stream`.

The expression `$("div.outer > *.stream").parent()` will at runtime return the ⟨**div** div="outer"/⟩ elements that contain streams (if any exist). Our current type system, however, will nevertheless give this expression the type `jQuery⟨0+⟨Element⟩, AnyJQ⟩`, even though *these* `Stream`s definitely have ⟨**div**/⟩s as parents. This is inherent in our semantics for local structure: developers are obligated to provide definitions for any content they intend to access via jQuery's APIs—beyond those definitions, the remainder of the page is unknown. One might re-engineer our system to dynamically refine the local structure-derived types to include the extra information above their root elements, but such complication both muddies the implementation and also confuses expectations of what the system can provide developers. Instead, developers must simply provide extra local structure information, and avoid the confusion entirely.

8 Related Work

XDuce and ℂDuce Languages such as XDuce [11, 19] and ℂDuce [2] embed an XML-processing language into a statically-typed general-purpose language, and extend the type language with document schemas. These languages differ from our jQuery setting in three crucial ways, all stemming from the fact that jQuery is a language for manipulating HTML that is embedded within JS.

First and foremost, XDuce and ℂDuce operate over well-schematized databases represented in XML, from which richly-typed structured data can be extracted. But HTML is not schematized: it is almost entirely free-form, and largely presentational in nature. As we argued in Section 4, mandating a global schema for HTML documents is an untenable requirement on developers. As such, XDuce and ℂDuce's approach cannot apply directly.

Second, XDuce and ℂDuce go to great lengths to support pattern matching over XML: in particular, their language of values includes *sequences* of tagged values, i.e. XML forests. XDuce and ℂDuce define *regular expression types* to precisely type these sequences. But jQuery does not have the luxury of enriching JS to support free-form sequences: instead, it encapsulates a sequence of values into a JS object. As the ℂDuce paper notes [2], regular expression types are not themselves types, but are only meaningful within the context of an element: this is exactly comparable to our kinding distinction between multiplicities and types.

Third, the operations native the HTML and DOM programming are simpler than those in XML processing: web programmers use CSS selectors [25] to query nodes in their documents, rather than XPath. Further, because JS has no pattern-matching construct, there is no need for an analysis of jQuery to define tree patterns or regular tree types (as in [11]), or check exhaustiveness of pattern matching (as in [5]). Instead, as we have shown, a simpler notion suffices.

Semanticizing the Web This work fits into a growing effort to design—or retrofit—semantics onto portions of the client-side environment, including JS [9, 14, 16], the event model [13], and the browser itself [3]. Some work has begun addressing the semantics of DOM manipulations [8], but it has focused on the low-level APIs and not on the higher-level abstractions that developers have adopted.

Structure and Size Types Other work [26] extends Haskell's type system with indexed types that describe sizes statically. Multiplicities specialize this model by admitting ranges in the indices. Our implementation also demonstrates how to use multiplicities without annotation burden, in the context of an important web library.

9 Future work: Interoperating with Non-jQuery Code

While our type system cleanly and accurately supports the query and navigation portions of jQuery, other challenges remain in analyzing the interactions between jQuery and raw JS code. We focus on two interactions: code written with types but without multiplicities, and code that might breach local structure invariants.

Relating Types and Multiplicities Multiplicities, as presented so far, are strictly segregated from types by the kinding system. However, suppose we had two nearly-identical list types, defined by client code and by a library:

```
1 type AMultList = Λ m :: M⟨∗⟩ . ...
2 type ATypeList = Λ t :: ∗ . ...
```

A value of type AMultList⟨1+⟨τ⟩⟩ cannot be used where one of type ATypeList⟨τ⟩ is expected, and yet any program expecting the latter would behave correctly when given the former: by its type, it clearly cannot distinguish between them.

However, upon returning from non-multiplicity-based code, we have to construct some multiplicity from a given type. Accordingly, we might add the two "subtype/multiplicity" rules

$$\textsc{Lax-TypMult}\ \frac{}{\tau <: \texttt{0+}\langle\tau\rangle} \qquad \textsc{Lax-MultTyp}\ \frac{}{\texttt{0+}\langle\tau\rangle <: \tau}$$

These two rules let us be "lax" about keeping multiplicities and types separate, without sacrificing soundness. Note that the utility of Lax-TypMult depends heavily on using occurrence typing to refine the resulting $\texttt{0+}\langle\tau\rangle$ multiplicity into something more precise. We have implemented these rules in our type checker, but they have not been needed in the examples we have seen.

Preserving Local Structure Our type for `jQuery` ensures only that developer-supplied callbacks typecheck with the appropriate receiver and argument types (see Section 3). This is an incomplete specification, as such excursions into low-level `JS` can easily be sources of bugs. We envision both dynamic and static solutions to this problem.

Dynamically, maintaining the local structure of a subset of a page amounts to a contract. The local structure definitions can easily be compiled into executable `JS` code that checks whether a given node conforms to particular local structure definitions, and these checks can be automatically wrapped around all jQuery APIs.

Statically, maintaining tree shapes may be analyzable by a tree logic [7]. The primary challenge is a reachability analysis identifying all the possible nodes that might be modified by arbitrary `JS`; we leave this entirely to future work.

References

[1] A. Abel. Polarized subtyping for sized types. *Mathematical Structures in Computer Science*, 18(5):797–822, Oct. 2008.

[2] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-centric general-purpose language. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2003.

[3] A. Bohannon and B. C. Pierce. Featherweight Firefox: formalizing the core of a web browser. In *USENIX Conference on Web Application Development (WebApps)*, 2010.

[4] BuiltWith. JQuery usage statistics. Retrieved Nov. 2012. `http://trends.builtwith.com/javascript/JQuery`.

[5] G. Castagna, D. Colazzo, and A. Frisch. Error mining for regular expression patterns. In *Italian conference on Theoretical Computer Science*, ICTCS'05, 2005.

[6] J. Chaffer and K. Swedberg. *Learning JQuery*. Packt Publishing Ltd., Birmingham, UK, 3rd edition, 2011.

[7] P. Gardner and M. Wheelhouse. Small specifications for tree update. In *international conference on Web services and formal methods*, WS-FM'09, 2010.

[8] P. A. Gardner, G. D. Smith, M. J. Wheelhouse, and U. D. Zarfaty. Local Hoare reasoning about DOM. In *ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, 2008.

[9] A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of JavaScript. In *European Conference on Object-Oriented Programming (ECOOP)*, 2010.

[10] A. Guha, C. Saftoiu, and S. Krishnamurthi. Typing local control and state using flow analysis. In *European Symposium on Programming Languages and Systems (ESOP)*, 2011.

[11] H. Hosoya and B. C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology (TOIT)*, 3(2):117–148, 2003.

[12] B. S. Lerner. *Designing for Extensibility and Planning for Conflict: Experiments in Web-Browser Design*. PhD thesis, University of Washington Computer Science & Engineering, Aug. 2011.

[13] B. S. Lerner, M. J. Carroll, D. P. Kimmel, H. Q. de la Vallee, and S. Krishnamurthi. Modeling and reasoning about DOM events. In *USENIX Conference on Web Application Development (WebApps)*, June 2012.

[14] S. Maffeis, J. C. Mitchell, and A. Taly. An operational semantics for JavaScript. In *Asian Symposium on Programming Languages and Systems (APLAS)*, 2008.

[15] J. G. Politz, S. A. Eliopoulos, A. Guha, and S. Krishnamurthi. ADsafety: type-based verification of JavaScript sandboxing. In *USENIX Security Symposium*, Aug. 2011.

[16] J. G. Politz, M. Carroll, B. S. Lerner, J. Pombrio, and S. Krishnamurthi. A tested semantics for getters, setters, and eval in JavaScript. In *Dynamic Languages Symposium (DLS)*, 2012.

[17] J. G. Politz, A. Guha, and S. Krishnamurthi. Semantics and types for objects with first-class member names. In *Workshop on Foundations of Object-Oriented Languages (FOOL)*, 2012.

[18] V. St-Amour, S. Tobin-Hochstadt, M. Flatt, and M. Felleisen. Typing the numeric tower. In *Practical Aspects of Declarative Languages (PADL)*, 2012.

[19] M. Sulzmann and K. Z. M. Lu. A type-safe embedding of XDuce into ML. In *Workshop on ML*, 2005.

[20] K. Swedberg. How to get anything you want - part 1. Written Nov. 2006. `http://www.learningjquery.com/2006/11/how-to-get-anything-you-want-part-1`.

[21] K. Swedberg. How to get anything you want - part 2. Written Dec. 2006. `http://www.learningjquery.com/2006/12/how-to-get-anything-you-want-part-2`.

[22] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of Typed Scheme. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2008.

[23] W3C. XML path language (XPath) 2.0. Written Dec. 2010. `http://www.w3.org/TR/xpath20/`.

[24] W3C. XQuery 1.0: An XML query language. Written Dec. 2010. `http://www.w3.org/TR/xquery/`.

[25] W3C. Selectors level 3. Written Sept. 2011. `http://www.w3.org/TR/selectors/`.

[26] C. Zenger. Indexed types. *Theoretical Computer Science*, 187(1–2):147–165, 1997.