Resource Reasoning about Mashups

Philippa A. Gardner, Gareth D. Smith, and Adam D. Wright

Imperial College, London, UK. {pg,gds,adw07}@doc.ic.ac.uk

Abstract. The growing "mashup" phenomenon involves websites using scripting languages alongside data to create complex applications that integrate data and code from many sources. This leads to problems with reliability, as either sources change unaware that they are a dependency of a remote service, or clients of a service use resources that are accidentally exposed, or the dynamic nature of the scripting languages cause unexpected interactions. We show how resource reasoning can be used to construct provably fault free mashup programs, where services deliberately expose a subset of their data and code, and clients ensure the integration of components is sound.

1 Introduction

The World Wide Web has evolved, from a collection of static pages serving scientific data, into a huge ecosystem where the boundary between "web page" and "software application" has become indistinct. Originally a textual markup system, the technologies underlying the web have been steadily augmented to a point where software developers can expect their users to have web browsers with powerful embedded *scripting languages* running on computers connected by persistent high speed Internet links. Whilst these technologies were mostly introduced to provide a more interactive user experience for the classical web, they are now used to develop complex software, hosted entirely within the web browser. For example, whereas 10 years ago one would expect an e-mail user to run special e-mail applications on their local computer, today many opt to visit a web-based e-mail system which often delivers a user experience comparable or superior to the locally installed software.

The complexity of building these applications is significant. Not only must developers write their desired functionality, they must work with a technology stack that was not originally designed to support such rich application development. Analogous to the use of libraries in traditional development, web developers lacking the data, expertise or capacity to implement a complete feature often turn to third party sources to provide it. The loosely bound nature of web technologies allows these resource integrations to occur at runtime, mediated by scripts running in the web browser. This technique is known as *Client Side Mashups* (from now, just "mashups"), as many previously separate sources of data and code have been "mashed together" to create a new whole.

One natural mashup example is the customisation of a geographical map with additional features. Such a map requires a significant database of information coupled with expertise in the rendering of map user interfaces. Many developers wish to convey geographical data, such as the location of their business, but cannot develop the mapping system themselves. They turn to map service providers (such as Google or Microsoft), who provide data and code which can be easily integrated into their project and extended with private customisations.

Programming techniques for mashups have evolved almost accidentally from developmental trends within the web engineering community. As such, there is no formal definition but there are common techniques and, inevitably, common problems. The combination of highly structured data and programming languages with minimal safety properties leads to *accidental dependencies*, where a client uses structure or code the service intended to keep as a private implementation detail (and hence breaks when a change is made). Additionally, issues of combining rapidly changing code from disparate development teams often leads to expectation mismatches, rendering the runtime code integration of a mashup unreliable; we call this the *fault free integration* problem. In this paper, we show how resource reasoning can be used to mitigate these problems, by allowing services and clients to construct formal proofs of their fault freedom from a set of component specifications.

Web Data and Programming Language We introduce an abstract data structure which describes locally stored web data. It is XML-like data which has been parsed by DOM [1], and thus can be viewed as an in-place memory store manipulated by our web programs. In addition to this DOM data stored locally, a web program may access data and code stored elsewhere on the web, identified by a URI. In practice, this data is represented in text files: JavaScript scripts or XML documents which may have embedded code in SCRIPT tags. We do not specify an XML or JavaScript parser, instead working with a pre-parsed web, an interface which may be provided by other layers in the software stack.

We introduce a "Reasonable Web" programming language consisting of a fragment of DOM for manipulating our web data, a function system for sharing named blocks of code in a dynamic fashion, and two commands for accessing external data and code. In [GSWZ08b], Gardner, Smith *et al.* developed Featherweight DOM[2], which captured the essence of DOM. Here, we provide a pragmatic fragment of DOM Core Level 1, which identifies the important commands used by mashup programmers.

We aim for a simple language, but still one that engineers developing real web applications would recognise. As such, we take some syntactic and semantic cues from JavaScript (the de facto language for web scripting). Our function system captures the highly dynamic nature of functions in procedural style JavaScript, where functions exist in a shared global namespace and can be replaced at any point in execution by an alternative (not necessarily related) implementation. This unfortunate language feature can lead to problems in mashup situations, which we begin to tame with our reasoning. Whereas browsers provide many

[1] The *Document Object Model*, A W3C Standard introduced in [dom]
[2] In [GSWZ08b,GSWZ08a], called Minimal DOM.

methods for the integration of remote data and code, we abstract all the concepts into two general "web fetch" commands, which we model after the AJAX[3] concept.

For an implementation of our language, see [PG]. This implementation is written in JavaScript, will run in most modern browsers, and can be hosted inside standard web pages entirely transparently. This system has been used, both to test the practical utility of our language choices as well as to develop non-trivial applications which we analyse with our reasoning.

Resource Reasoning We provide a reasoning system for the development and verification of mashups using our "Reasonable Web" programming language. In particular, we use the resource reasoning introduced by O'Hearn [IO01], which is based on an analysis of a program's use of resource: for example, the heap when using Separation Logic to reason about C programs [Rey02a], a tree structure when using Context Logic to reason about tree update [CGZ05], and here we adapt Gardner, Smith *et al.*'s resource reasoning for Featherweight DOM to reason about our "Reasonable Web" programs.

We show how the local style of specifications provided by resource reasoning allows us to describe both pure data, and code that acts on smaller segments of that data. The ability to describe XML style data, and the analysis of code footprints allows us to see precise effects of code and data on the larger environment, helping to ameliorate *accidental dependencies*. Our approach is designed to facilitate *component reasoning*, such that the client need not consider anything more than the published specifications of a service. Combining this with extensions of earlier fault avoidance work, we develop solutions to the *fault free integration* problem.

Example We demonstrate our technique via the development and verification of a substantial example, geographical maps. As mentioned, due to the complexity of implementation and difficulty of sourcing the data, they have become a commonly used mashup which provides interesting interaction between the service code providing the map, and the client who will augment the default data with their domain specific knowledge. Our reasoning allows our service to provide both structured map data and code that manipulates it, along with *specifications*. These specifications show the service is not exposing more than was intended, and will be used by our client to prove it is fault free.

2 Web Data Structures

In order to reason about web programming, we introduce abstract data structures to represent the XML-like data that web programs manipulate. First we define a structure to represent DOM data in the local web browser. The root of

[3] Asynchronous JavaScript and XML, a technique allowing JavaScript code to fetch XML (or indeed, other data) from remote sites

this structure is a grove $\mathbf{g} \in \mathrm{G}$, which is equivalent to the object heap in an object oriented DOM implementation. A grove may contain the DOM representations of various XML structure: elements $\mathbf{ele} \in \mathrm{ELE}$; attributes $\mathbf{attr} \in \mathrm{ATTR}$; documents $\mathbf{doc} \in \mathrm{DOC}$ and text nodes $\mathbf{txt} \in \mathrm{TXT}$. We also have intermediate structures: groves document-elements $\mathbf{de} \in \mathrm{DE}$; element-attributes $\mathbf{ea} \in \mathrm{EA}$; element forests $\mathbf{ef} \in \mathrm{EF}$; attribute forests $\mathbf{af} \in \mathrm{AF}$; strings $\mathbf{s} \in \mathrm{S}$ and characters $\mathbf{c} \in \mathrm{C}$. For each of these structures, denoted $\mathbf{d} \in \mathrm{D}$ we may write $\mathbf{d}{:}\mathrm{D}$ to say that $\mathbf{d}$ is of type D. This data structure is given in Figure 1.

groves	$\mathbf{g} ::= <\mathbf{doc}>_{\mathrm{G}} \mid <\mathbf{ele}>_{\mathrm{G}} \mid <\mathbf{attr}>_{\mathrm{G}} \mid <\mathbf{txt}>_{\mathrm{G}} \mid \varnothing_{\mathrm{G}} \mid \mathbf{g} \oplus_{\mathrm{G}} \mathbf{g}$	
documents	$\mathbf{doc} ::= \text{“\#document”}_{\mathbf{id}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{null}_{9}}^{\mathbf{null}} [\mathbf{de}]_{\mathbf{fid}} \mathbf{null}$	
document-element	$\mathbf{de} ::= <\mathbf{ele}>_{\mathrm{DE}} \mid \varnothing_{\mathrm{DE}}$	
elements	$\mathbf{ele} ::= \mathbf{s}_{\mathbf{id}} \{\!\!\{ \mathbf{ea} \}\!\!\}_{\mathbf{aid}_{1}}^{\mathbf{idref}} [\mathbf{ef}]_{\mathbf{fid}} \mathbf{null}$	where ‘#’ is not in $\mathbf{s}$
element attributes	$\mathbf{ea} ::= <\mathbf{attr}>_{\mathrm{EA}} \mid \varnothing_{\mathrm{EA}} \mid \mathbf{ea} \otimes_{\mathrm{EA}} \mathbf{ea}$	where the name of each $\mathbf{attr}$ is sibling-unique
element forests	$\mathbf{ef} ::= <\mathbf{ele}>_{\mathrm{EF}} \mid <\mathbf{txt}>_{\mathrm{EF}} \mid \varnothing_{\mathrm{EF}} \mid \mathbf{ef} \otimes_{\mathrm{EF}} \mathbf{ef}$	
attributes	$\mathbf{attr} ::= \ll \mathbf{s}_{\mathbf{id}} \mapsto [\mathbf{af}]_{\mathbf{fid}} \gg^{\mathbf{idref}}$	where ‘#’ is not in $\mathbf{s}$
attribute forests	$\mathbf{af} ::= <\mathbf{txt}>_{\mathrm{AF}} \mid \varnothing_{\mathrm{AF}} \mid \mathbf{af} \otimes_{\mathrm{AF}} \mathbf{af}$	
text	$\mathbf{txt} ::= \text{“\#text”}_{\mathbf{id}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{null}_{3}}^{\mathbf{idref}} [\varnothing_{\mathrm{EF}}]_{\mathbf{fid}} \mathbf{s}$	
strings	$\mathbf{s} ::= <\mathbf{c}>_{\mathrm{S}} \mid \varnothing_{\mathrm{S}} \mid \mathbf{s} \otimes_{\mathrm{S}} \mathbf{s}$	
characters	$\mathbf{c} ::= \text{‘a’} \mid \text{‘b’} \mid \text{‘c’} \ldots$	

where $\mathbf{id}, \mathbf{fid}, \mathbf{idref}, \mathbf{aid} \in \mathrm{ID}$ and $\mathbf{id}, \mathbf{fid}, \mathbf{aid}$ must be unique across the browser's grove. $\otimes_{\mathrm{D}}$ is associative with unit $\varnothing_{\mathrm{D}}$; $\oplus_{\mathrm{D}}$ is associative and commutative with unit $\varnothing_{\mathrm{D}}$.

Fig. 1. Web Browser DOM Structure

Node structures are written in the form

$$\mathbf{nodename}_{\mathbf{id}} \{\!\!\{ \mathbf{attributes} \}\!\!\}_{\mathbf{aid}_{\mathbf{type}}}^{\mathbf{owner}} [\mathbf{children}]_{\mathbf{fid}} \mathbf{value}$$

where **nodename** is the name of the node (which may be a special string such as “#text”), **attributes** is the NamedNodeMap of this node's attributes, **children** is the NodeList of this node's children and **value** is either the value of this node if applicable, or null. The values $\mathbf{id}, \mathbf{aid}, \mathbf{fid}$ are ID values, which may be thought of as heap locations in an object oriented DOM implementation. They identify the node, the node's attribute map (which may be null) and the node's child list respectively. If the node in question is a document then **owner** null. Otherwise it is a reference to the ID of this node's owner document. The type of this node is given by **type**, represented in the DOM specification as an integer between 1 and 12. In this work we deal only with nodes of type element, attribute, text node and document – types 1,2,3 and 9 respectively.

Attribute nodes behave sufficiently differently from all other DOM nodes that we give them their own unique syntax: $\ll \mathbf{name}_{\mathbf{id}} \mapsto [\mathbf{children}]_{\mathbf{fid}} \gg^{\mathbf{owner}}$ where $\mathbf{name}, \mathbf{children}, \mathbf{id}, \mathbf{fid}, \mathbf{owner}$ are as before. Attributes do not have attributes of their own, so there is no syntax for them. The type of an attribute is always “2”, and so is not written. The value of an attribute is the concatenation of the values of its children, and so must be calculated at run time.

As a notational convenience we will omit type annotations such as the S in $\otimes_{\mathrm{S}}$ when it is clear from the context, and we will use the shorthand "xyz" to refer to the string $<$'x'$>_{\mathrm{S}} \otimes <$'y'$>_{\mathrm{S}} \otimes <$'z'$>_{\mathrm{S}}$.

For example, we can represent the XML:

```
<head>
  <script>n = createElement(document, "body");</script>
  <title>An Empty Web Page</title>
</head>
```

using the DOM structure:

$$\left< \begin{array}{l}
\text{“head”}_{\mathbf{id1}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{aid1}}{}_{1}^{\mathbf{doc}}[\\
\quad <\text{“script”}_{\mathbf{id2}} \{\!\!\{ \varnothing \}\!\!\}_{\mathbf{aid2}}{}_{1}^{\mathbf{doc}}[\\
\qquad <\text{“\#text”}_{\mathbf{id4}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{null}}{}_{3}^{\mathbf{doc}}[\varnothing]_{\mathbf{fid4}}\text{“n = createDocument(document, “body”);”}> \\
\quad]_{\mathbf{fid2}}\mathbf{null}> \\
\quad \otimes \\
\quad <\text{“title”}_{\mathbf{id3}} \{\!\!\{ \varnothing \}\!\!\}_{\mathbf{aid3}}{}_{1}^{\mathbf{doc}}[\\
\qquad <\text{“\#text”}_{\mathbf{id5}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{null}}{}_{3}^{\mathbf{doc}}[\varnothing]_{\mathbf{fid5}}\text{“An Empty Web Page”}> \\
\quad]_{\mathbf{fid3}}\mathbf{null}>]_{\mathbf{fid1}}\mathbf{null}
\end{array} \right>_{\mathrm{G}}$$

Since this is DOM data, the code in the script node is not directly executable. From the point of view of a DOM parser, it is just a string. We do allow the dynamic execution of remote script code in our programming language, as discussed in Section 3. Indeed, this use case is one motivation for the next data structure we introduce.

In addition to the DOM data stored locally, a web program may access data and code stored elsewhere on the web, identified by a $uri \in \mathbf{URI}$. We do not specify how the XML or JavaScript is transfered and parsed. Instead we work with a pre-parsed web, an interface which may be provided by other layers in the software stack. This pre-parsed web structure is given in Figure 2. The parser is a partial function π from the global set of URIs, $\mathbf{URI}$, to web resources which may contain data, program code, or both. Note in particular that ID values in any parsed document must still be unique with respect to the browser's grove. We assume this property is maintained by the π function.

$$\begin{array}{lrl}
\text{Parsed Resource} & \mathbf{resource} & ::= (\mathbf{data}, \mathbf{code}) \\
\text{Parsed Data} & \mathbf{data} & ::= \mathbf{doc} \mid \varnothing \\
\text{Parsed Code} & \mathbf{code} & ::= \mathbf{C} \mid \varnothing \\
\text{Parser} & \pi & : \mathbf{URI} \rightharpoonup \mathbf{resource}
\end{array}$$

Where **C** is a syntactically valid program as per the grammar in section 3 and $\mathbf{doc} \in \mathrm{DOC}$.

Fig. 2. Data Structure Provided by Parser

Note that we use $\otimes$ rather than $\oplus$ to preserve the order of the element attributes in our model. The DOM specification deals with the preservation of the order of attributes when it describes the structure that contains them – the

NamedNodeMap. The DOM specification says “NamedNodeMaps are not maintained in any particular order.”, but it also says “ Objects contained in an object implementing NamedNodeMap may also be accessed by an ordinal but . . . this does not imply that the DOM specifies an order to these Nodes.” These sentences are hard to reconcile, so our choice in this matter is guided by the text of the pre-release draft of DOM Core Level 1, which said: “ DOM implementations should, when possible, preserve the ordering of objects in a NamedNodeMap in case the author of the source document assigned some meaning to this ordering that is not defined in the DOM, XML or HTML specifications.”

3 Language

We now introduce our “Reasonable Web” programming language. We view the work as a *web scripting language*, in the vein of JavaScript. The language consists of standard imperative features (expressions, conditionals, while loops, variable assignment and statement sequencing), augmented with: DOM commands to manipulate our tree structured heap; commands to work with non-local data and code; and a function system allowing the encapsulation and export of code fragments in a highly dynamic fashion. We give the grammar of our language in figures 3, 4 and 5. In the grammar, we use the convention that lower case strings refer to literals, whilst leading upper case letters name productions. We make explicit our expectation of brackets in the syntax by quoting, and use unmentioned rules for `FName`, `Id`, `Str`, `Int`, `Bool` and `Val` to refer to the set of valid function names, identifier, string, integer, boolean and all variable names respectively.

Expressions (Figure 3) have standard semantics. In line with practical languages, we provide a set of operations over booleans and so inherit the notation that `==` is equality, and `!=` is inequality. We distinguish four types of variable: Our Str, Int and Bool types are standard, whilst IDs store heap identifiers or **null**.

```
IdExpr     ::=    null | Id
StrExpr    ::=    null | empty | c | Str | StrExpr . StrExpr
IntExpr    ::=    n | Int | IntExpr + IntExpr | IntExpr - IntExpr
BoolExpr   ::=    false | true | Bool | BoolExpr == BoolExpr | IdExpr == IdExpr
                | StrExpr == StrExpr | IntExpr == IntExpr | IntExpr > IntExpr
                | IntExpr < IntExpr | IntExpr >= IntExpr | IntExpr <= IntExpr
                | BoolExpr != BoolExpr | IntExpr != IntExpr | StrExpr != StrExpr
                | IdExpr != IdExpr

Expression ::=    IdExpr | StrExpr | IntExpr | BoolExpr
```

Fig. 3. Grammar for expressions

Within the language (Figure 4), we distinguish the set of statements allowed in functions from those allowed outside, breaking the language into two natural scopes. We refer to the outer scope as “global”, and the scope within a function as “local”. Variables introduced at the global scope are bound everywhere, including within functions. Function bodies may declare “local” variables with the `var`

```
FStatement ::=
    Id   = IdExpr;
  | Str  = StrExpr;
  | Int  = IntExpr;
  | Bool = BoolExpr;
  | Val  = FName ‘(’ Expression?
                     (, Expression)* ‘)’;
  | if ‘(’ BoolExpr ‘)’ then { FStatement* }
                        else { FStatement* };
  | while ‘(’ BoolExpr ‘)’ { FStatement* };
  | skip;
  | Command;

Local      ::= var Name;
Return     ::= return Expression;

Program    ::=
    FStatement
 | function FName (Val1, ..., Valn)
   {
     Local*
     FStatement*
     Return
   };
 | Program ; Program
```

Fig. 4. Grammar for language features

```
Command ::=
    Id  = createElement(Id, Str)
  | Id  = createTextNode(Id, Str)
  | Id  = createAttribute(Id, Str)
  | Str = getNodeName(Id)
  | Str = getNodeValue(Id)
  | Int = getNodeType(Id)
  | Id  = getParentNode(Id)
  | Id  = getChildNodes(Id)
  | Id  = getAttributes(Id)
  | Id  = getOwnerDocument(Id)
  | Id  = appendChild(Id, Id)
  | Id  = removeChild(Id, Id)
  | Id  = item(Id, Int)
  | Id  = setNamedItem(Id, Id)
  | Id  = removedNamedItem(Id, Str)
  | str = substringData(Id, Int, Int)
  | deleteData(Id, Int, Int)
  | appendData(Id, Str)
```

Fig. 5. Grammar for commands

command that are local to the body code, and fresh at each invocation. Note we syntactically ensure functions are introduced only at global scope.

3.1 DOM Library

We provide a set of commands, acting as the interface to our tree structured heap. It is designed to conform to a large subset of W3C DOM specification. We model only the commands required to implement the essential elements of standard mashup engineering practice, and so consider only the node types of Document, Element, Text and Attr.

The standard DOM API gives a natural object interpretation of the tree structure within both our heap and web. For simplicity of presentation, we adopt a “flattened” view of the DOM APIs. Methods such as `p.appendChild(c)` become commands of the form `appendChild(p, c)`; likewise, object field access `n.value` become `setNodeValue(n)` and `getNodeValue(n)`. The full command set is given in Figure 5.

As with previous work ([GSWZ08a],[GSWZ08b]), we formally and compositionally specify a subset of the commands we wish to use, and can implement remaining commands in terms of that subset. Note there are no destructive commands in the language; even removing a node will merely place it at a new place in the grove. We thus naturally consider the language garbage collected. We take the view that documents originate from within the web, and so provide no specific commands for document creation. We consider the behaviours of our DOM manipulating commands in small groups, and use realistic variable names to aid presentation.

```
node = createElement(doc, nameStr)
node = createTextNode(doc, dataStr)
node = createAttribute(doc, nameStr)
```

These commands create new nodes within the document `doc`. They introduce respectively, a new element with the given `nameStr`, a new text node with the given `dataStr`, or a new attribute with the given `nameStr`, each recording the result in `node`. They fault if `doc` does not identify a document node, or if `createElement` or `createAttribute` is given a `nameStr` containing the '#' character.

```
str = getNodeName(node)          val = getNodeValue(node)
int = getNodeType(node)
```

These commands obtain properties of nodes within the grove. They obtain respectively, the name, value and type of the given node `node`, recording the result in the appropriate variable. They fault if `node` does not identify a valid node.

```
node = getParentNode(node)          nodes = getChildNodes(node)
nodes = getAttributes(node)         doc = getOwnerDocument(node)
```

These commands obtain nodes spatially related to `node`, assigning the result to the appropriate variable. Respectively, they get the parent node (or null if it has none), child node list, attribute set (or null if absent), and the owner document (or null if `node` refers to a document node). They fault if `node` does not identify a valid node.

```
node = appendChild(parentNode, newChildNode)
```

This command moves the tree `newChildNode` to the end of the child list of `parentNode`. It faults if `parentNode` or `newChildNode` do not identify valid nodes, or if `newChildNode` is an ancestor of `parentNode`. The command also faults if appending the node would break the following tree structure invariants: Document nodes may have at most one child, which must be an element; Text nodes may have no children; Document nodes cannot be the children of any node; Attribute nodes may have only text nodes as children.

```
node = removeChild(parentNode, childNode)
```

This command moves the node identified by `childNode` from the child list of `parentNode` to the root of the grove. It faults if either `parentNode` or `childNode` do not identify nodes, or if `childNode` is not in the child list of `parentNode`.

```
node = item(nodes, int)
```

The `item` command obtains the `int`$^{\text{th}}$ element from either the attributes or forest represented by `nodes`. If there are fewer than `int` elements in the structure, or if `int` < 0, it returns `null`. The command faults if `nodes` does not identify either attributes or a forest.

```
node = setNamedItem(map, newNode)
node = removedNamedItem(map, nameStr)
```

These commands represent the NamedNodeMap interface of DOM. The first adds the node given by `newNode` as the last entry in `map`, using the name of the `newNode` node. If this replaces a given node already stored with that name, the replaced node identifier is stored in `node` (otherwise, **null** is stored). The second removes the item with the given `nameStr`, storing the removed node identifier in `node`. They both fault if `map` does not refer to a valid map structure, if `nameStr` is null, if `newNode` does not refer to a valid element or text node, or if one attempts to use `removedNamedItem` on a `nameStr` that is not present in `map`.

```
str = substringData(node, offsetInt, countInt)
```

This command assigns to `str` the substring of the text node `node` starting at character index `offsetInt` with length `countInt`. If `offsetInt` + `countInt` exceeds the string length, all characters to the string end are returned. This command faults if `node` does not identify a text node, `offsetInt` or `countInt` are negative, or `offset` exceeds the string length.

```
deleteData(node, offsetInt, countInt)
```

This command is similar to `substringData`, but rather than returning the indexed string, it removes it from the value of the node.

```
appendData(node, str)
```

The command appends the string `str` to the end of the string contained in `node`, faulting if `node` does not identify a text node.

3.2 Web

The essence of a mashup is the acquisition of data and code from a third party source, and the integration of the results into a new whole. Whereas HTML and JavaScript provide several mechanisms to achieve this, we abstract into two commands for interacting with these external resources. We are aiming to capture the spirit of the AJAX concept, which provides JavaScript with the ability to download content from a given URI. Whereas in JavaScript, the user is then responsible for interpreting the result, we give only two possibilities with our commands.

```
x = fetchDocument(uri)
```

Enters into the grove the document data parsed from the given `uri`, and assigns the identifier of the added document node to `x`. Recall we assume that all identifiers from the parser are fresh to our local environment. It faults if the `uri` is not available in the parser.

```
runScript(uri)
```

This command runs the script code associated with the given `uri` within the environment of the code that called it. As such, all global name and function bindings that have been made are exposed to the remote script, and any alterations or additions it makes will be visible after the execution. This command will fault if either the given `uri` does not exist, or if the execution of the script code associated with `uri` faults.

Our "web" is considered as a mapping of URIs to pre-parsed document data and code. The notion that these sources are parsed removes issues of well-formed data and transfer failure from both the code and data; all references to an existing URL will certainly return well-formed code and data. It is valid to make repeated calls to both commands on the same URI. The result will be many copies of the same document, or repeated attempts at execution of the same script.

3.3 Functions

An important characteristic of a mashup service is that it provides not only data, but also some "behaviour" associated with that data. For example, a web based e-mail client doesn't simply present the user with a single large document containing all his e-mail. It provides the e-mail data, and also a way for the user to navigate that data, reply to e-mails and so on.

Our language's function mechanism not only provides a way for programmers to partition their code into manageable chunks, but also a way for service authors to export behaviour to a client service. For example, a web based e-mail client might export a function "reply" which takes the ID of a node containing e-mail data, and alters the local DOM to provide a reply dialog to the user.

To make it possible for functions to be exported in this manner, their behaviour is defined dynamically. The set of functions available to a program at a given time is a property of the program state, not a static property of the program. A program which refers to function names with no obvious lexical meaning may nonetheless be entirely correct. For example, the function in question may be introduced at run time from a remote source by a call to `runScript`.

Functions are entries in a *function table* **FT**, a partial function mapping names to a tuple consisting of local variable names, formal parameter names and an associated code block called the *body*. The function table may be updated by the function introduction command `function f(p1, ..., pm) {...}` . If a function of name f does not already exist in the function table, it will be introduced. If it does already exist, it will be overwritten. Functions must return a single value as the last statement of the body using the `return` statement and may be mutually recursive.

To simplify this presentation, we assume that the sets of local variable names and parameter names are disjoint from the set of global variable names and we preclude assignment to parameters. This last restriction is no real constraint, as users can always introduce a local variable and assign the parameter value to that.

3.4 Operational semantics

The behaviour of the language is described formally by an operational semantics. We use an evaluation relation $\rightsquigarrow$, relating tuples of form $s, \mathbf{FT}, \mathbf{g}, \mathtt{C}$ to either terminal states $s, \mathbf{FT}, \mathbf{g}$ or fault, where s is a store, $\mathbf{g}$ a grove, $\mathbf{FT}$ the function table, and $\mathtt{C}$ is a program. A store is a partial function from variable names to values, and we write $[\![E]\!]_s$ for the evaluation of the expression E in store s. The function table is a partial function from function names to triples of the form $(\mathtt{C}, \overline{\mathbf{p}}^m, \overline{\mathbf{l}}^n)$. This triple represents function body code, parameter names and local variable names. Groves are defined as in the data structure section. We often write the list `x1, ..., xr` of distinct names as $\overline{x}^r$, and write $\{\overline{x}^r\}$ for the set of the list elements.

The full semantics are given in [PG]. In defining the operational semantics we found it convenient to make use of the context structure we define here in Section 4. Here, we give only the notable semantics for the commands `fetch-Document` and `runScript`, and the interesting rules from the function system.

$$\frac{\begin{array}{c}\pi([\![\mathtt{uri}]\!]_s) = (\mathbf{\#document_{id}} \lbrace\!\!\lbrace\, \varnothing_{\mathrm{EA}} \,\rbrace\!\!\rbrace_{\mathbf{null}_{\mathbf{9}}^{\mathbf{null}}}[\mathbf{f}]_{\mathbf{fid}}\mathbf{null}, \mathbf{code}) \wedge \\ \mathbf{g}' \equiv \mathbf{g} \oplus <\mathbf{\#document_{id}} \lbrace\!\!\lbrace\, \varnothing_{\mathrm{EA}} \,\rbrace\!\!\rbrace_{\mathbf{null}_{\mathbf{9}}^{\mathbf{null}}}[\mathbf{f}]_{\mathbf{fid}}\mathbf{null}>_{\mathrm{G}}\end{array}}{s, \mathbf{FT}, \mathbf{g}, \mathtt{n} := \mathtt{fetchDocument}(\mathtt{uri}) \rightsquigarrow s[\mathtt{n} \leftarrow \mathbf{id}], \mathbf{FT}, \mathbf{g}'}$$

This states that the given `uri` expression, when evaluated in the current store, identifies at least a parsed document structure. The new grove is given by appending the document to the existing grove, and the store is updated so that the variable `n` reflects the identifier of the new document.

$$\frac{\pi([\![\mathtt{uri}]\!]_s) = (\mathbf{data}, \mathbf{code}) \wedge \mathbf{code} \neq \varnothing \wedge \quad s, \mathbf{FT}, \mathbf{g}, \mathbf{code} \rightsquigarrow s', \mathbf{FT}', \mathbf{g}'}{s, \mathbf{FT}, \mathbf{g}, \mathtt{runScript}(\mathtt{uri}) \rightsquigarrow s', \mathbf{FT}', \mathbf{g}'}$$

As in `fetchDocument`, this shows we can parse the given `uri` into some data we which we ignore, and some non-empty code. The code is executed in the same environment as that used to call the `runScript` command, and may produce an entirely new store, function table and grove. If the execution of the script faults, the semantic is undefined (and the fault condition propagates by rules not given here).

$$\frac{}{s, \mathbf{FT}, \mathbf{g}, \begin{array}{l}\mathtt{function\ f(p1,\ ...,\ pm)}\ \{ \\ \mathtt{local\ l1;\ ...\ local\ ln;\ C}\ \}\end{array} \rightsquigarrow s, \mathbf{FT}[\mathtt{f} \leftarrow (\mathtt{C}, \overline{\mathbf{p}^m}, \overline{\mathbf{l}^n})], \mathbf{g}}$$

Function introduction simply updates the function table, mapping the new function name to the code along with the parameter and local name vectors. If an existing function in the function table takes that value, it is overwritten (even if it has different parameter names or arity).

$$\frac{\begin{array}{c}\mathbf{FT}(\mathbf{f}) = (\mathtt{C}, \overline{\mathbf{p}}^m, \overline{\mathbf{l}}^n) \\ \text{savedParam}_i = s(\mathbf{p}_i) \quad \text{savedLocal}_j = s(\mathbf{l}_j) \\ [s|\mathbf{p}_i \leftarrow [\![\mathtt{e}_i]\!]_s]\backslash\{\overline{\mathbf{l}}^n\}, \mathbf{FT}, \mathbf{g}, \mathtt{C} \rightsquigarrow s', \mathbf{FT}', \mathbf{g}' \\ 1 \leq i \leq m, 1 \leq j \leq n \\ s'' = [s'|\mathbf{p}_i, \mathbf{l}_j \leftarrow \text{savedParam}_i, \text{savedLocal}_j, \mathtt{v} \leftarrow s'(\rho)]\end{array}}{s, \mathbf{FT}, \mathbf{g}, \mathtt{v} := \mathbf{f}(\mathtt{e}_1, \ldots, \mathtt{e}_n) \rightsquigarrow s'', \mathbf{FT}', \mathbf{g}'}$$

Function call requires that the requested function exists within the function table. The parameter names and local variable names have their current values stored, the parameter expressions are evaluated, and a new store is constructed where the parameter names map to the results of the parameter expression evaluation, and the names of the local variables are made undefined. The function code is then executed, resulting in a new store, function table and grove. The new store is updated to remap the local names to their original values, and to assign the return value (stored in the reserved variable ρ).

4 Assertion Language

In order to reason about programmes written in the web programming language, we define an assertion language, with which we can assert properties of the state of the local browser, and the web it accesses. We base our work on the assertion language of [GSWZ08b], itself derived from Context Logic.

With Context Logic, the fundamental idea is that, in order to provide resource reasoning about tree update, we must reason about both trees and the interim contexts. Hence, central to our assertion language is the concept of a context. As introduced in [CGZ05] and used to reason about DOM in [GSWZ08a] and [GSWZ08b], a context may be viewed as a tree with a hole in it. If a tree is placed into the hole, as long as there is no clash of identifiers, the result is a larger, valid tree. In our assertion language, we will find contexts useful as a way of splitting the DOM structure into disjoint parts which may be described separately. For example, the portion of the DOM structure that a command alters and the portion that is left unchanged. We give our context structure in Figure 6, following the shape of our data structure (Figure 1).

As before, we omit type annotations such as the S in $\otimes_{\mathrm{S}}$ when it is clear from the context. Just as data structure elements were divided into type sets G, DOC, DE, ELE, EA, EF, ATTR, AF, TXT, S and C, contexts are divided into sets CG, CDOC, CDE, CELE, CEA, CEF, CATTR, CAF, CTXT, CS and CC. In the case of contexts however, we wish our types to distinguish not only between contexts of a given outer shape, but also contexts containing a particular sort of hole. A context type judgement then, looks like $\mathbf{c}{:}\mathrm{D}_1{\rightarrow}\mathrm{D}_2$, which signifies a context containing a hole $-_{\mathrm{D}_1}$ such that if a datum of type D_1 is put in that hole, the resulting datum is of type D_2.

To formalise putting a datum into a context hole to obtain a new datum, we introduce a partial application function $\mathrm{ap}{:}((\mathrm{D}_1{\rightarrow}\mathrm{D}_2) \times \mathrm{D}_1) \rightharpoonup \mathrm{D}_2$ in Figure 7. Note that the application function is only defined for arguments which would

$$\begin{aligned}
\mathbf{cg} &::= <\mathbf{cdoc}>_{\mathrm{G}} \mid <\mathbf{cele}>_{\mathrm{G}} \mid <\mathbf{cattr}>_{\mathrm{G}} \mid <\mathbf{ctxt}>_{\mathrm{G}} \mid -_{\mathrm{G}} \mid \mathbf{cg} \oplus_{\mathrm{G}} \mathbf{g}\\
\mathbf{cdoc} &::= \text{“\#document”}_{\mathbf{id}} \Lbag\, \varnothing_{\mathrm{EA}} \,\Rbag_{\mathbf{null}}{}_{9}^{\mathbf{null}} [\mathbf{cde}]_{\mathbf{fid}} \mathbf{null} \mid -_{\mathrm{DOC}}\\
\mathbf{cde} &::= <\mathbf{cele}>_{\mathrm{DE}} \mid -_{\mathrm{DE}}\\
\mathbf{cele} &::= \mathbf{s_{id}} \Lbag\, \mathbf{ea} \,\Rbag_{\mathbf{aid}}{}_{1}^{\mathbf{idref}} [\mathbf{cef}]_{\mathbf{fid}} \mathbf{null} \mid \qquad\qquad \text{where ‘\#’} \notin \mathbf{s}\\
&\ \mathbf{s_{id}} \Lbag\, \mathbf{cea} \,\Rbag_{\mathbf{aid}}{}_{1}^{\mathbf{idref}} [\mathbf{ef}]_{\mathbf{fid}} \mathbf{null} \mid -_{\mathrm{ELE}}\\
\mathbf{cea} &::= <\mathbf{cattr}>_{\mathrm{EA}} \mid -_{\mathrm{EA}} \mid \mathbf{cea} \otimes_{\mathrm{EA}} \mathbf{ea} \mid \mathbf{ea} \otimes_{\mathrm{EA}} \mathbf{cea} \text{ where each } \mathbf{attr} \text{ name is sibling-unique}\\
\mathbf{cef} &::= <\mathbf{cele}>_{\mathrm{EF}} \mid <\mathbf{ctxt}>_{\mathrm{EF}} \mid -_{\mathrm{EF}} \mid \mathbf{cef} \otimes_{\mathrm{EF}} \mathbf{ef} \mid \mathbf{ef} \otimes_{\mathrm{EF}} \mathbf{cef}\\
\mathbf{cattr} &::= \ll \mathbf{s_{id}} \mapsto [\mathbf{caf}]_{\mathbf{fid}} \gg^{\mathbf{idref}} \mid -_{\mathrm{ATTR}} \qquad\qquad \text{where ‘\#’} \notin \mathbf{s}\\
\mathbf{caf} &::= <\mathbf{ctxt}>_{\mathrm{AF}} \mid -_{\mathrm{AF}} \mid \mathbf{caf} \otimes_{\mathrm{AF}} \mathbf{af} \mid \mathbf{af} \otimes_{\mathrm{AF}} \mathbf{caf}\\
\mathbf{ctxt} &::= \text{“\#text”}_{\mathbf{id}} \Lbag\, \varnothing_{\mathrm{EA}} \,\Rbag_{\mathbf{null}}{}_{3}^{\mathbf{idref}} [\varnothing_{\mathrm{TF}}]_{\mathbf{fid}} \mathbf{cs} \mid -_{\mathrm{TXT}}\\
\mathbf{cs} &::= <\mathbf{cc}>_{\mathrm{S}} \mid -_{\mathrm{S}} \mid \mathbf{cs} \otimes_{\mathrm{S}} \mathbf{s} \mid \mathbf{s} \otimes_{\mathrm{S}} \mathbf{cs}\\
\mathbf{cc} &::= -_{\mathrm{C}}
\end{aligned}$$

where $\mathbf{id}, \mathbf{fid}, \mathbf{idref}, \mathbf{aid} \in \mathrm{ID}$; $\mathbf{id}, \mathbf{fid}, \mathbf{aid}$ are unique ; $\otimes_{\mathrm{D}}$ is associative and $\oplus_{\mathrm{D}}$ is associative and commutative with unit $\varnothing_{\mathrm{D}}$ as before.

Fig. 6. Local DOM Contexts

result in a valid data structure, maintaining the uniqueness of all internal ids. We use $\mathrm{ap}(\mathbf{cd}, \mathbf{d}_1)\!\downarrow$ to mean “ap is defined for arguments $\mathbf{cd}, \mathbf{d}_1$”.

$$\begin{aligned}
\mathrm{ap}(-_{\mathrm{D}_1}, \mathbf{d}_1) &\triangleq \mathbf{d}_1\\
\mathrm{ap}(\mathbf{s_{id}} \Lbag\, \mathbf{ea} \,\Rbag_{\mathbf{aidn}}{}_{\mathbf{tp}}^{\mathbf{irn}} [\mathbf{con}]_{\mathbf{fid}} \mathbf{null}, \mathbf{d}_1) &\triangleq \mathbf{s_{id}} \Lbag\, \mathbf{ea} \,\Rbag_{\mathbf{aidn}}{}_{\mathbf{tp}}^{\mathbf{irn}} [\mathrm{ap}(\mathbf{con}, \mathbf{d}_1)]_{\mathbf{fid}} \mathbf{null}\\
\mathrm{ap}(\mathbf{s_{id}} \Lbag\, \mathbf{cea} \,\Rbag_{\mathbf{aid}}{}_{\mathbf{1}}^{\mathbf{idref}} [\mathbf{ef}]_{\mathbf{fid}} \mathbf{null}, \mathbf{d}_1) &\triangleq \mathbf{s_{id}} \Lbag\, \mathrm{ap}(\mathbf{cea}, \mathbf{d}_1) \,\Rbag_{\mathbf{aid}}{}_{\mathbf{1}}^{\mathbf{idref}} [\mathbf{ef}]_{\mathbf{fid}} \mathbf{null}\\
\mathrm{ap}(<\mathbf{con}>_{\mathrm{D}_2}, \mathbf{d}_1) &\triangleq <\mathrm{ap}(\mathbf{con}, \mathbf{d}_1)>_{\mathrm{D}_2}\\
\mathrm{ap}(\mathbf{con} \oplus \mathbf{d}_2, \mathbf{d}_1) &\triangleq \mathrm{ap}(\mathbf{con}, \mathbf{d}_1) \oplus \mathbf{d}_2\\
\mathrm{ap}(\mathbf{con} \otimes \mathbf{d}_2, \mathbf{d}_1) &\triangleq \mathrm{ap}(\mathbf{con}, \mathbf{d}_1) \otimes \mathbf{d}_2\\
\mathrm{ap}(\mathbf{d}_2 \otimes \mathbf{con}, \mathbf{d}_1) &\triangleq \mathbf{d}_2 \otimes \mathrm{ap}(\mathbf{con}, \mathbf{d}_1)\\
\mathrm{ap}(\ll \mathbf{s_{id}} \mapsto [\mathbf{con}]_{\mathbf{fid}} \gg^{\mathbf{idref}}, \mathbf{d}_1) &\triangleq \ll \mathbf{s_{id}} \mapsto [\mathrm{ap}(\mathbf{con}, \mathbf{d}_1)]_{\mathbf{fid}} \gg^{\mathbf{idref}}\\
\mathrm{ap}(\mathbf{\#text_{id}} \Lbag\, \varnothing_{\mathrm{EA}} \,\Rbag_{\mathbf{null}}{}_{\mathbf{3}}^{\mathbf{idref}} [\varnothing_{\mathrm{TF}}]_{\mathbf{fid}} \mathbf{cs}, \mathbf{d}_1) &\triangleq \mathbf{\#text_{id}} \Lbag\, \varnothing_{\mathrm{EA}} \,\Rbag_{\mathbf{null}}{}_{\mathbf{3}}^{\mathbf{idref}} [\varnothing_{\mathrm{TF}}]_{\mathbf{fid}} \mathrm{ap}(\mathbf{cs}, \mathbf{d}_1)
\end{aligned}$$

where:
$\mathbf{d}_1{:}\mathrm{D}_1$, $\mathbf{d}_2{:}\mathrm{D}_2$, $\mathbf{con}{:}(\mathrm{D}_1 \to \mathrm{D}_3)$, $\mathbf{ea}{:}\mathrm{EA}$, $\mathbf{ef}{:}\mathrm{EF}$, $\mathbf{f}{:}\mathrm{D}_4$ $\mathbf{tp} \in \{1, 3, 9\}$, $\mathbf{s} \in \mathrm{S}$,
$\mathbf{id}, \mathbf{fid}, \mathbf{idref}, \mathbf{aid} \in \mathrm{ID}$, $\mathbf{aidn}, \mathbf{irn} \in \mathrm{ID} \cup \{\mathbf{null}\}$

Fig. 7. The Application Function

4.1 Formulae of the logic

Context Logic consists of standard formulae constructed from the connectives of first-order logic, variables, expressions tests and quantification over variables. In addition, it has general *structural* formulae, and *specific formulae* applicable to our specific data structure. The structural formulae are constructed from an *application* connective for analysing context application, and its two corresponding *right adjoints.*

1. the application formulae $P \circ_{\mathrm{D}_1} P_1$ describes data of e.g. type D_2, which can be split into a context of type $\mathrm{D}_1 \to \mathrm{D}_2$ satisfying P, and disjoint sub-data of type D_1 satisfying P_1. The type information for the context hole cannot be derived from the given data, and so is annotated on the connective.
2. one right adjoint, $P \multimap_{\mathrm{D}_2} P_2$ describes data of e.g. type D_1 which, *whenever* it is successfully placed in a context of type $\mathrm{D}_1 \to \mathrm{D}_2$ satisfying P, results in data of type D_2 satisfying P_2. Again, the adjoint is annotated with type

information for the resulting data, which cannot be determined from the hole type.

3. the right adjoint $P_1 \multimap P_2$ describes a context of e.g. type $\mathrm{D}_1 \to \mathrm{D}_2$ which, whenever data of type D_1 satisfying P_1 is successfully inserted into it, results in data of type D_2 satisfying P_2. In this case, the type can be inferred from the type of the given data.

In our assertion language, the formulae categories break down as follows.

$$\begin{array}{ll}
P ::= P \Rightarrow P \mid \mathrm{false}_{\mathrm{D}} \mid \mathrm{false}_{\mathrm{D}_1\to\mathrm{D}_2} & \text{Boolean formulae} \\
\quad P \circ_{\mathrm{D}} P \mid P \circ\!-_{\mathrm{D}_2} P \mid P \multimap P & \text{structural formulae} \\
\quad \ldots (\text{See Fig } 8) \ldots & \text{DOM-specific formulae} \\
\quad \mathtt{var}_{\mathrm{E}} \mid \mathtt{Exp}_{\mathrm{V}} = \mathtt{Exp}_{\mathrm{V}} & \text{expression equality} \\
\quad \mathtt{Int}{=}|\mathtt{f}| & \text{length equality} \\
\quad \exists \mathtt{var}.\, P & \text{quantification} \\
\quad \gamma(\mathbf{f}) & \text{function existence} \\
\quad \omega_{\mathrm{D}}(\mathbf{uri}) & \text{Web data existence} \\
\quad \omega_{\mathrm{C}}(\mathbf{uri}) & \text{Web code existence}
\end{array}$$

where $\mathtt{var}_{\mathrm{E}}$ denotes a logical variable in $\textsc{Var}_{\mathrm{ENV}}$, $\mathrm{V}{\in}\{\textsc{Id}, \mathrm{S}, \mathbb{Z}, \mathbb{B}\}$, $\mathtt{var} \in \textsc{Var}_{\mathrm{ENV}} \cup \textsc{Var}_{\mathrm{STORE}}$, $\mathbf{f}{:}\mathrm{F}$, $\mathrm{F} \in \{\mathrm{DE}, \mathrm{EA}, \mathrm{EF}, \mathrm{AF}, \mathrm{S}\}$

$$\begin{array}{ll}
P ::= \ldots & \mid -_{\mathrm{D}} \mid P_{\mathtt{id}} \{\!\!\{ P \}\!\!\}_{\mathtt{aidn}}{}^{\mathtt{irn}}_{\mathtt{tp}} [P]_{\mathtt{fid}} \mathbf{null} \mid \\
& \text{“\#text”}_{\mathtt{id}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{null}_3}{}^{\mathtt{idref}} [\varnothing_{\mathrm{TF}}]_{\mathtt{fid}} P \mid \mathbf{c} \mid \\
& P \otimes_{\mathrm{D}} P \mid P \oplus_{\mathrm{D}} P \mid \varnothing_{\mathrm{D}} \mid {<}P{>}_{\mathrm{D}} \mid \\
& \ll P_{\mathtt{id}} \mapsto [P]_{\mathtt{fid}} \gg
\end{array}$$

Fig. 8. DOM Specific Formulae

The type annotations on the formulae enable us to define a simple typing relation $P\colon A$, where A is a data type D or a context type $\mathrm{D}_1{\to}\mathrm{D}_2$, by induction on the structure of formula P. The Boolean formulae and quantified formulae inherit their types from the sub-formulae. The equalities satisfy arbitrary A, since they are really outside the typing system as they test the store rather than the data and context structures. The formulae $\mathbf{\#text}_{\mathtt{id}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{null}_3}{}^{\mathtt{idref}} [\varnothing_{\mathrm{TF}}]_{\mathtt{fid}} P$ and ${<}P{>}_{\mathrm{EF}}$ have two typings, depending on whether they describe trees or tree contexts. $P_1 \otimes_{\mathrm{EF}} P_2$ also has the two typings. The context case has two options for typing subformulae, depending on which one describes the forest context. As the definition is mostly standard, we omit it here (it can be found in [PG]).

4.2 Satisfaction

The meaning of formulae written in our assertion language is given by a satisfaction relation. Recall that our programming language uses identifier, string, integer and boolean variables. For our resource reasoning (Section 5), we will also require data and context variables. Our assertion language therefore uses a logical environment e as well as the store s. The logical environment assigns values for data variables in $\mathrm{Var}_{\mathrm{D}}$ and context variables $\mathrm{Var}_{\mathrm{D}_1\to\mathrm{D}_2}$. However, there are no contexts of some types – for example $\mathrm{G}\to\mathrm{S}$. We therefore assume the corresponding variable sets are empty. To avoid ambiguity with string variables in the program store, the environment variable set of type S is also empty.

The satisfaction relation $e, s, \mathbf{FT}, \mathbf{a} \models_{\mathrm{A}} P$ is defined on environment e, variable store s, function table $\mathbf{FT}$, a datum or context $\mathbf{a}$ of type A, and formula P of type A by induction on P. The type A is a data or context type D or $\mathrm{D}_1\to\mathrm{D}_2$ and $\mathbf{cd}$ is a context of type $\mathrm{D}_1\to\mathrm{D}_2$. Satisfaction for the Boolean formulae is standard. We give satisfaction for the structural formulae in Fig 9, for our data structure formulae in Fig 10, with the remainder in Fig 11.

$$\begin{array}{lcl}
e, s, \mathbf{FT}, \mathbf{d}_2 \models_{\mathrm{D}_2} P_1 \circ_{\mathrm{D}_1} P_2 & \iff & \exists \mathbf{cd}{:}(\mathrm{D}_1\to\mathrm{D}_2), \mathbf{d}_1{:}\mathrm{D}_1.\, \mathbf{d}_2 = \mathrm{ap}(\mathbf{cd}, \mathbf{d}_1) \\
 & & \wedge\, e, s, \mathbf{FT}, \mathbf{cd} \models_{\mathrm{D}_1\to\mathrm{D}_2} P_1 \wedge e, s, \mathbf{FT}, \mathbf{d}_1 \models_{\mathrm{D}_1} P_2 \\
e, s, \mathbf{FT}, \mathbf{d}_1 \models_{\mathrm{D}_1} P_1 \circ\!\!-_{\mathrm{D}_2} P_2 & \iff & \forall \mathbf{cd}{:}(\mathrm{D}_1\to\mathrm{D}_2).\, (e, s, \mathbf{FT}, \mathbf{cd} \models_{\mathrm{D}_1\to\mathrm{D}_2} P_1 \wedge \\
 & & \mathrm{ap}(\mathbf{cd}, \mathbf{d}_1)\downarrow) \Rightarrow e, s, \mathbf{FT}, \mathrm{ap}(\mathbf{cd}, \mathbf{d}_1) \models_{\mathrm{D}_2} P_2 \\
e, s, \mathbf{FT}, \mathbf{cd}_2 \models_{\mathrm{D}_1\to\mathrm{D}_2} P_1 -\!\!\circ\, P_2 & \iff & \forall \mathbf{d}_1{:}\mathrm{D}_1.\, e, s, \mathbf{FT}, \mathbf{d}_1 \models_{\mathrm{D}_1} P_1 \wedge \mathrm{ap}(\mathbf{cd}_2, \mathbf{d}_1)\downarrow \\
 & & \Rightarrow e, s, \mathbf{FT}, \mathrm{ap}(\mathbf{cd}_2, \mathbf{d}_1) \models_{\mathrm{D}_2} P_2
\end{array}$$

Fig. 9. Satisfaction Relation for Structural Formulae

The standard classical connectives 'true', $\wedge$, $\vee$, $\neg$, $\forall$ are all derivable. We introduce notation for expressing 'somewhere, potentially deep down' $\Diamond_{\mathrm{D}_1\to\mathrm{D}_2}P$ and 'everywhere' $\Box_{\mathrm{D}_1\to\mathrm{D}_2}P$, where $\mathrm{D}_1, \mathrm{D}_2 \in \{\mathrm{T}, \mathrm{F}, \mathrm{G}, \mathrm{S}\}$. Similarly, we define the related concept of 'somewhere at this forest-level' $\Diamond_{\otimes}P$: and 'everywhere at this forest-level' $\Box_{\otimes}P$: We also introduce notation for saying 'any node' $\mathrm{true}_{\mathrm{NODE}}$:

$$\begin{array}{ll}
\Diamond_{\mathrm{D}_1\to\mathrm{D}_2}P & \triangleq \mathrm{true}_{\mathrm{D}_1\to\mathrm{D}_2} \circ_{\mathrm{D}_1} P \\
\Diamond_{\otimes}P & \triangleq (\mathrm{true}_{\mathrm{D}} \otimes_{\mathrm{D}} -_{\mathrm{D}} \otimes_{\mathrm{D}} \mathrm{true}_{\mathrm{D}}) \circ_{\mathrm{D}} P \\
\Box_{\mathrm{D}_1\to\mathrm{D}_2}P & \triangleq \neg\Diamond_{\mathrm{D}_1\to\mathrm{D}_2}\neg P \\
\Box_{\otimes}P & \triangleq \neg\Diamond_{\otimes}\neg P \\
\mathrm{true}_{\mathrm{NODE}} & \triangleq \mathrm{true}_{\mathrm{ELE}} \vee \mathrm{true}_{\mathrm{DOC}} \vee \mathrm{true}_{\mathrm{TXT}} \vee \mathrm{true}_{\mathrm{ATTR}}
\end{array}$$

5 Resource Reasoning

We now introduce our reasoning system, based upon the resource reasoning of O'Hearn [IO01]. We seek to reason in a component wise fashion, so that service authors can prove their code independently of clients, who themselves need no access to the service code to ensure validity. We use the concept of *documented specifications* to achieve this. Parties prove their work in the context of a set of assumptions for the behaviour of other components, analogous to the engineering concept of API documentation. If all parties provide these documented specifications only when they have proven them correct, and if all the assumptions meet, then soundness of the overall proof is assured.

A *function specification* is a tuple of the form

$$(f, \mathbf{P}, \mathbf{Q}, \overline{\mathbf{p}}^m, \bar{\mathbf{l}}^n)$$

Where f is from the set of function names, $\mathbf{P}, \mathbf{Q}$ are formulae of our logic and $\overline{\mathbf{p}}^m, \bar{\mathbf{l}}^n$ are lists drawn from the set of valid variable names representing the set

$$
\begin{aligned}
&e, s, \mathbf{FT}, \mathbf{cd} \models_{\mathrm{D}\to\mathrm{D}} -_{\mathrm{D}} && \iff \mathbf{cd} \equiv -_{\mathrm{D}}\\
&e, s, \mathbf{FT}, \mathbf{d} \models_{\mathrm{D}} P_{\mathtt{id}} \lbag\, P' \,\rbag_{\mathtt{aidn}}{}^{\mathtt{irn}}_{\mathtt{tp}}[P'']_{\mathtt{fid}}\mathbf{null} && \iff\\
&\qquad \exists \mathbf{s}{:}\mathrm{S}, \mathbf{ea}{:}\mathrm{EA}, \mathbf{d}''{:}\mathrm{D}''.\\
&\qquad (\mathbf{d} \equiv \mathbf{s}_{\llbracket\mathtt{id}\rrbracket_s} \lbag\, \mathbf{ea} \,\rbag_{\llbracket\mathtt{aidn}\rrbracket_s}{}^{\llbracket\mathtt{irn}\rrbracket_s}_{\llbracket\mathtt{tp}\rrbracket_s}[\mathbf{d}'']_{\llbracket\mathtt{fid}\rrbracket_s}\mathbf{null}) \land\\
&\qquad e, s, \mathbf{FT}, \mathbf{s} \models_{\mathrm{S}} P \land e, s, \mathbf{FT}, \mathbf{ea} \models_{\mathrm{EA}} P' \land\\
&\qquad e, s, \mathbf{FT}, \mathbf{d}'' \models''_{\mathrm{D}} P''\\
&e, s, \mathbf{FT}, \mathbf{cd} \models_{\mathrm{D}_1\to\mathrm{D}_2} P_{\mathtt{id}} \lbag\, P' \,\rbag_{\mathtt{aidn}}{}^{\mathtt{irn}}_{\mathtt{tp}}[P'']_{\mathtt{fid}}\mathbf{null} && \iff\\
&\qquad \begin{pmatrix} \exists \mathbf{s}{:}\mathrm{S}, \mathbf{c}{:}\mathrm{D}_1\to\mathrm{EA}, \mathbf{d"}{:}\mathrm{D"}.\\
(\mathbf{cd} \equiv \mathbf{s}_{\llbracket\mathtt{id}\rrbracket_s} \lbag\, \mathbf{c} \,\rbag_{\llbracket\mathtt{aidn}\rrbracket_s}{}^{\llbracket\mathtt{irn}\rrbracket_s}_{\llbracket\mathtt{tp}\rrbracket_s}[\mathbf{d"}]_{\llbracket\mathtt{fid}\rrbracket_s}\mathbf{null}) \land\\
e, s, \mathbf{FT}, \mathbf{s} \models_{\mathrm{S}} P \land e, s, \mathbf{FT}, \mathbf{c} \models_{\mathrm{D}_1\to\mathrm{EA}} P' \land\\
e, s, \mathbf{FT}, \mathbf{d"} \models_{\mathrm{D"}} P'' \end{pmatrix}\\
&\qquad \lor \begin{pmatrix} \exists \mathbf{s}{:}\mathrm{S}, \mathbf{ea}{:}\mathrm{EA}, \mathbf{c}{:}\mathrm{D}_1\to\mathrm{D"}.\\
(\mathbf{cd} \equiv \mathbf{s}_{\llbracket\mathtt{id}\rrbracket_s} \lbag\, \mathbf{ea} \,\rbag_{\llbracket\mathtt{aidn}\rrbracket_s}{}^{\llbracket\mathtt{irn}\rrbracket_s}_{\llbracket\mathtt{tp}\rrbracket_s}[\mathbf{c}]_{\llbracket\mathtt{fid}\rrbracket_s}\mathbf{null}) \land\\
e, s, \mathbf{FT}, \mathbf{s} \models_{\mathrm{S}} P \land e, s, \mathbf{FT}, \mathbf{ea} \models_{\mathrm{EA}} P' \land\\
e, s, \mathbf{FT}, \mathbf{c} \models_{\mathrm{D}_1\to\mathrm{D"}} P'' \end{pmatrix}\\
&e, s, \mathbf{FT}, \mathbf{d} \models_{\mathrm{D}} \text{“\#text”}_{\mathtt{id}} \lbag\, \varnothing_{\mathrm{EA}} \,\rbag_{\mathbf{null}_3}{}^{\mathtt{idref}}[\varnothing_{\mathrm{TF}}]_{\mathtt{fid}}P && \iff\\
&\qquad \exists \mathbf{s}{:}\mathrm{S}.\\
&\qquad (\mathbf{d} \equiv \text{“\#text”}_{\llbracket\mathtt{id}\rrbracket_s} \lbag\, \varnothing_{\mathrm{EA}} \,\rbag_{\mathbf{null}_3}{}^{\llbracket\mathtt{idref}\rrbracket_s}[\varnothing_{\mathrm{TF}}]_{\llbracket\mathtt{fid}\rrbracket_s}\mathbf{s} \land\\
&\qquad e, s, \mathbf{FT}, \mathbf{s} \models_{\mathrm{S}} P)\\
&e, s, \mathbf{FT}, \mathbf{cd} \models_{\mathrm{D}_1\to\mathrm{D}_2} \text{“\#text”}_{\llbracket\mathtt{id}\rrbracket_s} \lbag\, \varnothing_{\mathrm{EA}} \,\rbag_{\mathbf{null}_3}{}^{\llbracket\mathtt{idref}\rrbracket_s}[\varnothing_{\mathrm{TF}}]_{\llbracket\mathtt{fid}\rrbracket_s}P && \iff\\
&\qquad \exists \mathbf{c}{:}\mathrm{D}_1\to\mathrm{S}.\\
&\qquad (\mathbf{cd} \equiv \text{“\#text”}_{\llbracket\mathtt{id}\rrbracket_s} \lbag\, \varnothing_{\mathrm{EA}} \,\rbag_{\mathbf{null}_3}{}^{\llbracket\mathtt{idref}\rrbracket_s}[\varnothing_{\mathrm{TF}}]_{\llbracket\mathtt{fid}\rrbracket_s}\mathbf{c} \land\\
&\qquad e, s, \mathbf{FT}, \mathbf{c} \models_{\mathrm{D}_1\to\mathrm{S}} P)
\end{aligned}
$$

$$
\begin{aligned}
&e, s, \mathbf{FT}, \mathbf{c} \models_{\mathrm{C}} \mathbf{c}' && \iff \mathbf{c} = \mathbf{c}'\\
&e, s, \mathbf{FT}, \mathbf{d} \models_{\mathrm{D}} P' \otimes_{\mathrm{D}} P'' && \iff \exists \mathbf{d}'{:}\mathrm{D}, \mathbf{d}''{:}\mathrm{D}.\, (\mathbf{d} \equiv \mathbf{d}' \otimes_{\mathrm{D}} \mathbf{d}'') \land\\
& && \qquad e, s, \mathbf{FT}, \mathbf{d}' \models_{\mathrm{D}} P' \land e, s, \mathbf{FT}, \mathbf{d}'' \models_{\mathrm{D}} P''\\
&e, s, \mathbf{FT}, \mathbf{cd} \models_{\mathrm{D}_1\to\mathrm{D}_2} P' \otimes_{\mathrm{D}_2} P'' && \iff \exists \mathbf{cd}'{:}(\mathrm{D}_1\to\mathrm{D}_2), \mathbf{d}{:}\mathrm{D}_2.\\
& && \qquad ((\mathbf{cd} \equiv \mathbf{cd}' \otimes_{\mathrm{D}_2} \mathbf{d}) \land e, s, \mathbf{FT}, \mathbf{cd}' \models_{\mathrm{D}_1\to\mathrm{D}_2} P' \land e, s, \mathbf{FT}, \mathbf{d} \models_{\mathrm{D}_2} P'') \lor\\
& && \qquad ((\mathbf{cd} \equiv \mathbf{d} \otimes_{\mathrm{D}_2} \mathbf{cd}') \land e, s, \mathbf{FT}, \mathbf{d} \models_{\mathrm{D}_2} P' \land e, s, \mathbf{FT}, \mathbf{cd}' \models_{\mathrm{D}_1\to\mathrm{D}_2} P'')\\
&e, s, \mathbf{FT}, \mathbf{d} \models_{\mathrm{D}} P' \oplus_{\mathrm{D}} P'' && \iff \exists \mathbf{d}'{:}\mathrm{D}, \mathbf{d}''{:}\mathrm{D}.\, (\mathbf{d} \equiv \mathbf{d}' \oplus_{\mathrm{D}} \mathbf{d}'') \land\\
& && \qquad e, s, \mathbf{FT}, \mathbf{d}' \models_{\mathrm{D}} P' \land e, s, \mathbf{FT}, \mathbf{d}'' \models_{\mathrm{D}} P''\\
&e, s, \mathbf{FT}, \mathbf{cd} \models_{\mathrm{D}_1\to\mathrm{D}_2} P' \oplus_{\mathrm{D}} P'' && \iff \exists \mathbf{cd}'{:}(\mathrm{D}_1\to\mathrm{D}_2), \mathbf{d}{:}\mathrm{D}_2.\\
& && \qquad (\mathbf{cd} \equiv \mathbf{cd}' \oplus_{\mathrm{D}} \mathbf{d}) \land e, s, \mathbf{FT}, \mathbf{cd}' \models_{\mathrm{D}_1\to\mathrm{D}_2} P' \land e, s, \mathbf{FT}, \mathbf{d} \models_{\mathrm{D}_2} P''\\
&e, s, \mathbf{FT}, \mathbf{d} \models_{\mathrm{D}} \varnothing_{\mathrm{D}} && \iff \mathbf{d} \equiv \varnothing_{\mathrm{D}}\\[1ex]
&e, s, \mathbf{FT}, \mathbf{d} \models_{\mathrm{D}} {<}P{>}_{\mathrm{D}} && \iff \exists \mathbf{d}'{:}\mathrm{D}'.\, (\mathbf{d} \equiv {<}\mathbf{d}'{>}_{\mathrm{D}}) \land e, s, \mathbf{FT}, \mathbf{d}' \models_{\mathrm{D}'} P\\
&e, s, \mathbf{FT}, \mathbf{cd} \models_{\mathrm{D}_1\to\mathrm{D}_2} {<}P{>}_{\mathrm{D}_2} && \iff \exists \mathbf{cd}'{:}(\mathrm{D}_1\to\mathrm{D}_2').\, (\mathbf{cd}{\equiv}{<}\mathbf{cd}'{>}_{\mathrm{D}_2}) \land e, s, \mathbf{FT}, \mathbf{cd}' \models_{\mathrm{D}_1\to\mathrm{D}_2'} P
\end{aligned}
$$

$$
\begin{aligned}
&e, s, \mathbf{FT}, \mathbf{attr} \models_{\mathrm{ATTR}} \ll P_{\mathtt{id}} \mapsto [P']_{\mathtt{fid}} \gg^{\mathtt{idref}} && \iff\\
&\qquad \exists \mathbf{s}{:}\mathrm{S}, \mathbf{af}{:}\mathrm{AF}.\, (\mathbf{attr}{\equiv}{\ll}\mathbf{s}_{\llbracket\mathtt{id}\rrbracket_s} \mapsto [\mathbf{af}]_{\llbracket\mathtt{fid}\rrbracket_s} \gg^{\llbracket\mathtt{idref}\rrbracket_s})\\
&\qquad \land e, s, \mathbf{FT}, \mathbf{s} \models_{\mathrm{S}} P \land e, s, \mathbf{FT}, \mathbf{af} \models_{\mathrm{AF}} P'\\[1ex]
&e, s, \mathbf{FT}, \mathbf{cattr} \models_{\mathrm{D}_1\to\mathrm{ATTR}} \ll P_{\mathtt{id}} \mapsto [P']_{\mathtt{fid}} \gg^{\mathtt{idref}} && \iff\\
&\qquad \exists \mathbf{s}{:}\mathrm{S}, \mathbf{caf}{:}\mathrm{D}_1\to\mathrm{AF}.\, (\mathbf{attr}{\equiv}{\ll}\mathbf{s}_{\llbracket\mathtt{id}\rrbracket_s} \mapsto [\mathbf{caf}]_{\llbracket\mathtt{fid}\rrbracket_s} \gg^{\llbracket\mathtt{idref}\rrbracket_s})\\
&\qquad \land e, s, \mathbf{FT}, \mathbf{s} \models_{\mathrm{S}} P \land e, s, \mathbf{FT}, \mathbf{caf} \models_{\mathrm{D}_1\to\mathrm{AF}} P'
\end{aligned}
$$

Fig. 10. Satisfaction Relation for Data Structure Specific Formulae

$$
\begin{aligned}
&e, s, \mathbf{FT}, \mathbf{d} \models_{\mathrm{D}} \mathtt{var}_{\mathrm{E}} && \iff \mathbf{d} \equiv e(\mathtt{var}_{\mathrm{E}})\\
&e, s, \mathbf{FT}, \mathbf{d} \models_{\mathrm{D}} \mathtt{Exp}_{\mathrm{V}} = \mathtt{Exp}'_{\mathrm{V}} && \iff \llbracket\mathtt{Exp}_{\mathrm{V}}\rrbracket_s = \llbracket\mathtt{Exp}'_{\mathrm{V}}\rrbracket_s\\
&e, s, \mathbf{FT}, \mathbf{d} \models_{\mathrm{D}} \mathtt{Int} = |\mathtt{f}| && \iff \llbracket\mathtt{Int}\rrbracket_s = \mathrm{len}(e(\mathtt{f}))\\
&e, s, \mathbf{FT}, \mathbf{a} \models_{\mathrm{A}} \exists \mathtt{var}_{\mathrm{E}}.\, P && \iff \exists \mathbf{d}'.\, e[\mathtt{var}_{\mathrm{E}} \mapsto \mathbf{d}'], s, \mathbf{FT}, \mathbf{a} \models_{\mathrm{A}} P\\
&e, s, \mathbf{FT}, \mathbf{a} \models_{\mathrm{A}} \exists \mathtt{var}_{\mathrm{V}}.\, P && \iff \exists \mathbf{v}.\, e, s[\mathtt{var}_{\mathrm{V}} \mapsto \mathbf{v}], \mathbf{FT}, \mathbf{a} \models_{\mathrm{A}} P\\
&e, s, \mathbf{FT}, \mathbf{d} \models_{\mathrm{D}} \gamma(\mathbf{f}) && \iff \exists \mathbf{c}, \overline{\mathbf{p}}^m, \bar{\mathbf{l}}^n.\, \mathbf{FT}(\mathbf{f}) = (\mathbf{C}, \overline{\mathbf{p}}^m, \bar{\mathbf{l}}^n)\\
&e, s, \mathbf{FT}, \mathbf{d} \models_{\mathrm{D}} \omega_{\mathrm{D}}(\mathbf{uri}) && \iff \exists \mathbf{data}, \mathbf{code}.\, \pi(\mathbf{uri}) = (\mathbf{data}, \mathbf{code}) \land \mathbf{data} \neq \varnothing\\
&e, s, \mathbf{FT}, \mathbf{d} \models_{\mathrm{D}} \omega_{\mathrm{C}}(\mathbf{uri}) && \iff \exists \mathbf{data}, \mathbf{code}.\, \pi(\mathbf{uri}) = (\mathbf{data}, \mathbf{code}) \land \mathbf{code} \neq \varnothing
\end{aligned}
$$

where $\mathbf{f}{:}\mathrm{F}$, $\mathrm{F} \in \{\mathrm{DE}, \mathrm{EA}, \mathrm{EF}, \mathrm{AF}, \mathrm{S}\}$

Fig. 11. Satisfaction Relation for the Remaining Formulae

of parameter names and local variables used. Each represents the documented description of a given function, that will be published to consumers. We write sets of function specifications as $\mathcal{D}$. A set of function specifications is well formed only if there is at most one entry for each `f` and $(\text{free}(\mathbf{P}) \cup \text{free}(\mathbf{Q})) \cap \bar{\mathbf{l}}^m = \varnothing$, stating that no free names in the specification are within the local variables (explained shortly in context of the function call rules).

Web resource specifications are tuples of the forms

$$(uri_{\mathrm{D}}, \varnothing, \mathbf{Q}) \quad (uri_{\mathrm{C}}, \mathbf{P}, \mathbf{Q})$$

The former represents for web data, the later web code, where $uri \in \mathbf{URI}$ and $\mathbf{P}, \mathbf{Q}$ are formulae of our logic. We use a reserved name `this` in the specifications for web data, which will always refer to the document node being introduced.

These tuples represent published documentation for the descriptions of remote data, and behaviour of remote code. We write sets of these as $\mathcal{W}$, which are well formed only if each *uri* occurs at most once with a given D or C annotation. Note the precondition for web data introduction is vacuous; adding data to the environment is always a valid operation.

Our reasoning system is based upon Hoare Logic. We use an implicit global $\mathcal{D}$ and $\mathcal{W}$, giving judgements of the form

$$\overline{\{\mathbf{P}\}\mathtt{C}\{\mathbf{Q}\}}$$

This states that given our assumed sets of *demanded* function specifications $\mathcal{D}$, and assumed web specifications $\mathcal{W}$, the triple holds. We use a triple based upon O'Hearn's fault avoiding interpretation, stating that any code satisfying a specification cannot reduce to a fault state and, if it does not diverge, will leave a state satisfying the postcondition. This interpretation is identical to that in [GSWZ08b]. We adopt the standard Hoare Logic rules for skip, assignment, disjunction, conditionals, while, sequencing, implication and variable elimination. In addition, we use the *Frame Rule.*

$$\frac{\{P\}C\{Q\}}{\{K \circ_{\mathrm{D}} P\}C\{K \circ_{\mathrm{D}} Q\}} \text{ (If C does not modify K's free variables)}$$

The frame rule allows a triple to be extended by an arbitrary *frame*, so long as the variables mentioned in the frame are not modified by the code of the triple. This rule is a key concept in resource reasoning, as it allows us to "frame off" parts of the environment that are superfluous to the analysis in question. The semantic of $\circ$ ensures that this removed environment remains untouched.

The notion of free variables in a formulae K is standard, as is the set of variables modified by most of our commands. Of note is the set for function call, where we define mods(`v = f(e1, ..., en)`) to be the set modified by the body code of the function `f`.

Function Reasoning The function introduction rule performs both the sanity check that the specification provided in the documentation is satisfied by the function being introduced, as well as the act of asserting the runtime existence via γ.

$$\frac{\{\mathbf{P}\}\texttt{FStatement; return Expr;}\{\mathbf{Q}\}}{\begin{array}{c}\{\varnothing_{\mathrm{G}}\}\\ \texttt{function f(p1, ..., pn)}\\ \{\texttt{local l1;...;local ln;FStatement; return Expr;}\}\\ \{\gamma(\texttt{f}) \wedge \varnothing_{\mathrm{G}}\}\end{array}} \quad (\texttt{f}, \mathbf{P}, \mathbf{Q}, \overline{\mathbf{p}}^{m}, \bar{\mathbf{l}}^{n}) \in \mathcal{D}$$

Return statements simply assign to a reserved variable ρ.

$$\frac{}{\{\varnothing\}\texttt{return Expr}\{\rho = \texttt{Expr}\}}$$

The function call rule precondition is a simple check that the desired function has been introduced, along with a check that the precondition of the function is met. It is important that the parameter and local names, which are private to the function body, do not alter the environment of the call site. Our choice that global names must be disjoint from both parameter and local names prevents clashes with global variables that may be used. We achieve locality of parameter names with a substitution instance θ, replacing parameter names found in the specification with the expressions passed in the call (the restriction on parameter assignment ensures this is meaningful). Our restriction that free program variables found in function specifications cannot contain local variable names ensures that local names will not leak from the function. These last two restrictions are reminiscent of Parkinson's work in [Par05].

Let $\theta = \{\texttt{e}_1/\mathbf{p}_1, \ldots, \texttt{e}_m/\mathbf{p}_m\}$ be the substitution of all parameter names for the call site passed expressions.

$$\frac{}{\{\gamma(\texttt{f}) \wedge \theta(\mathbf{P})\}\texttt{v = f(e1, ..., em)}\{\gamma(\texttt{f}) \wedge \theta(\mathbf{Q})[\texttt{v}/\rho]\}} \quad (\texttt{f}, \mathbf{P}, \mathbf{Q}, \overline{\mathbf{p}}^{m}, \bar{\mathbf{l}}^{n}) \in \mathcal{D}$$

Web data and code reasoning The web data rule first ensures that the requested URI is available, then uses the data description stored in the documentation to assert the existence of the imported data.

$$\frac{}{\{\omega_{\mathrm{D}}(\texttt{uri})\}\texttt{n} = \texttt{fetchDocument}(\texttt{uri})\{\omega_{\mathrm{D}}(\texttt{uri}) \wedge \mathbf{Q}[\texttt{n}/\texttt{this}]\}} \quad (\texttt{uri}_{\mathrm{D}}, \varnothing, \mathbf{Q}) \in \mathcal{W}$$

The rule for script execution is a simple check that the given URI refers to a script, followed by propagation of the documented specification.

$$\frac{}{\{\omega_{\mathrm{C}}(\texttt{uri}) \wedge \mathbf{P}\}\texttt{runScript}(\texttt{uri})\{\omega_{\mathrm{C}}(\texttt{uri}) \wedge \mathbf{Q}\}} \quad (\texttt{uri}_{\mathrm{C}}, \mathbf{P}, \mathbf{Q}) \in \mathcal{W}$$

DOM Reasoning The effects of our language commands on the environment are captured by a series of axioms. In line with the local reasoning ethos, the specifications speak only to the footprint of the command in as small a fashion as we can render it. Typical use then requires the construction of a frame and use of the frame rule to apply the axiom in a given proof. We present here a subset of the axioms, choosing those that we use in our example, along with some illustrative cases. Many of the additional cases are similar (see [PG]).

$$\begin{gathered}
\{<\text{``\#document''}_{\mathtt{doc}} \langle\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\rangle_{\mathbf{null}}{}_{9}^{\mathbf{null}}[\mathtt{DE}]_{\mathtt{fid}}\mathbf{null}>_{\mathrm{G}} \wedge \mathtt{x} = \mathtt{y} \wedge \mathtt{name} \neq \mathbf{null} \wedge \text{`\#'} \notin \mathtt{name}\} \\
\mathtt{x} = \mathtt{createElement}(\mathtt{doc}, \mathtt{name}) \\
\left\{ \begin{array}{c} <\text{``\#document''}_{\mathtt{doc}\{\mathtt{y}/\mathtt{x}\}} \langle\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\rangle_{\mathbf{null}}{}_{9}^{\mathbf{null}}[\mathtt{DE}]_{\mathtt{fid}}\mathbf{null}>_{\mathrm{G}} \oplus \\ <\mathtt{name}_{\mathtt{x}} \langle\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\rangle_{\mathtt{aid'}}{}_{1}^{\mathtt{doc}}[\varnothing_{\mathrm{EF}}]_{\mathtt{fid'}}\mathbf{null}>_{\mathrm{G}} \end{array} \right\}
\end{gathered}$$

The precondition requires that the given doc expression identifies a document, with some document element captured in the logical variable DE. The given name cannot be null, nor contain the # character. We capture the current value of variable we will overwrite by assignment, which we use to ensure that if the program used the value in identifying the document, the result still makes sense. The postcondition shows that the document structure described in the precondition is unchanged (as assured by the logical variable DE), but that the DOM has been extended with a new element with the given name. The other `create` command cases are similar.

$$\begin{gathered}
\{\mathtt{name}_{\mathtt{n}} \langle\!\!\{ \mathtt{EA} \}\!\!\rangle_{\mathtt{aidn}}{}_{\mathtt{tp}}^{\mathtt{irn}}[\mathtt{F}]_{\mathtt{fid}}\mathtt{val} \wedge \mathtt{kids} = \mathtt{y}\} \\
\mathtt{kids} = \mathtt{getChildNodes}(\mathtt{n}) \\
\{\mathtt{name}_{\mathtt{n}\{\mathtt{y}/\mathtt{kids}\}} \langle\!\!\{ \mathtt{EA} \}\!\!\rangle_{\mathtt{aidn}}{}_{\mathtt{tp}}^{\mathtt{irn}}[\mathtt{F}]_{\mathtt{fid}}\mathtt{val} \wedge \mathtt{kids} = \mathtt{fid}\}
\end{gathered}$$

$$\begin{gathered}
\{\ll \mathtt{name}_{\mathtt{n}} \mapsto [\mathtt{af}]_{\mathtt{fid}} \gg^{\mathtt{doc}} \wedge \mathtt{kids} = \mathtt{y}\} \\
\mathtt{kids} = \mathtt{getChildNodes}(\mathtt{n}) \\
\{\ll \mathtt{name}_{\mathtt{n}\{\mathtt{y}/\mathtt{kids}\}} \mapsto [\mathtt{af}]_{\mathtt{fid}} \gg^{\mathtt{doc}} \wedge \mathtt{kids} = \mathtt{fid}\}
\end{gathered}$$

`getChildNodes` is representative of all the `get` commands. We need a case for both general elements, as well as attributes (due to their different notation). Both require that the given node identifier n corresponds to a node of the correct shape and both capture the properties of that node in logical variables. The postcondition shows the node is unchanged, but the variable under assignment has taken the identifier of the child forest.

$$\begin{gathered}
\left\{ \begin{array}{l} (\varnothing_{\mathrm{D}_1} \multimap (\mathtt{gc} \circ_{\mathrm{D}_2} \mathtt{s}_{\mathtt{parent}} \langle\!\!\{ \mathtt{EA} \}\!\!\rangle_{\mathtt{aid}}{}_{\mathbf{1}}^{\mathtt{idref}}[\mathtt{F}]_{\mathtt{fid}}\mathbf{null})) \\ \circ_{\mathrm{D}_1} (\mathtt{name}'_{\mathtt{newChild}} \langle\!\!\{ \mathtt{EA}' \}\!\!\rangle_{\mathtt{aidn}'}{}_{\mathtt{ettp}}^{\mathtt{idref'}}[\mathtt{F}']_{\mathtt{fid}'}\mathtt{val}') \end{array} \right\} \\
\mathtt{n} = \mathtt{appendChild}(\mathtt{parent}, \mathtt{newChild}) \\
\left\{ \begin{array}{l} (\mathtt{gc} \circ_{\mathrm{D}_2} \\ \mathtt{s}_{\mathtt{parent}} \langle\!\!\{ \mathtt{EA} \}\!\!\rangle_{\mathtt{aid}}{}_{\mathbf{1}}^{\mathtt{idref}} \begin{bmatrix} \mathtt{f} \otimes \\ <\mathtt{name}'_{\mathtt{newChild}} \langle\!\!\{ \mathtt{EA}' \}\!\!\rangle_{\mathtt{aidn}'}{}_{\mathtt{ettp}}^{\mathtt{idref'}}[\mathtt{f}']_{\mathtt{fid}'}\mathtt{val}'>_{\mathrm{EF}} \end{bmatrix}_{\mathtt{fid}} \mathtt{val}) \end{array} \right\} \\
\text{where } \mathtt{ettp} \in \{1, 3\}
\end{gathered}$$

The precondition of `appendChild` uses the adjoint operator to ensure that newChild cannot be an ancestor node of parent by splitting the data into a context, and the child node. The context must satisfy the property that putting the empty tree into the hole would leave a tree splittable into an arbitrary context, and the parent node.

$$\left\{\begin{array}{l}\mathtt{name}_{\mathtt{id}} \lbrace\!\lbrack\, \mathtt{EA} \,\rbrack\!\rbrace_{\mathtt{aidn}}{}^{\mathtt{docn}}_{\mathtt{tp}}[\mathtt{F}_1 \otimes \\ \quad <\mathtt{name}'_{\mathtt{id}'} \lbrace\!\lbrack\, \mathtt{EA}' \,\rbrack\!\rbrace_{\mathtt{aidn}'}{}^{\mathtt{doc}}_{\mathtt{tp}'}[\mathtt{F}']_{\mathtt{fid}'}\mathtt{val}'>_{\mathrm{D}} \\ \quad \otimes \mathtt{F}_2]_{\mathtt{list}}\mathtt{val} \wedge |\mathtt{F}_1| = \mathtt{int} \wedge \mathtt{list} = \mathtt{y}\end{array}\right\}$$
$$\mathtt{n} = \mathtt{item}(\mathtt{list}, \mathtt{int})$$
$$\left\{\begin{array}{l}\mathtt{name}_{\mathtt{id}} \lbrace\!\lbrack\, \mathtt{EA} \,\rbrack\!\rbrace_{\mathtt{aidn}}{}^{\mathtt{docn}}_{\mathtt{tp}}[\mathtt{F}_1 \otimes \\ \quad <\mathtt{name}'_{\mathtt{id}'} \lbrace\!\lbrack\, \mathtt{EA}' \,\rbrack\!\rbrace_{\mathtt{aidn}'}{}^{\mathtt{doc}}_{\mathtt{tp}'}[\mathtt{F}']_{\mathtt{fid}'}\mathtt{val}'>_{\mathrm{D}} \\ \quad \otimes \mathtt{F}_2]_{\mathtt{list}\{\mathtt{y}/\mathtt{n}\}}\mathtt{val} \wedge \mathtt{n} = \mathtt{id}'\end{array}\right\}$$

$$\left\{\mathtt{name}_{\mathtt{id}} \lbrace\!\lbrack\, \mathtt{EA} \,\rbrack\!\rbrace_{\mathtt{aidn}}{}^{\mathtt{docn}}_{\mathtt{tp}}[\mathtt{F}]_{\mathtt{list}}\mathtt{val} \wedge (|\mathtt{F}| \leq \mathtt{int} \vee \mathtt{int} < 0) \wedge \mathtt{list} = \mathtt{y}\right\}$$
$$\mathtt{n} = \mathtt{item}(\mathtt{list}, \mathtt{int})$$
$$\left\{\mathtt{name}_{\mathtt{id}} \lbrace\!\lbrack\, \mathtt{EA} \,\rbrack\!\rbrace_{\mathtt{aidn}}{}^{\mathtt{docn}}_{\mathtt{tp}}[\mathtt{F}]_{\mathtt{list}\{\mathtt{y}/\mathtt{n}\}}\mathtt{val} \wedge \mathtt{n} = \mathbf{null}\right\}$$

The `item` command also requires two cases, one to handle the index being beyond the ranges of the list, and one when it falls within range.

As with previous work ([GSWZ08b], [GSWZ08a]), the weakest preconditions can be derived, showing that the reasoning system is complete for straight line code.

6 Example

We now demonstrate our language and reasoning in the development of both a mashup service, and a consuming client. The service is a mapping system for a college campus. Our client mashup will be a student homepage, containing a college map with annotations pointing out locations of particular importance to the student in question.

The Map Service We define an XML data structure for communicating pure map data, free of specific user interface presentation. A map is presented as a hierarchy of *areas*, each having an associated name, pair of spatial coordinates and image URI. Areas also have zero or more *overlays*, each consisting of text data, with an optional associated image URI.

In our example, the root area represents the campus, with sub-areas for each building, which in turn may contain sub areas, and so on. Map data will typically be served without any overlays, which are then added by a mashup consumer to convey their domain specific additions. We define this data structure using a predicate (Fig 12) that fulfils the role typically taken by XML Schema or a DTD. This has the advantage of both providing a concise schema for the document, as well as the basis we need to reason about it. Note that the map predicate is exact, so that additional non-map data cannot be incorporated into any data that satisfies it.

To aid consumers in the use of the map data, we provide two associated functions. The first, `addOverlay`, adds a new overlay to a named area within the map. Its specification gives detail about the structural changes to a given map, without providing any information on how it is accomplished. The second

$$\mathrm{map}(\mathbf{id}) \triangleq \text{“map”}_{\mathbf{id}} \begin{array}{l} \Lbag \begin{array}{l} \ll \text{“title”} \mapsto \text{“\#text”}_{\mathbf{id2}} \Lbag \varnothing_{\mathrm{EA}} \Rbag_{\mathbf{null}}{}^{\mathbf{did}}_{3} [\varnothing_{\mathrm{TF}}]_{\mathbf{fid2}}\ \mathbf{s1} \gg^{\mathbf{did}} \otimes \\ \ll \text{“image”} \mapsto \text{“\#text”}_{\mathbf{id3}} \Lbag \varnothing_{\mathrm{EA}} \Rbag_{\mathbf{null}}{}^{\mathbf{did}}_{3} [\varnothing_{\mathrm{TF}}]_{\mathbf{fid3}}\ \mathbf{s2} \gg^{\mathbf{did}} \otimes \\ \ll \text{“width”} \mapsto \text{“\#text”}_{\mathbf{id4}} \Lbag \varnothing_{\mathrm{EA}} \Rbag_{\mathbf{null}}{}^{\mathbf{did}}_{3} [\varnothing_{\mathrm{TF}}]_{\mathbf{fid4}}\ \mathbf{s3} \gg^{\mathbf{did}} \otimes \\ \ll \text{“height”} \mapsto \text{“\#text”}_{\mathbf{id5}} \Lbag \varnothing_{\mathrm{EA}} \Rbag_{\mathbf{null}}{}^{\mathbf{did}}_{3} [\varnothing_{\mathrm{TF}}]_{\mathbf{fid5}}\ \mathbf{s4} \gg^{\mathbf{did}} \end{array} \Rbag_{\mathbf{aid}}{}^{\mathbf{did}}_{1} \\ [\mathrm{area}]_{\mathbf{fid}}\ \mathbf{null} \\ \land \mathrm{is\_int}(\mathbf{s3}) \land \mathrm{is\_int}(\mathbf{s4}) \end{array}$$

$$\mathrm{area} \triangleq \square_{\otimes} {<}\mathrm{true}_{\mathrm{NODE}}{>} \implies$$

$$\left(\begin{array}{l} \left(\begin{array}{l} \text{“area”}_{\mathbf{id}} \Lbag \begin{array}{l} \ll \text{“name”} \mapsto \text{“\#text”}_{\mathbf{id2}} \Lbag \varnothing_{\mathrm{EA}} \Rbag_{\mathbf{null}}{}^{\mathbf{did}}_{3} [\varnothing_{\mathrm{TF}}]_{\mathbf{fid6}}\ \mathbf{s1} \gg^{\mathbf{did}} \otimes \\ \ll \text{“image”} \mapsto \text{“\#text”}_{\mathbf{id3}} \Lbag \varnothing_{\mathrm{EA}} \Rbag_{\mathbf{null}}{}^{\mathbf{did}}_{3} [\varnothing_{\mathrm{TF}}]_{\mathbf{fid7}}\ \mathbf{s2} \gg^{\mathbf{did}} \otimes \\ \ll \text{“width”} \mapsto \text{“\#text”}_{\mathbf{id4}} \Lbag \varnothing_{\mathrm{EA}} \Rbag_{\mathbf{null}}{}^{\mathbf{did}}_{3} [\varnothing_{\mathrm{TF}}]_{\mathbf{fid8}}\ \mathbf{s3} \gg^{\mathbf{did}} \otimes \\ \ll \text{“height”} \mapsto \text{“\#text”}_{\mathbf{id5}} \Lbag \varnothing_{\mathrm{EA}} \Rbag_{\mathbf{null}}{}^{\mathbf{did}}_{3} [\varnothing_{\mathrm{TF}}]_{\mathbf{fid9}}\ \mathbf{s4} \gg^{\mathbf{did}} \otimes \\ \ll \text{“offsetx”} \mapsto \text{“\#text”}_{\mathbf{id6}} \Lbag \varnothing_{\mathrm{EA}} \Rbag_{\mathbf{null}}{}^{\mathbf{did}}_{3} [\varnothing_{\mathrm{TF}}]_{\mathbf{fid10}}\ \mathbf{s5} \gg^{\mathbf{did}} \otimes \\ \ll \text{“offsety”} \mapsto \text{“\#text”}_{\mathbf{id7}} \Lbag \varnothing_{\mathrm{EA}} \Rbag_{\mathbf{null}}{}^{\mathbf{did}}_{3} [\varnothing_{\mathrm{TF}}]_{\mathbf{fid11}}\ \mathbf{s6} \gg^{\mathbf{did}} \end{array} \Rbag_{\mathbf{aid}}{}^{\mathbf{did}}_{1} \\ \qquad [\mathrm{area}]_{\mathbf{fid}}\ \mathbf{null} \\ \land \mathrm{is\_int}(\mathbf{s3}) \land \mathrm{is\_int}(\mathbf{s4}) \land \mathrm{is\_int}(\mathbf{s5}) \land \mathrm{is\_int}(\mathbf{s6}) \end{array} \right) \\ \lor \mathrm{overlay} \end{array} \right)$$

$$\mathrm{overlay} \triangleq \text{“overlay”}_{\mathbf{id}} \begin{array}{l} \Lbag \ll \text{“image”} \mapsto \text{“\#text”}_{\mathbf{id2}} \Lbag \varnothing_{\mathrm{EA}} \Rbag_{\mathbf{null}}{}^{\mathbf{did}}_{3} [\varnothing_{\mathrm{TF}}]_{\mathbf{fid12}}\ \mathbf{s} \gg^{\mathbf{did}} \Rbag_{\mathbf{aid}}{}^{\mathbf{did}}_{1} \\ \left[\text{“\#text”}_{\mathbf{cid}} \Lbag \varnothing_{\mathrm{EA}} \Rbag_{\mathbf{null}}{}^{\mathbf{did}}_{3} [\varnothing_{\mathrm{TF}}]_{\mathbf{fid13}}\ \mathbf{s} \right]_{\mathbf{fid}}, \mathbf{null} \end{array}$$

Fig. 12. The map schema predicate

function, `layout`, is provided to convert the map data into an HTML presentation fit for display in a web browser. Both functions are specified in Figure 13.

Note that the post-condition of `layout` says only “true” of the data added to the passed HTML element. Formally, a satisfying implementation could insert any structure, containing any data. This allows great flexibility on the part of the map service, as it can change the layout behaviour at any time knowing that no proven client could depend on specific details. For example, it could upgrade a simple pictorial representation of the map to a fully scrolling, zooming Google-style map. However, it means the client must trust the service to satisfy an informal specification quite separate from the formal one we reason about here. The client must trust that the service will only use `layout` to add data which has the same “meaning” as the semantic map data provided.

The pre-condition of `addOverlay` shows an interesting behavioral leak in the map service code. In the act of implementing `addOverlay` and `layout`, we choose to use an auxiliary function `layoutOverlay`. This function is then exposed to any consumer of the service, and appears in the precondition of `addOverlay`. The unfortunate leak is essential in this type of function system, since we must ensure both that it is present, and not accidentally replaced by an alternative implementation.

The pair of the map data and map code forms the entire map service. We export the service specification as documentation, in Figure 14. Using only this documentation, we can implement a client side mashup of the map service. Note that we can see exactly the schema of the data and footprint of code we are exposing from this documentation, so we cannot be surprised by clients taking advantage of structure and behaviour it describes. We may choose to use a weaker (yet still strong enough to prove fault freedom) specification, to hide some of this detail.

The Map Client Our client acts as the consumer of both the map data and service code described in the previous section. We produce a web page containing

$$\left\{\begin{array}{c}\mathrm{map}(\mathtt{id})\ \wedge\\ \mathbf{cg}\circ\left(\text{“area”}_{\mathbf{id2}}\lessdot\begin{array}{l}\mathbf{ea}\otimes\ll\text{“name”}\mapsto\text{“\#text”}_{\mathbf{id3}}\lessdot\varnothing_{\mathrm{EA}}\gtrdot_{\mathbf{null}}{}^{\mathbf{did}}_{3}[\varnothing_{\mathrm{TF}}]_{\mathbf{fid14}}\ \mathtt{name}\gg^{\mathbf{did}}\\ \otimes\ \mathbf{ea}'\\ {[\mathbf{f}]_{\mathbf{fid}}\ \mathbf{null}}\end{array}\gtrdot_{\mathbf{aid}}{}^{\mathbf{did}}_{1}\right)\\ \wedge\ \neg(\mathtt{text}=\mathbf{null})\end{array}\right\}$$

$$\mathrm{addOverlay}(\mathtt{id},\mathtt{name},\mathtt{text},\mathtt{image})$$

$$\left\{\begin{array}{c}\mathrm{map}(\mathtt{id})\\ \wedge\ \mathbf{cg}\circ\left(\begin{array}{l}\text{“area”}_{\mathbf{id2}}\lessdot\mathbf{ea}\otimes\ll\text{“name”}\mapsto\text{“\#text”}_{\mathbf{id3}}\lessdot\varnothing_{\mathrm{EA}}\gtrdot_{\mathbf{null}}{}^{\mathbf{did}}_{3}[\varnothing_{\mathrm{TF}}]_{\mathbf{fid15}}\ \mathtt{name}\gg^{\mathbf{did}}\otimes\mathbf{ea}'\gtrdot_{\mathbf{aid}}{}^{\mathbf{did}}_{1}\\ \left[\begin{array}{l}\mathbf{f}\otimes\\ \left(\text{“overlay”}_{\mathbf{id}}\begin{array}{l}\lessdot\ll\text{“image”}\mapsto\text{“\#text”}_{\mathbf{iid}}\lessdot\varnothing_{\mathrm{EA}}\gtrdot_{\mathbf{null}}{}^{\mathbf{did}}_{3}[\varnothing_{\mathrm{TF}}]_{\mathbf{fid16}}\ \mathbf{s}\gg^{\mathbf{did}}\gtrdot_{\mathbf{aid}}\\ \left[\text{“\#text”}_{\mathbf{id'}}\lessdot\varnothing_{\mathrm{EA}}\gtrdot_{\mathbf{null}}{}^{\mathbf{did}}_{3}[\varnothing_{\mathrm{TF}}]_{\mathbf{fid17}}\ \mathtt{text}\right]_{\mathbf{fid}}\ \mathbf{null}\end{array}\right)\end{array}\right]_{\mathbf{fid}}\ \mathbf{null}\end{array}\right)\\ \wedge\ (\mathbf{s}=\mathtt{image}\vee(\mathtt{image}=\mathbf{null}\wedge\mathbf{s}=\varnothing_{\mathrm{S}}))\end{array}\right\}$$

$$\left\{\begin{array}{c}(\varnothing\multimap(\mathbf{cg}\circ(\mathbf{node}_{\mathtt{mapID}}\lessdot\mathbf{ea}\gtrdot_{\mathbf{aid}}{}^{\mathbf{did}}_{1}[\mathbf{f}]_{\mathbf{fid}}\mathbf{null}\wedge\mathrm{map}(\mathtt{mapID}))))\\ \circ\\ (\mathbf{cg'}\circ\mathbf{node}_{\mathtt{htmlID}}\lessdot\mathbf{ea'}\gtrdot_{\mathbf{aid'}}{}^{\mathbf{did'}}_{1}[\mathbf{f'}]_{\mathbf{fid'}}\mathbf{null})\wedge\gamma(\mathrm{layoutOverlay})\end{array}\right\}$$

$$\mathrm{layout}(\mathtt{mapID},\mathtt{htmlID})$$

$$\left\{\begin{array}{c}(\varnothing\multimap(\mathbf{cg}\circ(\mathbf{node}_{\mathtt{mapID}}\lessdot\mathbf{ea}\gtrdot_{\mathbf{aid}}{}^{\mathbf{did}}_{1}[\mathbf{f}]_{\mathbf{fid}}\mathbf{null}\wedge\mathrm{map}(\mathtt{mapID}))))\\ \circ\\ (\mathbf{cg'}\circ\mathbf{node}_{\mathtt{htmlID}}\lessdot\mathbf{ea'}\gtrdot_{\mathbf{aid'}}{}^{\mathbf{did'}}_{1}[\mathbf{f'}\otimes\text{“div”}_{\mathbf{id''}}\lessdot\mathrm{true}\gtrdot_{\mathbf{aid''}}{}^{\mathbf{did'}}_{1}[\mathrm{true}]_{\mathbf{fid''}}\mathbf{null}]_{\mathbf{fid'}}\mathbf{null})\wedge\gamma(\mathrm{layoutOverlay})\end{array}\right\}$$

Fig. 13. Function specifications

$$\mathcal{D}=\begin{array}{l}(\mathrm{addOverlay},P_{\mathrm{addOverlay}},Q_{\mathrm{addOverlay}},(\mathtt{id},\mathtt{name},\mathtt{text},\mathtt{image}),(\mathtt{doc},\mathtt{overlayNode},\mathtt{textNode}))\\ (\mathrm{layout},P_{\mathrm{layout}},Q_{\mathrm{layout}},(\mathtt{mapID},\mathtt{htmlID}),(\mathtt{htmlDoc},\mathtt{mapWidth},\mathtt{mapHeight},\mathtt{children}))\\ (\mathrm{layoutOverlay},P',Q',(\mathtt{overlayNode},\mathtt{parentArea}),(\mathtt{imgEle},\mathtt{docNode},\mathtt{overlayDiv}))\end{array}$$

$$\mathcal{W}=\begin{array}{l}(\mathrm{http://www.example.net/mapService/icMap.xml}_{\mathrm{D}},\varnothing,icMap)\\ (\mathrm{http://www.example.net/mapService/mapService.html}_{\mathrm{C}},\varnothing,\gamma(\mathrm{addOverlay})\wedge\gamma(\mathrm{layout})\wedge\gamma(\mathrm{layoutOverlay}))\end{array}$$

$$icMap\triangleq\begin{array}{l}\text{“\#document”}_{\mathtt{id1}}\lessdot\varnothing_{\mathrm{EA}}\gtrdot_{\mathbf{null}}{}^{\mathbf{null}}_{9}[<\text{“map”}_{\mathtt{id2}}\lessdot\varnothing\gtrdot_{\mathtt{aid}}{}^{\mathtt{id1}}_{1}[\mathrm{true}]_{\mathtt{fid2}}\mathbf{null}\wedge\mathrm{map}(\mathtt{id2})>]_{\mathtt{fid1}}\mathbf{null}\\ \quad\wedge\Diamond(\text{“area”}\lessdot\mathbf{ea}\otimes\ll\text{“name”}\mapsto[\#\mathrm{text}[]\text{“Huxley”}]\gg\otimes\mathbf{ea}'\gtrdot[\mathrm{true}])\wedge\\ \quad\Diamond(\text{“area”}\lessdot\mathbf{ea}\otimes\ll\text{“name”}\mapsto[\#\mathrm{text}[]\text{“Sherfield”}]\gg\otimes\mathbf{ea}'\gtrdot[\mathrm{true}])\wedge\\ \quad\Diamond(\text{“area”}\lessdot\mathbf{ea}\otimes\ll\text{“name”}\mapsto[\#\mathrm{text}[]\text{“Business School”}]\gg\otimes\mathbf{ea}'\gtrdot[\mathrm{true}])\end{array}$$

where $P_{\mathrm{addOverlay}}$, $Q_{\mathrm{addOverlay}}$ etc are shown in Figure 13, and P', Q' are elided for brevity.

Fig. 14. Service documentation

the usual HTML, HEAD and BODY elements. Also in the page is a script element containing the Reasonable Web code given in the derivation in Figure 15. This code downloads the campus map from the service, and uses the provided functions to add a number of overlays to the map before displaying it in the page. The derivation in Figure 15 shows that, under the published specifications, the client is fault free. Moreover, it will remain fault free under any changes the service implementation undergoes whilst still satisfying the exported behaviour specifications and data schema formulae.

Consider now a proposed upgrade to the map service, which expands the map data to add an optional data node indicating disabled access to buildings. The map data source no longer satisfies the schema predicate and the service authors are thus aware they have made a potentially breaking change, and publishing it may cause faults in clients using the service. They instead choose to include overlay nodes in the data, indicating the same information. Whilst not originally part of the informal specification, the formal validation allows this. Any proven clients will continue to function in a fault free fashion.

The dynamic nature of the function system allows the mashup consumer to provide its own reimplementation of any imported function. Normally a dangerous idea, note that as long as the reimplementation matches the specification given by the service provider, the operation is sensible and indeed desirable. It provides a safe method for behavioral customisation by clients; in our example, they may wish to customise the HTML layout by overriding the `layout`

$$\left\{\begin{array}{c}\omega_{\mathrm{D}}(\text{“http://example.net/mapService/icMap.xml”})\wedge\\ \omega_{\mathrm{C}}(\text{“http://example.net/mapService/mapService.html”})\wedge\\ \#\mathrm{document}_{\mathrm{this}}[\text{“html”}[\text{“head”}[\mathrm{true}]\text{“body”}[\mathrm{true}]]]\end{array}\right\}$$

$$\left\{\begin{array}{c}\omega_{\mathrm{D}}(\text{“http://www.example.net/mapService/icMap.xml”})\wedge\\ \omega_{\mathrm{C}}(\text{“http://www.example.net/mapService/mapService.html”})\end{array}\right\}$$

mapdoc = fetchDocument(“../mapService/icMap.xml”);

$$\left\{\begin{array}{c}\omega_{\mathrm{D}}(\text{“http://www.example.net/mapService/icMap.xml”})\wedge\\ \omega_{\mathrm{C}}(\text{“http://www.example.net/mapService/mapService.html”})\wedge\\ \Diamond\left(\begin{array}{l}\#\mathrm{document}_{\mathrm{mapdoc}}[\mathbf{node}_{\mathrm{id}}[\mathrm{true}]]\wedge\mathrm{map}(\mathrm{id})\wedge\\ \Diamond(\text{“area”}\lbag\,\mathbf{ea}\otimes\ll\text{“name”}\mapsto[\#\mathrm{text}[\,]\text{“Huxley”}]\gg\otimes\,\mathbf{ea'}\rbag[\mathrm{true}])\wedge\\ \Diamond(\text{“area”}\lbag\,\mathbf{ea}\otimes\ll\text{“name”}\mapsto[\#\mathrm{text}[\,]\text{“Sherfield”}]\gg\otimes\,\mathbf{ea'}\rbag[\mathrm{true}])\wedge\\ \Diamond(\text{“area”}\lbag\,\mathbf{ea}\otimes\ll\text{“name”}\mapsto[\#\mathrm{text}[\,]\text{“Business School”}]\gg\otimes\,\mathbf{ea'}\rbag[\mathrm{true}])\end{array}\right)\end{array}\right\}$$

runScript(“../mapService/mapService.html”);

$$\left\{\begin{array}{c}\omega_{\mathrm{D}}(\text{“http://www.example.net/mapService/icMap.xml”})\wedge\\ \omega_{\mathrm{C}}(\text{“http://www.example.net/mapService/mapService.html”})\wedge\\ \gamma(\mathrm{layout})\wedge\gamma(\mathrm{addOverlay})\wedge\gamma(\mathrm{layoutOverlay})\wedge\\ \Diamond\left(\begin{array}{l}\#\mathrm{document}_{\mathrm{mapdoc}}[\mathbf{node}_{\mathrm{id}}[\mathrm{f}]]\wedge\mathrm{map}(\mathrm{id})\wedge\\ \Diamond(\text{“area”}\lbag\,\mathbf{ea}\otimes\ll\text{“name”}\mapsto[\#\mathrm{text}[\,]\text{“Huxley”}]\gg\otimes\,\mathbf{ea'}\rbag[\mathrm{true}])\wedge\\ \Diamond(\text{“area”}\lbag\,\mathbf{ea}\otimes\ll\text{“name”}\mapsto[\#\mathrm{text}[\,]\text{“Sherfield”}]\gg\otimes\,\mathbf{ea'}\rbag[\mathrm{true}])\wedge\\ \Diamond(\text{“area”}\lbag\,\mathbf{ea}\otimes\ll\text{“name”}\mapsto[\#\mathrm{text}[\,]\text{“Business School”}]\gg\otimes\,\mathbf{ea'}\rbag[\mathrm{true}])\end{array}\right)\end{array}\right\}$$

$$\left\{\begin{array}{c}\gamma(\mathrm{layout})\wedge\gamma(\mathrm{addOverlay})\wedge\gamma(\mathrm{layoutOverlay})\\ \#\mathrm{document}_{\mathrm{this}}[\text{“html”}[\text{“head”}[\mathrm{true}]\text{“body”}[\mathrm{true}]]]\otimes\\ \Diamond\left(\begin{array}{l}\#\mathrm{document}_{\mathrm{mapdoc}}[\mathbf{node}_{\mathrm{id}}[\mathrm{f}]]\wedge\mathrm{map}(\mathrm{id})\wedge\\ \Diamond(\text{“area”}\lbag\,\mathbf{ea}\otimes\ll\text{“name”}\mapsto[\#\mathrm{text}[\,]\text{“Huxley”}]\gg\otimes\,\mathbf{ea'}\rbag[\mathrm{true}])\wedge\\ \Diamond(\text{“area”}\lbag\,\mathbf{ea}\otimes\ll\text{“name”}\mapsto[\#\mathrm{text}[\,]\text{“Sherfield”}]\gg\otimes\,\mathbf{ea'}\rbag[\mathrm{true}])\wedge\\ \Diamond(\text{“area”}\lbag\,\mathbf{ea}\otimes\ll\text{“name”}\mapsto[\#\mathrm{text}[\,]\text{“Business School”}]\gg\otimes\,\mathbf{ea'}\rbag[\mathrm{true}])\end{array}\right)\end{array}\right\}$$

map = getDocumentElement(mapdoc);

$$\left\{\begin{array}{c}\gamma(\mathrm{layout})\wedge\gamma(\mathrm{addOverlay})\wedge\gamma(\mathrm{layoutOverlay})\\ \#\mathrm{document}_{\mathrm{this}}[\text{“html”}[\text{“head”}[\mathrm{true}]\text{“body”}[\mathrm{true}]]]\otimes\\ \Diamond\left(\begin{array}{l}\#\mathrm{document}[\mathbf{node}_{\mathrm{map}}[\mathrm{f}]]\wedge\mathrm{map}(\mathrm{map})\wedge\\ \Diamond(\text{“area”}\lbag\,\mathbf{ea}\otimes\ll\text{“name”}\mapsto[\#\mathrm{text}[\,]\text{“Huxley”}]\gg\otimes\,\mathbf{ea'}\rbag[\mathrm{true}])\wedge\\ \Diamond(\text{“area”}\lbag\,\mathbf{ea}\otimes\ll\text{“name”}\mapsto[\#\mathrm{text}[\,]\text{“Sherfield”}]\gg\otimes\,\mathbf{ea'}\rbag[\mathrm{true}])\wedge\\ \Diamond(\text{“area”}\lbag\,\mathbf{ea}\otimes\ll\text{“name”}\mapsto[\#\mathrm{text}[\,]\text{“Business School”}]\gg\otimes\,\mathbf{ea'}\rbag[\mathrm{true}])\end{array}\right)\end{array}\right\}$$

addOverlay(map, “Huxley”, “wifi”, “wifi.png”);
addOverlay(map, “Sherfield”, “Cheap Food”, “hotdog.jpg”);
addOverlay(map, “Business School”, “Suits.”, “danger.jpg”);

$$\left\{\begin{array}{c}\gamma(\mathrm{layout})\wedge\gamma(\mathrm{addOverlay})\wedge\gamma(\mathrm{layoutOverlay})\\ \#\mathrm{document}_{\mathrm{this}}[\text{“html”}[\text{“head”}[\mathrm{true}]\text{“body”}[\mathrm{true}]]]\otimes\\ \Diamond(\#\mathrm{document}[\mathbf{node}_{\mathrm{map}}[\mathrm{f}]]\wedge\mathrm{map}(\mathrm{map}))\end{array}\right\}$$

kids = getChildNodes(this);
html = item(kids, 0);
kids = getChildNodes(html);
body = item(kids, 1);

$$\left\{\begin{array}{c}\gamma(\mathrm{layout})\wedge\gamma(\mathrm{addOverlay})\wedge\gamma(\mathrm{layoutOverlay})\\ \#\mathrm{document}_{\mathrm{this}}[\text{“html”}_{\mathrm{html}}[\text{“head”}[\mathrm{true}]\text{“body”}_{\mathrm{body}}[\mathrm{true}]]_{\mathrm{kids}}]\otimes\\ \Diamond(\#\mathrm{document}[\mathbf{node}_{\mathrm{map}}[\mathrm{f}]]\wedge\mathrm{map}(\mathrm{map}))\end{array}\right\}$$

layout(map, body);

$$\left\{\ \#\mathrm{document}_{\mathrm{this}}[\text{“html”}_{\mathrm{html}}[\text{“head”}[\mathrm{true}]\text{“body”}_{\mathrm{body}}[\mathrm{true}]]_{\mathrm{kids}}]\ \right\}$$

Fig. 15. Proof Of The Client Mashup

function. Any calls to service provided functions that may depend on the reintroduced name will still be sound, as they were proven with respect to the same specification.

7 Conclusions & Future Work

Using Context Logic, we have developed and demonstrated resource reasoning for our Reasonable Web language. This means we can use our methods to prove strong safety properties about non-trivial examples, as well as reveal the specific code and data being exposed by services. We have made deliberate choices to model some unfortunate language features present in practical web engineering, and have shown how our reasoning can reduce problems they cause.

In the future, Smith [Smi] plans to provide a complete formalisation of DOM Core Level 1, which we may adopt for the DOM aspects of our work. We also plan to investigate extensions to DOM, either formal (in the later edition standards), or pragmatic additions added by browsers.

Our implementation allows us to investigate further scenarios that can benefit from our techniques. We plan to maintain the implementation as we evolve the, to continuously test our reasoning concepts against engineering issues. We intend to make enhancements to our language, merging in further features from real web languages. Maffeis *et al.* [MMT08] have recently created an operational semantics for JavaScript, detailing the nuances of the language; we hope to use this work to guide our extensions.

We have shown that our work has practical benefits, and the realisation of these will be aided by automated tools. Work on symbolic execution in Smallfoot [BCO05], and recent advances in JStar [DP08] have shown that such tools can be developed for a decidable fragment of Separation Logic. We hope to develop a similarly tractable subset of Context Logic on which software can be based. An ambitious long term goal is the development of a system that can take a service, derive a specification that the author can refine, and then validate that the refinement is correct. The same tool can then be used to show client usage of the service is correct. In the short term, we have observed from writing examples that the code of typical services will have greater complexity than the code of clients consuming them. Whereas the creation and verification of service specifications may be complex, many client usages are simple calls into provided behaviour, and querying of data. Creating a framework for client authoring using our language, along with automation of these client proofs may prove a more attainable short term goal.

Acknowledgements The authors acknowledge financial support from EPSRC, Microsoft and the Royal Academy of Engineering.

References

[BCO05] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In Frank S.

de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *FMCO*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137. Springer, 2005.

[BL90] Berners-Lee. World wide web: Proposal for a hypertext project. 1990.

[CGZ05] Cristiano Calcagno, Philippa Gardner, and Uri Zarfaty. Context logic and tree update. *Principles of Programming Languages 2005 (32nd POPL'2005), ACM SIGPLAN Notices*, 40(1), January 2005.

[dom] Document Object Model Level 1 specification. http://www.w3.org/TR/REC-DOM-Level-1/.

[DP08] Dino Distefano and Matthew J. Parkinson. jStar: Towards practical verification for Java. In Gail E. Harris, editor, *OOPSLA*, pages 213–226. ACM, 2008.

[ECM97] *ECMAScript Language Specification*. European Computer Machinery Association, June 1997.

[GSWZ08a] Philippa Gardner, Gareth Smith, Mark J. Wheelhouse, and Uri Zarfaty. DOM: Towards a formal specification. In *PLAN-X*, 2008.

[GSWZ08b] Philippa A. Gardner, Gareth D. Smith, Mark J. Wheelhouse, and Uri D. Zarfaty. Local Hoare reasoning about DOM. In *Proceedings of the Twenty-Seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems: PODS'08, Vancouver, BC, Canada, June 9–11, 2008*, pages 261–270, pub-ACM:adr, 2008. ACM Press.

[Hoa83] Hoare. An axiomatic basis for computer programming. *CACM: Communications of the ACM*, 26, 1983.

[IO01] Ishtiaq and O'Hearn. BI as an assertion language for mutable data structures. In *POPL: 28th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2001.

[ISB$^+$07] Masayasu Ishikawa, Peter Stark, Mark Baker, Toshihiko Yamakami, Ted Wugofski, and Shin'ichi Matsui. XHTML$^{\text{TM}}$ basic 1.1. Candidate recommendation, W3C, July 2007. http://www.w3.org/TR/2007/CR-xhtml-basic-20070713.

[MMT08] Sergio Maffeis, John C. Mitchell, and Ankur Taly. An operational semantics for javascript. In G. Ramalingam, editor, *APLAS*, volume 5356 of *Lecture Notes in Computer Science*, pages 307–325. Springer, 2008.

[Ohe99] David Von Oheimb. Hoare logic for mutual recursion and local variables. In *Foundations of Software Technology and Theoretical Computer Science, volume 1738 of LNCS*, pages 168–180. Springer, 1999.

[Par05] Matthew J. Parkinson. Local reasoning for Java. Technical Report 654, University of Cambridge Computer Laboratory, November 2005. The author's Ph.D. dissertation.

[PG] A. Wright P. Gardner, G. Smith. Resource reasoning for mashups, technical report and demonstration. Technical report. http://www.doc.ic.ac.uk/∼adw07/rwl/.

[Rey02a] John C. Reynolds. Separation logic: A logic for shared mutable data structures. pages 55–74. IEEE Computer Society, 2002.

[Rey02b] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th IEEE Symposium on Logic in Computer Science (LICS-02))*, pages 55–74, Los Alamitos, July 22–25 2002. IEEE Computer Society.

[Smi] Gareth D Smith. PhD thesis. In preparation.

[W3C99] World Wide Web Consortium. *HTML 4.01 Specification*, December 1999. status: W3C Recommandation, http://www.w3.org/TR/html4/.