Draft PhD Transfer Report
Verifying Web Programs via Local Reasoning

Adam D. Wright

Supervisor: Philippa Gardner
Second Supervisor: Cristiano Calcagno

15th June 2009

1 Introduction

My background is commercial software engineering, a field I left through impatience with the state of robust programming techniques. I hoped to make some small steps in reducing my frustration via research in formal methods, replacing the unreliable, ad-hoc processes used with more rigorous techniques. My research proposal on Resource Reasoning stated that fruit might be found in

> *"...the completion of the modelling for "real" web application infrastructures (JavaScript and related paraphernalia), the authoring of verification tools, and the ramifications..."*

With eight months passed, this report will summarise my progress towards this goal, the changes in direction needed, and the impact both of and on wider research in Resource Reasoning.

1.1 Overview

There is a huge number of people writing software, with nearly 400,000 programmers working in the United States alone. Vast quantities of program code is written, and it all contain errors regardless of programmer skill. Inconsistent specifications, human error and the increasing complexity of software combine to mean that producing correctly functioning programs is extremely hard. The current state of fault finding revolves around *testing*; taking software components, and exercising them manually or automatically to ensure they behave as desired.

Testing is time consuming and complex. We must attempt to cover every possible situation our code might encounter. For every change we make, we need to retest the entire program, lest a small alteration interact with other code in an unfortunate way. We try to structure our programs to make these tasks easy, but the inevitability of human error means testing does not reliably tell us that software functions - it merely shows us that our tests pass. If the test process is flawed, incomplete or enacted incorrectly then the mechanism fails, and as program complexity increases and software faults can literally cost lives, this becomes a troubling state of affairs.

Testing tells us negative facts, a set of statements that "Fault X does not occur in program P", leaving us to hope we have covered the gamut of possible Xs. Much better would be a positive result: "Program P has no faults", where we can eliminate a whole class of errors in one pass. This approach is software *verification*. Whilst an active topic of research for many years, it has typically been ignored by the industry. This can be blamed on verifications lack of support for

common programming systems, the often poor precision of the techniques available, and the lack of automated tools. Verification theory is of little use to an engineer if they must use a restrictive programming language, if twenty false errors are reported for every real bug uncovered, and if the process takes longer than testing!

Even with these barriers, the need for fault free software is forcing the adoption of verification techniques. Critical systems software, such as flight control computers for air travel, are one situation where engineers can accept the limitations of verification in exchange for the guarantees it provides. The ASTRÉE project [21] is a verification tool for the C language, and was used to prove that Airbus flight software is free of runtime errors. Despite attempting to verify C programs, the tool was very much limited to the techniques used in this specific situation; C programs calling `malloc` were not permitted. Less critical, but still important systems have also been analysed. The SLAM engine from Microsoft attempts to verify device driver software written in C. Failures in these smaller programs can cause whole systems to collapse and so must be robust, but even here the tools must assume that heap usage is safe.

Evidently, the dynamic heap is a problem for verification. Many of the difficulties stem from the levels of indirection it introduces, and the run-time nature of the dynamism. The indirection given by references and pointers make it hard know exactly what areas of the heap are being altered - what appears to be a local change may in fact be altering a shared resource. Also, the correctness of a heap manipulating program is often hard to extract from the raw source code, as it is held by the programmer in a set of intuitions. They know they cannot dereference pointer `foo` if the `bar` flag is set and that their linked list allocations are always disjoint from their hashtable memory, but this information is not conveyed well in the source code.

Around ten years ago, the creation of *Separation Logic* gave us a breakthrough in reasoning about this problem and birthed the field of *Local Reasoning*. Local reasoning is built on the "resource separation" concept that captures the implicit disjoint nature of correct heap usage and allows us prove heap programs in a similar way to programmers informal intuitions. This new theory has allowed realistic patterns of dynamic resource usage to be reasoned with precision, and has the theory gave rise to tools such as SpaceInvader that are able to prove large C programs that use dynamic memory.

Dynamic resource is not restricted to the simple heap memory models of C. Indeed, the first thing most programmers do is layer on *high level abstractions* of their dynamic resource by building data structures that encode trees, graphs, lists and other program specific structures into the heap of their chosen language. Context Logic has provided a formal basis for the local reasoning of these higher level structures, and was the basis for Featherweight DOM [16], a formal specification of the highly popular W3C Document Object Model [2]. It gives both a proof of concept for reasoning about libraries at the level of their abstraction (rather than at the language implementation level), and a useful instance of the context reasoning system.

Whilst the DOM example shows we can specify such a library, we have no results about using the specification in a larger context, or in automating the verification techniques. This research attempts to show that we can use this formalism to verify programs authored with a combination of a non-trivial programming language and the DOM library, and that this process can be significantly eased with good tools. As "SpaceInvader" shows one can fruitfully apply Separation Logic to automatically verify C programs, we hope to show Context Logic can be used to automatically verify programs using DOM.

1.2 The state of the research

Our work so far has split into two main projects. Whilst Featherweight DOM is an excellent example of specification using Context Logic, we wanted a more expressive setting for exploring

and extending the program verification abilities. As the simple initial language used in Separation Logic was extended out to languages such as C and Java, we have attempted to extend the work of Featherweight DOM the wider setting of web programming and *mashups*. This has a resulted in a paper, currently in preparation for POPL 2010.

Secondly, we have sought to automate the program verification process provided by our reasoning. Focusing on the new *Segment logic*, we are attempting to find a tractable fragment that is useful for verification, but has a proof theory we can automate. This work is ongoing.

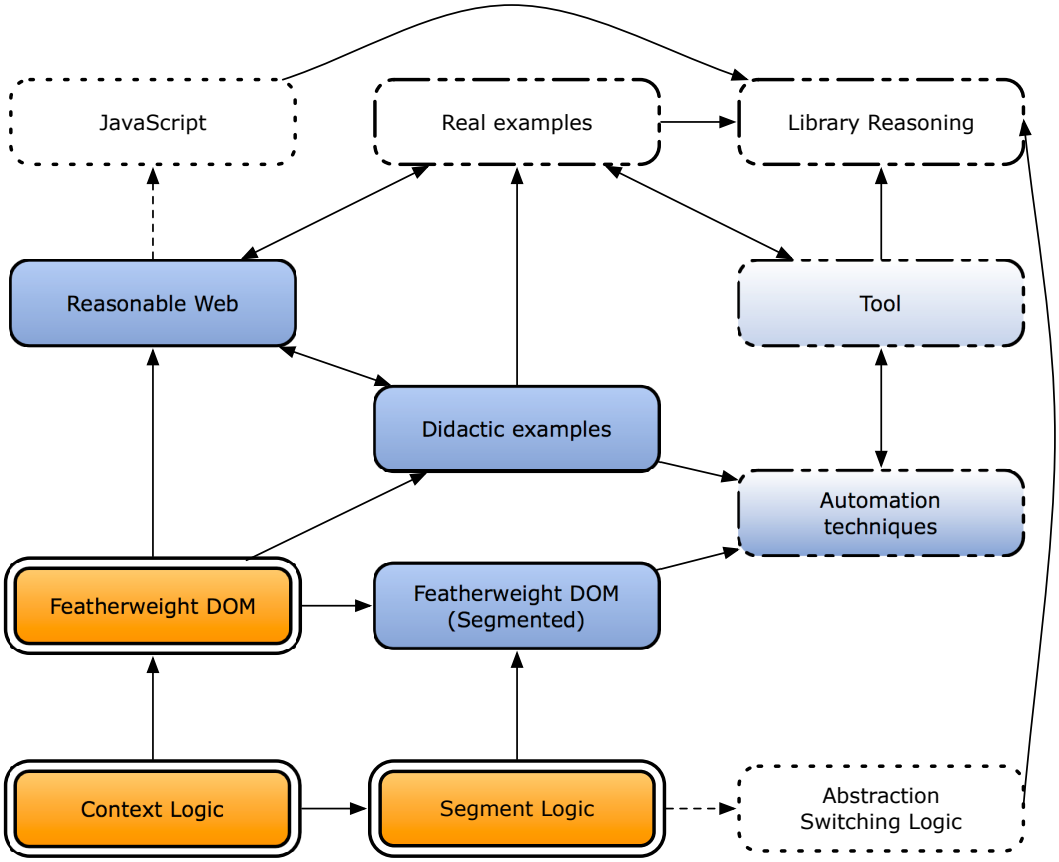

Figure 1: Graph of research. Double lined boxes are other people's work, complete single lines finished work from this project. Partial dotted lines are future work, and totally dotted represent possible avenues for completing the thesis

The graph in Figure 1 shows the context of the efforts so far, along with our plans for the future. The remainder of this report details the completed and underway areas of the research, and shows how we plan to conclude it within the PhD timeframe.

1. **Background**
   Provides an overview of program verification, focusing on the effects Local Reasoning has had on the field. From the motivations that drive it to the current state of the art in the formal underpinnings and practical applications. This also gives an expository style of literature review.

2. **The Reasonable Web**
   Describes a formalism for web programming, focusing in the mashup application. This work continues that of Featherweight DOM and provides a semantic domain in which questions of automation and expressivity can be asked whilst analysing a realistic yet tractable space of problems.

3. **Automation**
   Covers the initial work we have pursued in automating the use of Context Logic for DOM reasoning.

4. **Future work and Thesis plan**
   Gives an overview of the ramifications of the work so far, its context within the active Local Reasoning community, and how we shall extend the work to produce a PhD thesis.

2 Background, Literature & Motivation

The ideal program verification, "Arbitrary program P contains no bugs" is fundamentally undecidable for some definitions of "bug". There will always be a class of behaviours for Turing complete languages that are undesirable, but that we cannot find until they have occurred at runtime. Faced with this, the art of program verification becomes finding the most expressive languages which we can verify against the most useful specifications, whilst minimizing any false reports for bugs that don't actually exist.

The typical technique involves creating *abstract domains* for the underlying language semantics; mathematical approximations of the program behaviour that cover all possible executions, and give identifiable boundaries around areas of concern. For example, the abstract domain of the SLAM was *boolean programs* where the original C program variables are modelled by predicates representing relations between the modeled variables. The program is abstracted in terms of these relations, and automatic search is performed to find paths through the relations to known error states. If the system finds such a path, and it corresponds to a plausible execution of the C program, a real bug has been found; if it does not, a false bug has been found. The SLAM system attempts to refine the program models to eliminate false positives, but some remain inevitable.

2.1 A formal basis

The core idea for program verification with local reasoning is *Hoare Logic*, and is not a new concept. Introduced in [17], it attempts to capture the effects of a program via the use of assertions that specify code behaviour. This information is given in Hoare Triples, and notated

$$\{P\}\mathbb{C}\{Q\}$$

The code $\mathbb{C}$ is in an arbitrary programming language, and P and Q in a logically based *assertion* language. All triples in this paper will be given a *fault avoiding* interpretation; if P is true before $\mathbb{C}$ is executed, then $\mathbb{C}$ cannot enter a fault state. If it runs to a terminal state, Q will be true.

Hoare's original work used first order logic over imperative programs, where the logical variables were "punned" with program variables and so allowed FOL to express properties that the program state would attain at runtime. A proof of a Hoare triple is found by inference rules that reflect the semantics of commands, as well as some for exploiting the reasoning power of the underlying logic. For example

SEQUENCE
$$\frac{\{P\}\ \mathbb{C}_1\ \{S\} \qquad \{S\}\ \mathbb{C}_2\ \{Q\}}{\{P\}\ \mathbb{C}_1; \mathbb{C}_2\ \{Q\}}$$

CONDITIONAL
$$\frac{\{P \wedge B\}\ \mathbb{C}_1\ \{Q\} \qquad \{P \wedge \neg B\}\ \mathbb{C}_2\ \{Q\}}{\{P\}\ \texttt{if}\ B\ \texttt{then}\ \mathbb{C}_1\ \texttt{else}\ \mathbb{C}_2\ \{Q\}}$$

ASSIGN
$$\{P[E/\texttt{x}]\}\ \texttt{x := E}\ \{P\}$$

CONSEQUENCE
$$\frac{P' \implies P \qquad \{P\}\ \mathbb{C}\ \{Q\} \qquad Q \implies Q'}{\{P'\}\ \mathbb{C}\ \{Q'\}}$$

We can use a combination of the sequencing and assignment rules to prove

$$\{x = 1 \wedge y = 1\}\texttt{x = y + 1; y = y - 1;}\{x = 2 \wedge y = 0\}$$

We see the logical variable "punned" with program variables. The assertions form an abstract view of the program's initial and final state and so capture the effect of the code.

Evidently a powerful idea, it unfortunately does not adapt well to programming languages with heap semantics. Specifically, the technique breaks down when faced with *aliasing*. Imagine that the programming language is equipped a set of addressable cells as in the heap of C. Assume a notation `[x]` for the heap cell at address `x`, and consider

$$\{[x] = 1 \wedge [y] = 1\}\ \texttt{[x] = [y] + 1; [y] = [y] - 1;}\ \{[x] = 2 \wedge [y] = 0\}$$

If the values of `x` and `y` are distinct, the specification of the update is correct, as in figure 2.

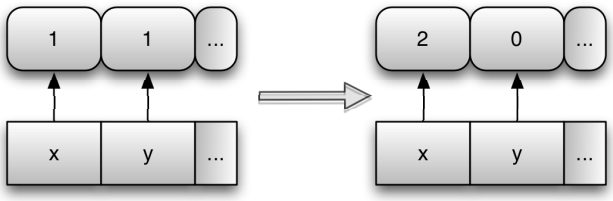

Figure 2: Heap update without aliasing

However, it may be the case that `x` = `y`, as in figure 3. The postcondition is then wrong - the simple rule for assignment has become flawed when faced with the heap.

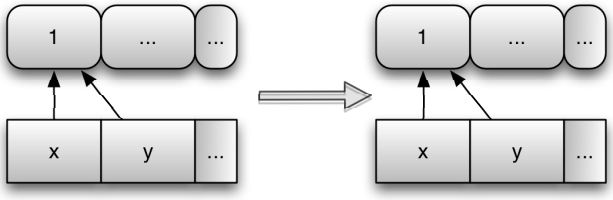

Figure 3: Heap update with aliasing

We can try and correct this via several methods. Encoding the possible options via disjunction is one choice. Another well explored route is reachability predicates, which include assertions

about which heap cells are "reachable" from others (i.e. via a chain of pointers). A very simple case of this concept would be to mandate that heap cell address inequality is explicit; effectively, our abstract interpretation would allow heap update only if the assertion `x` $\neq$ `y` was implied by the precondition. Such a restriction would allow us to know the post-condition is always true, but if we failed to show the property, we would reject the program. A huge class of potential executions have been eliminated to cater for the rare error case, a clear example of imprecision and the approximate nature of abstractions.

Perhaps more than this formal deficiency, these approaches do not match the intuition of the programmer. The assertions are about an entire heap, and do not concisely capture the disjointness that is implicit in code and read unconsciously by the its authors. For example, if we create a new heap cell with `x = new E` then having to change more assertions than those mentioning `x` is surprising - a small change to our mental model has resulted in a large change to the assertions about the model. Our mental "footprint" of the command - the sections of heap we are treading on - has not been reflected in the reasoning.

2.1.1 Separation Logic

Separation Logic takes a different approach to the heap. Based on Pym and O'Hearn's work work on the Logic of Bunched Implications ([22]), and first introduced in Reynold's work [27], Separation Logic extends First Order logic with *structural* connectives that exactly describe the current heap state.

Models of Separation Logic include a specific structure for the heap. A typical setting is a partial function from heap addresses to values, e.g. $h : Addr \rightharpoonup (Addr \cup \mathbb{Z})$. Separation of the heap model into *heaplets*, $h_1 \perp h_2$, is true if the heaplets h_1 and h_2 have disjoint domains. The logic exploits this notion in assertions by building complex heaps from disjoint heaplets with structural connectives. Standard connectives are

$$\textbf{emp} \qquad x \mapsto a \qquad P * Q \qquad P \mathrel{-\!\!*} Q$$

emp describes a heap that is empty, whilst $x \mapsto a$ describes a heap mapping just one address x to a. More complex heaps are built up either via normal conjunction or via the *separating conjunction* $*$, which says the heap can be separated into the disjoint heaplets, $h_1 \perp h_2$, where one satisfies P and the other Q. We have side-stepped the aliasing problem entirely, as any two $*$-connected formulae cannot contain mutual aliasing. Our example above becomes

$$\{x \mapsto 1 * y \mapsto 1\}\ \texttt{[x] = [y] + 1; [y] = [y] - 1;}\ \{x \mapsto 2 * y \mapsto 0\}$$

and is now sound, as the precondition asserts the separation of the heap via $*$: the case $x = y$ is forbidden.

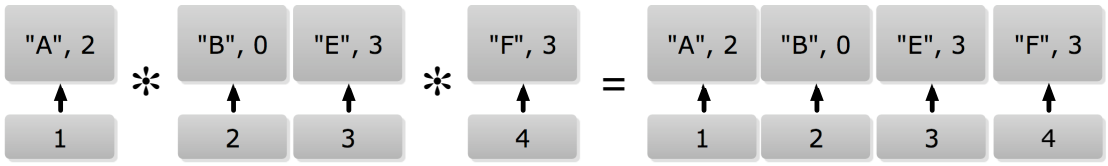

Figure 4: Visualisation of a Separation Logic heap. Arrows from variables to heap cells cannot cross $*$ boundaries; aliasing is forbidden between them

Primitive heap commands, such as update and memory allocation, are added to the proof theory via *small axioms*. They give the behavior of commands on the smallest heap that ensures their safe operation.

$$\begin{array}{ccc} \{\ \mathbf{emp}\ \} & \{\ \exists v.x \mapsto v\ \} & \{\ \exists v.x \mapsto v\ \} \\ \texttt{x = new E} & \texttt{[x] := E} & \texttt{delete x} \\ \{\ x \mapsto E\ \} & \{\ x \mapsto E\ \} & \{\ \mathbf{emp}\ \} \end{array}$$

That each axiom discusses only the heap cells it requires formalises our footprint intuition. We have captured our desire that small heap changes require only small changes to the assertions. This is the ethos of local reasoning, and O'Hearn and Yang summarise it thus

> *"To understand how a program works, it should be possible for reasoning and specification to be confined to the cells that the program actually accesses. The value of any other cell will automatically remain unchanged"*

Supporting this concept when proving programs is the vital new *frame rule*, allowing us to set aside parts of the heap during our verification. If the code under examination does not access variables mentioned in a $*$ connected heaplet, the separation property allows us to simply "set it aside" when proving something regarding that code. It is this rule that allows us to use the small axioms on non-trivial programs, and this rule that recaptures the programmers intuition about their heap usage. Their "obvious" thoughts about what heap is untouched is made concrete in the frame chosen in the proof.

$$\textsc{Frame}\ \frac{\{P\}\ \mathbb{C}\ \{Q\}}{\{P * R\}\ \mathbb{C}\ \{Q * R\}}\quad \text{If } mods(\mathbb{C}) \cap free(R) = \emptyset$$

The last connective, $P \mathrel{*\!\!-} Q$ is the right adjoint of separation. This is satisfied by any heap that, when extended by a heap satisfying P, will satisfy Q. Intuitively, it is a "hypothetical" operator; "If I were to add P, I'd get Q". One prominent use is backwards reasoning; starting with a post-condition and some code, we can step "backwards" to find the weakest precondition that ensures the postcondition.

2.1.2 More complex heaps

The cells within the heap model can be more complex than a single value. Figure 4 shows a heap addressed by integers, where each cell contains two data; we see how the domain disjointness allows us to $*$ together separate heaplets, which together form a larger complete heap.

Moreover, we can extend the class of models to anything which supports a "separability" concept. Separation Algebras [8] abstract the concept of resource to a cancellative, partial commutative monoid; this structure induces a separation between monoid elements via the domain of the operator. It becomes fair to say that we are not reasoning about "heaps" or "trees", but about any form of separable resource. The paper gives examples like variables as resource (where standard variables are considered as a separable primitive, rather than punned with logical variables), or heaps with permissions (where each heaplet can be associated with an "access" permission). Less abstractly, we can consider a model of database handles; a finite resource, managed by the application code where aliasing can occur.

Interestingly, Separation Logic is no stronger than first order logic. The work of [6] shows how any propositional Separation Logic formulae has an equivalent FOL expression. Why the Separation Logic has garnered so much success is, we believe, tied to the conceptual match it has with how people think about resource.

2.1.3 Abstraction

Our heap model so far very much matches the C heap; addresses indexing stored values. By modelling heap cells so they store two elements, we can encode singly linked lists in the same way a C program would, and discuss them logically using *inductive predicates*. In [26] the list predicate, list $\alpha(i, j)$ describes a list α spanning addresses i to j. It is defined as

$$\text{list } \epsilon(i,j) \triangleq i = \mathbf{emp} \wedge i = i \qquad \text{list } (a \cdot \alpha)(i,k) \triangleq \exists j. i \mapsto a, j * \text{list } \alpha(j,k)$$

This exactly matches the programmers intuition; the empty list exists between i and j if and only if the length of the address space is zero, and the heap is empty. A list of a followed by more list α is found from i to k iff we find a in the first subcell of address i, and the second contains some j that is the address of the rest of the list, pointing to a separate heap section. This separation is essential - it ensures we are discussing a list, rather than a cycle.

Similar constructions are possible for trees, graphs or more esoteric structures. Regardless of how we encode them into the heap, we are attempting to layer a more abstract *high level* structure onto the less abstract *low level* heap[1]. Anything we wish to prove regarding high level properties (i.e. a list append function always takes and returns a list) will inevitably be a proof using our low level choices. We cannot simply change the predicate definition and have things "just work", as the original proof will have unrolled the inductive predicates, and made assumptions about the structure of the heap cells it is using. Moreover, the notion of footprint for commands does not neatly survive the abstraction process. Implementation information hidden under the abstraction is needed to carry out the command, but may be extraneous from a high level perspective.

2.1.4 Context Logics

Context Logics were introduced by Calcagno, Gardner and Zarfaty, and encapsulate the idea that assertions of resource separation should retain a memory of the original shape of the data. We can disassemble a complex form into sub-elements, where at least one of the sub-elements retains information on the original location of the others. Where Separation logic can separate only homogeneous data structures (i.e. those that break down into structurally identical parts like heaplets or variables) we can now pull apart heterogeneous forms (suchs a trees or graphs, which break into structurally distinct elements), safe in the knowledge we can reassemble them. This allows us to make models that encode high level structures directly.

Three major variants on the Context theme exist: Single holed, multi holed and "Segment"; we shall consider single holed and segment logics in this presentation. In both, we use a high level model of simple tagged n-ary trees: A tree is either some data d and label x along with a child tree, the empty tree, or a pair of trees. This grammar admits arbitrary trees. For example, we can construct trees like that in figure 5. We make a restriction that no tree shall contain two identical labels - they are analogous to our heap addresses, in a "tree shaped heap".

$$T_{CL} ::= d_x[T_{CL}] \mid \varnothing \mid T_{CL} \otimes T_{CL}$$

Taking the "splitting with a record of the original" concept, we could break the tree of 5 apart into $\texttt{Foo}_1[\texttt{Bar}_2[\varnothing]\otimes !]$ and $\texttt{Baz}_3[\texttt{Frob}_2[\varnothing]]$, using ! to record the position of the removed data. We can glue it back together by simply substituting the later for the ! in the former. This is the essence of the Single Holed Context Logic (SHCL): It allows us to assert properties of higher level structures that can be pulled apart into two formulae; one for the data, and one for the surrounding that asserts the existence of hole. As the heap splitting ability is expressed with $*$ in

[1] "High" and "Low" being relative terms

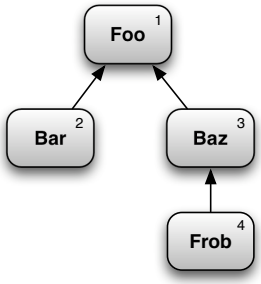

Figure 5: The tree $\texttt{Foo}_1[\texttt{Bar}_2[\varnothing] \otimes \texttt{Baz}_3[\texttt{Frob}_4[\varnothing]]]$

Separation Logic, we use the *context application* connective, $\mathcal{C}_{CL} \circ \mathcal{T}_{CL}$, describing data that can be split into a context satisfying a formulae $\mathcal{C}_{CL}$ and data satisfying $\mathcal{T}_{CL}$.

To formalise the intuition, we give structure for our tree that records the "hole", $-$.

$$C_{CL} ::= - \mid d_x[C_{CL}] \mid C_{CL} \otimes T_{CL} \mid T_{CL} \otimes C_{CL}$$

The formulae of the SHCL break down into two distinct classes (as well as the standard classical formulae inherited from FOL). The "specific formulae" describe the high level abstraction we are reasoning (in this case, trees). They are analogous to $\mapsto$ and **emp** in our Separation example, but are more distinct due to the additional structure that high level abstractions generally have. These are often typed into two subclasses, depending on the presence of context holes. In our case, $\mathcal{T}_{CL}$ asserts complete trees whereas $\mathcal{C}_{CL}$ describes incomplete trees (i.e. containing a hole).

$$\begin{array}{rcl} \mathcal{T}_{CL} & ::= & d_x[\mathcal{T}_{CL}] \mid \varnothing \mid \mathcal{T}_{CL} \otimes \mathcal{T}_{CL} \\ \mathcal{C}_{CL} & ::= & - \mid d_x[\mathcal{C}_{CL}] \mid \mathcal{C}_{CL} \otimes \mathcal{T}_{CL} \mid \mathcal{T}_{CL} \otimes \mathcal{C}_{CL} \end{array}$$

We extend the model with context holes for the purpose of the reasoning, and can then give satisfaction for this logic in a very direct way, as the structure of formulae map directly to the structure of the heap they are asserting. We will see a formal example in section 3.

The second class are the structural formulae, $\mathcal{S}_{CL}$, analogous to $*$ and its adjoint in Separation logic. They describe trees through the contextual breaks that are possible.

$$\mathcal{S}_{CL} ::= \mathcal{C}_{CL} \circ \mathcal{T}_{CL} \mid \mathcal{C}_{CL} \circ\!\!- \mathcal{T}_{CL} \mid \mathcal{T}_{CL} -\!\!\circ \mathcal{T}'_{CL}$$

The application connective, $\mathcal{C}_{CL} \circ \mathcal{T}_{CL}$ states that the current heap can be split into a disjoint context satisfying $\mathcal{C}_{CL}$, and a tree structure satisfying $\mathcal{T}_{CL}$. The asymmetry of $\circ$ gives rise to a pair of adjoints, $\circ\!\!-$ and $-\!\!\circ$. These serve the same purpose as "hypothetical" connectives, one for contexts and the other for data. The first is satisfied only if for every context satisfying $\mathcal{C}_F$, placing the current heap into the hole would produce a heap satisfying $\mathcal{T}_{CL}$. The second is satisfied by a context, such that if we placed data satisfying $\mathcal{T}_{CL}$ into it, the end result would satisfying $\mathcal{T}'_{CL}$. Whilst these are powerful tools for backwards reasoning, we will also see them appear in the small axioms of our tree reasoning system.

Figure 6 illustrates context application, where a large tree structure can be split into a context (retaining the hole of the removed data), and the data that has been removed. The "frame" idea

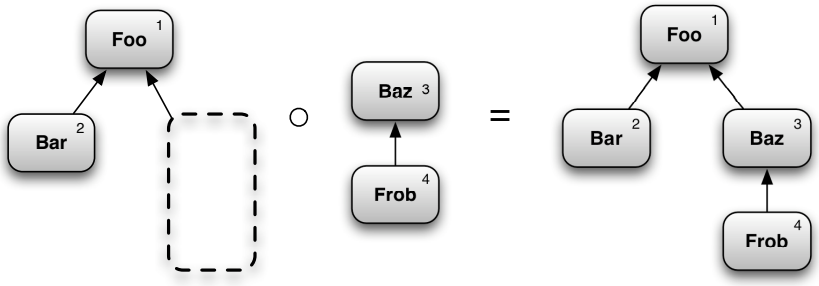

Figure 6: Visualisation of a Single Holed Context Logic heap

is perhaps even more intuitive here - we can drop the context "frame" over the data to form the original structure. The context frame rule shows allows us to use this in our reasoning.

$$\textsc{Ctx-Frame}\quad \frac{\{P\}\ \mathbb{C}\ \{Q\}}{\{R \circ P\}\ \mathbb{C}\ \{R \circ Q\}}\quad \text{If } mods(\mathbb{C}) \cap free(R) = \emptyset$$

Program reasoning with these high level concepts needs a programming language with such a tree shaped heap, and one with a much richer set of commands than the simple heap cell update/creation and destruction of C style heaps. Considering the label of a node as the "address", we may have one command that fetches the data associated with a tree node, and another which moves a node from its current parent to a new parent. We could axiomatize them as

$$\begin{array}{c}\{\ d_{\texttt{node}}[F]\ \}\\ \texttt{a := getData(node)}\\ \{\ d_{\texttt{node}}[F] \wedge \texttt{a} = d\ \}\end{array} \qquad \begin{array}{c}\{\ (\varnothing \multimap (C \circ (d_{\texttt{parent}}[F]))) \circ (d'_{\texttt{child}}[F'])\ \}\\ \texttt{move(child, parent)}\\ \{\ C \circ (d_{\texttt{parent}}[F \otimes t'_{\texttt{child}}[F']])\ \}\end{array}$$

The GetData axiom is simple; it asserts the structure of the identified node, including the data we need which is assigned it to a variable. We can use it on arbitrary trees by framing off the surrounding structure. Notice the *logical* variable F. These are artifacts of the reasoning, and we can view them as capturing arbitrary structure, allowing us to specify invariants between the pre and post conditions.

The Move axiom is more complex. In a tree, it is impossible to move a parent to one of its children so for fault freedom, the axiom must assert that this is not the case. We accomplish this with the adjoint, and can read the precondition as "The heap can be split into a context and the child node, such that we can place empty tree within the context hole left by removing the child, and still extract the parent". This captures the correct ancestral relationships, as if the parent were a descendent of the child, we could remove the child from a context, but placing an empty tree in the resultant hole would not leave us a context from which we could extract the parent. Notice that a logical variable C has captured the surrounding tree structure, up to at least the first mutual ancestor of the new parent and child. This is an artifact of the single holed reasoning; not really part of the command footprint, but needed by the logic to express the disjointness properties we required.

The single holed logic has been very effective for specifying tree update. Featherweight DOM, for example, formalises the W3C DOM specification. Our research on specifying a more expressive language for web programming has built on this work, and SHCL forms the logical basis for chapter

3. However, the "excess footprint" in the `appendChild` command is both inelegant and potentially harmful. Consider a network program, where we transfer elements of the tree structure over the network. The small axiom for `move` claims the covering context is needed, which is not the minimal footprint - it would cause much more data transfer than necessary. The axiom also uses the adjoint connective, thus introducing a universal quantification in satisfaction. This can prove awkward when it comes to automation of the reasoning techniques and is at the very least, a cumbersome expression.

2.1.5 Segment Logic

Segment Logic is a different approach to the context problem. The work of Wheelhouse and Gardner, it postdates single holed logic and had the goal of decreasing the footprint size for the move commands (effectively, eliminating the covering context C above). The theory of Segment logic draws inspiration from Context Logic, Separation Logic and Cardelli's Ambient Logic.

Models of the logic still include high level descriptions of the data structure, but are now built around *fragments*. These two concepts are defined below as T_{SL} and F_{SL} respectively.

$$\begin{aligned} T_{SL} &::= d_x[T_{SL}] \mid \varnothing \mid T_{SL} \otimes T_{SL} \mid \mathbf{x} \\ F_{SL} &::= \mathbf{x} \leftarrow T_{SL} \mid F_{SL} + F_{SL} \mid (\nu \mathbf{x})(F_{SL}) \end{aligned}$$

We have the expected tree structures in T_{SL}, but extend them with hole *labels* $\mathbf{x}$. Each tree is then given a root label to become a fragment, $\mathbf{x} \leftarrow T_{SL}$. A single data structure may have many fragments, $F + F$, and we are able to introduce and dismiss fragments using *restriction*, ν. Restriction binds a hole label, stating that the label within a tree formulae is linked to the same label that forms the head of a fragment. We consider data split via ν to be identical to data without the splitting and make this evident with a congruence, an example of which is

$$\mathbf{x} \leftarrow \mathtt{Foo}_1[\mathtt{Bar}_2[\varnothing] \otimes \mathtt{Baz}_3[\mathtt{Frob}_4[\varnothing]]] \equiv (\nu \mathbf{y}, \mathbf{z})(\mathbf{x} \leftarrow \mathtt{Foo}_1[\mathbf{y} \otimes \mathbf{z}] + \mathbf{y} \leftarrow \mathtt{Bar}_2[\varnothing] + \mathbf{z} \leftarrow \mathtt{Baz}_3[\mathtt{Frob}_4[\varnothing]])$$

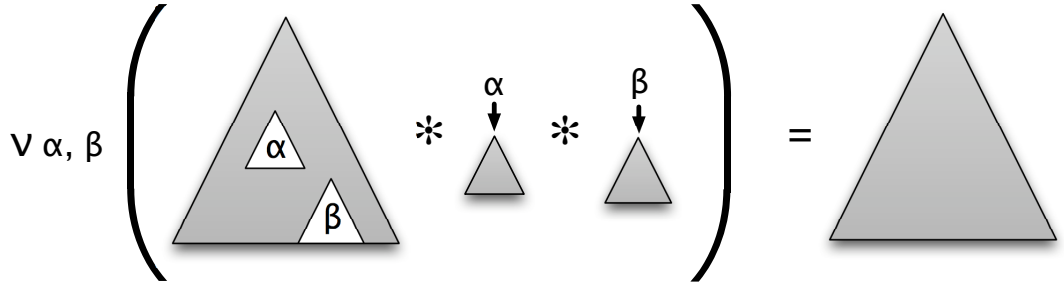

Figure 7: Visualisation of a Segment Logic heap

The logic exploits the "split and collapse" properties of the model to provide the structural reasoning. We have tree formulae $\mathcal{T}_{SF}$, and new structural formulae $\mathcal{S}_{SF}$.

$$\begin{aligned} \mathcal{T}_{SL} &::= d_x[\mathcal{T}_{SL}] \mid \varnothing \mid \mathcal{T}_{SL} \otimes \mathcal{T}_{SL} \mid \alpha \\ \mathcal{S}_{SL} &::= \alpha \circledR \mathcal{S}_{SL} \mid \forall \alpha . \mathcal{S}_{SL} \mid \alpha \leftarrow \mathcal{T}_{SL} \mid \mathcal{S}_{SL} * \mathcal{S}_{SL} \mid \alpha \multimap \circledR \mid \mathcal{S}_{SL} \mathrel{-\!\!*} \mathcal{S}_{SL} \mid \varnothing_{\mathcal{F}_{SL}} \end{aligned}$$

The tree formulae are as before, except we now have a hole assertion, α, indicating the presence of a hole in the structure. The $*$ connective of Separation logic is used, but rather than stating that the heap can be split into heaplets P and Q, it states that the structure exists as two fragments satisfying P and Q. We also have the expected adjoint.

The capacity of the structure to be split into fragments is managed using *Revelation*, ®, (familiar from the Ambient Logic). Revelation $\alpha \circledR F$ states that the data is restricted under a label α, and that if we "peel off' this α, the data under the restriction satisfies F. To allow new, fresh splittings, we give a $\nu\,\alpha$ quantification which asserts that the label α is fresh within the formulae and model. Between freshness and revelation, we can assert that the data structure can be split using a new hole, and that the splitting can be revealed as many fragments. Revelation also has an adjoint, essential for the weakest preconditions to exist.

We have two frame rules now. As expected, we can frame off $*$ segmented assertions. However, these are often concealed under revelation, and so we allow ourselves a revelation frame rule. In essence, this framing operation "forgets" that we introduced a split at all - we temporarily consider just the fragmented data. We also include an elimination rule for the freshness quantifier, analogous to the elimination rule for existential quantification in classical Hoare logic.

Rev-Frame
$$\frac{\{P\}\ \mathbb{C}\ \{Q\}}{\{\alpha \circledR P\}\ \mathbb{C}\ \{\alpha \circledR Q\}}$$

Seg-Frame
$$\frac{\{P\}\ \mathbb{C}\ \{Q\}}{\{R * P\}\ \mathbb{C}\ \{R * Q\}} \quad \text{If } mods(\mathbb{C}) \cap free(R) = \emptyset$$

Fresh-Elim
$$\frac{\{P\}\ \mathbb{C}\ \{Q\}}{\{\nu\,\alpha.P\}\ \mathbb{C}\ \{\nu\,\alpha.Q\}}$$

With these rules, we can write much smaller axioms in the form

$$\begin{array}{c} \{\ \alpha \leftarrow d_{\mathtt{x}}[\beta]\ \} \\ \texttt{a := getData(x)} \\ \{\ \alpha \leftarrow d_{\mathtt{x}}[\beta] \wedge a = d\ \} \end{array} \qquad \begin{array}{c} \{\ \alpha \leftarrow d_{\mathtt{p}}[\beta] * \gamma \leftarrow e_{\mathtt{c}}[X]\ \} \\ \texttt{move(c, p)} \\ \{\ \alpha \leftarrow d_{\mathtt{p}}[\beta \otimes e_{\mathtt{c}}[X]] * \gamma \leftarrow \varnothing\ \} \end{array} \quad \text{If } X \text{ has no context holes}$$

The covering context in `move` is now gone, though it has a (minor) side condition. We shall return to give a detailed exposition in chapter 4.

2.2 Efforts in automation

No engineer, even if trained in the techniques needed to find them, would write program proofs in the above systems. It is a fairly complex, time consuming process, and finding proofs can often be error prone. No matter how successful our theory, it will remain unused without a high degree of *automation*, both in specifying programs and then verifying them. Even researchers can benefit from automation techniques. When we search for examples of limits in the reasoning, particularly those that appear in complex and large programs, it is tedious to manually prove every program we wish to investigate.

The automated reasoning of formal logics is a very active area of research, though techniques for our forms of structural logic are relatively new. The above logics all extend first order principles so unsurprisingly their general forms are undecidable. Work in the local reasoning community has primarily focused on the spacial apparatus that forms the novel crux of the work. Other groups continue to search for methods of optimising proof search for, say, FOL with equalities, research that is largely orthogonal to the resource separation concepts.

The other issue is automation of Hoare reasoning; not deciding properties of purely logical formulae, but asking questions of the form "Is $\{P\}\mathbb{C}\{Q\}$ valid?". An elegant approach is to

exploit the *weakest preconditions* that the our logics admit for the commands and axioms of our languages (ignoring loops, for now). Starting with Q, we can step backwards across commands finding the weakest precondition at each point, until we find an expression describing the weakest environment in which the code will execute. If P implies the expression, we know the triple is true. Whilst a simple and attractive proposition, the weakest preconditions of commands tend to be rather complex, making extensive use of the structural adjoints. The resultant weakest precondition over complex code becomes extremely complex, and we have no results at all for creating an algorithm that will decide the problem in a sensible amount of time.

An alternative finding significant success is *Symbolic execution*. In the operational case, commands mutate the concrete state of the execution; the parallel can be used in the reasoning, whereby commands mutate the abstract symbolic state given by the assertion language. Rather than reason backwards with weakest preconditions, we reason in a forwards fashion, until a transformed pre-condition meets the postcondition of the code. This still requires us to answer an entailment between logical formulae, but the assertions that result from symbolic execution are typically much simpler than backwards reasoning provides.

2.2.1 Existing techniques and tools for Separation logic

There are now several tools for program verification based more or less on the Separation Logic concepts, each at various stages of maturity. Many remain under active development, both in terms of extending the theoretical achievements and practical utility. Most interesting from our current perspective is *Smallfoot*. The first research project into automating Separation Logic techniques, the goal of both the implementation and underlying techniques was

> *to see whether the simplicity of the by-hand proofs in separation logic could be transferred to an automatic setting*

It formed a proof of concept that Separation Logic can indeed be automated, at least in principle. The style of investigation used strongly guides our attempts in chapter 4.

Smallfoot operates over a simple imperative language with procedures, conditionals and loops. Pre and post conditions for procedures and specifications of loop invariants must be provided by the user. The goal of Smallfoot is, given a suitably annotated program $\mathbb{C}$, to find a proof that the program is correct, i.e. to show a proof of $\{P\}\mathbb{C}\{Q\}$.

The key techniques used are *symbolic execution* over *symbolic heaps*. The assertion language of Smallfoot is not full Separation logic. Rather, it attempts to abstract descriptions of concrete program states that actually occur in a small language, restricting the semantic domain to assertions describing simple, realistic heaps. These symbolic heaps allow the "points to" relation of Separation logic, $*$ separated heaps, and a small set of inductive predicates that have been lifted from admissible formulae to be a primitive element of the logic. These symbolic heaps are used because they admit a much simpler proof theory for deciding entailments, $SH_1 \vdash SH_2$.

This entailment theory utilises specific rules of inference on the two heaps: Normalisation rules, and Subtraction rules. We rewrite the left hand side of the entailment using normalisation rules; propagating asserted equalities throughout the formulae, unrolling inductive predicates and so forth. As we can, we subtract elements from both sides; tautological equalities, or $*$ conjuncts that are the same on both sides. Eventually, we subtract enough of the assertions that the truth of the entailment is evident. We shall see a theory of this form in section 4.

The formulae entailment question does not answer the Hoare triple validity problem alone. The $\mathbb{C}$ between the assertions will mutate the state of the heap, and these alterations are handled in "by-hand" proofs using the small axioms and rules of inference built on the language semantics. Smallfoot attempts to mirror this process with a set of symbolic execution rules for each command

that can occur. Intuitively are enacting the effects of commands on the abstract heap descriptions, much as the operational semantics handle concrete heaps. One example is

$$\textsc{Dispose}\quad \frac{\{\Pi \mid \Sigma\}\mathtt{C}\{\Pi' \mid \Sigma'\}}{\{\Pi \mid \Sigma * E \mapsto [\rho]\}\mathtt{dispose(}E\mathtt{);\ C}\{\Pi' \mid \Sigma'\}}$$

Reading this rule bottom-up, we see that to prove the triple surrounding code `dispose(E)` with continuation code `C`, the precondition must have a symbolic heap structure which maps E to some cell. If we have this, proving the triple is equivalent to finding a proof that the continuation code holds in the symbolic heap without E allocated.

With a family of similar rules and an algorithm to apply them, we eventually reduce a problem of the form $\{P\}\mathbb{C}\{Q\}$ to a set of propositions $\{P\}\mathtt{end}\{Q\}$ (a set because rules for handling conditions may introduce branching into the proof search tree). The operational effect of the `end` command is nothing, so the truth of the Hoare triple lies in the truth of $P \vdash Q$. We answer this question with the entailment theory.

There are some subtleties in this process; simple pattern matching for symbolic execution is not generally enough, as the symbolic heap may not be in a form that directly matches the command premise. A set of rearrangement rules are used, leveraging the entailment theory, to provide sound rewrites of heap that match command premises.

The frame problem also appears; specifications for procedures will be small, asserting just the needed structure. The symbolic heap at the call sites of these procedures may be larger, a problem solved in hand proofs by the Frame rule and a manually picked frame. Smallfoot utilises its entailment theory to find the frames of calls. The specific entailment algorithm means that given a larger precondition C and procedure precondition P, asking the question $C \vdash P$ will fail (as C asserts a different heap to P). However, in failing, it may eventually reach a state where it tries to prove $C' \vdash \mathbf{emp}$; i.e. the procedure specification requirement has been discharged, and we have some additional call site assertions. This C' acts as the frame.

SpaceInvader [13] is a more pragmatic tool, seeking to prove properties of real C programs. Most interesting from our perspective is the use of specific *abstraction* rules as part of proof search. These rules might be seen as the counterpart to the unrolling of inductive predicates, as they often "roll-up" specific concrete $\mapsto$ relations back into predicates. The user no longer need specify loop invariants, as they can often be found via a fixed point computation; performing the loop symbolic execution and abstracting until a fixed-point is reached.

Recent work on *abductive reasoning* has been incorporated into SpaceInvader. Rather than start with specifications and prove them, abductive reasoning seeks to determine what is needed as a precondition from a failed proof attempt. This is a kind of inverse frame inference; rather than determining what the excess of an assertion is, we attempt to determine what is missing. The SpaceInvader team support both, in a system they call "bi-abduction", which allows a large amount of C code to be analysed without any programmer provided assertions.

SpaceInvader still has limitations. The abstract domain contains predicate definitions and proof theory for lists, but does not yet handle trees or more complex data structures. JStar [12], a tool building on Parkinson's work modelling Java heaps in Separation Logic, has an interesting solution to this problem by providing users with the ability to augment its proof theory. Rather than a fixed proof algorithm that must be expanded in the program, the JStar inference engine is parameterised on a set of user rules for normalisation, subtraction and abstraction. Users can encode rules that model their own high level predicates.

2.2.2 Higher level structures

Given tools with improving bi-abductive reasoning ability, and user providable proof theories, one might question why we need the higher level abstraction at all. Can we not provide a low level implementation of the high level library, ensure that faulting behaviors in the high level specification are somehow exposed, then let a tool like SpaceInvader loose on the code?

We would answer that we cannot, for both a pragmatic and theoretic reason. In terms of the proof theory, the use of a low level implementation to prove a high level specification breaks one key role of the abstraction, as it is no longer a "black box" that can be replaced with any other code that satisfies the same high level specifications. As in the inductive predicates case, the low level implementation details are likely to be used within any proof, and we cannot (without extensive manual verification) be sure that these details have not been used in some part of the proof derivation.

Perhaps a stronger argument, we have no easy way of formally capturing what it means for the low level code to implement the high level specification - we blithely claim to have an implementation, but without some formal method of capturing what the high level abstraction is claiming (as we did using Context Logic for DOM), our claim that the low level code implements it is unverifiable. This concept has been investigated by Gardner and Zarfaty in [6], where they show how a high level specification in Context Logic can be translated to a specific implementation of the concepts to show the implementation is sound.

A more pragmatic approach is seen in how useful the resultant specifications are in developing software. If we assume the low level implementation captures the high level detail, we would like the reasoning system we use to give us insight into the programs by way of the specifications generated. This is useful both for diagnosing the errors when faults are found and we want to correct the error, as well as for investigating the behaviour of our programs (in essence, using the specifications to gain a better understanding of the program).

We developed a complete implementation of the Featherweight DOM system in C to analyse this concept[2]. The small program of figure 8 uses this library to create a high level structure in the heap containing a DOM text node. Despite Space Invader's lack of tree support, the program given is simple enough that Space Invader can analyse it, giving the set of output propositions in figure 9.

```
struct CDOM_Node* n = CDOM_createTextNode("Foo bar baz");
char* s = CDOM_getNodeName(n);
CDOM_appendData(n, "Blah");
char* t = CDOM_substringData(n, 0, 11);
CDOM_deleteData(n, 3, 4);
t = CDOM_substringData(n, 0, 12);
```

Figure 8: Small DOM program using CDOM

One should not read the propositions too closely, but simply see the density and complexity. The equivalent high level specification, using the Featherweight assertion language, is

$$\{\varnothing_{\mathrm{G}}\}$$
$$\mathbb{C}$$
$$\{\#\mathrm{text}_{\mathtt{n}}[\text{Foo blah baz}] \wedge \mathtt{s} = \text{“\#text”} \wedge \mathtt{t} = \text{“Foo blah baz”}\}$$

[2]The code can be found at `http:www.doc.ic.ac.uk/~adw07/code/CDOM.c`

```
  PROP 1:  &main$n|->0: * &main$t|->0: * &main$tmp___1|->0: * &main$s|->0:
  * &main$tmp___0|->0: * &main$tmp|->0:
  * &GB$CDOM_INDEX_SIZE_ERR|->_siltmp$7: * &GB$CDOM_NODE_TEXT|->0:
  * &GB$CDOM_GENERAL_ERR|->_siltmp$8:
  * &GB$CDOM_NO_DATA_ALLOWED_ERR|->_siltmp$6: * &main$argc|->_siltmp$0:
  * &main$argv|->_siltmp$1: * &GB$CDOM_NOT_FOUND_ERR|->_siltmp$5:
  * &GB$CDOM_NODE_ELEMENT|->_siltmp$2:
  * &GB$CDOM_INVALID_CHARACTER_ERR|->_siltmp$4:
  * &GB$CDOM_HIERARCHY_REQUEST_ERR|->_siltmp$3:


PROP 2:
  &main$n|->_siltmp$14: * &main$t|->0: * &main$tmp___1|->0:
  * &main$s|->_siltmp$13: * &main$tmp___0|->_siltmp$13:
  * &main$tmp|->_siltmp$14: * &GB$CDOM_INDEX_SIZE_ERR|->_siltmp$7:
  * &GB$CDOM_NODE_TEXT|->_siltmp$8: * &GB$CDOM_GENERAL_ERR|->_siltmp$9:
  * &GB$CDOM_NO_DATA_ALLOWED_ERR|->_siltmp$6: * &main$argc|->_siltmp$0:
  * &main$argv|->_siltmp$1: * &GB$CDOM_NOT_FOUND_ERR|->_siltmp$2:
  * &GB$CDOM_NODE_ELEMENT|->_siltmp$3:
  * &GB$CDOM_INVALID_CHARACTER_ERR|->_siltmp$4:
  * &GB$CDOM_HIERARCHY_REQUEST_ERR|->_siltmp$5: * _siltmp$11|->_siltmp$12:
  * _siltmp$13|->_siltmp$10: * _siltmp$14|->{children:0,
  childText:_siltmp$11, parent:0, tagName:_siltmp$13}:

PROP 3:
  &main$n|->_siltmp$15: * &main$t|->_siltmp$16: * &main$tmp___1|->0:
  * &main$s|->_siltmp$14: * &main$tmp___0|->_siltmp$14:
  * &main$tmp|->_siltmp$15: * &GB$CDOM_INDEX_SIZE_ERR|->_siltmp$7:
  * &GB$CDOM_NODE_TEXT|->_siltmp$8: * &GB$CDOM_GENERAL_ERR|->_siltmp$9:
  * &GB$CDOM_NO_DATA_ALLOWED_ERR|->_siltmp$6: * &main$argc|->_siltmp$0:
  * &main$argv|->_siltmp$1: * &GB$CDOM_NOT_FOUND_ERR|->_siltmp$2:
  * &GB$CDOM_NODE_ELEMENT|->_siltmp$3:
  * &GB$CDOM_INVALID_CHARACTER_ERR|->_siltmp$4:
  * &GB$CDOM_HIERARCHY_REQUEST_ERR|->_siltmp$5: * _siltmp$12|->_siltmp$13:
  * _siltmp$14|->_siltmp$10: * _siltmp$15|->{children:0,
  childText:_siltmp$12, parent:0, tagName:_siltmp$14, type:_siltmp$8}:
  * _siltmp$16|->_siltmp$11:
```

```
PROP 4:  &main$n|->_siltmp$16: * &main$t|->0:
  * &main$tmp___1|->_siltmp$7: * &main$s|->_siltmp$15:
  * &main$tmp___0|->_siltmp$15: * &main$tmp|->_siltmp$16:
  * &GB$CDOM_INDEX_SIZE_ERR|->_siltmp$14: * &GB$CDOM_NODE_TEXT|->_siltmp$11:
  * &GB$CDOM_GENERAL_ERR|->_siltmp$9:
  * &GB$CDOM_NO_DATA_ALLOWED_ERR|->_siltmp$8: * &main$argc|->_siltmp$0:
  * &main$argv|->_siltmp$1: * &GB$CDOM_NOT_FOUND_ERR|->_siltmp$2:
  * &GB$CDOM_NODE_ELEMENT|->_siltmp$3:
  * &GB$CDOM_INVALID_CHARACTER_ERR|->_siltmp$4:
  * &GB$CDOM_HIERARCHY_REQUEST_ERR|->_siltmp$5: * _siltmp$7|->_siltmp$6:
  * _siltmp$13|->_siltmp$12: * _siltmp$15|->_siltmp$10:
  * _siltmp$16|->{children:0, childText:_siltmp$13, parent:0,
  tagName:_siltmp$15, type:_siltmp$11}:

PROP 5:  &main$n|->_siltmp$17:
  * &main$t|->_siltmp$18: * &main$tmp___1|->_siltmp$7:
  * &main$s|->_siltmp$16: * &main$tmp___0|->_siltmp$16:
  * &main$tmp|->_siltmp$17: * &GB$CDOM_INDEX_SIZE_ERR|->_siltmp$9:
  * &GB$CDOM_NODE_TEXT|->_siltmp$10: * &GB$CDOM_GENERAL_ERR|->_siltmp$11:
  * &GB$CDOM_NO_DATA_ALLOWED_ERR|->_siltmp$8: * &main$argc|->_siltmp$0:
  * &main$argv|->_siltmp$1: * &GB$CDOM_NOT_FOUND_ERR|->_siltmp$2:
  * &GB$CDOM_NODE_ELEMENT|->_siltmp$3:
  * &GB$CDOM_INVALID_CHARACTER_ERR|->_siltmp$4:
  * &GB$CDOM_HIERARCHY_REQUEST_ERR|->_siltmp$5: * _siltmp$7|->_siltmp$6:
  * _siltmp$14|->_siltmp$15: * _siltmp$16|->_siltmp$12:
  * _siltmp$17|->{children:0, childText:_siltmp$14, parent:0,
  tagName:_siltmp$16, type:_siltmp$10}: * _siltmp$18|->_siltmp$13:
```

Figure 9: SpaceInvader generated assertions for the program in Figure 8

In the assertions SpaceInvader has found, the implementation details of CDOM have "spilled out" of the simple API. The varying heaps asserted are the possible results of the program, and do describe the memory states that can be arrived at, but because of the error reporting mechanism of C and need to encode the high level tree structure in the low level heap, the simple meaning of the program is very hard to discern.

2.3 Context Logic

Little work has been done on automating reasoning that uses Context Logic. Work by Dinsdale-Young uses NDFS automata to show that satisfiability and model checking of the (quantifier-free) multi-holed logic is decidable. For each formulae involving structural connectives, one can construct an automata that accepts a suitably encoded data structure, such as a list or tree, only when the data is a model of the corresponding formulae. Model checking becomes a case of running the generated automata, and satisfiability a search to see if the automata admits a path to an accepting state. Both techniques are well studied in automata theory.

This result is perhaps surprising, given the structure of satisfaction for the structural connectives. Useful perhaps for our work, it shows that the expressive power of the logic is no greater than that of regular languages. However, the work remains a decidability result, rather than a practical algorithm; the size of some generated automata are highly exponential in the size of the formulae. Moreover, no work has been performed at all for decidability of the Segment logic.

This is evidently an area that needs exploration, and we outline our first efforts in chapter 4.

3 Web reasoning and the mashup application

Featherweight DOM, along with Smith's DOM Core Level 1, shows that it is possible to formally specify the DOM API, embedded in a simple imperative language. Whilst a powerful result, it says

nothing about either the practicality of the approach, or what it can tell us regarding programs using DOM. It is not, as it stands, a basis we can use for exploring real engineering issues.

The Reasonable Web that occupied the first months of our research aims to define a sane perimeter to the formal mashup problem, capturing enough of the realistic issues to be compelling, in a compact enough manner to be tractable. The system is intended to support exploration within a large class of "real world" engineering problems. More than just the results specific to the mashup issues we investigate, this should show that Context Logic can be helpful in software engineering issues (much like Separation Logic and its current applications). It also completes a cycle of work on the use of high level specifications for libraries, spanning specification through to client program usage.

The rest of this chapter is taken mostly verbatim from the second draft of our paper, *Resource Reasoning for mashups*, edited to remove some redundancies with the rest of this report. It is a draft paper, and whilst the key ideas are complete, some reworking will occur before submission.

3.1 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 is indistinct. Originally a simple textual markup system, the technologies underlying the web have been augmented to a point where developers expect their users to have web browsers that can execute powerful embedded scripting languages, consume data in standardised formats, and download disparate resources through fast, persistent Internet links. This confluence of features has provided the seed of yet another evolutionary step: *Mashups*.

There is no formal definition of a mashup, but musicians may recognise the term as music composed by the aggregation of other song fragments. The sense here is identical, as mashups refer to web applications created by composing together several other external resources. The user visits a web page where embedded scripting code utilises code and data from other sites, either initially when fetched or dynamically in response to user interaction. Whether these resources were intended for reuse or were simply co-opted, the effect remains a page constructed by "mashing together" disparate pieces from multiple sources.

Developers create these mashups for many reasons. Sometimes, the technique is used because the author lacks the experience to write all the code themselves. They look to an external component authored by more experienced developers to provide some functionality. Sometimes, they are appealing to data that is private to another organisation, and partially shared through exposed components. Often though, they are simply a route to code reuse, easing the development of a complex application.

A common example is the geographical map, augmented with additional features. Such a map requires a significant database of expensive information coupled with expertise in developing accessible user interfaces. These data and skills are orthogonal to the problem that many developers are solving; they simply wish to convey, say, the location of their business along with their own information (perhaps local bus stops or recommended restaurants). 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. Other examples emerge constantly: the ubiquitous news feed, taking data from news outlets, selecting interesting stories and presenting it as an integrated part of the another site; social networking applications, where users can host features from Facebook on their own personal pages. The list is large and expands daily.

3.1.1 Problems with mashup development

Embedded scripts within pages suffer from standard software development challenges. The imperative heap used by JavaScript (the ubiquitous scripting language) introduces the standard problems that aliasing brings. The complex heap structure means authors are often unsure of the shape of data. Alongside these more typical concerns, mashup programming introduces significant challenges that are quite different from those of desktop software development.

Mashup component dependencies are resolved at runtime, and each component is imported into the shared memory execution environment of the users browser. Their successful use requires that the remote code and data depended on are still available and, more importantly, still behave as expected. Programmers using the components of others must hope they have not used some subtle detail of implementation the provider has changed, trust that some XML schema has not been updated, and believe the provider has not introduced new code in a namespace that clashes with another. Choosing to mashup is a choice to become highly dependent on the immutability of other web pages, which are typically highly changeable.

Hazards are also implicit in the nature of the shared execution situation of browser-based scripting. JavaScript allows code and data to be dynamically loaded and replaced. If the way a user has interacted with a page did not execute code that fetched a needed component, then further execution will fail when attempts are made to use behaviours it would have provided. If it happened to fetch a component that replaced named functions with incompatible alternatives, the expected behaviour will not occur. The language includes very few features to help avoid these problems. It is largely untyped, provides no method for declaring formal interfaces for code, and is quite happy to allow programs that appear to be separate full access to interfere with each other when run in the same environment.

Dependencies are also essentially unknown and unbounded, and as such there is no easily identifiable definition of a "mashup program". For example, if we use a third party service, we implicitly become dependent on any components it has used. These may not be clear and may change, rendering it very difficult to know exactly what web servers will be accessed by a user of a mashup. Especially in a firewalled or restricted environment, this can cause real difficulties in guaranteeing a robust user experience.

3.1.2 Structure of contributions

We first define what we mean by mashup, by introducing the *component* concept. We give a data structure encoding a subset of XML that represents the heap updated by our language. We provide a scripting language with associated operational semantics, analogous to and inspired by JavaScript but not seeking to replicate JavaScript semantics exactly. The language includes a minimal implementation of the W3C DOM API (extended from the presentation in FWDOM), as well as features that model the dynamically loading of other resources (which we abstract into two representative commands). It contains a dynamic function presentation, whereby functions can be introduced and redefined during program execution. We combine these into an *Abstract Web* by considering components as seen by a higher level portion of the browser software - pre-parsed and interpreted structures indexed by URIs - and show this view sensibly captures the problems seen by mashup authors.

We provide a reasoning system for the development and verification of mashup components written for our abstraction. In particular, we use the resource reasoning introduced by O'Hearn, which is based upon an analysis of a programs use of resource; for example, the heap when using Separation Logic to reason about C programs, and in our case, tree structures using Context Logic. We extend the presentation of [16] to provide an assertion language that can capture strong properties of our programs, and give a Hoare style logic for program verification.

We show how this language can be used to provide *specifications* of mashup components, that are sets of Hoare triples describing the data provided, as well as pre and post conditions of code. Each component is also given a *private specification* that matches and extends upon the public version. We show how this combination allows a proof theory based on component-wise reasoning; each resource is proven correct in isolation, dependant only on the public specifications of the others. They may also change freely, as long as alterations do not violate their public specifications. We discuss how basing our assertion languageon Context Logic allows us to give concise specifications and, moreover, that these specifications are a good match to the notations that programmers have when developing. We posit that these specifications form a natural interface to mashup components.

We show that our proof theory is sound, and give meta-theoretic results stating that, under our system, one can give a formal definition of mashup program, and construct *fault free* webs where users will never encounter unexpected errors. We demonstrate a proof of a mashup, built throughout the exposition. We conclude with related and future work, in particular discussing the automation of our proof theory as well as extensions to larger subsets of JavaScript and XML.

3.2 A semantics for mashups

We now introduce our *Abstract Web* as an operational model of the mashup problem. The full World Wide Web is a huge and complex entity, made from many types of arbitrary resources indexed by URIs. These raw resources are parsed by a *Web Browser*, which a user typically instructs to request HTML documents. These documents encode tree structured data, where data are the children of semantically and presentationally significant labeled nodes called *tags*. The browser parses these into an internal data structure, which is rendered as an interactive view of the page.

The tags instruct the browsers page presentation, and many either reference or embed other resources. Of particular interest is `SCRIPT`, which denotes either a link to or an embedding of a code resource. Almost always written in the JavaScript language, these script programs are executed automatically as the page is parsed, or in response to a user interaction. They have virtually full control over an environment consisting of: The current page state, parsed from the HTML document, and interfaced with via the W3C DOM API; The JavaScript heap, a store for JavaScript specific state; and the whole Internet, via a technique commonly called AJAX (among others), scripts can download and act upon resources from other URIs.

JavaScript is an untyped, late bound language, perhaps most reminiscent of Self. Functions are first class objects stored within the runtime state, and can be replaced or removed at will by script. Moreover, the language includes an `eval` command, which executes arbitrary strings as JavaScript programs within the current state. The language has evolved from initially simple roots into a remarkably complex system - the first known complete operational semantics for the language [20] runs to a daunting 30 pages.

To concentrate on the mashup issues, we will view the web as a set of mashup *components* indexed by URI. These components represent just the resources key to the problem: Tree structured documents (ala HTML) and a language that, whilst less complex than JavaScript, captures the issues of dynamic functions, tree shaped heap, and remote resource usage. We will be considering the actual execution of mashup "programs", and as such our data structures represent the facets available to script code at runtime.

3.2.1 Data structures

We first define the XML like data structure. Rather than modelling full HTML, we use a subset that elides node types we do not need. This data structure, given in 10 represents the in-memory

view of the document that our language will access, in essence providing a "tree shaped heap". We base this work on Featherweight DOM[3]. Tree nodes take the form

$$\mathbf{s}_{\mathbf{nodeID}\ \mathbf{type}}^{\mathbf{docID}}[\mathbf{F}]_{\mathbf{fid}}$$

Each consists of a name, **s**, the DOM type identifier of the node, **type**, a child *forest* **F**, and identifiers **nodeID**, **docID** and **fid**. These identifiers can be intuited as the address of the specific node, the owner document, and the child forest respectively. These identifiers must be unique across a document. Forests correspond to a flexible list of children belonging to a node. The grove is the top level of this heap structure. Many documents may exist in the heap, but it also serves as a place for nodes without a parent to exist.

We model standard element nodes, text nodes and document nodes, ignoring the others (i.e. attributes). This subset is large enough to construct real programs around, whilst retaining presentational simplicity. Notice that each different part of the data structure is *typed*, written G for grove, F for forest, T for tree, S for string and DE for document elements; in general, we say that **d** has type D (with D in the listed types). This typing structure is important for maintaining the sanity of the data structure, and defining the assertion language.

We define a congruence between DOM data structures, $\equiv$, that obeys the restrictions given in the figure. We use some notation to reduce verbosity in practice.

Notation 1. For strings $\langle'a'\rangle_{\mathbf{S}} \otimes_{\mathbf{S}} \langle'b'\rangle_{\mathbf{S}} \otimes_{\mathbf{S}} \langle'c'\rangle_{\mathbf{S}} \ldots$, we write just "abc...", and we will often omit type annotations when they are clear.

Consider this XML fragment of a document listing places on a map.

```
<place>
  Imperial DoC
  <postcode>
    SW7 2AZ
  </postcode>
</place>
```

One possible representation of it in our data structure is

$$\text{“\#document”}_{\mathbf{id1}\ 9}^{\mathbf{null}} \left[\begin{array}{l} \text{“place”}_{\mathbf{id3}\ 1}^{\mathbf{id1}} \left[\begin{array}{l} \text{“\#text”}_{\mathbf{id5}\ 2}^{\mathbf{id1}}[\text{“Imperial DoC”}]_{\mathbf{id6}} \\ \otimes \\ \text{“postcode”}_{\mathbf{id7}\ 1}^{\mathbf{id1}}[\text{“\#text”}_{\mathbf{id9}\ 2}^{\mathbf{id1}}[\\ \quad \text{“SW7 2AZ”} \\ ]_{\mathbf{id10}}]_{\mathbf{id8}} \end{array} \right]_{\mathbf{id4}} \end{array} \right]_{\mathbf{id2}}$$

Note that the identifiers that have been created, and the explicit DOM structure is now evident. We cannot just reuse XML notation; embedding these within it would create confusion between the serialised representation that XML gives, and the "memory model" view of the DOM data structure. There are many possible representations, depending on the identifiers chosen.

3.2.2 The store

Our language is imperative; both the tree shaped heap can be mutated at runtime, as well as the values assigned to variable names. The store represents the mapping of these variable names to their current value.

[3]DOM being the Document Object Model, a W3C standard for accessing tree data from code

groves	$\mathbf{g}$ ::=	$<\mathbf{doc}>_{\mathrm{G}} \mid <\mathbf{ele}>_{\mathrm{G}} \mid <\mathbf{txt}>_{\mathrm{G}}$ $\mid \varnothing_{\mathrm{G}} \mid \mathbf{g} \oplus_{\mathrm{G}} \mathbf{g}$
documents	$\mathbf{doc}$ ::=	$\text{“\#document”}_{\mathbf{id}_9}^{\mathbf{null}}[\mathbf{de}]_{\mathbf{fid}}$
document element	$\mathbf{de}$ ::=	$<\mathbf{ele}>_{\mathrm{DE}} \mid \varnothing_{\mathrm{DE}}$
elements	$\mathbf{ele}$ ::=	$\mathbf{s}_{\mathbf{id}_1}^{\mathbf{idref}}[\mathbf{f}]_{\mathbf{fid}}$ (‘#’ is not in $\mathbf{s}$)
element forests	$\mathbf{f}$ ::=	$<\mathbf{ele}>_{\mathrm{F}} \mid <\mathbf{txt}>_{\mathrm{F}} \mid \varnothing_{\mathrm{F}} \mid \mathbf{f} \otimes_{\mathrm{F}} \mathbf{f}$
text	$\mathbf{txt}$ ::=	$\text{“\#text”}_{\mathbf{id}_3}^{\mathbf{idref}}[\varnothing_{\mathrm{F}}]_{\mathbf{fid}}$
strings	$\mathbf{s}$ ::=	$<\mathbf{c}>_{\mathrm{S}} \mid \varnothing_{\mathrm{S}} \mid \mathbf{s} \otimes_{\mathrm{S}} \mathbf{s}$
characters	$\mathbf{c}$ ::=	‘a’ | ‘b’ | ‘c’ . . .

where $\mathbf{id}, \mathbf{fid}, \mathbf{idref} \in \mathrm{ID}$ and $\mathbf{id}, \mathbf{fid}$ 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 $\varnothing_{\mathrm{D}}$.

Figure 10: DOM Data Structure

Our language will use four primitive types; the standard String ($\mathbb{S}$), integer ($\mathbb{Z}$) and boolean ($\mathbb{B}$) values, as well as the ID ($\mathbb{I}$) type, representing DOM addresses. Each has a corresponding set of variable names, $\mathbf{Var}_{\mathbb{D}}, \mathbb{D} \in \{\mathbb{S}, \mathbb{Z}, \mathbb{B}, \mathbb{I}\}$. To represent the JavaScript store, we have a function s from these names to the values they hold.

Definition 2 (Store)**.** The *Store* is a total function, mapping from variable names to values.

$$s : (\boldsymbol{Var}_{\mathbb{S}} \to \mathbb{S}) \times (\boldsymbol{Var}_{\mathbb{Z}} \to \mathbb{Z}) \times (\boldsymbol{Var}_{\mathbb{B}} \to \mathbb{B}) \times (\boldsymbol{Var}_{\mathbb{I}} \to \mathbb{I})$$

This store will be updated as the program executes.

Notation 3. We write function update as $s[n \leftarrow v]$, defined as

$$s[n \leftarrow v](x) = \begin{cases} v \text{ if } n = x \\ s(x) \text{ otherwise} \end{cases}$$

3.2.3 The Language

In our language we aim to capture elements of the JavaScript spirit as it pertains to mashups, but are not aiming to replicate the language semantics. For example, our functions are not fully first class, but can be dynamically introduced and replaced. This captures the issues of ensuring that functions exist in the way mashup code expects, whilst giving a compact presentation. Similarly, we elide the JavaScript object system, but allow the *component local* variables and functions typically implemented via it. Again, we capture the usage in terms of mashups, without burdening ourselves unduly[4]. We do not consider events in this version of the work, leaving it for later.

The core of our language is an group of sequenced imperative commands including conditional and looping structures, along with user definable functions[5]. The complete grammar is given in figure 12. We distinguish between commands, which may mutate the heap and store, and expressions, which may not. The later, given in 11, are standard for the data types we include.

[4] We choose not to consider JS Object Notation (JSON) and its use for data transfer in this paper, as XML documents can accomplish the same work

[5] We use function in the JavaScript sense of named parameterised code blocks, even though our presentation might otherwise be called “procedures”

The commands of the language split in two; those for heap manipulation and those for data and code retrieval.

Our heap is modelled after the W3C DOM standard, and our commands follow the same ideal. As in previous work ([16]), we formally and compositionally specify a subset of the commands we wish to use, and can implement remaining commands in terms of that subset. We eschew the object oriented syntax of DOM for simplicity, and use a flattened version of the node interface (as sanctioned in [2]). Note that our language has no destructive commands for heap elements; once created, nodes cannot be explicitly destroyed. We thus naturally consider our language garbage collected. Among the primitive commands are

> `createElement(IdExpr, StrExpr);` Creates a new element in the grove, with the node name given by `StrExpr`, and parent document identifier by `IdExpr`. Faults if `IdExpr` does not reference a document node, or if `StrExpr` contains the "#" character.
>
> `Str = getNodeName(IdExpr);` Assigns the name of the node referenced by `IdExpr` to the variable `Str`. Faults if `IdExpr` does not reference a node.
>
> `Id = adoptNode(IdExpr_D, IdExpr_N);` Removes `IdExpr_N` from its parent, and transfers the ownership of it (and its children) to the document identified by `IdExpr_D`. Returns `IdExpr_N`, and faults if `IdExpr_D` does not reference a document, or `IdExpr_N` does not reference a node.
>
> `appendData(IdExpr, StrExpr);` Appends the string in `StrExpr` to end of the value of the text node identified by `IdExpr`. Faults if `IdExpr` does not reference a text node.
>
> `Id = item(IdExpr, IntExpr);` Obtains the `IntExpr`$^{\text{th}}$ element from the forest represented by `IdExpr`, storing the result in `Id`. If there are fewer than `IntExpr` elements in the forest, or if it is negative, returns **null**. The command will fault if `IdExpr` does not identify a forest.

The other DOM library commands are analogous to those demonstrated and behave as one would expect. We give a complete accounting in [15].

The remaining commands, key to modelling mashups, are those that allow data and code to be obtained from external resources. Whereas JavaScript and HTML provide many mechanisms for acquiring remote resources, we shall model the AJAX case typically used in mashups. In this scenario, users issue a JavaScript command to download an arbitrary resource from a URI. They then interpret that result as they wish, perhaps parsing it as an XML document or using `eval` to run it as a script. We give two explicit commands for the interesting cases.

> `Id = fetchDocument(StrExpr);` Parses the XML document at the URI expressed via `StrExpr` into a data structure. Adds the resultant document to the grove, storing the document ID in `Id`. Faults if `StrExpr` evaluates to an unknown URI.
>
> `runScript(StrExpr);` Parses the XML document at the URI expressed via `StrExpr`, extracting and concatenating any referenced or embedded scripts to produce a single program, which it then executes in the current environment. Faults if `StrExpr` evaluates to an unknown URI.

One is free to make as many calls to these commands as needed, even reusing the same URIs. The result be multiple copies of the same document in the grove, or multiple executions of the same script code.

```
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
```

Figure 11: Grammar for expressions

3.2.4 Functions

A main part of building a mashup component is being able to reuse behaviour provided by others. Functions provide this ability for our language, giving authors the chance to create named and parameterised blocks of reusable code to share. However, our user defined functions behave slightly unusually. Rather than being a property of the program, they are a property of the program state. Initially empty, the program populates a *function table* as it executes, accruing functions from either local code, or those mashed in via `runScript`. This table is highly mutable - a function may not exist at one point, be added later during execution, and replaced with an entirely new behaviour later still.

To this end, we consider the `function` construct to be an actual statement, with the imperative result of updating the function table.

Definition 4 (Function table)**.** A function table is a partial function **FT** from function names to the tuple of parameter names, and associated code.

$$\mathbf{FT} : \text{FNames} \rightharpoonup \overline{\text{VarNames}} \times \mathbb{C}$$

Function calls may be mutually recursive. Function bodies cannot directly introduce other functions, nor can they cannot assign to any of their formal parameter names.

3.2.5 Identifier scope

The visibility of identifiers to code is one major issue in mashup component authoring. By default, all identifiers in JavaScript are entered into the same global scope - a decision useful for sharing code, but also allowing for damaging interactions. However, programmers can declare variables that are local to functions, and by use of the object system, functions and variables that are localised to just one subsection of code.

We capture this concept by breaking our language into two scopes. *Component* scope contains code outside of function definitions. Inside function definitions, we are in *Function* scope. The primary difference is that we do not allow nested function declarations - all functions must be introduced at component scope.

Like JavaScript, any identifier introduced in any of these scopes will by default be visible to all code (even if imported from another URI with `runScript`). This is not always what the programmer wants, so we allow identifiers to be constrained to their declaring scope with the `local` annotation. Variables local to a function will not be visible outside the function body (or

```
Program ::=
    FStatement
  | local? function FName (Var, ..., Var)
    {
      local Var (, Var)*;
      FStatement*
      return Expression;
    }
  | Program ; Program

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

Command ::=
    Id  = createElement(IdExpr, StrExpr)
  | Id  = createTextNode(IdExpr, StrExpr)
  | Str = getNodeName(IdExpr)
  | Int = getNodeType(IdExpr)
  | Id  = getParentNode(IdExpr)
  | Id  = getChildNodes(IdExpr)
  | Id  = getOwnerDocument(IdExpr)
  | Id  = adoptNode(IdExpr, IdExpr)
  | Id  = appendChild(IdExpr, IdExpr)
  | Id  = removeChild(IdExpr, IdExpr)
  | Id  = item(IdExpr, IntExpr)
  | Str = substringData(IdExpr, IntExpr, IntExpr)
  | Id  = fetchDocument(StrExpr)
  | runScript(StrExpr)
  | deleteData(IdExpr, IntExpr, IntExpr)
  | appendData(IdExpr, StrExpr)
```

Productions begin with upper case letters, literals with lower case. We use unnamed rules `Name`, `Id`, `Str`, `Int`, `Bool` for the set of all variable names, and sets of typed variable names respectively.

Figure 12: Grammar for statements

```
function getNodeValue(n) {
  local i, s, t;
  i = 0; s = "";

  while (s != null) {
    t = s;
    s = substringData(n, i);
    i = i + 1;
  };

  return t;
};

kids = getChildren(root);
firstLocation = item(kids, 0);

if (firstLocation != null) {
  name = getNodeValue(firstLocation);
  moreInfo = getChildren(firstLocation);
  postcodeNode = item(moreInfo, 0);
  postcode = getNodeValue(postcodeNode);
} else {
  skip;
}
```

Figure 13: Example program. When executed against the grove given in subsection 3.2.1, concludes with name set to "Imperial DoC" and postcode "SW7 2AZ"

across function calls); Functions and variables local to component scope will not be visible to other components.

Over code $\mathbb{C}$, we use a function $local(\mathbb{C})$ that returns the set of local identifiers in the scope of $\mathbb{C}$. If given component code, it returns the local function names and local variables of that scope; if given a function body, it returns the local variables of the function.

3.2.6 The Abstract Web

We are interested in documents consisting of XML and script code, either embedded or references. Rather than deal with the explicit parsing of documents and the extraction of scripts, we view the web through the lens of a browser that has already performed the download and parsing phase.

We define an abstract web W as a partial function from the set of URIs $\mathbb{U}$ to a pair. The first element is the data structure representing the parsed XML; the second, any embedded or referenced code. The parsing does not remove code that was directly embedded in the document, which remains in the data structure, but is seen simply as a string with no special properties. We assume that the ID uniqueness property is preserved by W, so that all returned data structures have fresh identifiers.

Definition 5 (Abstract Web)**.** Let $\mathbb{U}$ be the set of URIs, and $\mathbb{C}$ be the set of all code valid in the grammar of figure 12. The partial function W represents a mapping from a given URI to the parsed data and code available there. Unavailable resources are modeled by absence from the

domain of W.

$$W : \mathbb{U} \rightharpoonup \mathbf{g} \times \mathbb{C}$$

If the parsing of a URI contains no XML data, the returned data $\mathbf{g} = \varnothing_{\mathrm{G}}$. Similarly, if it contains no code, the returned code will be `skip`.

We say that a web W is *closed* is no possible execution of a program contained in it would request a resource not in dom(W) via `fetchDocument` or `runScript`.

3.2.7 Operational formalism

Expressions are evaluated with respect to a store in the standard fashion, and we denote the evaluation of E with respect to s as $[\![\mathtt{E}]\!]_{\mathbf{s}}$. The meaning of the language is given by a small step operational semantics, using a judgement $\rightsquigarrow$. We use some notation so that $x_1, \ldots, x_n$ is rendered as $\overline{x}^n$, with the set of the elements as $\{\overline{x}^n\}$.

Definition 6. The evaluation relation $\rightsquigarrow$ is defined from tuples of states, function tables, groves and code to either a terminal state, function table and grove, or the **fault** state.

$$s, \mathbf{FT}, g, \mathbb{C} \rightsquigarrow s', \mathbf{FT}', \mathbf{G}' \text{ or } \mathbf{fault}$$

We show the more interesting rules below. The others are either standard, or can be found in [15]. In these judgements, we found it useful to use the context concept we introduce in subsection N. We handle scope issues via substitution of fresh names for any identifier declared local (effectively alpha conversion of any local names). Freshness is with respect to an explicit store, and implicit W; we never choose an identifier that we have created, or may create. The operation is given by

$$\Theta(\mathbb{C}, s) = \mathbb{C}[f_1/v_1, \ldots, f_n/v_n]$$

Where $v_i \in \mathit{local}(\mathbb{C})$ and f_i is fresh in both s and $\mathit{free}(W)$.

In line with our "Functions are just program state" ethos, function introduction simply adds the new code to the function table. Notice that any previous function with the same name is lost.

$$\frac{}{\begin{array}{c} s, \mathbf{FT}, \mathbf{g}, \texttt{function f(p1, ..., pm) \{ C \}} \\ \rightsquigarrow s, \mathbf{FT}[\texttt{f} \leftarrow (\overline{p}^m, \texttt{C})], \mathbf{g}' \end{array}} \quad \textsc{Func-Intro}$$

Function call first finds the matching function name from the function table, and extracts the parameter names and code. All local names in the code are freshened, and then the passed expressions are substituted through for the appropriate parameter names. This code is executed, resulting in a new store, function table and grove. The result of the call is this grove, and a store with the return value assigned appropriately.

$$\frac{\begin{array}{c} \mathbf{FT}(\texttt{f}) = (\overline{p}^m, \texttt{C}) \\ s, \mathbf{FT}, \mathbf{g}, \Theta(\texttt{C}, s)[e_i/p_i] \rightsquigarrow s', \mathbf{FT}', \mathbf{g}' \end{array}}{\begin{array}{c} s, \mathbf{FT}, \mathbf{g}, \texttt{v := f(e1, ..., en)} \\ \rightsquigarrow s'[\mathrm{ret}/\mathrm{v}], \mathbf{FT}', \mathbf{g}' \end{array}} \quad \textsc{Func-Call}$$

The `fetchDocument` rule appeals to the Abstract Web, finding the data structure associated with the URI. If it cannot be found, a fault occurs, and if returned correctly, it is simply added to our grove with its ID stored in the required variable.

$$\frac{\begin{array}{c}W([\![\texttt{uri}]\!]_s) = (\#\mathbf{document}_{\mathbf{id9}}^{\mathbf{null}}[\mathbf{f}]_{\mathfrak{fid}}\mathbf{null}, \mathbb{C}) \wedge \\ \mathbf{g}' \equiv \mathbf{g} \oplus <\#\mathbf{document}_{\mathbf{id9}}^{\mathbf{null}}[\mathbf{f}]_{\mathfrak{fid}}\mathbf{null}>_{\mathrm{G}}\end{array}}{\begin{array}{c}s, \mathbf{FT}, \mathbf{g}, \texttt{n} := \texttt{fetchDocument}(\texttt{uri}) \\ \rightsquigarrow s[\texttt{n} \leftarrow \mathbf{id}], \mathbf{FT}, \mathbf{g}'\end{array}} \quad \textsc{fetchDoc}$$

`runScript` can be viewed as a combination of function calling and document fetching. We must find the code associated with a URI, and freshen any local names, and then pass control to the code. Notice that code is explicitly executed in the same environment as the `runScript` command itself.

$$\frac{\begin{array}{c}W([\![\texttt{uri}]\!]_s) = (\mathbf{g}, \mathbb{C}) \wedge \\ s, \mathbf{FT}, \mathbf{g}, \Theta(\mathbb{C}, s) \rightsquigarrow s', \mathbf{FT}', \mathbf{g}'\end{array}}{s, \mathbf{FT}, \mathbf{g}, \texttt{runScript}(\texttt{uri}) \rightsquigarrow s', \mathbf{FT}', \mathbf{g}'} \quad \textsc{runScript}$$

3.3 Mashup specifications

Our mashup abstraction captures the key faults encountered when trying to construct mashup programs. We now consider how to *specify* mashups programs, and use these specifications to understand how they function, and guarantee that they cannot encounter these problems.

Like real mashups, faults occur in our language when

1. The shape of the heap is not that required by code
2. Functions which are called either do not exist, or are not the expected implementation
3. Code cannot find a remote resource when it is needed

The first is common to all programs, especially those using a mutable heap structure. Local reasoning has proven highly effective as a basis for assertion languages capturing a precise abstract semantics, and we shall take advantage of this. The later two errors are far more specific to the mashup problem we are modelling.

3.3.1 Context Logic

We develop now a language for formally expressing statements about our programs. The assertion language must capture properties of the store, the DOM heap, and the state of the function environment.

The basis of our reasoning is the Single Holed Context Logic introduced in 2.1.4. To enable this splitting of our data structure, we introduce *DOM Contexts* in figure 14. It shows the allowed location of the single admissible hole within the structure (there are multi-holed variants of Context Logic, but we can express our properties using only the single holed variety). Alongside this structure is a function $\mathbf{ap}(\mathbf{cd}, \mathbf{d})$, which fills the hole in $\mathbf{cd}$ with the datum $\mathbf{d}$, returning a complete structure. It is undefined when the hole type and data type does not match; we write $ap(\mathbf{cd}, \mathbf{d}) \downarrow$ to mean it is defined for arguments $\mathbf{cd}$ and $\mathbf{d}$.

Recall that 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, $P \circ\!\!- _{\mathrm{D}_2} P_2$ 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 $P_1 \mathrel{-\!\!\circ} P_2$ 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.

The formulae for our assertion language are given in figure 15. Notice that we type formulae, so there is a specific falsity, zero etc for trees, forests, groves and strings. The formulae $\gamma(\mathbf{f})$ is a form of existential assertion for functions, stating that some function with name $\mathbf{f}$ is present. Similarly, $\omega(\mathbf{uri})$ states that the abstract web has an entry for **uri**. The free names of a formulae are standard, except that we also consider $\mathbf{f}$ to be free in $\gamma(\mathbf{f})$.

3.3.2 Satisfaction

The semantic meaning of our formulae is given by a forcing relation, defined over a store, data structure, and a *logical environment*. This environment assigns values to context variables and *ghost* variables, which represent purely logical variables that remain invariant.

Definition 7. The logical environment e is a function from context and ghost variable names to their values. These values include DOM structure

The satisfaction relation is typed, so tree formulae are satisfied by trees, groves by groves and so forth. We include types for all the contexts that are actually used.

Definition 8 (Satisfaction)**.** The typed satisfaction relation $\vDash_{\mathrm{A}}$ is of the form

$$e, s, \mathbf{FT}, \mathbf{a} \vDash_{\mathrm{A}} P$$

where e is a logical environment, s a store, $\mathbf{FT}$ a function table, $\mathbf{a}$ a datum and P a formulae of our logic. The complete relation is given in figure 7.

3.3.3 Derived formulae

We derive the omitted formulae of first order logic in the standard way. We also introduce some notation for shorthand node descriptions.

Notation 9.

$$\begin{aligned} P_{\mathtt{id}}[P'] &\triangleq \exists \mathtt{fid}, \mathtt{oid}. P_{\mathtt{id1}}{}^{\mathtt{oid}}[P']_{\mathtt{fid}} \\ P[P'] &\triangleq \exists \mathtt{id}. P_{\mathtt{id}}[P'] \\ \#\text{text} &\triangleq \exists \mathtt{id}, \mathtt{oid}. \#\text{text}_{\mathtt{id2}}{}^{\mathtt{oid}}[\text{true}_{\mathrm{S}}] \end{aligned}$$

$$
\begin{array}{rl}
\mathbf{cg} ::= & <\mathbf{cdoc}>_{\mathrm{G}} \mid <\mathbf{cele}>_{\mathrm{G}} \mid <\mathbf{ctxt}>_{\mathrm{G}} \mid -_{\mathrm{G}} \mid \mathbf{cg} \oplus_{\mathrm{G}} \mathbf{g} \\
\mathbf{cdoc} ::= & \text{“\#document”}_{\mathbf{id}9}^{\mathbf{null}}[\mathbf{cde}]_{\mathbf{fid}} \mid -_{\mathrm{DOC}} \\
\mathbf{cde} ::= & <\mathbf{cele}>_{\mathrm{DE}} \mid -_{\mathrm{DE}} \\
\mathbf{cele} ::= & \mathbf{s}_{\mathbf{id}1}^{\mathbf{idref}}[\mathbf{cf}]_{\mathbf{fid}} \mid -_{\mathrm{ELE}} \qquad (\text{`\#'} \notin \mathbf{s}) \\
\mathbf{cf} ::= & <\mathbf{cele}>_{\mathrm{F}} \mid <\mathbf{ctxt}>_{\mathrm{F}} \mid -_{\mathrm{F}} \mid \mathbf{cf} \otimes_{\mathrm{F}} \mathbf{f} \mid \mathbf{f} \otimes_{\mathrm{F}} \mathbf{cf} \\
\mathbf{ctxt} ::= & \text{“\#text”}_{\mathbf{id}3}^{\mathbf{idref}}[\varnothing_{\mathrm{TF}}]_{\mathbf{fid}} \mid -_{\mathrm{TXT}} \\
\mathbf{cs} ::= & <\mathbf{cc}>_{\mathrm{S}} \mid -_{\mathrm{S}} \mid \mathbf{cs} \otimes_{\mathrm{S}} \mathbf{s} \mid \mathbf{s} \otimes_{\mathrm{S}} \mathbf{cs} \\
\mathbf{cc} ::= & -_{\mathrm{C}}
\end{array}
$$

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

Figure 14: Local DOM Contexts

$$
\begin{array}{rll}
P ::= & P \Rightarrow P \mid \mathrm{false}_{\mathrm{D}} \mid \mathrm{false}_{\mathrm{D}_1 \to \mathrm{D}_2} & \text{Boolean formulae} \\
& P \circ_{\mathrm{D}} P \mid P \circ\!\!-_{\mathrm{D}_2} P \mid P \multimap P & \text{structural formulae} \\
& \ldots (\text{See below}) \ldots & \text{DOM formulae} \\
& \mathtt{var}_{\mathrm{E}} \mid \mathtt{Exp}_{\mathrm{V}} = \mathtt{Exp}_{\mathrm{V}} & \text{expression equality} \\
& \mathtt{Int} = |\mathtt{f}| & \text{length equality} \\
& \exists \mathtt{var}.\, P & \text{quantification} \\
& \gamma(\mathbf{f}) & \text{function existence} \\
& \omega(\mathbf{uri}) & \text{Component existence}
\end{array}
$$

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

$$
\begin{array}{rl}
P ::= \ldots & \mid -_{\mathrm{D}} \mid P_{\mathtt{id}_{\mathtt{tp}}^{\mathtt{irn}}}[P]_{\mathtt{fid}} \mid \\
& \text{“\#text”}_{\mathtt{id}3}^{\mathtt{idref}}[\varnothing_{\mathrm{TF}}]_{\mathtt{fid}} \mid \mathbf{c} \mid \\
& P \otimes_{\mathrm{D}} P \mid P \oplus_{\mathrm{D}} P \mid \varnothing_{\mathrm{D}} \mid <P>_{\mathrm{D}}
\end{array}
$$

Figure 15: Formulae of our logic

$$
\begin{array}{ll}
e,s,\mathbf{FT},\mathbf{a} \models_{\mathrm{A}} P \Rightarrow P' & \iff e,s,\mathbf{FT},\mathbf{a} \models_{\mathrm{A}} P \Rightarrow e,s,\mathbf{FT},\mathbf{a} \models_{\mathrm{A}} P' \\
e,s,\mathbf{FT},\mathbf{d} \models_{\mathrm{D}} \mathrm{false}_{\mathrm{D}} & \text{never} \\
e,s,\mathbf{FT},\mathbf{cd} \models_{\mathrm{D_1 \to D_2}} \mathrm{false}_{\mathrm{D_1 \to D_2}} & \text{never}
\end{array}
$$

$$
\begin{array}{ll}
e,s,\mathbf{FT},\mathbf{d}_2 \models_{\mathrm{D}_2} P_1 \circ_{\mathrm{D}_1} P_2 & \iff \exists \mathbf{cd}{:}(\mathrm{D}_1 \to \mathrm{D}_2), \mathbf{d}_1{:}\mathrm{D}_1 . \mathbf{d}_2 = \mathrm{ap}(\mathbf{cd},\mathbf{d}_1) \wedge \\
 & \quad e,s,\mathbf{FT},\mathbf{cd} \models_{\mathrm{D_1 \to D_2}} P_1 \wedge e,s,\mathbf{FT},\mathbf{d}_1 \models_{\mathrm{D}_1} P_2 \\
e,s,\mathbf{FT},\mathbf{d}_1 \models_{\mathrm{D}_1} P_1 \circ\!\!- _{\mathrm{D}_2} P_2 & \iff \forall \mathbf{cd}{:}(\mathrm{D}_1 \to \mathrm{D}_2) . (e,s,\mathbf{FT},\mathbf{cd} \models_{\mathrm{D_1 \to D_2}} P_1 \wedge \mathrm{ap}(\mathbf{cd},\mathbf{d}_1)\downarrow) \\
 & \quad \Rightarrow e,s,\mathbf{FT},\mathrm{ap}(\mathbf{cd},\mathbf{d}_1) \models_{\mathrm{D}_2} P_2 \\
e,s,\mathbf{FT},\mathbf{cd}_2 \models_{\mathrm{D_1 \to D_2}} P_1 \multimap P_2 & \iff \forall \mathbf{d}_1{:}\mathrm{D}_1 . e,s,\mathbf{FT},\mathbf{d}_1 \models_{\mathrm{D}_1} P_1 \wedge \mathrm{ap}(\mathbf{cd}_2,\mathbf{d}_1)\downarrow \\
 & \quad \Rightarrow e,s,\mathbf{FT},\mathrm{ap}(\mathbf{cd}_2,\mathbf{d}_1) \models_{\mathrm{D}_2} P_2
\end{array}
$$

$$
\begin{array}{ll}
e,s,\mathbf{FT},\mathbf{cd} \models_{\mathrm{D \to D}} -_{\mathrm{D}} & \iff \mathbf{cd} \equiv -_{\mathrm{D}} \\
e,s,\mathbf{FT},\mathbf{d} \models_{\mathrm{D}} P_{\mathtt{id}\,\mathtt{tp}}^{\mathtt{irn}}[P'']_{\mathtt{fid}} & \iff \exists \mathbf{s}{:}\mathrm{S}, \mathbf{d}''{:}\mathrm{D}'' . (\mathbf{d} \equiv \mathbf{s}_{[\![\mathtt{id}]\!]_s\,[\![\mathtt{tp}]\!]_s}^{[\![\mathtt{irn}]\!]_s}[\mathbf{d}'']_{[\![\mathtt{fid}]\!]_s}) \wedge \\
 & \quad e,s,\mathbf{FT},\mathbf{s} \models_{\mathrm{S}} P \wedge e,s,\mathbf{FT},\mathbf{d}'' \models''_{\mathrm{D}} P'' \\
e,s,\mathbf{FT},\mathbf{cd} \models_{\mathrm{D_1 \to D_2}} P_{\mathtt{id}\,\mathtt{tp}}^{\mathtt{irn}}[P'']_{\mathtt{fid}} & \iff \\
 & \left(\begin{array}{l} \exists \mathbf{s}{:}\mathrm{S}, \mathbf{d''}{:}\mathrm{D''} . \\ (\mathbf{cd} \equiv \mathbf{s}_{[\![\mathtt{id}]\!]_s\,[\![\mathtt{tp}]\!]_s}^{[\![\mathtt{irn}]\!]_s}[\mathbf{d''}]_{[\![\mathtt{fid}]\!]_s}) \\ \wedge e,s,\mathbf{FT},\mathbf{s} \models_{\mathrm{S}} P \\ \wedge e,s,\mathbf{FT},\mathbf{d''} \models_{\mathrm{D''}} P'' \end{array} \right) \vee \\
 & \left(\begin{array}{l} \exists \mathbf{s}{:}\mathrm{S}, \mathbf{c}{:}\mathrm{D_1 \to D''} . \\ (\mathbf{cd} \equiv \mathbf{s}_{[\![\mathtt{id}]\!]_s\,[\![\mathtt{tp}]\!]_s}^{[\![\mathtt{irn}]\!]_s}[\mathbf{c}]_{[\![\mathtt{fid}]\!]_s}) \\ \wedge e,s,\mathbf{FT},\mathbf{s} \models_{\mathrm{S}} P \\ \wedge e,s,\mathbf{FT},\mathbf{c} \models_{\mathrm{D_1 \to D''}} P'' \end{array} \right)
\end{array}
$$

$$
\begin{array}{ll}
e,s,\mathbf{FT},\mathbf{d} \models_{\mathrm{D}} \text{“\#text”}_{\mathtt{id}\,3}^{\mathtt{idref}}[P]_{\mathtt{fid}} & \iff \exists \mathbf{s}{:}\mathrm{S} . (\mathbf{d} \equiv \text{“\#text”}_{[\![\mathtt{id}]\!]_s\,3}^{[\![\mathtt{idref}]\!]_s}[\mathbf{s}]_{[\![\mathtt{fid}]\!]_s} \wedge e,s,\mathbf{FT},\mathbf{s} \models_{\mathrm{S}} P) \\
e,s,\mathbf{FT},\mathbf{cd} \models_{\mathrm{D_1 \to D_2}} \text{“\#text”}_{[\![\mathtt{id}]\!]_s\,3}^{[\![\mathtt{idref}]\!]_s}[P]_{[\![\mathtt{fid}]\!]_s} & \iff \exists \mathbf{c}{:}\mathrm{D_1 \to S} . (\mathbf{cd} \equiv \text{“\#text”}_{[\![\mathtt{id}]\!]_s}^{[\![\mathtt{idref}]\!]_s\,3}[\mathbf{c}]_{[\![\mathtt{fid}]\!]_s} \wedge \\
 & \quad e,s,\mathbf{FT},\mathbf{c} \models_{\mathrm{D_1 \to S}} P)
\end{array}
$$

$$
\begin{array}{ll}
e,s,\mathbf{FT},\mathbf{c} \models_{\mathrm{C}} \mathbf{c}' & \iff \mathbf{c} = \mathbf{c}' \\
e,s,\mathbf{FT},\mathbf{d} \models_{\mathrm{D}} P' \otimes_{\mathrm{D}} P'' & \iff \exists \mathbf{d}'{:}\mathrm{D}, \mathbf{d}''{:}\mathrm{D} . (\mathbf{d} \equiv \mathbf{d}' \otimes_{\mathrm{D}} \mathbf{d}'') \wedge e,s,\mathbf{FT},\mathbf{d}' \models_{\mathrm{D}} P' \wedge e,s,\mathbf{FT},\mathbf{d}'' \models_{\mathrm{D}} P'' \\
e,s,\mathbf{FT},\mathbf{cd} \models_{\mathrm{D_1 \to D_2}} P' \otimes_{\mathrm{D}_2} P'' & \iff \exists \mathbf{cd}'{:}(\mathrm{D}_1 \to \mathrm{D}_2), \mathbf{d}{:}\mathrm{D}_2 . \\
 & \quad ((\mathbf{cd} \equiv \mathbf{cd}' \otimes_{\mathrm{D}_2} \mathbf{d}) \wedge e,s,\mathbf{FT},\mathbf{cd}' \models_{\mathrm{D_1 \to D_2}} P' \wedge e,s,\mathbf{FT},\mathbf{d} \models_{\mathrm{D}_2} P'') \vee \\
 & \quad ((\mathbf{cd} \equiv \mathbf{d} \otimes_{\mathrm{D}_2} \mathbf{cd}') \wedge e,s,\mathbf{FT},\mathbf{d} \models_{\mathrm{D}_2} P' \wedge e,s,\mathbf{FT},\mathbf{cd}' \models_{\mathrm{D_1 \to D_2}} P'') \\
e,s,\mathbf{FT},\mathbf{d} \models_{\mathrm{D}} P' \oplus_{\mathrm{D}} P'' & \iff \exists \mathbf{d}'{:}\mathrm{D}, \mathbf{d}''{:}\mathrm{D} . (\mathbf{d} \equiv \mathbf{d}' \oplus_{\mathrm{D}} \mathbf{d}'') \wedge e,s,\mathbf{FT},\mathbf{d}' \models_{\mathrm{D}} P' \wedge e,s,\mathbf{FT},\mathbf{d}'' \models_{\mathrm{D}} P'' \\
e,s,\mathbf{FT},\mathbf{cd} \models_{\mathrm{D_1 \to D_2}} P' \oplus_{\mathrm{D}} P'' & \iff \exists \mathbf{cd}'{:}(\mathrm{D}_1 \to \mathrm{D}_2), \mathbf{d}{:}\mathrm{D}_2 . \\
 & \quad (\mathbf{cd} \equiv \mathbf{cd}' \oplus_{\mathrm{D}} \mathbf{d}) \wedge e,s,\mathbf{FT},\mathbf{cd}' \models_{\mathrm{D_1 \to D_2}} P' \wedge e,s,\mathbf{FT},\mathbf{d} \models_{\mathrm{D}_2} P'' \\
e,s,\mathbf{FT},\mathbf{d} \models_{\mathrm{D}} \varnothing_{\mathrm{D}} & \iff \mathbf{d} \equiv \varnothing_{\mathrm{D}}
\end{array}
$$

$$
\begin{array}{ll}
e,s,\mathbf{FT},\mathbf{d} \models_{\mathrm{D}} <P>_{\mathrm{D}} & \iff \exists \mathbf{d}'{:}\mathrm{D}' . (\mathbf{d} \equiv <\mathbf{d}'>_{\mathrm{D}}) \wedge e,s,\mathbf{FT},\mathbf{d}' \models_{\mathrm{D}'} P \\
e,s,\mathbf{FT},\mathbf{cd} \models_{\mathrm{D_1 \to D_2}} <P>_{\mathrm{D}_2} & \iff \exists \mathbf{cd}'{:}(\mathrm{D}_1 \to \mathrm{D}'_2) . (\mathbf{cd} \equiv <\mathbf{cd}'>_{\mathrm{D}_2}) \wedge e,s,\mathbf{FT},\mathbf{cd}' \models_{\mathrm{D_1 \to D}'_2} P
\end{array}
$$

$$
\begin{array}{ll}
e,s,\mathbf{FT},\mathbf{d} \models_{\mathrm{D}} \mathtt{var}_{\mathrm{E}} & \iff \mathbf{d} \equiv e(\mathtt{var}_{\mathrm{E}}) \\
e,s,\mathbf{FT},\mathbf{d} \models_{\mathrm{D}} \mathtt{Exp}_{\mathrm{V}} = \mathtt{Exp}'_{\mathrm{V}} & \iff [\![\mathtt{Exp}_{\mathrm{V}}]\!]_s = [\![\mathtt{Exp}'_{\mathrm{V}}]\!]_s \\
e,s,\mathbf{FT},\mathbf{d} \models_{\mathrm{D}} \mathtt{Int} = |\mathtt{f}| & \iff [\![\mathtt{Int}]\!]_s = \mathrm{len}(e(\mathtt{f})) \\
e,s,\mathbf{FT},\mathbf{a} \models_{\mathrm{A}} \exists \mathtt{var}_{\mathrm{E}} . P & \iff \exists \mathbf{d}' . e[\mathtt{var}_{\mathrm{E}} \mapsto \mathbf{d}'], s,\mathbf{FT},\mathbf{a} \models_{\mathrm{A}} P \\
e,s,\mathbf{FT},\mathbf{a} \models_{\mathrm{A}} \exists \mathtt{var}_{\mathrm{V}} . P & \iff \exists \mathbf{v} . e, s[\mathtt{var}_{\mathrm{V}} \mapsto \mathbf{v}],\mathbf{FT},\mathbf{a} \models_{\mathrm{A}} P \\
e,s,\mathbf{FT},\mathbf{d} \models_{\mathrm{D}} \gamma(\mathbf{f}) & \iff \exists \mathbb{C}, \overline{\mathbf{p}}^m . \mathbf{FT}(\mathbf{f}) = (\mathbb{C}, \overline{\mathbf{p}}^m) \\
e,s,\mathbf{FT},\mathbf{d} \models_{\mathrm{D}} \omega(\mathbf{uri}) & \iff \exists \mathbf{data},\mathbf{code} . W(\mathbf{uri}) = (\mathbf{data},\mathbf{code})
\end{array}
$$

where $\mathbf{f}{:}\mathrm{F}$, $\mathrm{F} \in \{\mathrm{DE}, \mathrm{E}, \mathrm{S}\}$

Figure 16: Satisfaction Relation

Perhaps more importantly, the context application connective allows us to define spacial modal operators. These modalities give our assertions important abstractive power, moreso than typical type systems and schema languages. We can say "This code only needs one node of this type, anywhere in the tree - everything is is unneeded and unnoticed". These abstractions, along with the local reasoning, gives our assertion language surprising conciseness, given the level of detail formulae provide.

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

3.3.4 Example

Recall the XML fragment in section 3.2.1. Using these modalities, we can give a formulae that describes more general form, for a database of places. We can give a formulae that describes documents of that form in general.

$$
place \triangleq \text{“place”}\ [\#\text{text} \otimes \text{“postcode”}[\#\text{text}]]
$$

$$
places \triangleq \begin{array}{l} \exists id, fid.\ \text{“\#document”}_{id}^{\mathbf{null}}[\\ \quad \Box_{\otimes} \langle \mathrm{true}_{\mathrm{T}} \rangle_{\mathrm{F}} \implies place \\ ]_{\mathbf{fid}} \end{array}
$$

We can see *places* as schema for places databases, viewing formulae of our logic as a kind of schema language. If we wanted to further refine data that satisfies the above into data that contain "Imperial DoC", we can write another formulae.

$$
placesWithDoC \triangleq \begin{array}{l} \exists id, fid.\ \text{“\#document”}_{id}^{\mathbf{null}}[\\ \quad \Diamond_{\mathrm{T} \to \mathrm{F}}\ \#\text{text}[\text{“Imperial DoC”}]]_{\mathbf{fid}} \end{array}
$$

3.3.5 Hoare Reasoning

Our assertion language is used in the Hoare style program reasoning introduced in the background section. Judgements on program code $\mathbb{C}$ take the form $\{P\}\mathbb{C}\{Q\}$ and the *fault avoiding* partial correctness interpretation of these judgements is formalised as

Definition 10 (Fault avoiding semantic interpretation)**.**

$$
\begin{array}{l}
\vDash \{P\}\mathbb{C}\{Q\} \iff \\
\left(\begin{array}{l} P : \mathrm{G} \wedge Q : \mathrm{G} \wedge \\ \forall e, s, \mathbf{g} \vDash_{\mathrm{G}} P \implies \mathbb{C}, s, \mathbf{g} \not\rightsquigarrow \mathrm{fault} \wedge \\ \forall s', \mathbf{g}'. \mathbb{C}, s, \mathbf{g} \rightsquigarrow s', \mathbf{g}' \implies e, s', \mathbf{g}' \vDash_{\mathrm{G}} Q \end{array} \right) \\
\qquad \bigvee \\
\left(\begin{array}{l} P : \mathrm{T} \wedge Q : \mathrm{T} \wedge \\ \forall e, s, \mathbf{g} \vDash_{\mathrm{G}} \langle P \rangle_{\mathrm{G}} \implies \mathbb{C}, s, \mathbf{g} \not\rightsquigarrow \mathrm{fault} \wedge \\ \forall s', \mathbf{g}'. \mathbb{C}, s, \mathbf{g} \rightsquigarrow s', \mathbf{g}' \implies e, s', \mathbf{g}' \vDash_{\mathrm{G}} \langle Q \rangle_{\mathrm{G}} \end{array} \right)
\end{array}
$$

Note that disjunction, which exists only to handle both typed cases (groves and trees).

3.3.6 Specifications

Each Hoare triple can be viewed as a *specification* for the code it covers, where the precondition specifies all the states it can accept and the postcondition specifies all the states that can result. When reasoning say, a function call, we do not then need the actual code of the function as the specifications of that function - along with reasoning rules we give shortly - allow us to ensure that the call will be fault free.

We use this property to enable *component-wise reasoning*. Instead of having to have the entire web available so we can examine remote code and data in proofs, we rely on sets of specifications. When proving a single component u, the only code we need access to is that of u, appealing to specifications to provide anything else we need.

This is not an artificial concept. When engineering with components provided by a vendor, it is likely one has access to documentation stating the available operations and how they work. If a programmer is using an undocumented component, they (presumably) have some idea of the functionality it provides either from examining it, or via educated guesswork. Specifications capture this "documentation" intuition in a formal fashion.

We first give the format for specifying functions.

Definition 11 (Function Specifications)**.** A *function specification* is a tuple of the form

$$(\{P\}f(p_1, \ldots, p_n)\{Q\}, Mods)$$

P and Q are the pre and post condition of the function f with n parameters p_i. $Mods$ is the *modification set* of the function body code, and is the set of non-local variables it changes. This concept is vital in local reasoning. We use it to determine the parts of a formulae that may be needed to reason about some code.

One can determine $mods(\mathbb{C})$ purely syntactically from $\mathbb{C}$. All variables on the left of an assignment (or DOM command which assigns a value) are considered modified, unless the variable has been declared local in that scope or is a parameter name. Likewise, all function names are considered modified, unless the function has been declared local.

As function specifications give details of function behavior, we define component specifications, which give the details of each components provisions.

Definition 12 (Component specifications)**.** A *component specification* is a tuple of the form

$$(\{P\}U\{Q\}, Mods, D, H)$$

P and Q are the pre and post condition of the global code in component U. $Mods$ is the modification set of that global code. D is a DOM formulae describing the data available at the URI. H is a set of function specifications, one for each function declared in the component.

When proving a single component, we assume the existence of a set of component specifications, written as $\mathcal{W}$. This gives the formal set of all "documentation" that the component author has access to when writing his code. However, in view of this "documentation" interpretation, components should not have to expose details they wish to keep private. As such, we allow each component to be associated with a *public specification*, $\mathcal{S}_P$ and a *private specification*, $\mathcal{S}_R$. Components not being proven have their public specifications entered in $\mathcal{W}$; when proving a component, one uses the private. In our current presentation, the two are identical, except that private specifications may contain assertions of $\gamma(\mathbf{f})$ for names of local functions absent from the public specification.

Our inference rules will view a correctly formed $\mathcal{W}$ as implicit. We define $\mathcal{F} = \cup_i\{H_i \mid H_i \in H, (U, M, D, H) \in \mathcal{W}\}$, the union of all the function specifications.

3.3.7 Inference rules

We adopt the standard rules of Hoare Logic: skip, assignment, disjunction, conditional, while, sequencing, implication and variable elimination. Vital to supporting local reasoning is the *Frame rule*.

$$\frac{\{P\}\mathbb{C}\{Q\}}{\{K \circ_{\mathrm{D}} P\}\mathbb{C}\{K \circ_{\mathrm{D}} Q\}} \quad \text{If } \; free(K) \cap mods(\mathbb{C}) = \emptyset$$

We see how K, the *frame*, can safely be put aside for the duration a proof involving $\mathbb{C}$, as long as no variables modified by the code are free in K. It is a "surgical clean room" style of reasoning; we remove K from the picture, assured that $\mathbb{C}$ will not touch anything it asserts, whilst we operate on the remaining data. We see the need for the $Mods$ entry in each specification here; without it, we could never soundly apply the frame rule.

The inference rules for calling a procedure, fetching a document or running remote code make direct use of the specifications that we find in the W associated with a proof. One can interpret them as specialised forms of hypotheses; whenever we need to prove something about non-local code, we import an assumption from the specification table. Again, this mirrors the informal reasoning of a programmer using a component; he will typically refer to either documentation or interface specifications when using code he has not written.

$$\frac{\forall P, Q, M.\, (\{P\}f\{Q\}, M) \in \mathcal{F} \implies \{P\}\mathbb{C}\{Q\} \wedge Mods(\mathbb{C}) = M}{\{\varnothing_{\mathrm{G}}\}\texttt{function f(p1, ..., pn)}\ \{\mathbb{C}\}\{\gamma(\texttt{f}) \wedge \varnothing_{\mathrm{G}}\}} \quad \textsc{Func-Intro}$$

Notice that *every* specification for a function name found in the web we are considering must satisfy the code encountered. This allows us to have many possible specifications for one function, as long as our implementations satisfy them all.

$$\frac{}{\{\varnothing_{\mathrm{G}}\}\texttt{return E}\{\mathrm{ret} = \texttt{E} \wedge \varnothing_{\mathrm{G}}\}} \quad \textsc{Func-Return}$$

Parameters (recall that function bodies cannot assign to their formal parameters). We also achieve locality with a fresh variable substitution, much like in the operational case, where fresh here refers to "within the proof" using the rule. However, unlike the operational case, we do not have the function body available to us. We determine the needed substitutions from the Mods set of the function; this will never include local names, so we can intersect free names of the specification with this set to determine what names need freshening.

Let $\theta = \{e_1/p_1, \ldots, e_m/p_m\} \cup \{f_1/l_n, \ldots, f_n/l_n\}$ where e_i are the expressions of the call, the p_i are the parameter names, the f_j are fresh variables and l_j are local to the code (found as the set of free names from P and Q that are not present in M).

$$\frac{\exists P, Q, M.(\{P\}f\{Q\}, M) \in \mathcal{F}}{\{\gamma(\texttt{f}) \wedge \theta(P)\}\texttt{v = f(e1, ..., em)}\{\gamma(\texttt{f}) \wedge \theta(Q)[v/\mathrm{ret}]\}} \quad \textsc{Func-Call}$$

The function call rule is free to pick any specification it knows about when proving a call. Fetching a document needs the URI to be in the component set, then simply extends the grove with a new document as described by the specification. We use a distinguished variable `this` for the specification of the data, which is replaced by the real variable denoting the new grove at import time.

$$\frac{\exists P, Q, M, D, F.\, (\{P\}\texttt{uri}\{Q\}, M, D, F) \in \mathcal{W}}{\{\omega(uri)\}\texttt{n = fetchDocument(uri)}\{D[\texttt{n}/\texttt{this}]\}} \quad \textsc{fetchDocument}$$

The `runScript` rule can be viewed much like the function call rule. Local names are freshened away via the mods set.

$$\frac{\exists P, Q, M, D, F.\, (\{P\}\texttt{uri}\{Q\}, M, D, F) \in \mathcal{F}}{\{\omega(\texttt{uri}) \wedge P\}\texttt{runScript(uri)}\{Q\}}\ \textsc{runScript}$$

The sanity of entries in the specification table is thus paramount. The rule Func-Intro ensures that any function whose body we encounter will be checked, but this does not ensure that external dependencies are checked. We will account for this in subsection 4.

3.3.8 Small Axioms

Commands in our language are proven by a set of *Small Axioms*, so called as the conditions on each schema state only what the command needs to execute without fault, oblivious to the rest of the environment. Reasoning using these axioms thus relies on the frame rule to isolate the structure needed to satisfy the precondition of each axiom.

$$\frac{\left\{\begin{array}{c} <\text{“\#document”}_{\texttt{doc}9}^{\mathbf{null}}[\texttt{DE}]_{\texttt{fid}}\mathbf{null}>_{\mathrm{G}} \wedge \\ \texttt{x} = \texttt{y} \wedge \texttt{name} \neq \mathbf{null} \wedge \text{`\#'} \notin \texttt{name} \end{array}\right\}}{\frac{\texttt{x} = \texttt{createElement}(\texttt{doc}, \texttt{name})}{\left\{\begin{array}{c} <\text{“\#document”}_{\texttt{doc}\{\texttt{y/x}\}}^{\mathbf{null}}{}_{9}[\texttt{DE}]_{\texttt{fid}}\mathbf{null}>_{\mathrm{G}} \oplus \\ <\texttt{name}_{\texttt{x}1}^{\texttt{doc}}[\varnothing_{\mathrm{F}}]_{\texttt{fid'}}\mathbf{null}>_{\mathrm{G}} \end{array}\right\}}}$$

The precondition requires that the given doc expression identifies a document, and uses `DE` to capture this element in a logical variable. The given name is a asserted as non-null, and not containing #. The postcondition shows the document structure captured in the precondition is unchanged, but that the grove has been extended with a new element

The logical variable Y avoids the situation created by the statement `x = createElement(x, s)`. This is well typed, but would instantiate the above axiom in an inconsistent way. We capture the previous value of the assignee in a logical variable, which we us substitute into any expressions it may have been used in through the postcondition.

$$\frac{\{\texttt{name}_{\texttt{n}}{}_{\texttt{tp}}^{\texttt{irn}}[\texttt{F}]_{\texttt{fid}}\texttt{val} \wedge \texttt{kids} = \texttt{Y}\}}{\frac{\texttt{kids} = \texttt{getChildNodes}(\texttt{n})}{\{\texttt{name}_{\texttt{n}\{\texttt{Y/kids}\}}{}_{\texttt{tp}}^{\texttt{irn}}[\texttt{F}]_{\texttt{fid}}\texttt{val} \wedge \texttt{kids} = \texttt{fid}\}}}$$

`getChildNodes` is representative of all the `get` commands, asserting the structure under inspection exists, and capturing elements of the structures state for assignment to variables.

$$\frac{\left\{\begin{array}{c} (\varnothing_{\mathrm{D}_1} \multimap (\texttt{gc} \circ_{\mathrm{D}_2} \texttt{s}_{\texttt{parent}}{}_{\mathbf{1}}^{\texttt{idref}}[\texttt{F}]_{\texttt{fid}}\mathbf{null})) \\ \circ_{\mathrm{D}_1} \\ (\texttt{name}'_{\texttt{child}}{}_{\texttt{ettp}}^{\texttt{idref'}}[\texttt{F}']_{\texttt{fid'}}\texttt{val}') \end{array}\right\}}{\frac{\texttt{n} = \texttt{appendChild}(\texttt{parent}, \texttt{child})}{\left\{\begin{array}{c} \texttt{gc} \\ \circ_{\mathrm{D}_2} \\ \texttt{s}_{\texttt{parent}}{}_{\mathbf{1}}^{\texttt{idref}}\left[\ \texttt{F} \otimes <\texttt{name}'_{\texttt{child}}{}_{\texttt{ettp}}^{\texttt{idref'}}[\texttt{F}']_{\texttt{fid'}}\texttt{val}'>_{\mathrm{F}}\ \right]_{\texttt{fid}}\mathbf{null} \\ \text{where } \texttt{ettp} \in \{1, 3\} \end{array}\right\}}}$$

The axiom for `appendChild` is the most complex in our system. Fault freedom requires that `child` not be an ancestor of `parent`, a property we capture with the adjoint connective. We read the precondition as “A satisfying structure can be split into the `child` node, and a context such that if we place empty in the hole left by `child`, we can still extract the parent node”. If `parent` were a descendent of `child`, we will be able to pull out the child node, but the context from which it was removed would not contain the parent node.

$$\frac{\left\{\begin{array}{c}\mathtt{name}_{\mathtt{id}\,\mathtt{tp}}^{\mathtt{docn}}\left[\begin{array}{c}\mathtt{F}_1 \otimes \\ <\mathtt{name}'_{\mathtt{id}'\,\mathtt{tp}'}^{\mathtt{doc}}[\mathtt{F}']_{\mathtt{fid}'}\mathtt{val}'>_{\mathrm{D}} \\ \otimes\, \mathtt{F}_2 \\ |\mathtt{F}_1| = \mathtt{int} \wedge \mathtt{list} = \mathtt{Y}\end{array}\right]_{\mathtt{list}} \mathtt{val} \wedge\end{array}\right\}}{\mathtt{n} = \mathtt{item}(\mathtt{list}, \mathtt{int})}$$

$$\frac{\left\{\begin{array}{c}\mathtt{name}_{\mathtt{id}\,\mathtt{tp}}^{\mathtt{docn}}\left[\begin{array}{c}\mathtt{F}_1 \otimes \\ <\mathtt{name}'_{\mathtt{id}'\,\mathtt{tp}'}^{\mathtt{doc}}[\mathtt{F}']_{\mathtt{fid}'}\mathtt{val}'>_{\mathrm{D}} \\ \otimes\, \mathtt{F}_2 \\ \wedge\, \mathtt{n} = \mathtt{id}'\end{array}\right]_{\mathtt{list}\{\mathtt{Y}/\mathtt{n}\}} \mathtt{val}\end{array}\right\}}{\left\{\begin{array}{c}\mathtt{name}_{\mathtt{id}\,\mathtt{tp}}^{\mathtt{docn}}[\mathtt{F}]_{\mathtt{list}}\mathtt{val} \wedge \\ (|\mathtt{F}| \leq \mathtt{int} \vee \mathtt{int} < 0) \wedge \mathtt{list} = \mathtt{Y}\end{array}\right\}}{\mathtt{n} = \mathtt{item}(\mathtt{list}, \mathtt{int})}$$

$$\left\{\ \mathtt{name}_{\mathtt{id}\,\mathtt{tp}}^{\mathtt{docn}}[\mathtt{F}]_{\mathtt{list}\{\mathtt{Y}/\mathtt{n}\}}\mathtt{val} \wedge \mathtt{n} = \mathbf{null}\ \right\}$$

Notice that we need two small axioms for `item`, as it has two non-faulting executions (one for an in range index, one for an out of range).

These axioms should be unsurprising; they directly guided by the operational behaviour of the commands they model. Each axiom tries to give minimal environmental assertions needed to avoid faulting.

Lemma 13. *The small axioms are sound*

Proof. By direct appeal to the operational semantics □

3.4 Robustess in mashups

We show now how this reasoning system does indeed allow us to write robust mashups, and determine a good definition of "mashup program". We start with a meta-theroetic soundness theorem, showing that our logic does indeed perform as promised.

Theorem 14 (Soundness)**.** *For a single component U and a proof with respect to a set of specifications $\mathcal{W}$ components, the logic is sound up to the correctness of the assertions in $\mathcal{W}$.*

Proof. (Sketch, see [15] for full version). We appeal to Lemma 13, and the soundness of the frame rule. Non-obvious steps include the soundness of recursive procedures, when the bodies themselves may be redefined during the recursion; this is handled via this form of recursion appealing to the specifications table. □

Unsurprisingly, if specifications lies about the data or behaviour of pages, the resultant proof cannot be trusted. Thus this theorem only gives us confidence that the code of individual pages are fault free. To show that entire mashup programs are fault free, we first need to define the exact scope of a mashup, knowing only the URI of the component we have requested.

Initially, we note components that will be imported into a component are always mentioned either directly in the precondition, or indirectly through the specification of a component whose existence is asserted.

Lemma 15. *If the code associated with a page uses either* `fetchDocument(uri)` *or* `runScript(uri)`*, then* ω(`uri`) *(the assertion that* `uri` *exists) appears in the page precondition, or in the specification of a component whose existence is asserted within the precondition.*

Proof. (Sketch) The logic is sound with respect to the fault avoiding semantic interpretation. The small axiom for `fetchDocument` and `runScript` requires that $\omega(\texttt{uri})$ to be proven. No axiom or inference rule introduces assertions of the form ω, hence either the precondition of the code containing the call must contain it, or it must be in the specification of a component that is imported. □

From this, we give an exact definition of a mashup program. We use the *dependencies* of a given component at URI U to mean the set of V such that $\omega(V)$ appears unnegated in the precondition of U.

Definition 16 (Component Dependencies)**.** For a component with specification $(\{P\}U\{Q\}, Mods, D, F) \in \mathcal{W}$, define the *dependencies* of U as

$$depends(u) = \{v \mid P \implies \omega(v) \wedge \text{true}_\text{G}\}$$

We use this definition to find the smallest $\mathcal{W}'$, that contains all the dependencies of U along with the dependencies of the dependencies, ad infinitum until the set is closed. This may extend or contract what was placed in $\mathcal{W}$. Let the set dependency operator D over URIs Us be

$$D(Us) = \bigcup_{u \in Us} depends(u)$$

This operator has a least fixed point for a singleton set u, found by

$$\begin{aligned} D_u^0 &= \{u\} \\ D_u^{n+1} &= D(D_u^n) \\ &\vdots \\ D_u \uparrow &= \bigcup_{n \geq 0} D_u^n \end{aligned}$$

Notice that this operation is undefined if $\mathcal{W}$ does not contain component specifications that are needed. This is correct; we cannot soundly determine the dependencies of a component if some dependency of it it attempts to import code from a URI we know nothing about. If it exists though, we can use it as a basis for defining "mashup programs".

Theorem 17. *For a given set of component specifications $\mathcal{W}$, if $D_u \uparrow$ exists it induces a closed Abstract Web, represented by the elements of the result. If the preconditions of components in $\mathcal{W}$ do not mention unused resource (that is, assertions of the form $\omega(v)$ that can be removed from the specification without affecting the proof), then the result represents the smallest web on which the mashup can function without faults.*

Proof. (Sketch) If we have sufficient specifications in $\mathcal{W}$, then $D_u \uparrow$ exists as the set of components is finite and each iteration at most extends the set. The Abstract Web in the result is closed, as by lemma 15:, if a component uses another component d, $\omega(d)$ must appear in the precondition.

This web is the smallest for fault free functioning. We need no larger by soundness; there is no smaller, as if we were to remove any element of the abstract web, at least one precondition becomes unsatisfiable by the definition of *depends*. □

The intention is that we attempt to prove a component using a smaller web, and use the D to "push out" the size of the web until we have the minimal needed. To allow us use D_u to expand the set of specifications and find a fault free web, we need an lemma for $\mathcal{W}$ stating that we can extend it without rendering previous proofs unsound.

Lemma 18. *If we have proven a component with respect to some $\mathcal{W}$, the proof holds for any $\mathcal{W}' \supseteq \mathcal{W}$, as long as each component added in the extension can be proven w.r.t. $\mathcal{W}'$ and no precondition in $P \in \mathcal{W}$ is such that $P \implies \neg\omega(\mathbf{uri}) \wedge \mathbf{true}$ for some component* **uri** *in the extension.*

The previous results now let us define mashup program, and what it means for a one to be "fault free" from a users perspective.

Corollary 19. *If all components within the induced closed web of a given page exist and can be proven with the same $\mathcal{W}$, the browsing of any one cannot fault. We call such a set a* Reasonable Web*.*

The result in corollary 19 completes the safety proof, guaranteeing that a specification table gives good data about the underlying code. Moreover, the definition of a "Reasonable Web" gives us a strong candidate for a formal definition of mashup program. It shows us that, if a Reasonable Web for a given mashup component exists, it represents the smallest subset of the World Wide Web that defines the whole mashup program.

3.4.1 The revealing nature of specifications

Theorems 14 and 17 codify just two types of information that specifications give us. We are assured fault freedom, and can now give a formal definition of mashup programs that assures we know exactly what components make up our software. More than this though, component specifications provide interesting information about the functioning of the program at an abstract level.

Fault freedom does not, of course, imply the program works as we expect. The extra information that falls directly out of specifications can be very helpful in determining if our software does what we intend, and where it diverges from our expectations. The local nature of the heap descriptions shows us exactly the data we are exposing to the world; the shape of the document we are promising to provide. We can easily see if we are exposing a more concrete structure than we intended, or giving away implementation details that we may want to change. We give these statements here informally, though could construct further proven results. In building examples, we have found it surprising that the specifications give such detail to "at a glance" examinations that is otherwise not obvious.

4 Automation

We turn now to techniques for automating the program verification techniques provided by the Reasonable Web system. Given an arbitrary RW[6] program $\mathbb{C}$ with pre and post conditions P and Q, we seek to answer "Is $\{P\}\mathbb{C}\{Q\}$ true?", whilst minimizing the work the questioner must do. The ideal system would take the specified program, and instantly return "Yes", or "No, and you're wrong because...".

As discussed, being able to automate program reasoning is essential to its wider adoption in engineering. More interesting for our work, automating the reasoning greatly increases the speed with which we test our ideas, exploring the space of programs our techniques can handle without the tedious hand proofs. We do not currently seek to automatically find specifications for code, expecting the programmers to provide annotations on (at least) functions and the main body of the program.

[6] Reasonable Web, the language introduced earlier

The general validty question is undecidable. Even removing explicit quantification from , the paucity of automation techniques for Context Logic suggests starting with a simpler programming and assertion language. Our research interest lies in the structural connectives, rather than more general automation questions, so our experiments thus far have focused on a cut-down version of the Reasonable Web system, effectively reducing the language back to the DOM API.

The single holed logic was used to present the Reasonable Web, for thematic consistency with Featherweight DOM, and because the underlying formalisms of other choices were not fully developed. However, this form of the logic does not lend itself naturally to automation. Using SHCL, we recall that that the axiom for `appendChild` uses the separation adjoint and so implicitly introduces $\forall$ type requirements - even if we remove explicit universal quantification, it occurs in these structural connectives. The logic also allows structural separation as a connective at any point in a formulae. This property regularly used in hand proofs, so we must be able to handle formulae that assert splits described by sub-formulae that themselves also discuss structural splitting.

Whilst we may be able to overcome these issues, particularly in light of the decidability results of Dinsdale-Young, this situation not the "simple" start we seek. Fortunately, the Segment Logic suffers neither of these issues. If we re-express the DOM formalism in terms of it, we find adjoints are unnecessary for the small axioms. Spacial separation is restricted to just the fragment formulae level, and overall provides a much more uniform distinction between structural assertions and data structure assertions. This will prove advantageous, especially in terms of generalisation.

We follow the conceptual footsteps of Smallfoot for this initial project, using similar techniques of simplifying the logic to a *symbolic heap* and relying on symbolic execution to reduce the Hoare triple problem to a logical entailment. This requires four main stages of work. Initially, we respecify DOM in terms of the Segment logic (a task made fairly easy by Wheelhouse's work). With this, we explore one possible reduction of the complete logic to a semantic domain large enough to prove useful programs, but small enough to render a tractable automatic theory - the so called Symbolic Heap. For this heap, we will develop an algorithmic proof theory for entailment between heaps, and explain how to solve the frame problem needed to allow local reasoning. Finally, this comes together with the use of symbolic execution techniques coupled with *symbolic small axioms* to produce a system that can validate $\vdash \{P\}C\{Q\}$.

This research is very much preliminary, with a focus so far on building a primitive system that is guided by the kinds of programs we'd like to consider. The ethos is *start very small* - if there are two choices, take the simpler. We aim for a sound system, and one that is *modular*; ideally, four areas identified above will be handled in a stable and disjoint manner, so we can improve our techniques in any one of them without having to begin again in the others. Many questions remain open regarding this work, and it will take a significant amount of time to explore.

4.1 Segment DOM

Segment DOM is a Segment logic translation of the Featherweight DOM paper; it can also be seen as the Reasonable Web language without the component fetch commands functions or while loops. These restrictions ensure we can focus on the algorithmic issues around deciding Hoare triple truth.

The novel elements of this logic are due to the "Move" language, developed by Wheelhouse and Gardner, and the origin of the Segment logic. Indeed, both Wright and Wheelhouse formalised DOM independently based on that work and came to virtually identical conclusions; the translation is largely mechanical.

```
C ::=   appendChild(parent, newChild) | removeChild(parent, oldChild)
      | name := getNodeName(node) | id := getParentNode(node)
```

```
| fid := getChildNodes(node) | node := createNode(Name)
| node := item(list, Int) | str := substringData(node, Offset, Count)
| appendData(node, Arg) | deleteData(node, Offset, Count)
| node := createTextNode(Str)
| id := Id | str := Str | int := Int | bool := Bool
| C ; C | if Bool then C else C | skip | end
```

We have just the primitive value types needed to handle DOM in the language: Identifiers $\mathbb{I}$ (including the distinguished **null**), integers $\mathbb{Z}$ and strings $\mathbb{S}$. As in Featherweight DOM, the store is a total mapping from variable names to these values, and expressions are evaluated against such a store in the obvious way. Expressions are simple equality/inequality, literals or integer addition. The aim is the minimal expression language needed to write non-trivial programs, allowing us to focus on the decidability of structural concerns rather than expression operations. We need + to handle the numerically indexed `item` command.

$$\begin{array}{rcl} \texttt{Id} & ::= & \mathbf{null} \mid \texttt{id} \\ \texttt{Str} & ::= & \varnothing_{\mathrm{S}} \mid \mathbf{c} \mid \texttt{str} \mid \texttt{Str} \cdot \texttt{Str} \\ \texttt{Int} & ::= & \mathbf{n} \mid \texttt{int} \mid \texttt{Int} + \mathbf{n} \mid \texttt{Int} - \mathbf{n} \\ \texttt{Bool} & ::= & \texttt{Id} = \texttt{Id} \mid \texttt{Str} = \texttt{Str} \mid \texttt{Int} = \texttt{Int} \\ & & \mid \texttt{Id} \neq \texttt{Id} \mid \texttt{Str} \neq \texttt{Str} \mid \texttt{Int} \neq \texttt{Int} \end{array}$$

The heap data structure is a more radical departure from Reasonable Web. The operational model of the RW language was concrete trees; context notation was a syntactic convenience for updating them, but holes could never in the actual heaps of a program. In Segment DOM, context holes and a restriction operation are part of the data data structure.

$$\begin{array}{rcl} T & ::= & \mathbf{x}_{\mathrm{T}} \mid S_{\mathbf{id}}[F]_{\mathbf{fid}} \mid \#\mathrm{text}_{\mathbf{id}}[S] \\ F & ::= & \varnothing_{\mathrm{F}} \mid \mathbf{x}_{\mathrm{F}} \mid T \mid F \otimes F \\ G & ::= & \varnothing_{\mathrm{G}} \mid \mathbf{x}_{\mathrm{G}} \mid T \mid G \oplus G \\ S & ::= & \varnothing_{\mathrm{S}} \mid \mathbf{c} \mid S \cdot S \end{array}$$

[7]

The remainder of the structure is identical to Featherweight DOM. Holes of type X $\in \{\mathrm{T},\ \mathrm{F},\ \mathrm{G}\}$, $\mathbf{x}_{\mathrm{X}}$ within a structure C may be filled with data of the correct type X via an application function.

$$\begin{array}{rcl} \mathrm{ap}_{\mathbf{x}_{\mathrm{T}}}(C, T) & = & C[T/\mathbf{x}_{\mathrm{T}}] \text{ if } \mathbf{x}_{\mathrm{T}} \in fn(C) \text{ and } fn(C) \cap fn(T) \subseteq \{\mathbf{x}_{\mathrm{T}}\} \\ \mathrm{ap}_{\mathbf{x}_{\mathrm{F}}}(C, F) & = & C[F/\mathbf{x}_{\mathrm{F}}] \text{ if } \mathbf{x}_{\mathrm{F}} \in fn(C) \text{ and } fn(C) \cap fn(F) \subseteq \{\mathbf{x}_{\mathrm{F}}\} \\ \mathrm{ap}_{\mathbf{x}_{\mathrm{G}}}(C, G) & = & C[G/\mathbf{x}_{\mathrm{G}}] \text{ if } \mathbf{x}_{\mathrm{G}} \in fn(C) \text{ and } fn(C) \cap fn(G) \subseteq \{\mathbf{x}_{\mathrm{G}}\} \end{array}$$

We have the same identity and unit properties as the RW language. As with the tree formulae in our segment logic introduction (chapter 2), these DOM models are specific to our modelled domain, and give no structural separation. For this, we use *fragments*, DOM model elements each with a root context label, each representing one spacial element. The unusual element is the ν

[7] We do not include a $\varnothing_{\mathrm{T}}$, as we do not yet fully understand the ramifications of its inclusion on the typing structure

restriction operation; $\nu(\mathbf{x})(f)$ states that the context label $\mathbf{x}$ is bound within the fragment f, and cannot be seen outside it. ν binds the variable under it: all other hole variables are free within the structure.

$$\begin{aligned} f \quad ::= \quad & \varnothing_{\mathrm{SF}} \mid \mathbf{x}_{\mathrm{T}} \leftarrow T \mid \mathbf{x}_{\mathrm{F}} \leftarrow F \mid \mathbf{x}_{\mathrm{G}} \leftarrow G \\ & \mid (\nu \mathbf{x}_{\mathrm{T}})(f) \mid (\nu \mathbf{x}_{\mathrm{F}})(f) \mid (\nu \mathbf{x}_{\mathrm{G}})(f) \mid f + f \end{aligned}$$

These fragments allow us describe the same data in many ways. We are free to model a complex DOM as a single, hole free, fragment. We can also break it down into as many + separated fragments as we desired. The context labels act as both a handle to a fragment and if appearing under restriction, as a memory of how that labelled fragment fits into another. We use a congruence on fragments, stating that + is associative and commutative with identity $\varnothing_{\mathrm{SF}}$. Most importantly, it says introduction of splits under restriction does not change the data in a significant way. It is a purely syntactic operation, or a different "view" of the same data. We can introduce and eliminate splits freely, and extrude/rename them freely (as long as they do not clash with other free names). For example

$$\begin{aligned} & \nu \mathbf{a}(\mathbf{a}_{\mathrm{T}} \leftarrow \mathrm{foo}_{\mathtt{id}}[\mathbf{b}_{\mathrm{F}}]_{\mathtt{fid}} + \mathbf{b}_{\mathrm{F}} \leftarrow \mathrm{bar}_{\mathtt{id}'}[\varnothing]_{\mathtt{fid}'}) \\ \equiv \quad & \mathrm{ap}_{\mathbf{b}_{\mathrm{F}}}(\mathbf{a}_{\mathrm{T}} \leftarrow \mathrm{foo}_{\mathtt{id}}[\mathbf{b}_{\mathrm{F}}]_{\mathtt{fid}}, \mathrm{bar}_{\mathtt{id}'}[\varnothing]_{\mathtt{fid}'}) \\ \equiv \quad & \mathbf{a}_{\mathrm{T}} \leftarrow \mathrm{foo}_{\mathtt{id}}[\mathrm{bar}_{\mathtt{id}'}[\varnothing]_{\mathtt{fid}'}]_{\mathtt{fid}} \end{aligned}$$

There is some debate over the use of this form of model. Rather than giving an obvious "one to one" model for DOM trees, the inclusion of restriction makes our data structure act more of a language for describing trees. Whereas models of the separation logic heap (or the SHCL tree structure) can typically be interpreted onto some concrete data fairly directly, there are many possible segment models for each concrete tree. We shall discuss this in section 5.1.

4.1.1 Operational semantics

The behavior of commands is identical to the DOM and imperative language rules of the Reasonable Web. Writing the evaluation of an expression against store s as $[\![E]\!]_s$, we see the real changes are simply notational, re-expressing them using the new contextual notation, e.g.

$$\textsc{createNode} \quad \frac{[\![\mathbf{s}]\!]_s = \mathbf{s} \qquad \mathbf{f} \equiv (\nu \mathbf{y})(\mathbf{x}_{\mathrm{G}} \leftarrow \mathbf{y} + \mathbf{f}'') \qquad \mathbf{f}' \equiv (\nu \mathbf{y})(\mathbf{x}_{\mathrm{G}} \leftarrow \mathbf{y} \oplus \mathbf{s}_{\mathtt{id}}[\varnothing_{\mathrm{F}}]_{\mathtt{fid}} + \mathbf{f}'') \qquad \mathtt{id}, \mathtt{fid} \text{ fresh}}{s, \mathbf{f}, \mathtt{n := createNode(s)} \rightsquigarrow s[\mathtt{n} \mapsto \mathtt{id}], \mathbf{f}'}$$

$$\textsc{appendChild} \quad \frac{\begin{array}{c}[\![\mathtt{p}]\!]_s = \mathbf{p} \qquad [\![\mathtt{c}]\!]_s = \mathbf{c} \qquad \mathbf{f} \equiv (\nu \mathbf{w}, \mathbf{x}, \mathbf{y}, \mathbf{z})(\mathbf{f}'' + \mathbf{w}_{\mathrm{T}} \leftarrow \mathbf{s}_{\mathbf{p}}[\mathbf{x}_{\mathrm{F}}]_{\mathtt{fid}} + \mathbf{y}_{\mathrm{T}} \leftarrow \mathbf{s}'_{\mathbf{c}}[\mathbf{z}_{\mathrm{F}}]_{\mathtt{fid}'}) \\ \mathbf{f}' \equiv (\nu \mathbf{w}, \mathbf{x}, \mathbf{y}, \mathbf{z})(\mathbf{f}'' + \mathbf{w}_{\mathrm{T}} \leftarrow \mathbf{s}_{\mathbf{p}}[\mathbf{x}_{\mathrm{F}} \otimes \mathbf{s}'_{\mathbf{c}}[\mathbf{z}_{\mathrm{F}}]_{\mathtt{fid}'}]_{\mathtt{fid}} + \mathbf{y}_{\mathrm{T}} \leftarrow \varnothing_{\mathrm{T}})\end{array}}{s, \mathbf{f}, \mathtt{appendChild(p, c)} \rightsquigarrow s, \mathbf{f}'}$$

$$\textsc{getNodeName} \quad \frac{[\![\mathtt{n}]\!]_s = \mathbf{n} \qquad \mathbf{f} \equiv (\nu \mathbf{x}, \mathbf{y})(\mathbf{f}' + \mathbf{x} \leftarrow s_{\mathbf{n}}[\mathbf{y}]_{\mathtt{fid}})}{s, \mathbf{f}, \mathtt{x := getNodeName(n)} \rightsquigarrow s[\mathtt{x} \mapsto \mathbf{s}], \mathbf{f}}$$

4.1.2 Logic

We have adopted the key concepts of the Segment logic, introduced in section 2.1.4, unchanged. We still have segment separation between fragments, $*$, as well as the revelation operator $\circledR$ and fresh naming. We also introduce the adjoints of both $*$ and $\circledR$, useful for weakest preconditions.

The formulae appear in two classes: Segment formulae which form the structural assertion language, and DOM formulae which discuss our specific data structure.

$$
\begin{array}{rrcl}
\textbf{DOM Formulae} & P_{\mathrm{D}} & ::= & \\
\textit{Implication} & & | & P_{\mathrm{D}} \implies P_{\mathrm{D}} \\
\textit{Falsity} & & | & \textbf{false}_{\mathrm{D}} \\
\textit{Quantification} & & | & \exists v.P_{\mathrm{D}} \mid \exists lv.P_{\mathrm{D}} \\
\textit{Specific} & & | & P_{\mathrm{T}} \mid P_{\mathrm{F}} \mid P_{\mathrm{G}} \mid P_{\mathrm{S}} \\
\textit{Tree} & P_{\mathrm{T}} & ::= & \varnothing_{\mathrm{T}} \mid P_{\mathrm{S}\,\mathtt{id}}[P_{\mathrm{F}}]_{\mathtt{fid}} \mid \#\mathrm{text}_{\mathtt{id}}[P_{\mathrm{S}}] \mid \alpha_{\mathrm{T}} \mid @\alpha_{\mathrm{T}} \\
\textit{Forest} & P_{\mathrm{F}} & ::= & \varnothing_{\mathrm{F}} \mid P_{\mathrm{T}} \mid P_{\mathrm{F}} \otimes P_{\mathrm{F}} \mid \alpha_{\mathrm{F}} \mid @\alpha_{\mathrm{F}} \\
\textit{Grove} & P_{\mathrm{G}} & ::= & \varnothing_{\mathrm{G}} \mid P_{\mathrm{T}} \mid P_{\mathrm{G}} \oplus P_{\mathrm{G}} \mid \alpha_{\mathrm{G}} \mid @\alpha_{\mathrm{G}} \\
\textit{String} & P_{\mathrm{S}} & ::= & \varnothing_{\mathrm{S}} \mid \mathbf{c} \mid P_{\mathrm{S}} \cdot P_{\mathrm{S}}
\end{array}
$$

The standard connectives and quantifiers are included; v are the normal program variables (identifiers, integers, strings, and booleans). lv are the logical variables, which include DOM fragments. Within the specific formulae, those describing trees, forests and groves should be familiar from the Reasonable Web work. The assertions α_{X} state that a hole exists at that point in the data structure: the α are logical variables standing for hole labels. We also have $@\alpha_{\mathrm{X}}$, which asserts that there is a hole label somewhere within the free names of the data structure: effectively, that there exists an unbound hole name.

The entries for contexts and holes form the "memory" structure we must include in our specific formulae to support local reasoning with segments. Notice we are not obliged to include holes for everything; i.e. for simplicity, we have not introduced context holes for strings. This will mean we cannot assert local properties of strings, and any reasoning using them will have to assert the existence of the surrounding tree structure.

The segment formulae give us the local reasoning power over these DOM specific assertions. Let $\mathrm{X} \in \{T, F, G\}$.

$$
\begin{array}{rrcl}
\textbf{Segment Formulae} & P_{\mathrm{SF}} & ::= & \\
\textit{Implication} & & | & P_{\mathrm{SF}} \implies P_{\mathrm{SF}} \\
\textit{Falsity} & & | & \textbf{false}_{\mathrm{SF}} \\
\textit{Separation} & & | & P_{\mathrm{SF}} * P_{\mathrm{SF}} \mid P_{\mathrm{SF}} \mathrel{-\!\!*} P_{\mathrm{SF}} \\
\textit{Revelation} & & | & \alpha_{\mathrm{X}} \circledR P_{\mathrm{SF}} \mid \alpha_{\mathrm{X}} \mathrel{-\!\!\circledR} P_{\mathrm{SF}} \\
\textit{Empty} & & | & \varnothing_{\mathrm{SF}} \\
\textit{DOM Formulae} & & | & \alpha_{\mathrm{X}} \leftarrow P_{\mathrm{X}} \\
\textit{Quantification} & & | & \exists v.P_{\mathrm{SF}} \mid \exists lv.P_{\mathrm{S}} \mid \text{И}\, \alpha_{\mathrm{X}}.P_{\mathrm{SF}}
\end{array}
$$

Satisfaction for these formulae is given in Figure 17 . We derive the remainder of FOL in the standard way, as well as the **tree** assertion.

$$\textbf{tree}(P_{\mathrm{T}}) \triangleq P_{\mathrm{T}} \land \neg\exists\alpha_{\mathrm{T}}.@\alpha_{\mathrm{T}}$$

The **tree** assertion states that the data in the model at this point is *complete*; we are not talking about a partial data structure, but rather a total DOM tree. This will be used within our `appendChild` reasoning to ensure we are not creating circular structures.

A combination of revelation and fresh naming allows us to derive a *hiding quantifier*, similar to that of Cardelli's Ambient Logic. The primary use of, and a good intuition for, the hiding of

$$
\begin{array}{ll}
e, s, D \vDash_{\mathrm{D}} P_{\mathrm{D}} \implies P'_{\mathrm{D}} & \iff e, s, d \vDash_{\mathrm{D}} P_{\mathrm{D}} \implies e, s, d \vDash_{\mathrm{D}} P'_{\mathrm{D}} \\
e, s, D \vDash_{\mathrm{D}} \mathbf{false}_{\mathrm{D}} & \iff \text{Never} \\
e, s, D \vDash_{\mathrm{D}} \exists v. P_{\mathrm{D}} & \iff \exists w. e, s[v \mapsto w], d \vDash_{\mathrm{D}} P_{\mathrm{D}} \\
e, s, D \vDash_{\mathrm{D}} \exists l v. P_{\mathrm{D}} & \iff \exists w. e[lv \mapsto w], s, d \vDash_{\mathrm{D}} P_{\mathrm{D}} \\
 & \\
e, s, X \vDash_{\mathrm{X}} \varnothing_{\mathrm{X}} & \iff X \equiv \varnothing_{\mathrm{X}} \\
e, s, X \vDash_{\mathrm{X}} \alpha_{\mathrm{X}} & \iff X \equiv X = e(\alpha_{\mathrm{X}}) \\
e, s, X \vDash_{\mathrm{X}} @\alpha_{\mathrm{X}} & \iff e(\alpha_{\mathrm{X}}) \in \mathrm{free}(X) \\
 & \\
e, s, T \vDash_{\mathrm{T}} P_{\mathrm{S}\,\mathtt{id}}[P_{\mathrm{F}}]_{\mathtt{fid}} & \iff \exists S, F.\ T \equiv S_{s(\mathtt{id})}[F]_{s(\mathtt{fid})} \wedge e, s, F \vDash_{\mathrm{F}} P_{\mathrm{F}} \wedge e, s, S \vDash_{\mathrm{S}} P_{\mathrm{S}} \\
e, s, T \vDash_{\mathrm{T}} \#\mathrm{text}_{\mathtt{id}}[P_{\mathrm{S}}] & \iff \exists S.\ T \equiv \#\mathrm{text}_{s(\mathtt{id})}[S] \wedge e, s, S \vDash_{\mathrm{S}} P_{\mathrm{S}} \\
 & \\
e, s, F \vDash_{\mathrm{F}} P_{\mathrm{T}} & \iff e, s, T \vDash_{\mathrm{T}} P_{\mathrm{T}} \\
e, s, F \vDash_{\mathrm{F}} P_{\mathrm{F}} \otimes P'_{\mathrm{F}} & \iff \exists F_1, F_2.\ F \equiv F_1 \otimes F_2 \wedge e, s, F_1 \vDash_{\mathrm{F}} P_{\mathrm{F}} \wedge e, s, F_2 \vDash_{\mathrm{F}} P'_{\mathrm{F}} \\
 & \\
e, s, G \vDash_{\mathrm{G}} P_{\mathrm{T}} & \iff e, s, T \vDash_{\mathrm{T}} P_{\mathrm{T}} \\
e, s, G \vDash_{\mathrm{G}} P_{\mathrm{G}} \oplus P'_{\mathrm{G}} & \iff \exists G_1, G_2.\ G \equiv G_1 \otimes G_2 \wedge e, s, G_1 \vDash_{\mathrm{G}} P_{\mathrm{G}} \wedge e, s, G_2 \vDash_{\mathrm{G}} P'_{\mathrm{G}} \\
 & \\
e, s, S \vDash_{\mathrm{S}} \mathbf{c} & \iff S = \mathbf{c} \\
e, s, S \vDash_{\mathrm{S}} P_{\mathrm{S}} \cdot P'_{\mathrm{S}} & \iff \exists S_1, S_2.\ D \equiv S_1 \cdot S_2 \wedge e, s, S_1 \vDash_{\mathrm{S}} P_{\mathrm{S}} \wedge e, s, S_2 \vDash_{\mathrm{S}} P'_{\mathrm{S}} \\
 & \\
e, s, f \vDash_{\mathrm{SF}} P_{\mathrm{SF}} \implies P'_{\mathrm{SF}} & \iff e, s, f \vDash_{\mathrm{SF}} P_{\mathrm{SF}} \implies e, s, f \vDash_{\mathrm{SF}} P'_{\mathrm{S}} \\
e, s, f \vDash_{\mathrm{SF}} \mathbf{false}_{\mathrm{SF}} & \iff \text{Never} \\
e, s, f \vDash_{\mathrm{SF}} P_{\mathrm{S}} * P'_{\mathrm{SF}} & \iff \exists f_1, f_2. f \equiv f_1 + f_2 \wedge e, s, f_1 \vDash_{\mathrm{SF}} P_{\mathrm{SF}} \wedge e, s, f_2 \vDash_{\mathrm{SF}} P_{\mathrm{SF}} \\
e, s, f \vDash_{\mathrm{SF}} P_{\mathrm{SF}} \mathrel{-\!\!*} P'_{\mathrm{SF}} & \iff \forall f'. e, s, f' \vDash_{\mathrm{SF}} P_{\mathrm{SF}} \wedge (f + f') \text{ defined } \wedge e, s, f + f' \vDash_{\mathrm{SF}} P'_{\mathrm{SF}} \\
e, s, f \vDash_{\mathrm{SF}} \alpha_{\mathrm{X}} \circledR P_{\mathrm{SF}} & \iff \exists \mathbf{x}_{\mathrm{X}}, f'.\ e(\alpha_{\mathrm{X}}) = \mathbf{x}_{\mathrm{X}} \wedge f \equiv (\nu \mathbf{x}_{\mathrm{X}})(f') \wedge e, s, f' \vDash_{\mathrm{SF}} P_{\mathrm{SF}} \\
e, s, f \vDash_{\mathrm{SF}} \alpha_{\mathrm{X}} \mathrel{-\!\!\circledR} P_{\mathrm{SF}} & \iff \exists \mathbf{x}_{\mathrm{X}}, f'.\ e(\alpha_{\mathrm{X}}) = \mathbf{x}_{\mathrm{X}} \wedge f' \equiv (\nu \mathbf{x}_{\mathrm{X}})(f) \wedge e, s, f' \vDash_{\mathrm{SF}} P_{\mathrm{S}} \\
e, s, f \vDash_{\mathrm{SF}} \varnothing_{\mathrm{SF}} & \iff f \equiv \varnothing_{\mathrm{SF}} \\
e, s, f \vDash_{\mathrm{SF}} \alpha_{\mathrm{X}} \leftarrow P_{\mathrm{X}} & \iff \exists X, x. e(\alpha) = \mathbf{x}_{\mathrm{X}} \wedge f \equiv \mathbf{x}_{\mathrm{X}} \leftarrow X \wedge e, s, X \vDash_{\mathrm{D}} P_{\mathrm{X}} \\
e, s, f \vDash_{\mathrm{SF}} \exists v. P_{\mathrm{S}} & \iff \exists w. e, s[v \mapsto w], f \vDash_{\mathrm{SF}} P_{\mathrm{SF}} \\
e, s, f \vDash_{\mathrm{SF}} \exists l v. P_{\mathrm{S}} & \iff \exists w. e[lv \mapsto w], s, f \vDash_{\mathrm{SF}} P_{\mathrm{SF}} \\
e, s, f \vDash_{\mathrm{SF}} \mathcal{N} \alpha. P_{\mathrm{S}} & \iff \exists \mathbf{x}. \mathbf{x} \notin (free(f) \cup dom(e)) \wedge e[\alpha \mapsto \mathbf{x}], s, f, \vDash_{\mathrm{SF}} P_{\mathrm{SF}}
\end{array}
$$

Figure 17: Satisfaction Relation for Segment DOM, where $\mathrm{X} \in \{\mathrm{T},\ \mathrm{F},\ \mathrm{G}\}$

α_{X} is the assertion that a split can be introduced, where α will be used both to remember the original structural location, and as a handle to the split off data.

$$\mathrm{H}\alpha_{\mathrm{X}}.P_{\mathrm{SF}} \triangleq \mathcal{N}\,\alpha_{\mathrm{X}}.\alpha_{\mathrm{X}} \circledR P_{\mathrm{SF}}$$

4.1.3 Small Axioms

The small axioms for Segment DOM demonstrative examples of the power of the logic. A representative sample is

$$\begin{array}{c} \{\ \alpha_{\mathrm{G}} \leftarrow \varnothing_{\mathrm{G}}\ \} \\ \texttt{x := createNode(s)} \\ \{\ \alpha_{\mathrm{G}} \leftarrow \mathtt{s_n}[\varnothing_{\mathrm{F}}]_{\mathtt{fid}}\ \} \end{array} \qquad \begin{array}{c} \{\ \alpha_{\mathrm{T}} \leftarrow \mathtt{s_n}[\alpha_{\mathrm{F}}]_{\mathtt{fid}}\ \} \\ \texttt{nm := getNodeName(n)} \\ \{\ \alpha_{\mathrm{T}} \leftarrow \mathtt{s_n}[\alpha_{\mathrm{F}}]_{\mathtt{fid}} \wedge \mathtt{nm} = \mathtt{s}\ \} \end{array}$$

$$\begin{array}{c} \{\ \alpha_{\mathrm{T}} \leftarrow \mathtt{s_p}[\alpha_{\mathrm{F}}]_{\mathtt{fid}} * \alpha'_{\mathrm{F}} \leftarrow \mathtt{s'_c}[\mathbf{tree}(X)]_{\mathtt{fid}'}\ \} \\ \texttt{appendChild(p, c)} \\ \{\ \alpha_{\mathrm{T}} \leftarrow \mathtt{s_p}[\alpha_{\mathrm{F}} \otimes \mathtt{s'_c}[\mathbf{tree}(X)]_{\mathtt{fid}'}]_{\mathtt{fid}} * \alpha'_{\mathrm{F}} \leftarrow \varnothing_{\mathrm{F}}\ \} \end{array} \qquad \begin{array}{c} \{\ \alpha_{\mathrm{T}} \leftarrow \mathtt{s_n}[\alpha_{\mathrm{F}}]_{\mathtt{fid}} \wedge \mathtt{y} = \mathtt{f}\ \} \\ \texttt{f := getChildNodes(n)} \\ \{\ \alpha_{\mathrm{T}} \leftarrow \mathtt{s}_{\mathtt{n[y/f]}}[\alpha_{\mathrm{F}}]_{\mathtt{fid[y/f]}} \wedge f = \mathtt{fid}\ \} \end{array}$$

Notice the data previous handled by logical variables is now indirectly asserted via context holes. We can use as many holes as needed, and can thus segment any data unneeded by an axiom away. The concrete data asserted by each small axiom is now much closer to the actual footprint of the command. The only slight issue is with `appendChild`, which uses a predicate **tree** asserting that there are no context holes in the model at (and under) this point. We need this to avoid a potentially circular construction, where we could use `appendChild` to build a link from the child to the parent.

4.1.4 Inference rules

As discussed in the background, Segment Logic adds three new inference rules to classical Hoare logic. The two frame rules, Revelation and Separation act together to give local reasoning. Fresh name elimination is a natural analogue to the standard existential elimination rule.

$$\textsc{Rev-Frame}\ \frac{\{P_F\}\ \mathbb{C}\ \{Q_F\}}{\{\alpha \circledR P_F\}\ \mathbb{C}\ \{\alpha \circledR Q\}} \qquad \textsc{Fresh-Elim}\ \frac{\{P\}\ \mathbb{C}\ \{Q\}}{\{\mathcal{N}\,\alpha.P\}\ \mathbb{C}\ \{\mathcal{N}\,\alpha.Q\}}$$

$$\textsc{Segment-Frame}\ \frac{\{P\}\ \mathbb{C}\ \{Q\}}{\{R * P\}\ \mathbb{C}\ \{R * Q\}}\ \text{If } \mathit{mods}(\mathbb{C}) \cap \mathit{free}(R) = \emptyset$$

Separation Frame allows us to construct frames made from $*$ connected segment formulae, and behaves exactly like the Frame rule of O'Hearn's Separation logic. Using it, we can "frame off" unused fragments of the structure. Introducing these structural fragments is the job of Revelation, which states that one can consider the heap to be segmented using a name α as the handle to both remember the point of segmentation, and to refer to the removed data. Revelation frame allows us to frame away ®, the logical demonstration that the structure can be "dismantled" in a given way using the revealed name. Consider a formulae

$$\mathrm{foo}_{\mathtt{id}}[\mathrm{bar}_{\mathtt{id}'}[\varnothing_{\mathrm{F}}]_{\mathtt{fid}}]_{\mathtt{fid}'}$$

This formulae is true if and only if

$$\mathcal{V}\!\alpha_{\mathrm{F}}.\alpha_{\mathrm{F}}\,\text{®}(\beta_{\mathrm{T}} \leftarrow \mathrm{foo}_{\mathtt{id}}[\alpha_{\mathrm{F}}]_{\mathtt{fid}} * \alpha_{\mathrm{F}} \leftarrow \mathrm{bar}_{\mathtt{id}'}[\varnothing_{\mathrm{F}}]_{\mathtt{fid}'})$$

i.e. we can, using a totally new context handle reveal the structure falls apart, using some hole name we capture in α, into the given segmented representation. We can reason locally about the fragment we extracted with α by applying first freshness elimination, then revelation frame, then segmentation frame. This is also a good example of the derived hiding quantifier being used to assert a possible structural splitting, as the above is equivalent to

$$\mathrm{H}\alpha_{\mathrm{F}}.(\beta_{\mathrm{T}} \leftarrow \mathrm{foo}_{\mathtt{id}}[\alpha_{\mathrm{F}}]_{\mathtt{fid}} * \alpha_{\mathrm{F}} \leftarrow \mathrm{bar}_{\mathtt{id}'}[\varnothing_{\mathrm{F}}]_{\mathtt{fid}'})$$

4.2 The symbolic heap

The above complete presentation of Segment DOM is highly expressive, and it would be useful if we could decide entailments between those formulae. However, no research has been done so far on formal decidability results, let alone practical approaches for this complete logic.

Instead, we will try and find a tractable fragment, in the style of Smallfoot work. We approach the question via a pragmatic route: What sort of formulae do we use in proving programs? We call these fragments *symbolic heaps*, and introduce just one attempt here.

These fragments use the same models, but reduce the complexity of the assertion language to capture the kinds of formulae that turn up during hand proofs. The logical fragment is significantly less expressive than the complete version, but via careful choice becomes much easier to automate. The challenge is in finding a sufficiently large subset such that real programs can be analysed, without introducing insurmountable automation problems.

Our first attempt at a symbolic heap follows. We use a special syntax, $\Pi \mid \Sigma$, separation *pure* (DOM heap free) assertions to the left of the pipe, and DOM heap assertions to the right. The mechanisms for handling formulae fragments in each class will be quite different, and we may even be able to "outsource" the structural assertions to another automation system (i.e. Microsoft's Z3 SMT solver).[8].

$$\begin{array}{rcl}
E & ::= & x \mid \mathbf{null} \mid \mathbb{Z} \mid \mathbb{S} \mid E + \mathbb{Z} \mid E - \mathbb{Z} \\
P & ::= & E = E \mid E \neq E \\
\Pi & ::= & \mathbf{true} \mid \Pi \wedge P \\
\\
T & ::= & \mathbf{s}_{\mathbf{id}}[F]_{\mathbf{fid}} \mid \varnothing_{\mathrm{T}} \mid \alpha_{\mathrm{T}} \\
SF & ::= & \mathbf{tree}(T) \mid \mathbf{true} \mid \Box T \mid \Diamond T \\
CF & ::= & T \mid \varnothing_{\mathrm{F}} \mid \alpha_{\mathrm{F}} \\
F & ::= & CF \mid SF \mid CF \otimes F \mid CF \otimes SF \\
G & ::= & T \mid G \oplus G \mid \varnothing_{\mathrm{G}} \mid \alpha_{\mathrm{G}} \\
B & ::= & \alpha_{\mathrm{T}} \leftarrow T \mid \alpha_{\mathrm{F}} \leftarrow F \mid \alpha_{\mathrm{G}} \leftarrow G \\
\Sigma & ::= & B \mid \mathrm{H}\alpha_{\mathrm{D}}.\Sigma \mid \Sigma * \Sigma \\
 & & (\text{where } \mathrm{D} \in \{\mathrm{T},\ \mathrm{F},\ \mathrm{G}\}) \\
\\
\Psi & ::= & \Pi \mid \Sigma
\end{array}$$

[8]This first attempt is guided by our efforts to capture the essence of the schema language, Relax-NG

On these formulae, we define a syntactic equivalence $\equiv$ that respects commutativity of $\wedge$ and $*$, as well as symmetry of $=$ and $\neq$. This is different from logical equivalence, which is part of what we are trying to decide with our approach. We use it only to hide the tedious details within the upcoming manipulation algorithms that would otherwise have to "search" through formulae to find elements of the syntactic structure.

The satisfaction of the pure heap Π is fairly evident;. Satisfaction for T, G and CF are the same as T, G and F from the complete logic. The interesting changes come from SF, F and B, where the restrictions on formulae construction become more evident.

The new formulae classes represent the initial choices for "hard to decide" formulae that nonetheless are needed to make our abstract domain useful. These primarly focus on the issues surrounding structural **true**, the revelation/fresh naming operations, and negation. Consider

$$P \triangleq \texttt{foo}_{\texttt{id}}[\mathbf{true} \otimes \texttt{\#text}_{\texttt{id}'}[\varnothing_{\mathrm{S}}] \otimes \mathbf{true}]_{\texttt{fid}}$$

The true here acts as a kind of "black hole", catching anything, such that the above is satisfied by any data containing a text node with the correct ID. Deciding this type of entailment is likely to be quite hard. This issue, and those involving negation, are likely not insurmountable, but they are not in keeping with the "keep it simple" ethos.

To simplify the case, we notice that in program proofs, **true** is typically used for either saying "I don't care", or to define the modalities. The first typically either appears alone in forests or at the end of a stronger forest formulae, and the second tends to only appear as a singleton forest formulae. Truth is so used as it is often just there "collect garbage" that has accumulated (due to our language having no dispose command). The modalities act as an abstraction concept for us: "Everywhere under here is a Foo node", or "I know I have at least a Bar node", and as such typically appear alone.

As such, restricting the logic so these only appear at the end of forests is not particularly harsh. We can still express the majority of heaps asserted in proofs, but our proof theory no longer has the "black hole" nature of truth to deal with. Whenever we encounter the concept it provides on one side of an entailment, we know exactly the shape of the data on the other side it must apply to.

Similarly, in program proofs, revelation and fresh naming are rarely used other than in combination to assert a new splitting. We can lift the hiding operator into the logic as a primitive, and forget the complex semantics surrounding them, allowing only the assertions of "fresh splits". This has the additional property of ensuring that we can always use the logic in a way that puts the quantifier on the outside of any formulae.

The forms presented have been chosen such that

1. They are enough to write the small axioms for DOM
2. They capture the most useful cases encountered in proofs using the analogous constructs from the complete logic
3. Only things provable from the full logic can be proven (though they cannot prove all things the analogous full version can)
4. They have an easy decision process

How well these definitions meet the goals is not clear; they represent a first pass that will certainly change. The satisfaction relation for the augmented logic is partially demonstrated in 18. It represents a simple direct translation of the concepts expressed in the complete logic to their underlying meaning in the simplified vocabulary.

$$
\begin{array}{ll}
e, s, f \vDash \Diamond T & \iff \exists f_1, f_2, f_3.\ f \equiv f_1 \otimes f_2 \otimes f_2 \wedge e, s, f_2 \vDash T \\
e, s, f \vDash \Box T & \iff \nexists f_1, f_2, f_3.\ f \equiv f_1 \otimes f_2 \otimes f_2 \wedge f_{1,2,3} \neq \varnothing_{\mathrm{T}} \wedge e, s, f_2 \nvDash T \\
e, s, f \vDash \mathrm{H}\alpha_{\mathrm{X}}.\Sigma & \iff \exists \mathbf{x}_{\mathrm{X}}, f'.\ \mathbf{x}_{\mathrm{X}} \notin (dom(e) \cup \mathrm{free}(\Sigma)) \wedge f \equiv (\nu \mathbf{x}_{\mathrm{X}})(f') \wedge \\
 & \qquad e[\alpha_{\mathrm{X}} \mapsto \mathbf{x}_{\mathrm{X}}], s, f' \vDash \Sigma \\
e, s, f \vDash \mathbf{tree}(T) & \iff e, s, f \vDash T \wedge \nexists \mathbf{x}.\ s \in \mathrm{free}(f)
\end{array}
$$

Figure 18: Partial satisfaction relation for Symbolic Heaps

Notice that all the concepts here can be easily discussed using the full power of the logic. Indeed, in proofs using these results, we will often appeal to the stronger definitions.

4.2.1 Symbolic Small Axioms

As the structure of heap formulae has changed, so to must the small axioms. It is a fairly simple rewriting process, encoding the same fault avoiding needs into symbolic heap formulae. Symbolic small axioms also come with a set of *updated pure variables* - the changes they make on the pure heap, as a third | separated element.

$$
\begin{array}{cc}
\left\{\ \Pi \mid \alpha_{\mathrm{G}} \leftarrow \varnothing_{\mathrm{G}}\ \right\} & \left\{\ \Pi \mid \alpha_{\mathrm{T}} \leftarrow \mathtt{s}_{\mathtt{n}}[\alpha_{\mathrm{F}}]_{\mathtt{fid}}\ \right\} \\
\texttt{x := createNode(s)} & \texttt{nm := getNodeName(n)} \\
\left\{\ \Pi \mid \alpha_{\mathrm{G}} \leftarrow \mathtt{s}_{\mathtt{n}}[\varnothing_{\mathrm{F}}]_{\mathtt{fid}}\ \right\} & \left\{\ \Pi \mid_{\mathtt{nm}} \alpha_{\mathrm{T}} \leftarrow \mathtt{s}_{\mathtt{n}}[\alpha_{\mathrm{F}}]_{\mathtt{fid}} \mid nm = \mathtt{s}\ \right\}
\end{array}
$$

$$
\begin{array}{cc}
\left\{\ \Pi \mid \alpha_{\mathrm{T}} \leftarrow \mathtt{s}[\alpha_{\mathrm{F}}]_{\mathtt{fid}} * \alpha'_{\mathrm{T}} \leftarrow \mathtt{s}'[\mathbf{tree}(X)]_{\mathtt{fid}'}\ \right\} & \left\{\ \Pi \mid \alpha_{\mathrm{T}} \leftarrow \mathtt{s}_{\mathtt{n}}[\alpha_{\mathrm{F}}]_{\mathtt{fid}}\ \right\} \\
\texttt{appendChild(p, c)} & \texttt{f := getChildNodes(n)} \\
\left\{\ \Pi \mid \alpha_{\mathrm{T}} \leftarrow \mathtt{s}[\alpha_{\mathrm{F}} \otimes \mathtt{s}'[\mathbf{tree}(X)]_{\mathtt{fid}'}]_{\mathtt{fid}} * \alpha'_{\mathrm{T}} \leftarrow \varnothing_{\mathrm{T}}\ \right\} & \left\{\ \Pi \mid \alpha_{\mathrm{T}} \leftarrow \mathtt{s}_{\mathtt{n}}[\alpha_{\mathrm{F}}]_{\mathtt{fid}} \mid f = \mathtt{fid}\ \right\}
\end{array}
$$

Importantly, the structural formulae of the small axioms have not changed in a significant fashion. The symbolic heap definition has maintained enough power to There are also two classes of axiom: those where pure formulae are updated (i.e. an assignment has occurred), and those where only spacial formulae have been altered.

4.3 Deciding entailment

We give a proof theory to decide the validity of entailment between symbolic heaps, $\Psi_1 \vdash \Psi_2$, i.e. shows that all models of Ψ_1 are also models of Ψ_2. We can consider this as three distinct needs; the entailment of pure assertions, the entailment of DOM formulae that appear as fragments, and the handling of fragmentation.

4.3.1 Pure elements

We have a simple proof theory for our pure entailments, based directly on the Smallfoot rules. Equalities are propagated, and removed when tautological. We include a rule for the excluded middle, and perform arithmetic when possible.

The key concept here is "Simplify, then remove". We use sound rewrites to transform the entailments into equivalent versions where we can more simply see elements we can safely remove. Rules can be applied in any order to the same end. If we ever construct an inconsistency on the left, the entailment is valid. If we ever get to the situation where $\Pi \vdash \mathbf{true}$, then the entailment is valid. If we ever get stuck, the entailment is invalid. We can see the outlines of soundness by reading the rules top to bottom, and the outline of an algorithm by reading them bottom to

top. As we are only interested in pure assertions for now, we carry the structural heap without inspecting it (though substitutions are propagated through them).

$$\textsc{Axiom}\quad \Pi \mid \Sigma \vdash \mathbf{true} \mid \Sigma \qquad \textsc{Inconsistency}\quad \Pi \wedge E \neq E \mid \Sigma \vdash \Pi' \mid \Sigma' \qquad \textsc{=Propogation}\quad \frac{\Pi[E/x] \mid \Sigma[E/x] \vdash \Pi'[E/x] \mid \Sigma'[E/x]}{\Pi \wedge x = E \mid \Sigma \vdash \Pi' \mid \Sigma'}$$

$$\textsc{=TautLeft}\quad \frac{\Pi \mid \Sigma \vdash \Pi' \mid \Sigma'}{\Pi \wedge E = E \mid \Sigma \vdash \Pi' \mid \Sigma'} \qquad \textsc{=TautRight}\quad \frac{\Pi \mid \Sigma \vdash \Pi' \mid \Sigma'}{\Pi \mid \Sigma \vdash \Pi' \wedge E = E \mid \Sigma'}$$

$$\textsc{ExcludedMiddle}\quad \frac{\Pi \wedge E_1 = E_2 \mid \Sigma \vdash \Pi' \mid \Sigma' \quad \Pi \wedge E_1 \neq E_2 \mid \Sigma \vdash \Pi' \wedge \Sigma'}{\Pi \mid \Sigma \vdash \Pi' \mid \Sigma'} \quad \begin{array}{l} E_1 \not\equiv E_2 \\ E_1 = E_2, E_1 \neq E_2 \notin \Pi \\ fv(E_1, E_2) \subseteq fv(\Pi, \Sigma, \Pi', \Sigma') \end{array} \qquad \textsc{Hypothesis}\quad \frac{\Pi \wedge P \mid \Sigma \vdash \Pi' \mid \Sigma'}{\Pi \wedge P \mid \Sigma \vdash \Pi' \wedge P \mid \Sigma'}$$

$$\pm\textsc{PerformLeft}\quad \frac{\Pi \wedge E = E' \pm r \mid \Sigma \vdash \Pi' \mid \Sigma'}{\Pi \wedge E = E' \pm n \pm m \mid \Sigma \vdash \Pi' \mid \Sigma'} \quad (\text{where } r = n \pm m)$$

$$\pm\textsc{FinishLeft}\quad \frac{\Pi \wedge E = r \mid \Sigma \vdash \Pi' \mid \Sigma'}{\Pi \wedge E = n \pm m \mid \Sigma \vdash \Pi' \mid \Sigma'} \quad (\text{where } r = n \pm m)$$

$$\pm\textsc{PerformRight}\quad \frac{\Pi \mid \Sigma \vdash \Pi' \wedge E = E' \pm r \mid \Sigma'}{\Pi \mid \Sigma \vdash \Pi' \wedge E = E' \pm n \pm m \wedge P \mid \Sigma'} \quad (\text{where } r = n \pm m)$$

$$\pm\textsc{FinishRight}\quad \frac{\Pi \mid \Sigma \vdash \Pi' \wedge E = r \mid \Sigma'}{\Pi \mid \Sigma \vdash \Pi' \wedge E = n \pm m \mid \Sigma'} \quad (\text{where } r = n \pm m)$$

4.3.2 Deciding DOM entailment

Consider now entailment between the DOM heap formulae (T, SF, CF, F and G). We give entailment rules typed for each of the three DOM structures: Trees, forests and groves.

Assume $\mathrm{D} \in \{\mathrm{T},\ \mathrm{F},\ \mathrm{G}\}$, and assume that the identity properties of the various $\varnothing_{\mathrm{D}}$ are handled elsewhere. We will write $x \in P$ for the symbol x occurring syntactically in formulae P.

$$\textsc{Tree-True}\quad T \vdash_{\mathrm{T}} \mathbf{true} \qquad \textsc{Forest-True}\quad F \vdash_{\mathrm{F}} \mathbf{true} \qquad \textsc{Context}\quad \alpha_{\mathrm{D}} \vdash_{\mathrm{D}} \alpha_{\mathrm{D}} \qquad \textsc{Empty}\quad \varnothing_{\mathrm{D}} \vdash_{\mathrm{D}} \varnothing_{\mathrm{D}}$$

$$\textsc{Tree}\ \frac{F \vdash_{\mathrm{F}} F'}{\mathbf{s_{id}}[F]_{\mathbf{fid}} \vdash \mathbf{s_{id}}[F']_{\mathbf{fid}}}$$

$$\textsc{Tree-Tree}\ \frac{T \vdash_{\mathrm{T}} T'}{\mathbf{tree}(T) \vdash \mathbf{tree}(T')} \qquad \textsc{Real-Tree}\ \frac{T \vdash_{\mathrm{T}} T'}{\mathbf{T} \vdash \mathbf{tree}(T')}\ \text{and } \alpha \notin T$$

$$\textsc{Forest-Pair}\ \frac{T \vdash_{\mathrm{T}} T' \qquad F \vdash_{\mathrm{F}} F'}{T \otimes F \vdash_{\mathrm{F}} T' \otimes F'} \qquad \textsc{Forest-Tree}\ \frac{T \vdash_{\mathrm{T}} T'}{T \vdash_{\mathrm{F}} T'}$$

$$\textsc{Everywhere-More}\ \frac{T \vdash_{\mathrm{T}} T' \qquad F \vdash_{\mathrm{F}} \Box T'}{T \otimes F \vdash_{\mathrm{F}} \Box T'} \qquad \textsc{Everywhere-End}\ \frac{T \vdash_{\mathrm{T}} T'}{T \vdash_{\mathrm{F}} \Box T'}$$

$$\textsc{Somewhere}\ \frac{T \vdash_{\mathrm{T}} T'}{T \otimes F \vdash_{\mathrm{F}} \Diamond T'} \qquad \textsc{Somewhere-More}\ \frac{T \nvdash_{\mathrm{T}} T' \qquad F \vdash_{\mathrm{F}} \Diamond T'}{T \otimes F \vdash_{\mathrm{F}} \Diamond T'}$$

$$\textsc{Somewhere-End}\ \frac{T \vdash_{\mathrm{T}} T'}{T \vdash_{\mathrm{F}} \Diamond T'} \qquad \textsc{Every-EveryWhere}\ \frac{T \vdash_{\mathrm{T}} T'}{\Box T \vdash_{\mathrm{F}} \Box T'} \qquad \textsc{Some-SomeWhere}\ \frac{T \vdash_{\mathrm{T}} T'}{\Diamond T \vdash_{\mathrm{F}} \Diamond T'} \qquad \textsc{Every-SomeWhere}\ \frac{T \vdash_{\mathrm{T}} T'}{\Box T \vdash_{\mathrm{F}} \Diamond T'}$$

$$\textsc{Grove-Tree}\ \frac{T \vdash_{\mathrm{T}} T'}{T \vdash_{\mathrm{G}} T'} \qquad \textsc{Grove-Pair}\ \frac{G_1 \vdash_{\mathrm{G}} G_1' \qquad G_2 \vdash_{\mathrm{G}} G_2'}{G_1 \oplus G_2 \vdash_{\mathrm{G}} G_1' \oplus G_2'}$$

Again, we apply rules in any order that is possible, to the same end. Here, we require a negative antecedent (as we have no negation in our fragment). We have bivalancy, so the inability to prove P means $\neg P$ follows. We write $\nvdash$ for the inability to derive an entailment with our proof theory; effectively, it means the process gets stuck during a proof attempt. We also have a side condition, $\alpha \notin T$; this is easily checked in a purely syntactic fashion by ensuring no α occur in the formulae.

The majority of the rules are self evident pattern matching; more interesting are the handling of **true** and the modalities. As expected, anything entails **true**. For the modalities, we can entail a less abstract version into a more abstract by folding the less abstract components in under the modality, as long as the trees are suitably entailed. We ensure we do not have to backtrack in the somewhere case by only examining more of the forest structure if we have no found a suitable match.

Rules within this system can be read bottom to top, and top to bottom. Top to bottom, we see the outlines of soundness for any derivation. Bottom up, we see a form of "guided search" which can form into an algorithm. The structure of the antecedents, which fall directly from the symbolic heap, ensures that the search will terminate, and end quickly; most rules do not branch, and those that do ensure at least one of the branches is non-branching. Moreover, each antecedent is "smaller" than the consequent.

4.3.3 Logical and existential variables

Conspicuously absent from the approach so far is the handling of both ghost and existential variables, used regularly in the small axioms. We cannot avoid this question, but it is made more complex than the Smallfoot situation by the large class of heap structure and extensive use of logical variables in the small axioms to maintain invariants over pre and post conditions, as well as in the context hole names.

Our preliminary thoughts involve adding a class of "logical pure" variables that can describe DOM structure, but do not assert its presence in the heap. We also add "hatted" variables, $\hat{X}$, interpreted as existentially quantified (this is the approach of JStar). A unification process would

be required in certain entailment and frame inference situations to bind these existential variables to concrete data.

4.3.4 Deciding Symbolic entailment

We can view structural entailments as lists of $*$ conjuncts on both side. For each on the left, we attempt to show that it entails one of the right. Of course, the syntactic shape of the formulae may not match initially, but we can safely collapse away any hiding quantification introduced to provide a split.

No further splitting will be needed - if we have to introduce a split to find a matching conjunct, we will inevitably have a hole on the opposite side that we have introduced, that we cannot remove.

$$\textsc{Collapse-Hide-Left}\quad \frac{\Pi \mid \Sigma_1 * \Sigma[D/\alpha] * \Sigma_2 \vdash \Pi' \mid \Sigma'}{\Pi \mid \Sigma_1 * \mathrm{H}\alpha.(\Sigma * \alpha \leftarrow D) * \Sigma_2 \vdash \Pi' \mid \Sigma'} \quad (\text{where } \alpha \in fn(\Sigma))$$

$$\textsc{Collapse-Hide-Right}\quad \frac{\Pi \mid \Sigma \vdash \Pi' \mid \Sigma'_1 * \Sigma'[D/\alpha] * \Sigma'_2}{\Pi \mid \Sigma \vdash \Pi' \mid \Sigma'_1 * \mathrm{H}\alpha.(\Sigma' * \alpha \leftarrow D) * \Sigma'_2} \quad (\text{where } \alpha \in fn(\Sigma'))$$

$$\textsc{Fragment-Remove}\quad \frac{D \vdash_{\mathrm{D}} D' \qquad \Pi \mid \Sigma \vdash \Pi' \mid \Sigma'}{\Pi \mid \alpha_{\mathrm{D}} \leftarrow D * \Sigma \vdash \Pi' \mid \alpha_{\mathrm{D}} \leftarrow D' * \Sigma'}$$

The collapse rules simply removes splits within formulae that are under hiding quantification; we can just slot the formulae back together and remove the quantifier. Fragment removal subtracts DOM fragments from the right hand side of the entailment, by finding a corresponding left hand side fragment. In a naive implementation, we can simply try all possibilities until we succeed. More intelligently, we can take the *head* of each fragment on the left, and quickly find the set of potential matches on the right.

It is this rule that makes use of the DOM entailment theory and it is here (and only here) that we really invoke any DOM specific reasoning. These three rules, along with the pure entailment set, need to nothing about the meaning of the DOM formulae they are handling - it is purely a syntactic operation. We say the fragment entailment theory is *semantically agnostic*; it has no knowledge of the underlying meaning other than it's syntax.

Proposition 20. *The set of pure entailment rules, DOM entailment rules and fragment entailment rules form a sound proof theory for our fragment.*

We have not yet formally proven this result, but believe it to be true. We do not (yet) say complete; this is a much harder result to attain. Whilst completeness is certainly desirable, it is perhaps less important that expanding the size of programs we can soundly prove: it would, we think, be better to prove a class of size $10x$ programs soundly, than a class of x programs soundly and completely.

4.3.5 Frame inference

So far, we have only explored the question $\Pi \mid \Sigma \vdash \Pi' \mid \Sigma'$. This does not allow us to use local reasoning; often, the work we want to perform (e.g. applying a small axiom) will be executed on a much larger assertion that the axiom supports. The full proof theory gives us the frame rules

so we can set aside unneeded portions of the assertions, but only when we know what the frame actually is.

For the symbolic heap, finding a frame is essentially answering the question "Given $\Pi \mid \Sigma \vdash \Pi' \mid \Sigma'$, split Σ using hiding so that it contains * conjuncts entailing each of (possibly hidden) * conjunct Σ'. With this splitting, we can use an analogue of the revelation and separation frames to remove these pairs of conjuncts until the right hand side has no structural formulae. Any spacial assertions remaining on the left is a sound choice for the spacial frame.

The splitting process is driven the idea that we can freely introduce fresh structural splittings at various points in a formulae, corresponding to fragments of the data structure. For example, the following assertions are sound for any β, F, D_i, s.

$$\begin{aligned} \beta \leftarrow \mathbf{s_{id}}[F]_{\mathbf{fid}} &\iff \mathrm{H}\alpha_{\mathrm{F}}.(\beta \leftarrow \mathbf{s_{id}}[\alpha_{\mathrm{F}}]_{\mathbf{fid}} * \alpha_{\mathrm{F}} \leftarrow F) \\ \beta \leftarrow D_1 \otimes D_2 &\iff \mathrm{H}\alpha_{\mathrm{D}}.(\beta \leftarrow \alpha_{\mathrm{D}} \otimes D_2 * \alpha_{\mathrm{D}} \leftarrow D_1) \\ &\vdots \end{aligned}$$

We have seen the usefulness of this in the meta-theory by use of the three new Segment Logic rules. Fresh name elimination and revelation frame allow us to "frame off" the hiding quantification from a $*$ conjunct of fragments, whilst separation frame allows us to frame off individual fragments.

We can see that any finite formulae has finitely many possible context splitting introductions, as long as we do not introduce $*$ conjuncts of the form $\alpha_{\mathrm{D}} \leftarrow \beta_{\mathrm{D}'}$. Call formulae without these elements *sensible*. If we consider only sensible formulae, then one can evidently give an algorithm that rewrites a formulae into the set $\mathcal{S}$ of all possible split formulae (though this set will be exponential in the size of the original).

Given $\Pi \mid \Sigma \vdash \Pi' \mid \Sigma'$, we try and find an equivalent rewriting that makes the frame evident via the following steps. First, we collapse all hiding quantification on the left formulae, and extrude all hiding quantifiers on the right to be outermost. We now have a formulae where Σ contains no hiding, and Σ' is of the form $\mathrm{H}\alpha 1. \ldots . \mathrm{H}\alpha_n \Sigma''$ where Σ'' contains no hiding.

We then generate $\mathcal{S}$ for the left hand formulae. For each $s \in \mathcal{S}$, we take each $*$ conjunct, and see if it entails a conjunct within Σ''. If it does, we remove them both and continue with the new formulae. If, eventually, we have an empty Σ'', then the updated Σ forms a valid frame. Depending on $\Pi' \mid \Sigma'$, there may be many valid frames; we return the set, after collapsing any introduced hiding quantification that can be removed. Implicit in this process is some handling of the logical α context variables, and the unification between those freshly introduced and already present. We defer this problem until we have a better answer to the general logical variable question.

This process is extremely naive; a proof of concept at best. There are easy optimisations that reduce the search space; for example, we can "guide" the splitting of Σ based on the $*$ conjuncts of Σ'. If one were, say, $\alpha \leftarrow$ `foo`[...], we know that any splitting that does not introduce a split giving us `foo` at the head of a formulae will not be acceptable.

4.4 Symbolic execution

4.4.1 Symbolic execution of language features

In section 2, we explained the inference rules of Hoare logic as being designed to act on the assertions in a similar way to the operations of the commands on the program state. We can take this analogy further by considering them as performing direct symbolic update

As in Smallfoot, we create a set of symbolic execution rules that reduce the complexity of $\mathbb{C}$ in questions of the form $\vdash \{P\}\mathbb{C}\{Q\}$. By considering the semantics of command sequencing, we note

we can enact the effect of the first command in $\mathbb{C}_1; \mathbb{C}_2$ on the state P to obtain one or more Hoare triples over a new state P' and $\mathbb{C}_2$. The rules are straightforward. Assuming pattern matching of a triple with the conclusion of each, we read them in the normal way "To prove the conclusion true, it suffices to prove the premise true".

We write $\neg P$ to mean the inverse of the equality given in P.

$$\textsc{Assign}\quad \frac{\{x = E[x'/x] \wedge (\Pi \mid \Sigma)[x'/x]\}\texttt{C}\{\Pi' \mid \Sigma'\}}{\{\Pi \mid \Sigma\}\texttt{x := E; C}\{\Pi' \mid \Sigma'\}} \quad (x' \text{ fresh})$$

$$\textsc{Conditional}\quad \frac{\{\Pi \wedge P \mid \Sigma\}\text{C1; C}\{\Pi' \mid \Sigma'\} \qquad \{\Pi \wedge \neg P \mid \Sigma\}\text{C2; C}\{\Pi' \mid \Sigma'\}}{\{\Pi \mid \Sigma\}\texttt{if P then C1 else C2; C}\{\Pi' \mid \Sigma'\}}$$

$$\textsc{Skip}\quad \frac{\{\Pi \mid \Sigma\}C\{\Pi' \mid \Sigma'\}}{\{\Pi \mid \Sigma\}skip; C\{\Pi' \mid \Sigma'\}}$$

$$\textsc{End}\quad \frac{\Pi \mid \Sigma \vdash \Pi' \mid \Sigma'}{\{\Pi \mid \Sigma\}\texttt{end}\{\Pi' \mid \Sigma'\}}$$

4.4.2 Symbolic Small Axioms

The language feature rules do not handle our specific commands. We use the symbolic small axioms for this purpose, and give two general rules that appeal to them. The first handles small axioms that perform pure updates (i.e. of the form `x := C`), the second which do not. The difference is due to the Floyd style fresh variable introduction in the pure update case.

The complexity of our heap means we cannot follow the Smallfoot approach, where small axioms are encoded as update rules of the form we used above, and "excess" pure and structural heap are just pattern matched. We must perform spacial surgery, finding the actual frame, before we can apply our rules. We must also perform some with unification on ghost variables. We can then perform a symbolic update, and restore the surrounding frame.

We do not yet give rules for this process, as its sound application requires answers to the logical variable question we do not yet have.

4.4.3 Symbolic execution algorithm

We can finally give an algorithmic approach, utilising the mass of developed theory, to provide a process for deciding $\vdash \{P\}\mathbb{C}\{Q\}$. This algorithm makes assumptions about our expected handling of the Symbolic Small axiom logical and existential variable issues.

Let H be any Hoare triple of that form, `Pre` and `Post` functions that fetch the conditions. We also assume pattern matching that unifies triples with the rules given above, and refer to the `Continuation` of a command, $\mathbb{C}_2$ in $\mathbb{C}_1; \mathbb{C}_2$.

```
algorithm sym_exec(H):
  If H matches the conclusion of End
    Return Pre(H) Entails Post(H)
  End If

  If H matches the conclusion of Assign, Sequence OR Skip
    Return sym_exec(Premise(Matched Rule))
  EndIf
```

```
If H is a Symbolic Small Axiom with a Pure Update
  Freshen instances of the variable under assignment in H
EndIf

If H is a Symbolic Small Axiom
  Let F be the triple generated by the Frame-Inference process on Pre(H)
    against any Symbolic Small Axiom precondition for the command
  If Cannot Find F
    Return False
  EndIf
  Update F Based on Small Axiom Conclusion
  Restore Removed Frame giving H'
  Return sym_exec(Continuation(H))
EndIf

Return False
```

Theorem 21. *The algorithm terminates, and returns true if and only if the Hoare triple H is valid.*

The algorithm functions because the structure of the symbolic rules; each take program statements of the form $\mathbb{C}_1; \mathbb{C}_2$, and eliminates $\mathbb{C}_1$ via updates to the symbolic heap. The algorithm continues with $\mathbb{C}_2$, until there are no further continuations (the `end` command), and we have reduced the problem to the validity of an entailment.

The algorithm terminates, as each process used by it terminates. The proof search search is somewhat branching, slightly within the checks validity but significantly within the frame inference process. How much of a performance issue this will be remains an issue I shall investigate.

4.5 Current state and conclusions

This work is very preliminary; any of the above results may be unsound. The key to continued development in this area will be the mutual *stability* of each component. If we extend the DOM reasoning, it should not affect the Segment reasoning; similarly, we should ensure that extending the pure formulae does not break the frame inference algorithm. The system above is largely stable, due to the agnostic nature of the segment logic. We should be able to branch the development of the theory into "Segment issues" and "DOM issues", focusing on each separately without inference based on differing choices.

The next step is to complete the handling of existential and logical variables, then to to prove the soundness of the approaches. We can then extend the coverage to handle the Reasonable Web issues. We do not expect this to be particularly difficult: DOM `#document` nodes are a simple extension to the DOM entailment theory, and URI/function formulae can be added to the pure entailments with respect to a user provided web instance. In parallel with the theory, we will be developing an implementation of the results, allowing us to test the performance of the techniques.

5 Future work and a path to a thesis

5.1 Goldilocks and the $n + 1$ assertion languages

In this short report, we have seen four separate assertion languages: Standard FOL, Separation Logic with simple heap models, the Reasonable Web language using Context Logic, and the automatable theory using Segment Logic. Three fundamentally similar imperative programming systems have had four assertion languages, depending on the data structures being manipulated. Is this an inevitable situation, and will we need a new logic for every new scenario modelled?

It seems evident that classical first order has proven rather inadequate as the assertion language for resource based program reasoning, especially when compared to Separation Logic and the various Context approaches. However, each of these have their own nuances, that have necessitated the many approaches we've introduced and may prevent them from being a kind of "base level" theory on which resource reasoning can be constructed.

It may be that Separation Logic proves adequate despite the inelegance high level reasoning with it, much in the same way that C continues to be used to program even high level abstractions. If the high level abstraction concept is truly useful, then perhaps Context Logic is enough, when refined into the flexible form of Segment logic. Debate is ongoing, but we believe that some part of the answer lies in the balance of *handle availability* and *owned indirection* that each logic provides.

By handle, we mean an object we can use to refer to a specific element of the data structure being asserted - some part of it that we can "take hold of" and uniquely identify we have grabbed. We say "owned indirection" to mean how we can use these handles to discuss the purely abstract target of the handle, limiting the specific knowledge we are required to have whilst still asserting primacy over it. These two elements are a simplification of the resource intuition. We use handles to discuss distinct clusters of resource, and indirection both to form links to other resource and to "forget about it" when we do not need it.

With this interpretation, first order logic using normal variables to model the heap might be seen as "Handle heavy, indirection light". We can talk about heap via variable names (acting as handles), but cannot use these in a context free scenario. Any time we use a handle, we lack any concept of owned indirection; we cannot abstract away from the need for specific data knowledge, or the fight to find other logical users of it.

Separation logic's heap model goes to the opposite extreme. It is both handle and indirection heavy, but in an uncontrolled fashion. We have a handle for each heap cell with strong ownership information, but are forced to have this and must encode any structure we wish to reason about via it. This makes sense, given the "low level" nature of the reasoning. Programmers are thrown into the heap with full freedom, and must manage every detail themselves. As languages typically allow them to introduce abstract data type, the logic uses inductive predicates to regain some control of the chaos and hide some of the enforced indirections.

Single holed Context Logic is handle light, but with very flexible owned indirection. We can create a single handle when we need it, but only one at a time. Segment logic alters this, allowing us be both handle and indirection heavy, but in a controlled fashion - we introduce only what we want, and eliminate them when no longer needed. It would seem to be the most flexible of the approaches seen so far - we can use it to emulate any of the others, but not vice versa. Whilst first order logic has "too little", Separation logic "too much", with Segment logic being "just right".

This intuition can explain the use of restriction in the models of the segment logic, if we stop viewing them as a model of machine memory and start seeing them as the mental models of a programmer. Under this thesis, high level models do not just capture the high level of the data structures, they capture the high level of the programmers conceptual view. The models have become more like a "language of data structures", and we can sanely tie this to firmer mathematical

ground by noting exactly the structures described by this language (the trees described by models of the Segment DOM are exactly the DOM trees we are describing).

Though Segment Logic is certainly seems a step in the right direction, the search for a logic well balanced in both its level of indirection and structural reasoning support is nowhere near over. For example, the "somewhere, potentially deep down" modalitity that is definable within the SHCL that can no longer be simply written. Recall that this operation can be expressed by the existence of an arbitrary context that wraps data for which P is true.

$$\Box P \triangleq \exists C.C \circ P$$

For example, we can state the heap has a root node `root` with, somewhere beneath it, a `descendent` by writing `root`[$\Box$`descendent`[**true**]]. Call this usage *syntactically local* - formulae asserting structural properties can be written close to their need, and so be defined as a small sentence fragment for insertion. It also gives the formulae a very natural reading for users.

Segment logic allows structural separation only at the fragment level. We could not write what we might expect to be equivalent

$$\Box P \triangleq \exists \alpha.@\alpha * \alpha \leftarrow P$$

With the resultant formulae expanding out as `root`[$\exists \alpha.\ @\alpha * \alpha \leftarrow$ `descendent`[**true**]]. This syntactically local use is not be well formed, as fragments cannot appear within DOM Formulae. Although we can recover the same logical property, we have to do it by rewriting the formulae in a very global sense. We have "lost" one of the expressive techniques that have proven so valuable in local reasoning.

5.1.1 The need for multiple abstractions

Regardless of the logic chosen, it is facile to claim one abstract view of data in a complex program as "just right". Consider a *fringe list*: an n-ary tree where the preorder list of leaves is maintained as a linked list within the structure. When programming using it, the engineer will often consider it a tree in some code, and a list elsewhere. Their intuitive view of the data changes depending on the situation, and the data structure supports multiple abstract interpretations.

So, just as one switches mental models when programming, it seems natural to enable the switching of abstractions during formal reasoning. The Segment Logic offers an interesting possibly here, via its abstraction agnosticism - as long as the underlying model admits the context variables, then the Segment formulae do treat them as opaque. This suggests that one can introduce different revelation operators, where we expose data inside one abstraction in exactly the format we would like to consider it.

The questions this raises about the footprint of such a revelation are interesting; data local in one abstraction (e.g. 2 adjacent nodes in the list view of a fringe list) may be non-local in another (it may corresponds to two virtually disjoint branches in the tree). If we choose a model of trees at the low level, one logical break in the list abstraction may introduce several concrete context holes in the tree; if we view it as a list, what does it mean to asserting tree structure on the model? Exactly how to accomplish this is an open problem, but if solved, opens the doors to a different approach to program verification based on directly modeling the mixed abstractions the program is using.

5.1.2 Library Reasoning

As the complexity of software grows, programmers are becoming more reliant external libraries and frameworks. These introduce the levels of abstraction into languages needed to perform

complex tasks. As in the DOM case, and seen the example of section 2, specifying them in terms of their implementation ties the correctness of the abstract concept to a concrete implementation; we cannot swap implementations, or discuss code without getting bogged down in details.

DOM can be viewed as an instance of a language agnostic API, with a conveniently simple abstraction. Many similar libraries exist: OpenGL for graphics, OpenCL for parallel computation; we can even consider the varying implementations of the POSIX or libc APIs. However, it is rare to find one with a single underlying model as conveniently uniform as DOM trees: Most will have many differing data structures.

As we have reasoned about DOM at the DOM level, it is desirable to reason about other libraries at the level of their abstractions. We hope to generalise the DOM concept to other libraries, starting with another high level structural library and seeing what commonalities occur. This can be expected this to segue with the "multi-abstraction" reasoning hypothesised above to provide the beginnings of techniques that can reason about more general library code, which can be reasoned largely independently from the language hosting it.

This would seem to introduce a theoretical distinction between the fields of *language reasoning* and the potential *library reasoning*. Language reasoning focuses on abstract interpretations of fundamental language features; memory allocation, imperative heap update and so forth, loops and conditionals - all modelled by, say, Separation Logic. Library reasoning would seem to layer atop this, giving us reasoning distinct from the underlying implementation details, focusing on the application specific abstraction. We can see this difference in the Reasonable Web work; there is a large group of rules for reasoning language primitives (such as function introduction), but these make little to no use of the structural reasoning techniques. The small axioms of the language are entirely focussed on the DOM library reasoning; we could pull that out and replace it with an alternative tree update library without needing significant alterations to the remainder of the formalism.

5.2 A route to a thesis

The work this year has affirmed the belief that the tensions between the theoretically achievable and the practically useful drives this research. Each feeds the other, an uroboric process generating ever more refined ideas.

We are currently well into one complete cycle. Starting with large topic of "Practical web programming issues", the insufficiencies in existing formalisms have driving the development of the Reasonable Web. With this foundation, the inability to easily write large examples has motivated reasoning automation. Again, this practical need mandates new theoretic work in pragmatic subsets of the logic, allowing the upcoming development of tools. Finally, I will be able to consider non-trivial examples with ease, find the worst of the inevitable limitations, and begin again.

The remainder of my first year, and a large portion of my second will be in completing this iteration. The Reasonable Web work is mostly complete, and research in automation and tooling is progressing. We expect to meet the timetable of figure 19.

This will generate two papers, at approximately the milestones indicated.

1. **Milestone**: Submit revised paper, "Resource Reasoning for Mashups" (key elements given in chapter 3). Planned for submission to POPL 2010.

2. **Milestone**: Submit Automation paper. Planned for late 2009, or early 2010.

Once the first cycle of work is complete, two routes stand out from the possible continuations. The Reasonable Web system is a mere beginning, modelling a simplified version of the problem.

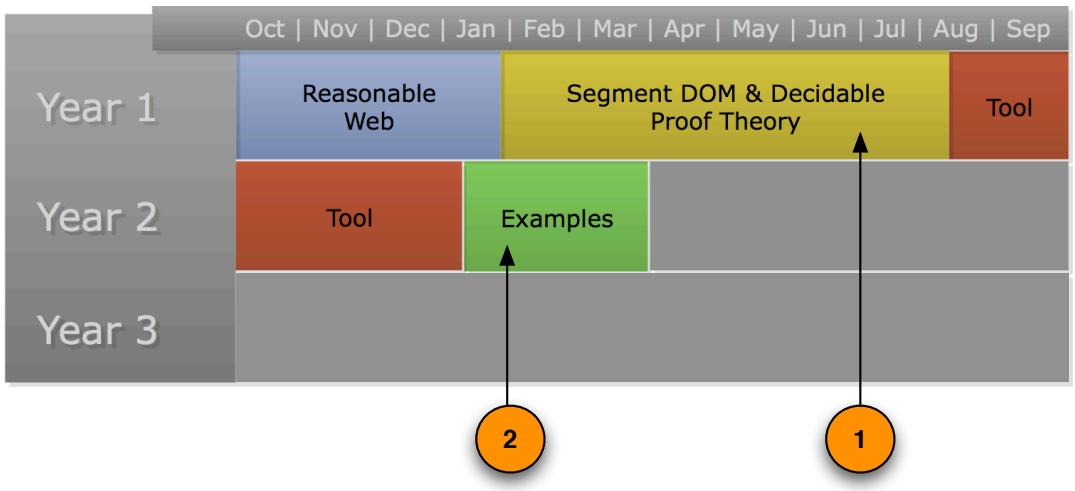

Figure 19: Completed and underway work, with labeled milestones

As client side web programs are almost always written in JavaScript and contain many features we have entirely ignored, it seems logical to pursue a more sophisticated formal model. Maffeis et al [20] have completed an operational semantics for the complete JavaScript standard, containing many features our reasoning cannot currently handle, but are common practice in web programming. The lack of support for several of these features limits our ability to handle many real programs.

Investigations will focus on the joining of the distinct yet interacting JavaScript and DOM heap models, and on development of a proof theory for JavaScript's more atypical, whilst trying to maintain tractable automation. We expect this would take the remainder of my investigative time.

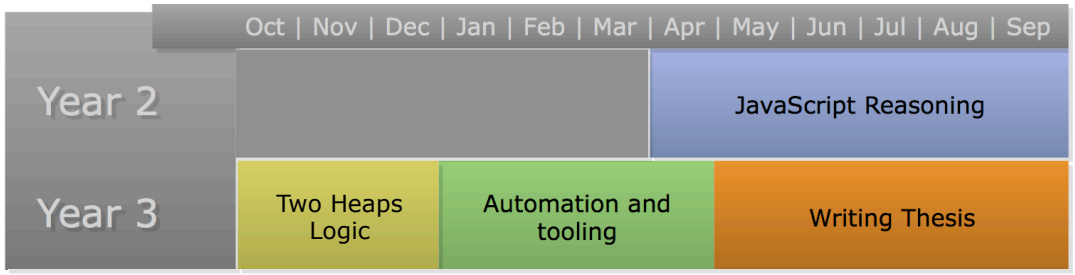

Figure 20: Schedule for JavaScript reasoning

Alternatively, it may be more natural to consider DOM as a specific instance of *Library Reasoning*, and continue my research in terms of this. This will involve finding or creating another library with a less homogeneous abstraction (i.e. the fringe lists), and constructing a local reasoning formalism that allows us to view the different levels of abstraction. From this, we hope to

draw some preliminary general results and ideas for expanding the library reasoning concept in step with automation.

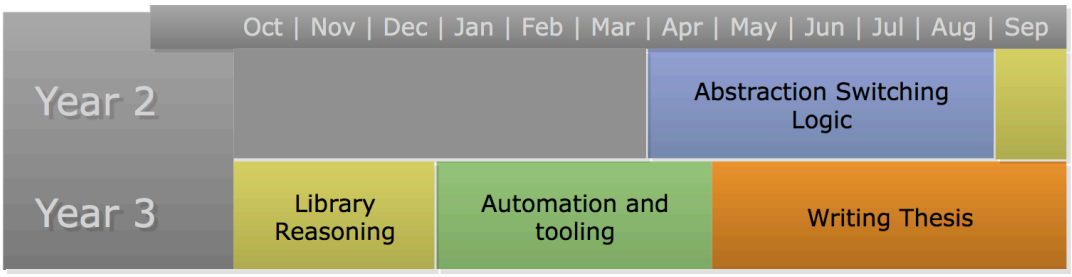

Figure 21: Schedule for Library reasoning

5.3 Expected thesis structure

The work in Figure 19 forms a "safe" and sizeable portion of my thesis. The Reasonable Web and associated automation theory is entirely complete in parts, well underway in others and we expect it to form around half the final document.

1. **Program Reasoning**
   Don't we already do this? & how testing has failed to scale
2. **Verification techniques**
   Existing techniques, and the impact of local reasoning
3. **The Reasonable Web**
   A local reasoning formalism for web programming
4. **Modelling Mashups**
   Examples, problems and verification for a specific web programming domain
5. **Automated Reasoning for Segment DOM**
   Minimising the burden of proof on the programmer
6. ...
7. **Conclusion and bibliography**
   Summary of contributions and future work

What fills the ellipsis will depend on which of the paths outlined above we follow. If we pursue the JavaScript route, at least three new chapters will be needed.

1. **Local Reasoning for JavaScript**
   Handling higher order functions, extensible objects and unfortunate language choices
2. **Integrating DOM and JavaScript**
   A tale of two heap models

3. **Automated Reasoning for JavaScript and DOM**
   Expanding on the tooling problem

Library reasoning would break down approximately as follows.

1. **Abstraction switching**
   When a single view is inadequate

2. **Library Reasoning**
   Generalising the high level view

3. **Automated Library Reasoning**
   Results in generalising the proof theory

5.4 Conclusions

This report has covered the background of local reasoning, a novel formalism for web programming, a verification system based on local reasoning, and the beginnings of an automation system for this formalism. It represents the start of a long research project, but lays foundations for the future research work we shall pursue, and shows the smallest beginnings of tantalising areas of program verification.

The near future work is set as the automation system outlined is incomplete, with many more questions than answers. Past this, we have outlined two distinct and clear routes that continue the research project. If we pursue a deepening of the web programming concept, it is possible we can create a system that is useful for not only programs mimicking realistic situations, but some actual web programs that many people use today. Following the library reasoning route will probably provide fewer directly applicable results, but will set the scene for broader future research, and hopefully make it easier for either us or others to deepen the focus on specific semantic domains.

Regardless, it seems evident that the local reasoning concept does scale past the "simple" structural reasoning of Separation logic into higher level structures, through specifications to "realistic" programming. Whether this work needs the special attention of theorists for each possible applicative situation, or can be generalised successfully to a more open topic will be an interesting question for the coming years.

References

[1] *ECMAScript Language Specification*. European Computer Machinery Association, June 1997. URL `http://www.ecma-international.org/publications/standards/Ecma-262.htm`.

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

[3] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In Kwangkeun Yi, editor, *APLAS*, volume 3780 of *Lecture Notes in Computer Science*, pages 52–68. Springer, 2005. ISBN 3-540-29735-9.

[4] 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. ISBN 3-540-36749-7. URL `http://dx.doi.org/10.1007/11804192_6`.

[5] Berners-Lee. World Wide Web: Proposal for a HyperText Project. 1990.

[6] Cristiano Calcagno, Philippa Gardner, and Matthew Hague. From separation logic to first-order logic. In Vladimiro Sassone, editor, *FoSSaCS*, volume 3441 of *Lecture Notes in Computer Science*, pages 395–409. Springer, 2005. ISBN 3-540-25388-2.

[7] 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.

[8] Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local action and abstract separation logic. In *LICS*, pages 366–378. IEEE Computer Society, 2007.

[9] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Space invading systems code. In Michael Hanus, editor, *LOPSTR*, volume 5438 of *Lecture Notes in Computer Science*, pages 1–3. Springer, 2008. ISBN 978-3-642-00514-5. URL `http://dx.doi.org/10.1007/978-3-642-00515-2`.

[10] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In Zhong Shao and Benjamin C. Pierce, editors, *POPL*, pages 289–300. ACM, 2009. ISBN 978-1-60558-379-2.

[11] Dino Distefano. Abductive inference for reasoning about heaps. In G. Ramalingam, editor, *APLAS*, volume 5356 of *Lecture Notes in Computer Science*, pages 1–2. Springer, 2008. ISBN 978-3-540-89329-5.

[12] Dino Distefano and Matthew J. Parkinson. jStar: Towards practical verification for Java. In Gail E. Harris, editor, *OOPSLA*, pages 213–226. ACM, 2008. ISBN 978-1-60558-215-3. URL `http://doi.acm.org/10.1145/1449764.1449782`.

[13] Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920 of *Lecture Notes in Computer Science*, pages 287–302, Berlin, 2006. Springer-Verlag. URL `http://dx.doi.org/10.1007/11691372_19`.

[14] Philippa Gardner, Gareth Smith, Mark J. Wheelhouse, and Uri Zarfaty. DOM: Towards a formal specification. In *PLAN-X*, 2008. URL `http://gemo.futurs.inria.fr/events/PLANX2008//papers/p18.pdf`.

[15] Philippa Gardner, Gareth Smith, and Adam Wright. Resource reasoning about mashups, technical report. 2009.

[16] 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. ISBN 1-59593-685-X. doi: http://doi.acm.org/10.1145/1376916.1376953.

[17] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.

[18] Masayasu Ishikawa, Peter Stark, Mark Baker, Toshihiko Yamakami, Ted Wugofski, and Shin'ichi Matsui. XHTML™ Basic 1.1. Candidate recommendation, W3C, July 2007. http://www.w3.org/TR/2007/CR-xhtml-basic-20070713.

[19] 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.

[20] 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. ISBN 978-3-540-89329-5. URL `http://dx.doi.org/10.1007/978-3-540-89330-1_22`.

[21] Laurent Mauborgne. ASTRÉE: Verification of absence of run-time error. In René Jacquart, editor, *Building the information Society (18th IFIP World Computer Congress)*, pages 384–392. The International Federation for Information Processing, Kluwer Academic Publishers, Aug 2004.

[22] P. O'Hearn and D. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2): 215–244, June 1999.

[23] 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.

[24] Matthew J. Parkinson. Local reasoning for Java. Technical Report 654, University of Cambridge Computer Laboratory, November 2005. URL `http://www.cl.cam.ac.uk/TechReports/UCAM-CL-TR-654.pdf`. The author's Ph.D. dissertation.

[25] Matthew J. Parkinson, Richard Bornat, and Cristiano Calcagno. Variables as resource in hoare logics. In *LICS*, pages 137–146. IEEE Computer Society, 2006. URL `http://doi.ieeecomputersociety.org/10.1109/LICS.2006.52`.

[26] 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.

[27] John C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In J. Davies, A. W. Roscoe, and J. Woodcock, editors, *Millennial Perspectives in Computer Science: Proceedings of the 1999 Oxford-Microsoft Symposium in honour of Sir Tony Hoare*. Palgrave Macmillan, 2000.

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