Imperial College London
Department of Computing

Local Reasoning about Web Programs

Gareth David Smith

October 25, 2010

Supervised by Philippa Gardner

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London
and the Diploma of Imperial College London

Declaration

I herewith certify that all material in this dissertation which is not my own work has been properly acknowledged.

Gareth David Smith

Abstract

Since 1990, the world wide web has evolved from a static collection of reference pages to a dynamic programming and application-hosting environment. At the core of this evolution is the programming language JavaScript and the XML update library "DOM". Every modern web browser contains a DOM implementation which allows JavaScript programs to read and alter the web page that the user is currently viewing. JavaScript and DOM are extremely successful, and this success may be in part due to their highly dynamic and tightly integrated nature. However, this very nature hinders formal program analysis and tool development. Even the implementation independent specification that defines DOM is largely written in the English language, and not using any formal system.

While client-side web programming was once a simple discipline of form validation and interface trickery, it is fast becoming a far more serious business encompassing application development for the emerging ubiquitous "cloud". As this evolution gains pace there is an increasing demand for client-side tool support of the sort commonly enjoyed by "enterprise" programmers, working in more easily analysed languages such as Java.

This thesis makes use of recent developments in program reasoning using context logic to provide the first formal, compositional specification for the Fundamental Interfaces of DOM Core Level 1. It presents both a big-step operational semantics for the necessary operations of the library and a context logic for reasoning about programs which use the library. Finally, it presents example programs that use the library and shows how context logic can be used to prove useful properties of those programs.

Acknowledgements

- to my supervisor Philippa Gardner, who has taught me and encouraged me;
- to my family, who are my closest friends;
- to my friends, who are my family.

Contents

1. Introduction

In his 2000 paper "Intuitionistic Reasoning about Shared Mutable Data Structure" [57], John Reynolds described programming with pointers as "a no-man's land inhabited by many useful and intuitively straight-forward programs that have been poorly served by both type systems and program-proving methodologies". He claimed that this no-man's land was "far more than a hacker's jungle", and proposed the methods which ultimately led to the development of separation logic and have since met with a great deal of success in that no-man's land.

Today, there is a growing body of work on reasoning about pointer programming, including significant success with automated reasoning about complete industrial programs[70]. While the no-man's lands of pointer programs may have been claimed and tamed, there is now a new world of programs which have been poorly served by both type systems and program-proving methodologies. Dynamically typed "scripting languages" sacrifice the safety of type systems on the altar of flexibility and short release-cycles. They are extremely difficult to reason about, and have been very successful on the fast paced world wide web, which rewards first-movers, and makes it easy to instantly distribute bug-fixes to an entire user base. This thesis takes the latest developments in the methodologies that were used to conquer the no-mans land of pointer programs, and applies them to forging a path through the new hacker's jungle.

1.1. The Hacker's Jungle

The origins of the world wide web lie in a proposal written in 1989 by Tim Berners-Lee[68] and re-written in 1990 with Robert Cailliau[69]. The proposal was to create "a large hypertext database with typed links", the purpose of which was to allow high energy physicists to share data, news and documentation. The first web pages were "static", meaning that any

two attempts to access the same page would return the same data, and that data would be composed of inert text and graphics.

In 1993 the Common Gateway Interface (CGI) was standardised[17], allowing web servers to serve not just static web pages, but also pages that were produced dynamically by a console program. This allowed for the creation of sites like search engines or other database query systems. This also opened the door for early versions of e-commerce sites such as Amazon. While this technology allowed a server to respond to each HTTP request with a unique, dynamically generated page; it did not introduce any dynamicism into the client – the web browser. Each page was still composed of text and images which could not respond to user-interaction without loading a new page from the server.

Three key technologies have since emerged in client-side web programming. In 1995, Netscape included a JavaScript implementation in the 2.0 release of their browser Netscape Navigator[26]. In the same year, Sun released the first version of their Java programming language, which introduced Java "applets". Hot on the heels of Netscape and Sun, Macromedia released "Flash" in 1996.

All three of these technologies aimed to enrich the user experience of browsing the web by allowing reactive code to run on the client machine. However, the specific goals of each technology differed slightly. JavaScript was initially most often used for simple form validation, while Java applets were used for "enterprise" client-server programming and Flash was mostly used for animation. These perceptions changed in 2004 and 2005 when Google unveiled Gmail and Google Maps. Both of these applications made use of JavaScript as their client-side programming language and are considered by many to have been the first wide deployments of the web programming techniques now known as "Ajax"[2]. The term "Ajax" is shorthand for "Asynchronous JavaScript and XML", and refers to the use of JavaScript running in a web browser and interacting in a timely fashion with the user while making asynchronous requests of the web server to ensure that any data the user requires is available in the client program when they require it. A good example of this technique is the scrolling map feature pioneered by Google Maps. The JavaScript program listens to the drag-events generated by the user's mouse, and uses those events to scroll the map in real time. In the background, the program requests adjacent map tiles from the server,

so that whenever the user drags a previously unseen area of map into view, that map tile will have already been loaded into program memory and the user need not wait for a network request.

In 2010, Java applets are rare on the web while Ajax web-applications are flourishing. JavaScript has become an international standard called ECMAScript[27]. What was Macromedia Flash is now called Adobe Flash, and features a programming language based on ECMAScript, which is called "ActionScript".

One of the keys to the success of Ajax is the tight integration of the JavaScript programming language with the user interface of the web browser. This integration is largely achieved through a library called DOM. DOM stands for "Document Object Model" and is a library for performing in-place XML update. Every modern web browser contains a DOM implementation which is used by JavaScript programs to read and alter the web page that the user is currently viewing.

The origins of DOM lie in several competing and incompatible XML update libraries developed during the 1990s. Today however, DOM is defined by an implementation neutral specification, maintained by the World Wide Web Consortium (W3C)[23]. The W3C first published "DOM Level 1" in 1998, and have built subsequent layers on top of that first standard. The current release of the specification, called "DOM Level 3" was published in 2004.

It is thanks to the implementation neutrality of the DOM and ECMAScript specifications that it is now possible to run the same client-side web program in Chrome, Firefox, Internet Explorer or Safari. In addition to implementations in web browsers, there are also DOM implementations available for most high-level programming languages. This makes it possible to write server-side CGI programs which use the same programming techniques for managing their XML. DOM also provides a convenient means for writing other programs that manipulate XML, without necessarily having anything to do with the web. The Open Document Format is an example of a modern XML-based format for storing word processor documents, spreadsheets, presentations, graphics and formulae. OpenOffice provides its own implementation of DOM in order to manipulate these documents.

It is instructive to compare the current state of DOM and JavaScript to that of their 1995 competitor, Java. DOM is successful, mature and

implementation neutral, but it suffers from being written in the English language rather than in a more precise, mathematical fashion. There are some cases which are unclear or ambiguous, and this leads to inconsistency in implementations and bugs in client code. This thesis represents the first formal mathematical specification for DOM. An operational semantics for JavaScript was only recently produced by Maffeis, Mitchell and Taly [49]. JavaScript programmers have no type system or any other kind of automated reasoning to support them. While many type systems have been devised for small subsets of the language[3, 64, 39], these subsets all ignore several key features of the language-proper. A recent survey by Richards, Lebresne, Burg and Vitek of JavaScript as it is used in the wild demonstrates that many of these features are in extremely wide use[60].This is in stark contrast with Java, which has been extensively studied from the point of view of precisely specified subsets of the language such as featherweight java[43], lightweight java[47], middlewight java[5] and ClassicJava[28]. Java has a powerful static type system which can be extended to support mechanisms such as ownership types[18, 7] and session types[42]. Parkinson introduced separation logic for reasoning about Java in 2005[56], and this work is being automated in the jStar tool[22]. This work all results in tools which support Java programmers in writing large reliable programs.

In the past, JavaScript and DOM programs have tended to be small. Such programs need little in the way of tool support, but benefit greatly from flexibility and tight browser integration. This is changing. As web programming and Ajax become more and more ubiquitous and complex, the demand for tools, libraries and other innovative methods of ensuring the reliability of web programs is increasing. Unit testing frameworks and well-designed libraries such as jQuery[45] can and do take some of the strain. However, thoroughly testing large complex code is difficult, especially when the number of platforms which must be tested is constantly increasing as new web browsers designed for new form-factors such as web-phones and tablet PCs are introduced to the market. Libraries can help reduce the complexity of application programs, but only at the expense of increasing the complexity of libraries. Whether it is the application author or the library author who is writing the complex code, it is clear that if programs are to be written in JavaScript and DOM then more advanced tools-support of the sort enjoyed by java will be essential.

Another solution to the problem of the increasing complexity of JavaScript and DOM programs is to write code in another language entirely. Better supported languages include Java, Haskell or the purpose-designed languages Flapjax or Milescript. In all these cases, the programmer can now write code in their language of choice and compile to JavaScript. This is not how Java was deployed on the web in 1995, but is how Google chose to deploy their new service"Wave", which was written entirely in Java and deployed using the Google Web Toolkit[19] which compiles the client-side portion to JavaScript. It is clear that even if web programmers are tempted away from JavaScript when writing programs, they will continue to compile to JavaScript for the foreseeable future. If DOM and JavaScript are to be the assembly language of the web, then a formal understanding of DOM and JavaScript will surely play a crucial role in developing compilers that target it. Since JavaScript seems to be resistant to static type analysis, and since a formal understanding of DOM and JavaScript will be important in the future development of programmers tools, and web language compilers, it is desirable to find another means of reasoning about DOM/JavaScript code.

This thesis concentrates on reasoning about DOM, while keeping in mind that future work will involve reasoning about JavaScript.

From the point of view of a JavaScript programmer, DOM is a large global data structure, which may be accessed by any part of a JavaScript program regardless of scope. This is reminiscent of a C programmer's heap, which is also a large global data structure which exists outside the normal rules of scope. Both structures play havoc with the modular design of programs, introduce problems of aliasing, and make formal reasoning difficult.

O'Hearn, Reynolds and others recently introduced separation logic which has met with a great deal of success in reasoning about C programs which make use of the heap [57, 58, 55, 59]. That work has led to a number of automated tools which have been very successful in finding and eliminating bugs in environments where testing is difficult and there is little support from a type system[4, 63, 70, 12].

Separation logic allows a mathematician or an automated tool to reason about the shape of a heap: to isolate only the small sub-heap that a particular subroutine requires to run, to reason about the effects of the subroutine on that sub-heap, and finally to use this reasoning to draw conclusions about

the whole global heap.

This methodology seems to be a sensible approach to reasoning about the large global structure of web programs. However, while the C heap is a simple bag of heap cells, the tree structure of DOM is more complex. Calcagno, Gardner and Zarfaty have shown[13] that separation logic is insufficient to reason about tree structures, and have introduced context logic to fill the gap. Context logic is inspired by ambient logic[16], and has been used to reason about programs that manipulate simple tree structures.

This thesis applies context logic to the task of reasoning about DOM programs.

1.2. Contributions and Thesis Outline

The key contributions of this thesis are a formal description of the Fundamental Interfaces of DOM Core Level 1, and an application of Gardner and Zarfaty's context logic for reasoning about DOM programs. DOM data is significantly more complex than the data models of previous context logic applications. The core conceptual difficulties presented by the DOM structure are distilled in "Featherweight DOM" and dealt with in Chapters 3 and 4. This work on Featherweight DOM then serves as a road map for work on the Fundamental Interfaces of DOM Core Level 1, which is covered in Chapters 6 and 7. Using this formal description and context logic, it is possible to reason in an intuitive and modular fashion about the correctness of programmes that use DOM to manipulate XML data.

Here is the outline of this thesis:

- Chapter 2 provides background on program verification techniques, beginning with Hoare logic and tracing the development of separation logic and context logic. It defines much of the mathematical notation and explains many of the techniques that are used in later chapters.

- Chapter 3 follows the example of featherweight java and presents "Featherweight DOM". Featherweight DOM concentrates on the Node interface of DOM Core Level One and demonstrates the conceptual core of our approach without getting lost in all the detail of the Fundamental Interfaces of DOM Core Level One. This presentation consists of a big-step operational semantics of the necessary operations of

Featherweight DOM and an implementation of the remaining operations in terms of those.

- Chapter 4 presents a context logic for Featherweight DOM, and demonstrates its use by reasoning about several example programs.
- Chapter 6 extends the work in Chapter 3 to cover all the Fundamental Interfaces of DOM Core Level One. The act of precisely specifying its behaviour uncovers a number of ambiguous or unclear descriptions in the W3C specification. As with Chapter 3, this chapter presents a core of operational semantics and implementations of remaining operations. It concludes with some example programs which demonstrate the features of the Fundamental Interfaces of DOM Core Level One.
- Chapter 7 presents a context logic for the Fundamental Interfaces of DOM Core Level One. As with Chapter 4, this logic is then used to reason about the example programs given in the previous chapter.
- Chapter 8 briefly describes the real world behaviour of key features of several DOM implementations and discusses some of the possible reasons for some common deviations from the specification.

1.3. Publications

While much of the work in this thesis is unpublished, other parts have previously appeared in the following publications:

- Gardner, Smith, Wheelhouse and Zarfaty. DOM: Towards a Formal Specification. Plan-X – a POPL Workshop, 2008[31]
- Gardner, Smith, Wheelhouse and Zarfaty. Local Hoare reasoning about DOM. Proceedings of the Twenty-Seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems: PODS, 2008[34]

  These papers introduced Featherweight DOM as it is presented here in Chapter 3. This was the first attempt to use local reasoning with context logic to reason about web programs.

- Gardner, Smith, Wright. Local Reasoning about Mashups. Theory workshop paper at the 3rd International Conference on Verified Software: Theories, Tools and Experiments 2010[32]

  This paper and the accompanying technical report use some of the techniques and specifications developed in this thesis, and apply them to the problem of reasoning about mashups.

2. Technical Background

This chapter describes the common notation and techniques used throughout the thesis. It introduces separation logic and context logic, which form the basis for the reasoning used for DOM.

The chapter starts with an introduction to program reasoning in Section 2.1. It begins with Hoare's seminal 1969 paper "An Axiomatic Basis for Computer Programming", and goes on to note several extensions made over the years by others. Section 2.2 introduces separation logic, which makes reasoning about pointer programs practical and has been successfully adapted to reasoning about Java programs. Section 2.3 introduces context logic, which is a generalisation of separation logic that enables us to reason about tree-like structures such as DOM.

2.1. Hoare Reasoning

In his 1969 paper "An Axiomatic Basis for Computer Programming"[40], Hoare laid the logical foundations for reasoning about the properties of computer programs. This paper took Floyd's treatment of flowcharts[29] and applied those notions to program texts, establishing the notation which has come to be known as the "Hoare Triple", the notion of a command axiom, and the first rules of inference with which one may make deductions about a program. Over the years since, this system has been refined and extended by the research community. It still remains recognisable as the deductive system described in [40].

The central concept of this system is the Hoare Triple[1] $\{P\}\mathtt{C}\{Q\}$. In this expression, P and Q are first-order logic predicates which describe the state of the program, and `C` is the program that the statement reasons about. The statement as a whole means "If the assertion P is true before initiation of

[1] In his 1969 paper, Hoare used the notation $P\{\mathtt{C}\}Q$, but the above notation has become standard.

a program C, then the assertion Q will be true on its completion". It must be noted that the triple does not guarantee successful termination of C. It may fail to terminate either "due to infinite loop; or ... due to violation of an implementation-defined limit". The particular implementation-defined limit we will typically be interested in here is a memory fault, which is caused when a program attempts to dereference an unallocated memory cell.

The practicality of Hoare's system in top-down program construction was demonstrated in papers such as his 1971 "Proof of a Program: FIND"[41]. In the years since, Hoare Reasoning has been re-used and extended in many ways. Here we are chiefly concerned with its use to reason about programs that destructively update data structures. In 1972 Burstall offered "Some techniques for proving correctness of programs which alter data structures"[9] which made use of what he called a "distinct nonrepeating tree system". The key intuition of this paper was that the nonrepeating tree system, written as $P_1 \& \ldots \& P_n$ contained n assertions P_i which each described a distinct region of storage. Thus, a single location could change only one P_i, and the complexity of reasoning about pointer programs could be managed.

In 1975, Dijkstra introduced the concept of "weakest preconditions"[20]. Dijkstra's weakest preconditions were inspired directly by Hoare's triples, with the chief difference that Dijkstra expected the programmer/logician to prove both the correctness of any result that the program might return, and also that the program would always terminate. He defined the weakest precondition $wp(S, R)$, where S denotes some statement list (or program) and R denotes some condition on the state of the system, to be "the weakest precondition for the initial state of the system such that activation of S is guaranteed to lead to a properly terminating activity leaving the system in a final state satisfying the postcondition R". Where Hoare would characterise a single command C with one or more axioms of the form $\{P\}\mathtt{C}\{Q\}$, Dijkstra would characterise that command with its most general weakest precondition. He would say "For any postcondition R, we have $wp(\mathtt{C}, R) = \ldots$". When we speak of "the weakest precondition of a command" without specifying a particular postcondition, it is this most general weakest precondition to which we are referring. While Dijkstra was not chiefly concerned with reasoning about pointer programs and destructive data update, his weakest

preconditions have become a standard tool in the program verification arsenal. As we will see, this concept of a weakest precondition, and in particular the most general weakest precondition for an arbitrary postcondition, turns out to be crucial in proving the completeness of later systems of program reasoning for loop-free code.

2.2. Separation Logic

In 2000, Reynolds[57] identified limitations in Burstall's techniques for proving programs which alter data structures. For example, for any given assertion $P_1 \ \& \ldots \& \ P_n$, the only allowed sharing was from variables into data fragments described by distinct P_i, or from one fragment P_i to another P_j. This meant that a given assertion $P_1 \ \& \ldots \& \ P_n$ could only describe structures with a fixed finite bound on the number of substructures. To overcome this and other limitations, Reynolds drew inspiration from Kripke's work[46] on intuitionistic logic and introduced "Intuitionistic Reasoning about Shared Mutable Data Structure"[57]. Similar intuitionistic semantics were developed independently by Ishtiaq and O'Hearn using the logic of bunched implication[52]. Ishtiaq and O'Hearn also devised a classical version of the logic which is more expressive. In particular, it can express storage deallocation. Reynolds then extended this classical version, adding pointer arithmetic, while O'Hearn and Yang introduced the concept of "small axioms". This early work by O'Hearn, Reynolds and Yang was distilled in what is now known as "separation logic"[58, 55, 59].

The key concept of separation logic is the notion of "local Hoare reasoning", whereby one reasons not about a program's affect on the whole state of a machine, but rather only about its affect on the state that is essential to its running. This is achieved by the use of a separating conjunction $*$ which is used in a similar way to the & of Burstall, and by a new inference rule called "the Frame Rule" which allows a local proof to be generalised to one which describes data that a program leaves unchanged. As we will see, the $*$ of separation logic does not require the rigid structure of Burstall's distinct nonrepeating tree system, and so can describe structures without a fixed finite bound on the number of substructures.

The remainder of this section briefly describes separation logic as it is applied to reasoning about a C-like language, with a shared heap.

2.2.1. The Heap

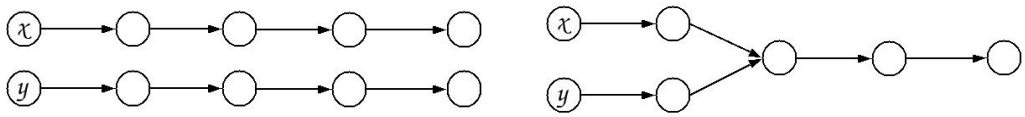
A key characteristic of any C-like language is a dynamically-allocated storage mechanism known as a "heap". A heap is a finite collection of cells, which are indexed by the positive natural numbers. Each cell stores a single value which can be interpreted simply as an integer, or as a pointer which either refers to another cell in the heap, or is the special value 0 which is not associated with any cell. If the cell indexed by x stores the value y, we write $x \mapsto y$. Cells which store multiple values can be emulated by using adjacent cells. The notation $x \mapsto y, z$ is commonly used to refer to a heap in which $x \mapsto y$ and $(x+1) \mapsto z$. The empty heap is referred to by the notation *emp*.

2.2.2. The Assertion Language

The assertion language of separation logic extends first-order logic with two new operators: separating conjunction "$*$", and separating implication "$-\!\!*$". Given our heap and C-like language, the assertion"$P*Q$" states that P holds over one portion of the heap and that Q hold over a separate, disjoint portion of the heap. Using this operator we can, for example, cleanly specify a linked list:

$$\begin{aligned} \text{list}(x) &\triangleq & x = 0 \\ \text{list}(x) &\triangleq \exists h, t. & x \mapsto h, t * \text{list}(t) \end{aligned}$$

Furthermore, we can cleanly specify a pair of disjoint lists $\text{list}(x) * \text{list}(y)$. Note that this predicate holds for disjoint lists such as those depicted on the left below, but not for lists with sharing such as those on the right:



The assertion "$P -\!\!* Q$", is satisfied by any heap h_0 such that for all heaps h_1 satisfying P, $h_0 * h_1$ satisfies Q. This is particularly useful in describing the weakest preconditions of commands.

2.2.3. Program Reasoning

Traditional Hoare triples $\{P\}\mathtt{C}\{Q\}$ are interpreted "loosely" – meaning that they provide no guarantees about any state not explicitly mentioned in

either P or Q. By contrast, the local Hoare reasoning of separation logic interprets these triples "tightly". This means that: the command `C` may only access state which is either described in P or explicitly allocated by `C` itself. The triple guarantees that all the state not mentioned by the predicates P and Q remains unchanged by the command `C`.

In order to reason about a program using tightly interpreted separation logic triples, that program must be composed entirely of "local" commands, as described in [71]. A command `C` is local if and only if it satisfies both the safety monotonicity property and the frame property. The safety-monotonicity property says that, if a heap contains all the cells necessary for safe execution of a command, then so does every larger heap. The frame property says that if a command can be executed safely on a small heap then execution on any larger heap can be traced back to the small state.

One advantage of local specifications is that they allow us to reason about a program in the same way as a programmer is likely to think informally about it. That is, by considering only that portion of the state that the program changes. To see how the specified program behaves in a larger state, we can apply the Frame Rule, as introduced in [55]:

$$\frac{\{P\}\texttt{C}\{Q\}}{\{R * P\}\texttt{C}\{R * Q\}}$$

where `C` modifies no variables that are free in R

Using tightly interpreted Hoare triples, we can specify commands "locally" – mentioning only the data that the command requires, and relying on the Frame Rule to scale the specification up to arbitrary states. Where Hoare provides command "Axioms", we may now provide "Small Axioms". This avoids the issues of aliasing which make non-local Hoare logic command axioms impractical to use with shared mutable data structures.

For example, we can use the "list" predicate from the previous section to specify a subroutine which disposes of a linked list:

$$\{\text{list}(x)\} \quad \texttt{disposeList}(x) \quad \{emp\}$$

The procedure "disposeList" acts only on the heap cells that form the linked list referred to in the precondition list(x). The postcondition states

that the heap is empty, which is to say that the list cells have been deallocated. Using this specification and the Frame Rule given above, we can derive a description of how the `disposeList` procedure behaves when there are two lists present in the heap.

$$\{\text{list}(x) * \text{list}(y)\} \quad \texttt{disposeList}(x) \quad \{\text{list}(y)\}$$

This example demonstrates two important points. Firstly, the use of $*$ in the Frame Rule (and hence in the derived local Hoare triple) precludes the possibility that the list beginning with cell x shares any elements with the list beginning with cell y. That is to say, there will be no unexpected behaviour due to aliasing between the two distinct portions of the heap. Secondly, neither of the lists in this example has a fixed finite bound on its number of substructures – the reasoning holds regardless of the length of the list.

While the examples given here have been low-level C-style examples, it is possible to use separation logic to reason about object systems such as the java object-heap [56, 21]. In addition, separation logic has been used to reason about resources and concurrency[51, 15], permission accounting[6] and information hiding[53]. It has also been married with other reasoning systems such as rely/guarantee[65] and cyclic proof[8]. Finally, separation logic reasoning has been automated in a number of tools including Smallfoot[4], Space Invader[63], SLAyer[61] and jStar[22]. Of particular note here are the first automatic proofs of pointer usage in entire industrial programs: Microsoft and Linux device drivers of up to 10,000 lines of code[70] and new techniques which pave the way to automatically verifying programs of over a million lines of code[12].

2.3. Context Logic

In 2005, Calcagno, Gardner and Zarfaty drew inspiration from the ambient logic of Cardelli and Gordon[16] to produce a generalisation of separation logic which was capable of elegantly handling tree structures[13, 72]. A key motivation for this work was the discovery that neither separation logic nor ambient logic were sufficient for local reasoning about tree programs. This insufficiency is formally described in [10] which introduces the concept

of "parametric expressivity". Previous results by Lozes had shown that for closed formulae, both the assertion languages of separation logic and of context logic were no more expressive than propositional logic. In [10], Calcagno et al. observed that in program reasoning it is often necessary to parameterise a formula. For example, when expressing the weakest precondition for the command `disposeList`, which is "$P * \mathrm{list}(x)$", the P stands for the arbitrary postcondition which we wish to satisfy after the execution of `disposeList`. [10] shows that when this kind of parametric formula is taken into account, context logic is indeed more expressive than previous systems.

Context logic can be applied to many kinds of structured data, as has been discussed in [14]. It is shown in [13] that if one applies context logic to a relatively simple data structure such as a heap the context logic collapses neatly into separation logic. Here, we are primarily interested in trees.

2.3.1. The Tree

In this thesis we will be using context logic to reason about complex tree-like structures which correspond to DOM data. This section illustrates context logic with the example of simple trees, defined as follows:

$$T ::= \varnothing \mid a[T] \mid T \otimes T$$

A tree is either empty, a node with a unique label and a subtree, or the collection of subtrees. The composition operator $\otimes$ is associative and non-commutative with unit $\varnothing$. The empty tree $\varnothing$ may be considered analogous to the empty heap.

This tree may be manipulated by a program just as a C program manipulates the heap. For reasoning about these manipulations, it is helpful to consider the natural contexts of these trees:

$$C ::= \_ \mid a[C] \mid C \otimes T \mid T \otimes C$$

Notice that each context must contain precisely one hole $\_$. There is a simple context application function $\mathrm{ap} : C \times T \rightharpoonup T$ which inserts a tree into the hole in a context, and returns the resulting tree.

2.3.2. The Assertion Language

The assertion language of context logic follows separation logic in extending first-order logic with new operators. Where separation logic introduced separating conjunction "$*$", context logic introduces separating application "$\circ$". Separating application is not symmetric, so in place of separation logic's "$-\!\!*$", context logic's "$\circ$" has two right-adjoints: "$-\!\!\circ$" and "$\circ\!\!-$".

The formula $K \circ P$ is satisfied by a tree if that tree can be split into a context satisfying K and disjoint tree satisfying P.

The formula $P -\!\!\circ Q$ is satisfied by a context c if, whenever a tree satisfying P is inserted into the hole in k, the resulting tree satisfies Q.

The formula $K \circ\!\!- P$ is satisfied by a tree t if, whenever it is inserted into a context satisfying K, the resulting tree satisfies P.

Unlike separation logic, context formula may be partitioned into two types: tree formulae and context formulae. The treatment above is typical in that it uses P and Q to refer to tree formulae, and K to refer to context formulae. Since there are two types of formulae, there are also two types of true: true_T is satisfied by any tree and true_C is satisfied by any context.

Using context logic, it is possible to specify complex ancestral relationships in a tree. For example, the formula:

$$(\varnothing -\!\!\circ (\text{true}_\text{C} \circ x[\text{true}_\text{T}])) \circ y[\text{true}_\text{T}]$$

This is a complex formula which demonstrates several features of the context logic assertion language, and also turns out to be a particularly useful shape for program reasoning in later chapters. It describes any tree which contains at least two nodes labeled x and y, in which y is not an ancestor of x.

The formula describes a context $(\varnothing -\!\!\circ (\text{true}_\text{C} \circ x[\text{true}_\text{T}]))$ applied to a tree containing only y and its children. The context is simply any context that contains a node x somewhere within it. Since a context may contain no nodes whose ancestors are the context hole we can be sure that after the context application, the node y is not an ancestor of x in the resulting tree.

2.3.3. Program Reasoning

As with separation logic, context logic may only be used to reason about programs which are local. The safety-monotonicity and frame properties are essentially the same as in separation logic, but re-stated in terms of contexts. Also as with separation logic we have the Frame Rule, re-stated to use separating application instead of separating conjunction:

$$\frac{\{P\}\mathtt{C}\{Q\}}{\{K \circ P\}\mathtt{C}\{K \circ Q\}}$$

We can use context logic to specify a command "move y under x" which is similar to the DOM command `appendChild`. This command takes y from wherever it is in the tree, and places it at the end of the list of x's children. If y is an ancestor of x, then this command should fail rather than attempt to create a structure with a loop, or an unreachable subtree. The axiom for this command is as follows:

$$\begin{gathered}\{(\varnothing \multimap (C \circ x[T])) \circ (y[T'])\} \\ \text{move } y \text{ under } x \\ \{C \circ x[T \otimes y[T']]\}\end{gathered}$$

The precondition is of the form discussed earlier, which specifies that `y` is not an ancestor of `x`. We use variables such as C and T to specify that the parts of the tree around x and y which we have examined do not change during the execution of the command.

As with separation logic, we can apply the Frame Rule to see how this command behaves in the presence of additional disjoint data:

$$\begin{gathered}\{z[w[\mathrm{true_T}] \otimes \_] \circ ((\varnothing \multimap (C \circ x[T])) \circ (y[T']))\} \\ \text{move } y \text{ under } x \\ \{z[w[\mathrm{true_T}] \otimes \_] \circ (C \circ x[T \otimes y[T']])\}\end{gathered}$$

It is these sorts of techniques which we will apply in future chapters to reasoning about DOM programs.

In this thesis, context logic is used to reason about trees and tree-like structures. However, it can be applied to a wide variety of data structures[14, 72]. There have been a number of developments in understanding context logic, including adjunct elimination[11], and completeness results[10]. There

has been some success in using context logic in conjunction with separation logic to relate high level specifications to low level implementations of those specifications[33, 30]. This thesis represents the first attempt to apply context logic in an industrial problem-space[34], and has provided the motivation for the most recent evolution of context logic: the introduction of segment structures which allow smaller specifications for move commands[35].

3. Featherweight DOM

This chapter describes "Featherweight DOM", which is a small XML update language designed by the author to demonstrate the conceptual core of DOM.

The W3C DOM specification is divided into a number of levels, of which the Level 1 is the most fundamental. The Level 1 specification is itself separated into two parts: Core, which "provides a low-level set of fundamental interfaces that can represent any structured document"; and HTML, which "provides additional, higher-level interfaces... to provide a more convenient view of an HTML document". Furthermore, the DOM Core specification presents "two somewhat different sets of interfaces to an XML/HTML document; one presenting an 'object-oriented' approach with a hierarchy of inheritance, and a 'simplified' view that allows all manipulation to be done via the Node interface".

Featherweight DOM captures the spirit of the Node interface of DOM Core Level 1 focusing on the tree structure and simple text nodes. Featherweight DOM does not deal directly with more complex DOM structure, such as Attributes, DocumentFragments and so on. For a more comprehensive and rigorously compliant treatment of DOM Core Level 1, see Chapter 6.

Featherweight DOM consists of two parts: The DOM library functionality of Featherweight DOM and the simple imperative language into which the DOM functionality is embedded. This chapter defines an abstract data structure, and presents the operational semantics for Featherweight DOM.

3.1. Featherweight DOM Data Structures

We introduce an abstract data structure to represent the XML-like data that Featherweight DOM programs manipulate. This data structure is given in Definition 2. Following the shape of this data structure, we also give the corresponding context structure in Definition 4. This is essential to

our reasoning and also helpful in describing the operational semantics for Featherweight DOM.

In the following structure, elements **ele** and text nodes **txt** represent their namesakes in [23]. The grove structure **g** may be regarded as analogous to the object heap, and forests **f** represent the contents of the NodeList object that contains the children of a given Element[1].

A DOM Element in turn corresponds to an XML element. A DOM text node is an artifact of the DOM choice to model all XML data in terms of nodes: XML documents may contain text at any point in their structure, and DOM models this by wrapping that text in a node which may be moved around and otherwise manipulated in the same way as, for example, an element node. A side effect of this choice is that the same XML document may have more than one DOM representation. For example, the XML ⟨p⟩`This is a paragraph`⟨/p⟩ contains the text "This is a paragraph". This text may be represented in DOM as a single text node containing the string "This is a paragraph", or as two adjacent text nodes containing the strings "This i" and "s a paragraph", or as three adjacent text nodes containing the strings "This i", "", "s a paragraph".

Definition 1 (Strings)**.** Given a finite set CHAR of text characters, with a distinguished character '#', strings $\mathbf{s} \in \mathrm{S}$ are defined by: (with $\mathbf{c} \in$ CHAR)

$$\text{strings} \in \mathrm{S} \qquad \mathbf{s} \qquad ::= \varnothing_{\mathrm{S}} \mid <\mathbf{c}>_{\mathrm{S}} \mid \mathbf{s} \otimes_{\mathrm{S}} \mathbf{s}$$

As a notational convenience, we will use the shorthand "abc" to refer to the string $<\text{'a'}>_{\mathrm{S}} \otimes_{\mathrm{S}} <\text{'b'}>_{\mathrm{S}} \otimes_{\mathrm{S}} <\text{'c'}>_{\mathrm{S}}$.

[1] We follow the DOM specification in that we write DOM type names in UpperCamelCase – hence "Element". When describing XML elements however, it is usual to use standard English capitalization.

Definition 2 (Featherweight Data)**.** Given an infinite set ID of node identifiers and the strings S defined in Definition 1, groves $\mathbf{g} \in \mathrm{G}$, elements $\mathbf{ele} \in \mathrm{ELE}$, forests $\mathbf{f} \in \mathrm{F}$, text nodes $\mathbf{txt} \in \mathrm{TXT}$, and are defined by: (with $\mathbf{id}, \mathbf{fid} \in \mathrm{ID}$)

$$\begin{array}{lrl} \text{groves} \in \mathrm{G} & \mathbf{g} & ::= \varnothing_{\mathrm{G}} \mid <\mathbf{ele}>_{\mathrm{G}} \mid <\mathbf{txt}>_{\mathrm{G}} \mid \mathbf{g} \oplus \mathbf{g} \\ \text{elements} \in \mathrm{ELE} & \mathbf{ele} & ::= \mathbf{s}_{\mathbf{id}}[\mathbf{f}]_{\mathbf{fid}} \text{ where } \text{`\#'} \notin \mathbf{s} \\ \text{forests} \in \mathrm{F} & \mathbf{f} & ::= \varnothing_{\mathrm{F}} \mid <\mathbf{ele}>_{\mathrm{F}} \mid <\mathbf{txt}>_{\mathrm{F}} \mid \mathbf{f} \otimes_{\mathrm{F}} \mathbf{f} \\ \text{text nodes} \in \mathrm{TXT} & \mathbf{txt} & ::= \text{“\#text”}_{\mathbf{id}}\mathbf{s} \end{array}$$

For well-formedness, the identifiers must be unique. The set of possible data types $\{\mathrm{G}, \mathrm{ELE}, \mathrm{F}, \mathrm{TXT}, \mathrm{S}\}$ is written $\mathcal{D}$. Given an arbitrary data type $\mathrm{D} \in \mathcal{D}$, $\mathbf{d} \in \mathrm{D}$ denotes arbitrary data, and $\mathbf{id} \notin \mathbf{d}$ states that the identifier $\mathbf{id}$ is not found in the data structure $\mathbf{d}$.

Definition 3 (Congruence)**.** The structural congruence $\equiv$ is the least equivalence relation on $\bigcup_{\mathrm{D} \in \mathcal{D}} \mathrm{D}$ satisfying:

$$\begin{array}{ll} \text{Associativity of } \oplus & \mathbf{g}_1 \oplus (\mathbf{g}_2 \oplus \mathbf{g}_3) \equiv (\mathbf{g}_1 \oplus \mathbf{g}_2) \oplus \mathbf{g}_3 \\ \text{Commutativity of } \oplus & \mathbf{g}_1 \oplus \mathbf{g}_2 \equiv \mathbf{g}_2 \oplus \mathbf{g}_1 \\ \text{Identity of } \oplus & \mathbf{g} \equiv \mathbf{g} \oplus \varnothing_{\mathrm{G}} \\ \text{Associativity of } \otimes_{\mathrm{F}} & \mathbf{f}_1 \otimes_{\mathrm{F}} (\mathbf{f}_2 \otimes_{\mathrm{F}} \mathbf{f}_3) \equiv (\mathbf{f}_1 \otimes_{\mathrm{F}} \mathbf{f}_2) \otimes_{\mathrm{F}} \mathbf{f}_3 \\ \text{Identity of } \otimes_{\mathrm{F}} & \mathbf{f} \equiv \mathbf{f} \otimes_{\mathrm{F}} \varnothing_{\mathrm{F}} \\ \text{Associativity of } \otimes_{\mathrm{S}} & \mathbf{s}_1 \otimes_{\mathrm{S}} (\mathbf{s}_2 \otimes_{\mathrm{S}} \mathbf{s}_3) \equiv (\mathbf{s}_1 \otimes_{\mathrm{S}} \mathbf{s}_2) \otimes_{\mathrm{S}} \mathbf{s}_3 \\ \text{Identity of } \otimes_{\mathrm{S}} & \mathbf{s} \equiv \mathbf{s} \otimes_{\mathrm{S}} \varnothing_{\mathrm{S}} \end{array}$$

Notice that text nodes **txt** use the string “#text” which contains the distinguished character ‘#’. Element nodes on the other hand may have an arbitrary string **s**, but that string must not contain the distinguished character ‘#’. This is as specified in the W3C specification[23].

The XML fragment mentioned earlier: ⟨p⟩`This is a paragraph`⟨/p⟩ can now be represented as the Featherweight DOM structure:

$$\text{“p”}_{\mathbf{id}}[<\text{“\#text”}_{\mathbf{id}'}\text{“This is a paragraph”}>_{\mathrm{F}}]_{\mathbf{fid}}$$

It can also be represented by the following, equally valid, Featherweight

DOM structure:

$$\text{“p”}_{\mathbf{id}}[<\text{“\#text”}_{\mathbf{id}'}\text{“This i”}>_{\mathrm{F}} \otimes_{\mathrm{F}} \text{“\#text”}_{\mathbf{id}''}\text{“s a paragraph”}>_{\mathrm{F}}]_{\mathbf{fid}}$$

Each of these featherweight DOM structures corresponds to a DOM structure allowed by [23]. These structures are not equivalent under $\equiv$, and they both accurately represent the XML fragment in question. The string associated with a Featherweight DOM element structure corresponds to the nodeName attribute of the corresponding DOM Node, and hence to the tag of the corresponding XML element. The string associated with Featherweight DOM text structures corresponds to the nodeValue attribute of the corresponding DOM Node, and hence to a portion of the text in the corresponding XML document. The **id** and **fid** values do not correspond to any part the XML data. They may be regarded as object addresses for DOM nodes in an object heap. Their purpose is to uniquely identify Featherweight DOM structures so that Featherweight DOM programs can refer to, and update them. The **id** corresponds to the heap-address of the Node object in question while the **fid** corresponds to the heap-address of the NodeList that contains the children of the Node object in question.

Definition 4 (Featherweight Contexts)**.** Given an infinite set ID of node identifiers and the data structures defined in Definition 2, grove contexts $\mathbf{cg} \in$ CG, element contexts $\mathbf{cele} \in$ CELE, forest contexts $\mathbf{cf} \in$ CF, text contexts $\mathbf{ctxt} \in$ CTXT, and string contexts $\mathbf{cs} \in$ CS are defined by:

grove contexts	$\mathbf{cg}$	$::= -_{\mathrm{G}} \mid <\mathbf{cele}>_{\mathrm{G}} \mid <\mathbf{ctxt}>_{\mathrm{G}} \mid \mathbf{cg} \oplus \mathbf{g}$
element contexts	$\mathbf{cele}$	$::= -_{\mathrm{ELE}} \mid \mathbf{s}_{\mathbf{id}}[\mathbf{cf}]_{\mathbf{fid}}$ where ‘#’ $\notin \mathbf{s}$
forest contexts	$\mathbf{cf}$	$::= -_{\mathrm{F}} \mid <\mathbf{cele}>_{\mathrm{F}} \mid <\mathbf{ctxt}>_{\mathrm{F}} \mid \mathbf{cf} \otimes_{\mathrm{F}} \mathbf{f} \mid \mathbf{f} \otimes_{\mathrm{F}} \mathbf{cf}$
text contexts	$\mathbf{ctxt}$	$::= -_{\mathrm{TXT}} \mid \text{“\#text”}_{\mathbf{id}}\text{“}\mathbf{cs}\text{”}$
string contexts	$\mathbf{cs}$	$::= -_{\mathrm{S}} \mid \mathbf{cs} \otimes_{\mathrm{S}} \mathbf{s} \mid \mathbf{s} \otimes_{\mathrm{S}} \mathbf{cs}$

As before, element names may not contain “#”, identifiers are unique and there is an analogous congruence $\equiv$. The set of possible context types $\{\mathrm{CG}, \mathrm{CELE}, \mathrm{CF}, \mathrm{CTXT}, \mathrm{CS}\}$ is written $\mathcal{CD}$.

Given data types $\mathrm{D}_1, \mathrm{D}_2 \in \mathcal{D}$ and a context type $\mathrm{CD}_2 \in \mathcal{CD}$ which corresponds to data type D_2, we sometimes write $\mathbf{cd}\colon \mathrm{D}_1 \to \mathrm{D}_2$ to denote a context $\mathbf{cd} \in \mathrm{CD}_2$ with hole $-_{\mathrm{D}_1}$. We call $\mathrm{D}_1 \to \mathrm{D}_2$ the context type of $\mathbf{cd}$.

$$
\begin{array}{lll}
\mathrm{ap}(-_{\mathrm{G}}, \mathbf{g}) & \triangleq \mathbf{g} & \\
\mathrm{ap}(<\mathbf{ct}>_{\mathrm{G}}, \mathbf{d}_1) & \triangleq <\mathrm{ap}(\mathbf{ct}, \mathbf{d}_1)>_{\mathrm{G}} & \\
\mathrm{ap}(\mathbf{cg} \oplus \mathbf{g}, \mathbf{d}_1) & \triangleq \mathrm{ap}(\mathbf{cg}, \mathbf{d}_1) \oplus \mathbf{g} & \\
 & & \\
\mathrm{ap}(-_{\mathrm{ELE}}, \mathbf{ele}) & \triangleq \mathbf{ele} & \\
\mathrm{ap}(\mathbf{s_{id}}[\mathbf{cf}]_{\mathbf{fid}}, \mathbf{d}_1) & \triangleq \mathbf{s_{id}}[\mathrm{ap}(\mathbf{cf}, \mathbf{d}_1)]_{\mathbf{fid}} & \text{if } \mathbf{id}, \mathbf{fid} \notin \mathbf{d}_1 \\
 & \text{undefined} & \text{otherwise} \\
 & & \\
\mathrm{ap}(-_{\mathrm{F}}, \mathbf{f}) & \triangleq \mathbf{f} & \\
\mathrm{ap}(<\mathbf{ct}>_{\mathrm{F}}, \mathbf{d}_1) & \triangleq <\mathrm{ap}(\mathbf{ct}, \mathbf{d}_1)>_{\mathrm{F}} & \\
\mathrm{ap}(\mathbf{cf} \otimes_{\mathrm{F}} \mathbf{f}, \mathbf{d}_1) & \triangleq \mathrm{ap}(\mathbf{cf}, \mathbf{d}_1) \otimes_{\mathrm{F}} \mathbf{f} & \\
\mathrm{ap}(\mathbf{f} \otimes_{\mathrm{F}} \mathbf{cf}, \mathbf{d}_1) & \triangleq \mathbf{f} \otimes_{\mathrm{F}} \mathrm{ap}(\mathbf{cf}, \mathbf{d}_1) & \\
 & & \\
\mathrm{ap}(-_{\mathrm{TXT}}, \mathbf{txt}) & \triangleq \mathbf{txt} & \\
\mathrm{ap}(\text{“\#text”}_{\mathtt{id}}\text{“}\mathbf{cs}\text{”}, \mathbf{s}) & \triangleq \text{“\#text”}_{\mathtt{id}}\text{“ ap}(\mathbf{cs}, \mathbf{s})\text{ ”} & \\
 & & \\
\mathrm{ap}(-_{\mathrm{S}}, \mathbf{s}) & \triangleq \mathbf{s} & \\
\mathrm{ap}(\mathbf{cs} \otimes_{\mathrm{S}} \mathbf{s}', \mathbf{s}) & \triangleq \mathrm{ap}(\mathbf{cs}, \mathbf{s}) \otimes_{\mathrm{S}} \mathbf{s}' & \\
\mathrm{ap}(\mathbf{s}' \otimes_{\mathrm{S}} \mathbf{cs}, \mathbf{s}) & \triangleq \mathbf{s}' \otimes_{\mathrm{S}} \mathrm{ap}(\mathbf{cs}, \mathbf{s}) &
\end{array}
$$

Notice that ap is a partial function. The notation $\mathrm{ap}(\mathbf{cd}, \mathbf{d})\downarrow$ states that $\mathrm{ap}(\mathbf{cd}, \mathbf{d})$ is defined.

Figure 3.1.: Context Application

We define the partial application function ap : $(D_1 \to D_2) \times D_1 \rightharpoonup D_2$, which returns a result if there is no clash of identifiers between the arguments of the function.

Definition 5 (Context Application)**.** Given data types $D_1, D_2 \in \mathcal{D}$, the partial application function ap : $(D_1 \to D_2) \times D_1 \rightharpoonup D_2$ is defined by induction on the structure of the first argument in Figure 3.1 (and undefined where not given). The notation $\mathrm{ap}(\mathbf{cd}, \mathbf{d})\downarrow$ states that $\mathrm{ap}(\mathbf{cd}, \mathbf{d})$ is defined.

Using the application function we can represent the XML fragment ⟨p⟩`This is a paragraph`⟨/p⟩ as a pair of disjoint Featherweight DOM structures:

$$\mathrm{ap}(\text{“p”}_{\mathbf{id}}[-_{\mathrm{F}}]_{\mathbf{fid}}, <\text{“\#text”}_{\mathbf{id}'}\ \text{“This is a paragraph”}>_{\mathrm{F}})$$

Here the application function has been used to isolate a text node from its parent. As we shall see, this is particularly useful if we wish to define a command which operates in the same way on the text node regardless of its context.

3.2. A Simple Imperative Language

At the core of Featherweight DOM is the simple imperative programming language into which the DOM library is embedded. This language is as simple as possible, since our focus here is on the DOM library. The language is dynamically typed and consists of sequential composition, assignment, conditional and loop commands and a simple procedure system with dynamic scope. These design decisions serve to both keep the language simple so that we may focus on reasoning about the Featherweight DOM library, and also to support possible future development in the direction of JavaScript.

3.2.1. Program State

As normal for in-place update, our language depends on a variable store s. To distinguish program variables from expressions in our reasoning, program variable names will always be written in lower case.

Definition 6 (The Variable Store)**.** Given the special value **null** and the sets of identifiers ID, strings S, integers $\mathbb{Z}$ and booleans $\mathbb{B}$, a store is a finite

partial function from variables to values:
s:$\mathrm{Var}_{\mathrm{PROG}} \rightharpoonup (\{\mathbf{null}\} \cup \mathrm{ID} \cup \mathrm{S} \cup \mathbb{Z} \cup \mathbb{B})$

Variable lookup of a variable `var` in a store s is written $s(\texttt{var})$. The notation $[s|\texttt{var} \leftarrow \mathbf{v}]$ describes an updated store, which differs from an existing store s only in that the variable `var` has the value $\mathbf{v}$. If the existing store s contains a variable `var`, then that variable is overwritten in the new store. If the existing store s does not contain a variable `var`, then the new store is extended with that variable. The notation $[s \setminus \texttt{var}]$ describes an updated store which differs from an existing store s only in that the variable `var` has been removed.

The special value **null** may be thought of as a distinguished identifier value which refers to no data structure. As we shall see, there are a number of DOM commands which always return either an identifier in ID or the special value **null**.

3.2.2. Expressions

We introduce expressions `Expr` $\in$ Exp which do not alter the program state. Due to the dynamically typed nature of this language, the values of some syntactically correct expressions are undefined. If a program command attempts to evaluate such an expression, it will fault. To distinguish expressions from variables in our reasoning, expression names will always be written in UpperCamelCase.

Definition 7 (Expressions)**.** Given the special value **null**, the empty string $\varnothing_{\mathrm{S}}$, characters $\mathbf{c} \in$ CHAR, integers $\mathbf{n} \in \mathbb{Z}$, booleans true and false, and variables `var` $\in \mathrm{Var}_{\mathrm{PROG}}$, expressions `Expr` $\in$ Exp are defined by:

`Expr` ::=	**null** $\mid \varnothing_{\mathrm{S}} \mid <\mathbf{c}>_{\mathrm{S}} \mid \mathbf{n} \mid$ true $\mid$ false	literal constants
	$\mid$ `var`	variables
	$\mid$ `Expr` $=$ `Expr`	equality test
	$\mid$ `Expr` $\otimes_{\mathrm{S}}$ `Expr`	string concatenation
	$\mid$ len(`Expr`)	string length
	$\mid$ `Expr` $+$ `Expr` $\mid$ `Expr` $-$ `Expr` $\mid$ `Expr` $\times$ `Expr` $\mid$ `Expr` $\div$ `Expr`	arithmetic operations
	$\mid$ `Expr` $\wedge$ `Expr` $\mid$ `Expr` $\vee$ `Expr` $\mid \neg$`Expr`	boolean operations

$$
\begin{aligned}
[\![\mathbf{null}]\!]_s &\triangleq \mathbf{null} \\
[\![\varnothing_{\mathrm{S}}]\!]_s &\triangleq \varnothing_{\mathrm{S}} \\
[\![<\mathbf{c}>_{\mathrm{S}}]\!]_s &\triangleq <\mathbf{c}>_{\mathrm{S}} \\
[\![\mathbf{n}]\!]_s &\triangleq \mathbf{n} \\
[\![\mathrm{true}]\!]_s &\triangleq \mathrm{true} \\
[\![\mathrm{false}]\!]_s &\triangleq \mathrm{false} \\
[\![\mathtt{var}]\!]_s &\triangleq s(\mathtt{var}) \text{ iff } \mathtt{var} \in \mathrm{dom}(s) \\
[\![\mathtt{Expr} = \mathtt{Expr}']\!]_s &\triangleq [\![\mathtt{Expr}]\!]_s = [\![\mathtt{Expr}']\!]_s \\
[\![\mathtt{Expr} \otimes_{\mathrm{S}} \mathtt{Expr}']\!]_s &\triangleq [\![\mathtt{Expr}]\!]_s \otimes_{\mathrm{S}} [\![\mathtt{Expr}']\!]_s \text{ iff } [\![\mathtt{Expr}]\!]_s, [\![\mathtt{Expr}']\!]_s \in \mathrm{S} \\
[\![\mathrm{len}(\mathtt{Expr})]\!]_s &\triangleq \mathrm{len}([\![\mathtt{Expr}]\!]_s) \\
[\![\mathtt{Expr} + \mathtt{Expr}']\!]_s &\triangleq [\![\mathtt{Expr}]\!]_s + [\![\mathtt{Expr}']\!]_s \text{ iff } [\![\mathtt{Expr}]\!]_s, [\![\mathtt{Expr}']\!]_s \in \mathbb{Z} \\
[\![\mathtt{Expr} - \mathtt{Expr}']\!]_s &\triangleq [\![\mathtt{Expr}]\!]_s - [\![\mathtt{Expr}']\!]_s \text{ iff } [\![\mathtt{Expr}]\!]_s, [\![\mathtt{Expr}']\!]_s \in \mathbb{Z} \\
[\![\mathtt{Expr} \times \mathtt{Expr}']\!]_s &\triangleq [\![\mathtt{Expr}]\!]_s \times [\![\mathtt{Expr}']\!]_s \text{ iff } [\![\mathtt{Expr}]\!]_s, [\![\mathtt{Expr}']\!]_s \in \mathbb{Z} \\
[\![\mathtt{Expr} \div \mathtt{Expr}']\!]_s &\triangleq [\![\mathtt{Expr}]\!]_s \div [\![\mathtt{Expr}']\!]_s \text{ iff } [\![\mathtt{Expr}]\!]_s, [\![\mathtt{Expr}']\!]_s \in \mathbb{Z} \\
[\![\mathtt{Expr} \wedge \mathtt{Expr}']\!]_s &\triangleq [\![\mathtt{Expr}]\!]_s \wedge [\![\mathtt{Expr}']\!]_s \text{ iff } [\![\mathtt{Expr}]\!]_s, [\![\mathtt{Expr}']\!]_s \in \mathbb{B} \\
[\![\mathtt{Expr} \vee \mathtt{Expr}']\!]_s &\triangleq [\![\mathtt{Expr}]\!]_s \vee [\![\mathtt{Expr}']\!]_s \text{ iff } [\![\mathtt{Expr}]\!]_s, [\![\mathtt{Expr}']\!]_s \in \mathbb{B} \\
[\![\neg \mathtt{LExpr}]\!]_s &\triangleq \neg [\![\mathtt{LExpr}]\!]_s \text{ iff } [\![\mathtt{LExpr}]\!]_s \in \mathbb{B}
\end{aligned}
$$

Note that this function is partial, and therefore that not all syntactically correct expressions can be evaluated. Attempting to evaluate such an expression will result in a runtime fault.

Figure 3.2.: Expression Evaluation

In order to define our expression evaluation, we require the auxiliary function len.

Definition 8 (Forest Length)**.** The length of a string or forest is defined by:

$$
\begin{aligned}
\mathrm{len}(\varnothing_{\mathrm{D}}) &\triangleq 0 \\
\mathrm{len}(<\mathbf{d}>_{\mathrm{D}}) &\triangleq 1 \\
\mathrm{len}(\mathbf{d}_1 \otimes_{\mathrm{D}} \mathbf{d}_2) &\triangleq \mathrm{len}(\mathbf{d}_1) + \mathrm{len}(\mathbf{d}_2)
\end{aligned}
$$

where $\mathrm{D} \in \{\mathrm{F}, \mathrm{S}\}$, and $\mathbf{d}_1, \mathbf{d}_2 \in \mathrm{D}$, and $\mathbf{d} \in \mathrm{ELE} \cup \mathrm{TXT} \cup \mathrm{C}$

Definition 9 (Expression Evaluation)**.** The evaluation of expression `Expr` in store s is defined by the partial function given in Figure 3.2.

Note that not all syntactically correct expressions can be successfully evaluated. Attempting to evaluate such an expression at runtime will result in a runtime fault.

3.2.3. Standard Imperative Commands

The core of the Featherweight DOM language is a set of imperative commands that the reader will be familiar with from many other languages.

Definition 10 (Featherweight DOM Commands)**.** The commands of the Featherweight DOM language are:

$\mathtt{C} ::= \mathtt{var} := \mathtt{Expr}$	assignment
$\mid \mathtt{C}; \mathtt{C}$	sequential composition
$\mid \mathtt{if\ Expr\ then\ C\ else\ C\ fi}$	conditionals
$\mid \mathtt{while\ Expr\ do\ C\ od}$	loops
$\mid \mathtt{local\ var} : \mathtt{C\ endloc}$	local variable declaration
$\mid \mathtt{v}{:=}\mathtt{procname}(\overline{\mathtt{params}}) \triangleq \mathtt{C}$	procedure declaration
$\mid \mathtt{v}{:=}\mathtt{procname}(\overline{\mathtt{vars}})$	procedure call
$\mid \mathtt{C}_{\mathrm{DOM}}$	DOM commands (see Section 3.3)

We deal with procedures in Section 3.2.4 and the DOM commands in Section 3.3. We give big-step operational semantics for the remaining commands here.

Definition 11 (Imperative Commands)**.** The operational semantics of Featherweight DOM's standard imperative commands are given in Figure 3.3 by an evaluation relation $\rightsquigarrow$ relating configuration triples $s, \mathbf{g}, \mathtt{C}$, terminal states $s, \mathbf{g}$, and faults.

For convenience, we also have the following simple syntactic sugars.
An else-less if:

$$\mathtt{if\ Expr\ then\ C\ fi} \triangleq \mathtt{if\ Expr\ then\ C\ else\ skip\ fi}$$

$$\frac{[\![\texttt{Expr}]\!]_s = \mathbf{val}}{s, \mathbf{g}, \texttt{var:=Expr} \rightsquigarrow [s|\texttt{var} \leftarrow \mathbf{val}], \mathbf{g}} \qquad \frac{s, \mathbf{g}, \texttt{C}_1 \rightsquigarrow s', \mathbf{g}' \quad s', \mathbf{g}', \texttt{C}_2 \rightsquigarrow s'', \mathbf{g}''}{s, \mathbf{g}, (\texttt{C}_1 \texttt{ ; } \texttt{C}_2) \rightsquigarrow s'', \mathbf{g}''}$$

$$\frac{[\![\texttt{Expr}]\!]s = \text{true} \quad s, \mathbf{g}, \texttt{C}_1 \rightsquigarrow s', \mathbf{g}'}{s, \mathbf{g}, \texttt{if Expr then C}_1 \texttt{ else C}_2 \texttt{ fi} \rightsquigarrow s', \mathbf{g}'}$$

$$\frac{[\![\texttt{Expr}]\!]s = \text{false} \quad s, \mathbf{g}, \texttt{C}_2 \rightsquigarrow s', \mathbf{g}'}{s, \mathbf{g}, \texttt{if Expr then C}_1 \texttt{ else C}_2 \texttt{ fi} \rightsquigarrow s', \mathbf{g}'}$$

$$\frac{s, \mathbf{g}, \texttt{if Expr then C ; while Expr do C od else skip fi} \rightsquigarrow s', \mathbf{g}'}{s, \mathbf{g}, \texttt{while Expr do C od} \rightsquigarrow s', \mathbf{g}'}$$

$$\frac{\mathbf{val} = s(\texttt{var}) \quad [s|\texttt{var} \leftarrow \mathbf{null}], \mathbf{g}, \texttt{C} \rightsquigarrow s', \mathbf{g}'}{s, \mathbf{g}, \texttt{local var : C endloc} \rightsquigarrow [s'|\texttt{var} \leftarrow \mathbf{val}], \mathbf{g}'}$$

$$\frac{\texttt{var} \notin \text{dom}(s) \quad [s|\texttt{var} \leftarrow \mathbf{null}], \mathbf{g}, \texttt{C} \rightsquigarrow s', \mathbf{g}'}{s, \mathbf{g}, \texttt{local var : C endloc} \rightsquigarrow [s' \setminus \texttt{var}], \mathbf{g}'}$$

$$\frac{}{s, \mathbf{g}, \texttt{skip} \rightsquigarrow s, \mathbf{g}}$$

Figure 3.3.: Imperative Commands

We may also group local scope declarations:

```
local x, y, ··· z :        ≜      local x :
    C                                 local y :
endloc                                    ···
                                          local z :
                                              C
                                          endloc
                                  endloc
                              endloc
```

3.2.4. Procedures

We provide a primitive procedure system with dynamic scope. Procedures are defined statically for each program with the following syntax:

$$\texttt{v:=procname}(\overline{\texttt{params}}) \triangleq \texttt{C}$$

where `procname` is the name of the procedure, `v` is a variable, $\overline{\texttt{params}}$ is

a vector of variables and C is the body of the procedure.

Definition 12 (procs)**.** Procedures are represented during program execution as a partial function procs which maps procedure names procname to the tuple $(\mathtt{v}, \overline{\mathtt{params}}, \mathtt{C})$ which contains the return variable v, a vector of the procedure's parameter variables $\overline{\mathtt{params}}$ and the procedure's body C.

Procedure call

Given a partial function procs taking procedure names to procedures, the semantics of procedure call are given by the following rule:

$$\frac{\begin{array}{l}\mathtt{procs}(\mathtt{procname}) = (\mathtt{v}', \overline{\mathtt{params}}, \mathtt{C}) \\ |\overline{\mathtt{params}}| = |\overline{\mathtt{vars}}| \\ \overline{\mathtt{params}} = \mathtt{p}_1 \ldots \mathtt{p}_n \\ \overline{\mathtt{vars}} = \mathtt{v}_1 \ldots \mathtt{v}_n \\ s, \mathbf{g}, \mathtt{C}\{\mathtt{v}/\mathtt{v}', \mathtt{v}_1/\mathtt{p}_1, \ldots, \mathtt{v}_n/\mathtt{p}_n\} \rightsquigarrow s', \mathbf{g}'\end{array}}{s, \mathbf{g}, \mathtt{v}{:=}\mathtt{procname}(\overline{\mathtt{vars}}) \rightsquigarrow s', \mathbf{g}'}$$

where $\overline{\mathtt{vars}}$ is a vector of variables.

Note that procedure call is only defined when the number of parameter expressions matches the number of parameter variables.

Ignoring Return Values and Void Procedures

Since there are circumstances in which one might wish to ignore the return value of a procedure, we define syntactic sugars and a special variable for doing so.

Definition 13 (The devnull variable)**.** The distinguished variable devnull behaves exactly like any other variable, except that no user-defined program may mention it. It exists solely for use in the following syntactic sugars.

When we wish to ignore the return value of a procedure or DOM command "var:=command($\overline{\mathtt{vars}}$)", we may use the syntactic sugar:

$$\mathtt{command}(\overline{\mathtt{vars}})$$

which expands to:

$$\begin{array}{l}\texttt{local devnull :}\\ \quad\texttt{devnull:=command}(\overline{\texttt{vars}})\\ \texttt{endloc}\end{array}$$

Since the `devnull` variable is not mentioned in any user-defined program, the only effect of this syntactic sugar is to calculate the return value of the procedure as normal, and then to ignore it.

When we wish to define a procedure which returns no value, we may use the syntactic sugar:

$$\texttt{procname}(\overline{\texttt{params}}) \triangleq \texttt{C}$$

which expands to:

$$\texttt{devnull:=procname}(\overline{\texttt{params}}) \triangleq \texttt{C}$$

Since the `devnull` variable is not mentioned in any user-defined program (and in particular, is not mentioned in `C`), the effect of this syntactic sugar is to produce a procedure which has no effect on its return variable. The procedure is therefore most naturally called using the syntactic sugar for ignoring a return value.

3.3. Featherweight DOM Commands

The essential character of Featherweight DOM is found not in its simple imperative core, but in the DOM library Commands embedded in the simple imperative language. The DOM library commands are defined in this section.

Recall that we are compiling the essence of the Node interface defined in [23]. That interface defines a number of commands of which we specify an essential subset. Since that interface makes extensive use of NodeList objects, we also specify commands from the NodeList interface. In addition to these, we also specify two commands from the Document interface: `createElement` and `createTextNode`. We require these two commands in order to create new nodes in our data structure. Finally, in order to make it possible to write more interesting example programs, we specify some

commands from the CharacterData interface. We then use the resulting language to implement the remaining Node interface commands.

Since we are embedding the Featherweight DOM Commands in a simple imperative language rather than an object oriented language, we do not insist that all commands (for example, `appendChild`) return a value if this would result in them simply returning one of their arguments unchanged. This simplifies our Featherweight presentation without sacrificing functionality. For a more slavish adherence to the letter of the W3C DOM Specification, see Chapter 6.

Recall the standard imperative commands `C` defined in Section 3.2.3. Here, we extend that language with DOM commands.

Definition 14 (DOM Library Commands)**.** The DOM library commands for Featherweight DOM are:

$$
\begin{aligned}
\texttt{C}_{\text{DOM}} ::= \ & \texttt{appendChild}(\texttt{parent}, \texttt{newChild}) \\
| \ & \texttt{removeChild}(\texttt{parent}, \texttt{oldChild}) \\
| \ & \texttt{name} := \texttt{getNodeName}(\texttt{node}) \\
| \ & \texttt{id} := \texttt{getParentNode}(\texttt{node}) \\
| \ & \texttt{fid} := \texttt{getChildNodes}(\texttt{node}) \\
| \ & \texttt{node} := \texttt{createElement}(\texttt{Name}) \\
| \ & \texttt{node} := \texttt{createTextNode}(\texttt{Str}) \\
| \ & \texttt{node} := \texttt{item}(\texttt{list}, \texttt{Int}) \\
| \ & \texttt{str}{:=}\texttt{substringData}(\texttt{node}, \texttt{Offset}, \texttt{Count}) \\
| \ & \texttt{appendData}(\texttt{node}, \texttt{Arg}) \\
| \ & \texttt{deleteData}(\texttt{node}, \texttt{Offset}, \texttt{Count})
\end{aligned}
$$

The DOM commands have the following behaviour and requirements:

`appendChild (parent, newChild)` moves the tree `newChild` to the end of `parent`'s child list. It requires that `parent` is an element node, and that `newChild` is an element and is not an ancestor of `parent`.

`removeChild (parent,oldChild)` removes the node `oldChild` and its subtree from `parent`'s child forest and re-inserts it at the root of the grove. It requires that `parent` is an element and `oldChild` is a child of `parent`.

`name := getNodeName (node)` assigns to the variable `var` the nodeName

value of `node`. It requires that `node` is either a text node or an element. If `node` is a text node, then `var` is set to "#text".

`var := getParentNode (node)` assigns to the variable `var` the identifier of the parent of `node`, if it exists, and **null** otherwise. It requires that `node` is either a text node or an element.

`var := getChildNodes (node)` assigns to the variable `var` the identifier of the child forest of the element `node`. It requires that `node` is an element. We differ slightly from the letter of the W3C specification in that we do not require this command to return a reference to an empty NodeList when it is called on a text node. This simplifies our Featherweight presentation without sacrificing functionality. For a more rigid treatment of the W3C specification, see the definition of `getChildNodes` in Chapter 6.

`var := createElement (Name)` creates a new element node at the root of the grove, with fresh **id** and **fid** and a name equal to `Name`. The identifier **id** of the new node is recorded in the variable `var`. It requires that `Name` is a string and that $\# \notin [\![\texttt{Name}]\!]_s$.

`var := createTextNode (Str)` creates a new text node at the root of the grove, with fresh **id** and a value equal to `Str`. The identifier **id** of the new node is recorded in the variable `var`. It requires that `Str` is a string.

`var := item (list, Int)` sets the variable `var` to the (`Int` + 1)th node in the list pointed to by `list`, setting it to **null** if `Int` evaluates to an invalid index. It faults if `list` does not correspond to the `fid` of an existing structure or if `Int` is not an integer.

`var := substringData (node,Offset,Count)` assigns to the variable `var` a substring of the string of the text node `node`. That substring will be of length `Count` and will start with (`Offset`+1)th character of the string of the text node. If `Offset`+`Count` exceeds the string length, then all the characters to the string end are returned. This command requires that `node` is a text node, that `Offset` and `Count` be non-negative integers, and that `Offset` be at most the string length.

`appendData(node, Arg)` appends the string `Arg` to the end of the string contained in `node`. It requires that `node` exists and be a text node and that `Arg` evaluate to a string.

`deleteData(node, Offset, Count)` deletes a substring of length `Count` starting with the (`Offset` + 1)th character of the value of the text node referred to by `node`. `Count`. If `Offset` + `Count` exceeds the string length, then all the characters to the string end are deleted. This command requires that `node` is a text node, that `Offset` and `Count` be non-negative integers, and that `Offset` be at most the string length.

`var := createTextNode(Str)` creates a new text node at the grove level, with fresh **id** and the string contained within the text node set to `Str`. It records the new node's identifier in the variable `var` and requires that `Str` evaluate to a string.

Definition 15 (Semantics of The DOM Library Commands)**.** The operational semantics of these Featherweight DOM commands are given in Figures 3.4 and 3.5 by an evaluation relation $\rightsquigarrow$ relating configuration triples $s, \mathbf{g}, \mathtt{C}$, terminal states $s, \mathbf{g}$, and faults.

Featherweight DOM is "minimal" in the sense that each specified DOM command cannot be implemented in terms of the remaining commands. Those Node interface commands which are not specified as part of Featherweight DOM can be implemented in terms of those which are. These implementations may be found in Appendix A.1.

$$\frac{\begin{pmatrix}\mathbf{t} \equiv \mathbf{name}'_{s(\texttt{newChild})}[\mathbf{f}']_{\mathbf{fid}'} \\ \vee\, \mathbf{t} \equiv \text{“\#text”}_{s(\texttt{newChild})}\,\text{“}\mathbf{str}\text{”}\end{pmatrix} \quad \begin{array}{c}\mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}', <\mathbf{t}>_{\mathrm{G}}) \\ \mathrm{ap}(\mathbf{cg}', \varnothing_{\mathrm{G}}) \equiv \mathrm{ap}(\mathbf{cg}, \mathbf{name}_{s(\texttt{parent})}[\mathbf{f}]_{\mathbf{fid}}) \\ \mathbf{g}' \equiv \mathrm{ap}(\mathbf{cg}, \mathbf{name}_{s(\texttt{parent})}[\mathbf{f} \otimes_{\mathrm{F}} <\mathbf{t}>_{\mathrm{F}}]_{\mathbf{fid}})\end{array}}{s, \mathbf{g}, \texttt{appendChild}(\texttt{parent}, \texttt{newChild}) \rightsquigarrow s, \mathbf{g}'}$$

$$\frac{\begin{pmatrix}\mathbf{t} \equiv \mathbf{name}'_{s(\texttt{newChild})}[\mathbf{f}']_{\mathbf{fid}'} \\ \vee\, \mathbf{t} \equiv \text{“\#text”}_{s(\texttt{newChild})}\,\text{“}\mathbf{str}\text{”}\end{pmatrix} \quad \begin{array}{c}\mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}', <\mathbf{t}>_{\mathrm{F}}) \\ \mathrm{ap}(\mathbf{cg}', \varnothing_{\mathrm{F}}) \equiv \mathrm{ap}(\mathbf{cg}, \mathbf{name}_{s(\texttt{parent})}[\mathbf{f}]_{\mathbf{fid}}) \\ \mathbf{g}' \equiv \mathrm{ap}(\mathbf{cg}, \mathbf{name}_{s(\texttt{parent})}[\mathbf{f} \otimes_{\mathrm{F}} <\mathbf{t}>_{\mathrm{F}}]_{\mathbf{fid}})\end{array}}{s, \mathbf{g}, \texttt{appendChild}(\texttt{parent}, \texttt{newChild}) \rightsquigarrow s, \mathbf{g}'}$$

$$\frac{\begin{array}{c}(\mathbf{t} \equiv \mathbf{name}'_{s(\texttt{oldChild})}[\mathbf{f}]_{\mathbf{fid}} \vee \mathbf{t} \equiv \text{“\#text”}_{s(\texttt{oldChild})}\,\text{“}\mathbf{str}\text{”}) \\ \mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}, \mathbf{name}_{s(\texttt{parent})}[\mathbf{f}_1 \otimes_{\mathrm{F}} <\mathbf{t}>_{\mathrm{F}} \otimes_{\mathrm{F}} \mathbf{f}_2]'_{\mathbf{fid}}) \\ \mathbf{g}' \equiv \mathrm{ap}(\mathbf{cg}, \mathbf{name}_{s(\texttt{parent})}[\mathbf{f}_1 \otimes_{\mathrm{F}} \mathbf{f}_2]'_{\mathbf{fid}}) \oplus <\mathbf{t}>_{\mathrm{G}}\end{array}}{s, \mathbf{g}, \texttt{removeChild}, (\texttt{parent}, \texttt{oldChild}) \rightsquigarrow s, \mathbf{g}'}$$

$$\frac{\mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}, \mathbf{name}_{s(\texttt{node})}[\mathbf{f}]_{\mathbf{fid}})}{s, \mathbf{g}, \texttt{var} := \texttt{getNodeName}(\texttt{node}) \rightsquigarrow [s|\texttt{var} \leftarrow \mathbf{name}], \mathbf{g}}$$

$$\frac{\mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}, \text{“\#text”}_{s(\texttt{node})}\,\text{“}\mathbf{str}\text{”})}{s, \mathbf{g}, \texttt{var} := \texttt{getNodeName}(\texttt{node}) \rightsquigarrow [s|\texttt{var} \leftarrow \text{“\#text”}], \mathbf{g}}$$

$$\frac{\begin{pmatrix}\mathbf{t} \equiv \mathbf{name}'_{s(\texttt{node})}[\mathbf{f}]'_{\mathbf{fid}} \\ \vee\, \mathbf{t} \equiv \text{“\#text”}_{s(\texttt{node})}\,\text{“}\mathbf{str}\text{”}\end{pmatrix} \quad \mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}, \mathbf{name}_{\mathbf{id}}[\mathbf{f}_1 \otimes_{\mathrm{F}} <\mathbf{t}>_{\mathrm{F}} \otimes_{\mathrm{F}} \mathbf{f}_2]_{\mathbf{fid}})}{s, \mathbf{g}, \texttt{var} := \texttt{getParentNode}(\texttt{node}) \rightsquigarrow [s|\texttt{var} \leftarrow \mathbf{id}], \mathbf{g}}$$

$$\frac{\begin{pmatrix}\mathbf{t} \equiv \mathbf{name}'_{s(\texttt{node})}[\mathbf{f}]'_{\mathbf{fid}} \\ \vee\, \mathbf{t} \equiv \text{“\#text”}_{s(\texttt{node})}\,\text{“}\mathbf{str}\text{”}\end{pmatrix} \quad \mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}, <\mathbf{t}>_{\mathrm{G}})}{s, \mathbf{g}, \texttt{var} := \texttt{getParentNode}(\texttt{node}) \rightsquigarrow [s|\texttt{var} \leftarrow \mathbf{null}], \mathbf{g}}$$

$$\frac{\mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}, \mathbf{name}_{s(\texttt{node})}[\mathbf{f}]_{\mathbf{fid}})}{s, \mathbf{g}, \texttt{var} := \texttt{getChildNodes}(\texttt{node}) \rightsquigarrow [s|\texttt{var} \leftarrow \mathbf{fid}], \mathbf{g}}$$

Figure 3.4.: Semantics of Featherweight DOM Commands (Part 1)

$$\frac{\# \notin [\![\texttt{Name}]\!]_s \quad \mathbf{g}' \equiv \mathbf{g} \oplus <[\![\texttt{Name}]\!]_s{}_{\mathbf{node}}[\varnothing_{\mathrm{F}}]_{\mathbf{fid}}>_{\mathrm{G}} \quad \mathbf{node},\mathbf{fid} \text{ fresh}}{s, \mathbf{g}, \texttt{var} := \texttt{createElement}(\texttt{Name}) \rightsquigarrow [s|\texttt{var} \leftarrow \mathbf{node}], \mathbf{g}'}$$

$$\frac{\mathbf{g}' \equiv \mathbf{g} \oplus <\text{“\#text”}_{\mathbf{node}}[\![\texttt{Str}]\!]_s>_{\mathrm{G}} \quad \mathbf{node} \text{ fresh}}{s, \mathbf{g}, \texttt{var} := \texttt{createTextNode}(\texttt{Str}) \rightsquigarrow s[\texttt{var} \leftarrow \mathbf{node}], \mathbf{g}'}$$

$$\frac{\begin{array}{c}(\mathbf{t} \equiv \mathbf{name}'_{\mathbf{node}}[\mathbf{f}]_{\mathbf{fid}} \vee \mathbf{t} \equiv \text{“\#text”}_{\mathbf{node}}\text{“}\mathbf{str}\text{”}) \\ \mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}, \mathbf{name}_{\mathbf{id}}[\mathbf{f}_1 \otimes_{\mathrm{F}} <\mathbf{t}>_{\mathrm{F}} \otimes_{\mathrm{F}} \mathbf{f}_2]_{s(\texttt{list})}) \quad \mathrm{len}(\mathbf{f}_1) = [\![\texttt{Int}]\!]_s\end{array}}{s, \mathbf{g}, \texttt{var} := \texttt{item}(\texttt{list}, \texttt{Int}) \rightsquigarrow [s|\texttt{var} \leftarrow \mathbf{node}], \mathbf{g}}$$

$$\frac{\mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}, \mathbf{name}_{\mathbf{id}}[\mathbf{f}]_{s(\texttt{list})}) \quad [\![\texttt{Int}]\!]_s \geq \mathrm{len}(\mathbf{f}) \vee [\![\texttt{Int}]\!]_s < 0}{s, \mathbf{g}, \texttt{var} := \texttt{item}(\texttt{list}, \texttt{Int}) \rightsquigarrow [s|\texttt{var} \leftarrow \mathbf{null}], \mathbf{g}}$$

$$\frac{\mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}, \text{“\#text”}_{s(\texttt{node})}\mathbf{str}_1 \otimes_{\mathrm{S}} \mathbf{str} \otimes_{\mathrm{S}} \mathbf{str}_2) \quad \mathrm{len}(\mathbf{str}_1) = [\![\texttt{Offset}]\!]_s \quad \mathrm{len}(\mathbf{str}) = [\![\texttt{Count}]\!]_s}{s, \mathbf{g}, \texttt{var} := \texttt{substringData}(\texttt{node}, \texttt{Offset}, \texttt{Count}) \rightsquigarrow [s|\texttt{var} \leftarrow \mathbf{str}], \mathbf{g}}$$

$$\frac{\mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}, \text{“\#text”}_{s(\texttt{node})}\mathbf{str}_1 \otimes_{\mathrm{S}} \mathbf{str}) \quad \mathrm{len}(\mathbf{str}_1) = [\![\texttt{Offset}]\!]_s \quad \mathrm{len}(\mathbf{str}) < [\![\texttt{Count}]\!]_s}{s, \mathbf{g}, \texttt{var} := \texttt{substringData}(\texttt{node}, \texttt{Offset}, \texttt{Count}) \rightsquigarrow [s|\texttt{var} \leftarrow \mathbf{str}], \mathbf{g}}$$

$$\frac{\begin{array}{c}\mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}, \text{“\#text”}_{s(\texttt{node})}\mathbf{str}) \\ \mathbf{g}' \equiv \mathrm{ap}(\mathbf{cg}, \text{“\#text”}_{s(\texttt{node})}\mathbf{str} \otimes_{\mathrm{S}} [\![\texttt{Arg}]\!]_s)\end{array}}{s, \mathbf{g}, \texttt{appendData}(\texttt{node}, \texttt{Arg}) \rightsquigarrow s, \mathbf{g}'}$$

$$\frac{\begin{array}{c}\mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}, \text{“\#text”}_{s(\texttt{node})}\mathbf{str}_1 \otimes_{\mathrm{S}} \mathbf{str} \otimes_{\mathrm{S}} \mathbf{str}_2) \\ \mathbf{g}' \equiv \mathrm{ap}(\mathbf{cg}, \text{“\#text”}_{s(\texttt{node})}\mathbf{str}_1 \otimes_{\mathrm{S}} \mathbf{str}_2) \\ \mathrm{len}(\mathbf{str}_1) = [\![\texttt{Offset}]\!]_s \quad \mathrm{len}(\mathbf{str}) = [\![\texttt{Count}]\!]_s\end{array}}{s, \mathbf{g}, \texttt{deleteData}(\texttt{node}, \texttt{Offset}, \texttt{Count}) \rightsquigarrow s, \mathbf{g}'} \qquad \frac{\begin{array}{c}\mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}, \text{“\#text”}_{s(\texttt{node})}\mathbf{str}_1 \otimes_{\mathrm{S}} \mathbf{str}) \\ \mathbf{g}' \equiv \mathrm{ap}(\mathbf{cg}, \text{“\#text”}_{s(\texttt{node})}\mathbf{str}_1) \\ \mathrm{len}(\mathbf{str}_1) = [\![\texttt{Offset}]\!]_s \quad \mathrm{len}(\mathbf{str}) < [\![\texttt{Count}]\!]_s\end{array}}{s, \mathbf{g}, \texttt{deleteData}(\texttt{node}, \texttt{Offset}, \texttt{Count}) \rightsquigarrow s, \mathbf{g}'}$$

Figure 3.5.: Semantics of Featherweight DOM Commands (Part 2)

4. Context Logic for Featherweight DOM

This chapter presents Context Logic for Featherweight DOM.

We follow previous work on Context Logic for analysing simple trees [13]. Although the basic reasoning is the same, the transition was by no means easy due to the comparative complexity of the DOM structures.

Context Logic consists of standard formulae constructed from the connectives of first-order logic, variables, expression tests, and quantification over variables. In addition, it has general *structural* formulae and *model-specific* formulae which reflect the data structure being modeled. We adapt the structural formulae of Context Logic slightly, so as to be able to handle the multiple types of DOM data. The structural formulae consist of an *application* connective for analysing context application, and its two corresponding *right adjoints*. For DOM data types $\mathrm{D}, \mathrm{D}_1, \mathrm{D}_2 \in \mathcal{D}$:

- the application formula $P \circ_{\mathrm{D}_1} P_1$ 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 subdata of type D_1 satisfying P_1; the application connective is annotated with type information about the context hole, since this cannot be determined from the given data;
- 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; the adjoint is annotated with type information about the resulting data, since this cannot be determined from the hole type; and
- the other 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; there is no type annotation as it can be inferred from the type of the given data.

With DOM-specific formulae we can analyse the data and context structure of DOM. The DOM-specific formulae have a direct correspondence with the data structures given in definitions 2 and 4. For example, the specific forest formula $<\text{“name”}_{\texttt{id}}[\text{true}_{\text{F}}]_{\texttt{fid}}>_{\text{F}}$ describes a forest containing one tree with top node labelled “name”, an arbitrary subforest, and identifiers determined by the values of the identifier variables given by the store. The formula:

$$\exists \texttt{ID}, \texttt{FID}.\, \text{true}_{(\text{F}\to\text{ELE})} \circ_{\text{F}} \langle \text{“name”}_{\texttt{ID}}[\text{true}_{\text{F}}]_{\texttt{FID}} \rangle_{\text{F}}$$

describes a tree which can be split into a context and a subforest containing one tree with top node labelled “name”. The formula:

$$\exists \texttt{ID}, \texttt{FID}.\, (\varnothing_{\text{F}} \multimap P) \circ_{\text{F}} \langle \text{“name”}_{\texttt{ID}}[\text{true}_{\text{F}}]_{\texttt{FID}} \rangle_{\text{F}}$$

describes a tree that can be split into a context and a subforest containing a tree with top node “name”. This time the context satisfies the property that, when the empty forest is put into the context hole, the resulting tree satisfies formula P. Finally, the formula:

$$\exists \texttt{ID}, \texttt{FID}, \texttt{ID}', \texttt{FID}'.$$
$$(\varnothing_{\text{F}} \multimap (\text{true}_{(\text{ELE}\to\text{G})} \circ_{\text{ELE}} \text{“name2”}_{\texttt{ID}'}[\text{true}_{\text{F}}]_{\texttt{FID}'})) \circ_{\text{F}} \langle \text{“name1”}_{\texttt{ID}}[\text{true}_{\text{F}}]_{\texttt{FID}} \rangle_{\text{F}}$$

describes a tree that can be split into a context and a subforest containing a tree with top node “name1”. The context contains the node “name2”. The node “name1” may be a descendent of “node2”, but may not be an ancestor of “node2”. Formulae of this shape are particularly useful in describing commands which move nodes around the tree.

4.1. Logical Variables

In addition to the program store s which stores program variables, we also introduce a logical environment e which stores auxiliary or logical variables. Logical variables may contain any value which may be stored in the program store s, and in addition may contain any data or context value. The logical environment e is not mentioned in the operational semantics for the Featherweight DOM language, and so values stored in it have no affect whatsoever on program execution. They exist solely to facilitate reasoning

about programs.

Definition 16 (Logical environment)**.** A logical environment e is a finite partial function sending logical variables $\mathrm{Var}_{\mathrm{LOGIC}}$ to their values:

$$e{:}\mathrm{Var}_{\mathrm{LOGIC}} \to (\{\mathbf{null}\} \cup \mathrm{ID} \cup \mathbb{Z} \cup \mathbb{B} \cup \mathrm{D} \cup (\mathrm{D}_1 \to \mathrm{D}_2))$$

where $\mathrm{D}, \mathrm{D}_1, \mathrm{D}_2 \in \mathcal{D}$ and $\mathrm{VAR}_{\mathrm{LOGIC}}$ denotes the set of logical variables.

To distinguish logical variables from program variables and expressions in our reasoning, we adopt the convention that logical variables will always be written in `UPPERCASE`.

In the example formulae given earlier, logical variables were existentially quantified to represent unknown values. For example:

$$\exists \mathtt{ID}, \mathtt{FID}.\, \mathrm{true}_{(\mathrm{F} \to \mathrm{ELE})} \circ_{\mathrm{F}} \langle \text{“name”}\, {}_{\mathtt{ID}}[\mathrm{true}_{\mathrm{F}}]_{\mathtt{FID}} \rangle_{\mathrm{F}}$$

In this example `ID` and `FID` are logical variables which represent the identifiers of the node in question. This technique is useful if we wish to discover the **id** of a node in the tree, and later assert that a program variable takes this value. In addition, it is possible to use logical variables to store the value of whole forests. For example, consider the formula:

$$\exists \mathtt{ID}, \mathtt{FID}.\, \mathrm{true}_{(\mathrm{F} \to \mathrm{ELE})} \circ_{\mathrm{F}} \langle \text{“name”}\, {}_{\mathtt{ID}}[\mathtt{F}{:}\mathrm{F}]_{\mathtt{FID}} \rangle_{\mathrm{F}}$$

This example is identical to the last one, except that it records the value of the subforest of the “name” node in the logical variable `F`. If this sort of formula is used in the precondition of a command, the variable `F` may be used in the postcondition to assert that the subforest has not changed. The notation :F declares the type of the variable `F`, and is explained in Section 4.4

4.2. Logical Expressions

Since we will wish to, for example, compare program and logical variables in our logical formulae, we introduce “logical expressions”. The definition of logical expressions and their evaluation function is identical to that of program expressions and their evaluation, except that logical expressions

$$
\begin{aligned}
[\![\mathbf{null}]\!]_{s,e} &\triangleq \mathbf{null} \\
[\![\varnothing_{\mathrm{S}}]\!]_{s,e} &\triangleq \varnothing_{\mathrm{S}} \\
[\![<\mathbf{c}>_{\mathrm{S}}]\!]_{s,e} &\triangleq <\mathbf{c}>_{\mathrm{S}} \\
[\![\mathbf{n}]\!]_{s,e} &\triangleq \mathbf{n} \\
[\![\mathrm{true}]\!]_{s,e} &\triangleq \mathrm{true} \\
[\![\mathrm{false}]\!]_{s,e} &\triangleq \mathrm{false} \\
[\![\mathtt{var}]\!]_{s,e} &\triangleq s(\mathtt{var}) \text{ iff } \mathtt{var} \in \mathrm{dom}(s) \\
[\![\mathtt{VAR}]\!]_{s,e} &\triangleq e(\mathtt{VAR}) \text{ iff } \mathtt{VAR} \in \mathrm{dom}(e) \\
[\![\mathtt{LExpr} = \mathtt{LExpr'}]\!]_{s,e} &\triangleq [\![\mathtt{LExpr}]\!]_{s,e} = [\![\mathtt{LExpr'}]\!]_{s,e} \\
[\![\mathtt{LExpr} \otimes_{\mathrm{S}} \mathtt{LExpr'}]\!]_{s,e} &\triangleq [\![\mathtt{LExpr}]\!]_{s,e} \otimes_{\mathrm{S}} [\![\mathtt{LExpr'}]\!]_{s,e} \text{ iff } [\![\mathtt{LExpr}]\!]_{s,e}, [\![\mathtt{LExpr'}]\!]_{s,e} \in \mathrm{S} \\
[\![\mathrm{len}(\mathtt{LExpr})]\!]_{s,e} &\triangleq \mathrm{len}([\![\mathtt{LExpr}]\!]_{s,e}) \\
[\![\mathtt{LExpr} + \mathtt{LExpr'}]\!]_{s,e} &\triangleq [\![\mathtt{LExpr}]\!]_{s,e} + [\![\mathtt{LExpr'}]\!]_{s,e} \text{ iff } [\![\mathtt{LExpr}]\!]_{s,e}, [\![\mathtt{LExpr'}]\!]_{s,e} \in \mathbb{Z} \\
[\![\mathtt{LExpr} - \mathtt{LExpr'}]\!]_{s,e} &\triangleq [\![\mathtt{LExpr}]\!]_{s,e} - [\![\mathtt{LExpr'}]\!]_{s,e} \text{ iff } [\![\mathtt{LExpr}]\!]_{s,e}, [\![\mathtt{LExpr'}]\!]_{s,e} \in \mathbb{Z} \\
[\![\mathtt{LExpr} \times \mathtt{LExpr'}]\!]_{s,e} &\triangleq [\![\mathtt{LExpr}]\!]_{s,e} \times [\![\mathtt{LExpr'}]\!]_{s,e} \text{ iff } [\![\mathtt{LExpr}]\!]_{s,e}, [\![\mathtt{LExpr'}]\!]_{s,e} \in \mathbb{Z} \\
[\![\mathtt{LExpr} \div \mathtt{LExpr'}]\!]_{s,e} &\triangleq [\![\mathtt{LExpr}]\!]_{s,e} \div [\![\mathtt{LExpr'}]\!]_{s,e} \text{ iff } [\![\mathtt{LExpr}]\!]_{s,e}, [\![\mathtt{LExpr'}]\!]_{s,e} \in \mathbb{Z} \\
[\![\mathtt{LExpr} \wedge \mathtt{LExpr'}]\!]_{s,e} &\triangleq [\![\mathtt{LExpr}]\!]_{s,e} \wedge [\![\mathtt{LExpr'}]\!]_{s,e} \text{ iff } [\![\mathtt{LExpr}]\!]_{s,e}, [\![\mathtt{LExpr'}]\!]_{s,e} \in \mathbb{B} \\
[\![\mathtt{LExpr} \vee \mathtt{LExpr'}]\!]_{s,e} &\triangleq [\![\mathtt{LExpr}]\!]_{s,e} \vee [\![\mathtt{LExpr'}]\!]_{s,e} \text{ iff } [\![\mathtt{LExpr}]\!]_{s,e}, [\![\mathtt{LExpr'}]\!]_{s,e} \in \mathbb{B} \\
[\![\neg \mathtt{LExpr}]\!]_{s,e} &\triangleq \neg [\![\mathtt{LExpr}]\!]_{s,e} \text{ iff } [\![\mathtt{LExpr}]\!]_{s,e} \in \mathbb{B}
\end{aligned}
$$

Figure 4.1.: Logical Expression Evaluation

can refer to logical variables, and where a program expression evaluation would cause a program fault, a logical expression evaluation will render the formula unsatisfiable.

Definition 17 (Logical Expressions)**.**

LExpr ::=	**null** $\mid \varnothing_{\mathrm{D}} \mid <\mathbf{c}>_{\mathrm{S}} \mid \mathbf{n} \mid$ true $\mid$ false $\mid$	literal constants
	var $\mid$ VAR $\mid$	program and logical variables
	LExpr $=$ LExpr $\mid$	equality test
	LExpr $\otimes_{\mathrm{S}}$ LExpr $\mid$	string concatenation
	len(LExpr) $\mid$	string length
	LExpr $+$ LExpr $\mid$ LExpr $-$ LExpr $\mid$ LExpr $\times$ LExpr $\mid$ LExpr $\div$ LExpr $\mid$	arithmetic operations
	LExpr $\wedge$ LExpr $\mid$ LExpr $\vee$ LExpr $\mid \neg$LExpr	boolean operations

As in Chapter 3.1 we have the shorthand "abc" and we write logical expression names in `UpperCamelCase`.

Definition 18 (Logical Expression Evaluation)**.** The evaluation of logical expression `LExpr` in store s and environment e is defined by the function given in Figure 4.1.

Note that this evaluation is a partial function, since not all expressions can be successfully evaluated. If such an expression is part of a formula, then that formula is unsatisfiable.

Lemma 19 (Program and Logic Expression Equivalence)**.** *For any program expression* ***Expr****, store* s *and environment* e *such that* $[\![\textbf{\textit{Expr}}]\!]_s$ *does not* fault *we have* $[\![\textbf{\textit{Expr}}]\!]_s \equiv [\![\textbf{\textit{Expr}}]\!]_{s,e}$

Proof. The proof follows directly from the definitions of program and logical expressions (which are identical except for the addition of environment variables to logical expressions) and the evaluation functions of program and logical expressions (which are identical except for the addition of environment variable evaluation for logical expressions). □

In the example formulae given earlier, logical expressions were used to specify the names of some of the nodes in the tree. For example:

$$\exists \texttt{ID}, \texttt{FID}.\, \text{true}_{(\text{F}\rightarrow\text{ELE})} \circ_{\text{F}} \langle \text{“name”}_{\texttt{ID}}[\text{true}_{\text{F}}]_{\texttt{FID}} \rangle_{\text{F}}$$

In this example, the string “name” is a simple logical expression.

4.3. Logical Formulae

We present the logical formulae for Featherweight DOM, following the informal discussion at the beginning of this chapter.

Definition 20 (Featherweight DOM Formulae)**.** Let A denote a data or context type of the form D or $(\text{D}_1 \rightarrow \text{D}_2)$ where $\text{D}, \text{D}_1, \text{D}_2 \in \mathcal{D}$. The Feath-

erweight DOM Formulae are defined by:

$$
\begin{array}{lll}
P ::= \neg P \mid P \wedge P \mid P \vee P \mid \mathrm{true}_{\mathrm{A}} \mid \mathrm{false}_{\mathrm{A}} & & \text{boolean formulae} \\
\quad \mid P \circ_{\mathrm{D}_1} P \mid P \circ\!\!-_{\mathrm{D}_2} P \mid P \multimap P & & \text{structural formulae} \\
\quad \mid P_{\mathrm{DOM}} & & \text{DOM-specific formulae (see below)} \\
\quad \mid \mathtt{VAR}{:}\mathrm{A} & & \text{type-annotated logical variables} \\
\quad \mid \mathtt{LExpr} \doteq \mathtt{LExpr} \mid \mathtt{LExpr} <\cdot\, \mathtt{LExpr} & & \text{expression equality and inequality} \\
\quad \mid \mathtt{LExpr} \in \mathtt{LExpr} & & \text{string inclusion} \\
\quad \mid \mathtt{LExpr} \in \mathrm{D} & & \text{type checking} \\
\quad \mid \exists \mathtt{VAR}.\, P & & \text{quantification}
\end{array}
$$

where VAR denotes a logical variable.

The *DOM-specific* formulae are:

$$
\begin{array}{rl}
P_{\mathrm{DOM}} ::= & -_{\mathrm{ELE}} \mid P_{\mathtt{Id}}[P]_{\mathtt{Fid}} \\
& \mid -_{\mathrm{TXT}} \mid \text{“\#text”}_{\mathtt{Id}} P \\
& \mid \varnothing_{\mathrm{F}} \mid -_{\mathrm{F}} \mid <P>_{\mathrm{F}} \mid P \otimes_{\mathrm{F}} P \\
& \mid \varnothing_{\mathrm{G}} \mid -_{\mathrm{G}} \mid <P>_{\mathrm{G}} \mid P \oplus P \\
& \mid \varnothing_{\mathrm{S}} \mid -_{\mathrm{S}} \mid <P>_{\mathrm{S}} \mid P \otimes_{\mathrm{S}} P \mid \mathtt{LExpr}
\end{array}
$$

Notice that we use the notation $\doteq$ and $<\cdot$ to compare expressions at the formula level. This allows us to distinguish between an untyped expression which tests the equality of two subexpressions, and a typed formula which tests the equality of two untyped expressions. Notice also that just as we may compare the values of untyped expressions using $\doteq$, we may also check their types using the notation $\mathtt{LExpr} \in \mathrm{D}$. For example, the formula $\varnothing_{\mathrm{G}} \wedge (\mathtt{VAR} \in \mathrm{ELE})$ is satisfied by the grove $\varnothing_{\mathrm{G}}$ if and only if the value in the logical variable VAR is of type ELE.

Another symptom of the boundary between untyped expressions and typed formulae is the need to type-annotate logical variables VAR when they appear in formulae. This annotation takes the form $\mathtt{VAR}{:}\mathrm{A}$.

The types of formulae and the consequences of referring to untyped expressions in typed formulae are described more thoroughly in the next section.

$$\frac{P{:}\mathrm{A}}{\neg P{:}\mathrm{A}} \qquad \frac{P_1{:}\mathrm{A} \quad P_2{:}\mathrm{A}}{(P_1 \wedge P_2){:}\mathrm{A}} \qquad \frac{P_1{:}\mathrm{A} \quad P_2{:}\mathrm{A}}{(P_1 \vee P_2){:}\mathrm{A}}$$

$$\frac{}{\mathrm{true}_\mathrm{A}{:}\mathrm{A}} \qquad \frac{}{\mathrm{false}_\mathrm{A}{:}\mathrm{A}}$$

Figure 4.2.: Boolean Formula Types

$$\frac{P_1{:}\mathrm{D}_2 \quad P_2{:}\mathrm{D}_1}{(P_1 \circ_{\mathrm{D}_1} P_2){:}\mathrm{D}_2} \qquad \frac{P_1{:}\mathrm{D}_1{\to}\mathrm{D}_2 \quad P_2{:}\mathrm{D}_2}{(P_1 \circ\!\!-_{\mathrm{D}_2} P_2){:}\mathrm{D}_1} \qquad \frac{P_1{:}\mathrm{D}_1 \quad P_2{:}\mathrm{D}_2}{(P_1 \multimap P_2){:}\mathrm{D}_1{\to}\mathrm{D}_2}$$

Figure 4.3.: Structural Formula Types

4.4. Formula Types

The type annotations on the formulae enable us to define a simple typing relation $P{:}A$, where A is a data or context type, by induction on the structure of formula P. For a formula to be satisfied by a datum or context $\mathbf{a} \in \mathrm{A}$, the formula in question must also be of type A.

The Boolean formulae and quantified formulae inherit their types from the subformulae. Formulae which only compare expressions (such as $\doteq$ and $<\cdot$) will satisfy arbitrary A, since they test the store rather than the data and context structures and hence are really outside the typing system. The DOM-Specific formulae have specific types.

Definition 21 (Formula Types)**.** Let A denote a data or context type of the form D or $(\mathrm{D}_1{\to}\mathrm{D}_2)$ where $\mathrm{D}, \mathrm{D}_1, \mathrm{D}_2 \in \mathcal{D}$. The types for the boolean formulae are given in Figure 4.2. The types for the structural formulae are given in Figure 4.3. DOM-specific formulae types are given in Figure 4.4 while the remaining formula types are given in Figure 4.5.

The boundary between untyped logical expressions and typed formulae is managed by the use of type annotation. In particular, logical variables must be annotated using the notation `VAR`:A. If the value of a logical variable does not match the type of its annotation, then the formula is unsatisfiable.

A formula which compares untyped expressions using $\doteq$ does not require the two expressions being compared to be of any particular type, but it is only satisfied if they both evaluate to the same value. This in turn implies

$$\frac{}{-_{\mathrm{D}}:\mathrm{D}\to\mathrm{D}} \qquad \frac{P:\mathrm{S} \quad P':\mathrm{F}}{P_{\texttt{Id}}[P']_{\texttt{FId}}:\mathrm{ELE}} \qquad \frac{P:\mathrm{S} \quad P':\mathrm{D}\to\mathrm{F}}{P_{\texttt{Id}}[P']_{\texttt{FId}}:\mathrm{D}\to\mathrm{ELE}}$$

$$\frac{P:\mathrm{S}}{\text{“\#text”}_{\texttt{Id}}P:\mathrm{TXT}} \qquad \frac{P:\mathrm{S}\to\mathrm{S}}{\text{“\#text”}_{\texttt{Id}}P:\mathrm{S}\to\mathrm{TXT}} \qquad \frac{}{\varnothing_{\mathrm{D}}:\mathrm{D}}$$

$$\frac{P_1:\mathrm{D}\to\mathrm{F} \quad P_2:\mathrm{F}}{(P_1 \otimes_{\mathrm{F}} P_2):\mathrm{D}\to\mathrm{F}} \qquad \frac{P_1:\mathrm{F} \quad P_2:\mathrm{D}\to\mathrm{F}}{(P_1 \otimes_{\mathrm{F}} P_2):\mathrm{D}\to\mathrm{F}} \qquad \frac{P:\mathrm{TXT}}{<P>_{\mathrm{F}}:\mathrm{F}}$$

$$\frac{P:\mathrm{ELE}}{<P>_{\mathrm{F}}:\mathrm{F}} \qquad \frac{P:\mathrm{D}\to\mathrm{TXT}}{<P>_{\mathrm{F}}:\mathrm{D}\to\mathrm{F}} \qquad \frac{P:\mathrm{D}\to\mathrm{ELE}}{<P>_{\mathrm{F}}:\mathrm{D}\to\mathrm{F}}$$

$$\frac{P_1:\mathrm{F} \quad P_2:\mathrm{F}}{(P_1 \otimes_{\mathrm{F}} P_2):\mathrm{F}} \qquad \frac{P:\mathrm{TXT}}{<P>_{\mathrm{G}}:\mathrm{G}} \qquad \frac{P:\mathrm{ELE}}{<P>_{\mathrm{G}}:\mathrm{G}}$$

$$\frac{P:\mathrm{D}\to\mathrm{TXT}}{<P>_{\mathrm{G}}:\mathrm{D}\to\mathrm{G}} \qquad \frac{P:\mathrm{D}\to\mathrm{ELE}}{<P>_{\mathrm{G}}:\mathrm{D}\to\mathrm{G}} \qquad \frac{P_1:\mathrm{G} \quad P_2:\mathrm{G}}{(P_1 \oplus P_2):\mathrm{G}}$$

$$\frac{P_1:\mathrm{D}\to\mathrm{G} \quad P_2:\mathrm{G}}{(P_1 \oplus P_2):\mathrm{D}\to\mathrm{G}} \qquad \frac{P:\mathrm{C}}{<P>_{\mathrm{S}}:\mathrm{S}} \qquad \frac{P_1:\mathrm{S} \quad P_2:\mathrm{S}}{(P_1 \otimes_{\mathrm{S}} P_2):\mathrm{S}}$$

$$\frac{}{\texttt{LExpr}:\mathrm{S}} \qquad \frac{P_1:\mathrm{S} \quad P_2:\mathrm{D}\to\mathrm{S}}{(P_1 \otimes_{\mathrm{S}} P_2):\mathrm{D}\to\mathrm{S}} \qquad \frac{P_1:\mathrm{D}\to\mathrm{S} \quad P_2:\mathrm{S}}{(P_1 \otimes_{\mathrm{S}} P_2):\mathrm{D}\to\mathrm{S}}$$

Figure 4.4.: DOM-Specific Formula Types

$$\frac{}{(\texttt{VAR}:\mathrm{A}):\mathrm{A}} \qquad \frac{}{(\texttt{LExpr} \doteq \texttt{LExpr}'):\mathrm{A}} \qquad \frac{}{(\texttt{LExpr} <\cdot\, \texttt{LExpr}'):\mathrm{A}}$$

$$\frac{}{(\texttt{LExpr} \in \texttt{LExpr}'):\mathrm{A}} \qquad \frac{}{(\texttt{LExpr} \in \mathrm{D}):\mathrm{A}} \qquad \frac{P:\mathrm{A}}{(\exists \texttt{VAR}.\, P):\mathrm{A}}$$

Figure 4.5.: Remaining Formula Types

that the expressions evaluate to values of the same type. In contrast, a formula which compares untyped expressions using $<\cdot$ does require that the two expressions being compared evaluate to a specific type – they must be integers. In all other cases, it is unsatisfiable.

Finally, notice that untyped expressions may appear without annotation as a part of a DOM-specific string formula. In this case the formula is of type S and is only satisfiable if the expression in question evaluates to a string. For example, note the difference between the formula `VAR`:ELE and the formula `VAR`. The formula `VAR`:ELE is of type ELE and is satisfied by the element node which is equal to the element node stored in the logical variable `VAR`. If the logical variable `VAR` does not contain an element node, the formula is unsatisfiable. The formula `VAR` is of type S and is satisfied by the string which is equal to the value stored in `VAR`. If the logical variable `VAR` does not contain a string, the formula is unsatisfiable.

Recall the example formula given earlier:

$$\exists \texttt{ID}, \texttt{FID}.\ \text{true}_{(\text{F}\rightarrow\text{ELE})} \circ_{\text{F}} \langle \text{“name”}_{\texttt{ID}}[\texttt{F}{:}\text{F}]_{\texttt{FID}} \rangle_{\text{F}}$$

The purpose of the type annotations in that formula should now be clear. In particular, notice that the logical variable `F` may contain any value of any type. However, the formula is only satisfiable if the logical variable `F` contains a forest of type F as asserted by the type annotation :F.

Sometimes, it is convenient to write a formula with an under-specified type. For example:

$$<\text{true}_{\text{T}}>_{\text{F}} \qquad \text{where T} \in \{\text{ELE}, \text{TXT}\}$$

This formula is really shorthand for the following two formulae:

$$<\text{true}_{\text{ELE}}>_{\text{F}}$$
$$<\text{true}_{\text{TXT}}>_{\text{F}}$$

Together, these formulae are satisfied by a singleton forest which contains either an element or a text node.

Similarly, consider the formula:

$$<\texttt{T}{:}\text{T}>_{\text{F}} \qquad \text{where T} \in \{\text{ELE}, \text{TXT}\}$$

This formula is identical to the previous example, except that it also asserts that the element or node in question be equal to the one stored in the logical variable `T`. This pattern is particularly useful when we wish to refer to a node with a particular **id** without requiring that that node be of a particular type. For example, the precondition of a command might refer to the node identified by the program variable "`node`":

$$(\texttt{NAME}_{\texttt{node}}[\texttt{F:F}]_{\texttt{FID}} \vee \texttt{NAME}_{\texttt{node}}\texttt{VAL}) \wedge \texttt{T:T}$$
$$\text{where } \mathrm{T} \in \{\mathrm{ELE}, \mathrm{TXT}\}$$

This formula may be satisfied by either an element or a text node, so long as that node is referred to by the program variable `node`. Notice also that the logical variable `T` must contain the element or text node in question. This allows us to refer to the same variable in both the precondition and postcondition of a command. For example in the axiom for "`getNodeName`", we use the variable `T` to assert that the grove is not changed by the command.

4.5. Satisfaction

Definition 22 (Satisfaction for Featherweight DOM)**.** The semantics of our logic for Featherweight DOM is given by the satisfaction relation $\models_{\mathrm{A}}$ where A denotes a data or context type of the form D or $(\mathrm{D}_1 \to \mathrm{D}_2)$ for $\mathrm{D}, \mathrm{D}_1, \mathrm{D}_2 \in \mathcal{D}$.

The satisfaction relation for boolean formulae is given in Figure 4.6. The satisfaction relation for the structural formulae is given in Figure 4.7. The relation for DOM-specific formulae is given in Figures 4.8 and 4.9 while the relation for the remaining formulae[1] is in Figure 4.10.

Notice that the formula $\neg P$ is not satisfied by data or contexts of a different type to P. The formula $\neg P$ is satisfied by all data or contexts which do not satisfy P, so long as they are also of the same type as P. For example, the formula $\exists \texttt{ID}, \texttt{FID}.\ \text{“p”}_{\texttt{ID}}[\mathrm{true}_{\mathrm{F}}]_{\texttt{FID}}$ is satisfied by all elements with the

[1] Notice that meaning of the notation $\doteq$ is significantly different from the similar notation routinely used in separation logic. In those separation logic the notation , $x \doteq y$ is equivalent to $x = y \wedge \texttt{emp}$, and so the footprint of $x \doteq y$ is of size zero. In this work we use $\doteq$ to distinguish the untyped expression $\texttt{Expr} = \texttt{Expr}'$ from the typed formula $\texttt{Expr} \doteq \texttt{Expr}'$.The boundary between untyped expressions and typed formulae is discussed further in Section 4.4.

$$
\begin{array}{ll}
e,s,\mathbf{a} \models_{\mathrm{A}} \neg P & \iff P{:}\mathrm{A} \wedge e,s,\mathbf{a} \not\models_{\mathrm{A}} P \\
e,s,\mathbf{a} \models_{\mathrm{A}} P_1 \wedge P_2 & \iff (e,s,\mathbf{a} \models_{\mathrm{A}} P_1) \wedge (e,s,\mathbf{a} \models_{\mathrm{A}} P_2) \\
e,s,\mathbf{a} \models_{\mathrm{A}} P_1 \vee P_2 & \iff (e,s,\mathbf{a} \models_{\mathrm{A}} P_1) \vee (e,s,\mathbf{a} \models_{\mathrm{A}} P_2) \\
e,s,\mathbf{a} \models_{\mathrm{A}} \mathrm{true}_{\mathrm{A}} & \textit{always} \\
e,s,\mathbf{a} \models_{\mathrm{A}} \mathrm{false}_{\mathrm{A}} & \textit{never}
\end{array}
$$

Figure 4.6.: Satisfaction for Boolean Formulae

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

Figure 4.7.: Satisfaction for Structural Formulae

nodeName "p". The formula $\neg(\exists \mathtt{ID},\mathtt{FID}.\ \text{“p”}_{\mathtt{ID}}[\mathrm{true}_{\mathrm{F}}]_{\mathtt{FID}})$ is satisfied by all elements whose nodeName is not "p".

4.6. Derived Notation

Expression inequalities such as $\cdot >$, $<\dot{=}$ and $>\dot{=}$ are derivable in the usual way, as is string exclusion $\notin$. We also introduce notation for expressing 'somewhere, potentially deep down' $\Diamond_{\mathrm{D}_1 \to \mathrm{D}_2} P$ and 'everywhere from here down' $\Box_{\mathrm{D}_1 \to \mathrm{D}_2} P$. Similarly, we define the related concept of "somewhere at this forest-level" $\Diamond_{\otimes}(P)$ and "everywhere at this forest-level" $\Box_{\otimes}(P)$:

$$
\begin{array}{ll}
\Diamond_{(\mathrm{D}_1 \to \mathrm{D}_2)} P \triangleq \mathrm{true}_{(\mathrm{D}_1 \to \mathrm{D}_2)} \circ_{\mathrm{D}_1} P & \Diamond_{\otimes} P \triangleq (\mathrm{true}_{\mathrm{F}} \otimes_{\mathrm{F}} <P>_{\mathrm{F}} \otimes_{\mathrm{F}} \mathrm{true}_{\mathrm{F}}) \\
\Box_{(\mathrm{D}_1 \to \mathrm{D}_2)} P \triangleq \neg \Diamond_{(\mathrm{D}_1 \to \mathrm{D}_2)} \neg P & \Box_{\otimes} P \triangleq \neg \Diamond_{\otimes} \neg P
\end{array}
$$

Finally, it is also sometimes convenient to write formula without IDs. So

$$e, s, \mathbf{cd} \models_{(\mathrm{ELE}\to\mathrm{ELE})} -_{\mathrm{ELE}} \iff \mathbf{cd} \equiv -_{\mathrm{ELE}}$$

$$e, s, \mathbf{d} \models_{\mathrm{ELE}} P_{\mathtt{Id}}[P']_{\mathtt{Fid}} \iff \exists \mathbf{s}{:}\mathrm{S}, \mathbf{f}{:}\mathrm{F}.\, (\mathbf{d} \equiv \mathbf{s}_{[\![\mathtt{Id}]\!]_{s,e}}[\mathbf{f}]_{[\![\mathtt{Fid}]\!]_{s,e}}) \wedge e, s, \mathbf{s} \models_{\mathrm{S}} P \wedge e, s, \mathbf{f} \models_{\mathrm{F}} P'$$

$$e, s, \mathbf{cd} \models_{(\mathrm{D}\to\mathrm{ELE})} P_{\mathtt{Id}}[P']_{\mathtt{Fid}} \iff \exists \mathbf{s}{:}\mathrm{S}, \mathbf{cf}{:}(\mathrm{D}\to\mathrm{F}).\, (\mathbf{cd} \equiv \mathbf{s}_{[\![\mathtt{Id}]\!]_{s,e}}[\mathbf{cf}]_{[\![\mathtt{Fid}]\!]_{s,e}}) \wedge e, s, \mathbf{s} \models_{\mathrm{S}} P \wedge e, s, \mathbf{cf} \models_{(\mathrm{D}\to\mathrm{F})} P'$$

$$e, s, \mathbf{cd} \models_{(\mathrm{TXT}\to\mathrm{TXT})} -_{\mathrm{TXT}} \iff \mathbf{cd} \equiv -_{\mathrm{TXT}}$$

$$e, s, \mathbf{d} \models_{\mathrm{TXT}} \text{“\#text”}_{\mathtt{Id}} P \iff \exists \mathbf{s}{:}\mathrm{S}.\, (\mathbf{d} \equiv \text{“\#text”}_{[\![\mathtt{Id}]\!]_{s,e}} \text{“}\mathbf{s}\text{”}) \wedge e, s, \mathbf{s} \models_{\mathrm{S}} P$$

$$e, s, \mathbf{cd} \models_{(\mathrm{S}\to\mathrm{TXT})} \text{“\#text”}_{\mathtt{Id}} \text{“P”} \iff \exists \mathbf{cs}{:}(\mathrm{S}\to\mathrm{S}).\, (\mathbf{cd} \equiv \text{“\#text”}_{[\![\mathtt{Id}]\!]_{s,e}} \text{“}\mathbf{cs}\text{”}) \wedge e, s, \mathbf{cs} \models_{(\mathrm{S}\to\mathrm{S})} P$$

$$e, s, \mathbf{d} \models_{\mathrm{F}} \varnothing_{\mathrm{F}} \iff \mathbf{d} \equiv \varnothing_{\mathrm{F}}$$

$$e, s, \mathbf{cd} \models_{(\mathrm{F}\to\mathrm{F})} -_{\mathrm{F}} \iff \mathbf{cd} \equiv -_{\mathrm{F}}$$

$$e, s, \mathbf{d} \models_{\mathrm{F}} <P>_{\mathrm{F}} \iff \exists \mathbf{d}'{:}\mathrm{D}.\, (\mathbf{d} \equiv <\mathbf{d}'>_{\mathrm{F}}) \wedge e, s, \mathbf{d}' \models_{\mathrm{D}} P$$

$$e, s, \mathbf{cd} \models_{(\mathrm{D}_1\to\mathrm{F})} <P>_{\mathrm{F}} \iff \exists \mathbf{cd}'{:}(\mathrm{D}_1\to\mathrm{D}_2).\, (\mathbf{cd} \equiv <\mathbf{cd}'>_{\mathrm{F}}) \wedge e, s, \mathbf{cd}' \models_{(\mathrm{D}_1\to\mathrm{D}_2)} P$$

$$e, s, \mathbf{d} \models_{\mathrm{F}} P_1 \otimes_{\mathrm{F}} P_2 \iff \exists \mathbf{f}_1{:}\mathrm{F}, \mathbf{f}_2{:}\mathrm{F}.\, (\mathbf{d} \equiv \mathbf{f}_1 \otimes_{\mathrm{F}} \mathbf{f}_2) \wedge e, s, \mathbf{f}_1 \models_{\mathrm{F}} P_1 \wedge e, s, \mathbf{f}_2 \models_{\mathrm{F}} P_2$$

$$e, s, \mathbf{cd} \models_{(\mathrm{D}\to\mathrm{F})} P_1 \otimes_{\mathrm{F}} P_2 \iff \exists \mathbf{cd}'{:}(\mathrm{D}\to\mathrm{F}), \mathbf{f}'{:}\mathrm{F}.\, ((\mathbf{cd} \equiv \mathbf{cd}' \otimes_{\mathrm{F}} \mathbf{f}') \wedge e, s, \mathbf{cd}' \models_{(\mathrm{D}\to\mathrm{F})} P_1 \wedge e, s, \mathbf{f}' \models_{\mathrm{F}} P_2) \vee ((\mathbf{cd} \equiv \mathbf{f}' \otimes_{\mathrm{F}} \mathbf{cd}') \wedge e, s, \mathbf{f}' \models_{\mathrm{F}} P_1 \wedge e, s, \mathbf{cd}' \models_{(\mathrm{D}\to\mathrm{F})} P_2)$$

Figure 4.8.: Satisfaction for DOM-Specific Formulae (Part 1)

$$
\begin{array}{lll}
e, s, \mathbf{d} \models_{\mathrm{G}} \varnothing_{\mathrm{G}} & \iff & \mathbf{d} \equiv \varnothing_{\mathrm{G}} \\
e, s, \mathbf{cd} \models_{(\mathrm{G}\to\mathrm{G})} -_{\mathrm{G}} & \iff & \mathbf{cd} \equiv -_{\mathrm{G}} \\
e, s, \mathbf{d} \models_{\mathrm{G}} <P>_{\mathrm{G}} & \iff & \exists \mathbf{d}'{:}\mathrm{D}.\, (\mathbf{d} \equiv <\mathbf{d}'>_{\mathrm{G}}) \wedge e, s, \mathbf{d}' \models_{\mathrm{D}} P \\
e, s, \mathbf{cd} \models_{(\mathrm{D}_1\to\mathrm{G})} <P>_{\mathrm{G}} & \iff & \exists \mathbf{cd}'{:}(\mathrm{D}_1\to\mathrm{D}_2).\, (\mathbf{cd} \equiv <\mathbf{cd}'>_{\mathrm{G}}) \wedge e, s, \mathbf{cd}' \models_{(\mathrm{D}_1\to\mathrm{D}_2)} P \\
e, s, \mathbf{d} \models_{\mathrm{G}} P_1 \oplus P_2 & \iff & \exists \mathbf{d}_1{:}\mathrm{G}, \mathbf{d}_2{:}\mathrm{G}.\, (\mathbf{d} \equiv \mathbf{d}_1 \oplus \mathbf{d}_2) \wedge \\
 & & e, s, \mathbf{d}_1 \models_{\mathrm{G}} P_1 \wedge e, s, \mathbf{d}_2 \models_{\mathrm{G}} P_2 \\
e, s, \mathbf{cd} \models_{(\mathrm{D}\to\mathrm{G})} P_1 \oplus P_2 & \iff & \exists \mathbf{cd}'{:}(\mathrm{D}\to\mathrm{G}), \mathbf{g}{:}\mathrm{G}.\, (\mathbf{cd} \equiv \mathbf{cd}' \oplus \mathbf{g}) \wedge \\
 & & e, s, \mathbf{cd}' \models_{(\mathrm{D}\to\mathrm{G})} P_1 \wedge e, s, \mathbf{g} \models_{\mathrm{G}} P_2 \\
e, s, \mathbf{cd} \models_{(\mathrm{S}\to\mathrm{S})} -_{\mathrm{S}} & \iff & \mathbf{cd} \equiv -_{\mathrm{S}} \\
e, s, \mathbf{d} \models_{\mathrm{S}} \texttt{LExpr} & \iff & \mathbf{d} \equiv [\![\texttt{LExpr}]\!]_{s,e} \wedge [\![\texttt{LExpr}]\!]_{s,e} \in \mathrm{S} \cup \{\mathbf{null}\} \\
e, s, \mathbf{d} \models_{\mathrm{S}} P_1 \otimes_{\mathrm{S}} P_2 & \iff & \exists \mathbf{d}_1{:}\mathrm{S}, \mathbf{d}_2{:}\mathrm{S}.\, (\mathbf{d} \equiv \mathbf{d}_1 \otimes_{\mathrm{S}} \mathbf{d}_2) \wedge \\
 & & e, s, \mathbf{d}_1 \models_{\mathrm{S}} P_1 \wedge e, s, \mathbf{d}_2 \models_{\mathrm{S}} P_2 \\
e, s, \mathbf{cd} \models_{(\mathrm{S}\to\mathrm{S})} P_1 \otimes_{\mathrm{S}} P_2 & \iff & \exists \mathbf{cd}'{:}(\mathrm{S}\to\mathrm{S}), \mathbf{d}{:}(\mathrm{S}). \\
 & & ((\mathbf{cd}{\equiv}\mathbf{cd}' \otimes_{\mathrm{S}} \mathbf{d}) \wedge e, s, \mathbf{cd}'{\models_{(\mathrm{S}\to\mathrm{S})}} P_1 \wedge e, s, \mathbf{d}{\models_{\mathrm{S}}} P_2) \vee \\
 & & ((\mathbf{cd}{\equiv}\mathbf{d} \otimes_{\mathrm{S}} \mathbf{cd}') \wedge e, s, \mathbf{d}{\models_{\mathrm{S}}} P_1 \wedge e, s, \mathbf{cd}'{\models_{(\mathrm{S}\to\mathrm{S})}} P_2)
\end{array}
$$

Figure 4.9.: Satisfaction for DOM-Specific Formulae (Part 2)

$$
\begin{array}{lll}
e, s, \mathbf{a} \models_{\mathrm{A}} \texttt{VAR}{:}\mathrm{A} & \iff & e(\texttt{VAR}) \in \mathrm{A} \wedge \mathbf{a} \equiv e(\texttt{VAR}) \\
e, s, \mathbf{a} \models_{\mathrm{A}} \texttt{LExpr} \doteq \texttt{LExpr}' & \iff & [\![\texttt{LExpr}]\!]_{s,e} = [\![\texttt{LExpr}']\!]_{s,e} \\
e, s, \mathbf{a} \models_{\mathrm{A}} \texttt{LExpr} <\cdot\, \texttt{LExpr}' & \iff & [\![\texttt{LExpr}]\!]_{s,e}, [\![\texttt{LExpr}']\!]_{s,e} \in \mathbb{Z} \wedge [\![\texttt{LExpr}]\!]_{s,e} < [\![\texttt{LExpr}']\!]_{s,e} \\
e, s, \mathbf{a} \models_{\mathrm{A}} \texttt{LExpr} \in \texttt{LExpr}' & \iff & [\![\texttt{LExpr}]\!]_{s,e}, [\![\texttt{LExpr}']\!]_{s,e} \in \mathrm{S} \wedge [\![\texttt{LExpr}]\!]_{s,e} \in [\![\texttt{LExpr}']\!]_{s,e} \\
e, s, \mathbf{a} \models_{\mathrm{A}} \texttt{LExpr} \in \mathrm{D} & \iff & [\![\texttt{LExpr}]\!]_{s,e} \in \mathrm{D} \\
e, s, \mathbf{a} \models_{\mathrm{A}} \exists \texttt{VAR}.\, P & \iff & \exists \mathbf{v}.\, [e|\texttt{VAR} \leftarrow \mathbf{v}], s, \mathbf{a} \models_{\mathrm{A}} P
\end{array}
$$

Figure 4.10.: Satisfaction for the Remaining Formulae

we introduce the shorthand notation:

$$\begin{aligned} P[P'] &\triangleq \exists \mathtt{ID}, \mathtt{FID}.\, P_{\mathtt{ID}}[P']_{\mathtt{FID}} \\ \text{“\#text”}\, P &\triangleq \exists \mathtt{ID}.\, \text{“\#text”}_{\mathtt{ID}} P \\ \text{“\#text”} &\triangleq \exists \mathtt{ID}, \mathtt{VAL}.\, \text{“\#text”}_{\mathtt{ID}} \mathtt{VAL} \end{aligned}$$

The notations $\Box_{\otimes}$ and $\Box_{(\mathrm{D}_1 \to \mathrm{D}_2)}$ deserve further comment. Since their behaviour is quite similar, we will illustrate with $\Box_{\otimes}$. While this notation is described above using the phrase “everywhere at this forest-level”, a more accurate description might be “everywhere at this forest level which is of the same type as the following formula”. This is because of the way logical negation interacts with the types of our formulae. Recall that the formula $\neg P$ is satisfied by all data or contexts which do not satisfy P, so long as they are also of the same type as P. For example, the formula “p”$[\mathrm{true}_{\mathrm{F}}]$ is satisfied by all elements with the nodeName “p” and the formula $\neg$(“p”$[\mathrm{true}_{\mathrm{F}}]$) is satisfied by all elements whose nodeName is not “p”.

With this in mind, consider the following formulae:

- $\Diamond_{\otimes}(\mathrm{true}_{\mathrm{ELE}})$ describes a forest which contains at least one element node.
- $\Diamond_{\otimes}$(“p”$[\mathrm{true}_{\mathrm{F}}]$) describes a forest which contains at least one element node with nodeName “p”.
- $\Diamond_{\otimes}(\neg$“p”$[\mathrm{true}_{\mathrm{F}}])$ describes a forest which contains at least one element node with a nodeName which isn’t “p”. In particular, this formula is not satisfied by a forest which contains only text nodes.
- $\Diamond_{\otimes}(\neg\mathrm{true}_{\mathrm{ELE}})$ describes a forest which contains at least one element node which does not satisfy $\mathrm{true}_{\mathrm{ELE}}$. Since there are no element nodes which do not satisfy $\mathrm{true}_{\mathrm{ELE}}$, this formula is not satisfiable.
- $\neg\Diamond_{\otimes}(\neg\mathrm{true}_{\mathrm{ELE}})$ describes a forest which does not contain any element node which does not satisfy $\mathrm{true}_{\mathrm{ELE}}$. Since there are no element nodes which do not satisfy $\mathrm{true}_{\mathrm{ELE}}$, this formula is trivially satisfied by all forests.
- $\neg\Diamond_{\otimes}(\neg$“p”$[\mathrm{true}_{\mathrm{F}}])$ describes a forest which does not contain any element node with a nodeName which isn’t “p”. That is to say, all

element nodes in the forest must have the nodeName "p". This formula is satisfied by a forest which contains only text nodes.

In this light of this interaction between logical negation, the types of our formulae and the "somewhere" notation, consider the following formulae:

- $\Box_{\otimes}(\text{true}_{\text{ELE}})$ describes any forest in which all element nodes satisfy true_{ELE}. Since all element nodes satisfy true_{ELE} by definition, this formula is satisfied by all forests.
- $\Box_{\otimes}(\text{"p"}[\text{true}_{\text{F}}])$ describes any forest in which all element nodes have the nodeName "p". This formula is satisfied by a forest which contains only text nodes.
- $\neg\Diamond_{\otimes}(\text{true}_{\text{TXT}}) \wedge \Box_{\otimes}(\text{"p"}[\text{true}_{\text{F}}])$ describes a forest which contains only element nodes, in which all those element nodes have nodeName "p".

The notation $\Box_{(\text{D}_1 \to \text{D}_2)}$ is similar. For example, the formula $\neg\Diamond_{(\text{TXT}\to\text{F})}(\text{true}_{\text{TXT}}) \wedge \Box_{(\text{ELE}\to\text{F})}(\text{"p"}[\text{true}_{\text{F}}])$ describes a forest in which there are no text nodes, and all the elements have nodeName "p", and all the children of those elements are elements with nodeName "p", as are all their children and so on.

4.7. Program Reasoning

Program reasoning for Featherweight DOM follows the example of the program reasoning for the Basic Tree Update language (BTU) defined in Zarfaty's thesis[72]. Featherweight DOM includes several commands that have no counterpart in BTU, but the more significant difference is the extra complexity of the Featherweight DOM data structure. For example, the tree structure of BTU is uniform, meaning that there is no difference between a node at the top of the tree and a node arbitrarily deep in the tree. The semantics of BTU are defined on this uniform structure, which makes the definition of fault-avoiding Hoare triples in the style of O'Hearn et al[55] relatively simple. In contrast, the Featherweight DOM structure has a distinguished "grove layer" consisting of structures wrapped up using $<\ldots>_{\text{G}}$ and composed using $\oplus$. No node in the grove layer may have a parent, and the semantics of Featherweight DOM commands are only defined at

the grove level (See Definiions 11 and 15). The commands `appendChild`, `removeChild`, `createElement`, `createTextNode` do intuitively act at the grove level:

- `appendChild` takes a subtree from its current position and appends it to the subtree of an element node. The element node that serves as the new parent of the subtree might potentially be anywhere in the grove. There may be no common ancestor element of both the subtree to be moved and the new parent element, which means that the structure the command affects is a grove-level structure.
- `removeChild` takes a subtree from its current position and places it at the grove-level.
- `createElement` creates a new element node which is placed at the grove-level.
- `createTextNode` creates a new text node which is placed at the grove-level.

However, the commands `getNodeName`, `getChildNodes`, `item`, `substringData`, `appendData`, `deleteData` essentially act on specific subtrees identified by the command, rather than at the grove level. For example the command `getNodeName` returns the name of a particular node, whether that node is at the grove level or arbitrarily deep in the tree. The command `getParentNode` is a hybrid, having different behaviour at the subtree level (where it returns the parent) and the grove level (where it returns **null**). We therefore provide two forms of local Hoare triple, depending on whether we are reasoning about trees or groves. We use O'Hearn's fault-avoiding partial correctness interpretation of triples, which requires not only partial correctness, but also fault avoidance. This says that if a state satisfies a precondition, then the command cannot fault and the resulting state must satisfy the postcondition.

Definition 23 (Local Hoare Triples)**.** Given a Featherweight DOM command `C` and two grove formulae P:G, Q:G, the Hoare triple $\{P\}\texttt{C}\{Q\}$ is said to hold iff whenever $e, s, \mathbf{g} \models_{\mathrm{G}} P$ then:

partial correctness $\forall s', \mathbf{g}'.\ s, \mathbf{g}, \texttt{C} \rightsquigarrow s', \mathbf{g}' \Rightarrow e, s', \mathbf{g}' \models_{\mathrm{G}} Q$

fault avoidance $s, \mathbf{g}, \mathtt{C} \not\rightsquigarrow \text{fault}$

Similarly, given a Featherweight DOM command C and two tree formulae $P{:}\mathrm{T}, Q{:}\mathrm{T}$ where $\mathrm{T} \in \{\mathrm{ELE}, \mathrm{TXT}\}$, the Hoare triple $\{P\}\mathtt{C}\{Q\}$ is said to hold iff whenever $e, s, \mathbf{g} \models_{\mathrm{G}} <P>_{\mathrm{G}}$ then:

- $\forall s', \mathbf{g}'.\, s, \mathbf{g}, \mathtt{C} \rightsquigarrow s', \mathbf{g}' \Rightarrow e, s', \mathbf{g}' \models_{\mathrm{G}} <Q>_{\mathrm{G}}$ (partial correctness)
- $s, \mathbf{g}, \mathtt{C} \not\rightsquigarrow \text{fault}$ (fault avoidance)

Notice that our interpretation of the Hoare triples on trees coerces those trees to singleton groves using $<->_{\mathrm{G}}$. This is allows us to reason at the tree level even though $\rightsquigarrow$ is only defined for configuration triples containing groves.

Definition 24 (Command Axioms)**.** The axioms for the basic Featherweight DOM commands are given in Figures 4.11 and 4.12.

Notice the use of the type placeholder "D" in axioms such as that of `appendChild` which hold for multiple types. By way of illustration, the `appendChild` axiom is equivalent to the following four axioms:

$$\{(\varnothing_{\mathrm{G}} \multimap (\mathtt{C}{:}\mathrm{ELE}{\to}\mathrm{G} \circ_{\mathrm{ELE}} (\mathtt{NAME}_{\mathtt{parent}}[\mathtt{F}{:}\mathrm{F}]_{\mathtt{FID}}))) \circ_{\mathrm{G}} <\mathtt{NAME}'_{\mathtt{newChild}}[\mathtt{F}'{:}\mathrm{F}]_{\mathtt{FID}'} \wedge \mathtt{T}{:}\mathrm{ELE}>_{\mathrm{G}}\}$$
$$\mathtt{appendChild}(\mathtt{parent}, \mathtt{newChild})$$
$$\{\mathtt{C}{:}\mathrm{ELE}{\to}\mathrm{G} \circ_{\mathrm{ELE}} (\mathtt{NAME}_{\mathtt{parent}}[\mathtt{F}{:}\mathrm{F} \otimes_{\mathrm{F}} <\mathtt{T}{:}\mathrm{ELE}>_{\mathrm{F}}]_{\mathtt{FID}})\}$$

$$\{(\varnothing_{\mathrm{G}} \multimap (\mathtt{C}{:}\mathrm{ELE}{\to}\mathrm{G} \circ_{\mathrm{ELE}} (\mathtt{NAME}_{\mathtt{parent}}[\mathtt{F}{:}\mathrm{F}]_{\mathtt{FID}}))) \circ_{\mathrm{G}} <\text{“\#text”}_{\mathtt{newChild}}\mathtt{VAL} \wedge \mathtt{T}{:}\mathrm{TXT}>_{\mathrm{G}}\}$$
$$\mathtt{appendChild}(\mathtt{parent}, \mathtt{newChild})$$
$$\{\mathtt{C}{:}\mathrm{ELE}{\to}\mathrm{G} \circ_{\mathrm{ELE}} (\mathtt{NAME}_{\mathtt{parent}}[\mathtt{F}{:}\mathrm{F} \otimes_{\mathrm{F}} <\mathtt{T}{:}\mathrm{TXT}>_{\mathrm{F}}]_{\mathtt{FID}})\}$$

$$\{(\varnothing_{\mathrm{F}} \multimap (\mathtt{C}{:}\mathrm{ELE}{\to}\mathrm{G} \circ_{\mathrm{ELE}} (\mathtt{NAME}_{\mathtt{parent}}[\mathtt{F}{:}\mathrm{F}]_{\mathtt{FID}}))) \circ_{\mathrm{F}} <\mathtt{NAME}'_{\mathtt{newChild}}[\mathtt{F}'{:}\mathrm{F}]_{\mathtt{FID}'} \wedge \mathtt{T}{:}\mathrm{ELE}>_{\mathrm{F}}\}$$
$$\mathtt{appendChild}(\mathtt{parent}, \mathtt{newChild})$$
$$\{\mathtt{C}{:}\mathrm{ELE}{\to}\mathrm{G} \circ_{\mathrm{ELE}} (\mathtt{NAME}_{\mathtt{parent}}[\mathtt{F}{:}\mathrm{F} \otimes_{\mathrm{F}} <\mathtt{T}{:}\mathrm{ELE}>_{\mathrm{F}}]_{\mathtt{FID}})\}$$

$$\{(\varnothing_{\mathrm{F}} \multimap (\mathtt{C}{:}\mathrm{ELE}{\to}\mathrm{G} \circ_{\mathrm{ELE}} (\mathtt{NAME}_{\mathtt{parent}}[\mathtt{F}{:}\mathrm{F}]_{\mathtt{FID}}))) \circ_{\mathrm{F}} <\text{“\#text”}_{\mathtt{newChild}}\mathtt{VAL}>_{\mathrm{F}} \wedge \mathtt{T}{:}\mathrm{TXT}\}$$
$$\mathtt{appendChild}(\mathtt{parent}, \mathtt{newChild})$$
$$\{\mathtt{C}{:}\mathrm{ELE}{\to}\mathrm{G} \circ_{\mathrm{ELE}} (\mathtt{NAME}_{\mathtt{parent}}[\mathtt{F}{:}\mathrm{F} \otimes_{\mathrm{F}} <\mathtt{T}{:}\mathrm{TXT}>_{\mathrm{F}}]_{\mathtt{FID}})\}$$

$$\left\{ \begin{array}{c} (\varnothing_{\mathrm{D}} \multimap (\mathtt{C}{:}\mathrm{ELE}{\to}\mathrm{G} \circ_{\mathrm{ELE}} (\mathtt{NAME}_{\mathtt{parent}}[\mathtt{F}{:}\mathrm{F}]_{\mathtt{FID}}))) \circ_{\mathrm{D}} \\ <(\mathtt{NAME}'_{\mathtt{newChild}}[\mathtt{F}'{:}\mathrm{F}]_{\mathtt{FID}'} \vee \text{“\#text”}_{\mathtt{newChild}}\mathtt{VAL}) \wedge \mathtt{T}{:}\mathrm{D}'>_{\mathrm{D}} \end{array} \right\}$$
$$\mathtt{appendChild}(\mathtt{parent}, \mathtt{newChild})$$
$$\{\mathtt{C}{:}\mathrm{ELE}{\to}\mathrm{G} \circ_{\mathrm{ELE}} (\mathtt{NAME}_{\mathtt{parent}}[\mathtt{F}{:}\mathrm{F} \otimes_{\mathrm{F}} <\mathtt{T}{:}\mathrm{D}'>_{\mathrm{F}}]_{\mathtt{FID}})\}$$
$$\text{where } \mathrm{D} \in \{\mathrm{F}, \mathrm{G}\}, \mathrm{D}' \in \{\mathrm{ELE}, \mathrm{TXT}\}$$

$$\left\{ \begin{array}{l} <\mathtt{C}{:}\mathrm{ELE}{\to}\mathrm{G} \circ_{\mathrm{ELE}} (\mathtt{NAME}_{\mathtt{parent}}[\\ \quad \mathtt{F}_1{:}\mathrm{F} \otimes_{\mathrm{F}} <(\mathtt{NAME}'_{\mathtt{oldChild}}[\mathtt{F}'{:}\mathrm{F}]_{\mathtt{FID}'} \vee \text{“\#text”}_{\mathtt{oldChild}}\mathtt{VAL}) \wedge \mathtt{T}{:}\mathrm{D}>_{\mathrm{F}} \otimes_{\mathrm{F}} \mathtt{F}_2{:}\mathrm{F} \\ ]_{\mathtt{FID}})>_{\mathrm{G}} \end{array} \right\}$$
$$\mathtt{removeChild}(\mathtt{parent}, \mathtt{oldChild})$$
$$\{<\mathtt{C}{:}\mathrm{ELE}{\to}\mathrm{G} \circ_{\mathrm{ELE}} (\mathtt{NAME}_{\mathtt{parent}}[\mathtt{F}_1{:}\mathrm{F} \otimes_{\mathrm{F}} \mathtt{F}_2{:}\mathrm{F}]_{\mathtt{FID}})>_{\mathrm{G}} \oplus <\mathtt{T}{:}\mathrm{D}>_{\mathrm{G}}\}$$
$$\text{where } \mathrm{D} \in \{\mathrm{ELE}, \mathrm{TXT}\}$$

$$\{(\mathtt{NAME}_{\mathtt{node}}[\mathtt{F}{:}\mathrm{F}]_{\mathtt{FID}} \vee \mathtt{NAME}_{\mathtt{node}}\mathtt{VAL}) \wedge \mathtt{T}{:}\mathrm{D}\}$$
$$\mathtt{var} := \mathtt{getNodeName}(\mathtt{node})$$
$$\{\mathtt{T}{:}\mathrm{D} \wedge (\mathtt{var} \doteq \mathtt{NAME})\}$$
$$\text{where } \mathrm{D} \in \{\mathrm{ELE}, \mathrm{TXT}\}$$

$$\{\mathtt{NAME}'_{\mathtt{ID}'}[\mathtt{F}_1{:}\mathrm{F} \otimes_{\mathrm{F}} <(\mathtt{NAME}_{\mathtt{node}}[\mathtt{F}{:}\mathrm{F}]_{\mathrm{FID}} \vee \text{“\#text”}_{\mathtt{node}}\mathtt{VAL}) \wedge \mathtt{T}{:}\mathrm{D}>_{\mathrm{F}} \otimes_{\mathrm{F}} \mathtt{F}_2{:}\mathrm{F}]_{\mathtt{FID}'}\}$$
$$\mathtt{var} := \mathtt{getParentNode}(\mathtt{node})$$
$$\{\mathtt{NAME}'_{\mathtt{ID}'}[\mathtt{F}_1{:}\mathrm{F} \otimes_{\mathrm{F}} <\mathtt{T}{:}\mathrm{D}>_{\mathrm{F}} \otimes_{\mathrm{F}} \mathtt{F}_2{:}\mathrm{F}]_{\mathtt{FID}'} \wedge (\mathtt{var} \doteq \mathtt{ID}')\}$$
$$\text{where } \mathrm{D} \in \{\mathrm{ELE}, \mathrm{TXT}\}$$

$$\{<(\mathtt{NAME}_{\mathtt{node}}[\mathtt{F}{:}\mathrm{F}]_{\mathrm{FID}} \vee \text{“\#text”}_{\mathtt{node}}\mathtt{VAL}) \wedge \mathtt{T}{:}\mathrm{D}>_{\mathrm{G}}\}$$
$$\mathtt{var} := \mathtt{getParentNode}(\mathtt{node})$$
$$\{<\mathtt{T}{:}\mathrm{D}>_{\mathrm{G}} \wedge (\mathtt{var} \doteq \mathbf{null})\}$$
$$\text{where } \mathrm{D} \in \{\mathrm{ELE}, \mathrm{TXT}\}$$

$$\{\mathtt{NAME}_{\mathtt{Y}}[\mathtt{F}{:}\mathrm{F}]_{\mathtt{FID}} \wedge \mathtt{node} \doteq \mathtt{Y}\}$$
$$\mathtt{var} := \mathtt{getChildNodes}(\mathtt{node})$$
$$\{\mathtt{NAME}_{\mathtt{Y}}[\mathtt{F}{:}\mathrm{F}]_{\mathtt{FID}} \wedge (\mathtt{var} \doteq \mathtt{FID})\}$$

$$\{\varnothing_{\mathrm{G}} \wedge \mathtt{Name} \in \mathrm{S} \wedge \text{`\#'} \notin \mathtt{Name} \wedge \mathtt{var} \doteq \mathtt{Y}\}$$
$$\mathtt{var} := \mathtt{createElement}(\mathtt{Name})$$
$$\{<\mathtt{Name}\{\mathtt{Y}/\mathtt{var}\}_{\mathtt{var}}[\varnothing_{\mathrm{F}}]_{\mathtt{FID}}>_{\mathrm{G}}\}$$

$$\left\{ \begin{array}{l} \mathtt{NAME}_{\mathtt{ID}}[\mathtt{F}_1{:}\mathrm{F} \otimes_{\mathrm{F}} <(\mathtt{NAME}'_{\mathtt{ID}'}[\mathtt{F}'{:}\mathrm{F}]_{\mathrm{FID}'} \vee \text{“\#text”}_{\mathtt{ID}'}\mathtt{VAL}') \wedge \mathtt{T}{:}\mathrm{D}>_{\mathrm{F}} \otimes_{\mathrm{F}} \mathtt{F}_2{:}\mathrm{F}]_{\mathtt{list}} \\ \quad \wedge (\mathtt{Int} \doteq \mathrm{len}(\mathtt{F}_1)) \wedge \mathtt{Y} \doteq \mathtt{list} \end{array} \right\}$$
$$\mathtt{var} := \mathtt{item}(\mathtt{list}, \mathtt{Int})$$
$$\{\mathtt{NAME}_{\mathtt{ID}}[\mathtt{F}_1{:}\mathrm{F} \otimes_{\mathrm{F}} <\mathtt{T}{:}\mathrm{D}>_{\mathrm{F}} \otimes_{\mathrm{F}} \mathtt{F}_2{:}\mathrm{F}]_{\mathtt{Y}} \wedge (\mathtt{var} \doteq \mathtt{ID}')\}$$
$$\text{where } \mathrm{D} \in \{\mathrm{ELE}, \mathrm{TXT}\}$$

$$\{\mathtt{NAME}_{\mathtt{ID}}[\mathtt{F}{:}\mathrm{F}]_{\mathtt{list}} \wedge \mathtt{list} \doteq \mathtt{Y} \wedge (\mathtt{Int} <\cdot\, 0 \vee \mathtt{Int} >\doteq \mathrm{len}(\mathtt{F}))\}$$
$$\mathtt{var} := \mathtt{item}(\mathtt{list}, \mathtt{Int})$$
$$\{\mathtt{NAME}_{\mathtt{ID}}[\mathtt{F}{:}\mathrm{F}]_{\mathtt{Y}} \wedge (\mathtt{var} \doteq \mathbf{null})\}$$

Figure 4.11.: Featherweight DOM Axioms (Part 1)

$$\left\{\begin{array}{r}\text{“\#text”}_{\texttt{node}}(\texttt{STR}_1 \otimes_{\mathrm{S}} \texttt{STR}' \otimes_{\mathrm{S}} \texttt{STR}_2) \wedge (\texttt{Offset} \doteq \mathrm{len}(\texttt{STR}_1)) \\ \wedge\, (\texttt{Count} \doteq \mathrm{len}(\texttt{STR}') \wedge \texttt{node} \doteq \texttt{Y})\end{array}\right\}$$
$$\texttt{var} := \texttt{substringData}(\texttt{node}, \texttt{Offset}, \texttt{Count})$$
$$\{\text{“\#text”}_{\texttt{Y}}(\texttt{STR}_1 \otimes_{\mathrm{S}} \texttt{STR}' \otimes_{\mathrm{S}} \texttt{STR}_2) \wedge (\texttt{var} \doteq \texttt{STR}')\}$$

$$\left\{\begin{array}{r}\text{“\#text”}_{\texttt{node}}(\texttt{STR}_1 \otimes_{\mathrm{S}} \texttt{STR}') \wedge (\texttt{Offset} \doteq \mathrm{len}(\texttt{STR}_1)) \\ \wedge\, (\texttt{Count} \cdot > \mathrm{len}(\texttt{STR}')) \wedge \texttt{node} \doteq \texttt{Y}\end{array}\right\}$$
$$\texttt{var} := \texttt{substringData}(\texttt{node}, \texttt{Offset}, \texttt{Count})$$
$$\{\text{“\#text”}_{\texttt{Y}}(\texttt{STR}_1 \otimes_{\mathrm{S}} \texttt{STR}') \wedge (\texttt{var} \doteq \texttt{STR}')\}$$

$$\{\text{“\#text”}_{\texttt{node}}\texttt{STR} \wedge \texttt{Arg} \in \mathrm{S}\}$$
$$\texttt{appendData}(\texttt{node}, \texttt{Arg})$$
$$\{\text{“\#text”}_{\texttt{node}}(\texttt{STR} \otimes_{\mathrm{S}} \texttt{Arg})\}$$

$$\left\{\begin{array}{r}\text{“\#text”}_{\texttt{node}}(\texttt{STR}_1 \otimes_{\mathrm{S}} \texttt{STR} \otimes_{\mathrm{S}} \texttt{STR}_2) \wedge (\texttt{Offset} \doteq \mathrm{len}(\texttt{STR}_1)) \\ \wedge\, (\texttt{Count} \doteq \mathrm{len}(\texttt{STR}))\end{array}\right\}$$
$$\texttt{deleteData}(\texttt{node}, \texttt{Offset}, \texttt{Count})$$
$$\{\text{“\#text”}_{\texttt{node}}(\texttt{STR}_1 \otimes_{\mathrm{S}} \texttt{STR}_2)\}$$

$$\left\{\begin{array}{r}\text{“\#text”}_{\texttt{node}}(\texttt{STR}_1 \otimes_{\mathrm{S}} \texttt{STR}) \wedge (\texttt{Offset} \doteq \mathrm{len}(\texttt{STR}_1)) \\ \wedge\, (\texttt{Count} \cdot > \mathrm{len}(\texttt{STR}))\end{array}\right\}$$
$$\texttt{deleteData}(\texttt{node}, \texttt{Offset}, \texttt{Count})$$
$$\{\text{“\#text”}_{\texttt{node}}\texttt{STR}_1\}$$

$$\{\varnothing_{\mathrm{G}} \wedge \texttt{var} \doteq \texttt{Y} \wedge \texttt{Str} \in \mathrm{S}\}$$
$$\texttt{var} := \texttt{createTextNode}(\texttt{Str})$$
$$\{<\text{“\#text”}_{\texttt{var}}\texttt{Str}\{\texttt{Y}/\texttt{var}\}>_{\mathrm{G}}\}$$

$$\{\varnothing_{\mathrm{G}} \wedge (\texttt{Y} \doteq \texttt{Exp})\}$$
$$\texttt{var} := \texttt{Exp}$$
$$\{\varnothing_{\mathrm{G}} \wedge (\texttt{var} \doteq \texttt{Y})\}$$

$$\{\varnothing_{\mathrm{G}}\}$$
$$\texttt{skip}$$
$$\{\varnothing_{\mathrm{G}}\}$$

Figure 4.12.: Featherweight DOM Axioms (Part 2)

All of these "`appendChild`" axioms make use of the pattern "$(\varnothing \multimap (K \circ \text{parent})) \circ \text{newChild}$" in the precondition to assert that "newChild" is not an ancestor of "parent". They also use the logical variables such as `C` and `F` to assert that portions of the grove remain unchanged by the command. Finally, they all use the logical variable `T` to assert that the "newChild" node remains internally unchanged despite its move into the children of "parent".

The axiom of "`removeChild`" is similar in that it uses a type placeholder "D" and describes the movement of a node in the grove. It is different in that the move is always to the grove-level, rather than to the child-list of any other element. As a result there is no need to specify that any node is not an ancestor of the node being moved, and so there is no need to use "$\varnothing \multimap$".

Definition 25 (Inference Rules)**.** The local reasoning inference rules include the standard Hoare Logic Rules for Sequencing, If-Then-Else, While-Do, Local Block, Consequence, Disjunction, Auxiliary Variable Elimination, and a local reasoning rule called the Frame Rule. They are given in Figure 4.13.

4.8. Soundness

4.8.1. Defining Soundness

In order for any Hoare reasoning system to be of any use, we require that for every triple $\{P\}\mathtt{C}\{Q\}$ which can be derived in the system, and for every concrete state which satisfies P, the result of successfully running the command `C` in that state is a new concrete state which satisfies Q. More formally, we must prove the interpretation of Hoare Triples which is given for Featherweight DOM in Definition 23. For Featherweight DOM, that interpretation depends on the types of the formulae P and Q, which may be either tree or grove formulae. If P and Q are grove formulae, then the interpretation of the triple $\{P\}\mathtt{C}\{Q\}$ consists of partial correctness and fault avoidance as follows:

partial correctness $\forall s', \mathbf{g}'.\, s, \mathbf{g}, \mathtt{C} \rightsquigarrow s', \mathbf{g}' \Rightarrow e, s', \mathbf{g}' \models_{\mathrm{G}} Q$

fault avoidance $s, \mathbf{g}, \mathtt{C} \not\rightsquigarrow \text{fault}$

If the formulae P and Q are tree formulae, then the interpretation of the triple $\{P\}\mathtt{C}\{Q\}$ consists of partial correctness and fault avoidance as above,

SEQUENCE: $$\frac{\{P\}\mathtt{C}_1\{Q\} \quad \{Q\}\mathtt{C}_2\{R\}}{\{P\}\mathtt{C}_1\ ;\ \mathtt{C}_2\{R\}}$$

IF THEN ELSE: $$\frac{\{(\mathtt{Bool} \doteq \text{true}) \wedge P\}\mathtt{C}_1\{Q\} \quad \{(\mathtt{Bool} \doteq \text{false}) \wedge P\}\mathtt{C}_2\{Q\}}{\{P\}\mathtt{if\ Bool\ then\ C}_1\ \mathtt{else\ C}_2\ \mathtt{fi}\ \{Q\}}$$

WHILE: $$\frac{\{(\mathtt{Bool} \doteq \text{true}) \wedge P\}\mathtt{C}\{P\}}{\{P\}\mathtt{while\ Bool\ do\ C\ od}\ \{(\mathtt{Bool} \doteq \text{false}) \wedge P\}}$$

LOCAL BLOCK: $$\frac{\{\mathtt{var} = \mathbf{null} \wedge P\}\mathtt{C}\{Q\}}{\{P\}\mathtt{local\ var : C\ endloc}\ \{Q\}} \quad \mathtt{var} \notin \text{free}(P, Q)$$

$$\frac{\{P\}\mathtt{local\ var' : C\{var'/var\}\ endloc}\ \{Q\}}{\{P\}\mathtt{local\ var : C\ endloc}\ \{Q\}} \quad \mathtt{var'} \notin \text{free}(\mathtt{C})$$

CONSEQUENCE: $$\frac{P' \Rightarrow P \quad \{P\}\mathtt{C}\{Q\} \quad Q \Rightarrow Q'}{\{P'\}\mathtt{C}\{Q'\}}$$

DISJUNCTION: $$\frac{\{P\}\mathtt{C}\{Q\} \quad \{P'\}\mathtt{C}\{Q'\}}{\{P \vee P'\}\mathtt{C}\{Q \vee Q'\}}$$

AUX VAR ELIM: $$\frac{\{P\}\mathtt{C}\{Q\}}{\{\exists \mathtt{VAR}.\ P\}\mathtt{C}\{\exists \mathtt{VAR}.\ Q\}}$$

FRAME RULE: $$\frac{\{P\}\mathtt{C}\{Q\}}{\{K \circ_{\mathrm{D}} P\}\mathtt{C}\{K \circ_{\mathrm{D}} Q\}} \quad \begin{array}{l}\text{mod}(\mathtt{C}) \cap \text{free}(K) = \emptyset \\ \mathrm{D} \in \{\mathrm{ELE}, \mathrm{TXT}, \mathrm{G}\}\end{array}$$

Where free(K) is the set of free variables and mod(C) is the set of variables changed by C.

Figure 4.13.: Inference Rules for Featherweight DOM

but with the data lifted to the grove level so as to match the operational semantics which are defined on groves. This is a small but significant difference between Featherweight DOM and previous work, in which there was only one sort of triple, and therefore only one definition of partial correctness and fault avoidance. The significance of this difference is explored in the following sections, which detail the techniques used to prove the soundness of previous systems, and one crucial problem that occurs when trying to use those techniques to prove a sound system for reasoning about Featherweight DOM.

4.8.2. An Existing Approach to Soundness for Local Reasoning

In [72], Zarfaty proves the soundness of the Frame Rule in his system by using a natural adaptation of the technique first introduced in [55]. That technique is to introduce a definition of a "local command", to prove that all programs written in the programming language are "local", and then to prove that the Frame Rule is correct when used to reason about local programs.

4.8.3. The Problem with the Existing Approach

Zarfaty relied on a definition of locality that comprised two properties:

safety monotonicity $s, \mathbf{t}, \mathtt{C} \not\rightarrow \text{fault} \wedge \text{ap}(\mathbf{c}, \mathbf{t})\downarrow \implies s, \text{ap}(\mathbf{c}, \mathbf{t}), \mathtt{C} \not\rightarrow \text{fault}$

frame property $s, \mathbf{t}, \mathtt{C} \not\rightarrow \text{fault} \wedge \text{ap}(\mathbf{c}, \mathbf{t})\downarrow \wedge s, \text{ap}(\mathbf{c}, \mathbf{t}), \mathtt{C} \rightsquigarrow s', \mathbf{t}' \implies \exists \mathbf{t}''$ such that $s, \mathbf{t}, \mathtt{C} \rightsquigarrow s', \mathbf{t}'' \wedge \mathbf{t}' \equiv \text{ap}(\mathbf{c}, \mathbf{t}'')$

Consider naive adaptations of these properties into the context of Featherweight DOM, using $\mathbf{t}$ to denote a concrete DOM element or text node, $\mathbf{g}$ for a grove and $\mathbf{cg}$ for a grove context, which may have any type of hole:

Safety Monotonicity

$$(s, \mathbf{g}, \mathtt{C} \not\rightarrow \text{fault} \wedge \text{ap}(\mathbf{cg}, \mathbf{g})\downarrow \implies s, \text{ap}(\mathbf{cg}, \mathbf{g}), \mathtt{C} \not\rightarrow \text{fault})$$
$$\wedge\ (s, <\mathbf{t}>_{\text{G}}, \mathtt{C} \not\rightarrow \text{fault} \wedge \text{ap}(\mathbf{cg}, \mathbf{t})\downarrow \implies s, \text{ap}(\mathbf{cg}, \mathbf{t}), \mathtt{C} \not\rightarrow \text{fault})$$

Frame Property

$$
\begin{array}{l}
(s, \mathbf{g}, \mathtt{C} \not\rightsquigarrow \text{fault} \wedge \text{ap}(\mathbf{cg}, \mathbf{g})\downarrow \wedge s, \text{ap}(\mathbf{cg}, \mathbf{g}), \mathtt{C} \rightsquigarrow s', \mathbf{g}' \implies \exists \mathbf{g}'' \\
\quad \text{such that } \mathbf{g}' \equiv \text{ap}(\mathbf{cg}, \mathbf{g}'') \wedge s, \mathbf{g}, \mathtt{C} \rightsquigarrow s', \mathbf{g}'') \\
\wedge (s, <\mathbf{t}>_{\text{G}}, \mathtt{C} \not\rightsquigarrow \text{fault} \wedge \text{ap}(\mathbf{cg}, \mathbf{t})\downarrow \wedge s, \text{ap}(\mathbf{cg}, \mathbf{t}), \mathtt{C} \rightsquigarrow s', \mathbf{g}' \implies \exists \mathbf{t}'' \\
\quad \text{such that } \mathbf{g}' \equiv \text{ap}(\mathbf{cg}, \mathbf{t}'') \wedge s, <\mathbf{t}>_{\text{G}}, \mathtt{C} \rightsquigarrow s', <\mathbf{t}''>_{\text{G}})
\end{array}
$$

For languages in which all commands satisfy these properties, it is possible to prove the soundness of the frame rule using the same technique as Zarfaty used in [72]. Unfortunately, DOM specifies one essential command which does not satisfy these properties: `getParentNode`. This command is defined by the following two operational rules:

$$
\frac{\begin{pmatrix} \mathbf{t} \equiv \mathbf{name}'_{s(\mathtt{node})}[\mathbf{f}]'_{\mathbf{fid}} \\ \vee\, \mathbf{t} \equiv \text{“\#text”}_{s(\mathtt{node})}\,\text{“}\mathbf{str}\text{”} \end{pmatrix} \quad \mathbf{g} \equiv \text{ap}(\mathbf{cg}, \mathbf{name}_{\mathbf{id}}[\mathbf{f}_1 \otimes_{\text{F}} <\mathbf{t}>_{\text{F}} \otimes_{\text{F}} \mathbf{f}_2]_{\mathbf{fid}})}{s, \mathbf{g}, \mathtt{var} := \mathtt{getParentNode}(\mathtt{node}) \rightsquigarrow [s|\mathtt{var} \leftarrow \mathbf{id}], \mathbf{g}}
$$

$$
\frac{\begin{pmatrix} \mathbf{t} \equiv \mathbf{name}'_{s(\mathtt{node})}[\mathbf{f}]'_{\mathbf{fid}} \\ \vee\, \mathbf{t} \equiv \text{“\#text”}_{s(\mathtt{node})}\,\text{“}\mathbf{str}\text{”} \end{pmatrix} \quad \mathbf{g} \equiv \text{ap}(\mathbf{cg}, <\mathbf{t}>_{\text{G}})}{s, \mathbf{g}, \mathtt{var} := \mathtt{getParentNode}(\mathtt{node}) \rightsquigarrow [s|\mathtt{var} \leftarrow \mathbf{null}], \mathbf{g}}
$$

Consider the behaviour of the command `x := getParentNode(y)` when run on a grove $<\text{“parent”}_1[<\text{“child”}_2[\varnothing_{\text{F}}]_3>_{\text{F}}]_4>_{\text{G}}$ and store in which y maps to 2. After running the command, the variable x will be equal to 1. Now consider the Frame Property. If we choose **cg** to be equal to $\text{“parent”}_1[<-_{\text{ELE}}>_{\text{F}}]_4$ and **t** to be equal to $\text{“child”}_2[\varnothing_{\text{F}}]_3$, the frame property then requires that the behaviour of the `getParentNode` command on the grove $<\mathbf{t}>_{\text{G}}$ also result in the variable x being set equal to 1. In fact, the behaviour of this command on this grove is to set x to be equal to **null**.

It is clear that, at least given this definition of locality, the `getParentNode` command is not local. Rather than try to find a definition of locality which does allow the `getParentNode` command, we take this opportunity to present a method for allowing local reasoning about non-local commands.

4.8.4. A New Approach

Outline

The proof of soundness of reasoning with context logic for Featherweight DOM follows the proof of soundness of reasoning with context logic about the Basic Tree Update (BTU) language provided by Zarfaty [72]. However while Zarfaty's tree structure was uniform, the Featherweight DOM structure is not. As we have seen, some Featherweight DOM commands act at the tree level, some at the grove level and some at both.

Both BTU and Featherweight DOM have only one non-standard inference rule, and this is the Frame Rule. The most interesting part of Zarfaty's soundness proof is his handling of this rule, which in turn closely follows the approach of O'Hearn et al. That approach is to define a notion of a "local command", and to prove that the Frame Rule is sound when reasoning about local commands.

Since Zarfaty's tree structure is uniform, a uniform notion of locality is natural. However, the Featherweight DOM structure is not uniform and calls for different notions of locality at the grove and tree levels.

The command `getParentNode` is particularly problematic, since it operates at both the grove level *and* the tree level and behaves differently at different levels. In order to prove the Frame Rule sound for commands like this, we need to know which sort of locality we are dealing with in each case.

Our approach is to define locality not simply as a property of a command, but as a property of a command with respect to a formula. In this section, we define what it means for a command `C` to be local to all groves or trees described by a formula P. We show that if a command is local to a formula P, then it is also local to extensions of that formula $K \circ_{\mathrm{D}} P$. We show that all Featherweight DOM commands are local to the preconditions of their axioms. Finally, we show that all Featherweight DOM inference rules propagate the property of locality and that the frame rule is sound for reasoning about commands which are local to their precondition.

4.8.5. Defining Locality

First we define what it means for a command to be local with respect to a formula, and we show an important property of locality.

Definition 26 (Locality)**.** Locality is defined differently at the grove and tree levels.

A command $\mathtt{C}$ is local to a footprint described by formula P where P:G if it satisfies the following two properties:

grove-level safety-monotonicity

$$
\begin{array}{l}
s, \mathbf{d}, \mathtt{C} \not\rightsquigarrow \text{fault} \wedge \text{ap}(\mathbf{c}, \mathbf{d})\downarrow \wedge \\
(\exists e, \mathbf{d}_{\text{foot}}, \mathbf{c}_{\text{surplus}}.\, \mathbf{d} \equiv \text{ap}(\mathbf{c}_{\text{surplus}}, \mathbf{d}_{\text{foot}}) \wedge e, s, \mathbf{d}_{\text{foot}} \models_{\text{G}} P) \\
\quad \Longrightarrow\ s, \text{ap}(\mathbf{c}, \mathbf{d}), \mathtt{C} \not\rightsquigarrow \text{fault}
\end{array}
$$

grove-level frame property

$$
\begin{array}{l}
s, \mathbf{d}, \mathtt{C} \not\rightsquigarrow \text{fault} \wedge \text{ap}(\mathbf{c}, \mathbf{d})\downarrow \wedge s, \text{ap}(\mathbf{c}, \mathbf{d}), \mathtt{C} \rightsquigarrow s', \mathbf{d}_2 \wedge \\
(\exists e, \mathbf{d}_{\text{foot}}, \mathbf{c}_{\text{surplus}}.\, \mathbf{d} \equiv \text{ap}(\mathbf{c}_{\text{surplus}}, \mathbf{d}_{\text{foot}}) \wedge e, s, \mathbf{d}_{\text{foot}} \models_{\text{G}} P) \\
\quad \Longrightarrow\ \exists \mathbf{d}_3.\, s, \mathbf{d}, \mathtt{C} \rightsquigarrow s', \mathbf{d}_3 \wedge \mathbf{d}_2 \equiv \text{ap}(\mathbf{c}, \mathbf{d}_3)
\end{array}
$$

A command $\mathtt{C}$ is local to a footprint described by formula P where P:T, T $\in$ {ELE, TXT} if it satisfies the following four properties:

tree-level safety-monotonicity

$$
\begin{array}{l}
s, <\mathbf{d}>_{\text{G}}, \mathtt{C} \not\rightsquigarrow \text{fault} \wedge \text{ap}(\mathbf{c}, \mathbf{d})\downarrow \wedge \mathbf{c}{:}(\text{D}{\rightarrow}\text{G}) \wedge \\
(\exists e, \mathbf{d}_{\text{foot}}, \mathbf{c}_{\text{surplus}}.\, \mathbf{d} \equiv \text{ap}(\mathbf{c}_{\text{surplus}}, \mathbf{d}_{\text{foot}}) \wedge e, s, \mathbf{d}_{\text{foot}} \models_{\text{T}} P) \\
\quad \Longrightarrow\ s, \text{ap}(\mathbf{c}, \mathbf{d}), \mathtt{C} \not\rightsquigarrow \text{fault}
\end{array}
$$

tree-level frame property

$$
\begin{array}{l}
s, <\mathbf{d}>_{\text{G}}, \mathtt{C} \not\rightsquigarrow \text{fault} \wedge \text{ap}(\mathbf{c}, \mathbf{d})\downarrow \wedge s, \text{ap}(\mathbf{c}, \mathbf{d}), \mathtt{C} \rightsquigarrow s', \mathbf{d}_2 \wedge \\
(\exists e, \mathbf{d}_{\text{foot}}, \mathbf{c}_{\text{surplus}}.\, \mathbf{d} \equiv \text{ap}(\mathbf{c}_{\text{surplus}}, \mathbf{d}_{\text{foot}}) \wedge e, s, \mathbf{d}_{\text{foot}} \models_{\text{T}} P) \\
\Longrightarrow \\
\exists \mathbf{d}_3.\, s, <\mathbf{d}>_{\text{G}}, \mathtt{C} \rightsquigarrow s', <\mathbf{d}_3>_{\text{G}} \wedge \mathbf{d}_2 \equiv \text{ap}(\mathbf{c}, \mathbf{d}_3)
\end{array}
$$

node-grove-level safety-monotonicity

$$
\begin{array}{l}
s, \mathbf{d}, \mathtt{C} \not\rightsquigarrow \text{fault} \wedge \text{ap}(\mathbf{c}, \mathbf{d})\downarrow \wedge \mathbf{c}{:}(\text{D}{\rightarrow}\text{G}) \wedge \\
(\exists e, \mathbf{d}_{\text{foot}}, \mathbf{c}_{\text{surplus}}.\, \mathbf{d} \equiv \text{ap}(\mathbf{c}_{\text{surplus}}, \mathbf{d}_{\text{foot}}) \wedge e, s, \mathbf{d}_{\text{foot}} \models_{\text{T}} P) \\
\quad \Longrightarrow\ s, \text{ap}(\mathbf{c}, \mathbf{d}), \mathtt{C} \not\rightsquigarrow \text{fault}
\end{array}
$$

node-grove-level frame property

$$
\begin{array}{l}
s, \mathbf{d}, \mathtt{C} \not\rightsquigarrow \text{fault} \wedge \text{ap}(\mathbf{c}, \mathbf{d})\downarrow \wedge s, \text{ap}(\mathbf{c}, \mathbf{d}), \mathtt{C} \rightsquigarrow s', \mathbf{d}_2 \wedge \\
(\exists e, \mathbf{d}_{\text{foot}}, \mathbf{c}_{\text{surplus}}.\, \mathbf{d} \equiv \text{ap}(\mathbf{c}_{\text{surplus}}, \mathbf{d}_{\text{foot}}) \wedge e, s, \mathbf{d}_{\text{foot}} \models_{\text{T}} P) \\
\Longrightarrow \\
\exists \mathbf{d}_3.\, s, \mathbf{d}, \mathtt{C} \rightsquigarrow s', \mathbf{d}_3 \wedge \mathbf{d}_2 \equiv \text{ap}(\mathbf{c}, \mathbf{d}_3)
\end{array}
$$

In order to show the soundness of the frame rule, we will require that if a command $\mathtt{C}$ is local in the triple $\{P\}\mathtt{C}\{Q\}$ then it must also be local in the extended triple $\{K \circ_{\text{D}} P\}\mathtt{C}\{K \circ_{\text{D}} Q\}$. This property is the monotonicity of locality as defined below.

Lemma 27 (Monotonicity of Locality)**.** *If a command $\mathtt{C}$ is local to a footprint described by P then it is also local to all extensions of that footprint described by $K \circ_{\text{D}} P$.*

Proof. Consider the three possible cases separately: The case in which P:G ; the case in which P:T and K:(T→T′) for $\text{T}, \text{T}' \in \{\text{ELE}, \text{TXT}\}$; and the case in which P:T and K:(T→G) for $\text{T} \in \{\text{ELE}, \text{TXT}\}$.

- First, consider the case in which P:G, Q:G.

  The aim is to prove that grove-level safety-monotonicity of $\mathtt{C}$ with respect to P implies grove-level safety-monotonicity of $\mathtt{C}$ with respect to $K \circ_{\text{G}} P$ and that the grove-level frame property of $\mathtt{C}$ with respect to P implies the grove-level frame-property of $\mathtt{C}$ with respect to $K \circ_{\text{G}} P$.

  grove-level safety-monotonicity
  From the given premise:

  $$
  \begin{array}{l}
  s, \mathbf{d}, \mathtt{C} \not\rightsquigarrow \text{fault} \wedge \text{ap}(\mathbf{c}, \mathbf{d})\downarrow \wedge \\
  (\exists e, \mathbf{d}_{\text{foot}}, \mathbf{c}_{\text{surplus}}.\, \mathbf{d} \equiv \text{ap}(\mathbf{c}_{\text{surplus}}, \mathbf{d}_{\text{foot}}) \wedge e, s, \mathbf{d}_{\text{foot}} \models_{\text{G}} P) \\
  \Longrightarrow\ s, \text{ap}(\mathbf{c}, \mathbf{d}), \mathtt{C} \not\rightsquigarrow \text{fault}
  \end{array}
  $$

  In order the show the conclusion:

  $$
  \begin{array}{l}
  s, \mathbf{d}, \mathtt{C} \not\rightsquigarrow \text{fault} \wedge \text{ap}(\mathbf{c}, \mathbf{d})\downarrow \wedge \\
  (\exists e, \mathbf{d}_{\text{foot}}, \mathbf{c}_{\text{surplus}}.\, \mathbf{d} \equiv \text{ap}(\mathbf{c}_{\text{surplus}}, \mathbf{d}_{\text{foot}}) \wedge e, s, \mathbf{d}_{\text{foot}} \models_{\text{G}} K \circ_{\text{G}} P) \\
  \Longrightarrow\ s, \text{ap}(\mathbf{c}, \mathbf{d}), \mathtt{C} \not\rightsquigarrow \text{fault}
  \end{array}
  $$

It is sufficient to show:

$$(\exists e, \mathbf{d}_{\text{foot}}, \mathbf{c}_{\text{surplus}}.\, \mathbf{d} \equiv \text{ap}(\mathbf{c}_{\text{surplus}}, \mathbf{d}_{\text{foot}}) \wedge e, s, \mathbf{d}_{\text{foot}} \models_{\text{G}} K \circ_{\text{G}} P)$$
$$\implies$$
$$(\exists e, \mathbf{d}_{\text{foot}}, \mathbf{c}_{\text{surplus}}.\, \mathbf{d} \equiv \text{ap}(\mathbf{c}_{\text{surplus}}, \mathbf{d}_{\text{foot}}) \wedge e, s, \mathbf{d}_{\text{foot}} \models_{\text{G}} P)$$

By the definition of context application, it can be seen that:

$$(\exists e, \mathbf{d}_{\text{foot}}, \mathbf{c}_{\text{surplus}}, \mathbf{d}'_{\text{foot}}, \mathbf{c}'_{\text{surplus}}.\, \mathbf{d} \equiv \text{ap}(\mathbf{c}_{\text{surplus}}, \mathbf{d}_{\text{foot}}) \wedge$$
$$\mathbf{d}_{\text{foot}} \equiv \text{ap}(\mathbf{c}'_{\text{surplus}}, \mathbf{d}'_{\text{foot}}) \wedge e, s, \mathbf{d}'_{\text{foot}} \models_{\text{G}} P)$$
$$\implies$$
$$(\exists e, \mathbf{d}_{\text{foot}}, \mathbf{c}_{\text{surplus}}.\, \mathbf{d} \equiv \text{ap}(\mathbf{c}_{\text{surplus}}, \mathbf{d}_{\text{foot}}) \wedge e, s, \mathbf{d}_{\text{foot}} \models_{\text{G}} P)$$

Since $\mathbf{d} \equiv \text{ap}(\mathbf{c}_{\text{surplus}}, \mathbf{d}_{\text{foot}})$ and $\mathbf{d}_{\text{foot}} \equiv \text{ap}(\mathbf{c}'_{\text{surplus}}, \mathbf{d}'_{\text{foot}})$ it is possible to choose $\mathbf{c}''_{\text{surplus}}$ such that $\mathbf{d} \equiv \text{ap}(\mathbf{c}''_{\text{surplus}}, \mathbf{d}'_{\text{foot}})$. This leads to:

$$(\exists e, \mathbf{d}'_{\text{foot}}, \mathbf{c}''_{\text{surplus}}.\, \mathbf{d} \equiv \text{ap}(\mathbf{c}''_{\text{surplus}}, \mathbf{d}'_{\text{foot}}) \wedge e, s, \mathbf{d}'_{\text{foot}} \models_{\text{G}} P)$$
$$\implies$$
$$(\exists e, \mathbf{d}_{\text{foot}}, \mathbf{c}_{\text{surplus}}.\, \mathbf{d} \equiv \text{ap}(\mathbf{c}_{\text{surplus}}, \mathbf{d}_{\text{foot}}) \wedge e, s, \mathbf{d}_{\text{foot}} \models_{\text{G}} P)$$

which is true by the definition of existential quantification.

grove-level frame property
From the given premise:

$$s, \mathbf{d}, \mathtt{C} \not\rightsquigarrow \text{fault} \wedge \text{ap}(\mathbf{c}, \mathbf{d})\downarrow \wedge s, \text{ap}(\mathbf{c}, \mathbf{d}), \mathtt{C} \rightsquigarrow s', \mathbf{d}_2 \wedge$$
$$(\exists e, \mathbf{d}_{\text{foot}}, \mathbf{c}_{\text{surplus}}.\, \mathbf{d} \equiv \text{ap}(\mathbf{c}_{\text{surplus}}, \mathbf{d}_{\text{foot}}) \wedge e, s, \mathbf{d}_{\text{foot}} \models_{\text{G}} P)$$
$$\implies \exists \mathbf{d}_3.\, s, \mathbf{d}, \mathtt{C} \rightsquigarrow s', \mathbf{d}_3 \wedge \mathbf{d}_2 \equiv \text{ap}(\mathbf{c}, \mathbf{d}_3)$$

In order to show the conclusion:

$$s, \mathbf{d}, \mathtt{C} \not\rightsquigarrow \text{fault} \wedge \text{ap}(\mathbf{c}, \mathbf{d})\downarrow \wedge s, \text{ap}(\mathbf{c}, \mathbf{d}), \mathtt{C} \rightsquigarrow s', \mathbf{d}_2 \wedge$$
$$(\exists e, \mathbf{d}_{\text{foot}}, \mathbf{c}_{\text{surplus}}.\, \mathbf{d} \equiv \text{ap}(\mathbf{c}_{\text{surplus}}, \mathbf{d}_{\text{foot}}) \wedge e, s, \mathbf{d}_{\text{foot}} \models_{\text{G}} K \circ_{\text{G}} P)$$
$$\implies \exists \mathbf{d}_3.\, s, \mathbf{d}, \mathtt{C} \rightsquigarrow s', \mathbf{d}_3 \wedge \mathbf{d}_2 \equiv \text{ap}(\mathbf{c}, \mathbf{d}_3)$$

It is sufficient to show:

$$(\exists e, \mathbf{d}_{\text{foot}}, \mathbf{c}_{\text{surplus}}.\, \mathbf{d} \equiv \text{ap}(\mathbf{c}_{\text{surplus}}, \mathbf{d}_{\text{foot}}) \wedge e, s, \mathbf{d}_{\text{foot}} \models_{\text{G}} K \circ_{\text{G}} P)$$
$$\implies$$
$$(\exists e, \mathbf{d}_{\text{foot}}, \mathbf{c}_{\text{surplus}}.\, \mathbf{d} \equiv \text{ap}(\mathbf{c}_{\text{surplus}}, \mathbf{d}_{\text{foot}}) \wedge e, s, \mathbf{d}_{\text{foot}} \models_{\text{G}} P)$$

which is shown above, in the argument for grove-level safety-monotonicity.

- Next, consider the case in which P:T, Q:T and K:(T→T′) for T, T′ ∈ {ELE, TXT}.

  tree-level safety-monotonicity
  From the given premise:

  $$s, <\mathbf{d}>_{\mathrm{G}}, \mathtt{C} \not\rightsquigarrow \text{fault} \wedge \text{ap}(\mathbf{c}, \mathbf{d})\downarrow \wedge \mathbf{c}\text{:(D}\rightarrow\text{G)} \wedge$$
  $$(\exists e, \mathbf{d}_{\text{foot}}, \mathbf{c}_{\text{surplus}}.\, \mathbf{d} \equiv \text{ap}(\mathbf{c}_{\text{surplus}}, \mathbf{d}_{\text{foot}}) \wedge e, s, \mathbf{d}_{\text{foot}} \models_{\mathrm{T}} P)$$
  $$\Longrightarrow\ s, \text{ap}(\mathbf{c}, \mathbf{d}), \mathtt{C} \not\rightsquigarrow \text{fault}$$

  In order to show the conclusion:

  $$s, <\mathbf{d}>_{\mathrm{G}}, \mathtt{C} \not\rightsquigarrow \text{fault} \wedge \text{ap}(\mathbf{c}, \mathbf{d})\downarrow \wedge \mathbf{c}\text{:(D}\rightarrow\text{G)} \wedge$$
  $$(\exists e, \mathbf{d}_{\text{foot}}, \mathbf{c}_{\text{surplus}}.\, \mathbf{d} \equiv \text{ap}(\mathbf{c}_{\text{surplus}}, \mathbf{d}_{\text{foot}}) \wedge e, s, \mathbf{d}_{\text{foot}} \models_{\mathrm{T}} K \circ_{\mathrm{T}'} P)$$
  $$\Longrightarrow\ s, \text{ap}(\mathbf{c}, \mathbf{d}), \mathtt{C} \not\rightsquigarrow \text{fault}$$

  It is sufficient to show:

  $$(\exists e, \mathbf{d}_{\text{foot}}, \mathbf{c}_{\text{surplus}}.\, \mathbf{d} \equiv \text{ap}(\mathbf{c}_{\text{surplus}}, \mathbf{d}_{\text{foot}}) \wedge e, s, \mathbf{d}_{\text{foot}} \models_{\mathrm{T}} K \circ_{\mathrm{T}'} P)$$
  $$\Longrightarrow$$
  $$(\exists e, \mathbf{d}_{\text{foot}}, \mathbf{c}_{\text{surplus}}.\, \mathbf{d} \equiv \text{ap}(\mathbf{c}_{\text{surplus}}, \mathbf{d}_{\text{foot}}) \wedge e, s, \mathbf{d}_{\text{foot}} \models_{\mathrm{T}} P)$$

  The argument is then the same as that for grove-level safety-monotonicity, but with the type T substituted in place of the type G.

  tree-level frame property, node-grove-level safety-monotonicity and node-grove-level frame property
  As with tree-level safety-monotonicity, the arguments for the tree-level frame property and the node-grove-level frame property are the same as for the grove-level frame property, but with the type T substituted in place of the type G. For node-grove-level safety-monotonicity the argument is the same as grove-level safety-monotonicity but with the type T substituted in place of the type G.

- Finally, consider the case in which P:T, Q:T and K:(T→G) for T ∈ {ELE, TXT}. This case is more interesting, because it involves moving from tree-level safety-monotonicity and frame properties to grove-level safety-monotonicity and frame properties.

grove-level safety-monotonicity
Consider arbitrary $s, \mathbf{d}, \mathbf{c}$ which satisfy:

$$s, \mathbf{d}, \mathtt{C} \not\rightsquigarrow \text{fault} \wedge \text{ap}(\mathbf{c}, \mathbf{d})\downarrow \wedge$$
$$(\exists e, \mathbf{d}_{\text{foot}}, \mathbf{c}_{\text{surplus}}.\, \mathbf{d} \equiv \text{ap}(\mathbf{c}_{\text{surplus}}, \mathbf{d}_{\text{foot}}) \wedge e, s, \mathbf{d}_{\text{foot}} \models_{\text{G}} K \circ_{\text{T}} P)$$

The goal is to show that $s, \text{ap}(\mathbf{c}, \mathbf{d}), \mathtt{C} \not\rightsquigarrow \text{fault}$.

By the definition of context application, it can be seen that:

$$s, \mathbf{d}, \mathtt{C} \not\rightsquigarrow \text{fault} \wedge \text{ap}(\mathbf{c}, \mathbf{d})\downarrow \wedge$$
$$(\exists e, \mathbf{d}_{\text{foot}}, \mathbf{c}_{\text{surplus}}, \mathbf{d}'_{\text{foot}}, \mathbf{c}'_{\text{surplus}}.\, \mathbf{d} \equiv \text{ap}(\mathbf{c}_{\text{surplus}}, \mathbf{d}_{\text{foot}}) \wedge$$
$$\mathbf{d}_{\text{foot}} \equiv \text{ap}(\mathbf{c}'_{\text{surplus}}, \mathbf{d}'_{\text{foot}}) \wedge e, s, \mathbf{d}'_{\text{foot}} \models_{\text{T}} P)$$

Since $\mathbf{d} \equiv \text{ap}(\mathbf{c}_{\text{surplus}}, \mathbf{d}_{\text{foot}})$ and $\mathbf{d}_{\text{foot}} \equiv \text{ap}(\mathbf{c}'_{\text{surplus}}, \mathbf{d}'_{\text{foot}})$ it is possible to choose $\mathbf{c}''_{\text{surplus}}$ such that $\mathbf{d} \equiv \text{ap}(\mathbf{c}''_{\text{surplus}}, \mathbf{d}'_{\text{foot}})$. This leads to:

$$s, \mathbf{d}, \mathtt{C} \not\rightsquigarrow \text{fault} \wedge \text{ap}(\mathbf{c}, \mathbf{d})\downarrow \wedge$$
$$(\exists e, \mathbf{d}'_{\text{foot}}, \mathbf{c}''_{\text{surplus}}.\, \mathbf{d} \equiv \text{ap}(\mathbf{c}''_{\text{surplus}}, \mathbf{d}'_{\text{foot}}) \wedge e, s, \mathbf{d}'_{\text{foot}} \models_{\text{T}} P)$$

Finally, by node-grove-level safety monotonicity of $\mathtt{C}$ with respect to P, it can be seen that $s, \text{ap}(\mathbf{c}, \mathbf{d})\mathtt{C}, \not\rightsquigarrow \text{fault}$, which is the goal.

grove-level frame property
Consider arbitrary $s, \mathbf{d}, \mathbf{c}$ which satisfy:

$$s, \mathbf{d}, \mathtt{C} \not\rightsquigarrow \text{fault} \wedge \text{ap}(\mathbf{c}, \mathbf{d})\downarrow \wedge s, \text{ap}(\mathbf{c}, \mathbf{d}), \mathtt{C} \rightsquigarrow s', \mathbf{d}_2 \wedge$$
$$(\exists e, \mathbf{d}_{\text{foot}}, \mathbf{c}_{\text{surplus}}.\, \mathbf{d} \equiv \text{ap}(\mathbf{c}_{\text{surplus}}, \mathbf{d}_{\text{foot}}) \wedge e, s, \mathbf{d}_{\text{foot}} \models_{\text{G}} K \circ_{\text{T}} P)$$

The goal is to show that:

$$\exists \mathbf{d}_3.\, s, \mathbf{d}, \mathtt{C} \rightsquigarrow s', \mathbf{d}_3 \wedge \mathbf{d}_2 \equiv \text{ap}(\mathbf{c}, \mathbf{d}_3)$$

By the definition of context application, it can be seen that:

$$s, \mathbf{d}, \mathtt{C} \not\rightsquigarrow \text{fault} \wedge \text{ap}(\mathbf{c}, \mathbf{d})\downarrow \wedge s, \text{ap}(\mathbf{c}, \mathbf{d}), \mathtt{C} \rightsquigarrow s', \mathbf{d}_2 \wedge$$
$$(\exists e, \mathbf{d}_{\text{foot}}, \mathbf{c}_{\text{surplus}}, \mathbf{d}'_{\text{foot}}, \mathbf{c}'_{\text{surplus}}.\, \mathbf{d} \equiv \text{ap}(\mathbf{c}_{\text{surplus}}, \mathbf{d}_{\text{foot}}) \wedge$$
$$\mathbf{d}_{\text{foot}} \equiv \text{ap}(\mathbf{c}'_{\text{surplus}}, \mathbf{d}'_{\text{foot}}) \wedge e, s, \mathbf{d}'_{\text{foot}} \models_{\text{G}} P)$$

Since $\mathbf{d} \equiv \text{ap}(\mathbf{c}_{\text{surplus}}, \mathbf{d}_{\text{foot}})$ and $\mathbf{d}_{\text{foot}} \equiv \text{ap}(\mathbf{c}'_{\text{surplus}}, \mathbf{d}'_{\text{foot}})$ it is possible

to choose $\mathbf{c}''_{\text{surplus}}$ such that $\mathbf{d} \equiv \text{ap}(\mathbf{c}''_{\text{surplus}}, \mathbf{d}'_{\text{foot}})$. This leads to:

$$s, \mathbf{d}, \texttt{C} \not\rightsquigarrow \text{fault} \wedge \text{ap}(\mathbf{c}, \mathbf{d})\downarrow \wedge s, \text{ap}(\mathbf{c}, \mathbf{d}), \texttt{C} \rightsquigarrow s', \mathbf{d}_2 \wedge$$
$$(\exists e, \mathbf{d}'_{\text{foot}}, \mathbf{c}''_{\text{surplus}}.\ \mathbf{d} \equiv \text{ap}(\mathbf{c}''_{\text{surplus}}, \mathbf{d}'_{\text{foot}}) \wedge e, s, \mathbf{d}'_{\text{foot}} \models_{\text{G}} P)$$

Finally, by the node-grove-level frame property of C with respect to P it can be seen that:

$$\exists \mathbf{d}_3.\ s, \mathbf{d}, \texttt{C} \rightsquigarrow s', \mathbf{d}_3 \wedge \mathbf{d}_2 \equiv \text{ap}(\mathbf{c}, \mathbf{d}_3)$$

which is the goal.

□

4.8.6. Locality of the Featherweight DOM Commands

We show that each command of Featherweight DOM is local with respect to the preconditions of its axioms.

Lemma 28 (Locality of `getParentNode`)**.** *The command `getParentNode` is local to the preconditions of its axioms.*

Proof. This command is defined by two rules.

Rule 1:

$$\frac{\begin{array}{c}\mathbf{t} \equiv \mathbf{name}'_{s(\texttt{node})}[\mathbf{f}]'_{\mathbf{fid}} \\ \vee\, \mathbf{t} \equiv \text{“\#text”}_{s(\texttt{node})}\text{“}\mathbf{str}\text{”}\end{array} \qquad \mathbf{g} \equiv \text{ap}(\mathbf{cg}, \mathbf{name}_{\mathbf{id}}[\mathbf{f}_1 \otimes_{\text{F}} <\mathbf{t}>_{\text{F}} \otimes_{\text{F}} \mathbf{f}_2]_{\mathbf{fid}})}{s, \mathbf{g}, \texttt{id} := \texttt{getParentNode(node)} \rightsquigarrow [s|\texttt{id} \leftarrow \mathbf{id}], \mathbf{g}}$$

Rule 2:

$$\frac{\begin{array}{c}\mathbf{t} \equiv \mathbf{name}'_{s(\texttt{node})}[\mathbf{f}]'_{\mathbf{fid}} \\ \vee\, \mathbf{t} \equiv \text{“\#text”}_{s(\texttt{node})}\text{“}\mathbf{str}\text{”}\end{array} \qquad \mathbf{g} \equiv \text{ap}(\mathbf{cg}, <\mathbf{t}>_{\text{G}})}{s, \mathbf{g}, \texttt{id} := \texttt{getParentNode(node)} \rightsquigarrow [s|\texttt{id} \leftarrow \mathbf{null}], \mathbf{g}}$$

This command has two axioms, with the following two preconditions:

$$P \triangleq \begin{array}{l}\{<(\texttt{NAME}_{\texttt{node}}[\texttt{F:F}]_{\text{FID}} \vee \text{“\#text”}_{\texttt{node}}\texttt{VAL}) \wedge \texttt{T:T}>_{\text{G}}\} \\ \text{where T} \in \{\text{ELE}, \text{TXT}\}\end{array}$$

and:

$$P' \triangleq \begin{array}{l}\{\mathtt{NAME}'_{\mathtt{ID}'}[\mathtt{F}_1{:}\mathrm{F} \otimes_{\mathrm{F}} {<}(\mathtt{NAME}_{\mathtt{node}}[\mathtt{F}{:}\mathrm{F}]_{\mathrm{FID}} \vee \text{“\#text”}_{\mathtt{node}}\mathtt{VAL}) \wedge \mathtt{T}{:}\mathrm{T}{>}_{\mathrm{F}} \otimes_{\mathrm{F}} \mathtt{F}_2{:}\mathrm{F}]_{\mathtt{FID}'}\} \\ \text{where } \mathrm{T} \in \{\mathrm{ELE}, \mathrm{TXT}\}\end{array}$$

Consider each precondition in turn.

The first precondition P is of type G, and so it is necessary to show the grove-level safety-monotonicity and frame properties.

For safety-monotonicity, consider arbitrary $s, \mathbf{d}, \mathbf{c}$ such that:

$$\begin{array}{l} s, \mathbf{d}, \mathtt{id} := \mathtt{getParentNode}(\mathtt{node}) \not\rightsquigarrow \text{fault} \wedge \mathrm{ap}(\mathbf{c}, \mathbf{d})\downarrow \wedge \\ (\exists e, \mathbf{d}_{\text{foot}}, \mathbf{c}_{\text{surplus}}.\ \mathbf{d} \equiv \mathrm{ap}(\mathbf{c}_{\text{surplus}}, \mathbf{d}_{\text{foot}}) \wedge e, s, \mathbf{d}_{\text{foot}} \models_{\mathrm{G}} P \end{array}$$

The goal is to show that $s, \mathrm{ap}(\mathbf{c}, \mathbf{d}), \mathtt{id} := \mathtt{getParentNode}(\mathtt{node}) \not\rightsquigarrow$ fault.

Since $e, s, \mathbf{d}_{\text{foot}} \models_{\mathrm{G}} P$ (which stipulates that the node referred to by `node` is at the grove-level) there is no need to consider Rule 1 (which only describes the case in which the node referred to by `node` has a parent).

Since $e, s, \mathbf{d}_{\text{foot}} \models_{\mathrm{G}} P$, it can be seen that that $\mathbf{d}_{\text{foot}}$ must match the pattern denoted $\mathbf{t}$ in Rule 2.

By the definition of Rule 2, for all $\mathbf{g}, \mathbf{cg}$ such that $\mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}, \mathbf{d}_{\text{foot}})$, $s, \mathbf{g}, \mathtt{id} := \mathtt{getParentNode}(\mathtt{node}) \not\rightsquigarrow$ fault.

Since $\mathtt{d} \equiv \mathrm{ap}(\mathbf{c}_{\text{surplus}}, \mathbf{d}_{\text{foot}})$ it follows that for all $\mathbf{g}, \mathbf{cg}$ such that $\mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}, \mathbf{d})$ it can be seen that:

$$s, \mathbf{g}, \mathtt{id} := \mathtt{getParentNode}(\mathtt{node}) \not\rightsquigarrow \text{fault}$$

It follows directly that $s, \mathrm{ap}(\mathbf{c}, \mathbf{d}), \mathtt{id} := \mathtt{getParentNode}(\mathtt{node}) \not\rightsquigarrow$ fault, which is the goal.

For the frame property, consider arbitrary $s, \mathbf{d}, \mathbf{c}, s', \mathbf{d}_2$ such that:

$$\begin{array}{l} s, \mathbf{d}, \mathtt{id} := \mathtt{getParentNode}(\mathtt{node}) \not\rightsquigarrow \text{fault} \wedge \mathrm{ap}(\mathbf{c}, \mathbf{d})\downarrow \wedge \\ s, \mathrm{ap}(\mathbf{c}, \mathbf{d}), \mathtt{id} := \mathtt{getParentNode}(\mathtt{node}) \rightsquigarrow s', \mathbf{d}_2 \wedge \\ (\exists e, \mathbf{d}_{\text{foot}}, \mathbf{c}_{\text{surplus}}.\ \mathbf{d} \equiv \mathrm{ap}(\mathbf{c}_{\text{surplus}}, \mathbf{d}_{\text{foot}}) \wedge e, s, \mathbf{d}_{\text{foot}} \models_{\mathrm{G}} P) \end{array}$$

The goal is to show that:

$$\exists \mathbf{d}_3.\, s, \mathbf{d}, \texttt{id} := \texttt{getParentNode}(\texttt{node}) \rightsquigarrow s', \mathbf{d}_3 \wedge \mathbf{d}_2 \equiv \text{ap}(\mathbf{c}, \mathbf{d}_3)$$

As with safety-monotonicity, there is no need to consider Rule 1.

As with safety-monotonicity, $\mathbf{d}_{\text{foot}}$ matches the pattern denoted $\mathbf{t}$.

Similarly, by Rule 2, for all $\mathbf{g}, \mathbf{cg}$ such that $\mathbf{g} \equiv \text{ap}(\mathbf{cg}, \mathbf{d}_{\text{foot}})$ it follows that:

$$s, \mathbf{g}, \texttt{id} := \texttt{getParentNode}(\texttt{node}) \rightsquigarrow [s|\texttt{id} \leftarrow \textbf{null}], \mathbf{g}$$

Since $\texttt{d} \equiv \text{ap}(\mathbf{c}_{\text{surplus}}, \mathbf{d}_{\text{foot}})$ it follows by Rule 1 that:

$$s, \text{ap}(\mathbf{c}, \mathbf{d}), \texttt{id} := \texttt{getParentNode}(\texttt{node}) \rightsquigarrow [s|\texttt{id} \leftarrow \textbf{null}], \text{ap}(\mathbf{c}, \mathbf{d})$$

Which implies that $s' = [s|\texttt{id} \leftarrow \textbf{null}]$ and that $\mathbf{d}_2 \equiv \text{ap}(\mathbf{c}, \mathbf{d})$.

In addition, by Rule 1 it can be seen that:

$$s, \mathbf{d}, \texttt{id} := \texttt{getParentNode}(\texttt{node}) \rightsquigarrow s', \mathbf{d}$$

Since $\mathbf{d}_2 \equiv \text{ap}(\mathbf{c}, \mathbf{d})$ it is possible to choose $\mathbf{d}_3 \equiv \mathbf{d}$ and show that:

$$\exists \mathbf{d}_3.\, s, \mathbf{d}, \texttt{id} := \texttt{getParentNode}(\texttt{node}) \rightsquigarrow s', \mathbf{d}_3 \wedge \mathbf{d}_2 \equiv \text{ap}(\mathbf{c}, \mathbf{d}_3)$$

which is the goal.

The second precondition P' is of type ELE, and so it is necessary to show the tree-level safety-monotonicity and frame properties and the node-grove-level safety monotonicity and frame properties.

For safety-monotonicity, consider arbitrary $s, \mathbf{d}, \mathbf{c}$ such that:

$$s, <\mathbf{d}>_{\text{G}}, \texttt{id} := \texttt{getParentNode}(\texttt{node}) \not\rightsquigarrow \text{fault} \wedge \text{ap}(\mathbf{c}, \mathbf{d})\downarrow \wedge \mathbf{c}{:}(\text{D}{\rightarrow}\text{G}) \wedge (\exists e, \mathbf{d}_{\text{foot}}, \mathbf{c}_{\text{surplus}}.\, \mathbf{d} \equiv \text{ap}(\mathbf{c}_{\text{surplus}}, \mathbf{d}_{\text{foot}}) \wedge e, s, \mathbf{d}_{\text{foot}} \models_{\text{G}} P'$$

The goal is to show that $s, \text{ap}(\mathbf{c}, \mathbf{d}), \texttt{id} := \texttt{getParentNode}(\texttt{node}) \not\rightsquigarrow$ fault.

Since $e, s, \mathbf{d}_{\text{foot}} \models_{\text{ELE}} P'$ (which stipulates that the node referred to by `node` has a parent) there is no need to consider Rule 2 (which only describes the case in which the node referred to by `node` is at the grove level).

Since $e, s, \mathbf{d}_{\text{foot}} \models_{\text{G}} P'$, it can be seen that that $\mathbf{d}_{\text{foot}}$ must match the pattern denoted $\mathbf{name}_{\mathbf{id}}[\mathbf{f}_1 \otimes_{\text{F}} \mathbf{t} \otimes_{\text{F}} \mathbf{f}_2]_{\mathbf{fid}}$ in Rule 1.

By the definition of Rule 1, for all $\mathbf{g}, \mathbf{cg}$ such that $\mathbf{g} \equiv \text{ap}(\mathbf{cg}, \mathbf{d}_{\text{foot}})$, $s, \mathbf{g}, \texttt{id} := \texttt{getParentNode}(\texttt{node}) \not\rightsquigarrow$ fault.

Since $\texttt{d} \equiv \text{ap}(\mathbf{c}_{\text{surplus}}, \mathbf{d}_{\text{foot}})$ it follows that for all $\mathbf{g}, \mathbf{cg}$ such that $\mathbf{g} \equiv \text{ap}(\mathbf{cg}, \mathbf{d})$ it can be seen that:

$$s, \mathbf{g}, \texttt{id} := \texttt{getParentNode}(\texttt{node}) \not\rightsquigarrow \text{fault}$$

It follows directly that $s, \text{ap}(\mathbf{c}, \mathbf{d}), \texttt{id} := \texttt{getParentNode}(\texttt{node}) \not\rightsquigarrow$ fault, which is the goal.

For the frame property, consider arbitrary $s, \mathbf{d}, \mathbf{c}, s', \mathbf{d}_2$ such that:

$$\begin{aligned}
&s, <\mathbf{d}>_{\text{G}}, \texttt{id} := \texttt{getParentNode}(\texttt{node}) \not\rightsquigarrow \text{fault} \wedge \text{ap}(\mathbf{c}, \mathbf{d})\downarrow \wedge \\
&s, \text{ap}(\mathbf{c}, \mathbf{d}), \texttt{id} := \texttt{getParentNode}(\texttt{node}) \rightsquigarrow s', \mathbf{d}_2 \wedge \\
&(\exists e, \mathbf{d}_{\text{foot}}, \mathbf{c}_{\text{surplus}}.\, \mathbf{d} \equiv \text{ap}(\mathbf{c}_{\text{surplus}}, \mathbf{d}_{\text{foot}}) \wedge e, s, \mathbf{d}_{\text{foot}} \models_{\text{T}} P)
\end{aligned}$$

The goal is to show that:

$$\exists \mathbf{d}_3.\, s, <\mathbf{d}>_{\text{G}}, \texttt{id} := \texttt{getParentNode}(\texttt{node}) \rightsquigarrow s', <\mathbf{d}_3>_{\text{G}} \wedge \mathbf{d}_2 \equiv \text{ap}(\mathbf{c}, \mathbf{d}_3)$$

As with safety-monotonicity, there is no need to consider Rule 2.

As with safety-monotonicity, $\mathbf{d}_{\text{foot}}$ matches the pattern denoted $\mathbf{name}_{\mathbf{id}}[\mathbf{f}_1 \otimes_{\text{F}} \mathbf{t} \otimes_{\text{F}} \mathbf{f}_2]_{\mathbf{fid}}$.

Since $\texttt{d} \equiv \text{ap}(\mathbf{c}_{\text{surplus}}, \mathbf{d}_{\text{foot}})$ it follows by Rule 1 that:

$$s, \text{ap}(\mathbf{c}, \mathbf{d}), \texttt{id} := \texttt{getParentNode}(\texttt{node}) \rightsquigarrow [s|\texttt{id} \leftarrow \mathbf{id}], \text{ap}(\mathbf{c}, \mathbf{d})$$

Which implies that $s' = [s|\texttt{id} \leftarrow \mathbf{id}]$ and that $\mathbf{d}_2 \equiv \text{ap}(\mathbf{c}, \mathbf{d})$.

In addition, by Rule 1 it can be seen that:

$$s, <\mathbf{d}>_{\mathrm{G}}, \mathtt{id} := \mathtt{getParentNode}(\mathtt{node}) \rightsquigarrow s', <\mathbf{d}>_{\mathrm{G}}$$

Since $\mathbf{d}_2 \equiv \mathrm{ap}(\mathbf{c}, \mathbf{d})$ it is possible to choose $\mathbf{d}_3 \equiv \mathbf{d}$ and show that:

$$\exists \mathbf{d}_3.\, s, <\mathbf{d}>_{\mathrm{G}}, \mathtt{id} := \mathtt{getParentNode}(\mathtt{node}) \rightsquigarrow s', <\mathbf{d}_3>_{\mathrm{G}} \wedge \mathbf{d}_2 \equiv \mathrm{ap}(\mathbf{c}, \mathbf{d}_3)$$

which is the goal.

For node-grove-level safety-monotonicity, consider that the only difference between tree-level safety-monotonicity is that the clause in the premise of tree-level safety-monotonicity $s, <\mathbf{d}>_{\mathrm{G}}, \mathtt{C} \not\rightsquigarrow \mathrm{fault}$ is changed in node-grove-level safety-monotonicity to $s, \mathbf{d}, \mathtt{C} \not\rightsquigarrow \mathrm{fault}$. Since the proof of tree-level safety-monotonicity for `getParentNode` given above does not use this clause of the premise, the proof of node-grove-level safety-monotonicity is identical.

For the node-grove-level frame property, the proof is almost identical to that of the tree-level frame property. Consider arbitrary $s, \mathbf{d}, \mathbf{c}, s', \mathbf{d}_2$ such that:

$$\begin{array}{l} s, \mathbf{d}, \mathtt{id} := \mathtt{getParentNode}(\mathtt{node}) \not\rightsquigarrow \mathrm{fault} \wedge \mathrm{ap}(\mathbf{c}, \mathbf{d})\downarrow \wedge \\ s, \mathrm{ap}(\mathbf{c}, \mathbf{d}), \mathtt{id} := \mathtt{getParentNode}(\mathtt{node}) \rightsquigarrow s', \mathbf{d}_2 \wedge \\ (\exists e, \mathbf{d}_{\mathrm{foot}}, \mathbf{c}_{\mathrm{surplus}}.\, \mathbf{d} \equiv \mathrm{ap}(\mathbf{c}_{\mathrm{surplus}}, \mathbf{d}_{\mathrm{foot}}) \wedge e, s, \mathbf{d}_{\mathrm{foot}} \models_{\mathrm{T}} P) \end{array}$$

The goal is to show that:

$$\exists \mathbf{d}_3.\, s, \mathbf{d}, \mathtt{id} := \mathtt{getParentNode}(\mathtt{node}) \rightsquigarrow s', \mathbf{d}_3 \wedge \mathbf{d}_2 \equiv \mathrm{ap}(\mathbf{c}, \mathbf{d}_3)$$

As with tree-level safety-monotonicity and the tree-level frame property, there is no need to consider Rule 2.

As with tree-level safety-monotonicity and the tree-level frame property, $\mathbf{d}_{\mathrm{foot}}$ matches the pattern denoted $\mathbf{name}_{\mathbf{id}}[\mathbf{f}_1 \otimes_{\mathrm{F}} \mathbf{t} \otimes_{\mathrm{F}} \mathbf{f}_2]_{\mathbf{fid}}$.

Since $\mathtt{d} \equiv \mathrm{ap}(\mathbf{c}_{\mathrm{surplus}}, \mathbf{d}_{\mathrm{foot}})$ it follows by Rule 1 that:

$$s, \mathrm{ap}(\mathbf{c}, \mathbf{d}), \mathtt{id} := \mathtt{getParentNode}(\mathtt{node}) \rightsquigarrow [s|\mathtt{id} \leftarrow \mathbf{id}], \mathrm{ap}(\mathbf{c}, \mathbf{d})$$

Which implies that $s' = [s|\mathtt{id} \leftarrow \mathbf{id}]$ and that $\mathbf{d}_2 \equiv \mathrm{ap}(\mathbf{c}, \mathbf{d})$.

In addition, by Rule 1 it can be seen that:

$$s, \mathbf{d}, \texttt{id} := \texttt{getParentNode}(\texttt{node}) \rightsquigarrow s', \mathbf{d}$$

Since $\mathbf{d}_2 \equiv \text{ap}(\mathbf{c}, \mathbf{d})$ it is possible to choose $\mathbf{d}_3 \equiv \mathbf{d}$ and show that:

$$\exists \mathbf{d}_3.\, s, \mathbf{d}, \texttt{id} := \texttt{getParentNode}(\texttt{node}) \rightsquigarrow s', \mathbf{d}_3 \wedge \mathbf{d}_2 \equiv \text{ap}(\mathbf{c}, \mathbf{d}_3)$$

which is the goal.

□

Lemma 29 (Locality of `appendChild`)**.** *The command* ***`appendChild`*** *is local to the precondition of its axiom.*

Proof. This command is defined by two rules:

Rule 1:

$$\frac{\begin{array}{c}\mathbf{t} \equiv \mathbf{name}'_{s(\texttt{newChild})}[\mathbf{f}']_{\mathbf{fid}'} \\ \vee\, \mathbf{t} \equiv \text{“\#text”}_{s(\texttt{newChild})}\,\text{“}\mathbf{str}\text{”}\end{array} \quad \begin{array}{c}\mathbf{g} \equiv \text{ap}(\mathbf{cg}', <\mathbf{t}>_{\text{G}}) \\ \text{ap}(\mathbf{cg}', \varnothing_{\text{G}}) \equiv \text{ap}(\mathbf{cg}, \mathbf{name}_{s(\texttt{parent})}[\mathbf{f}]_{\mathbf{fid}}) \\ \mathbf{g}' \equiv \text{ap}(\mathbf{cg}, \mathbf{name}_{s(\texttt{parent})}[\mathbf{f} \otimes_{\text{F}} <\mathbf{t}>_{\text{F}}]_{\mathbf{fid}})\end{array}}{s, \mathbf{g}, \texttt{appendChild}(\texttt{parent}, \texttt{newChild}) \rightsquigarrow s, \mathbf{g}'}$$

Rule 2:

$$\frac{\begin{array}{c}\mathbf{t} \equiv \mathbf{name}'_{s(\texttt{newChild})}[\mathbf{f}']_{\mathbf{fid}'} \\ \vee\, \mathbf{t} \equiv \text{“\#text”}_{s(\texttt{newChild})}\,\text{“}\mathbf{str}\text{”}\end{array} \quad \begin{array}{c}\mathbf{g} \equiv \text{ap}(\mathbf{cg}', <\mathbf{t}>_{\text{F}}) \\ \text{ap}(\mathbf{cg}', \varnothing_{\text{F}}) \equiv \text{ap}(\mathbf{cg}, \mathbf{name}_{s(\texttt{parent})}[\mathbf{f}]_{\mathbf{fid}}) \\ \mathbf{g}' \equiv \text{ap}(\mathbf{cg}, \mathbf{name}_{s(\texttt{parent})}[\mathbf{f} \otimes_{\text{F}} <\mathbf{t}>_{\text{F}}]_{\mathbf{fid}})\end{array}}{s, \mathbf{g}, \texttt{appendChild}(\texttt{parent}, \texttt{newChild}) \rightsquigarrow s, \mathbf{g}'}$$

This command has one axiom, with the following precondition:

$$P \triangleq \begin{array}{l}(\varnothing_{\text{D}} \multimap (\texttt{C:ELE}{\rightarrow}\text{G} \circ_{\text{ELE}} (\texttt{NAME}_{\texttt{parent}}[\texttt{F:F}]_{\texttt{FID}}))) \circ_{\text{D}} \\ <(\texttt{NAME}'_{\texttt{newChild}}[\texttt{F}':\texttt{F}]_{\texttt{FID}'} \vee \text{“\#text”}_{\texttt{newChild}}\texttt{VAL}) \wedge \texttt{T:D}'>_{\text{D}}\end{array}$$

Since this axiom is of type G, it is sufficient to show that `appendChild` satisfies grove-level safety-monotonicity and the grove-level frame rule.

For safety-monotonicity, consider arbitrary $s, \mathbf{d}, \mathbf{c}$ such that:

$$\begin{array}{l}s, \mathbf{d}, \texttt{appendChild}(\texttt{parent}, \texttt{newChild}) \not\rightsquigarrow \text{fault} \wedge \text{ap}(\mathbf{c}, \mathbf{d})\downarrow \wedge \\ (\exists e, \mathbf{d}_{\text{foot}}, \mathbf{c}_{\text{surplus}}.\, \mathbf{d} \equiv \text{ap}(\mathbf{c}_{\text{surplus}}, \mathbf{d}_{\text{foot}}) \wedge e, s, \mathbf{d}_{\text{foot}} \models_{\text{G}} P\end{array}$$

The goal is to show that $s, \mathrm{ap}(\mathbf{c}, \mathbf{d}), \texttt{appendChild}(\texttt{parent}, \texttt{newChild}) \not\rightsquigarrow$ fault.

Regardless of which rule determines the result of executing $s, \mathbf{d}, \texttt{appendChild}(\texttt{parent}, \texttt{newChild})$, it follows that $\mathbf{d} \equiv \mathrm{ap}(\mathbf{cg}', <\mathbf{t}>_{\mathrm{D}})$.

It is therefore possible to choose $\mathbf{cg}''$ such that $\mathrm{ap}(\mathbf{c}, \mathbf{d}) \equiv \mathrm{ap}(\mathbf{cg}'', <\mathbf{t}>_{\mathrm{D}})$.

It follows directly from the operational semantics that $s, \mathrm{ap}(\mathbf{c}, \mathbf{d}), \texttt{appendChild}(\texttt{parent}, \texttt{newChild}) \not\rightsquigarrow$ fault, which is the goal.

For the frame property, consider arbitrary $s, \mathbf{d}, \mathbf{c}, s', \mathbf{d}_2$ such that:

$$\begin{aligned} &s, \mathbf{d}, \texttt{appendChild}(\texttt{parent}, \texttt{newChild}) \not\rightsquigarrow \text{fault} \wedge \mathrm{ap}(\mathbf{c}, \mathbf{d})\downarrow \wedge \\ &s, \mathrm{ap}(\mathbf{c}, \mathbf{d}), \texttt{appendChild}(\texttt{parent}, \texttt{newChild}) \rightsquigarrow s', \mathbf{d}_2 \wedge \\ &(\exists e, \mathbf{d}_{\text{foot}}, \mathbf{c}_{\text{surplus}}.\, \mathbf{d} \equiv \mathrm{ap}(\mathbf{c}_{\text{surplus}}, \mathbf{d}_{\text{foot}}) \wedge e, s, \mathbf{d}_{\text{foot}} \models_{\mathrm{G}} P) \end{aligned}$$

The goal is to show that:

$$\exists \mathbf{d}_3.\, s, \mathbf{d}, \texttt{appendChild}(\texttt{parent}, \texttt{newChild}) \rightsquigarrow s', \mathbf{d}_3 \wedge \mathbf{d}_2 \equiv \mathrm{ap}(\mathbf{c}, \mathbf{d}_3)$$

Regardless of which rule determines the result of executing $s, \mathbf{d}, \texttt{appendChild}(\texttt{parent}, \texttt{newChild})$, it follows that $\mathbf{d} \equiv \mathrm{ap}(\mathbf{cg}', <\mathbf{t}>_{\mathrm{D}})$.

Since $\mathrm{ap}(\mathbf{cg}', \varnothing_{\mathrm{D}}) \equiv \mathrm{ap}(\mathtt{cg}, \mathbf{name}_{s(\texttt{parent})}[\mathbf{f}]_{\mathbf{fid}})$, it is possible to choose $\mathbf{cg}_{\text{big}}, \mathbf{cg}'_{\text{big}}$ such that $\mathrm{ap}(\mathbf{c}, \mathbf{d}) \equiv \mathrm{ap}(\mathbf{cg}'_{\text{big}}, <\mathbf{t}>_{\mathrm{D}})$ and $\mathrm{ap}(\mathbf{cg}'_{\text{big}}, \varnothing_{\mathrm{D}}) \equiv \mathrm{ap}(\mathtt{cg}_{\text{big}}, \mathbf{name}_{s(\texttt{parent})}[\mathbf{f}]_{\mathbf{fid}})$

Since $s, \mathrm{ap}(\mathbf{c}, \mathbf{d}), \texttt{appendChild}(\texttt{parent}, \texttt{newChild}) \rightsquigarrow s', \mathbf{d}_2$, it follows directly from the operational semantics that

$$\exists \mathbf{d}_3.\, s, \mathbf{d}, \texttt{appendChild}(\texttt{parent}, \texttt{newChild}) \rightsquigarrow s', \mathbf{d}_3 \wedge \mathbf{d}_2 \equiv \mathrm{ap}(\mathbf{c}, \mathbf{d}_3)$$

which is the goal.

□

Lemma 30 (Locality of the Remaining Commands)**.** *The remaining commands are local to the preconditions of their axioms.*

Proof Sketch. The argument for `removeChild` is identical to that of `appendChild`. Assignment is trivially local to any formula, since it is independent of the grove.

Now consider the commands `getNodeName`, `getChildNodes`, `item`, `substringData`, `appendData` and `deleteData`. Each of these commands act only on a single node, and will only fault if that node is missing or otherwise defective – for example if `substringData` is called on a node which is not a text node. The safety monotonicity properties always hold for these commands, since the successful application of a context to the given tree will still leave the requisite node in the resulting tree. Notice also that these commands all describe the transformation of a tree $\mathrm{ap}(\mathbf{cg}, \mathrm{T})$ to a new tree $\mathrm{ap}(\mathbf{cg}, \mathbf{t}')$. The frame properties hold since running the command on a larger well defined tree $\mathrm{ap}(\mathbf{cg}', \mathrm{ap}(\mathbf{cg}, \mathbf{t}))$ will result in the tree $\mathrm{ap}(\mathbf{cg}', \mathrm{ap}(\mathbf{cg}, \mathbf{t}'))$ – the execution can clearly be tracked back to the smaller state.

The commands `createElement` and `createTextNode` both operate on an empty grove and will not fault on any larger grove, hence the grove-level safety-monotonicity property holds. These commands are similar to the ones discussed in the previous paragraph in that they describe the transformation of a grove $\mathrm{ap}(\mathbf{cg}, \mathrm{G})$ to a new grove $\mathrm{ap}(\mathbf{cg}, \mathbf{g}')$. The grove-level frame property holds since running the command on a larger well defined grove $\mathrm{ap}(\mathbf{cg}', \mathrm{ap}(\mathbf{cg}, \mathbf{g}))$ will result in the grove $\mathrm{ap}(\mathbf{cg}', \mathrm{ap}(\mathbf{cg}, \mathbf{g}'))$, which can clearly be tracked back to the smaller state. □

4.8.7. Soundness and Locality Propagation

Since the soundness of our frame rule will rely on the locality of the command being reasoned about, we are obliged to show not only partial correctness and fault avoidance for each derivable Hoare triple, but also locality. We show all three properties by simultaneous induction on the derivation of Hoare triples. The base cases are the axioms, which are sound by standard arguments and are local by Lemmas 28, 29 and 30. The inductive steps are the inference rules, which we deal with in the following lemmas.

Lemma 31 (Soundness and Locality for Sequence)**.** *If the premises of the sequential composition rule are sound and local with respect to their preconditions, then the conclusion of the sequential composition rule is sound and local with respect to its precondition.*

Proof. The argument for the soundness of the sequential composition rule is standard. The argument for locality follows.

The inference rule for sequential composition is:

$$\frac{\{P\}\mathtt{C}_1\{Q\} \quad \{Q\}\mathtt{C}_2\{R\}}{\{P\}\mathtt{C}_1 \,;\, \mathtt{C}_2\{R\}}$$

The semantics of sequential composition are defined by:

$$\frac{s, \mathbf{g}, \mathtt{C}_1 \rightsquigarrow s', \mathbf{g}' \qquad s', \mathbf{g}', \mathtt{C}_2 \rightsquigarrow s'', \mathbf{g}''}{s, \mathbf{g}, (\mathtt{C}_1 \,;\, \mathtt{C}_2) \rightsquigarrow s'', \mathbf{g}''}$$

The goal is to show that if $\mathtt{C}_1$ is local to P and $\mathtt{C}_2$ is local to Q then $\mathtt{C}_1 \,;\, \mathtt{C}_2$ is local to P.

Note that by the definition of Hoare triples, P, Q and R must all be of the same type. Now consider the two possible cases separately: The case in which P:G , and the case in which P:T where $\mathrm{T} \in \{\mathrm{ELE}, \mathrm{TXT}\}$.

The case in which P:G The goal in this case is to show that $\mathtt{C}_1 \,;\, \mathtt{C}_2$ satisfies the grove-level safety-monotonicity and frame properties given that both $\mathtt{C}_1$ and $\mathtt{C}_2$ do.

To show safety-monotonicity choose arbitrary $s, \mathbf{d}, \mathbf{c}$ such that:

$$s, \mathbf{d}, \mathtt{C}_1 \,;\, \mathtt{C}_2 \not\rightsquigarrow \text{fault} \wedge \mathrm{ap}(\mathbf{c}, \mathbf{d})\downarrow \wedge$$
$$(\exists e, \mathbf{d}_{\text{foot}}, \mathbf{c}_{\text{surplus}}.\, \mathbf{d} \equiv \mathrm{ap}(\mathbf{c}_{\text{surplus}}, \mathbf{d}_{\text{foot}}) \wedge e, s, \mathbf{d}_{\text{foot}} \models_{\mathrm{G}} P)$$

The goal is to show that $s, \mathrm{ap}(\mathbf{c}, \mathbf{d}), \mathtt{C}_1 \,;\, \mathtt{C}_2 \not\rightsquigarrow \text{fault}$.

By the safety-monotonicity property of $\mathtt{C}_1$ with respect to P it follows that $s, \mathrm{ap}(\mathbf{c}, \mathbf{d}), \mathtt{C}_1 \not\rightsquigarrow \text{fault}$.

By the premise $\{P\}\mathtt{C}_1\{Q\}$ it follows that $s, \mathrm{ap}(\mathbf{c}, \mathbf{d}), \mathtt{C}_1 \rightsquigarrow s', \mathbf{g}'$ such that $\exists e.\, e, s'\mathbf{g}' \models_{\mathrm{G}} Q$.

By the safety-monotonicity property of $\mathtt{C}_2$ with respect to Q, it follows that $s', \mathrm{ap}(\mathbf{c}, \mathbf{g}), \mathtt{C}_2 \not\rightsquigarrow \text{fault}$

It then follows from the operational semantics of $\mathtt{C}_1 \,;\, \mathtt{C}_2$ that $s, \mathrm{ap}(\mathbf{c}, \mathbf{d}), \mathtt{C}_1 \,;\, \mathtt{C}_2 \not\rightsquigarrow \text{fault}$ which is the goal.

To show the frame property choose arbitrary $s, \mathbf{d}, \mathbf{c}, \mathbf{d}_2$ such that:

$$s, \mathbf{d}, \mathtt{C}_1 \,;\, \mathtt{C}_2 \not\rightsquigarrow \text{fault} \wedge \mathrm{ap}(\mathbf{c}, \mathbf{d})\downarrow \wedge s, \mathrm{ap}(\mathbf{c}, \mathbf{d}), \mathtt{C}_1 \,;\, \mathtt{C}_2 \rightsquigarrow s', \mathbf{d}_2 \wedge$$
$$(\exists e, \mathbf{d}_{\text{foot}}, \mathbf{c}_{\text{surplus}}.\, \mathbf{d} \equiv \mathrm{ap}(\mathbf{c}_{\text{surplus}}, \mathbf{d}_{\text{foot}}) \wedge e, s, \mathbf{d}_{\text{foot}} \models_{\mathrm{G}} P)$$

The goal is to show that $\exists \mathbf{d}_3.\, s, \mathbf{d}, \mathtt{C}_1 \,;\, \mathtt{C}_2 \rightsquigarrow s', \mathbf{d}_3 \wedge \mathbf{d}_2 \equiv \mathrm{ap}(\mathbf{c}, \mathbf{d}_3)$

By the operational semantics of $\mathtt{C}_1 \,;\, \mathtt{C}2$ it is possible to choose intermediate state $s'', \mathbf{d}'_2$ such that $s, \mathrm{ap}(\mathbf{c}, \mathbf{d}), \mathtt{C}_1 \rightsquigarrow s'', \mathbf{d}'_2$ and $s'', \mathbf{d}'_2, \mathtt{C}_2 \rightsquigarrow s', \mathbf{d}_2$

By the grove-level frame property of $\mathtt{C}_1$ with respect to P it is possible to choose an intermediate small state $\mathbf{d}'_3$ such that
$s, \mathbf{d}, \mathtt{C}_1 \rightsquigarrow s'', \mathbf{d}'_3 \wedge \mathbf{d}'_2 \equiv \mathrm{ap}(\mathbf{c}, \mathbf{d}'_3)$

From $s, \mathbf{d}, \mathtt{C}_1 \,;\, \mathtt{C}_2 \not\rightsquigarrow$ fault and the operational semantics of $\mathtt{C}_1 \,;\, \mathtt{C}2$ it follows that $s'', \mathbf{d}'_3, \mathtt{C}_2 \not\rightsquigarrow$ fault.

By the premise $\{P\}\mathtt{C}_1\{Q\}$ it follows that $\exists e.\, e, s'', \mathbf{d}'_3 \models_{\mathrm{G}} Q$.

By the grove-level frame property of $\mathtt{C}_2$ with respect to Q and since $s'', \mathbf{d}'_2, \mathtt{C}_2 \rightsquigarrow s', \mathbf{d}_2$ and $\mathbf{d}'_2 \equiv \mathrm{ap}(\mathbf{c}, \mathbf{d}'_3)$ it follows that $\exists \mathbf{d}_3.\, s'', \mathbf{d}'_3, \mathtt{C}_2 \rightsquigarrow s', \mathbf{d}_3 \wedge \mathbf{d}_2 \equiv \mathrm{ap}(\mathbf{c}, \mathbf{d}_3)$

By the operational semantics of $\mathtt{C}_1 \,;\, \mathtt{C}_2$ it follows that $\exists \mathbf{d}_3.\, s, \mathbf{d}, \mathtt{C}_1 \,;\, \mathtt{C}_2 \rightsquigarrow s', \mathbf{d}_3 \wedge \mathbf{d}_2 \equiv \mathrm{ap}(\mathbf{c}, \mathbf{d}_3)$, which is the goal.

The case in which P:**T** The goal in this case is to show that $\mathtt{C}_1 \,;\, \mathtt{C}_2$ satisfies the tree-level safety-monotonicity and frame properties and the node-grove-level safety-monotonicity and frame properties given that both $\mathtt{C}_1$ and $\mathtt{C}_2$ do.

The arguments for tree-level and node-grove-level safety-monotonicity are identical to the argument above for grove-level safety-monotonicity.

The arguments for the tree-level and node-grove-level frame properties are identical to the argument above for the grove-level frame property.

□

Lemma 32 (Soundness and Locality for the Frame Rule)**.** *If a command* $\mathtt{C}$ *is local to the precondition* P *of a Hoare triple* $\{P\}\mathtt{C}\{Q\}$*, then the Frame Rule can be safely used to extend that Hoare triple and* $\mathtt{C}$ *will be local with respect to the precondition of the extended Hoare triple.*

Proof. Locality for the Frame Rule follows directly from Lemma 27, the monotonicity of locality. Soundness for the Frame Rule may be split into three cases: The case in which $\{P\}\mathtt{C}\{Q\}$ is grove triple with P:G, Q:G ; The

case in which $\{P\}\mathtt{C}\{Q\}$ is a tree-level triple with P:T, Q:T and K:(T→G) where T ∈ {ELE, TXT} ; and the case in which $\{P\}\mathtt{C}\{Q\}$ is a tree-level triple with P:T, Q:T and K:(T→T′) where T, T′ ∈ {ELE, TXT}.

- First consider the case that the premise $\{P\}\mathtt{C}\{Q\}$ is grove triple with P:G, Q:G. The goal is to show partial correctness and fault-avoidance.

  Let $e, s, \mathbf{g}$ be an arbitrary environment, store and grove satisfying $e, s, \mathbf{g} \models_{\mathrm{G}} K \circ_{\mathrm{G}} P$.

  By the definition of context application, the grove may be split: $\mathbf{g} \equiv \mathrm{ap}(\mathbf{c}, \mathbf{g}_0)$ where $e, s, \mathbf{g}_0 \models_{\mathrm{G}} P$ and $e, s, \mathbf{c} \models_{(\mathrm{G}\to\mathrm{G})} K$.

  By the fault avoidance of the premise $\{P\}\mathtt{C}\{Q\}$ it can be seen that $s, \mathbf{g}_0, \mathtt{C} \not\rightsquigarrow$ fault.

  By the grove-level safety-monotonicity property with respect to $\{P\}\mathtt{C}\{Q\}$, it can be seen that $s, \mathrm{ap}(\mathbf{c}, \mathbf{g}_0), \mathtt{C} \not\rightsquigarrow$ fault. Since $\mathbf{g} \equiv \mathrm{ap}(\mathbf{c}, \mathbf{g}_0)$ fault-avoidance follows directly.

  Since fault avoidance holds, it is possible to introduce $\mathbf{g}', s'$ such that $s, \mathbf{g}, \mathtt{C} \rightsquigarrow s', \mathbf{g}'$.

  By the grove-level frame property there exists $\mathbf{g}'_0$ such that $s, \mathbf{g}_0, \mathtt{C} \rightsquigarrow s', \mathbf{g}'_0$ and $\mathbf{g}' \equiv \mathrm{ap}(\mathbf{c}, \mathbf{g}'_0)$.

  By partial correctness of the premise it can be shown that $e, s', \mathbf{g}'_0 \models_{\mathrm{G}} Q$.

  Since $\mathbf{g}' \equiv \mathrm{ap}(\mathbf{c}, \mathbf{g}'_0)$ and $\mathbf{c} \models_{(\mathrm{G}\to\mathrm{G})} K$ it is possible to show $e, s', \mathbf{g}' \models_{\mathrm{G}} K \circ_{\mathrm{G}} Q$ which provides partial correctness for the conclusion.

  Both partial correctness and fault avoidance have now been shown in this case.

- Next consider the case that the premise $\{P\}\mathtt{C}\{Q\}$ is a tree-level triple with P:T, Q:T and K:(T→G) where T ∈ {ELE, TXT}. The goal is to show partial correctness and fault avoidance.

  This case follows the pattern of the grove-level case.

  Choose arbitrary $e, s, \mathbf{g}$ satisfying $e, s, \mathbf{g} \models_{\mathrm{G}} K \circ_{\mathrm{T}} P$.

  Split the grove as before, but this time into a context and a *tree*: $\mathbf{g} \equiv \mathrm{ap}(\mathbf{c}, \mathbf{t}_0)$ where $e, s, \mathbf{t}_0 \models_{\mathrm{T}} P$ and $e, s, \mathbf{c} \models_{(\mathrm{T}\to\mathrm{G})} K$.

By the fault avoidance of the premise $\{P\}\mathtt{C}\{Q\}$ it is shown that $s, <\mathbf{t}_0>_{\mathrm{G}}, \mathtt{C} \not\rightarrow$ fault.

By the tree-level safety-monotonicity property, it is shown that $s, \mathrm{ap}(\mathbf{c}, \mathbf{t}_0), \mathtt{C} \not\rightarrow$ fault, and since $\mathbf{g} \equiv \mathrm{ap}(\mathbf{c}, \mathbf{t}_0)$ this provides fault-avoidance for the conclusion.

Since fault avoidance holds, it is possible to introduce $s', \mathbf{g}'$ such that $s, \mathbf{g}, \mathtt{C} \rightsquigarrow s', \mathbf{g}'$.

By the tree-level frame property there exists $\mathbf{t}_0'$ such that $s, <\mathbf{t}_0>_{\mathrm{G}}, \mathtt{C} \rightsquigarrow s', <\mathbf{t}_0'>_{\mathrm{G}}$ and $\mathbf{g}' \equiv \mathrm{ap}(\mathbf{c}, \mathbf{t}_0')$.

By partial correctness of the premise it follows that $e, s', <\mathbf{g}_0'>_{\mathrm{G}} \models_{\mathrm{G}} <Q>_{\mathrm{G}}$.

By the satisfaction of $<Q>_{\mathrm{G}}$ it follows that $e, s', \mathbf{g}_0 \models_{\mathrm{T}} Q$ and hence $e, s', \mathbf{g}' \models_{\mathrm{G}} K \circ_{\mathrm{T}} Q$ which provides partial correctness for the conclusion.

Both partial correctness and fault avoidance have now been shown in this case.

- Finally consider the case where the premise $\{P\}\mathtt{C}\{Q\}$ is a tree-level triple with $P{:}\mathrm{T}, Q{:}\mathrm{T}$ and $K{:}(\mathrm{T}{\rightarrow}\mathrm{T}')$ where $\mathrm{T}, \mathrm{T}' \in \{\mathrm{ELE}, \mathrm{TXT}\}$. The goal is to show partial correctness and fault avoidance.

  This case follows the pattern of the previous cases.

  Choose arbitrary $e, s, \mathbf{g}$ satisfying $e, s, \mathbf{g} \models_{\mathrm{G}} <K \circ_{\mathrm{T}} P>_{\mathrm{G}}$

  Split the grove as before, into a context and a tree: $\mathbf{g} \equiv \mathrm{ap}(\mathbf{c}, \mathbf{t}_0)$ where
  $e, s, \mathbf{t}_0 \models_{\mathrm{T}} P$ and $e, s, \mathbf{c} \models_{(\mathrm{T}\rightarrow\mathrm{G})} <K>_{\mathrm{G}}$.

  By the fault avoidance of the premise $\{P\}\mathtt{C}\{Q\}$ it follows that $s, <\mathbf{t}_0>_{\mathrm{G}}, \mathtt{C} \not\rightarrow$ fault.

  By the tree-level safety-monotonicity property, it follows that $s, \mathrm{ap}(\mathbf{c}, \mathbf{t}_0), \mathtt{C} \not\rightarrow$ fault, and since $\mathbf{g} \equiv \mathrm{ap}(\mathbf{c}, \mathbf{t}_0)$ this provides fault-avoidance for the conclusion.

  Since fault avoidance holds, it is possible to introduce $s', \mathbf{g}'$ such that $s, \mathbf{g}, \mathtt{C} \rightsquigarrow s', \mathbf{g}'$.

By the tree-level frame property there exists $\mathbf{t}_0'$ such that $s, <\mathbf{t}_0>_{\mathrm{G}}, \mathtt{C} \rightsquigarrow s', <\mathbf{t}_0'>_{\mathrm{G}}$ and $\mathbf{g}' \equiv \mathrm{ap}(\mathbf{c}, \mathbf{t}_0')$.

By partial correctness of the premise it follows that $e, s', <\mathbf{g}_0'>_{\mathrm{G}} \models_{\mathrm{G}} <Q>_{\mathrm{G}}$, and hence $e, s', \mathbf{g}' \models_{\mathrm{G}} <K \circ_{\mathrm{T}} Q>_{\mathrm{G}}$ which provides partial correctness for the conclusion.

Both partial correctness and fault avoidance have now been shown in this case.

□

Lemma 33 (Soundness and Locality for the Remaining Inference Rules)**.** *If the premises of the remaining rules are sound and local, then the conclusions of those rules are sound and local.*

Proof Sketch. The arguments for soundness of the remaining rules are standard.

The inference rules for conditional blocks, while loops and local blocks directly propagate the preconditions of their subcommands, changing only non-structural conditions such as $\mathtt{Bool} \doteq \mathrm{true}$. It follows that if the subcommands are local to their preconditions then the conditional blocks, loops and local blocks are local to their preconditions.

Similarly, locality propagation for the inference rules for consequence, disjunction and auxiliary variable elimination follow directly from the definitions of implication, disjuction and auxiliary variables. □

Theorem 34 (Soundness of Featherweight DOM Reasoning)**.** *Every derivable Hoare triple in Featherweight DOM is sound.*

Proof. A Featherweight DOM Hoare triple is sound if it satisfies the two properties given in Definition 23: partial correctness and fault avoidance. In addition, it is desirable to show that every command in Featherweight DOM is local with respect to the precondition of every derivable Hoare triple as defined in Definition 26.

All three of these properties are shown by simultaneous induction on the derivation of Hoare triples. The base cases are the axioms, which are sound by standard arguments and are local with respect to their preconditions by Lemmas 28, 29 and 30. The inductive steps are the inference rules, the

conclusions of which are sound and local with respect to their preconditions by Lemmas 31, 32, and 33. □

5. Featherweight DOM Examples

This chapter demonstrates Featherweight DOM reasoning with a number of examples. First, it addresses the issue of completing the Node Interface functionality by implementing additional commands in Featherweight DOM. Next it uses the W3C DOM Compliance tests to drive a discussion about the relationship between specification and a correct implementation. Finally it demonstrates the sort of reasoning one might find in a more industrial context, proving schema-invariance properties of a more realistic program-fragment.

5.1. Additional Commands

Recall that Featherweight DOM reflects the essence of the Node interface defined in [23]. The remaining Node interface commands are implemented in Appendix A.1. In this section, we select some representative commands from that appendix, describe their implementation, and prove their behaviour using context logic.

5.1.1. getPreviousSibling

The object attribute "previousSibling" is defined on the Node interface of DOM Core Level 1. We implement the getter method `getPreviousSibling`, using an auxiliary command '`getIndex`' which is not in DOM Core Level 1 (and which is also useful in its own right). `getIndex` returns the index of a given node in a given list. The implementations of `getIndex` and `getPreviousSibling` are given in Figure 5.1.

The `getIndex` command uses a simple while loop to do a linear search of the nodes in the parameter `nodeList`, counting the elements in turn until the target `node` is found. It then returns the position of that node. The `getPreviousSibling` command uses `getParentNode` and `getChildNodes` to obtain the list of siblings of the parameter `node`. It then uses `getIndex`

```
n := getIndex(nodeList, node) ≜
    local current, returnval :
        returnval := 0 ;
        current := item(nodeList, returnval) ;
        while (current ≠ node ∧ current ≠ null) do
            returnval := returnval + 1 ;
            current := item(nodelist, returnval)
        od ;
        n := returnval
    endloc
```

```
sibling := getPreviousSibling(node) ≜
    local parent, children, n :
        parent := getParentNode(node) ;
        if parent = null then
            sibling := null
        else
            children := getChildNodes(parent) ;
            n := getIndex(children, node) ;
            sibling := item(nodelist, n − 1)
        fi
    endloc
```

Figure 5.1.: getIndex and getPreviousSibling

$$
\left\{\begin{array}{l}
\mathtt{NAME_{ID}}[\\
\quad \mathtt{F_1}{:}\mathrm{F} \otimes_{\mathrm{F}} \\
\quad <(\mathtt{NAME'_{node}}[\mathtt{F'}{:}\mathrm{F}]_{\mathrm{FID'}} \vee \text{“\#text”}_{\mathtt{node}}\mathtt{VAL'}) \wedge \mathtt{T}{:}\mathrm{D}>_{\mathrm{F}} \\
\quad \otimes_{\mathrm{F}} \mathtt{F_2}{:}\mathrm{F} \\
]_{\mathtt{Y}} \\
\wedge\, \mathtt{Y} \doteq \mathtt{nodelist}
\end{array}\right\}
$$

$\mathtt{n} := \mathtt{getIndex}(\mathtt{nodeList}, \mathtt{node}) \triangleq$

$\mathtt{local\ current}, \mathtt{returnval}:$

$\mathtt{returnval} := 0\ ;\ \mathtt{current} := \mathtt{item}(\mathtt{nodeList}, \mathtt{returnval})\ ;$

$$
\left\{\begin{array}{l}
\exists \mathtt{NAME''}, \mathtt{F''}, \mathtt{FID''}, \mathtt{VAL''}, \mathtt{F_3}, \mathtt{F_4}.\ \mathtt{returnval} \doteq \mathrm{len}(\mathtt{F_3}) \wedge \mathtt{Y} \doteq \mathtt{nodeList} \\
\wedge\, \mathtt{NAME_{ID}}[\\
\quad \left((\mathtt{F_1}{:}\mathrm{F} \otimes_{\mathrm{F}} <(\mathtt{NAME'_{node}}[\mathtt{F'}{:}\mathrm{F}]_{\mathrm{FID'}} \vee \text{“\#text”}_{\mathtt{node}}\mathtt{VAL'}) \wedge \mathtt{T}{:}\mathrm{D}>_{\mathrm{F}}) \wedge \right. \\
\quad \left.(\mathtt{F_3}{:}\mathrm{F} \otimes_{\mathrm{F}} <(\mathtt{NAME''_{current}}[\mathtt{F''}{:}\mathrm{F}]_{\mathrm{FID''}} \vee \text{“\#text”}_{\mathtt{node}}\mathtt{VAL''})>_{\mathrm{F}} \otimes_{\mathrm{F}} \mathtt{F_4}{:}\mathrm{F})\right) \otimes_{\mathrm{F}} \mathtt{F_2}{:}\mathrm{F} \\
]_{\mathtt{Y}}
\end{array}\right\}
$$

$\mathtt{while}\ (\mathtt{current} \neq \mathtt{node} \wedge \mathtt{current} \neq \mathbf{null})\ \mathtt{do}$

$$
\left\{\begin{array}{l}
\exists \mathtt{NAME''}, \mathtt{F''}, \mathtt{FID''}, \mathtt{VAL''}, \mathtt{F_3}, \mathtt{F_4}, \mathtt{NAME'''}, \mathtt{ID'''}, \mathtt{F'''}, \mathtt{FID'''}, \mathtt{VAL'''}. \\
\mathtt{returnval} \doteq \mathrm{len}(\mathtt{F_3}) \wedge \mathtt{Y} \doteq \mathtt{nodeList} \\
\wedge\, \mathtt{NAME_{ID}}[\\
\quad \left((\mathtt{F_1}{:}\mathrm{F} \otimes_{\mathrm{F}} <(\mathtt{NAME'_{node}}[\mathtt{F'}{:}\mathrm{F}]_{\mathrm{FID'}} \vee \text{“\#text”}_{\mathtt{node}}\mathtt{VAL'}) \wedge \mathtt{T}{:}\mathrm{D}>_{\mathrm{F}}) \wedge \right. \\
\quad \left.\left(\begin{array}{l}\mathtt{F_3}{:}\mathrm{F} \otimes_{\mathrm{F}} \\ <(\mathtt{NAME''_{current}}[\mathtt{F''}{:}\mathrm{F}]_{\mathrm{FID''}} \vee \text{“\#text”}_{\mathtt{node}}\mathtt{VAL''})>_{\mathrm{F}} \otimes_{\mathrm{F}} \\ <(\mathtt{NAME'''_{ID'''}}[\mathtt{F'''}{:}\mathrm{F}]_{\mathrm{FID'''}} \vee \text{“\#text”}_{\mathtt{node}}\mathtt{VAL'''})>_{\mathrm{F}} \otimes_{\mathrm{F}} \\ \mathtt{F_4}{:}\mathrm{F}\end{array}\right)\right) \otimes_{\mathrm{F}} \mathtt{F_2}{:}\mathrm{F} \\
]_{\mathtt{Y}}
\end{array}\right\}
$$

$\mathtt{returnval} := \mathtt{returnval} + 1\ ;\ \mathtt{current} := \mathtt{item}(\mathtt{nodelist}, \mathtt{returnval})$

$$
\left\{\begin{array}{l}
\exists \mathtt{NAME'''}, \mathtt{F'''}, \mathtt{FID'''}, \mathtt{VAL'''}, \mathtt{F'_3}, \mathtt{F'_4}.\ \mathtt{returnval} \doteq \mathrm{len}(\mathtt{F_3}) \wedge \mathtt{Y} \doteq \mathtt{nodelist} \\
\wedge\, \mathtt{NAME_{ID}}[\\
\quad \left((\mathtt{F_1}{:}\mathrm{F} \otimes_{\mathrm{F}} <(\mathtt{NAME'_{node}}[\mathtt{F'}{:}\mathrm{F}]_{\mathrm{FID'}} \vee \text{“\#text”}_{\mathtt{node}}\mathtt{VAL'}) \wedge \mathtt{T}{:}\mathrm{D}>_{\mathrm{F}}) \wedge \right. \\
\quad \left.(\mathtt{F'_3}{:}\mathrm{F} \otimes_{\mathrm{F}} <(\mathtt{NAME'''_{current}}[\mathtt{F'''}{:}\mathrm{F}]_{\mathrm{FID'''}} \vee \text{“\#text”}_{\mathtt{node}}\mathtt{VAL'''})>_{\mathrm{F}} \otimes_{\mathrm{F}} \mathtt{F_4}{:}\mathrm{F})\right) \otimes_{\mathrm{F}} \mathtt{F_2}{:}\mathrm{F} \\
]_{\mathtt{Y}}
\end{array}\right\}
$$

$\mathtt{od}\ ;$

$$
\left\{\begin{array}{l}
\exists \mathtt{NAME'''}, \mathtt{F'''}, \mathtt{FID'''}, \mathtt{VAL'''}, \mathtt{F'_3}, \mathtt{F'_4}. \\
\mathtt{returnval} \doteq \mathrm{len}(\mathtt{F_3}) \wedge \mathtt{current} \doteq \mathtt{node} \wedge \mathtt{Y} \doteq \mathtt{nodelist} \\
\wedge\, \mathtt{NAME_{ID}}[\\
\quad \left((\mathtt{F_1}{:}\mathrm{F} \otimes_{\mathrm{F}} <(\mathtt{NAME'_{node}}[\mathtt{F'}{:}\mathrm{F}]_{\mathrm{FID'}} \vee \text{“\#text”}_{\mathtt{node}}\mathtt{VAL'}) \wedge \mathtt{T}{:}\mathrm{D}>_{\mathrm{F}}) \wedge \right. \\
\quad \left.(\mathtt{F'_3}{:}\mathrm{F} \otimes_{\mathrm{F}} <(\mathtt{NAME'''_{current}}[\mathtt{F'''}{:}\mathrm{F}]_{\mathrm{FID'''}} \vee \text{“\#text”}_{\mathtt{node}}\mathtt{VAL'''})>_{\mathrm{F}} \otimes_{\mathrm{F}} \mathtt{F_4}{:}\mathrm{F})\right) \otimes_{\mathrm{F}} \mathtt{F_2}{:}\mathrm{F} \\
]_{\mathtt{Y}}
\end{array}\right\}
$$

$\mathtt{n} := \mathtt{returnval}$

$\mathtt{endloc}$

$$
\left\{\mathtt{NAME_{ID}}[\mathtt{F_1}{:}\mathrm{F} \otimes_{\mathrm{F}} <\mathtt{T}{:}\mathrm{D}>_{\mathrm{F}} \otimes_{\mathrm{F}} \mathtt{F_2}{:}\mathrm{F}]_{\mathtt{Y}} \wedge (\mathtt{n} = \mathrm{len}(\mathtt{F_1}))\right\}
$$

where $\mathrm{D}, \mathrm{D'} \in \{\mathrm{ELE}, \mathrm{TXT}\}$

Figure 5.2.: Derivation for the getIndex Specification

$\{<(\mathtt{NAME}_{\mathtt{node}}[\mathtt{F}{:}\mathrm{F}] \vee \text{“\#text”}_{\mathtt{node}}\mathtt{VAL}) \wedge \mathtt{T}{:}\mathrm{D}>_{\mathrm{G}}\}$
$\mathtt{sibling} := \mathtt{getPreviousSibling}(\mathtt{node}) \triangleq$
    local parent, children, n :
        parent := getParentNode(node) ;
        $\{<(\mathtt{NAME}_{\mathtt{node}}[\mathtt{F}{:}\mathrm{F}] \vee \text{“\#text”}_{\mathtt{node}}\mathtt{VAL}) \wedge \mathtt{T}{:}\mathrm{D}>_{\mathrm{G}} \wedge \mathtt{parent} \doteq \mathbf{null}\}$
        if parent = **null** **then** sibling := **null** **else** ...
    endloc
$\{<(\mathtt{NAME}_{\mathtt{node}}[\mathtt{F}{:}\mathrm{F}] \vee \text{“\#text”}_{\mathtt{node}}\mathtt{VAL}) \wedge \mathtt{T}{:}\mathrm{D}>_{\mathrm{G}} \wedge (\mathtt{sibling} \doteq \mathbf{null})\}$
where $\mathrm{D} \in \{\mathrm{ELE}, \mathrm{TXT}\}$

$\{\mathtt{NAME}_{\mathtt{ID}}[<(\mathtt{NAME}''_{\mathtt{node}}[\mathtt{F}'':\mathrm{F}]_{\mathtt{FID}''} \vee \mathtt{NAME}''_{\mathtt{node}}\mathtt{VAL}'') \wedge \mathtt{T}'':\mathrm{D}''>_{\mathrm{F}} \otimes_{\mathrm{F}} \mathtt{F}_2{:}\mathrm{F}]_{\mathtt{FID}}\}$
$\mathtt{sibling} := \mathtt{getPreviousSibling}(\mathtt{node}) \triangleq$
    local parent, children, n :
        parent := getParentNode(node) ;
        if parent := **null** **then** ... **else**
            $\left\{\begin{array}{l}\mathtt{NAME}_{\mathtt{ID}}[<(\mathtt{NAME}''_{\mathtt{node}}[\mathtt{F}'':\mathrm{F}]_{\mathtt{FID}''} \vee \mathtt{NAME}''_{\mathtt{node}}\mathtt{VAL}'') \wedge \mathtt{T}'':\mathrm{D}''>_{\mathrm{F}} \otimes_{\mathrm{F}} \mathtt{F}_2{:}\mathrm{F}]_{\mathtt{FID}} \\ \wedge\ \mathtt{parent} \doteq \mathtt{ID}\end{array}\right\}$
            children := getChildNodes(parent) ; n := getIndex(children, node) ;
            $\left\{\begin{array}{l}\mathtt{NAME}_{\mathtt{ID}}[<(\mathtt{NAME}''_{\mathtt{node}}[\mathtt{F}'':\mathrm{F}]_{\mathtt{FID}''} \vee \mathtt{NAME}''_{\mathtt{node}}\mathtt{VAL}'') \wedge \mathtt{T}'':\mathrm{D}''>_{\mathrm{F}} \otimes_{\mathrm{F}} \mathtt{F}_2{:}\mathrm{F}]_{\mathtt{FID}} \\ \wedge\ \mathtt{parent} \doteq \mathtt{ID} \wedge \mathtt{children} \doteq \mathtt{FID} \wedge \mathtt{n} \doteq 0\end{array}\right\}$
            sibling := item(nodelist, n − 1)
        fi
    endloc
$\{\mathtt{NAME}_{\mathtt{ID}}[<(\mathtt{NAME}''_{\mathtt{node}}[\mathtt{F}'':\mathrm{F}]_{\mathtt{FID}''} \vee \mathtt{NAME}''_{\mathtt{node}}\mathtt{VAL}'') \wedge \mathtt{T}'':\mathrm{D}''>_{\mathrm{F}} \otimes_{\mathrm{F}} \mathtt{F}_2{:}\mathrm{F}]_{\mathtt{FID}} \wedge (\mathtt{sibling} \doteq \mathbf{null})\}$
where $\mathrm{D} \in \{\mathrm{ELE}, \mathrm{TXT}\}$

Figure 5.3.: Derivations for the getPreviousSibling Specifications (Part One)

$$
\left\{\begin{array}{l}
\mathtt{NAME}_{\mathtt{ID}}[\\
\quad \mathtt{F}_1{:}F \otimes_{\mathrm{F}} \\
\quad <(\mathtt{NAME}'_{\mathtt{ID}'}[\mathtt{F}'{:}\mathrm{F}]_{\mathtt{FID}'} \vee \mathtt{NAME}'_{\mathtt{ID}'}\mathtt{VAL}') \wedge \mathtt{T}'{:}\mathrm{D}'>_{\mathrm{F}} \\
\quad \otimes_{\mathrm{F}} <(\mathtt{NAME}''_{\mathtt{node}}[\mathtt{F}''{:}\mathrm{F}]_{\mathtt{FID}''} \vee \mathtt{NAME}''_{\mathtt{node}}\mathtt{VAL}'') \wedge \mathtt{T}''{:}\mathrm{D}''>_{\mathrm{F}} \\
\quad \otimes_{\mathrm{F}} \mathtt{F}_2{:}\mathrm{F} \\
]_{\mathtt{FID}}
\end{array}\right\}
$$

$\mathtt{sibling} := \mathtt{getPreviousSibling}(\mathtt{node}) \triangleq$

```
local parent, children, n :
    parent := getParentNode(node) ;
    if parent := null then ... else
```

$$
\left\{\begin{array}{l}
\mathtt{NAME}_{\mathtt{ID}}[\\
\quad \mathtt{F}_1{:}F \otimes_{\mathrm{F}} \\
\quad <(\mathtt{NAME}'_{\mathtt{ID}'}[\mathtt{F}'{:}\mathrm{F}]_{\mathtt{FID}'} \vee \mathtt{NAME}'_{\mathtt{ID}'}\mathtt{VAL}') \wedge \mathtt{T}'{:}\mathrm{D}'>_{\mathrm{F}} \\
\quad \otimes_{\mathrm{F}} <(\mathtt{NAME}''_{\mathtt{node}}[\mathtt{F}''{:}\mathrm{F}]_{\mathtt{FID}''} \vee \mathtt{NAME}''_{\mathtt{node}}\mathtt{VAL}'') \wedge \mathtt{T}''{:}\mathrm{D}''>_{\mathrm{F}} \\
\quad \otimes_{\mathrm{F}} \mathtt{F}_2{:}\mathrm{F} \\
]_{\mathtt{FID}} \\
\wedge\, \mathtt{parent} \doteq \mathtt{ID}
\end{array}\right\}
$$

```
        children := getChildNodes(parent) ; n := getIndex(children, node) ;
```

$$
\left\{\begin{array}{l}
\mathtt{NAME}_{\mathtt{ID}}[\\
\quad \mathtt{F}_1{:}F \otimes_{\mathrm{F}} \\
\quad <(\mathtt{NAME}'_{\mathtt{ID}'}[\mathtt{F}'{:}\mathrm{F}]_{\mathtt{FID}'} \vee \mathtt{NAME}'_{\mathtt{ID}'}\mathtt{VAL}') \wedge \mathtt{T}'{:}\mathrm{D}'>_{\mathrm{F}} \\
\quad \otimes_{\mathrm{F}} <(\mathtt{NAME}''_{\mathtt{node}}[\mathtt{F}''{:}\mathrm{F}]_{\mathtt{FID}''} \vee \mathtt{NAME}''_{\mathtt{node}}\mathtt{VAL}'') \wedge \mathtt{T}''{:}\mathrm{D}''>_{\mathrm{F}} \\
\quad \otimes_{\mathrm{F}} \mathtt{F}_2{:}\mathrm{F} \\
]_{\mathtt{FID}} \\
\wedge\, \mathtt{parent} \doteq \mathtt{ID} \wedge \mathtt{children} \doteq \mathtt{FID} \wedge (\mathtt{n}-1) \doteq \mathrm{len}(\mathtt{F}_1)
\end{array}\right\}
$$

```
        sibling := item(nodelist, n − 1)
    fi
endloc
```

$$
\left\{\begin{array}{l}
\mathtt{NAME}_{\mathtt{ID}}[\\
\quad \mathtt{F}_1{:}F \otimes_{\mathrm{F}} \\
\quad <(\mathtt{NAME}'_{\mathtt{ID}'}[\mathtt{F}'{:}\mathrm{F}]_{\mathtt{FID}'} \vee \mathtt{NAME}'_{\mathtt{ID}'}\mathtt{VAL}') \wedge \mathtt{T}'{:}\mathrm{D}'>_{\mathrm{F}} \\
\quad \otimes_{\mathrm{F}} <(\mathtt{NAME}''_{\mathtt{node}}[\mathtt{F}''{:}\mathrm{F}]_{\mathtt{FID}''} \vee \mathtt{NAME}''_{\mathtt{node}}\mathtt{VAL}'') \wedge \mathtt{T}''{:}\mathrm{D}''>_{\mathrm{F}} \\
\quad \otimes_{\mathrm{F}} \mathtt{F}_2{:}\mathrm{F} \\
]_{\mathtt{FID}}\} \\
\wedge\, (\mathtt{sibling} = \mathtt{ID}')
\end{array}\right\}
$$

where $\mathrm{D}', \mathrm{D}'' \in \{\mathrm{ELE}, \mathrm{TXT}\}$

Figure 5.4.: Derivations for the getPreviousSibling Specifications (Part Two)

to find the position of `node` in that list, and `item` to return the previous one if it exists or **null** otherwise. If `node` is a root level node and therefore has no siblings, `getPreviousSibling` returns **null**.

`getIndex` is described by two complementary specifications. When `node` is an element of `nodeList` (the only case used by `getPreviousSibling`), the specification is:

$$\left\{ \begin{array}{l} \mathtt{NAME_{ID}}[\mathtt{F_1}{:}\mathrm{F} \otimes_{\mathrm{F}} {<}(\mathtt{NAME'_{node}}[\mathtt{F'}{:}\mathrm{F}]_{\mathtt{FID'}} \vee \text{“\#text”}_{\mathtt{node}}\mathtt{VAL'}) \wedge \mathtt{T}{:}\mathrm{D}{>}_{\mathrm{F}} \otimes_{\mathrm{F}} \mathtt{F_2}{:}\mathrm{F}]_{\mathtt{Y}} \\ \quad \wedge \mathtt{Y} \doteq \mathtt{nodeList} \end{array} \right\}$$
$$\mathtt{n} := \mathtt{getIndex}(\mathtt{nodeList}, \mathtt{node})$$
$$\{\mathtt{NAME_{ID}}[\mathtt{F_1}{:}\mathrm{F} \otimes_{\mathrm{F}} {<}\mathtt{T}{:}\mathrm{D}{>}_{\mathrm{F}} \otimes_{\mathrm{F}} \mathtt{F_2}{:}\mathrm{F}]_{\mathtt{Y}} \wedge (\mathtt{n} \doteq \mathrm{len}(\mathtt{F_1}))\}$$
$$\text{where } \mathrm{D} \in \{\mathrm{ELE}, \mathrm{TXT}\}$$

The precondition states that a tree identified by `node` is a child of a tree with a child list identified by `nodeList`. The postcondition states that the tree is unaltered, and that the store now records the position of the tree `node` in the variable `n`.

`getPreviousSibling`, meanwhile, is described using three specifications, corresponding to when the node is at the grove level, the beginning of a nodeList, or elsewhere.

If the node is at the grove level:

$$\{{<}(\mathtt{NAME_{node}}[\mathtt{F}{:}\mathrm{F}] \vee \text{“\#text”}_{\mathtt{node}}\mathtt{VAL}) \wedge \mathtt{T}{:}\mathrm{D}{>}_{\mathrm{G}}\}$$
$$\mathtt{sibling} := \mathtt{getPreviousSibling}(\mathtt{node})$$
$$\{{<}(\mathtt{NAME_{node}}[\mathtt{F}{:}\mathrm{F}] \vee \text{“\#text”}_{\mathtt{node}}\mathtt{VAL}) \wedge \mathtt{T}{:}\mathrm{D}{>}_{\mathrm{G}} \wedge (\mathtt{sibling} \doteq \mathbf{null})\}$$
$$\text{where } \mathrm{D} \in \{\mathrm{ELE}, \mathrm{TXT}\}$$

If the node is at the beginning of a nodeList:

$$\{\mathtt{NAME_{ID}}[{<}(\mathtt{NAME''_{node}}[\mathtt{F''}{:}\mathrm{F}]_{\mathtt{FID''}} \vee \mathtt{NAME''_{node}}\mathtt{VAL''}) \wedge \mathtt{T''}{:}\mathrm{D''}{>}_{\mathrm{F}} \otimes_{\mathrm{F}} \mathtt{F_2}{:}\mathrm{F}]_{\mathtt{FID}}\}$$
$$\mathtt{sibling} := \mathtt{getPreviousSibling}(\mathtt{node})$$
$$\left\{ \begin{array}{l} \mathtt{NAME_{ID}}[{<}(\mathtt{NAME''_{node}}[\mathtt{F''}{:}\mathrm{F}]_{\mathtt{FID''}} \vee \mathtt{NAME''_{node}}\mathtt{VAL''}) \wedge \mathtt{T''}{:}\mathrm{D''}{>}_{\mathrm{F}} \otimes_{\mathrm{F}} \mathtt{F_2}{:}\mathrm{F}]_{\mathtt{FID}} \\ \quad \wedge (\mathtt{sibling} \doteq \mathbf{null}) \end{array} \right\}$$
$$\text{where } \mathrm{D} \in \{\mathrm{ELE}, \mathrm{TXT}\}$$

```
firstChild := getFirstChild(node) ≜
    local kids :
        kids := getChildNodes(node) ;
        firstChild := item(kids, 0) ;
    endloc
```

Figure 5.5.: `getFirstChild`

If the node is elsewhere in a nodeList:

$$\left\{\begin{array}{l} \mathtt{NAME}_{\mathtt{ID}}[\\ \quad \mathtt{F}_1{:}F \otimes_{\mathrm{F}} \\ \quad <(\mathtt{NAME}'_{\mathtt{ID}'}[\mathtt{F}'{:}\mathrm{F}]_{\mathtt{FID}'} \vee \mathtt{NAME}'_{\mathtt{ID}'}\mathtt{VAL}') \wedge \mathtt{T}'{:}\mathrm{D}'>_{\mathrm{F}} \\ \quad \otimes_{\mathrm{F}} <(\mathtt{NAME}''_{\mathtt{node}}[\mathtt{F}''{:}\mathrm{F}]_{\mathtt{FID}''} \vee \mathtt{NAME}''_{\mathtt{node}}\mathtt{VAL}'') \wedge \mathtt{T}''{:}\mathrm{D}''>_{\mathrm{F}} \\ \quad \otimes_{\mathrm{F}} \mathtt{F}_2{:}\mathrm{F} \\ ]_{\mathtt{FID}} \end{array}\right\}$$

$\mathtt{sibling} := \mathtt{getPreviousSibling}(\mathtt{node})$

$$\left\{\begin{array}{l} \mathtt{NAME}_{\mathtt{ID}}[\\ \quad \mathtt{F}_1{:}F \otimes_{\mathrm{F}} \\ \quad <(\mathtt{NAME}'_{\mathtt{ID}'}[\mathtt{F}'{:}\mathrm{F}]_{\mathtt{FID}'} \vee \mathtt{NAME}'_{\mathtt{ID}'}\mathtt{VAL}') \wedge \mathtt{T}'{:}\mathrm{D}'>_{\mathrm{F}} \\ \quad \otimes_{\mathrm{F}} <(\mathtt{NAME}''_{\mathtt{node}}[\mathtt{F}''{:}\mathrm{F}]_{\mathtt{FID}''} \vee \mathtt{NAME}''_{\mathtt{node}}\mathtt{VAL}'') \wedge \mathtt{T}''{:}\mathrm{D}''>_{\mathrm{F}} \\ \quad \otimes_{\mathrm{F}} \mathtt{F}_2{:}\mathrm{F} \\ ]_{\mathtt{FID}}\} \\ \quad \wedge (\mathtt{sibling} \doteq \mathtt{ID}') \end{array}\right\}$$

where $\mathrm{D}', \mathrm{D}'' \in \{\mathrm{ELE}, \mathrm{TXT}\}$

The derivations for the specifications are given in Figures 5.3 and 5.4.

5.1.2. getFirstChild

The read-only object attribute "firstChild" is defined on the Node interface of DOM Core Level 1. We implement the getter command `getFirstChild` in Figure 5.5.

This command simply uses `getChildNodes` and `item` to discover the first child of the node `node` and assign its ID value to the variable `firstChild`. This command can be described with the following two axioms. In the case

where node has children:

$$\left\{ \begin{array}{l} \mathtt{NAME}_{\mathtt{node}}[<(\mathtt{NAME}'_{\mathtt{ID}'}[\mathtt{F}':\mathrm{F}]_{\mathrm{FID}'} \vee \text{“\#text”}_{\mathtt{ID}'}\mathtt{VAL}') \wedge \mathtt{T}:\mathrm{D}>_{\mathrm{F}} \otimes :\mathrm{F}\mathtt{F}_2:\mathrm{F}]_{\mathtt{FID}} \\ \wedge\, \mathtt{Y} \doteq \mathtt{node} \end{array} \right\}$$
$$\mathtt{firstChild} := \mathtt{getFirstChild}(\mathtt{node})$$
$$\left\{ \begin{array}{l} \mathtt{NAME}_{\mathtt{Y}}[<\mathtt{T}:\mathrm{D}>_{\mathrm{F}} \otimes :\mathrm{F}\mathtt{F}_2:\mathrm{F}]_{\mathtt{FID}} \\ \wedge\, \mathtt{firstChild} \doteq \mathtt{ID}' \end{array} \right\}$$

The precondition states the variable node refers to an element node which has at least one child. That child may be either an element or a text node. The postcondition adds that the variable firstChild refers to the first child of the node referred to by the variable node.

In the case where node has no children:

$$\left\{ \begin{array}{l} \mathtt{NAME}_{\mathtt{node}}[\varnothing_{\mathrm{F}}]_{\mathtt{FID}} \\ \wedge\, \mathtt{Y} \doteq \mathtt{node} \end{array} \right\}$$
$$\mathtt{firstChild} := \mathtt{getFirstChild}(\mathtt{node}) \triangleq$$
$$\left\{ \begin{array}{l} \mathtt{NAME}_{\mathtt{Y}}[\varnothing_{\mathrm{F}}]_{\mathtt{FID}} \\ \wedge\, \mathtt{firstChild} \doteq \mathbf{null} \end{array} \right\}$$

The precondition states that node refers to an element node which has no children. The postcondition adds that the variable firstChild takes the value **null**.

The derivations of these specifications are given in Figure 5.6

5.1.3. getDataLength

The CharacterData interface of DOM Core Level 1 defines the read-only object attribute “length”. We implement this functionality with the getter command getDataLength. This command returns the length of the data stored in a given text node. The implementation of getDataLength is given in Figure 5.7

This command uses a simple while loop to traverse the value of the text node referred to by node, keeping a count of the number of characters encountered. By the time the command has terminated, the length of the value of the text node node has been recorded in the variable length. This behaviour can be given by the following specification:

$$\left\{ \begin{array}{l} \mathtt{NAME}_{\mathtt{node}}[<(\mathtt{NAME}'_{\mathtt{ID}'}[\mathtt{F}':\mathrm{F}]_{\mathrm{FID}'} \vee \text{“\#text”}_{\mathtt{ID}'}\mathtt{VAL}') \wedge \mathtt{T}:\mathrm{D}>_{\mathrm{F}} \otimes_{\mathrm{F}} \mathtt{F}_2:\mathrm{F}]_{\mathtt{FID}} \\ \wedge\, \mathtt{Y} \doteq \mathtt{node} \end{array} \right\}$$
$\mathtt{firstChild} := \mathtt{getFirstChild}(\mathtt{node}) \triangleq$
    $\mathtt{local\ kids}:$
        $\mathtt{kids} := \mathtt{getChildNodes}(\mathtt{node})\ ;$
        $\{\ \mathtt{NAME}_{\mathtt{Y}}[<(\mathtt{NAME}'_{\mathtt{ID}'}[\mathtt{F}':\mathrm{F}]_{\mathrm{FID}'} \vee \text{“\#text”}_{\mathtt{ID}'}\mathtt{VAL}') \wedge \mathtt{T}:\mathrm{D}>_{\mathrm{F}} \otimes_{\mathrm{F}} \mathtt{F}_2:\mathrm{F}]_{\mathtt{kids}}\ \}$
        $\mathtt{firstChild} := \mathtt{item}(\mathtt{kids}, 0)\ ;$
    $\mathtt{endloc}$
$$\left\{ \begin{array}{l} \mathtt{NAME}_{\mathtt{Y}}[<\mathtt{T}:\mathrm{D}>_{\mathrm{F}} \otimes_{\mathrm{F}} \mathtt{F}_2:\mathrm{F}]_{\mathtt{FID}} \\ \wedge\, \mathtt{firstChild} \doteq \mathtt{ID}' \end{array} \right\}$$

$$\left\{ \begin{array}{l} \mathtt{NAME}_{\mathtt{node}}[\varnothing_{\mathrm{F}}]_{\mathtt{FID}} \\ \wedge\, \mathtt{Y} \doteq \mathtt{node} \end{array} \right\}$$
$\mathtt{firstChild} := \mathtt{getFirstChild}(\mathtt{node}) \triangleq$
    $\mathtt{local\ kids}:$
        $\mathtt{kids} := \mathtt{getChildNodes}(\mathtt{node})\ ;$
        $\{\ \mathtt{NAME}_{\mathtt{Y}}[\varnothing_{\mathrm{F}}]_{\mathtt{kids}}\ \}$
        $\mathtt{firstChild} := \mathtt{item}(\mathtt{kids}, 0)\ ;$
    $\mathtt{endloc}$
$$\left\{ \begin{array}{l} \mathtt{NAME}_{\mathtt{Y}}[\varnothing_{\mathrm{F}}]_{\mathtt{FID}} \\ \wedge\, \mathtt{firstChild} \doteq \mathbf{null} \end{array} \right\}$$

Figure 5.6.: Derivation of the `getFirstChild` axioms

$\mathtt{length} := \mathtt{getDataLength}(\mathtt{node}) \triangleq$
    $\mathtt{local\ str}:$
        $\mathtt{length} := 0\ ;$
        $\mathtt{str} := \mathtt{substringData}(\mathtt{node}, \mathtt{length}, 1)\ ;$
        $\mathtt{while\ str} \neq \varnothing_{\mathrm{S}}\ \mathtt{do}$
            $\mathtt{length} := \mathtt{length} + 1\ ;$
            $\mathtt{str} := \mathtt{substringData}(\mathtt{node}, \mathtt{length}, 1)$
        $\mathtt{od}$
    $\mathtt{endloc}$

Figure 5.7.: The `getDataLength` Command

$$\left\{ \text{“\#text”}_{\texttt{Y}}\texttt{VAL} \wedge \texttt{Y} \doteq \texttt{node} \right\}$$
$$\texttt{length} := \texttt{getDataLength}(\texttt{node})$$
$$\left\{ \begin{array}{l} \text{“\#text”}_{\texttt{Y}}\texttt{VAL} \\ \wedge\, \text{len}(\texttt{VAL}) \doteq \texttt{length} \end{array} \right\}$$

The precondition states that the variable `node` refers to a text node. The postcondition adds that the variable `length` contains the length of the value of that text node. The derivation of this specification is given in Figure 5.8.

5.1.4. getData

The CharacterData interface of DOM Core Level 1 defines the object attribute “data”. We implement the read-functionality of this object attribute with the getter command `getData`, which returns the data stored in the value of a text node. The implementation is in Figure 5.9.

This command simply uses the previously defined `getDataLength` command and the `substringData` command to assign the value of a text node to the variable `data`. The specification is:

$$\left\{ \text{“\#text”}_{\texttt{node}}\texttt{VAL} \wedge \texttt{Y} \doteq \texttt{node} \right\}$$
$$\texttt{data} := \texttt{getData}(\texttt{node}) \triangleq$$
$$\left\{ \text{“\#text”}_{\texttt{Y}}\texttt{VAL} \wedge \texttt{data} \doteq \texttt{VAL} \right\}$$

The precondition simply states that `node` refers to a text node, while the postcondition adds that the variable `data` has taken the value of that text node. This specification is derived in Figure 5.10

5.2. Weakest Preconditions

In [72], Zarfaty used the inference rules of his program reasoning to derive the weakest preconditions of each of the commands in his BTU programming language from their axioms. In doing so, he showed that his reasoning was complete for straight-line code. In this section, we follow his example and present the derivation of the weakest precondition for the `getNodeName` command. The derivations for the remaining commands are given in Appendix A.2.

$$
\begin{array}{l}
\{\ \text{“\#text”}_{\mathtt{Y}}\mathtt{VAL} \wedge \mathtt{Y} \doteq \mathtt{node}\ \} \\
\mathtt{length} := \mathtt{getDataLength}(\mathtt{node}) \triangleq \\
\quad \mathtt{local\ str}: \\
\qquad \mathtt{length} := 0\ ; \\
\qquad \{\ \text{“\#text”}_{\mathtt{Y}}\mathtt{VAL} \wedge \mathtt{length} \doteq 0\ \} \\
\qquad \mathtt{str} := \mathtt{substringData}(\mathtt{node}, \mathtt{length}, 1)\ ; \\
\qquad \left\{ \begin{array}{l} \exists \mathtt{A}, \mathtt{B}, \mathtt{C}.\ \text{“\#text”}_{\mathtt{Y}}\mathtt{VAL} \wedge \mathtt{VAL} \equiv \mathtt{A} \otimes_{\mathrm{S}} \mathtt{B} \otimes_{\mathrm{S}} \mathtt{C} \\ \wedge\ \mathtt{length} \doteq 0 \wedge \mathrm{len}(\mathtt{A}) \doteq \mathtt{length} \wedge \\ ((\mathtt{str} \doteq \mathtt{B} \wedge \mathrm{len}(\mathtt{B}) \doteq 1) \vee (\mathtt{str} \doteq \mathtt{B} \otimes_{\mathrm{S}} \mathtt{C} \wedge \mathrm{len}(\mathtt{B} \otimes_{\mathrm{S}} \mathtt{C}) <\cdot 1)) \end{array} \right\} \\
\qquad \left\{ \begin{array}{l} \exists \mathtt{A}, \mathtt{B}, \mathtt{C}.\ \text{“\#text”}_{\mathtt{Y}}\mathtt{VAL} \wedge \mathtt{VAL} \equiv \mathtt{A} \otimes_{\mathrm{S}} \mathtt{B} \otimes_{\mathrm{S}} \mathtt{C} \\ \wedge\ \mathrm{len}(\mathtt{A}) \doteq \mathtt{length} \wedge \\ ((\mathtt{str} \doteq \mathtt{B} \wedge \mathrm{len}(\mathtt{B}) \doteq 1) \vee (\mathtt{str} \doteq \mathtt{B} \otimes_{\mathrm{S}} \mathtt{C} \wedge \mathrm{len}(\mathtt{B} \otimes_{\mathrm{S}} \mathtt{C}) <\cdot 1)) \end{array} \right\} \\
\qquad \mathtt{while\ str} \neq \varnothing_{\mathrm{S}}\ \mathtt{do} \\
\qquad\quad \left\{ \begin{array}{l} \exists \mathtt{A}, \mathtt{B}, \mathtt{C}.\ \text{“\#text”}_{\mathtt{Y}}\mathtt{VAL} \wedge \mathtt{VAL} \equiv \mathtt{A} \otimes_{\mathrm{S}} \mathtt{B} \otimes_{\mathrm{S}} \mathtt{C} \\ \wedge\ \mathrm{len}(\mathtt{A}) \doteq \mathtt{length} \wedge \\ (\mathtt{str} \doteq \mathtt{B} \wedge \mathrm{len}(\mathtt{B}) \doteq 1) \end{array} \right\} \\
\qquad\quad \mathtt{length} := \mathtt{length} + 1\ ; \\
\qquad\quad \left\{ \begin{array}{l} \exists \mathtt{A}, \mathtt{B}, \mathtt{C}.\ \text{“\#text”}_{\mathtt{Y}}\mathtt{VAL} \wedge \mathtt{VAL} \equiv \mathtt{A} \otimes_{\mathrm{S}} \mathtt{B} \otimes_{\mathrm{S}} \mathtt{C} \\ \wedge\ \mathrm{len}(\mathtt{A} \otimes_{\mathrm{S}} \mathtt{B}) \doteq \mathtt{length} \wedge \\ (\mathtt{str} \doteq \mathtt{B} \wedge \mathrm{len}(\mathtt{B}) \doteq 1) \end{array} \right\} \\
\qquad\quad \mathtt{str} := \mathtt{substringData}(\mathtt{node}, \mathtt{length}, 1) \\
\qquad\quad \left\{ \begin{array}{l} \exists \mathtt{A}, \mathtt{B}, \mathtt{C}.\ \text{“\#text”}_{\mathtt{Y}}\mathtt{VAL} \wedge \mathtt{VAL} \equiv \mathtt{A} \otimes_{\mathrm{S}} \mathtt{B} \otimes_{\mathrm{S}} \mathtt{C} \\ \wedge\ \mathrm{len}(\mathtt{A}) \doteq \mathtt{length} \wedge \\ ((\mathtt{str} \doteq \mathtt{B} \wedge \mathrm{len}(\mathtt{B}) \doteq 1) \vee (\mathtt{str} \doteq \mathtt{B} \otimes_{\mathrm{S}} \mathtt{C} \wedge \mathrm{len}(\mathtt{B} \otimes_{\mathrm{S}} \mathtt{C}) <\cdot 1)) \end{array} \right\} \\
\qquad \mathtt{od} \\
\qquad \left\{ \begin{array}{l} \mathtt{str} \doteq \varnothing_{\mathrm{S}} \wedge \\ \exists \mathtt{A}, \mathtt{B}, \mathtt{C}.\ \text{“\#text”}_{\mathtt{Y}}\mathtt{VAL} \wedge \mathtt{VAL} \equiv \mathtt{A} \otimes_{\mathrm{S}} \mathtt{B} \otimes_{\mathrm{S}} \mathtt{C} \\ \wedge\ \mathrm{len}(\mathtt{A}) \doteq \mathtt{length} \wedge \\ ((\mathtt{str} \doteq \mathtt{B} \wedge \mathrm{len}(\mathtt{B}) \doteq 1) \vee (\mathtt{str} \doteq \mathtt{B} \otimes_{\mathrm{S}} \mathtt{C} \wedge \mathrm{len}(\mathtt{B} \otimes_{\mathrm{S}} \mathtt{C}) <\cdot 1)) \end{array} \right\} \\
\qquad \left\{ \begin{array}{l} \exists \mathtt{A}, \mathtt{B}, \mathtt{C}.\ \text{“\#text”}_{\mathtt{Y}}\mathtt{VAL} \wedge \mathtt{VAL} \equiv \mathtt{A} \otimes_{\mathrm{S}} \mathtt{B} \otimes_{\mathrm{S}} \mathtt{C} \\ \wedge\ \mathrm{len}(\mathtt{A}) \doteq \mathtt{length} \wedge \\ (\mathtt{str} \doteq \mathtt{B} \otimes_{\mathrm{S}} \mathtt{C} \wedge \mathrm{len}(\mathtt{B} \otimes_{\mathrm{S}} \mathtt{C}) <\cdot 1) \end{array} \right\} \\
\quad \mathtt{endloc} \\
\left\{ \begin{array}{l} \text{“\#text”}_{\mathtt{Y}}\mathtt{VAL} \\ \wedge\ \mathrm{len}(\mathtt{VAL}) \doteq \mathtt{length} \end{array} \right\}
\end{array}
$$

Figure 5.8.: Derivation of `getDataLength`

$$
\begin{array}{l}
\texttt{data} := \texttt{getData}(\texttt{node}) \triangleq \\
\quad \texttt{local length} : \\
\quad\quad \texttt{length} := \texttt{getDataLength}(\texttt{node}) \ ; \\
\quad\quad \texttt{data} := \texttt{substringData}(\texttt{node}, 0, \texttt{length}) \\
\quad \texttt{endloc}
\end{array}
$$

Figure 5.9.: The `getData` command

$$
\begin{array}{l}
\{ \ \text{“\#text”}_{\texttt{node}}\texttt{VAL} \wedge \texttt{Y} \doteq \texttt{node} \ \} \\
\texttt{data} := \texttt{getData}(\texttt{node}) \triangleq \\
\quad \texttt{local length} : \\
\quad\quad \texttt{length} := \texttt{getDataLength}(\texttt{node}) \ ; \\
\quad\quad \{ \ \text{“\#text”}_{\texttt{node}}\texttt{VAL} \wedge \texttt{Y} \doteq \texttt{node} \wedge \texttt{length} \doteq \text{len}(\texttt{VAL}) \ \} \\
\quad\quad \texttt{data} := \texttt{substringData}(\texttt{node}, 0, \texttt{length}) \\
\quad \texttt{endloc} \\
\quad \{ \ \text{“\#text”}_{\texttt{Y}}\texttt{VAL} \wedge \texttt{data} \doteq \texttt{VAL} \ \}
\end{array}
$$

Figure 5.10.: Derivation of the `getData` Command

The axiom for the `getNodeName` command is:

$$
\begin{array}{l}
\{(\texttt{NAME}_{\texttt{node}}[\texttt{F:F}]_{\texttt{FID}} \vee \texttt{NAME}_{\texttt{node}}\texttt{VAL}) \wedge \texttt{T:D}\} \\
\texttt{var} := \texttt{getNodeName}(\texttt{node}) \\
\{\texttt{T:D} \wedge (\texttt{var} \doteq \texttt{NAME})\}
\end{array}
$$

In order to derive the weakest precondition of this command, we first apply the frame rule to the axiom using a carefully chosen frame. We choose this frame such that the resulting context application will leave a postcondition equivalent to “P”, which is the general postcondition we wish to find the weakest precondition for. Choosing such a frame is straightforward, since we may use the adjoint “$-\!\circ$”. Once we have applied this frame, we use the rules of consequence and variable elimination to simplify the postcondition to “P”. The derivation is given in Figure 5.11.

5.3. Compliance Testing

The W3C provide a collection of “conformance tests”[24], each of which tests for a specific behaviour of a DOM implementation. Tests suites of

$\{(\texttt{NAME}_{\texttt{node}}[\texttt{F:F}]_{\texttt{FID}} \vee \text{“\#text”}_{\texttt{node}}\texttt{VAL}) \wedge \texttt{T:D}\}$
$\texttt{var} := \texttt{getNodeName}(\texttt{node})$
$\{\texttt{T:D} \wedge (\texttt{var} \doteq \texttt{NAME})\}$

FRAME

$\left\{\begin{array}{l}(\texttt{T:D} \multimap P\{\texttt{NAME}/\texttt{var}\}) \circ_{\text{D}} \\ ((\texttt{NAME}_{\texttt{node}}[\texttt{F:F}]_{\texttt{FID}} \vee \text{“\#text”}_{\texttt{node}}\texttt{VAL}) \wedge \texttt{T:D})\end{array}\right\}$
$\texttt{var} := \texttt{getNodeName}(\texttt{node})$
$\left\{\begin{array}{l}(\texttt{T:D} \multimap P\{\texttt{NAME}/\texttt{var}\}) \circ_{\text{D}} \\ (\texttt{T:D} \wedge (\texttt{var} \doteq \texttt{NAME}))\end{array}\right\}$

CONS

$\{\Diamond_{\text{D}\to\text{D}'}((\texttt{NAME}_{\texttt{node}}[\texttt{F:F}]_{\texttt{FID}} \vee \text{“\#text”}_{\texttt{node}}\texttt{VAL}) \wedge \texttt{T:D}) \wedge P\{\texttt{NAME}/\texttt{var}\}\}$
$\texttt{var} := \texttt{getNodeName}(\texttt{node})$
$\{P\{\texttt{NAME}/\texttt{var}\} \wedge (\texttt{var} \doteq \texttt{NAME})\}$

CONS/ELIM

$\left\{\begin{array}{l}\exists \texttt{NAME}, \texttt{F}, \texttt{FID}, \texttt{VAL}, \texttt{T}. \\ \Diamond_{\text{D}\to\text{D}'}((\texttt{NAME}_{\texttt{node}}[\texttt{F:F}]_{\texttt{FID}} \vee \text{“\#text”}_{\texttt{node}}\texttt{VAL}) \wedge \texttt{T:D}) \wedge P\{\texttt{NAME}/\texttt{var}\}\end{array}\right\}$
$\texttt{var} := \texttt{getNodeName}(\texttt{node})$
$\{P\}$

where $\text{D} \in \{\text{ELE}, \text{TXT}\}, \text{D}' \in \{\text{ELE}, \text{TXT}, \text{F}, \text{G}\}$

Figure 5.11.: Weakest Precondition Derivation for `getNodeName`

this sort can provide good evidence that a particular implementation of DOM satisfies the specification. In [33] there is ongoing work to connect high level context logic reasoning about specifications such as DOM to low level reasoning about implementations of such libraries. One goal of that line of research is to make it more feasible to prove complete implementations against formal specifications of the sort presented in this thesis. Conformance tests are still useful however, since they can be seen as another partial representation of the specification itself. If a formalism such as the one given in this thesis truely represents the informal specification in the specification document and the conformance tests, then it should be possible to use that formalism to prove that each of the tests pass.

This section presents an adaptation of a typical test to the context of Featherweight DOM. The test in question is the "characterdataAppendDataGetData" test, which tests the results of appending data to a text node and then getting the value of that text node. The test expects that the value of the text node should be the concatenation of its original value and the value that has been appended.

We make two particular changes to the original code of the test given in [24]:

Firstly, rather than using runtime "assert" statements of the sort that may be found in many programming languages to test the results of running the test program, we use our assertion language to describe the expected result, and then we prove that the program always terminates in a state satisfying that assertion.

Secondly, since Featherweight DOM does not contain any methods for loading XML files or for performing a search after the fashion of the DOM Level 1 command "getElementsByTagName", we start the test in a state pre-seeded with the appropriate data.

This test operates on the data given in Figure 5.12 Using the code given in Figure 5.13 where the childlist of the first "employee" node can be referred to using the variable "employeeChildren". The test will pass if the value of the program variable "childData" at the end of the program is the string "Margaret Martin, Esquire". The postcondition of the test program is therefore $\{\texttt{childData} \doteq \text{“Margaret Martin, Esquire”} \wedge \text{true}_{\text{ELE}}\}$ while the

```
<employees>
         <employee>
                <employeeId>EMP0001</employeeId>
                <name>Margaret Martin</name>
                <position>Accountant</position>
                <salary>56,000</salary>
                <gender>Female</gender>
                <address>1230 North Ave. Dallas, Texas 98551</address>
        </employee>
         <employee>
                <employeeId>EMP0002</employeeId>
                <name>Martha Raynolds</name>
                <position>Secretary</position>
                <salary>35,000</salary>
                <gender>Female</gender>
                <address>1900 Dallas Road Dallas, Texas 98554</address>
        </employee>
</employees>
```

Figure 5.12.: Test Data for the "characterdataAppendDataGetData" Test

```
local nameNode, child, childData :
    nameNode := item(employeeChildren, 1) ;
    child := getFirstChild(nameNode) ;
    appendData(child, ", Esquire") ;
    childData := getData(child)
endloc
```

Figure 5.13.: The "characterdataAppendDataGetData" Test

precondition is:

$$
\left\{
\begin{array}{l}
\text{“employees”}_{\mathtt{ID}_1}[\\
\quad <\text{“employee”}_{\mathtt{ID}_2}[\\
\quad\quad <\text{“employeeId”}_{\mathtt{ID}_3}[<\text{“\#text”}_{\mathtt{ID}_4}\text{“EMP0001”}>_{\mathrm{F}}]_{\mathtt{FID}_3}>_{\mathrm{F}} \otimes_{\mathrm{F}} \\
\quad\quad <\text{“name”}_{\mathtt{ID}_5}[<\text{“\#text”}_{\mathtt{ID}_6}\text{“Margaret Martin”}>_{\mathrm{F}}]_{\mathtt{FID}_5}>_{\mathrm{F}} \otimes_{\mathrm{F}} \\
\quad\quad <\text{“position”}_{\mathtt{ID}_7}[<\text{“\#text”}_{\mathtt{ID}_8}\text{“Accountant”}>_{\mathrm{F}}]_{\mathtt{FID}_7}>_{\mathrm{F}} \otimes_{\mathrm{F}} \\
\quad\quad <\text{“salary”}_{\mathtt{ID}_9}[<\text{“\#text”}_{\mathtt{ID}_{10}}\text{“56,000”}>_{\mathrm{F}}]_{\mathtt{FID}_9}>_{\mathrm{F}} \otimes_{\mathrm{F}} \\
\quad\quad <\text{“gender”}_{\mathtt{ID}_{11}}[<\text{“\#text”}_{\mathtt{ID}_{12}}\text{“Female”}>_{\mathrm{F}}]_{\mathtt{FID}_{11}}>_{\mathrm{F}} \otimes_{\mathrm{F}} \\
\quad\quad <\text{“address”}_{\mathtt{ID}_{13}}[\\
\quad\quad\quad <\text{“\#text”}_{\mathtt{ID}_{14}}\text{“1230 North Ave. Dallas, Texas 98551”}>_{\mathrm{F}} \\
\quad\quad]_{\mathtt{FID}_{13}}>_{\mathrm{F}} \\
\quad]_{\mathtt{employeeChildren}}>_{\mathrm{F}} \otimes_{\mathrm{F}} \\
\quad <\text{“employee”}_{\mathtt{ID}_{15}}[\\
\quad\quad <\text{“employeeId”}_{\mathtt{ID}_{16}}[<\text{“\#text”}_{\mathtt{ID}_{17}}\text{“EMP0002”}>_{\mathrm{F}}]_{\mathtt{FID}_{16}}>_{\mathrm{F}} \otimes_{\mathrm{F}} \\
\quad\quad <\text{“name”}_{\mathtt{ID}_{18}}[<\text{“\#text”}_{\mathtt{ID}_{19}}\text{“Martha Raynolds”}>_{\mathrm{F}}]_{\mathtt{FID}_{18}}>_{\mathrm{F}} \otimes_{\mathrm{F}} \\
\quad\quad <\text{“position”}_{\mathtt{ID}_{20}}[<\text{“\#text”}_{\mathtt{ID}_{21}}\text{“Secretary”}>_{\mathrm{F}}]_{\mathtt{FID}_{20}}>_{\mathrm{F}} \otimes_{\mathrm{F}} \\
\quad\quad <\text{“salary”}_{\mathtt{ID}_{22}}[<\text{“\#text”}_{\mathtt{ID}_{23}}\text{“35,000”}>_{\mathrm{F}}]_{\mathtt{FID}_{22}}>_{\mathrm{F}} \otimes_{\mathrm{F}} \\
\quad\quad <\text{“gender”}_{\mathtt{ID}_{24}}[<\text{“\#text”}_{\mathtt{ID}_{25}}\text{“Female”}>_{\mathrm{F}}]_{\mathtt{FID}_{24}}>_{\mathrm{F}} \otimes_{\mathrm{F}} \\
\quad\quad <\text{“address”}_{\mathtt{ID}_{26}}[\\
\quad\quad\quad <\text{“\#text”}_{\mathtt{ID}_{27}}\text{“1900 Dallas Road Dallas, Texas 98554”}>_{\mathrm{F}} \\
\quad\quad]_{\mathtt{FID}_{26}}>_{\mathrm{F}} \\
\quad]_{\mathtt{FID}_{15}}>_{\mathrm{F}} \\
]_{\mathtt{FID}_1}
\end{array}
\right\}
$$

Since the precondition describes more data than the program uses, the first step in proving the program is to use the frame rule to observe that it is sufficient to show that the postcondition (which may describe element

data of any size) holds given the smaller precondition:

$$\left\{\begin{array}{l}
\text{“employee”}_{\texttt{ID}_2}[\\
\quad <\text{“employeeId”}_{\texttt{ID}_3}[<\text{“\#text”}_{\texttt{ID}_4}\text{“EMP0001”}>_{\mathrm{F}}]_{\texttt{FID}_3}>_{\mathrm{F}} \otimes_{\mathrm{F}} \\
\quad <\text{“name”}_{\texttt{ID}_5}[<\text{“\#text”}_{\texttt{ID}_6}\text{“Margaret Martin”}>_{\mathrm{F}}]_{\texttt{FID}_5}>_{\mathrm{F}} \otimes_{\mathrm{F}} \\
\quad <\text{“position”}_{\texttt{ID}_7}[<\text{“\#text”}_{\texttt{ID}_8}\text{“Accountant”}>_{\mathrm{F}}]_{\texttt{FID}_7}>_{\mathrm{F}} \otimes_{\mathrm{F}} \\
\quad <\text{“salary”}_{\texttt{ID}_9}[<\text{“\#text”}_{\texttt{ID}_{10}}\text{“56,000”}>_{\mathrm{F}}]_{\texttt{FID}_9}>_{\mathrm{F}} \otimes_{\mathrm{F}} \\
\quad <\text{“gender”}_{\texttt{ID}_{11}}[<\text{“\#text”}_{\texttt{ID}_{12}}\text{“Female”}>_{\mathrm{F}}]_{\texttt{FID}_{11}}>_{\mathrm{F}} \otimes_{\mathrm{F}} \\
\quad <\text{“address”}_{\texttt{ID}_{13}}[\\
\quad\quad <\text{“\#text”}_{\texttt{ID}_{14}}\text{“1230 North Ave. Dallas, Texas 98551”}>_{\mathrm{F}} \\
\quad]_{\texttt{FID}_{13}}>_{\mathrm{F}} \\
]_{\texttt{employeeChildren}}
\end{array}\right\}$$

The proof of the test program is given in Figure 5.14

5.4. Proving Schema Invariants

When reasoning about programs, it is often desirable to prove a particular property about a program rather than proving the whole (often complex) specification. One example of this involves proving XML schema invariants. For example, consider writing a program to update an XML addressBook document which complies with the XML schema in Figure 5.15

The schema asserts that the root element of the document should be an `addressBook` node, whose children should be zero or more `household` nodes. These `household` nodes should contain one or more `person` nodes, one `address` node and one `phone` node. Each of these third-level nodes should contain data of type ‘string’.

$$
\left\{\begin{array}{l}
\text{“employee”}_{\mathtt{ID}_2}[\\
\quad <\text{“employeeId”}_{\mathtt{ID}_3}[<\text{“\#text”}_{\mathtt{ID}_4}\ \text{“EMP0001”}>_{\mathrm{F}}]_{\mathtt{FID}_3}>_{\mathrm{F}} \otimes_{\mathrm{F}} \\
\quad <\text{“name”}_{\mathtt{ID}_5}[<\text{“\#text”}_{\mathtt{ID}_6}\ \text{“Margaret Martin”}>_{\mathrm{F}}]_{\mathtt{FID}_5}>_{\mathrm{F}} \otimes_{\mathrm{F}} \\
\quad <\text{“position”}_{\mathtt{ID}_7}[<\text{“\#text”}_{\mathtt{ID}_8}\ \text{“Accountant”}>_{\mathrm{F}}]_{\mathtt{FID}_7}>_{\mathrm{F}} \otimes_{\mathrm{F}} \\
\quad <\text{“salary”}_{\mathtt{ID}_9}[<\text{“\#text”}_{\mathtt{ID}_{10}}\ \text{“56,000”}>_{\mathrm{F}}]_{\mathtt{FID}_9}>_{\mathrm{F}} \otimes_{\mathrm{F}} \\
\quad <\text{“gender”}_{\mathtt{ID}_{11}}[<\text{“\#text”}_{\mathtt{ID}_{12}}\ \text{“Female”}>_{\mathrm{F}}]_{\mathtt{FID}_{11}}>_{\mathrm{F}} \otimes_{\mathrm{F}} \\
\quad <\text{“address”}_{\mathtt{ID}_{13}}[\\
\quad\quad <\text{“\#text”}_{\mathtt{ID}_{14}}\ \text{“1230 North Ave. Dallas, Texas 98551”}>_{\mathrm{F}} \\
\quad]_{\mathtt{FID}_{13}}>_{\mathrm{F}} \\
]_{\mathtt{employeeChildren}}
\end{array}\right\}
$$

$\mathtt{local\ nameNode, child, childData}$:

$\mathtt{nameNode} := \mathtt{item}(\mathtt{employeeChildren}, 1)$;

$$
\left\{\begin{array}{l}
\text{“employee”}_{\mathtt{ID}_2}[\\
\quad <\text{“employeeId”}_{\mathtt{ID}_3}[<\text{“\#text”}_{\mathtt{ID}_4}\ \text{“EMP0001”}>_{\mathrm{F}}]_{\mathtt{FID}_3}>_{\mathrm{F}} \otimes_{\mathrm{F}} \\
\quad <\text{“name”}_{\mathtt{nameNode}}[<\text{“\#text”}_{\mathtt{ID}_6}\ \text{“Margaret Martin”}>_{\mathrm{F}}]_{\mathtt{FID}_5}>_{\mathrm{F}} \otimes_{\mathrm{F}} \\
\quad <\text{“position”}_{\mathtt{ID}_7}[<\text{“\#text”}_{\mathtt{ID}_8}\ \text{“Accountant”}>_{\mathrm{F}}]_{\mathtt{FID}_7}>_{\mathrm{F}} \otimes_{\mathrm{F}} \\
\quad <\text{“salary”}_{\mathtt{ID}_9}[<\text{“\#text”}_{\mathtt{ID}_{10}}\ \text{“56,000”}>_{\mathrm{F}}]_{\mathtt{FID}_9}>_{\mathrm{F}} \otimes_{\mathrm{F}} \\
\quad <\text{“gender”}_{\mathtt{ID}_{11}}[<\text{“\#text”}_{\mathtt{ID}_{12}}\ \text{“Female”}>_{\mathrm{F}}]_{\mathtt{FID}_{11}}>_{\mathrm{F}} \otimes_{\mathrm{F}} \\
\quad <\text{“address”}_{\mathtt{ID}_{13}}[\\
\quad\quad <\text{“\#text”}_{\mathtt{ID}_{14}}\ \text{“1230 North Ave. Dallas, Texas 98551”}>_{\mathrm{F}} \\
\quad]_{\mathtt{FID}_{13}}>_{\mathrm{F}} \\
]_{\mathtt{employeeChildren}}
\end{array}\right\}
$$

$\mathtt{child} := \mathtt{getFirstChild}(\mathtt{nameNode})$;

$$
\left\{\begin{array}{l}
\text{“employee”}_{\mathtt{ID}_2}[\\
\quad <\text{“employeeId”}_{\mathtt{ID}_3}[<\text{“\#text”}_{\mathtt{ID}_4}\ \text{“EMP0001”}>_{\mathrm{F}}]_{\mathtt{FID}_3}>_{\mathrm{F}} \otimes_{\mathrm{F}} \\
\quad <\text{“name”}_{\mathtt{nameNode}}[<\text{“\#text”}_{\mathtt{child}}\ \text{“Margaret Martin”}>_{\mathrm{F}}]_{\mathtt{FID}_5}>_{\mathrm{F}} \otimes_{\mathrm{F}} \\
\quad <\text{“position”}_{\mathtt{ID}_7}[<\text{“\#text”}_{\mathtt{ID}_8}\ \text{“Accountant”}>_{\mathrm{F}}]_{\mathtt{FID}_7}>_{\mathrm{F}} \otimes_{\mathrm{F}} \\
\quad <\text{“salary”}_{\mathtt{ID}_9}[<\text{“\#text”}_{\mathtt{ID}_{10}}\ \text{“56,000”}>_{\mathrm{F}}]_{\mathtt{FID}_9}>_{\mathrm{F}} \otimes_{\mathrm{F}} \\
\quad <\text{“gender”}_{\mathtt{ID}_{11}}[<\text{“\#text”}_{\mathtt{ID}_{12}}\ \text{“Female”}>_{\mathrm{F}}]_{\mathtt{FID}_{11}}>_{\mathrm{F}} \otimes_{\mathrm{F}} \\
\quad <\text{“address”}_{\mathtt{ID}_{13}}[\\
\quad\quad <\text{“\#text”}_{\mathtt{ID}_{14}}\ \text{“1230 North Ave. Dallas, Texas 98551”}>_{\mathrm{F}} \\
\quad]_{\mathtt{FID}_{13}}>_{\mathrm{F}} \\
]_{\mathtt{employeeChildren}}
\end{array}\right\}
$$

$\mathtt{appendData}(\mathtt{child}, \text{“, Esquire”})$;

$$
\left\{\begin{array}{l}
\text{“employee”}_{\mathtt{ID}_2}[\\
\quad <\text{“employeeId”}_{\mathtt{ID}_3}[<\text{“\#text”}_{\mathtt{ID}_4}\ \text{“EMP0001”}>_{\mathrm{F}}]_{\mathtt{FID}_3}>_{\mathrm{F}} \otimes_{\mathrm{F}} \\
\quad <\text{“name”}_{\mathtt{nameNode}}[<\text{“\#text”}_{\mathtt{child}}\ \text{“Margaret Martin, Esquire”}>_{\mathrm{F}}]_{\mathtt{FID}_5}>_{\mathrm{F}} \otimes_{\mathrm{F}} \\
\quad <\text{“position”}_{\mathtt{ID}_7}[<\text{“\#text”}_{\mathtt{ID}_8}\ \text{“Accountant”}>_{\mathrm{F}}]_{\mathtt{FID}_7}>_{\mathrm{F}} \otimes_{\mathrm{F}} \\
\quad <\text{“salary”}_{\mathtt{ID}_9}[<\text{“\#text”}_{\mathtt{ID}_{10}}\ \text{“56,000”}>_{\mathrm{F}}]_{\mathtt{FID}_9}>_{\mathrm{F}} \otimes_{\mathrm{F}} \\
\quad <\text{“gender”}_{\mathtt{ID}_{11}}[<\text{“\#text”}_{\mathtt{ID}_{12}}\ \text{“Female”}>_{\mathrm{F}}]_{\mathtt{FID}_{11}}>_{\mathrm{F}} \otimes_{\mathrm{F}} \\
\quad <\text{“address”}_{\mathtt{ID}_{13}}[\\
\quad\quad <\text{“\#text”}_{\mathtt{ID}_{14}}\ \text{“1230 North Ave. Dallas, Texas 98551”}>_{\mathrm{F}} \\
\quad]_{\mathtt{FID}_{13}}>_{\mathrm{F}} \\
]_{\mathtt{employeeChildren}}
\end{array}\right\}
$$

$\mathtt{childData} := \mathtt{getData}(\mathtt{child})$

$$
\left\{\begin{array}{l}
\text{“employee”}_{\mathtt{ID}_2}[\\
\quad <\text{“employeeId”}_{\mathtt{ID}_3}[<\text{“\#text”}_{\mathtt{ID}_4}\ \text{“EMP0001”}>_{\mathrm{F}}]_{\mathtt{FID}_3}>_{\mathrm{F}} \otimes_{\mathrm{F}} \\
\quad <\text{“name”}_{\mathtt{nameNode}}[<\text{“\#text”}_{\mathtt{child}}\ \text{“Margaret Martin, Esquire”}>_{\mathrm{F}}]_{\mathtt{FID}_5}>_{\mathrm{F}} \otimes_{\mathrm{F}} \\
\quad <\text{“position”}_{\mathtt{ID}_7}[<\text{“\#text”}_{\mathtt{ID}_8}\ \text{“Accountant”}>_{\mathrm{F}}]_{\mathtt{FID}_7}>_{\mathrm{F}} \otimes_{\mathrm{F}} \\
\quad <\text{“salary”}_{\mathtt{ID}_9}[<\text{“\#text”}_{\mathtt{ID}_{10}}\ \text{“56,000”}>_{\mathrm{F}}]_{\mathtt{FID}_9}>_{\mathrm{F}} \otimes_{\mathrm{F}} \\
\quad <\text{“gender”}_{\mathtt{ID}_{11}}[<\text{“\#text”}_{\mathtt{ID}_{12}}\ \text{“Female”}>_{\mathrm{F}}]_{\mathtt{FID}_{11}}>_{\mathrm{F}} \otimes_{\mathrm{F}} \\
\quad <\text{“address”}_{\mathtt{ID}_{13}}[\\
\quad\quad <\text{“\#text”}_{\mathtt{ID}_{14}}\ \text{“1230 North Ave. Dallas, Texas 98551”}>_{\mathrm{F}} \\
\quad]_{\mathtt{FID}_{13}}>_{\mathrm{F}} \\
]_{\mathtt{employeeChildren}} \\
\wedge\ \mathtt{childData} \doteq \text{“Margaret Martin, Esquire”}
\end{array}\right\}
$$

$\mathtt{endloc}$

$\{\mathtt{childData} \doteq \text{“Margaret Martin, Esquire”} \wedge \mathrm{true}_{\mathrm{ELE}}\}$

Figure 5.14.: Proof of the “characterdataAppendDataGetData” Test

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
<xs:element name="addressBook">
  <xs:element name="household" minOccurs="0"
              maxOccurs="unbounded">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="person" maxOccurs="unbounded">
          <xs:complexType>
            <element name="name" type="string"/>
          </xs:complexType>
        </xs:element>
        <xs:element name="address" type="string"/>
        <xs:element name="phone" type="string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:element>
```

Figure 5.15.: Addressbook Schema

We can specify this XML schema using the grove formula S:

$$S \triangleq <\texttt{addressBook}[\text{households}]>_{\text{G}} \text{ where}$$

$$\begin{aligned}
&\text{households} \triangleq \neg\Diamond_{\otimes}(\text{true}_{\text{TXT}}) \wedge \Box_{\otimes}(\\
&\quad \texttt{household}[\\
&\qquad <\texttt{person}[\text{txts}]>_{\text{F}} \otimes_{\text{F}} \text{people} \otimes_{\text{F}} \\
&\qquad <\texttt{address}[\text{txts}]>_{\text{F}} \otimes_{\text{F}} <\texttt{phone}[\text{txts}]>_{\text{F}} \\
&\quad] \\
&) \\
&\\
&\text{txts} \triangleq <\text{“\#text”}>_{\text{F}} \otimes_{\text{F}} (\neg\Diamond_{\otimes}(\text{true}_{\text{ELE}})\Box_{\otimes}(\text{“\#text”})) \\
&\text{people} \triangleq \neg\Diamond(\text{true}_{\text{TXT}}) \wedge \Box_{\otimes}(\texttt{person}[\text{txts}])
\end{aligned}$$

Now consider a Featherweight DOM program which updates the addressBook document when a specified person `leaver` moves house.

The program moves `leaver` out of its current `household`, into a newly created house with a user-supplied address and phone number. The program checks if the original `household` is now empty and, if it is, deletes it.

```
moveHouse(leaver, newAddress, newPhone) ≜
    local house, book, newHouse, addr, adtxt, phn, kids, firstChild, firstName :
        // Move leaver into a new house.
        house := getParentNode(leaver);
        book := getParentNode(house);
        newHouse := createElement("household");
        appendChild(newHouse, leaver);
        addr := createElement("address");
        adtxt := createTextNode(newAddress);
        appendChild(addr, adtxt);
        appendChild(newHouse, addr);
        phn := createElement("phone");
        phntxt := createTextNode(newPhone);
        appendChild(phn, phntxt);
        appendChild(newHouse, phn);
        appendChild(book, newHouse);

        // Remove old house if empty.
        kids := getChildNodes(house);
        firstChild := item(kids, 0);
        firstName := getNodeName(firstChild);
        if firstName = "person" then
            skip
        else
            removeChild(book, house);
        fi
    endloc
```

The safety condition for `moveHouse` is that `leaver` refers to a `person` node and `newAddress` and `newPhone` are String variables. This can be simply expressed by the formula

$$P \triangleq \Diamond_{\mathrm{T}\rightarrow\mathrm{G}} \text{“person”}_{\texttt{leaver}}[\mathrm{true}_{\mathrm{F}}] \wedge \texttt{newAddress} \in \mathrm{S} \wedge \texttt{newPhone} \in \mathrm{S}$$

We can prove that `moveHouse` maintains the schema formula S provided that this safety formula P also holds:

$$\{S \wedge P\}\texttt{moveHouse(leaver, newAddress, newPhone)}\{S \oplus \mathrm{true}_{\mathrm{G}}\}$$

As discussed in Chapter 3.2, we treat Featherweight DOM as a garbage collected language. We thus have true_G in the postcondition to refer to uncollected garbage generated by the program, which is safely ignored. The proof is in Figure 5.16.

This example is particularly enticing since each step in the reasoning is relatively mechanical, and since the properties we wish to prove can be easily derived from assertions in schema languages that are already in widespread use. A long term goal of this research must be to automate this process. It seems feasible that a tool could be produced after the fashion of the W3C HTML Validator[66]. Where the HTML validator can assert that a web page is valid HTML, the new tool could assert that no matter what the embedded javascript does with the web page, it will remain valid HTML.

$\{S \wedge P\}$

$$\left\{\begin{array}{l} <\text{“addressBook”}[\\ \quad \text{households} \otimes_F <\text{“household”}[\\ \quad\quad \text{people} \otimes_F <\text{“person”}_{\texttt{leaver}}[\text{txts}]>_F \otimes_F \text{people} \otimes_F \\ \quad\quad <\text{“address”}[\text{txts}]>_F \otimes_F <\text{“phone”}[\text{txts}]>_F \\ \quad]>_F \otimes_F \text{households} \\ ]>_G \end{array}\right\}$$

`moveHouse(leaver, newAddress, newPhone)` $\triangleq$

```
local house, book, newHouse, addr, adtxt, phn, kids, firstChild, firstName :
    // Move leaver into a new house.
    house := getParentNode(leaver); book := getParentNode(house);
    newHouse := createElement(“household”); appendChild(newHouse, leaver);
    addr := createElement(“address”); adtxt := createTextNode(newAddress);
    appendChild(addr, adtxt); appendChild(newHouse, addr);
    phn := createElement(“phone”); phntxt := createTextNode(newPhone);
    appendChild(phn, phntxt); appendChild(newHouse, phn);
```

$$\left\{\begin{array}{l} <\text{“addressBook”}_{\texttt{book}}[\\ \quad \text{households} \otimes_F <\text{“household”}_{\texttt{house}}[\\ \quad\quad \text{people} \otimes_F <\text{“address”}[\text{txts}]>_F \otimes <\text{“phone”}[\text{txts}]>_F \\ \quad]>_F \otimes_F \text{households} \\ ]>_G \\ \oplus <\text{“household”}_{\texttt{newHouse}}[\\ \quad <\text{“person”}_{\texttt{leaver}}[\text{txts}]>_F \otimes_F \\ \quad <\text{“address”}[\texttt{newAddress}]>_F \otimes_F \\ \quad <\text{“phone”}[\texttt{newPhone}]>_F \\ ]>_G \end{array}\right\}$$

```
    appendChild(book, newHouse);
```

$$\left\{\begin{array}{l} <\text{“addressBook”}_{\texttt{book}}[\\ \quad \text{households} \otimes_F <\text{“household”}_{\texttt{house}}[\\ \quad\quad \text{people} \otimes_F <\text{“address”}[\text{txts}]>_F \otimes_F <\text{“phone”}[\text{txts}]>_F] \\ \quad]>_F \otimes_F \text{households} \\ >_G \end{array}\right\}$$

```
    // Remove old house if empty.
    kids := getChildNodes(house);
    firstChild := item(kids, 0);
    firstName := getNodeName(firstChild);
    if firstName = “person” then
        skip
```

$$\left\{\begin{array}{l} <\text{“addressBook”}[\\ \quad \text{households} \otimes_F <\text{“household”}[\\ \quad\quad <\text{“person”}[\text{txts}]>_F \otimes_F \\ \quad\quad \text{people} \otimes_F \\ \quad\quad <\text{“address”}[\text{txts}]>_F \otimes_F \\ \quad\quad <\text{“phone”}[\text{txts}]>_F \\ \quad]>_F \otimes_F \text{households} \end{array} \right. \left. \begin{array}{l} \\ \\ \\ \\ \\ \\ ]>_G \end{array}\right\}$$

```
    else
        removeChild(book, house);
        {<“addressBook”[households]>G ⊕ trueG}
    fi
endloc
```

$\{S \oplus \text{true}_G\}$

Figure 5.16.: Schema Preservation Derivation

6. DOM Core Level 1, The Fundamental Interfaces

This chapter presents a formal specification of the Fundamental Interfaces portion of DOM Core Level 1. This specification consists of an abstract data structure for representing DOM data, and an operational semantics of the commands DOM provides over that structure.

DOM Core Level 1 is the smallest subset of DOM which may be implemented by a compliant browser. Having shown the intuitions behind the conceptual core of DOM with Featherweight DOM, DOM Core Level 1 provides a natural next step to show that Featherweight-DOM-style reasoning scales well to substantial industrial specifications.

DOM Core Level 1 is divided into two parts. By far the larger part is the "Fundamental Interfaces", which describe all the central DOM concepts and commands necessary to manipulate HTML in a simple web browser:

Begin Quote

> The interfaces within this section are considered fundamental, and must be fully implemented by all conforming implementations of the DOM, including all HTML DOM implementations.

End Quote

A much smaller portion is termed the "Extended Interfaces", which are designed to allow the programmer to distinguish between various different XML dialects, and perform various extra functions that are not related to the manipulation of HTML:

Begin Quote

> objects that expose these interfaces will never be encountered in a DOM implementation that deals only with HTML. As such,

HTML-only DOM implementations do not need to have objects that implement these interfaces.

End Quote

This thesis is concerned with formalising and reasoning about a complete HTML-only DOM specification, consisting of all the functionality described in the Fundamental Interfaces of DOM Core Level 1. This will provide a solid base from which to pursue future work: reasoning about JavaScript, larger fragments of DOM and other web programming libraries.

6.1. Notation

In the following section, we will introduce a large number of notations, many of which follow a common pattern. For ease of understanding, and for mnemonic value, we present the pattern informally here. What DOM refers to as a "Node", will generally look like this:

$$name_{\mathbf{id}} \{\!|\, ats \,|\!\}_{\mathbf{aid}}^{\substack{\mathbf{idref} \\ \mathbf{typenum}}} [kids]_{\mathbf{fid}} value$$

We will formally define several sorts of node which follow this pattern. All these nodes will have a *name*, a potentially null list of attributes $\{\!|\, ats \,|\!\}_{\mathbf{aid}}$ and a potentially empty list of children $[kids]_{\mathbf{fid}}$. The node as a whole may be referenced by its **id**, and its child-list may be referenced by its **fid**. If a node has an attribute list, that list may be referenced by its **aid**. A node also has a *value*, which is a string (and may contain, for example, the text in a text node); a **typenum** which is a natural number ; and an **idref** which references the **id** of another node – the document which "owns" this node.

We will also introduce a number of specific notations for when particular node types don't perfectly fit this general pattern. These will be explained as they appear.

6.2. Data Structure

In order to reason about DOM, we introduce an abstract data structure to represent the XML-like data that DOM programs manipulate. As with Featherweight DOM, this structure represents parsed XML which is to be

manipulated in-memory by DOM. This data structure is given in Figure 6.1. As with Featherweight DOM, we also introduce a natural "Context" structure in Figure 6.2. Since these data structures are significantly larger than those in Featherweight DOM, we also show the relationships between data and context structures in Figure 6.3. Note in particular that some data structures have no analogous context. As a notational convenience, we will use the shorthand "abc" to refer to the string $\langle\text{`a'}\rangle_S \otimes_S \langle\text{`b'}\rangle_S \otimes_S \langle\text{`c'}\rangle_S$.

Definition 35 (Data Structure)**.** Given an infinite set ID of node identifiers and a finite set C of characters, with distinguished characters '#' and '*', we define the DOM Data Structure as in Figure 6.1. For well-formedness we also require:

- $\mathbf{id}, \mathbf{fid}, \mathbf{aid} \in \text{ID}$ must be unique across the whole data structure;
- the **idref** $\in$ ID in a node must be equal to the **id** of an existing Document node elsewhere in the data structure, which we call that node's "owner document";
- if the parent of a node is a Document node, then the owner document of that node must be that Document node;
- no node may be owned by a different Document than its parent;
- the **idref** in an element search must refer to the **id** of an element or a document node, which we call that search's "search root".

Finally we have a simple structural congruence, denoted $\equiv$, which states that:

- for any type D, D-composition $\otimes_D$ is associative with unit $\varnothing_D$;
- grove-composition $\oplus$ is associative and commutative with unit $\varnothing_G$.

In Chapter 6.3.2, we define a language for manipulating this data structure. Every command in that language preserves these requirements.

In the following subsections, we explain some of decisions apparent in the data structure described above in Definition 35 and Figure 6.1. We believe that through careful choices in the data structure we can make the precise definition of the behaviour of DOM commands significantly simpler. For this reason we will often motivate choices made here with reference

groves $\mathbf{g} \in \mathrm{G}$	$\mathbf{g}$ ::=	$<\mathbf{doc}>_{\mathrm{G}} \mid <\mathbf{ele}>_{\mathrm{G}} \mid <\mathbf{frag}>_{\mathrm{G}} \mid <\mathbf{attr}>_{\mathrm{G}} \mid <\mathbf{txt}>_{\mathrm{G}} \mid$ $<\mathbf{es}>_{\mathrm{G}} \mid <\mathbf{comm}>_{\mathrm{G}} \mid \varnothing_{\mathrm{G}} \mid \mathbf{g} \oplus \mathbf{g}$
documents $\mathbf{doc} \in \mathrm{DOC}$	$\mathbf{doc}$ ::=	$\text{“\#document”}_{\mathbf{id}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{null}_9}^{\mathbf{null}} [\mathbf{df}]_{\mathbf{fid}} \mathbf{null}$
document forest $\mathbf{df} \in \mathrm{DF}$	$\mathbf{df}$ ::=	$<\mathbf{dnel}, \mathbf{de}, \mathbf{dnel}>_{\mathrm{DF}}$ where the second $\mathbf{dnel}$ is $\varnothing_{\mathrm{DNEL}}$ if $\mathbf{de}$ is $\varnothing_{\mathrm{DE}}$
non-element df $\mathbf{dnel} \in \mathrm{DNEL}$	$\mathbf{dnel}$ ::=	$<\mathbf{comm}>_{\mathrm{DNEL}} \mid \varnothing_{\mathrm{DNEL}} \mid \mathbf{dnel} \otimes_{\mathrm{DNEL}} \mathbf{dnel}$
document element $\mathbf{de} \in \mathrm{DE}$	$\mathbf{de}$ ::=	$<\mathbf{ele}>_{\mathrm{DE}} \mid \varnothing_{\mathrm{DE}}$
document fragments $\mathbf{frag} \in \mathrm{FRAG}$	$\mathbf{frag}$ ::=	$\text{“\#document-fragment”}_{\mathbf{id}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{null}_{11}}^{\mathbf{idref}} [\mathbf{fragf}]_{\mathbf{fid}} \mathbf{null}$
document fragment forests $\mathbf{fragf} \in \mathrm{FRAGF}$	$\mathbf{fragf}$ ::=	$<\mathbf{ele}>_{\mathrm{FRAGF}} \mid <\mathbf{txt}>_{\mathrm{FRAGF}} \mid <\mathbf{comm}>_{\mathrm{FRAGF}} \mid \varnothing_{\mathrm{FRAGF}}$ $\mid \mathbf{fragf} \otimes_{\mathrm{FRAGF}} \mathbf{fragf}$
elements $\mathbf{ele} \in \mathrm{ELE}$	$\mathbf{ele}$ ::=	$\mathbf{s}_{\mathbf{id}} \{\!\!\{ \mathbf{ea} \}\!\!\}_{\mathbf{aid}_1}^{\mathbf{idref}} [\mathbf{ef}]_{\mathbf{fid}} \mathbf{null}$ where ‘#’, ‘*’ $\notin \mathbf{s}$
element attributes $\mathbf{ea} \in \mathrm{EA}$	$\mathbf{ea}$ ::=	$<\mathbf{attr}>_{\mathrm{EA}} \mid \varnothing_{\mathrm{EA}} \mid \mathbf{ea} \otimes_{\mathrm{EA}} \mathbf{ea}$ where the name string $\mathbf{s}$ of each $\mathbf{attr}$ is sibling unique
element forests $\mathbf{ef} \in \mathrm{EF}$	$\mathbf{ef}$ ::=	$<\mathbf{ele}>_{\mathrm{EF}} \mid <\mathbf{txt}>_{\mathrm{EF}} \mid <\mathbf{comm}>_{\mathrm{EF}} \mid \varnothing_{\mathrm{EF}} \mid \mathbf{ef} \otimes_{\mathrm{EF}} \mathbf{ef}$
attributes $\mathbf{attr} \in \mathrm{ATTR}$	$\mathbf{attr}$ ::=	$\ll \mathbf{s}_{\mathbf{id}} \mapsto [\mathbf{af}]_{\mathbf{fid}} \gg_{\mathbf{bool}}^{\mathbf{idref}}$ where ‘#’,‘*’ $\notin \mathbf{s}$
attribute forests $\mathbf{af} \in \mathrm{AF}$	$\mathbf{af}$ ::=	$<\mathbf{txt}>_{\mathrm{AF}} \mid \varnothing_{\mathrm{AF}} \mid \mathbf{af} \otimes_{\mathrm{AF}} \mathbf{af}$
comments $\mathbf{comm} \in \mathrm{COMM}$	$\mathbf{comm}$ ::=	$\text{“\#comment”}_{\mathbf{id}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{null}_8}^{\mathbf{idref}} [\mathbf{cf}]_{\mathbf{fid}} \mathbf{s}$
comment forests $\mathbf{cf} \in \mathrm{CF}$	$\mathbf{cf}$ ::=	$\varnothing_{\mathrm{CF}}$
text $\mathbf{txt} \in \mathrm{TXT}$	$\mathbf{txt}$ ::=	$\text{“\#text”}_{\mathbf{id}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{null}_3}^{\mathbf{idref}} [\mathbf{tf}]_{\mathbf{fid}} \mathbf{s}$
text forests $\mathbf{tf} \in \mathrm{TF}$	$\mathbf{tf}$ ::=	$\varnothing_{\mathrm{TF}}$
element searches $\mathbf{es} \in \mathrm{ES}$	$\mathbf{es}$ ::=	$\mathbf{s}_{\mathbf{fid}}^{\mathbf{idref}}$
strings $\mathbf{s} \in \mathrm{S}$	$\mathbf{s}$ ::=	$<\mathbf{c}>_{\mathrm{S}} \mid \varnothing_{\mathrm{S}} \mid \mathbf{s} \otimes_{\mathrm{S}} \mathbf{s}$ where $\mathbf{c} \in \mathrm{C}$
booleans $\mathbf{bool} \in \mathrm{BOOL}$	$\mathbf{bool}$ ::=	true | false

where $\mathbf{id}, \mathbf{fid}, \mathbf{idref}, \mathbf{aid} \in \mathrm{ID}$, $\mathbf{id}, \mathbf{fid}, \mathbf{aid}$ must be unique ; the $\mathbf{idref} \in \mathrm{ID}$ in a node must be equal to the $\mathbf{id}$ of an existing Document node elsewhere in the data structure ; if the parent of a node is a Document node, then the owner document of that node must be that Document node ; no node may be owned by a different Document than its parent ; the $\mathbf{idref}$ in an element search must refer to the $\mathbf{id}$ of an element or a document node.

Figure 6.1.: DOM Data Structure

$$
\begin{array}{rl}
\mathbf{cg} ::= & <\mathbf{cdoc}>_{\mathrm{G}} \mid <\mathbf{cele}>_{\mathrm{G}} \mid <\mathbf{cfrag}>_{\mathrm{G}} \mid <\mathbf{cattr}>_{\mathrm{G}} \mid <\mathbf{ctxt}>_{\mathrm{G}} \mid \\
& <\mathbf{ces}>_{\mathrm{G}} \mid <\mathbf{comm}>_{\mathrm{G}} \mid -_{\mathrm{G}} \mid \mathbf{cg} \oplus \mathbf{g} \\
\\
\mathbf{cdoc} ::= & \text{“\#document”}_{\mathbf{id}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{null}_9}^{\mathbf{null}} [\mathbf{cdf}]_{\mathbf{fid}} \mathbf{null} \mid -_{\mathrm{DOC}} \\
\\
\mathbf{cdf} ::= & <\mathbf{cdnel}, \mathbf{de}, \mathbf{dnel}>_{\mathrm{DF}} \mid <\mathbf{dnel}, \mathbf{cde}, \mathbf{dnel}>_{\mathrm{DF}} \\
& <\mathbf{dnel}, \mathbf{de}, \mathbf{cdnel}>_{\mathrm{DF}} \mid -_{\mathrm{DF}} \\
\\
\mathbf{cdnel} ::= & <\mathbf{ccomm}>_{\mathrm{DNEL}} \mid -_{\mathrm{DNEL}} \mid \mathbf{cdnel} \otimes_{\mathrm{DNEL}} \mathbf{dnel} \mid \mathbf{dnel} \otimes_{\mathrm{DNEL}} \mathbf{cdnel} \\
\\
\mathbf{cde} ::= & <\mathbf{cele}>_{\mathrm{DE}} \mid -_{\mathrm{DE}} \\
\\
\mathbf{cfrag} ::= & \text{“\#document-fragment”}_{\mathbf{id}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{null}_{11}}^{\mathbf{idref}} [\mathbf{cfragf}]_{\mathbf{fid}} \mathbf{null} \mid -_{\mathrm{FRAG}} \\
\\
\mathbf{cfragf} ::= & <\mathbf{cele}>_{\mathrm{FRAGF}} \mid <\mathbf{ctxt}>_{\mathrm{FRAGF}} \mid <\mathbf{ccomm}>_{\mathrm{FRAGF}} \mid -_{\mathrm{FRAGF}} \\
& \mid \mathbf{cfragf} \otimes_{\mathrm{FRAGF}} \mathbf{fragf} \mid \mathbf{fragf} \otimes_{\mathrm{FRAGF}} \mathbf{cfragf} \\
\\
\mathbf{cele} ::= & \mathbf{s}_{\mathbf{id}} \{\!\!\{ \mathbf{ea} \}\!\!\}_{\mathbf{aid}_1}^{\mathbf{idref}} [\mathbf{cef}]_{\mathbf{fid}} \mathbf{null} \mid \qquad \text{where ‘\#’, ‘*’} \notin \mathbf{s} \\
& \mathbf{s}_{\mathbf{id}} \{\!\!\{ \mathbf{cea} \}\!\!\}_{\mathbf{aid}_1}^{\mathbf{idref}} [\mathbf{ef}]_{\mathbf{fid}} \mathbf{null} \mid -_{\mathrm{ELE}} \\
\\
\mathbf{cea} ::= & <\mathbf{cattr}>_{\mathrm{EA}} \mid -_{\mathrm{EA}} \mid \mathbf{cea} \otimes_{\mathrm{EA}} \mathbf{ea} \mid \mathbf{ea} \otimes_{\mathrm{EA}} \mathbf{cea} \qquad \begin{array}{l}\text{where the name string } \mathbf{s} \text{ of} \\ \text{each } \mathbf{attr} \text{ is sibling unique}\end{array} \\
\\
\mathbf{cef} ::= & <\mathbf{cele}>_{\mathrm{EF}} \mid <\mathbf{ctxt}>_{\mathrm{EF}} \mid <\mathbf{ccomm}>_{\mathrm{EF}} \mid -_{\mathrm{EF}} \mid \mathbf{cef} \otimes_{\mathrm{EF}} \mathbf{ef} \mid \mathbf{ef} \otimes_{\mathrm{EF}} \mathbf{cef} \\
\\
\mathbf{cattr} ::= & \ll \mathbf{s}_{\mathbf{id}} \mapsto [\mathbf{caf}]_{\mathbf{fid}} \gg_{\mathbf{bool}}^{\mathbf{idref}} \mid -_{\mathrm{ATTR}} \qquad \text{where ‘\#’, ‘*’} \notin \mathbf{s} \\
\\
\mathbf{caf} ::= & <\mathbf{ctxt}>_{\mathrm{AF}} \mid -_{\mathrm{AF}} \mid \mathbf{caf} \otimes_{\mathrm{AF}} \mathbf{af} \mid \mathbf{af} \otimes_{\mathrm{AF}} \mathbf{caf} \\
\\
\mathbf{ccomm} ::= & \text{“\#comment”}_{\mathbf{id}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{null}_8}^{\mathbf{idref}} [\varnothing_{\mathrm{CF}}]_{\mathbf{fid}} \mathbf{cs} \mid -_{\mathrm{COMM}} \\
\\
\mathbf{ctxt} ::= & \text{“\#text”}_{\mathbf{id}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{null}_3}^{\mathbf{idref}} [\varnothing_{\mathrm{TF}}]_{\mathbf{fid}} \mathbf{cs} \mid -_{\mathrm{TXT}} \\
\\
\mathbf{cs} ::= & -_{\mathrm{S}} \mid \mathbf{cs} \otimes_{\mathrm{S}} \mathbf{s} \mid \mathbf{s} \otimes_{\mathrm{S}} \mathbf{cs}
\end{array}
$$

where $\mathbf{id}, \mathbf{fid}, \mathbf{idref}, \mathbf{aid} \in \mathrm{ID}$, $\mathbf{id}, \mathbf{fid}, \mathbf{aid}$ must be unique and $\mathbf{idref} \in \mathrm{ID}$ must be equal to the **id** of an existing Document node elsewhere in the data structure ; if the parent of a node is a Document node, then the owner document of that node must be that Document node ; no node may be owned by a different Document than its parent

Figure 6.2.: DOM Context Structure

	Data	Contexts
groves	**g** ∈ G	**cg** ∈ CG
documents	**doc** ∈ DOC	**cdoc** ∈ CDOC
document-forest	**df** ∈ DF	**cdf** ∈ CDF
non-element df	**dnel** ∈ DNEL	**cdnel** ∈ CDNEL
document-element	**de** ∈ DE	**cde** ∈ CDE
elements	**ele** ∈ ELE	**cele** ∈ CELE
element attributes	**ea** ∈ EA	**cea** ∈ CEA
element forests	**ef** ∈ EF	**cef** ∈ CEF
document fragments	**frag** ∈ FRAG	**cfrag** ∈ CFRAG
document fragment forests	**fragf** ∈ FRAGF	**cfragf** ∈ CFRAGF
attributes	**attr** ∈ ATTR	**cattr** ∈ CATTR
attribute forests	**af** ∈ AF	**caf** ∈ CAF
comments	**comm** ∈ COMM	**ccomm** ∈ CCOMM
comment forests	**cf** ∈ CF	
text	**txt** ∈ TXT	**ctxt** ∈ CTXT
text forests	**tf** ∈ TF	
strings	**s** ∈ S	**cs** ∈ CS
element searches	**es** ∈ ES	

Figure 6.3.: Data Structure Sets

to commands we define later in Chapter 6.3, and their less precise W3C specifications [23].

6.2.1. Node Types

DOM allows a programmer to distinguish between different types of node (elements, documents, etc) by means of the read-only object attribute "nodeType", which is represented in Chapter 6.3 of this document as the getter command "getNodeType". This object attribute is defined by DOM to return an integer, which communicates to the programmer the type of the node. The integer "1" means the node is an Element Node, "2" means an Attr and so on. When we refer to the "type number" of a node, we are referring to this integer, as given in Definition 36. When we refer to the "type" of a node, we are referring to the set of such nodes, as given in Figure 6.1.

Definition 36 (Type Numbers)**.** The type number of a node is an integer between 1 and 12.

When we wish to refer to a particular type number in a program, we will

use the constants defined by [23] in the Node interface. They are written as **ELEMENT_NODE**, **ATTRIBUTE_NODE** and so on.

The data structure given in Figure 6.1 may be divided into the disjoint sets on the left of Figure 6.3. We consider these sets to be the "types" of the data they contain, and for convenience we group these types into further sets.

Definition 37 (Node Type Groupings)**.** The types of the data defined in Figure 6.1 are grouped into the following sets:

$$\begin{aligned} \text{Nodes } \mathcal{N} &\triangleq \{\text{DOC}, \text{ELE}, \text{FRAG}, \text{ATTR}, \text{COMM}, \text{TXT}, \text{ES}\} \\ \text{Forests } \mathcal{F} &\triangleq \{\text{DF}, \text{DNEL}, \text{DE}, \text{EA}, \text{EF}, \text{FRAGF}, \text{AF}, \text{CF}, \text{TF}, \text{S}\} \\ \text{Data } \mathcal{D} &\triangleq \mathcal{N} \cup \mathcal{F} \cup \{\text{G}\} \end{aligned}$$

If we have a set D such that $\text{D} \in \mathcal{D}$ and a datum $\mathbf{d} \in \text{D}$, we may consider **d** to be of type D, written as **d**:D.

Note in particular that since our data structure allows nodes of a given type (for example, elements $\in$ ELE) to appear in more than one type of forest (for example, both fragment forests $\in$ FRAGF and also element forests $\in$ EF), we distinguish between singleton elements of different forest-types using an angle bracket notation:

$$<\mathit{structure}>_{\text{D}}$$

When we refer to the "type" of a structure (as opposed to the "type number"), it is this D to which we are referring.

6.2.2. The Grove

At the top level of the structure defined in 6.1 is the grove. As with Featherweight DOM, this is analogous to an object heap in object oriented programming, and may contain all types of node and list.

6.2.3. Document Nodes

The name of a document is always "#document". Document nodes have no attributes. In our notation the attribute list of a document node therefore has a **null aid**, and $\varnothing_{\text{EA}}$ contents. The value of a document node is always

null. The type number of a document node is 9. The owner document of a document node is **null**.

Documents have as children any number of comment nodes, and zero or one element nodes. We represent the pattern of child nodes using the "document-forest" structure:

$$<\mathbf{dnel}, \mathbf{de}, \mathbf{dnel}>_{\mathrm{DF}}$$

If the document has no element children, then the second **dnel** will be $\varnothing_{\mathrm{DNEL}}$. If the document has one element child, thenthe first **dnel** will contain all the comments that come before the element in the document order, while the second **dnel** will contain all the comments that come after the element in the document order. This property is preserved by all the commands in Chapter 6.3

6.2.4. Document Fragments

The name of a document fragment is always "#document-fragment". Document fragment nodes have no attributes. In our notation, the attribute list of a document fragment node has a **null aid** and $\varnothing_{\mathrm{EA}}$ contents. The value of a document fragment node is always **null**. Document fragments have 0 or more elements, text nodes or comments as children. The type number of a document fragment node is 11. All document fragment nodes have a non-**null** owner document.

6.2.5. Element Nodes

The name of an element node may be any string that doesn't contain "#", and corresponds to the "tag" of an element in an XML document. Element nodes have 0 or more attribute-children, and so the **aid** of the node is not **null**. Element nodes have 0 or more other element nodes, comment nodes or text nodes as children. The type number of an element node is 1. Every element node has a non-**null** owner document. The value of an element node is always **null**.

6.2.6. Attributes

While an XML attribute, or attr node in the DOM specification is a a subclass of Node, the specification also says:

Begin Quote

> Attr objects inherit the Node interface, but since they are not actually child nodes of the element they describe, the DOM does not consider them part of the document tree ... The DOM takes the view that attributes are properties of elements rather than having a separate identity from the elements they are associated with; ... In short, users and implementors of the DOM need to be aware that Attr nodes have some things in common with other objects inheriting the Node interface, but they also are quite distinct.

End Quote

An attr node behaves sufficiently differently from other nodes that in this formalisation, we give it a unique syntax. If a node informally looks like:

$$name_{\mathbf{id}} \{\!| \; ats \; |\!\}_{\mathbf{aid}}^{idref}{}_{type} [kids]_{\mathbf{fid}} value$$

Then an analogous informal attr looks like:

$$\ll name_{\mathbf{id}} \mapsto [kids]_{\mathbf{fid}} \gg_{\mathbf{specified}}^{\mathbf{idref}}$$

The type number of an attr is 2, but since attr nodes have unique syntax, this number does not need to be literally recorded in that syntax. Commands which use the type number data of their arguments (such as the getNodeType command defined in Chapter 6.3) will assume a type number of 2 when passed any argument that looks like an attr.

The name of an attr may be an arbitrary string, but that string must uniquely identify the attr among its siblings in the attribute forest of its parent element node. Attr nodes only have text nodes as children. Attr nodes have no attributes of their own. All attributes have a non-**null** owner document.

Attr nodes have a property that other nodes do not. In the informal presentation above, it is named "specified" while in Figure 6.1 it is a boolean. This property corresponds to the object attribute called "specified" in [23].In [23], the behaviour associated with the "specified" property is complex and potentially confusing. We deal with it in full in Chapter 6.3.2. For now, it is enough to be aware that attr nodes carry a boolean payload we call "specified".

Finally, [23] specifies that attr nodes should have a "value", such as might be returned by a call to the "getNodeValue" getter command specified in Chapter 6.3:

Begin Quote

> On retrieval, the value of the attribute is returned as a string.
> ...
> On setting, this creates a Text node with the unparsed contents of the string.

End Quote

The reason setting the value of an attr node creates a new Text node is that the value of an XML attribute as represented in DOM may be viewed as the "value" property of an attr object, or as the concatenation of the "value" properties of its child Text nodes. This enables a programmer to set the value of an XML attribute to a literal string with a single assignment to the "value" property of the object; or to edit the value of an attribute to append some text from elsewhere in the document with a single call to appendChild. These two views of the value of an xml attribute must be kept consistent at all times.

In this formalisation of DOM, the canonical form of the value of an XML attibute is the child list of the attr and the text nodes in that list. When a call is made to the getter command "getNodeValue" defined in Chapter 6.3, that command constructs a literal value on the fly by concatenating the values of the text nodes in the attr's child list. When a call is made to the setter command "setNodeValue", the command creates a new text node as per the specification quoted above. That text node is added to the child list of the attr, and all previous children are removed.

6.2.7. Comments and Text Nodes

The names of comments and text nodes are "#comment" and "#text" respectively, and their type numbers are 8 and 3. Neither node type has attributes, and so always has an **aid** equal to **null**. Both node types must always have non-**null** owner documents, and neither may have children.

Despite not having children, both comments and text nodes always have non-**null fid**s. While this is not consistent with the **nil aid** values, both decisions are made explicit in the documentation for the "Node" interface in [23]:

Begin Quote

	nodeName	nodeValue	attributes
	...		
Text	#text	content of the text node	null
	...		
Comment	#comment	content of the comment	null
	...		

childNodes A NodeList that contains all children of this node. If there are no children, this is a NodeList containing no nodes.

End Quote

The values of comment and text nodes, as suggested by this quote, represent the content of those nodes. They are strings.

6.2.8. Element Searches

All but one of the substructures presented in Figure 6.1 directly correspond to structures explicitly described in [23]. The "element search" structure however, is motivated by the following interaction of two requirements of the W3C specification.

[23] specifies a method both on the Document and the Element interfaces which is called "getElementsByTagName". The method is first described in the Document interface:

Begin Quote

> **getElementsByTagName** Returns a NodeList of all the Elements with a given tag name in the order in which they would be encountered in a preorder traversal of the Document tree.

End Quote

The description on the Element interface differs only in its substitution of "Element Tree" in place of "Document Tree".

[23] also says:

Begin Quote

> NodeLists and NamedNodeMaps in the DOM are "live", that is, changes to the underlying document structure are reflected in all relevant NodeLists and NamedNodeMaps. For example, if a DOM user gets a NodeList object containing the children of an Element, then subsequently adds more children to that element (or removes children, or modifies them), those changes are automatically reflected in the NodeList without further action on the user's part. Likewise changes to a Node in the tree are reflected in all references to that Node in NodeLists and NamedNodeMaps.

End Quote

Taken together, these two clauses mandate the following potential behaviour:

Suppose we have an element "e", with no children. If we call "l = e.getElementsByTagName('foo')" then l will be assigned an empty NodeList. If we call "l.length", the return value will be 0.

If we then use "appendChild" to add an element with name "foo" to the childlist of e, then the NodeList l will automatically be updated to contain that new element. The return value of "l.length" will now be 1.

It is tempting to assert that this is not the desired behaviour of the specification – that surely the "live update" policy quoted above only applies to NodeLists which are actually part of the tree structure of a document.

Surely it should not apply to NodeLists which contain the results of a query which has been applied to that structure. This view of the specification is opposed by the following quote, taken from the section of the specification of the Node interface that deals with the Node's children:

Begin Quote

> **childNodes** A NodeList that contains all children of this node. If there are no children, this is a NodeList containing no nodes. The content of the returned NodeList is "live" in the sense that, for instance, changes to the children of the node object that it was created from are immediately reflected in the nodes returned by the NodeList accessors; it is not a static snapshot of the content of the node. This is true for every NodeList, including the ones returned by the getElementsByTagName method.

End Quote

In order to allow for this required behaviour, the command "getElementsByTagName" as defined in Chapter 6.3 does not return a static NodeList. Instead it does not perform a tree search at all, but returns an element search structure which records the parameters of the search:

$$\mathbf{s}^{\mathbf{idref}}_{\mathbf{fid}}$$

The string **s** is the search term – either the tag name being searched for, or the special character "*" which is a wildcard. The **idref** is a reference to the root of the search. This is either an element or a document node. Finally the **fid** is analogous to the **fid** of any other structure which has children.

An element search may be treated as a node list, in which case any attempt to access the data in it will result in an on-the-fly search of the current state of the tree. For example, calling "getLength()" will result in the entire subtree of the search root being searched for element nodes with names matching the search string **s**. The results will be counted, and that count will be returned.

By returning an element search structure rather than a node list structure, we are able to provide the "live update" behaviour required by [23].

6.2.9. Strings and Characters

[23] has this to say on the subject of strings:

Begin Quote

> To ensure interoperability, the DOM specifies the DOMString type as follows:
>
> - A DOMString is a sequence of 16-bit quantities. This may be expressed in IDL terms as:
>
>   ```
>   typedef sequence<unsigned short> DOMString;
>   ```
>
> - Applications must encode DOMString using UTF-16 (defined in Appendix C.3 of [UNICODE] and Amendment 1 of [ISO-10646]).The UTF-16 encoding was chosen because of its widespread industry practice. Please note that for both HTML and XML, the document character set (and therefore the notation of numeric character references) is based on UCS-4. A single numeric character reference in a source document may therefore in some cases correspond to two array positions in a DOMString (a high surrogate and a low surrogate). **Note:** *Even though the DOM defines the name of the string type to be DOMString, bindings may used different names. For, example for Java, DOMString is bound to the String type because it also uses UTF-16 as its encoding.*

End Quote

Since this formalisation is concerned with behaviour rather than bit-representation, it is effectively character-set-independent, requiring only that literal characters in the programming language have *some* representation in memory. In Figure 6.1, we simply specify that strings are sequences of characters. We do not explicitly ennumerate all the allowable UTF-16 characters.

6.2.10. Contexts

Just as in Featherweight DOM, we give a context structure which mirrors the DOM data structure.

Definition 38 (DOM Context Structure)**.** Given an infinite set ID of node identifiers and a finite set C of characters, with distinguished characters '#' and '*', we define the DOM context structure as in Figure 6.2, following the pattern of the DOM data structure given in Figure 6.1. For well formedness, the structure of contexts follows the requirements of DOM data as outlined in Definition 35: $\mathbf{id}, \mathbf{fid}, \mathbf{aid} \in \mathrm{ID}$ must be unique across the whole data structure and $\mathbf{idref} \in \mathrm{ID}$ must be equal to an existing **id** elsewhere in the data structure. An **idref** in a node must refer to the **id** of a document node, which we call that node's "owner document". Any node which is a child of a document must be owned by that document, and no other node may be owned by a different document to its parent. Document nodes have no owner document, and so their **idref**s are set to **null**.

Finally, we have a simple structural congruence, denoted $\equiv$, which states that:

- for any type D, D and D-context composition $\otimes_{\mathrm{D}}$ is associative with unit $\varnothing_{\mathrm{T}}$;
- grove-composition $\oplus$ is associative and commutative with unit $\varnothing_{\mathrm{G}}$.

Note that each hole is annotated with a type (see Chapter 6.2.1), which corresponds to the type of data that can be inserted into that hole using the application function ap (defined in Definition 41).

Just as the data structures in 6.1 can be divided into the disjoint sets on the left of Figure 6.3, so may the context structures in 6.2 be divided into the disjoint sets on the right of Figure 6.3. For contexts however, we wish to differentiate between types of a finer granularity.

Definition 39 (Context Types)**.** Recall the type grouping $\mathcal{D}$ given in Definition 37. Given a context **cd** and node types $\mathrm{D}_1, \mathrm{D}_2 \in \mathcal{D}$, the context **cd** has type $\mathrm{D}_1 \to \mathrm{D}_2$ (written $\mathbf{cd}{:}(\mathrm{D}_1 \to \mathrm{D}_2)$ iff:

- $\mathbf{cd} \in \mathrm{CD}_2$, where CD_2 is the context set corresponding to D_2 in Figure 6.3;

- **cd** contains a hole $-_{\mathrm{D}_1}$.

The analogy between context types and function types is intentional. In Section 6.2.11 we will define the partial application function "ap" to preserve the types $((\mathrm{D}_1{\to}\mathrm{D}_2) \times \mathrm{D}_1) \rightharpoonup \mathrm{D}_2$.

Recall the type groupings given in Definition 37. Sometimes we will wish to refer generically to structure which may be either a datum or a context. In these cases we will often use the type $\mathrm{A} \in \mathcal{A}$:

Definition 40 (The Biggest Type Group)**.**

$$\text{Data and Contexts } \mathcal{A} \triangleq \mathcal{D} \cup \bigcup_{\mathrm{D}_1,\mathrm{D}_2 \in \mathcal{D}} \{\mathrm{D}_1{\to}\mathrm{D}_2\}$$

6.2.11. Context Application

Context application is defined using an application function $\mathrm{ap}{:}((\mathrm{D}_1{\to}\mathrm{D}_2) \times \mathrm{D}_1) \rightharpoonup \mathrm{D}_2$. Since the application function behaves the same way with every kind of node, we use a general notation to specify the application function on several node-types at once. Similarly with D-composition $\otimes_{\mathrm{D}}$.

Definition 41 (Context Application)**.** Given data types $\mathrm{D}_1, \mathrm{D}_2 \in \mathcal{D}$, the partial application function $\mathrm{ap}{:}((\mathrm{D}_1{\to}\mathrm{D}_2) \times \mathrm{D}_1) \rightharpoonup \mathrm{D}_2$ is defined as in Figure 6.4.

$\mathrm{ap}(\mathbf{cd}, \mathbf{d})\downarrow$ denotes that $\mathrm{ap}(\mathbf{cd}, \mathbf{d})$ is defined.

The partial application function inserts a data structure into a context hole of a matching type. The function is partial, since various rules about the resulting data structure must be preserved. It is not permissible to insert a node with a particular ID into a hole in a structure which already contains a node of that ID for example. This function is essential for defining our local reasoning in Chapter 7, but also turns out to be very useful for defining the operational semantics of our language in Section 6.3.

6.3. The Language

We present DOM Core Level One as an abstract data structure paired with a language which is used to manipulate that data structure. The language follows the language of Featherweight DOM presented in Part 3.2. It contains

$$
\begin{array}{rcl}
\mathrm{ap}(-_{\mathrm{D}_1}, \mathbf{d}_1) & \triangleq & \mathbf{d}_1 \\
\mathrm{ap}(\mathbf{name_{id}} \{\!\!\{ \mathbf{ea} \}\!\!\}^{\mathbf{irn}}_{\mathbf{aidn_{tp}}} [\mathbf{con}]_{\mathbf{fid}} \mathbf{val}, \mathbf{d}_1) & \triangleq & \mathbf{name_{id}} \{\!\!\{ \mathbf{ea} \}\!\!\}^{\mathbf{irn}}_{\mathbf{aidn_{tp}}} [\mathrm{ap}(\mathbf{con}, \mathbf{d}_1)]_{\mathbf{fid}} \mathbf{val} \\
\mathrm{ap}(\mathbf{s_{id}} \{\!\!\{ \mathbf{con} \}\!\!\}^{\mathbf{idref}}_{\mathbf{aid_1}} [\mathbf{ef}]_{\mathbf{fid}} \mathbf{null}, \mathbf{d}_1) & \triangleq & \mathbf{s_{id}} \{\!\!\{ \mathrm{ap}(\mathbf{con}, \mathbf{d}_1) \}\!\!\}^{\mathbf{idref}}_{\mathbf{aid_1}} [\mathbf{ef}]_{\mathbf{fid}} \mathbf{null} \\
\mathrm{ap}(<\mathbf{con}>_{\mathrm{D}_2}, \mathbf{d}_1) & \triangleq & <\mathrm{ap}(\mathbf{con}, \mathbf{d}_1)>_{\mathrm{D}_2} \\
\mathrm{ap}(\mathbf{con} \oplus \mathbf{d}_2, \mathbf{d}_1) & \triangleq & \mathrm{ap}(\mathbf{con}, \mathbf{d}_1) \oplus \mathbf{d}_2 \\
\mathrm{ap}(\mathbf{con} \otimes_{\mathrm{D}_2} \mathbf{d}_2, \mathbf{d}_1) & \triangleq & \mathrm{ap}(\mathbf{con}, \mathbf{d}_1) \otimes_{\mathrm{D}_2} \mathbf{d}_2 \\
\mathrm{ap}(\mathbf{d}_2 \otimes_{\mathrm{D}_2} \mathbf{con}, \mathbf{d}_1) & \triangleq & \mathbf{d}_2 \otimes_{\mathrm{D}_2} \mathrm{ap}(\mathbf{con}, \mathbf{d}_1) \\
\mathrm{ap}(\ll \mathbf{s_{id}} \mapsto [\mathbf{con}]_{\mathbf{fid}} \gg^{\mathbf{idref}}_{\mathbf{bool}}, \mathbf{d}_1) & \triangleq & \ll \mathbf{s_{id}} \mapsto [\mathrm{ap}(\mathbf{con}, \mathbf{d}_1)]_{\mathbf{fid}} \gg^{\mathbf{idref}}_{\mathbf{bool}} \\
\mathrm{ap}(\mathbf{name_{id}} \{\!\!\{ \mathbf{ea} \}\!\!\}^{\mathbf{irn}}_{\mathbf{aidn_{tp}}} [\mathbf{f}]_{\mathbf{fid}} \mathbf{con}, \mathbf{d}_1) & \triangleq & \mathbf{name_{id}} \{\!\!\{ \mathbf{ea} \}\!\!\}^{\mathbf{irn}}_{\mathbf{aidn_{tp}}} [\mathbf{f}]_{\mathbf{fid}} \mathrm{ap}(\mathbf{con}, \mathbf{d}_1) \\
\mathrm{ap}(<\mathbf{con}, \mathbf{de}, \mathbf{dnel}>_{\mathrm{DF}}, \mathbf{d}_1) & \triangleq & <\mathrm{ap}(\mathbf{con}, \mathbf{d}_1), \mathbf{de}, \mathbf{dnel}>_{\mathrm{DF}} \\
\mathrm{ap}(<\mathbf{dnel}, \mathbf{con}, \mathbf{dnel}'>_{\mathrm{DF}}, \mathbf{d}_1) & \triangleq & <\mathbf{dnel}, \mathrm{ap}(\mathbf{con}, \mathbf{d}_1), \mathbf{dnel}'>_{\mathrm{DF}} \\
\mathrm{ap}(<\mathbf{dnel}, \mathbf{de}, \mathbf{con}>_{\mathrm{DF}}, \mathbf{d}_1) & \triangleq & <\mathbf{dnel}, \mathbf{de}, \mathrm{ap}(\mathbf{con}, \mathbf{d}_1)>_{\mathrm{DF}}
\end{array}
$$

where:
$\mathbf{d}_1$:D_1, $\mathbf{d}_2$:D_2, $\mathbf{con}$:($\mathrm{D}_1 \to \mathrm{D}_3$), $\mathbf{ea}$:EA, $\mathbf{ef}$:EF, $\mathbf{f}$:D_4
$\mathrm{D}_1, \mathrm{D}_2, \mathrm{D}_3, \mathrm{D}_4 \in \mathcal{D}$

$\mathbf{name} \in \mathrm{S}$

$\mathbf{val} \in \{\mathbf{null}\} \cup \mathrm{S}$, $\mathbf{tp} \in \{1 \ldots 12\}$, $\mathbf{s} \in \mathrm{S}$,
$\mathbf{id}, \mathbf{fid}, \mathbf{idref}, \mathbf{aid} \in \mathrm{ID}$, $\mathbf{aidn}, \mathbf{irn} \in \mathrm{ID} \cup \{\mathbf{null}\}$

Figure 6.4.: The Application Function

standard simple imperative features, described in Chapter 6.3.1, and a large number of commands originating in [23], described in Chapter 6.3.2. Since there is a lot of repeated functionality in [23], we do not exhaustively define every method. Instead, we select a subset of those methods which we define using operational semantics (in Chapter 6.3.2), and define the behaviour of the remaining methods in terms of that subset (in Appendix B.1).

In keeping with the imperative nature of separating our DOM abstract data structure from the programming language, we present DOM in terms of imperative commands rather than methods. Where [23] says "foo.appendChild(bar)", we say "appendChild(foo,bar)". Where [23] specifies an object attribute, we provide a getter and setter command to return and update the relevant data structure. In doing this, we focus on the language-neutral behaviour we intend to specify, rather than getting bogged down in the details of incidentally specifying a particular object-oriented language.

Similarly, we do not reason about exception-handling mechanisms. We take the view of [23]:

Begin Quote

> DOM operations only raise exceptions in "exceptional" circumstances, i.e., when an operation is impossible to perform (either for logical reasons, because data is lost, or because the implementation has become unstable). In general, DOM methods return specific error values in ordinary processing situation, such as out-of-bound errors when using NodeList.

End Quote

Since programs should not rely on exceptions as part of their normal operation, we assume that fault-free programs should be exception-free. Where [23] specifies that an exception should be thrown, our language will fault. The fault-avoiding reasoning given in Chapter 7 will guarantee that any program which satisfies a given Hoare triple will never throw an exception during the normal cause of operation.

6.3.1. The Host Language

As with Featherweight DOM, we define the host language, which we use to demonstrate the use of the DOM library described in Chapter 6.3.2.

This language is not explicitly specified by [23], but aims to be as simple as possible while still allowing us to write substantial DOM programs. This language is therefore dynamically typed, with simple statically defined procedures, dynamic scope and basic imperative control structures. In addition to minimising the constructions needed to describe the language, these choices will simplify the process of evolving our language in the direction of JavaScript in future.

Program State

In addition to the DOM data structure described in Chapter 6.2, the host language has a varible store, defined exactly as in Featherweight DOM.

Definition 42 (The Variable Store)**.** A variable store s is a finite partial function from variables to values:
s:$\mathrm{Var}_{\mathrm{PROG}} \rightarrow (\{\mathbf{null}\} \cup \mathrm{ID} \cup \mathrm{S} \cup \mathbb{Z} \cup \mathbb{B})$

Variable look up of a variable `var` in the store s, is written $s(\texttt{var})$. The notation $[s|\texttt{var} \leftarrow \mathbf{v}]$ extends an existing store s with a new variable `var` containing value $\mathbf{v}$ or to overwrites the existing value of `var` with $\mathbf{v}$. The notation $[s \setminus \texttt{var}]$ removes a variable `var` from the store s. .

Expressions

We introduce expressions $\texttt{Expr} \in \mathrm{Exp}$ which do not alter the program state. Due to the dynamically typed nature of this language, the values of some syntactically correct expressions are undefined. If a program command attempts to evaluate such an expression, it will fault

Definition 43 (Expressions)**.** The expressions $\texttt{Expr} \in \mathrm{Exp}$ of the programming language are:

$\texttt{Expr} ::=$	$\mathbf{null} \mid <\mathbf{c}>_{\mathrm{S}} \mid \mathbf{n} \mid \mathrm{true} \mid \mathrm{false} \mid$	literals: null, characters, integers and booleans
	$\texttt{var} \mid$	variables
	$\texttt{Expr} = \texttt{Expr} \mid$	equality test
	$\texttt{Expr} \otimes_{\mathrm{S}} \texttt{Expr} \mid$	string concatenation
	$\mathrm{len}(\texttt{Expr}) \mid$	string length
	$\texttt{Expr} + \texttt{Expr} \mid \texttt{Expr} - \texttt{Expr} \mid$ $\texttt{Expr} \times \texttt{Expr} \mid \texttt{Expr} \div \texttt{Expr} \mid$	arithmetic operations
	$\texttt{Expr} \wedge \texttt{Expr} \mid \texttt{Expr} \vee \texttt{Expr} \mid \neg\texttt{Expr}$	boolean operations

$$
\begin{aligned}
[\![\mathbf{null}]\!]_s &\triangleq \mathbf{null} \\
[\![\varnothing_{\mathrm{S}}]\!]_s &\triangleq \varnothing_{\mathrm{S}} \\
[\![<\mathbf{c}>_{\mathrm{S}}]\!]_s &\triangleq <\mathbf{c}>_{\mathrm{S}} \\
[\![\mathbf{n}]\!]_s &\triangleq \mathbf{n} \\
[\![\mathrm{true}]\!]_s &\triangleq \mathrm{true} \\
[\![\mathrm{false}]\!]_s &\triangleq \mathrm{false} \\
[\![\mathtt{var}]\!]_s &\triangleq s(\mathtt{var}) \text{ iff } \mathtt{var} \in \mathrm{dom}(s) \\
[\![\mathtt{Expr} = \mathtt{Expr}']\!]_s &\triangleq [\![\mathtt{Expr}]\!]_s = [\![\mathtt{Expr}']\!]_s \\
[\![\mathtt{Expr} \otimes_{\mathrm{S}} \mathtt{Expr}']\!]_s &\triangleq [\![\mathtt{Expr}]\!]_s \otimes_{\mathrm{S}} [\![\mathtt{Expr}']\!]_s \text{ iff } [\![\mathtt{Expr}]\!]_s, [\![\mathtt{Expr}']\!]_s \in \mathrm{S} \\
[\![\mathrm{len}(\mathtt{Expr})]\!]_s &\triangleq \mathrm{len}([\![\mathtt{Expr}]\!]_s) \\
[\![\mathtt{Expr} + \mathtt{Expr}']\!]_s &\triangleq [\![\mathtt{Expr}]\!]_s + [\![\mathtt{Expr}']\!]_s \text{ iff } [\![\mathtt{Expr}]\!]_s, [\![\mathtt{Expr}']\!]_s \in \mathbb{Z} \\
[\![\mathtt{Expr} - \mathtt{Expr}']\!]_s &\triangleq [\![\mathtt{Expr}]\!]_s - [\![\mathtt{Expr}']\!]_s \text{ iff } [\![\mathtt{Expr}]\!]_s, [\![\mathtt{Expr}']\!]_s \in \mathbb{Z} \\
[\![\mathtt{Expr} \times \mathtt{Expr}']\!]_s &\triangleq [\![\mathtt{Expr}]\!]_s \times [\![\mathtt{Expr}']\!]_s \text{ iff } [\![\mathtt{Expr}]\!]_s, [\![\mathtt{Expr}']\!]_s \in \mathbb{Z} \\
[\![\mathtt{Expr} \div \mathtt{Expr}']\!]_s &\triangleq [\![\mathtt{Expr}]\!]_s \div [\![\mathtt{Expr}']\!]_s \text{ iff } [\![\mathtt{Expr}]\!]_s, [\![\mathtt{Expr}']\!]_s \in \mathbb{Z} \\
[\![\mathtt{Expr} \wedge \mathtt{Expr}']\!]_s &\triangleq [\![\mathtt{Expr}]\!]_s \wedge [\![\mathtt{Expr}']\!]_s \text{ iff } [\![\mathtt{Expr}]\!]_s, [\![\mathtt{Expr}']\!]_s \in \mathbb{B} \\
[\![\mathtt{Expr} \vee \mathtt{Expr}']\!]_s &\triangleq [\![\mathtt{Expr}]\!]_s \vee [\![\mathtt{Expr}']\!]_s \text{ iff } [\![\mathtt{Expr}]\!]_s, [\![\mathtt{Expr}']\!]_s \in \mathbb{B} \\
[\![\neg \mathtt{Expr}]\!]_s &\triangleq \neg [\![\mathtt{Expr}]\!]_s \text{ iff } [\![\mathtt{Expr}]\!]_s \in \mathbb{B}
\end{aligned}
$$

Figure 6.5.: The Evaluation of Expressions

Just as in Chapter 6.2, we write "abc" to refer to the string $<$'a'$>_{\mathrm{S}} \otimes_{\mathrm{S}} <$'b'$>_{\mathrm{S}} \otimes_{\mathrm{S}} <$'c'$>_{\mathrm{S}}$

To distinguish program variables from expressions in our reasoning, we adopt the convention that program variable names are always written entirely in `lowercase`, while expression names will always be written in `UpperCamelCase`.

In order to define our expression evaluation, we require the auxiliary function len.

Definition 44 (Forest Length)**.** Recall $\mathcal{N}$ and $\mathcal{F}$, from Definition 37. The length of a string or forest is defined by:

$$
\begin{aligned}
\mathrm{len}(\varnothing_{\mathrm{D}}) &\triangleq 0 \\
\mathrm{len}(<\mathbf{d}>_{\mathrm{D}}) &\triangleq 1 \\
\mathrm{len}(\mathbf{d}_1 \otimes_{\mathrm{D}} \mathbf{d}_2) &\triangleq \mathrm{len}(\mathbf{d}_1) + \mathrm{len}(\mathbf{d}_2)
\end{aligned}
$$

where $\mathrm{D} \in \mathcal{F}$, $\mathbf{d}_1, \mathbf{d}_2 \in \mathrm{D}$, $\mathbf{d} \in \mathrm{D}'$ and $\mathrm{D}' \in \mathcal{N}$

Definition 45 (Expression Evaluation)**.** The evaluation $[\![\texttt{Expr}]\!]_s$ of expression `Expr` in store s is given in Figure 6.5.

Note that this evaluation is a partial function, since not all expressions can be successfully evaluated. Attempting to evaluate such an expression at runtime will result in a runtime fault.

The DTD Fragment

XML Documents in the real world exist in the context of a "DTD" or "Document Type Definition" which describes what subset of XML is valid in the current context. For example, on the web, we use xhtml[44]. In DOM, we have no need to model the whole DTD, but the behaviour of DOM is affected by a fragment of the DTD in the current context. We model this context as follows:

Definition 46 (DTD Fragments)**.** We define a DTD fragment d:$(\mathrm{S}, \mathrm{S}) \rightharpoonup \mathrm{S}$ which is a partial function mapping element and attribute names to default attribute values. The full use of this structure is described in section 6.3.2.

Standard Imperative Commands

As with Featherweight DOM, we describe a minimal imperative language, with which we can use the DOM library.

Definition 47 (Imperative Commands)**.** The commands of the imperative language are:

$\texttt{C} ::= \texttt{var} := \texttt{Expr}$	assignment
$\mid \texttt{C}; \texttt{C}$	sequential composition
$\mid$ `if Expr then C else C fi`	conditionals
$\mid$ `while Expr do C od`	loops
$\mid$ `local var : C endloc`	local variable declaration
$\mid$ `skip`	skip
$\mid \texttt{v:=procname}(\overline{\texttt{params}}) \triangleq \texttt{C}$	procedure declaration
$\mid \texttt{v:=procname}(\overline{\texttt{vars}})$	procedure call
$\mid \texttt{C}_{\mathrm{DOM}}$	DOM commands (see Definition 49)

where $\overline{\texttt{params}}$ is a vector of parameters, and $\overline{\texttt{vars}}$ is a vector of variables..

$$\frac{[\![\texttt{Expr}]\!]_s = \mathbf{val}}{s, d, \mathbf{g}, \texttt{var:=Expr} \rightsquigarrow [s|\texttt{var}\leftarrow\mathbf{val}], d, \mathbf{g}}$$

$$\frac{s, d, \mathbf{g}, \texttt{C}_1 \rightsquigarrow s', d, \mathbf{g}' \quad s', d, \mathbf{g}', \texttt{C}_2 \rightsquigarrow s'', d, \mathbf{g}''}{s, d, \mathbf{g}, (\texttt{C}_1 \texttt{ ; } \texttt{C}_2) \rightsquigarrow s'', d, \mathbf{g}''}$$

$$\frac{[\![\texttt{Bool}]\!]s = \text{true} \quad s, d, \mathbf{g}, \texttt{C}_1 \rightsquigarrow s', d, \mathbf{g}'}{s, d, \mathbf{g}, \texttt{if Bool then C}_1 \texttt{ else C}_2 \texttt{ fi} \rightsquigarrow s', d, \mathbf{g}'}$$

$$\frac{[\![\texttt{Bool}]\!]s = \text{false} \quad s, d, \mathbf{g}, \texttt{C}_2 \rightsquigarrow s', d, \mathbf{g}'}{s, d, \mathbf{g}, \texttt{if Bool then C}_1 \texttt{ else C}_2 \texttt{ fi} \rightsquigarrow s', d, \mathbf{g}'}$$

$$\frac{s, d, \mathbf{g}, \texttt{if Bool then C ; while Bool do C od else skip fi} \rightsquigarrow s', d, \mathbf{g}'}{s, d, \mathbf{g}, \texttt{while Bool do C od} \rightsquigarrow s', d, \mathbf{g}'}$$

$$\frac{\mathbf{val} = s(\texttt{var}) \quad [s|\texttt{var} \leftarrow \mathbf{null}], d, \mathbf{g}, \texttt{C} \rightsquigarrow s', d, \mathbf{g}'}{p, d, \mathbf{g}, \texttt{local var : C endloc} \rightsquigarrow [s'|\texttt{var} \leftarrow \mathbf{val}], d, \mathbf{g}'}$$

$$\frac{\texttt{var} \notin \text{dom}(s) \quad [s|\texttt{var} \leftarrow \mathbf{null}], d, \mathbf{g}, \texttt{C} \rightsquigarrow s', d, \mathbf{g}'}{s, d, \mathbf{g}, \texttt{local var : C endloc} \rightsquigarrow [s' \setminus \texttt{var}], d, \mathbf{g}'}$$

$$\frac{}{s, d, \mathbf{g}, \texttt{skip} \rightsquigarrow s, d, \mathbf{g}}$$

Figure 6.6.: Imperative Commands

We deal with procedures in Section 6.3.1 and the DOM commands in Section 6.3.2. We give big-step operational semantics for the remaining commands here.

Definition 48 (Imperative Commands)**.** The operational semantics of the standard imperative commands are given in Figure 6.6 by an evaluation relation $\rightsquigarrow$ relating configuration quadruples $s, d, \mathbf{g}, \texttt{C}$, terminal states $s, d, \mathbf{g}$, and faults.

The simple imperative commands for DOM Core Level 1 behave exactly as the simple imperative commands for Featherweight DOM. We include them here however, since the $\rightsquigarrow$ relation for Featherweight DOM relates configuration triples, not quadruples, to terminal states. The new element in the Core Level 1 configuration quadruples is the DTD Fragment d, defined in Definition 46, and is only used by certain DOM commands defined in Section 6.3.2.

As with Featherweight DOM, we use the simple syntactic sugars for else-less if and grouped local scope declarations

Procedures

As with Featherweight DOM, we provide a primitive procedure system with dynamic scope. Procedures are defined statically for each program exactly as for Featherweight DOM in Section 3.2.4.Exactly as in Featherweight DOM, procedures are represented during program execution as a partial function `procs` which maps procedure names `procname` to the tuple $(\mathtt{v}, \overline{\mathtt{params}}, \mathtt{C})$ which contains the return variable `v`, a vector of the procedure's parameter variables $\overline{\mathtt{params}}$, and the procedure's body `C`.

Procedure call is defined almost exactly as for Featherweight DOM, except that the rule uses a Core Level 1 configuration quadruples rather than a Featherweight DOM configuration triple. As with the simple imperative commands, the extra element in this quadruples is the DTD Fragment d:

$$\frac{\begin{array}{l}\mathtt{procs}(\mathtt{procname}) = (\mathtt{v}', \overline{\mathtt{params}}, \mathtt{C}) \\ |\overline{\mathtt{params}}| = |\overline{\mathtt{vars}}| \\ s, d, \mathbf{g}, \mathtt{C}\{\mathtt{v}/\mathtt{v}', \forall \mathtt{p}_i \in \overline{\mathtt{params}}, \mathtt{v}_i \in \overline{\mathtt{vars}}.\, \mathtt{v}_i/\mathtt{p}_i\} \rightsquigarrow s', d, \mathbf{g}'\end{array}}{s, d, \mathbf{g}, \mathtt{v}{:=}\mathtt{procname}(\overline{\mathtt{vars}}) \rightsquigarrow s', d, \mathbf{g}'}$$

where $\overline{\mathtt{vars}}$ is a vector of variables.

We have the same `devnull` variable and syntactic sugars for ignoring the return values of procedures as were given for Featherweight DOM in Section 3.2.4.

6.3.2. DOM Library Commands

Next, we complete our language by defining operational semantics for each command in DOM. We present the commands from each Interface of DOM separately.

Definition 49 (DOM Library Commands)**.** The DOM Library Commands

are:

$\mathtt{C}_{\mathrm{DOM}} ::= \mathtt{C}_{\mathrm{DOC}}$	Document Commands, see Definition 50
$\mid \mathtt{C}_{\mathrm{NODE}}$	Node Commands, see Definition 51
$\mid \mathtt{C}_{\mathrm{NODELIST}}$	NodeList Commands, see Definition 53
$\mid \mathtt{C}_{\mathrm{ELE}}$	Element Commands, see Definition 55
$\mid \mathtt{C}_{\mathrm{ATTR}}$	Attr Commands, see Definition 56
$\mid \mathtt{C}_{\mathrm{MAP}}$	NamedNodeMap Commands, see Definition 57
$\mid \mathtt{C}_{\mathrm{CD}}$	CharacterData Commands, see Definition 58

Any attempt to execute a syntactically correct program which is not covered by the following semantics will result in a runtime fault. For example, calling `createElement`(x, s) in an environment where x is not the id of a Document node will fault. Wherever [23] calls for an exception to be thrown, we will fault in this way.

These commands are defined in groups corresponding to the interfaces in which they are defined in the DOM specification, but first it is best to explain the behaviour of "specified", which affects several commands from various interfaces.

The "Specified" Property and The DTD Fragment

Recall that in Section 6.2.6, we described the general shape of the Attr data structure as being:

$$\ll name_{\mathbf{id}} \mapsto [kids]_{\mathbf{fid}} \gg^{\mathbf{idref}}_{\mathbf{specified}}$$

The boolean property "specified" corresponds to the object attribute of the same name defined on the Attr interface. The behaviour of DOM implementations in their interpretation of this part of the specification is quite varied. We investigate this behaviour in Chapter 8 and Appendix C.

Since there is no consensus among DOM implementations for how this property should behave, it is all the more important that we carefully interpret the specification document. The "specified" property is described in [23] as follows:

Begin Quote

If this attribute was explicitly given a value in the original document, this is true; otherwise, it is false. Note that the implementation is in charge of this attribute, not the user. If the user changes the value of the attribute (even if it ends up having the same value as the default value) then the specified flag is automatically flipped to true. To re-specify the attribute as the default value from the DTD, the user must delete the attribute. The implementation will then make a new attribute available with specified set to false and the default value (if one exists).

In summary:

- If the attribute has an assigned value in the document then specified is true, and the value is the assigned value.
- If the attribute has no assigned value in the document and has a default value in the DTD, then specified is false, and the value is the default value in the DTD.
- If the attribute has no assigned value in the document and has a value of #IMPLIED in the DTD, then the attribute does not appear in the structure model of the document.

End Quote

The first important thing to notice about the value of specified then, is that it will be automatically changed to true if any change is made to the attribute in question. Thus, any command in the following semantics which changes an Attr node, will also change its "specified" property.

The second important thing to notice is that there are only two ways the value of specified for any node can ever be false. Firstly, it may have been parsed that way - which is outside the scope of this document. Secondly, it might be that an Attr node was removed, and re-created by DOM using the default value in the Document Type Definition (DTD - [25]).

Modelling the behaviour of the DTD is beyond the scope of this document, so we define a minimal fragment sufficient for our needs. That DTD fragment is defined above in Definition 46: a partial function d mapping element and attribute names to default attribute values. Whenever an Attr is removed from a NamedNodeMap, the name of the Attr, and the name of

the Element which owns that Attr will be looked up in the DTD fragment d. If they are not found there, then the removal will happen as normal. If, on the other hand, a default attribute value is found for that Element-Attr pair, then a new Attr will be created with that value, and inserted into the NamedNodeMap in question.

The DTD fragment d cannot be altered programatically. It is defined for a given run of the program, according to the type of document that the program is manipulating. For example, in a web browser, the value of d will likely be a description of the default values of various HTML attributes.

Document

We define the Document Interface commands, following the "Document Interface" of [23].

Definition 50 (Document Interface Commands)**.** The Document Interface Commands are:

$$\begin{aligned}
\mathsf{C}_{\mathrm{DOC}} ::= {} & \mathtt{x} := \mathtt{createElement}(\mathtt{Doc}, \mathtt{TagName}) \\
| \; & \mathtt{x} := \mathtt{createDocumentFragment}(\mathtt{Doc}) \\
| \; & \mathtt{x} := \mathtt{createTextNode}(\mathtt{Doc}, \mathtt{Data}) \\
| \; & \mathtt{x} := \mathtt{createComment}(\mathtt{Doc}, \mathtt{Data}) \\
| \; & \mathtt{x} := \mathtt{createAttribute}(\mathtt{Doc}, \mathtt{Name}) \\
| \; & \mathtt{x} := \mathtt{createDocument}() \\
| \; & \mathtt{x} := \mathtt{getElementsByTagName}(\mathtt{Doc}, \mathtt{TagName})
\end{aligned}$$

Since there are no classes in [23] which subclass Document, all the Document Interface commands take as their first argument a document node `Doc`, which we know must have the shape:

$$\text{“\#document”}_{[\![\mathtt{Doc}]\!]_s} \{\!| \varnothing_{\mathrm{EA}} |\!\}_{\mathbf{null}_9}^{\mathbf{null}} [\mathbf{f}]_{\mathbf{fid}'} \mathbf{null}$$

with nil document owner, no attributes and a type number of 9.

Node Creation [23] provides methods for creating every type of node apart from Documents. Document nodes cannot be created programatically using DOM, and are assumed to come into existence when an XML document is parsed. DOM provides a means of editing documents, not cre-

ating them. Furthermore, the methods provided for creating all other types of node do so in the context of a Document, which will be the new Node's Owner Document.

Each of the following commands therefore execute in a state in which the grove contains at least one document. That document is identified by "`Doc`" – the first parameter of the command. The command then alters the state only in that it creates a single new Node, and assigns its id to the return variable. The commands `createElement` and `createAttribute` also take a second parameter "`Name`", which is a string copied into the new node. The commands `createTextNode` and `createComment` take as their second parameter the string "`Data`", which is similarly copied into the new node.

In all the following commands $\mathbf{id}, \mathbf{aid}$, $\mathbf{fid}$ and $\mathbf{fid}' \in \mathrm{ID}$ are fresh.

$$\frac{\mathbf{g} \equiv \mathbf{g}'' \oplus <\text{“\#document”}_{[\![\texttt{Doc}]\!]_s} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathbf{null}}_{\mathbf{null}_9} [\mathbf{f}]_{\mathbf{fid}'}\mathbf{null}>_{\mathrm{G}} \quad \mathbf{g}' \equiv \mathbf{g} \oplus <[\![\texttt{TagName}]\!]_{s\,\mathbf{id}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{[\![\texttt{Doc}]\!]_s}_{\mathbf{aid}_1} [\varnothing_{\mathrm{EF}}]_{\mathbf{fid}}\mathbf{null}>_{\mathrm{G}}}{s, d, \mathbf{g}, \mathtt{x} := \texttt{createElement}(\texttt{Doc}, \texttt{TagName}) \rightsquigarrow [s|\mathtt{x} \leftarrow \mathbf{id}], d, \mathbf{g}'}$$

In the case that we wish to provide an HTML (rather than general XML) implementation of DOM, we can replace the above `createElement` command with the following:

$$\frac{\mathbf{g} \equiv \mathbf{g}'' \oplus <\text{“\#document”}_{[\![\texttt{Doc}]\!]_s} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathbf{null}}_{\mathbf{null}_9} [\mathbf{f}]_{\mathbf{fid}'}\mathbf{null}>_{\mathrm{G}} \quad \mathbf{g}' \equiv \mathbf{g} \oplus <\mathrm{toUpper}([\![\texttt{TagName}]\!]_s)_{\mathbf{id}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{[\![\texttt{Doc}]\!]_s}_{\mathbf{aid}_1} [\varnothing_{\mathrm{EF}}]_{\mathbf{fid}}\mathbf{null}>_{\mathrm{G}}}{s, d, \mathbf{g}, \mathtt{x} := \texttt{createElement}(\texttt{Doc}, \texttt{TagName}) \rightsquigarrow [s|\mathtt{x} \leftarrow \mathbf{id}], d, \mathbf{g}'}$$

which makes use of the auxiliary function "toUpper(**s**)", which maps every alphabetic character in **s** to its uppercase equivalent. On non-alphabetic characters, toUpper is the identity function. The HTML version of the command then maps `TagName` to the canonical uppercase. In practice, most web browsers use the XML version of this command. As such, in our examples, so will we.

$$\frac{\mathbf{g} \equiv \mathbf{g}'' \oplus <\text{“\#document”}_{[\![\texttt{Doc}]\!]_s} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathbf{null}}_{\mathbf{null}_9} [\mathbf{f}]_{\mathbf{fid}'}\mathbf{null}>_{\mathrm{G}} \quad \mathbf{g}' \equiv \mathbf{g} \oplus <\text{“\#document-fragment”}_{\mathbf{id}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{[\![\texttt{Doc}]\!]_s}_{\mathbf{null}_{11}} [\varnothing_{\mathrm{FRAGF}}]_{\mathbf{fid}}\mathbf{null}>_{\mathrm{G}}}{s, d, \mathbf{g}, \mathtt{x} := \texttt{createDocumentFragment}(\texttt{Doc}) \rightsquigarrow [s|\mathtt{x} \leftarrow \mathbf{id}], d, \mathbf{g}'}$$

$$\frac{\mathbf{g} \equiv \mathbf{g}'' \oplus <\text{“\#document”}_{[\![\texttt{Doc}]\!]_s} \{\!\!\{ \varnothing_{\text{EA}} \}\!\!\}_{\mathbf{null}_9}^{\mathbf{null}} [\mathbf{f}]_{\mathbf{fid}'} \mathbf{null}>_{\text{G}} \quad \mathbf{g}' \equiv \mathbf{g} \oplus <\text{“\#text-node”}_{\mathbf{id}} \{\!\!\{ \varnothing_{\text{EA}} \}\!\!\}_{\mathbf{null}_3}^{[\![\texttt{Doc}]\!]_s} [\varnothing_{\text{TF}}]_{\mathbf{fid}} [\![\texttt{Data}]\!]_s>_{\text{G}}}{s, d, \mathbf{g}, \mathtt{x} := \texttt{createTextNode}(\texttt{Doc}, \texttt{Data}) \rightsquigarrow [s|\mathtt{x} \leftarrow \mathbf{id}], d, \mathbf{g}'}$$

$$\frac{\mathbf{g} \equiv \mathbf{g}'' \oplus <\text{“\#document”}_{[\![\texttt{Doc}]\!]_s} \{\!\!\{ \varnothing_{\text{EA}} \}\!\!\}_{\mathbf{null}_9}^{\mathbf{null}} [\mathbf{f}]_{\mathbf{fid}'} \mathbf{null}>_{\text{G}} \quad \mathbf{g}' \equiv \mathbf{g} \oplus <\text{“\#comment”}_{\mathbf{id}} \{\!\!\{ \varnothing_{\text{EA}} \}\!\!\}_{\mathbf{null}_8}^{[\![\texttt{Doc}]\!]_s} [\varnothing_{\text{CF}}]_{\mathbf{fid}} [\![\texttt{Data}]\!]_s>_{\text{G}}}{s, d, \mathbf{g}, \mathtt{x} := \texttt{createComment}(\texttt{Doc}, \texttt{Data}) \rightsquigarrow [s|\mathtt{x} \leftarrow \mathbf{id}], d, \mathbf{g}'}$$

$$\frac{\mathbf{g} \equiv \mathbf{g}'' \oplus <\text{“\#document”}_{[\![\texttt{Doc}]\!]_s} \{\!\!\{ \varnothing_{\text{EA}} \}\!\!\}_{\mathbf{null}_9}^{\mathbf{null}} [\mathbf{f}]_{\mathbf{fid}'} \mathbf{null}>_{\text{G}} \quad \mathbf{g}' \equiv \mathbf{g} \oplus <\ll [\![\texttt{Name}]\!]_{s\,\mathbf{id}} \mapsto [\varnothing_{\text{AF}}]_{\mathbf{fid}} \gg_{\mathbf{true}}^{[\![\texttt{Doc}]\!]_s} >_{\text{G}}}{s, d, \mathbf{g}, \mathtt{x} := \texttt{createAttribute}(\texttt{Doc}, \texttt{Name}) \rightsquigarrow [s|\mathtt{x} \leftarrow \mathbf{id}], d, \mathbf{g}'}$$

The command `createAttribute` is particularly interesting because Attr nodes behave differently to other Node objects. As described in Sections 6.2.6 and 6.3.2, Attr nodes have a boolean field called “specified”. This field tells us whether the value of the attribute was specified explicitly, or whether it was set by default in the DTD. Since all attributes created by `createAttribute` are created explicitly by the programmer, all attributes created in this way always have a “specified” value of “true”.

Finally, we provide the `createDocument`, even though it is not specified in DOM Core Level 1. We provide this command in the interests of completeness of this language, since without it, the only way to create a new Document node after the initial document parse (which is beyond the scope of this thesis) is to clone an existing Document node and delete all its contents. Since the command “cloneNode” is in all other respects a composite command, we choose to include the simple command `createDocument` in our core language, and implement the far more complex `cloneNode` in Appendix B.1.

$$\frac{\mathbf{g}' \equiv \mathbf{g} \oplus <\text{“\#document”}_{\mathbf{id}} \{\!\!\{ \varnothing_{\text{EA}} \}\!\!\}_{\mathbf{null}_9}^{\mathbf{null}} [\varnothing_{\text{DNEL}}, \varnothing_{\text{DE}}, \varnothing_{\text{DNEL}}]_{\mathbf{fid}} \mathbf{null}>_{\text{G}}}{s, d, \mathbf{g}, \mathtt{x} := \texttt{createDocument}() \rightsquigarrow [s|\mathtt{x} \leftarrow \mathbf{id}], d, \mathbf{g}'}$$

`getElementsByTagName` As described in Chapter 6.2.8, this method is best implemented “lazily” by caching the parameters of the method in a new structure, and generating the results of the call each time they are accessed. This command therefore operates in a similar manner to the node creation commands.

$$\frac{\begin{array}{l}\mathbf{g} \equiv \mathbf{g}'' \oplus <\text{“\#document”}_{[\![\texttt{Doc}]\!]_s} \{\!\!\{ \varnothing_{\text{EA}} \}\!\!\}^{\mathbf{null}}_{\mathbf{null9}} [\mathbf{f}]_{\mathbf{fid}} \mathbf{null}>_{\text{G}} \\ \mathbf{g}' \equiv \mathbf{g} \oplus <^{[\![\texttt{Doc}]\!]_s}_{[\![\texttt{TagName}]\!]_s \mathbf{id}}>_{\text{G}}\end{array}}{s, d, \mathbf{g}, \texttt{x} := \texttt{getElementsByTagName}(\texttt{Doc}, \texttt{TagName}) \rightsquigarrow [s|\texttt{x} \leftarrow \mathbf{id}], d, \mathbf{g}'}$$

Node

The Node interface is subclassed by the Document, DocumentFragment, Element, Attribute, Comment and Text interfaces. The following commands therefore take as their first argument a node N, which may be of any type $\text{D} \in (\mathcal{N} \setminus \text{ES})$, where $\mathcal{N}$ is the “Nodes” type grouping from Definition 37 and ES is the type of Element Search nodes. We therefore present cases for the general node shape $\mathbf{name}_{[\![\texttt{N}]\!]_s} \{\!\!\{ \mathbf{af} \}\!\!\}^{\mathbf{irn}}_{\mathbf{aidn_{tp}}} [\mathbf{f}]_{\mathbf{fid}} \mathbf{val}$ and for the special case of Attr nodes $\ll \mathbf{name}_{[\![\texttt{N}]\!]_s} \mapsto [\mathbf{af}]_{\mathbf{fid}} \gg^{\mathbf{doc}}_{\mathbf{specified}}$.

Definition 51 (Node Interface Commands)**.** The Node Interface Commands are:

$$\begin{aligned}\text{C}_{\text{NODE}} ::= {} & \texttt{nm} := \texttt{getNodeName}(\texttt{N}) \\ | \ & \texttt{v} := \texttt{getNodeValueHelper}(\texttt{N}) \\ | \ & \texttt{setNodeValueHelper}(\texttt{N}, \texttt{Str}) \\ | \ & \texttt{i} := \texttt{getNodeType}(\texttt{N}) \\ | \ & \texttt{p} := \texttt{getParentNode}(\texttt{N}) \\ | \ & \texttt{kids} := \texttt{getChildNodes}(\texttt{N}) \\ | \ & \texttt{ats} := \texttt{getAttributes}(\texttt{N}) \\ | \ & \texttt{od} := \texttt{getOwnerDocument}(\texttt{N}) \\ | \ & \texttt{n} := \texttt{appendChild}(\texttt{Parent}, \texttt{NewChild}) \\ | \ & \texttt{n} := \texttt{removeChild}(\texttt{Parent}, \texttt{OldChild})\end{aligned}$$

`getNodeName` This is a “getter” command, which provides the functionality of the read-only object attribute “nodeName” as specified in [23]. It returns the name of the node N, regardless of its type.

$$\frac{\mathbf{g} \equiv \text{ap}(\mathbf{cg}, \mathbf{name}_{[\![\texttt{N}]\!]_s} \{\!\!\{ \mathbf{af} \}\!\!\}^{\mathbf{irn}}_{\mathbf{aidn_{tp}}} [\mathbf{f}]_{\mathbf{fid}} \mathbf{val})}{s, d, \mathbf{g}, \texttt{nm} := \texttt{getNodeName}(\texttt{N}) \rightsquigarrow [s|\texttt{nm} \leftarrow \mathbf{name}], d, \mathbf{g}}$$

$$\frac{\mathbf{g} \equiv \text{ap}(\mathbf{cg}, \ll \mathbf{name}_{[\![\texttt{N}]\!]_s} \mapsto [\mathbf{af}]_{\mathbf{fid}} \gg^{\mathbf{doc}}_{\mathbf{specified}})}{s, d, \mathbf{g}, \texttt{nm} := \texttt{getNodeName}(\texttt{N}) \rightsquigarrow [s|\texttt{nm} \leftarrow \mathbf{name}], d, \mathbf{g}}$$

`getNodeValue` The command `getNodeValue` provides part of the functionality of the object attribute "nodeValue" (the other part being provided by `setNodeValue`). It returns the value of the node `N`, or **null** if `N` is of a type that has no value. `getNodeValue` can be implemented by making use of existing commands, in conjunction with the following helper command:

$$\frac{\mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}, \mathbf{name}_{[\![\mathtt{N}]\!]_s} \{\!\!\{ \mathbf{af} \}\!\!\}^{\mathbf{irn}}_{\mathbf{aidn_{tp}}} [\mathbf{f}]_{\mathbf{fid}} \mathbf{val})}{s, d, \mathbf{g}, \mathtt{v} := \mathtt{getNodeValueHelper}(\mathtt{N}) \rightsquigarrow [s|\mathtt{v} \leftarrow \mathbf{val}], d, \mathbf{g}}$$

Note that this helper command is equivalent to `getNodeValue` in the non-Attr case. For Attribute nodes, we provide a more complex composite command which accesses every child of `N` in Chapter B.1.

`setNodeValue` This command provides the remaining part of the functionality of the object attribute "nodeValue". It allows the programmer to set the value of any node which allows it. `setNodeValue` can be implemented by making use of existing commands, in conjunction with the following helper command:

$$\frac{\begin{array}{l}\mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}, \mathbf{name}_{[\![\mathtt{N}]\!]_s} \{\!\!\{ \mathbf{af} \}\!\!\}^{\mathbf{irn}}_{\mathbf{aidn_{tp}}} [\mathbf{f}]_{\mathbf{fid}} \mathbf{val}) \\ \mathbf{g}' \equiv \mathrm{ap}(\mathbf{cg}, \mathbf{name}_{[\![\mathtt{N}]\!]_s} \{\!\!\{ \mathbf{af} \}\!\!\}^{\mathbf{irn}}_{\mathbf{aidn_{tp}}} [\mathbf{f}]_{\mathbf{fid}} [\![\mathtt{Str}]\!]_s)\end{array}}{s, d, \mathbf{g}, \mathtt{setNodeValueHelper}(\mathtt{N}, \mathtt{Str}) \rightsquigarrow s, d, \mathbf{g}'}$$

Note that this helper command is equivalent to `setNodeValue` in the non-Attr case. For Attribute nodes we provide a more complex composite command which accesses every child of `N` in Chapter B.1.

Which node types allow their values to be set is determined by the data structure itself. For example, consider a Grove $\mathbf{g}$ containing an Element node `Ele`:

$$\mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}, \mathbf{name}_{[\![\mathtt{Ele}]\!]_s} \{\!\!\{ \mathbf{af} \}\!\!\}^{\mathbf{idref}}_{\mathbf{aid}_1} [\mathbf{f}]_{\mathbf{fid}} \mathbf{null})$$

There can be no data structure $\mathbf{g}'$ containing any Element Node `Ele` with a non-**null** value, and so it is impossible to call $\mathtt{setNodeValueHelper}(\mathtt{Ele}, \mathtt{S})$.

`getNodeType` This command provides the functionality of the read-only object attribute "nodeType". Note that the type number of every node type other than Attribute nodes is recorded in the node data structure.

Attribute nodes have a unique structure, and so the type number "2" is not explicitly recorded.

$$\frac{\mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}, \mathbf{name}_{[\![\mathtt{N}]\!]_s} \{\!\!\{ \mathbf{af} \}\!\!\}^{\mathbf{irn}}_{\mathbf{aidn}_{\mathbf{tp}}} [\mathbf{f}]_{\mathbf{fid}} \mathbf{val})}{s, d, \mathbf{g}, \mathtt{i} := \mathtt{getNodeType(N)} \rightsquigarrow [s|\mathtt{i} \leftarrow \mathbf{tp}], d, \mathbf{g}}$$

$$\frac{\mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}, \ll \mathbf{name}_{[\![\mathtt{N}]\!]_s} \mapsto [\mathbf{af}]_{\mathbf{fid}} \gg^{\mathbf{doc}}_{\mathbf{specified}})}{s, d, \mathbf{g}, \mathtt{i} := \mathtt{getNodeType(N)} \rightsquigarrow [s|\mathtt{i} \leftarrow \mathbf{2}], d, \mathbf{g}}$$

`getParentNode` This command provides the functionality of the read-only object attribute "parentNode". It returns a reference to the node which is `N`'s parent. If `N` has no parent, or is an Attribute node, `getParentNode` returns **null**.

$$\frac{\begin{array}{l}\mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}, \mathbf{name}_{\mathbf{id}} \{\!\!\{ \mathbf{af} \}\!\!\}^{\mathbf{irn}}_{\mathbf{aidn}_{\mathbf{tp}}} [\\ \quad \mathbf{f}_1 \otimes_{\mathrm{D}_2} <\mathbf{name}'_{[\![\mathtt{N}]\!]_s} \{\!\!\{ \mathbf{af}' \}\!\!\}^{\mathbf{irn}'}_{\mathbf{aidn}'_{\mathbf{tp}'}} [\mathbf{f}']_{\mathbf{fid}'} \mathbf{val}'>_{\mathrm{D}_2} \otimes_{\mathrm{D}_2} \mathbf{f}_2 \\ ]_{\mathbf{fid}} \mathbf{val})\end{array}}{s, d, \mathbf{g}, \mathtt{p} := \mathtt{getParentNode(N)} \rightsquigarrow [s|\mathtt{p} \leftarrow \mathbf{id}], d, \mathbf{g}}$$

$$\frac{\mathbf{g} \equiv \mathbf{g}' \oplus <\mathbf{name}_{[\![\mathtt{N}]\!]_s} \{\!\!\{ \mathbf{af} \}\!\!\}^{\mathbf{irn}}_{\mathbf{aidn}_{\mathbf{tp}}} [\mathbf{f}]_{\mathbf{fid}} \mathbf{val}>_{\mathrm{G}}}{s, d, \mathbf{g}, \mathtt{p} := \mathtt{getParentNode(N)} \rightsquigarrow [s|\mathtt{p} \leftarrow \mathbf{null}], d, \mathbf{g}}$$

$$\frac{\mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}, \ll \mathbf{name}_{[\![\mathtt{N}]\!]_s} \mapsto [\mathbf{af}]_{\mathbf{fid}} \gg^{\mathbf{doc}}_{\mathbf{specified}})}{s, d, \mathbf{g}, \mathtt{p} := \mathtt{getParentNode(N)} \rightsquigarrow [s|\mathtt{p} \leftarrow \mathbf{null}], d, \mathbf{g}}$$

$$\frac{\begin{array}{l}\mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}, \ll \mathbf{name}_{\mathbf{id}} \mapsto [\\ \quad \mathbf{af}_1 \otimes_{\mathrm{AF}} <\mathbf{name}'_{[\![\mathtt{N}]\!]_s} \{\!\!\{ \mathbf{af}' \}\!\!\}^{\mathbf{irn}'}_{\mathbf{aidn}'_{\mathbf{tp}'}} [\mathbf{f}']_{\mathbf{fid}'} \mathbf{val}'>_{\mathrm{AF}} \otimes_{\mathrm{AF}} \mathbf{af}_2 \\ ]_{\mathbf{fid}} \gg^{\mathbf{doc}}_{\mathbf{specified}})\end{array}}{s, d, \mathbf{g}, \mathtt{p} := \mathtt{getParentNode(N)} \rightsquigarrow [s|\mathtt{p} \leftarrow \mathbf{id}], d, \mathbf{g}}$$

$$\frac{\begin{array}{l}\mathbf{g} \equiv \mathbf{g}' \oplus \\ <\text{“\#document”}_{\mathbf{id}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathbf{null}}_{\mathbf{null}_{\mathbf{9}}} [\\ \quad <\mathbf{dnel}_1 \otimes_{\mathrm{DNEL}} \\ \quad <\mathbf{name}'_{[\![\mathtt{N}]\!]_s} \{\!\!\{ \mathbf{af}' \}\!\!\}^{\mathbf{irn}'}_{\mathbf{aidn}'_{\mathbf{tp}'}} [\mathbf{f}']_{\mathbf{fid}'} \mathbf{val}'>_{\mathrm{DNEL}} \\ \quad \otimes_{\mathrm{DNEL}} \mathbf{dnel}_2, \mathbf{de}, \mathbf{dnel}>_{\mathrm{DF}} \\ ]_{\mathbf{fid}} \mathbf{null}>_{\mathrm{G}}\end{array}}{s, d, \mathbf{g}, \mathtt{p} := \mathtt{getParentNode(N)} \rightsquigarrow [s|\mathtt{p} \leftarrow \mathbf{id}], d, \mathbf{g}}$$

$$\frac{\begin{array}{l}\mathbf{g} \equiv \mathbf{g}'\oplus \\ <\text{“\#document”}_{\mathbf{id}}\{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathbf{null}}_{\mathbf{null_9}}[\\ \quad <\mathbf{dnel}_1, <\mathbf{s}_{[\![\mathtt{N}]\!]_s}\{\!\!\{ \mathbf{af}' \}\!\!\}^{\mathbf{idref}}_{\mathbf{aid_1}}[\mathbf{f}]_{\mathbf{fid}'}\mathbf{null}>_{\mathrm{DE}}, \mathbf{dnel}_2 \\ ]_{\mathbf{fid}}\mathbf{null}>_{\mathrm{G}}\end{array}}{s, d, \mathbf{g}, \mathtt{p} := \mathtt{getParentNode(N)} \rightsquigarrow [s|\mathtt{p} \leftarrow \mathbf{id}], d, \mathbf{g}}$$

$$\frac{\begin{array}{l}\mathbf{g} \equiv \mathbf{g}'\oplus \\ <\text{“\#document”}_{\mathbf{id}}\{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathbf{null}}_{\mathbf{null_9}}[\\ \quad <\mathbf{dnel}, \mathbf{de}, \mathbf{dnel}_1 \otimes_{\mathrm{DNEL}} \\ \quad <\mathbf{name}'_{[\![\mathtt{N}]\!]_s}\{\!\!\{ \mathbf{af}' \}\!\!\}^{\mathbf{irn}'}_{\mathbf{aidn}'\mathbf{tp}'}[\mathbf{f}']_{\mathbf{fid}'}\mathbf{val}'>_{\mathrm{DNEL}} \\ \quad \otimes_{\mathrm{DNEL}} \mathbf{dnel}_2>_{\mathrm{DF}} \\ ]_{\mathbf{fid}}\mathbf{null}>_{\mathrm{G}}\end{array}}{s, d, \mathbf{g}, \mathtt{p} := \mathtt{getParentNode(N)} \rightsquigarrow [s|\mathtt{p} \leftarrow \mathbf{id}], d, \mathbf{g}}$$

getChildNodes This command provides the functionality of the readonly object attribute “childNodes”. It returns the “**fid**” of the children of the node `N`, which can then be passed to any command which [23] specifies as expecting a NodeList object.

[23] describes childNodes as:

Begin Quote

> A NodeList that contains all children of this node. If there are no children, this is a NodeList containing no nodes. The content of the returned NodeList is “live” in the sense that, for instance, changes to the children of the node object that it was created from are immediately reflected in the nodes returned by the NodeList accessors; it is not a static snapshot of the content of the node.

End Quote

Note in particular that it is not possible to have a **null** childNodes list. Even node types which may have no children under any circumstances will always have an empty childNodes list. This means, for example, that in languages which support it, it is possible to safely write programs such as:

```
getGrandChildren(n) = concat (map getChildNodes (getChildNodes(n)))
```

... without worrying about handling **null** cases.

The "live" nature of the returned data structure is handled by returning a reference to the actual child structure of the node `N`. If `N`'s children change, then the elements in the NodeList referred to by **fid** will change by definition.

$$\frac{\mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}, \mathbf{name}_{[\![\mathtt{N}]\!]_s} \{\!| \mathbf{af} |\!\}^{\mathbf{irn}}_{\mathbf{aidn}_{\mathbf{tp}}} [\mathbf{f}]_{\mathbf{fid}} \mathbf{val})}{s, d, \mathbf{g}, \mathtt{kids} := \mathtt{getChildNodes(N)} \rightsquigarrow [s|\mathtt{kids} \leftarrow \mathbf{fid}], d, \mathbf{g}}$$

$$\frac{\mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}, \ll \mathbf{name}_{[\![\mathtt{N}]\!]_s} \mapsto [\mathbf{af}]_{\mathbf{fid}} \gg^{\mathbf{doc}}_{\mathbf{specified}})}{s, d, \mathbf{g}, \mathtt{kids} := \mathtt{getChildNodes(N)} \rightsquigarrow [s|\mathtt{kids} \leftarrow \mathbf{fid}], d, \mathbf{g}}$$

`getAttributes` This command provides the functionality of the read-only object attribute "attributes". It returns the **aid** of the node `N`'s attribute map unless `N` is of a type that cannot have attributes, in which case it returns **null**.

$$\frac{\mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}, \mathbf{name}_{[\![\mathtt{N}]\!]_s} \{\!| \mathbf{af} |\!\}^{\mathbf{irn}}_{\mathbf{aidn}_{\mathbf{tp}}} [\mathbf{f}]_{\mathbf{fid}} \mathbf{val})}{s, d, \mathbf{g}, \mathtt{ats} := \mathtt{getAttributes(N)} \rightsquigarrow [s|\mathtt{ats} \leftarrow \mathbf{aidn}], d, \mathbf{g}}$$

$$\frac{\mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}, \ll \mathbf{name}_{[\![\mathtt{N}]\!]_s} \mapsto [\mathbf{af}]_{\mathbf{fid}} \gg^{\mathbf{doc}}_{\mathbf{specified}})}{s, d, \mathbf{g}, \mathtt{ats} := \mathtt{getAttributes(N)} \rightsquigarrow [s|\mathtt{ats} \leftarrow \mathbf{null}, d, \mathbf{g}}$$

`getOwnerDocument` This command provides the functionality of the read-only object attribute "ownerDocument". All Nodes except Document nodes are owned by a Document node. In the case that `N` is not a Document node, this command returns the **id** of the Document node that owns `N`. In the case that `N` is a Document node, this command returns **null**.

$$\frac{\mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}, \mathbf{name}_{[\![\mathtt{N}]\!]_s} \{\!| \mathbf{af} |\!\}^{\mathbf{irn}}_{\mathbf{aidn}_{\mathbf{tp}}} [\mathbf{f}]_{\mathbf{fid}} \mathbf{val})}{s, d, \mathbf{g}, \mathtt{od} := \mathtt{getOwnerDocument(N)} \rightsquigarrow [s|\mathtt{od} \leftarrow \mathbf{irn}], d, \mathbf{g}}$$

$$\frac{\mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}, \ll \mathbf{name}_{[\![\mathtt{N}]\!]_s} \mapsto [\mathbf{af}]_{\mathbf{fid}} \gg^{\mathbf{doc}}_{\mathbf{specified}})}{s, d, \mathbf{g}, \mathtt{od} := \mathtt{getOwnerDocument(N)} \rightsquigarrow [s|\mathtt{od} \leftarrow \mathbf{doc}, d, \mathbf{g}}$$

`appendChild` The next challenge is the class of commands which move nodes around in the grove. If we provide "`appendChild`" and "`removeChild`", we can implement "insertBefore" and "replaceChild".

[23] says:

Begin Quote

> Adds the node newChild to the end of the list of children of this node. If the newChild is already in the tree, it is first removed.

End Quote

This is a move command. It moves the node `NewChild` from wherever it currently is, to become the last child of the node described in [23] as "this node", and here as `Parent`. If `NewChild` doesn't exist, then the command faults.

First, we present the two possible cases for if the node `Parent` is a DocumentFragment or Element node.

The simplest case for appendChild is:

$$\frac{\begin{array}{l}\mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}', <\mathbf{name}_{[\![\texttt{NewChild}]\!]_s} \{\!| \mathbf{af} |\!\}^{\mathbf{idref}}_{\mathbf{aidn}\,\mathbf{ndftp}} [\mathbf{f}]_{\mathbf{fid}} \mathbf{val} >_{\mathrm{D}'_1}) \\ \mathrm{ap}(\mathbf{cg}', \varnothing_{\mathrm{D}_1}) \equiv \mathrm{ap}(\mathbf{cg}, \mathbf{name}'_{[\![\texttt{Parent}]\!]_s} \{\!| \mathbf{af}' |\!\}^{\mathbf{idref}}_{\mathbf{aidn}'\,\mathbf{dfetp}} [\mathbf{fnd}']_{\mathbf{fid}'} \mathbf{null}) \\ \mathbf{g}' \equiv \mathrm{ap}(\mathbf{cg}, \mathbf{name}'_{[\![\texttt{Parent}]\!]_s} \{\!| \mathbf{af}' |\!\}^{\mathbf{idref}}_{\mathbf{aidn}'\,\mathbf{dfetp}} [\\ \quad \mathbf{fnd}' \otimes_{\mathrm{D}_2} \\ \quad <\mathbf{name}_{[\![\texttt{NewChild}]\!]_s} \{\!| \mathbf{af} |\!\}^{\mathbf{idref}}_{\mathbf{aidn}\,\mathbf{ndftp}} [\mathbf{f}]_{\mathbf{fid}} \mathbf{val} >_{\mathrm{D}_2} \\ ]_{\mathbf{fid}'} \mathbf{null})\end{array}}{s, d, \mathbf{g}, \mathtt{n} := \mathtt{appendChild}(\mathtt{Parent}, \mathtt{NewChild}) \rightsquigarrow [s|\mathtt{n} \leftarrow [\![\mathtt{NewChild}]\!]_s], d, \mathbf{g}'}$$

where $\mathbf{dfetp} \in \{11, 1\}$, ensuring that the `Parent` node is either a DocumentFragment or an Element node, and $\mathbf{ndftp} \in \{1, 3, 8, 9\}$, ensuring that the `NewChild` node is not a DocumentFragment node.

If the new child to be inserted is a document fragment, then we add the forest of the new child, not the child itself.

Begin Quote

> When a DocumentFragment is inserted into a Document (or indeed any other Node that may take children) the children of the DocumentFragment and not the DocumentFragment itself are inserted into the Node. This makes the DocumentFragment very useful when the user wishes to create nodes that are siblings; the DocumentFragment acts as the parent of these nodes

so that the user can use the standard methods from the Node interface, such as insertBefore() and appendChild().

End Quote

To define this command, we make use of a partial function "cast", which transforms a forest from one type into another. For example, it can be used to transform the forest of children of a Document Fragment into an Element forest, which may be appended to the child list of an Element node.

Definition 52 (The cast function)**.** The function "cast" takes a type parameter D and a forest **f**. It returns a new forest of type D which contains the same elements as **f**.

$$\begin{aligned} \mathrm{cast}(\mathrm{D}_1, \varnothing_{\mathrm{D}_2}) &\triangleq \varnothing_{\mathrm{D}_1} \\ \mathrm{cast}(\mathrm{D}_1, \mathbf{d}_2 \otimes_{\mathrm{D}_2} \mathbf{d}_2') &\triangleq \mathrm{cast}(\mathrm{D}_1, \mathbf{d}_2) \otimes_{\mathrm{D}_1} \mathrm{cast}(\mathrm{D}_1, \mathbf{d}_2') \\ \mathrm{cast}(\mathrm{D}_1, <\mathbf{f}>_{\mathrm{D}_2}) &\triangleq <\mathbf{f}>_{\mathrm{D}_1} \end{aligned}$$

$$\begin{array}{l} \mathbf{g} \equiv <\text{“\#document-fragment”}_{[\![\texttt{NewChild}]\!]_s} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathbf{idref}}_{\mathbf{null}\,\mathbf{11}} [\mathbf{f}]_{\mathbf{fid}}\mathbf{null}>_{\mathrm{G}} \oplus \\ \quad \mathrm{ap}(\mathbf{cg}, \mathbf{name}'_{[\![\texttt{Parent}]\!]_s} \{\!\!\{ \mathbf{af}' \}\!\!\}^{\mathbf{idref}}_{\mathbf{aidn}'\,\mathbf{dfetp}} [\mathbf{fnd}']_{\mathbf{fid}'}\mathbf{val}') \\ \mathbf{g}' \equiv \mathrm{ap}(\mathbf{cg}, \mathbf{name}'_{[\![\texttt{Parent}]\!]_s} \{\!\!\{ \mathbf{af}' \}\!\!\}^{\mathbf{idref}}_{\mathbf{aidn}'\,\mathbf{dfetp}} \left[\; \mathbf{fnd}' \otimes_{\mathrm{FND}'} \mathrm{cast}(\mathrm{FND}', \mathbf{f}) \; \right]_{\mathbf{fid}'} \mathbf{val}') \\ \quad \oplus <\text{“\#document-fragment”}_{[\![\texttt{NewChild}]\!]_s} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathbf{idref}}_{\mathbf{null}\,\mathbf{11}} [\varnothing_{\mathrm{FRAGF}}]_{\mathbf{fid}}\mathbf{null}>_{\mathrm{G}} \\ \hline s, d, \mathbf{g}, \mathtt{n} := \mathtt{appendChild(Parent, NewChild)}; \rightsquigarrow [s|\mathtt{n} \leftarrow [\![\texttt{NewChild}]\!]_s], d, \mathbf{g}' \end{array}$$

where $\mathbf{dfetp} \in \{11, 1\}$, ensuring that the `Parent` node is either a DocumentFragment or an Element node and $\mathbf{fnd}' \in \mathrm{FND}'$, ensuring that the new children are cast to the correct type.

Next we present the synonymous cases if the parent is an Attr. Note that because this command modifies the Attr, its value of "specified" will be changed to true, as explained in Chapter 6.3.2.

The simpler case where the node `NewChild` is not a Document Fragment

is given by

$$\frac{\begin{array}{l}\mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}', <\mathbf{name}_{[\![\texttt{NewChild}]\!]_s} \{\!\!\{ \mathbf{af} \}\!\!\}^{\mathbf{idref}}_{\mathbf{aidn}\,\mathbf{ndftp}} [\mathbf{f}]_{\mathbf{fid}} \mathbf{val}>_{\mathrm{D}'_1}) \\ \mathrm{ap}(\mathbf{cg}', \varnothing_{\mathrm{D}'_1}) \equiv \mathrm{ap}(\mathbf{cg}, \ll \mathbf{name}'_{[\![\texttt{Parent}]\!]_s} \mapsto [\mathbf{f}']_{\mathbf{fid}'} \gg^{\mathbf{idref}}_{\mathbf{specified}}) \\ \mathbf{g}' \equiv \mathrm{ap}(\mathbf{cg}, \ll \mathbf{name}'_{[\![\texttt{Parent}]\!]_s} \mapsto [\\ \quad \mathbf{f}' \otimes_{\mathrm{D}_2} \\ \quad <\mathbf{name}_{[\![\texttt{NewChild}]\!]_s} \{\!\!\{ \mathbf{af} \}\!\!\}^{\mathbf{idref}}_{\mathbf{aidn}\,\mathbf{ndftp}} [\mathbf{f}]_{\mathbf{fid}} \mathbf{val}>_{\mathrm{D}'_2} \\ ]_{\mathbf{fid}'} \gg^{\mathbf{idref}}_{\mathbf{true}})\end{array}}{s, d, \mathbf{g}, \mathtt{n} := \mathtt{appendChild}(\mathtt{Parent}, \mathtt{NewChild}); \rightsquigarrow [s|\mathtt{n} \leftarrow [\![\mathtt{NewChild}]\!]_s], d, \mathbf{g}'}$$

where $\mathbf{ndftp} \in \{1, 3, 8, 9\}$, ensuring that the `NewChild` node is not a DocumentFragment node. The case when the child is also a document fragment is given by

$$\frac{\begin{array}{l}\mathbf{g} \equiv <\text{“\#document-fragment”}_{[\![\texttt{NewChild}]\!]_s} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathbf{idref}}_{\mathbf{null}\,\mathbf{11}} [\mathbf{f}]_{\mathbf{fid}} \mathbf{null}>_{\mathrm{G}} \oplus \\ \quad \mathrm{ap}(\mathbf{cg}, \ll \mathbf{name}'_{[\![\texttt{Parent}]\!]_s} \mapsto [\mathbf{f}']_{\mathbf{fid}'} \gg^{\mathbf{idref}}_{\mathbf{specified}}) \\ \mathbf{g}' \equiv \mathrm{ap}(\mathbf{cg}, \ll \mathbf{name}'_{[\![\texttt{Parent}]\!]_s} \mapsto \left[\; \mathbf{f}' \otimes_{\mathrm{EA}} \mathrm{cast}(\mathrm{EA}, \mathbf{f}) \; \right]_{\mathbf{fid}'} \gg^{\mathbf{idref}}_{\mathbf{true}}) \\ \oplus <\text{“\#document-fragment”}_{[\![\texttt{NewChild}]\!]_s} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathbf{idref}}_{\mathbf{null}\,\mathbf{11}} [\varnothing_{\mathrm{FRAGF}}]_{\mathbf{fid}} \mathbf{null}>_{\mathrm{G}}\end{array}}{s, d, \mathbf{g}, \mathtt{n} := \mathtt{appendChild}(\mathtt{Parent}, \mathtt{NewChild}); \rightsquigarrow [s|\mathtt{n} \leftarrow [\![\mathtt{NewChild}]\!]_s], d, \mathbf{g}'}$$

Finally, we present the cases where `Parent` is a Document node. Recall that the forest of a Document node as a triplet of $\mathbf{dnel}, \mathbf{de}, \mathbf{dnel}$. The middle entry in the triplet is restricted to being a single element, or $\varnothing_{\mathrm{DE}}$. The outer entries are true forests, which may not contain Elements. If the middle entry is $\varnothing_{\mathrm{DE}}$, then the right-most entry must be $\varnothing_{\mathrm{DNEL}}$. This property is maintained by these commands. Most particularly, the command `removeChild`, when removing an element from beneath a Document node, will simultaneously move all Comment nodes from the right-most entry in this triplet to the left-most.

First we present the appendChild cases where `Parent` is a Document node and in which one of the children to append – either `NewChild` or if `NewChild`

is a Document Fragment, then it's children – is an element:

$$\frac{\begin{array}{l}\mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}, <\mathbf{s}_{[\![\texttt{NewChild}]\!]_s} \{\!\!\{ \mathbf{af} \}\!\!\}_{\mathbf{aid}\,\mathbf{1}}^{[\![\texttt{Parent}]\!]_s} [\mathbf{f}]_{\mathbf{fid}}\mathbf{null}>_{\mathrm{D}'_1}) \\ \quad \oplus <\text{“\#document”}_{[\![\texttt{Parent}]\!]_s} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{null}\,\mathbf{9}}^{\mathbf{null}} [\mathbf{f}', \varnothing_{\mathrm{DE}}, \varnothing_{\mathrm{DNEL}}]_{\mathbf{fid}'}\mathbf{null}>_{\mathrm{G}} \\ \mathbf{g}' \equiv \mathrm{ap}(\mathbf{cg}, \varnothing_{\mathrm{D}'_1}) \\ \quad \oplus <\text{“\#document”}_{[\![\texttt{Parent}]\!]_s} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{null}\,\mathbf{9}}^{\mathbf{null}} [\\ \qquad \mathbf{f}', \\ \qquad <\mathbf{s}_{[\![\texttt{NewChild}]\!]_s} \{\!\!\{ \mathbf{af} \}\!\!\}_{\mathbf{aid}\,\mathbf{1}}^{[\![\texttt{Parent}]\!]_s} [\mathbf{f}]_{\mathbf{fid}}\mathbf{val}>_{\mathrm{DE}}, \\ \qquad \varnothing_{\mathrm{DNEL}} \\ \quad]_{\mathbf{fid}'}\mathbf{null}>_{\mathrm{G}}\end{array}}{s, d, \mathbf{g}, \mathtt{n} := \mathtt{appendChild}(\mathtt{Parent}, \mathtt{NewChild}); \rightsquigarrow [s|\mathtt{n} \leftarrow [\![\mathtt{NewChild}]\!]_s], d, \mathbf{g}'}$$

$$\frac{\begin{array}{l}\mathbf{g} \equiv \mathbf{g}'' \oplus \\ \quad <\text{“\#document-fragment”}_{[\![\texttt{NewChild}]\!]_s} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{null}\,\mathbf{11}}^{[\![\texttt{Parent}]\!]_s} [\\ \qquad \mathbf{f}_1 \otimes_{\mathrm{FRAGF}} \\ \qquad <\mathbf{s}_{[\![\texttt{NewChild}]\!]_s} \{\!\!\{ \mathbf{af} \}\!\!\}_{\mathbf{aid}\,\mathbf{1}}^{[\![\texttt{Parent}]\!]_s} [\mathbf{f}]_{\mathbf{fid}}\mathbf{null}>_{\mathrm{FRAGF}} \\ \qquad \otimes_{\mathrm{FRAGF}} \mathbf{f}_2 \\ \quad]_{\mathbf{fid}}\mathbf{null}>_{\mathrm{G}} \\ \quad \oplus <\text{“\#document”}_{[\![\texttt{Parent}]\!]_s} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{null}\,\mathbf{9}}^{\mathbf{null}} [\mathbf{f}', \varnothing_{\mathrm{DE}}, \varnothing_{\mathrm{DNEL}}]_{\mathbf{fid}'}\mathbf{null}>_{\mathrm{G}} \\ \mathbf{g}' \equiv \mathbf{g}'' \oplus <\text{“\#document-fragment”}_{[\![\texttt{NewChild}]\!]_s} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{null}\,\mathbf{11}}^{[\![\texttt{Parent}]\!]_s} \left[\ \varnothing_{\mathrm{FRAGF}} \ \right]_{\mathbf{fid}} \mathbf{null}>_{\mathrm{G}} \\ \quad <\text{“\#document”}_{[\![\texttt{Parent}]\!]_s} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{null}\,\mathbf{9}}^{\mathbf{null}} [\\ \qquad \mathbf{f}' \otimes_{\mathrm{DNEL}} \mathrm{cast}(\mathrm{DNEL}, \mathbf{f}_1), \\ \qquad <\mathbf{s}_{[\![\texttt{NewChild}]\!]_s} \{\!\!\{ \mathbf{af} \}\!\!\}_{\mathbf{aid}\,\mathbf{1}}^{[\![\texttt{Parent}]\!]_s} [\mathbf{f}]_{\mathbf{fid}}\mathbf{null}>_{\mathrm{DE}}, \\ \qquad \mathrm{cast}(\mathrm{DNEL}, \mathbf{f}_2) \\ \quad]_{\mathbf{fid}'}\mathbf{null}>_{\mathrm{G}}\end{array}}{s, d, \mathbf{g}, \mathtt{n} := \mathtt{appendChild}(\mathtt{Parent}, \mathtt{NewChild}); \rightsquigarrow [s|\mathtt{n} \leftarrow [\![\mathtt{NewChild}]\!]_s], d, \mathbf{g}'}$$

Next, we present the 4 remaining cases, in which none of the children to append are elements. In these cases, recall that the type number of a comment is 8 and the type number of a Document Fragment is 11. In the first case, `NewChild` is a Comment, and `Parent` does not have an Element child.

$$\frac{\begin{array}{l}\mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}, <\mathbf{name}_{[\![\texttt{NewChild}]\!]_s} \{\!\!\{ \mathbf{ea} \}\!\!\}_{\mathbf{aidn}\,\mathbf{8}}^{[\![\texttt{Parent}]\!]_s} [\mathbf{f}]_{\mathbf{fid}}\mathbf{val}>_{\mathrm{D}_1}) \oplus \\ \quad <\text{“\#document”}_{[\![\texttt{Parent}]\!]_s} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{null}\,\mathbf{9}}^{\mathbf{null}} [\mathbf{dnel}, \varnothing_{\mathrm{DE}}, \varnothing_{\mathrm{DNEL}}]_{\mathbf{fid}'}\mathbf{null}>_{\mathrm{G}} \\ \mathbf{g}' \equiv \mathrm{ap}(\mathbf{cg}, \varnothing_{\mathrm{D}_1}) \oplus \\ \quad <\text{“\#document”}_{[\![\texttt{Parent}]\!]_s} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{null}\,\mathbf{9}}^{\mathbf{null}} [\\ \qquad \mathbf{dnel} \otimes_{\mathrm{DNEL}} \\ \qquad <\mathbf{name}_{[\![\texttt{NewChild}]\!]_s} \{\!\!\{ \mathbf{ea} \}\!\!\}_{\mathbf{aidn}\,\mathbf{8}}^{[\![\texttt{Parent}]\!]_s} [\mathbf{f}]_{\mathbf{fid}}\mathbf{val}>_{\mathrm{DNEL}}, \\ \qquad \varnothing_{\mathrm{DE}}, \varnothing_{\mathrm{DNEL}} \\ \quad]_{\mathbf{fid}'}\mathbf{null}>_{\mathrm{G}}\end{array}}{s, d, \mathbf{g}, \mathtt{n} := \mathtt{appendChild}(\mathtt{Parent}, \mathtt{NewChild}); \rightsquigarrow [s|\mathtt{n} \leftarrow [\![\mathtt{NewChild}]\!]_s], d, \mathbf{g}'}$$

In the second the case, `NewChild` is a Comment and `Parent` does already have an Element child.

$$\frac{\begin{array}{l}\mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}, <\mathbf{name}_{[\![\texttt{NewChild}]\!]_s} \{\!\!\{ \mathbf{ea} \}\!\!\}_{\mathbf{aidn}\,\mathbf{8}}^{[\![\texttt{Parent}]\!]_s} [\mathbf{f}]_{\mathbf{fid}} \mathbf{val}>_{\mathrm{D}_1}) \oplus \\ \quad <\text{“\#document”}_{[\![\texttt{Parent}]\!]_s} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{null}\,\mathbf{9}}^{\mathbf{null}} [\mathbf{dnel}_1, \mathbf{de}, \mathbf{dnel}_2]_{\mathbf{fid}'\,\mathbf{null}}>_{\mathrm{G}} \\ \mathbf{g}' \equiv \mathrm{ap}(\mathbf{cg}, \varnothing_{\mathrm{D}_1}) \oplus \\ \quad <\text{“\#document”}_{[\![\texttt{Parent}]\!]_s} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{null}\,\mathbf{9}}^{\mathbf{null}} [\\ \qquad \mathbf{dnel}_1, \mathbf{de}, \mathbf{dnel}_2 \otimes_{\mathrm{DNEL}} \\ \qquad <\mathbf{name}_{[\![\texttt{NewChild}]\!]_s} \{\!\!\{ \mathbf{ea} \}\!\!\}_{\mathbf{aidn}\,\mathbf{8}}^{[\![\texttt{Parent}]\!]_s} [\mathbf{f}]_{\mathbf{fid}}\,\mathbf{val}>_{\mathrm{DNEL}} \\ \quad]_{\mathbf{fid}'\,\mathbf{null}}>_{\mathrm{G}} \\ \quad \wedge\, \mathbf{dnel}_1 \neq \varnothing_{\mathrm{DNEL}} \wedge \mathbf{de} \neq \varnothing_{\mathrm{DE}}\end{array}}{s, d, \mathbf{g}, \mathtt{n} := \mathtt{appendChild}(\mathtt{Parent}, \mathtt{NewChild}); \rightsquigarrow [s|\mathtt{n} \leftarrow [\![\texttt{NewChild}]\!]_s], d, \mathbf{g}'}$$

In the third case, `NewChild` is a Document Fragment and `Parent` does not have an Element child.

$$\frac{\begin{array}{l}\mathbf{g} \equiv \mathbf{g}'' \oplus \\ \quad <\text{“\#document-fragment”}_{[\![\texttt{NewChild}]\!]_s} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{null}\,\mathbf{11}}^{[\![\texttt{Parent}]\!]_s} [\mathbf{f}]_{\mathbf{fid}}\,\mathbf{null}>_{\mathrm{G}} \\ \quad \oplus <\text{“\#document”}_{[\![\texttt{Parent}]\!]_s} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{null}\,\mathbf{9}}^{\mathbf{null}} [\mathbf{dnel}, \varnothing_{\mathrm{DE}}, \varnothing_{\mathrm{DNEL}}]_{\mathbf{fid}'\,\mathbf{null}}>_{\mathrm{G}} \\ \mathbf{g}' \equiv \mathbf{g}'' \oplus \\ \quad <\text{“\#document-fragment”}_{[\![\texttt{NewChild}]\!]_s} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{null}\,\mathbf{11}}^{[\![\texttt{Parent}]\!]_s} [\varnothing_{\mathrm{FRAGF}}]_{\mathbf{fid}}\,\mathbf{null}>_{\mathrm{G}} \\ \quad \oplus <\text{“\#document”}_{[\![\texttt{Parent}]\!]_s} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{null}\,\mathbf{9}}^{\mathbf{null}} [\mathbf{dnel} \otimes_{\mathrm{DNEL}} \mathrm{cast}(\mathrm{DNEL}, \mathbf{f}), \varnothing_{\mathrm{DE}}, \varnothing_{\mathrm{DNEL}}]_{\mathbf{fid}'\,\mathbf{null}}>_{\mathrm{G}}\end{array}}{s, d, \mathbf{g}, \mathtt{n} := \mathtt{appendChild}(\mathtt{Parent}, \mathtt{NewChild}); \rightsquigarrow [s|\mathtt{n} \leftarrow [\![\texttt{NewChild}]\!]_s], d, \mathbf{g}'}$$

In the fourth case, `NewChild` is a Document Fragment and `Parent` does already have an Element child.

$$\frac{\begin{array}{l}\mathbf{g} \equiv \mathbf{g}'' \oplus \\ \quad <\text{“\#document-fragment”}_{[\![\texttt{NewChild}]\!]_s} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{null}\,\mathbf{11}}^{[\![\texttt{Parent}]\!]_s} [\mathbf{f}]_{\mathbf{fid}}\,\mathbf{null}>_{\mathrm{G}} \\ \quad \oplus <\text{“\#document”}_{[\![\texttt{Parent}]\!]_s} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{null}\,\mathbf{9}}^{\mathbf{null}} [\mathbf{dnel}_1, \mathbf{de}, \mathbf{dnel}_2]_{\mathbf{fid}'\,\mathbf{null}}>_{\mathrm{G}} \\ \mathbf{g}' \equiv \mathbf{g}'' \oplus \\ \quad <\text{“\#document-fragment”}_{[\![\texttt{NewChild}]\!]_s} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{null}\,\mathbf{11}}^{[\![\texttt{Parent}]\!]_s} [\varnothing_{\mathrm{FRAGF}}]_{\mathbf{fid}}\,\mathbf{null}>_{\mathrm{G}} \\ \quad \oplus <\text{“\#document”}_{[\![\texttt{Parent}]\!]_s} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{null}\,\mathbf{9}}^{\mathbf{null}} [\mathbf{dnel}_1, \mathbf{de}, \mathbf{dnel}_2 \otimes_{\mathrm{DNEL}} \mathrm{cast}(\mathrm{DNEL}, \mathbf{f})]_{\mathbf{fid}'\,\mathbf{null}}>_{\mathrm{G}} \\ \wedge\, \mathbf{dnel}_1 \neq \varnothing_{\mathrm{DNEL}} \wedge \mathbf{de} \neq \varnothing_{\mathrm{DE}}\end{array}}{s, d, \mathbf{g}, \mathtt{n} := \mathtt{appendChild}(\mathtt{Parent}, \mathtt{NewChild}); \rightsquigarrow [s|\mathtt{n} \leftarrow [\![\texttt{NewChild}]\!]_s], d, \mathbf{g}'}$$

`removeChild` The command “removeChild” is simpler, since the destination of the move is always the root-level of the grove. In the usual case, we have:

$$\frac{\begin{array}{l}\mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}, \mathbf{name}_{[\![\texttt{Parent}]\!]_s} \{\!\!\{ \mathbf{ea} \}\!\!\}^{\mathbf{idref}}_{\mathbf{aidn}\,\mathbf{dfetp}} [\\ \quad \mathbf{f}_1 \otimes_{\mathrm{D}_2} \\ \quad < \mathbf{name}'_{[\![\texttt{OldChild}]\!]_s} \{\!\!\{ \mathbf{ea}' \}\!\!\}^{\mathbf{idref}}_{\mathbf{aidn}'\,\mathbf{tp}'} [\mathbf{f}']_{\mathbf{fid}'} \mathbf{val}' >_{\mathrm{D}_2} \\ \quad \otimes_{\mathrm{D}_2} \mathbf{f}_2 \\ ]_{\mathbf{fid}} \mathbf{val}) \\ \mathbf{g}' \equiv \mathrm{ap}(\mathbf{cg}, \mathbf{name}_{[\![\texttt{Parent}]\!]_s} \{\!\!\{ \mathbf{ea} \}\!\!\}^{\mathbf{idref}}_{\mathbf{aidn}\,\mathbf{dfetp}} [\mathbf{f}_1 \otimes_{\mathrm{D}_2} \mathbf{f}_2]_{\mathbf{fid}} \mathbf{val}) \\ \quad \oplus < \mathbf{name}'_{[\![\texttt{OldChild}]\!]_s} \{\!\!\{ \mathbf{ea}' \}\!\!\}^{\mathbf{idref}}_{\mathbf{aidn}'\,\mathbf{tp}'} [\mathbf{f}']_{\mathbf{fid}'} \mathbf{val}' >_{\mathrm{G}}\end{array}}{s, d, \mathbf{g}, \mathtt{n} := \mathtt{removeChild(Parent, OldChild)}; \rightsquigarrow [s|\mathtt{n} \leftarrow [\![\texttt{OldChild}]\!]_s], d, \mathbf{g}'}$$

where $\mathbf{dfetp} \in \{11, 1\}$, ensuring that the `Parent` node is either a DocumentFragment or an Element node.

In the case where the old parent node is an attribute we must remember to set "specified" to true as explained in Chapter 6.3.2:

$$\frac{\begin{array}{l}\mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}, \ll \mathbf{name}_{[\![\texttt{Parent}]\!]_s} \mapsto [\\ \quad \mathbf{f}_1 \otimes_{\mathrm{D}_2} \\ \quad < \mathbf{name}'_{[\![\texttt{OldChild}]\!]_s} \{\!\!\{ \mathbf{ea}' \}\!\!\}^{\mathbf{idref}}_{\mathbf{aidn}'\,\mathbf{tp}'} [\mathbf{f}']_{\mathbf{fid}'} \mathbf{val}' >_{\mathrm{D}_2} \\ \quad \otimes_{\mathrm{D}_2} \mathbf{f}_2 \\ ]_{\mathbf{fid}} \gg^{\mathbf{idref}}_{\mathbf{specified}}) \\ \mathbf{g}' \equiv \mathrm{ap}(\mathbf{cg}, \ll \mathbf{name}_{[\![\texttt{Parent}]\!]_s} \mapsto [\mathbf{f}_1 \otimes_{\mathrm{D}_2} \mathbf{f}_2]_{\mathbf{fid}} \gg^{\mathbf{idref}}_{\mathbf{true}}) \\ \quad \oplus < \mathbf{name}'_{[\![\texttt{OldChild}]\!]_s} \{\!\!\{ \mathbf{ea}' \}\!\!\}^{\mathbf{idref}}_{\mathbf{aidn}'\,\mathbf{tp}'} [\mathbf{f}']_{\mathbf{fid}'} \mathbf{val}' >_{\mathrm{G}}\end{array}}{s, d, \mathbf{g}, \mathtt{n} := \mathtt{removeChild(Parent, OldChild)}; \rightsquigarrow [s|\mathtt{n} \leftarrow [\![\texttt{OldChild}]\!]_s], d, \mathbf{g}'}$$

Finally, we present the three cases where the old parent node is a document node. Recall that the forest of a Document node is a triplet of $\mathbf{dnel}, \mathbf{de}, \mathbf{dnel}$. Removing a node from each part of this triplet is handled separately. First we present the case which handles the left-most part.

$$\frac{\begin{array}{l}\mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}, \text{“\#document”}_{[\![\texttt{Parent}]\!]_s} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathbf{null}}_{\mathbf{null_9}}[\\ \quad <\mathbf{f}_1 \otimes_{\mathrm{DNEL}} \\ \quad <\mathbf{name}'_{[\![\texttt{OldChild}]\!]_s} \{\!\!\{ \mathbf{ea}' \}\!\!\}^{[\![\texttt{Parent}]\!]_s}_{\mathbf{tp}'}[\mathbf{f}']_{\mathbf{fid}'}\mathbf{val}'>_{\mathrm{DNEL}} \\ \quad \otimes_{\mathrm{DNEL}} \mathbf{f}_2, \\ \quad \mathbf{de}, \mathbf{dnel}>_{\mathrm{DF}} \\ ]_{\mathbf{fid}}\mathbf{null}) \\ \mathbf{g}' \equiv \mathrm{ap}(\mathbf{cg}, \text{“\#document”}_{[\![\texttt{Parent}]\!]_s} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathbf{null}}_{\mathbf{null_9}}[<\mathbf{f}_1 \otimes_{\mathrm{DNEL}} \mathbf{f}_2, \mathbf{de}, \mathbf{dnel}>_{\mathrm{DF}}]_{\mathbf{fid}}\mathbf{null}) \\ \quad \oplus <\mathbf{name}'_{[\![\texttt{OldChild}]\!]_s} \{\!\!\{ \mathbf{ea}' \}\!\!\}^{[\![\texttt{Parent}]\!]_s}_{\mathbf{tp}'}[\mathbf{f}']_{\mathbf{fid}'}\mathbf{val}'>_{\mathrm{G}}\end{array}}{s, d, \mathbf{g}, \mathtt{n} := \mathtt{removeChild}(\mathtt{Parent}, \mathtt{NewChild}); \rightsquigarrow [s|\mathtt{n} \leftarrow [\![\mathtt{OldChild}]\!]_s], d, \mathbf{g}'}$$

The case which handles the right-most part of the triplet is similar.

$$\frac{\begin{array}{l}\mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}, \text{“\#document”}_{[\![\texttt{Parent}]\!]_s} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathbf{null}}_{\mathbf{null_9}}[\\ \quad <\mathbf{dnel}, \mathbf{de}, \\ \quad \mathbf{f}_1 \otimes_{\mathrm{DNEL}} \\ \quad <\mathbf{name}'_{[\![\texttt{OldChild}]\!]_s} \{\!\!\{ \mathbf{ea}' \}\!\!\}^{[\![\texttt{Parent}]\!]_s}_{\mathbf{tp}'}[\mathbf{f}']_{\mathbf{fid}'}\mathbf{val}'>_{\mathrm{DNEL}} \\ \quad \otimes_{\mathrm{DNEL}} \mathbf{f}_2>_{\mathrm{DF}} \\ ]_{\mathbf{fid}}\mathbf{null}) \\ \mathbf{g}' \equiv \mathrm{ap}(\mathbf{cg}, \text{“\#document”}_{[\![\texttt{Parent}]\!]_s} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathbf{null}}_{\mathbf{null_9}}[<\mathbf{dnel}, \mathbf{de}, \mathbf{f}_1 \otimes_{\mathrm{DNEL}} \mathbf{f}_2>_{\mathrm{DF}}]_{\mathbf{fid}}\mathbf{null}) \\ \quad \oplus <\mathbf{name}'_{[\![\texttt{OldChild}]\!]_s} \{\!\!\{ \mathbf{ea}' \}\!\!\}^{[\![\texttt{Parent}]\!]_s}_{\mathbf{tp}'}[\mathbf{f}']_{\mathbf{fid}'}\mathbf{val}'>_{\mathrm{G}}\end{array}}{s, d, \mathbf{g}, \mathtt{n} := \mathtt{removeChild}(\mathtt{Parent}, \mathtt{NewChild}); \rightsquigarrow [s|\mathtt{n} \leftarrow [\![\mathtt{OldChild}]\!]_s], d, \mathbf{g}'}$$

Recall that if the middle entry is $\varnothing_{\mathrm{DE}}$, then the right-most entry must be $\varnothing_{\mathrm{DNEL}}$. We maintain this property in this command by ensuring that whenever the element in the middle is removed, all the Comment nodes from the right entry are simultaneously moved into the left entry.

$$\frac{\begin{array}{l}\mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}, \text{“\#document”}_{[\![\texttt{Parent}]\!]_s} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathbf{null}}_{\mathbf{null_9}}[\\ \quad <\mathbf{dnel}_1, \\ \quad <\mathbf{name}'_{[\![\texttt{OldChild}]\!]_s} \{\!\!\{ \mathbf{ea}' \}\!\!\}^{[\![\texttt{Parent}]\!]_s}_{\mathbf{id}'\mathbf{1}}[\mathbf{f}']_{\mathbf{fid}'}\mathbf{null}>_{\mathrm{DE}}, \\ \quad \mathbf{dnel}_2>_{\mathrm{DF}} \\ ]_{\mathbf{fid}}\mathbf{null}) \\ \mathbf{g}' \equiv \mathrm{ap}(\mathbf{cg}, \text{“\#document”}_{[\![\texttt{Parent}]\!]_s} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathbf{null}}_{\mathbf{null_9}}[<\mathbf{dnel}_1 \otimes_{\mathrm{DNEL}} \mathbf{dnel}_2, \varnothing_{\mathrm{DE}}, \varnothing_{\mathrm{DNEL}}>_{\mathrm{DF}}]_{\mathbf{fid}}\mathbf{null}) \\ \quad \oplus <\mathbf{name}'_{[\![\texttt{OldChild}]\!]_s} \{\!\!\{ \mathbf{ea}' \}\!\!\}^{[\![\texttt{Parent}]\!]_s}_{\mathbf{id}'\mathbf{1}}[\mathbf{f}']_{\mathbf{fid}'}\mathbf{null}>_{\mathrm{G}}\end{array}}{s, d, \mathbf{g}, \mathtt{n} := \mathtt{removeChild}(\mathtt{Parent}, \mathtt{NewChild}); \rightsquigarrow [s|\mathtt{n} \leftarrow [\![\mathtt{OldChild}]\!]_s], d, \mathbf{g}'}$$

NodeList

The NodeList interface only provides one vital command, and that is “item”.

Definition 53. The NodeList Interface commands are:

$$\mathtt{C}_{\mathrm{NODELIST}} ::= \mathtt{n} := \mathtt{item}(\mathtt{List}, \mathtt{Int})$$

This command returns a reference to the ith element of a given list. We have several different kinds of lists, over which we present the command "item" as a unified interface. Recall the definition of the function "len" given in Definition 44. We use that function in the following definitions.

All nodes have a NodeList of children as a part of their structure. These nodes are of the shape:

$$\mathbf{name_{id}} \{\!\!\{ \mathbf{af} \}\!\!\}^{\mathbf{irn}}_{\mathbf{aidn_{tp}}} [\mathbf{f}]_{[\![\texttt{List}]\!]_s} \mathbf{val}$$

where $\mathbf{aidn}, \mathbf{irn} \in \mathrm{ID} \cup \{\mathbf{null}\}$, $\mathbf{id} \in \mathrm{ID}$, $\mathbf{val} \in \mathrm{S} \cup \{\mathbf{null}\}$, $\mathbf{name} \in \mathrm{S}$, $\mathbf{tp} \in \{9, 11, 1, 8, 3\}$

The NodeList containing the children of a node of this shape can be accessed with the command defined by the following two rules:

$$\frac{\begin{array}{l}\mathrm{len}(\mathbf{f}_1) = [\![\texttt{Int}]\!]_s \wedge \\ \mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}, \mathbf{name_{id}} \{\!\!\{ \mathbf{ea} \}\!\!\}^{\mathbf{idref}}_{\mathbf{aidn_{tp}}} [\\ \quad \mathbf{f}_1 \otimes_{\mathrm{D}_2} \\ \quad < \mathbf{name'_{id'}} \{\!\!\{ \mathbf{ea'} \}\!\!\}^{\mathbf{idref}}_{\mathbf{aidn'_{tp'}}} [\mathbf{f'}]_{\mathbf{fid'}} \mathbf{val'} >_{\mathrm{D}_2} \\ \quad \otimes_{\mathrm{D}_2} \mathbf{f}_2 \\ ]_{[\![\texttt{List}]\!]_s} \mathbf{val})\end{array}}{s, d, \mathbf{g}, \mathtt{n} := \mathtt{item}(\mathtt{List}, \mathtt{Int}) \rightsquigarrow [s|\mathtt{n} \leftarrow \mathbf{id'}], d, \mathbf{g}}$$

$$\frac{\begin{array}{l}(\mathrm{len}(\mathbf{f}) \leq [\![\texttt{Int}]\!]_s \vee [\![\texttt{Int}]\!]_s < 0) \wedge \\ \mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}, \mathbf{name_{id}} \{\!\!\{ \mathbf{ea} \}\!\!\}^{\mathbf{idref}}_{\mathbf{aidn_{tp}}} [\mathbf{f}]_{[\![\texttt{List}]\!]_s} \mathbf{val})\end{array}}{s, d, \mathbf{g}, \mathtt{n} := \mathtt{item}(\mathtt{List}, \mathtt{Int}) \rightsquigarrow [s|\mathtt{n} \leftarrow \mathbf{null}], d, \mathbf{g}}$$

Attributes also have a NodeList as part of their structure:

$$\ll \mathbf{name_{id}} \mapsto [\mathbf{af}]_{[\![\texttt{List}]\!]_s} \gg^{\mathbf{doc}}_{\mathbf{specified}}$$

The corresponding command to access the list of children of an Attribute is similar to the usual node case:

$$\frac{\begin{array}{l}\mathrm{len}(\mathbf{f}_1) = [\![\texttt{Int}]\!]_s \wedge \\ \mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}, \ll \mathbf{name}_{\mathbf{id}} \mapsto [\\ \quad \mathbf{f}_1 \otimes_{\mathrm{D}_2} \\ \quad <\mathbf{name}'_{\mathbf{id}'} \{\!\!\{ \mathbf{ea}' \}\!\!\}^{\mathbf{idref}}_{\mathbf{aidn}'\mathbf{tp}'} [\mathbf{f}']_{\mathbf{fid}'} \mathbf{val}' >_{\mathrm{D}_2} \\ \quad \otimes_{\mathrm{D}_2} \mathbf{f}_2 \\ ]_{[\![\texttt{List}]\!]_s} \gg^{\mathbf{idref}}_{\mathbf{specified}})\end{array}}{s, d, \mathbf{g}, \mathtt{n} := \mathtt{item}(\mathtt{List}, \mathtt{Int}) \rightsquigarrow [s|\mathtt{n} \leftarrow \mathbf{id}'], d, \mathbf{g}}$$

$$\frac{\begin{array}{l}(\mathrm{len}(\mathbf{f}) \leq [\![\texttt{Int}]\!]_s \vee [\![\texttt{Int}]\!]_s < 0) \wedge \\ \mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}, \ll \mathbf{name}_{\mathbf{id}} \mapsto [\mathbf{f}]_{[\![\texttt{List}]\!]_s} \gg^{\mathbf{idref}}_{\mathbf{specified}})\end{array}}{s, d, \mathbf{g}, \mathtt{n} := \mathtt{item}(\mathtt{List}, \mathtt{Int}) \rightsquigarrow [s|\mathtt{n} \leftarrow \mathbf{null}], d, \mathbf{g}}$$

Documents are a little more complex, because of their more complex forest structure. None-the-less, the rules follow the same basic pattern. First, we present the case when the node to be accessed is in the left part of the structure.

$$\frac{\begin{array}{l}\mathrm{len}(\mathbf{f}_1) = [\![\texttt{Int}]\!]_s \wedge \\ \mathbf{g} \equiv \mathbf{g}' \oplus <\text{“\#document”}_{\mathbf{id}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathbf{null}}_{\mathbf{g}} [\\ \quad <\mathbf{f}_1 \otimes_{\mathrm{DNEL}} \\ \quad <\mathbf{name}'_{\mathbf{id}'} \{\!\!\{ \mathbf{ea}' \}\!\!\}^{\mathbf{idref}}_{\mathbf{aidn}'\mathbf{tp}'} [\mathbf{f}']_{\mathbf{fid}'} \mathbf{val}' >_{\mathrm{DNEL}} \\ \quad \otimes_{\mathrm{DNEL}} \mathbf{f}_2, \mathbf{de}, \mathbf{dnel} >_{\mathrm{DF}} \\ ]_{[\![\texttt{List}]\!]_s} \mathbf{null} >_{\mathrm{G}}\end{array}}{s, d, \mathbf{g}, \mathtt{n} := \mathtt{item}(\mathtt{List}, \mathtt{Int}) \rightsquigarrow [s|\mathtt{n} \leftarrow \mathbf{id}'], d, \mathbf{g}}$$

Next, we present the case when the node to be accessed is in the central part of the structure.

$$\frac{\begin{array}{l}\mathrm{len}(\mathbf{dnel}_1) = [\![\texttt{Int}]\!]_s \wedge \\ \mathbf{g} \equiv \mathbf{g}' \oplus <\text{“\#document”}_{\mathbf{id}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathbf{null}}_{\mathbf{g}} [\\ \quad <\mathbf{dnel}_1, \\ \quad <\mathbf{name}'_{\mathbf{id}'} \{\!\!\{ \mathbf{ea}' \}\!\!\}^{\mathbf{idref}}_{\mathbf{aidn}'\mathbf{tp}'} [\mathbf{f}']_{\mathbf{fid}'} \mathbf{val}' >_{\mathrm{DE}}, \\ \quad \mathbf{dnel}_2 >_{\mathrm{DF}} \\ ]_{[\![\texttt{List}]\!]_s} \mathbf{null} >_{\mathrm{G}}\end{array}}{s, d, \mathbf{g}, \mathtt{n} := \mathtt{item}(\mathtt{List}, \mathtt{Int}) \rightsquigarrow [s|\mathtt{n} \leftarrow \mathbf{id}'], d, \mathbf{g}}$$

Next, we present the case when the node to be accessed is in the right part of the structure.

$$\frac{\begin{array}{l}\text{len}(\mathbf{dnel}) + \text{len}(\mathbf{de}) + \text{len}(\mathbf{f}_1) = [\![\texttt{Int}]\!]_s \wedge \\ \mathbf{g} \equiv \mathbf{g}' \oplus <\text{“\#document”}_{\mathbf{id}} \{\!\!\{ \varnothing_{\text{EA}} \}\!\!\}^{\mathbf{null}}_{\mathbf{null_9}}[\\ \quad <\mathbf{dnel}, \mathbf{de}, \mathbf{f}_1 \otimes_{\text{DNEL}} \\ \quad <\mathbf{name}'_{\mathbf{id}'} \{\!\!\{ \mathbf{ea}' \}\!\!\}^{\mathbf{idref}}_{\mathbf{aidn}'\mathbf{tp}'}[\mathbf{f}']_{\mathbf{fid}'}\mathbf{val}'>_{\text{DNEL}} \\ \quad \otimes_{\text{DNEL}} \mathbf{f}_2>_{\text{DF}} \\ ]_{[\![\texttt{List}]\!]_s}\mathbf{null}>_{\text{G}}\end{array}}{s, d, \mathbf{g}, \mathtt{n} := \texttt{item}(\texttt{List}, \texttt{Int}) \rightsquigarrow [s|\mathtt{n} \leftarrow \mathbf{id}'], d, \mathbf{g}}$$

Finally, the case in which the index is out of range.

$$\frac{\begin{array}{l}(\text{len}(\mathbf{dnel}_1) + \text{len}(\mathbf{de}) + \text{len}(\mathbf{dnel}_2) \leq [\![\texttt{Int}]\!]_s \vee [\![\texttt{Int}]\!]_s < 0) \wedge \\ \mathbf{g} \equiv \mathbf{g} \oplus <\text{“\#document”}_{\mathbf{id}} \{\!\!\{ \varnothing_{\text{EA}} \}\!\!\}^{\mathbf{null}}_{\mathbf{null_9}}[<\mathbf{dnel}_1, \mathbf{de}, \mathbf{dnel}_2>_{\text{DF}}]_{[\![\texttt{List}]\!]_s}\mathbf{null}>_{\text{G}})\end{array}}{s, d, \mathbf{g}, \mathtt{n} := \texttt{item}(\texttt{List}, \texttt{Int}) \rightsquigarrow [s|\mathtt{n} \leftarrow \mathbf{null}], d, \mathbf{g}}$$

Recall the Element Search structure described in Section 6.2.8, and the `getElementsByTagName` command described in section 6.3.2. The command `getElementsByTagName` does not perform a search when it is called. Instead, it saves the parameters of the search, in a new Element Search structure so that the search may be performed lazily whenever the structure is accessed by the `item` command. The Element Search structure has the following shape:

$$\mathbf{id}\ \mathbf{pattern}_{[\![\texttt{List}]\!]_s}$$

where **pattern** $\in$ S is the string that we will search for, **id** identifies the root of the tree we will search in, and $[\![\texttt{List}]\!]_s$ identifies this Element Search structure itself. We define the `item` command over the Element Search data structure using a new auxiliary function “dosearch”, to perform a “Document Order Search”. The effect of calling `item` on an Element Search object will be to perform a document order search that constructs a forest of items, and then to select the appropriate item from that forest.

Definition 54 (The “dosearch” function)**.** The “dosearch” function is given in Figure 6.7.

With this function, we can define:

$$
\begin{array}{rl}
\text{dosearch}(\mathbf{s}, \mathbf{s_{id}} \lessdot \mathbf{ea} \gtrdot_{\mathbf{aid_1}}^{\mathbf{idref}} [\mathbf{f}]_{\mathbf{fid}} \mathbf{null}) & \triangleq <\mathbf{s_{id}} \lessdot \mathbf{ea} \gtrdot_{\mathbf{aid_1}}^{\mathbf{idref}} [\mathbf{f}]_{\mathbf{fid}} \mathbf{null}>_{\text{EF}} \otimes_{\text{EF}} \text{dosearch}(\mathbf{s}, \mathbf{f}) \\
\text{dosearch}(\mathbf{*}, \mathbf{s_{id}} \lessdot \mathbf{ea} \gtrdot_{\mathbf{aid_1}}^{\mathbf{idref}} [\mathbf{f}]_{\mathbf{fid}} \mathbf{null}) & \triangleq <\mathbf{s_{id}} \lessdot \mathbf{ea} \gtrdot_{\mathbf{aid_1}}^{\mathbf{idref}} [\mathbf{f}]_{\mathbf{fid}} \mathbf{null}>_{\text{EF}} \otimes_{\text{EF}} \text{dosearch}(\mathbf{s}, \mathbf{f}) \\
\text{dosearch}(\mathbf{s}, \mathbf{name_{id}} \lessdot \mathbf{ea} \gtrdot_{\mathbf{aidn_{tp}}}^{\mathbf{irn}} [\mathbf{f}]_{\mathbf{fid}} \mathbf{val}) & \triangleq \text{dosearch}(\mathbf{s}, \mathbf{f}) \quad \text{where } (\mathbf{s} \neq \mathbf{name} \wedge \mathbf{s} \neq \mathbf{*}) \vee \mathbf{tp} \neq 1 \\
\text{dosearch}(\mathbf{s}, <\mathbf{d}_1>_{\text{D}_2}) & \triangleq \text{dosearch}(\mathbf{s}, \mathbf{d}_1) \\
\text{dosearch}(\mathbf{s}, \mathbf{d}_2 \otimes_{\text{D}_2} \mathbf{d}_2') & \triangleq \text{dosearch}(\mathbf{s}, \mathbf{d}_2) \otimes_{\text{EF}} \text{dosearch}(\mathbf{s}, \mathbf{d}_2') \\
\text{dosearch}(\mathbf{s}, \varnothing_{\text{D}_2}) & \triangleq \varnothing_{\text{D}_2}
\end{array}
$$

Figure 6.7.: The "dosearch" Function

$$
\frac{\begin{array}{l}
\mathbf{g} \equiv <_{\mathbf{pattern}_{[\![\texttt{List}]\!]_s}}^{\mathbf{id}}>_{\text{G}} \oplus \text{ap}(\mathbf{cg}, \mathbf{name_{id}} \lessdot \mathbf{ea} \gtrdot_{\mathbf{aidn_{tp}}}^{\mathbf{irn}} [\mathbf{f}]\mathbf{val}) \\
\wedge \text{dosearch}(\mathbf{pattern}, \mathbf{f}) \equiv \mathbf{f}_1 \otimes_{\text{EF}} <\mathbf{s_{id'}} \lessdot \mathbf{ea'} \gtrdot_{\mathbf{aid'_1}}^{\mathbf{idref}} [\mathbf{f}]_{\mathbf{fid'}} \mathbf{null}>_{\text{EF}} \otimes_{\text{EF}} \mathbf{f}_2 \\
\wedge \text{len}(\mathbf{f}_1) = [\![\texttt{Int}]\!]_s
\end{array}}{s, d, \mathbf{g}, \texttt{n} := \texttt{item}(\texttt{List}, \texttt{Int}) \rightsquigarrow [s|\texttt{n} \leftarrow \mathbf{id'}], d, \mathbf{g}}
$$

$$
\frac{\begin{array}{l}
\mathbf{g} \equiv <_{\mathbf{pattern}_{[\![\texttt{List}]\!]_s}}^{\mathbf{id}}>_{\text{G}} \oplus \text{ap}(\mathbf{cg}, \mathbf{name_{id}} \lessdot \mathbf{ea} \gtrdot_{\mathbf{aidn_{tp}}}^{\mathbf{irn}} [\mathbf{f}]\mathbf{val}) \\
\wedge \text{dosearch}(\mathbf{pattern}, \mathbf{f}) \equiv \mathbf{f'} \\
\wedge (\text{len}(\mathbf{f'}) \leq [\![\texttt{Int}]\!]_s \vee [\![\texttt{Int}]\!]_s < 0)
\end{array}}{s, d, \mathbf{g}, \texttt{n} := \texttt{item}(\texttt{List}, \texttt{Int}) \rightsquigarrow [s|\texttt{n} \leftarrow \mathbf{null}], d, \mathbf{g}}
$$

Element

The Element interface defines only one essential method.

Definition 55 (Element Interface Commands)**.** The Element Interface Commands are:

$$
\texttt{C}_{ELE} ::= \texttt{x} := \texttt{getElementsByTagName}(\texttt{Ele}, \texttt{TagName})
$$

Since there are no classes in [23] which subclass Element, we can be specific about the shape of the node `Ele`. The node `Ele` is an Element with a type number of 1, a **null** value and a non-**null** attribute list. We define the `getElementsByTagName` command as follows:

$$
\frac{\begin{array}{l}
\mathbf{g} \equiv \text{ap}(\mathbf{cg}, \mathbf{s}_{[\![\texttt{Ele}]\!]_s} \lessdot \mathbf{ea} \gtrdot_{\mathbf{aid_1}}^{\mathbf{idref}} [\mathbf{f}]_{\mathbf{fid}} \mathbf{null}) \\
\mathbf{g'} \equiv \mathbf{g} \oplus <_{[\![\texttt{TagName}]\!]_s \mathbf{id}}^{[\![\texttt{Ele}]\!]_s}>_{\text{G}}
\end{array}}{s, d, \mathbf{g}, \texttt{x} := \texttt{getElementsByTagName}(\texttt{Ele}, \texttt{TagName}) \rightsquigarrow [s|\texttt{x} \leftarrow \mathbf{id}], d, \mathbf{g'}}
$$

This command is essentially the same as the one described for Document nodes in Chapter 6.3.2. The only difference is in the type of the node that becomes the root of the search.

Attr

There are no classes in [23] which subclass Attr, and there is just one vital behaviour which is not covered by the Node interface. That behaviour is that of the read-only object attribute "specified" the meaning of which is discussed in Chapter 6.3.2. We specify that behaviour here with a simple "getter" command.

Definition 56 (Attr Interface Commands)**.** The Attr Interface Commands are:

$$\mathtt{C}_{\mathrm{ATTR}} ::= \mathtt{sp} := \mathtt{getSpecified}(\mathtt{N})$$

The operational rule for `getSpecified` is:

$$\frac{\mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}, \ll \mathbf{name}_{[\![\mathtt{N}]\!]_s} \mapsto [\mathbf{ea}]_{\mathbf{fid}} \gg^{\mathbf{idref}}_{\mathbf{specified}})}{s, d, \mathbf{g}, \mathtt{sp} := \mathtt{getSpecified}(\mathtt{N}) \rightsquigarrow [s|\mathtt{sp} \leftarrow \mathbf{specified}], d, \mathbf{g}}$$

NamedNodeMap

NamedNodeMap is another collection like NodeList. NamedNodeMaps exist as part of Element nodes, to store a list of Attrs. As a result, the first parameter "`Map`" of any command in this section will be an ID, identifying the NamedNodeMap in an Element (with type number 1 and a **null** value) that looks like this:

$$\mathbf{name}_{\mathbf{id}} \{\!\!\{ \mathbf{ea} \}\!\!\}^{\mathbf{idref}}_{[\![\mathtt{Map}]\!]_s \mathbf{1}} [\mathbf{f}]_{\mathbf{fid}} \mathbf{null}$$

where $\mathbf{id}, \mathbf{fid} \in \mathrm{ID}$

The NamedNodeMap interface defines 3 essential commands.

Definition 57 (NamedNodeMap Interface Commands)**.** The NamedNodeMap Interface Commands are:

$$\begin{aligned}\mathtt{C}_{\mathrm{MAP}} ::= {} & \mathtt{n} := \mathtt{item}(\mathtt{Map}, \mathtt{Int}) \\ | \ & \mathtt{n} := \mathtt{setNamedItem}(\mathtt{Map}, \mathtt{Arg}) \\ | \ & \mathtt{n} := \mathtt{removeNamedItem}(\mathtt{Map}, \mathtt{Name})\end{aligned}$$

`item` As with NodeList, the most vital method is `item`:

$$\frac{\begin{array}{l}\mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}, \mathbf{name_{id}} \{ \\ \quad \mathbf{ea}_1 \otimes_{\mathrm{EA}} <\!\!\ll \mathbf{name'_{id'}} \mapsto [\mathbf{f'}] \gg^{\mathbf{idref}}_{\mathbf{specified}} >_{\mathrm{EA}} \otimes_{\mathrm{EA}} \mathbf{ea}_2 \\ \} \, {}^{\mathbf{idref}}_{\llbracket \mathtt{Map} \rrbracket_s \mathbf{1}} [\mathbf{f}]_{\mathbf{fid}} \mathbf{null}) \\ \wedge \, \mathrm{len}(\mathbf{ea}_1) = \llbracket \mathtt{Int} \rrbracket_s\end{array}}{s, d, \mathbf{g}, \mathtt{n} := \mathtt{item}(\mathtt{Map}, \mathtt{Int}) \rightsquigarrow [s|\mathtt{n} \leftarrow \mathbf{id'}], d, \mathbf{g}}$$

$$\frac{\begin{array}{l}\mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}, \mathbf{name_{id}} \{ \, \mathbf{ea} \, \} \, {}^{\mathbf{idref}}_{\llbracket \mathtt{Map} \rrbracket_s \mathbf{1}} [\mathbf{f}]_{\mathbf{fid}} \mathbf{null}) \\ \wedge \, (\mathrm{len}(\mathbf{ea}) \leq \llbracket \mathtt{Int} \rrbracket_s \vee \llbracket \mathtt{Int} \rrbracket_s < 0)\end{array}}{s, d, \mathbf{g}, \mathtt{n} := \mathtt{item}(\mathtt{Map}, \mathtt{Int}) \rightsquigarrow [s|\mathtt{n} \leftarrow \mathbf{null}], d, \mathbf{g}}$$

`setNamedItem` The command "setNamedItem" is a move command, which moves a node into a namedNodeMap, possibly replacing a node of the same name. This command returns the ID of any node it replaces, or **null** if there was no node of the same name in the `Map`. Notice that when we "remove" an attribute from an element to make way for a new node of the same name, we do not destroy it. We move the "removed" node to the Grove level, where it may be garbage-collected at a later time. First we give the case in which there is no node of the same name in the target `Map`.

$$\frac{\begin{array}{l}\mathbf{g} \equiv \mathrm{ap}(\mathbf{cg'}, <\!\!\ll \mathbf{name}_{\llbracket \mathtt{Arg} \rrbracket_s} \mapsto [\mathbf{f}]_{\mathbf{fid}} \gg^{\mathbf{idref}}_{\mathbf{specified}} >_{\mathrm{D}'_1}) \\ \mathrm{ap}(\mathbf{cg'}, \varnothing_{\mathrm{EA}}) \equiv \mathrm{ap}(\mathbf{cg}, \mathbf{name'_{id'}} \{ \, \mathbf{ea} \, \} \, {}^{\mathbf{idref}}_{\llbracket \mathtt{Map} \rrbracket_s \mathbf{tp'}} [\mathbf{f}]_{\mathbf{fid'}} \mathbf{val'}) \\ \mathbf{g'} \equiv \mathrm{ap}(\mathbf{cg}, \mathbf{name'_{id'}} \{ \, \mathbf{ea} \otimes_{\mathrm{EA}} <\!\!\ll \mathbf{name}_{\llbracket \mathtt{Arg} \rrbracket_s} \mapsto [\mathbf{f}]_{\mathbf{fid}} \gg^{\mathbf{idref}}_{\mathbf{true}} >_{\mathrm{EA}} \} \, {}^{\mathbf{idref}}_{\llbracket \mathtt{Map} \rrbracket_s \mathbf{tp'}} [\mathbf{f}]_{\mathbf{fid'}} \mathbf{val'})\end{array}}{s, d, \mathbf{g}, \mathtt{n} := \mathtt{setNamedItem}(\mathtt{Map}, \mathtt{Arg}) \rightsquigarrow [s|\mathtt{n} \leftarrow \mathbf{null}], d, \mathbf{g}}$$

Note that since we know that $\mathbf{g'}$ contains $\{ \, \mathbf{ea} \otimes_{\mathrm{EA}} <\!\!\ll \mathbf{name}_{\llbracket \mathtt{Arg} \rrbracket_s} \mapsto [\mathbf{f}]_{\mathbf{fid}} \gg^{\mathbf{idref}}_{\mathbf{true}} >_{\mathrm{EA}} \} \, {}^{\mathbf{idref}}_{\llbracket \mathtt{Map} \rrbracket_s}$, and since the DOM data structure demands that Attr nodes in these structures be sibling unique, it follows that there is no Attr node with name **name** in **ea**.

Next we give the case in which there is already a node of the same name in the target `Map`.

$$\mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}', <\ll \mathbf{name}_{[\![\texttt{Arg}]\!]_s} \mapsto [\mathbf{f}]_{\mathbf{fid}} \gg^{\mathbf{idref}}_{\mathbf{specified}} >_{\mathrm{D}'_1})$$
$$\mathrm{ap}(\mathbf{cg}', \varnothing_{\mathrm{EA}}) \equiv \mathrm{ap}(\mathbf{cg}, \mathbf{name}'_{\mathbf{id}'} \nleftarrow$$
$$\mathbf{ea}_1 \otimes_{\mathrm{EA}}$$
$$<\ll \mathbf{name}_{\mathbf{id}'''} \mapsto [\mathbf{f}''']_{\mathbf{fid}'''} \gg^{\mathbf{idref}}_{\mathbf{specified}} >_{\mathrm{EA}}$$
$$\otimes_{\mathrm{EA}} \mathbf{ea}_2$$
$$\nrightarrow^{\mathbf{idref}}_{[\![\texttt{Map}]\!]_s \mathbf{tp}'} [\mathbf{f}]_{\mathbf{fid}'} \mathbf{val}')$$
$$\mathbf{g}' \equiv \mathrm{ap}(\mathbf{cg}, \mathbf{name}'_{\mathbf{id}'} \nleftarrow \mathbf{ea}_1 \otimes_{\mathrm{EA}} \mathbf{ea}_2 \otimes_{\mathrm{EA}} <\ll \mathbf{name}_{[\![\texttt{Arg}]\!]_s} \mapsto [\mathbf{f}]_{\mathbf{fid}} \gg^{\mathbf{idref}}_{\mathbf{true}} >_{\mathrm{EA}} \nrightarrow^{\mathbf{idref}}_{[\![\texttt{Map}]\!]_s \mathbf{tp}'} [\mathbf{f}]_{\mathbf{fid}'} \mathbf{val}') \oplus$$
$$<\ll \mathbf{name}_{\mathbf{id}'''} \mapsto [\mathbf{f}''']_{\mathbf{fid}'''} \gg^{\mathbf{idref}}_{\mathbf{true}} >_{\mathrm{G}}$$

$$s, d, \mathbf{g}, \mathtt{n} := \mathtt{setNamedItem}(\mathtt{Map}, \mathtt{Arg}) \rightsquigarrow [s|\mathtt{n} \leftarrow \mathbf{id}'''], d, \mathbf{g}$$

As with all commands that alter Attr nodes, this command always sets "specified" to true as explained in Chapter 6.3.2.

`removeNamedItem` The command "`removeNamedItem`" is specified in [23] as follows:

Begin Quote

```
removeNamedItem
    Removes a node specified by name. If the removed node is an
    Attr with a default value it is immediately replaced.

Parameters
        name                The name of a node to remove.

Return Value
        The node removed from the map or null if no node with
        such a name exists.

Exceptions
        DOMException
                NOT_FOUND_ERR: Raised if there is no node named
                name in the map.
```

End Quote

Note that in the case where there is no node named "name" in the map, this specification appears to provide two possible inconsistent behaviours:

- It may return **null**
- It may throw the DOMException "NOT_FOUND_ERR".

In this work, we will specify that in that case, the command should fault. This is how we represent all instances in which a command may throw an exception, which is one of the two possible readings of the above quote. This interpretation is not supported by the following quote, from elsewhere in the document:

Begin Quote

> In general, DOM methods return specific error values in ordinary processing situation, such as out-of-bound errors when using NodeList.

End Quote

This suggests that perhaps the preferred reading of the first quote is the **null** option. However, [23] is sufficiently vague on this matter that we feel justified in hedging out bets against both possible outcomes. By specifying that in this case our command should fault, we ensure that our fault-avoiding reasoning (presented in Chapter 7) will make no guarantees of any kind about programs that encounter this case. Any program that we prove will therefore work on any implementation of DOM, regardless of how the author of that implementation may have read this particular section of [23].

This command is further complicated by the behaviour of the "specified" object attribute of the Attr nodes. As explained in Chapter 6.3.2: if an Attr node is removed from a NamedNodeMap, and if the name of the Attr node together with its parent Element appear in the DTD fragment d, then after the Attr has been removed from the NamedNodeMap, a brand new one will be created with that default value, and inserted into the NamedNodeMap.

First, we give the case in which no default value can be found in d.

$$\frac{\begin{array}{l}(\mathbf{s}, [\![\texttt{Name}]\!]_s) \notin \mathrm{dom}(d) \wedge \\ \mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}, \mathbf{s_{id}} \{ \\ \quad \mathbf{ea}_1 \otimes_{\mathrm{EA}} <\ll [\![\texttt{Name}]\!]_{s\,\mathbf{id}'} \mapsto [\mathbf{f}]_{\mathbf{fid}'} \gg^{\mathbf{idref}}_{\mathbf{specified}} >_{\mathrm{EA}} \otimes_{\mathrm{EA}} \mathbf{ea}_2 \\ \} \rangle^{\mathbf{idref}}_{[\![\texttt{Map}]\!]_s \mathbf{1}} [\mathbf{f}]_{\mathbf{fid}} \mathbf{null}) \\ \mathbf{g}' \equiv \mathrm{ap}(\mathbf{cg}, \mathbf{s_{id}} \{ \\ \quad \mathbf{ea}_1 \otimes_{\mathrm{EA}} \mathbf{ea}_2 \\ \} \rangle^{\mathbf{idref}}_{[\![\texttt{Map}]\!]_s \mathbf{1}} [\mathbf{f}]_{\mathbf{fid}} \mathbf{null}) \oplus <\ll [\![\texttt{Name}]\!]_{s\,\mathbf{id}'} \mapsto [\mathbf{f}]_{\mathbf{fid}} \gg^{\mathbf{idref}}_{\mathbf{specified}} >_{\mathrm{G}}\end{array}}{s, d, \mathbf{g}, \texttt{n} := \texttt{removeNamedItem}(\texttt{Map}, \texttt{Name}) \rightsquigarrow [s|\texttt{n} \leftarrow \mathbf{id}'], d, \mathbf{g}'}$$

Next, we give the case in which a default value "**defaultVal**" is found in d, and so a new default Attr must be created.

$$\frac{\begin{array}{l}d(\mathbf{s}, [\![\texttt{Name}]\!]_s) = \mathbf{defaultVal} \wedge \\ \mathbf{dValNode} = \text{“\#text”}_{\mathbf{id}'''} \{ \varnothing_{\mathrm{EA}} \rangle^{\mathbf{idref}}_{\mathbf{null}\,\mathbf{3}} [\varnothing_{\mathrm{TF}}]_{\mathbf{fid}'''} \mathbf{defaultVal} \wedge \\ \mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}, \mathbf{s_{id}} \{ \\ \quad \mathbf{ea}_1 \otimes_{\mathrm{EA}} <\ll [\![\texttt{Name}]\!]_{s\,\mathbf{id}'} \mapsto [\mathbf{f}]_{\mathbf{fid}'} \gg^{\mathbf{idref}}_{\mathbf{specified}} >_{\mathrm{EA}} \otimes_{\mathrm{EA}} \mathbf{ea}_2 \\ \} \rangle^{\mathbf{idref}}_{[\![\texttt{Map}]\!]_s \mathbf{1}} [\mathbf{f}]_{\mathbf{fid}} \mathbf{null}) \\ \mathbf{g}' \equiv \mathrm{ap}(\mathbf{cg}, \mathbf{s_{id}} \{ \\ \quad \mathbf{ea}_1 \otimes_{\mathrm{EA}} \\ \quad <\ll [\![\texttt{Name}]\!]_{s\,\mathbf{id}''} \mapsto [<\mathbf{dValNode}>_{\mathrm{EA}}]_{\mathbf{fid}''} \gg^{\mathbf{idref}}_{\mathbf{false}} >_{\mathrm{EA}} \\ \quad \otimes_{\mathrm{EA}} \mathbf{ea}_2 \\ \} \rangle^{\mathbf{idref}}_{[\![\texttt{Map}]\!]_s \mathbf{1}} [\mathbf{f}]_{\mathbf{fid}} \mathbf{null}) \\ \oplus <\ll [\![\texttt{Name}]\!]_{s\,\mathbf{id}'} \mapsto [\mathbf{f}]_{\mathbf{fid}} \gg^{\mathbf{idref}}_{\mathbf{specified}} >_{\mathrm{G}}\end{array}}{s, d, \mathbf{g}, \texttt{n} := \texttt{removeNamedItem}(\texttt{Map}, \texttt{Name}) \rightsquigarrow [s|\texttt{n} \leftarrow \mathbf{id}'], d, \mathbf{g}'}$$

where $\mathbf{id}'', \mathbf{id}''', \mathbf{fid}'', \mathbf{fid}'''$ are fresh.

Character Data

The CharacterData interface is subclassed by the TextNode and Comment interfaces, although as [23] says:

Begin Quote

> No DOM objects correspond directly to CharacterData

End Quote

This means that a Character Data node is either a TextNode or a Comment, and therefore that the Character Data commands take as their first argument a node `N`, which has the following shape:

$$\mathbf{name}_{[\![\mathtt{N}]\!]_s} \{\!| \varnothing_{\mathrm{EA}} |\!\}^{\mathbf{idref}}_{\mathbf{null}\ \mathbf{txtp}} [\mathbf{f}]\mathbf{s}$$

where **txtp** $\in \{3, 8\}$, ensuring that the node is either a TextNode or a Comment. Neither TextNodes nor Comments may have attributes, so their **aid** must always be **null**. Both node types always have a non-**null** owner document and a non-**null** value **s**.

Definition 58 (CharacterData Interface Commands)**.** The CharacterData Interface Commands are:

$$\begin{aligned} &\mathtt{C}_{\mathrm{CD}} ::= \mathtt{str} := \mathtt{substringData}(\mathtt{N}, \mathtt{Offset}, \mathtt{Count}) \\ &\mid \mathtt{appendData}(\mathtt{N}, \mathtt{Arg}) \\ &\mid \mathtt{deleteData}(\mathtt{N}, \mathtt{Offset}, \mathtt{Count}) \end{aligned}$$

In all these commands, [23] specifies that:

Begin Quote

> All offsets in this interface start from 0.

End Quote

`substringData` In [23], this command is specified as follows:

Begin Quote

```
substringData
    Extracts a range of data from the node.

    Parameters
        offset          Start offset of substring to extract.
        count           The number of characters to extract.

    Return Value
        The specified substring. If the sum of offset and count
```

```
        exceeds the length, then all characters to the end of
        the data are returned.

    Exceptions
        DOMException
            INDEX_SIZE_ERR: Raised if the specified offset is
            negative or greater than the number of characters
            in data, or if the specified count is negative.

            DOMSTRING_SIZE_ERR: Raised if the specified range
            of text does not fit into a DOMString.
```

End Quote

We handle this in two cases. First, the case where the sum of `Offset` and `Count` does not exceed the length of the text value:

$$\frac{\begin{array}{c}\mathrm{len}(\mathbf{s}_1) = [\![\texttt{Offset}]\!]_s \wedge \mathrm{len}(\mathbf{s}_2) = [\![\texttt{Count}]\!]_s \wedge \\ \mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}, \mathbf{name}_{[\![\texttt{N}]\!]_s} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathbf{idref}}_{\mathbf{null}\,\mathbf{txtp}}[\mathbf{f}]\mathbf{s}_1 \otimes_{\mathrm{S}} \mathbf{s}_2 \otimes_{\mathrm{S}} \mathbf{s}_3)\end{array}}{s, d, \mathbf{g}, \texttt{str} := \texttt{substringData}(\texttt{N}, \texttt{Offset}, \texttt{Count}) \rightsquigarrow [s|\texttt{str} \leftarrow \mathbf{s}_2], d, \mathbf{g}}$$

Second, the case where the sum of `Offset` and `Count` does exceed the length of the text value:

$$\frac{\begin{array}{c}\mathrm{len}(\mathbf{s}_1) = [\![\texttt{Offset}]\!]_s \wedge \mathrm{len}(\mathbf{s}_2) < [\![\texttt{Count}]\!]_s \wedge \\ \mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}, \mathbf{name}_{[\![\texttt{N}]\!]_s} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathbf{idref}}_{\mathbf{null}\,\mathbf{txtp}}[\mathbf{f}]\mathbf{s}_1 \otimes_{\mathrm{S}} \mathbf{s}_2)\end{array}}{s, d, \mathbf{g}, \texttt{str} := \texttt{substringData}(\texttt{N}, \texttt{Offset}, \texttt{Count}) \rightsquigarrow [s|\texttt{str} \leftarrow \mathbf{s}_2], d, \mathbf{g}}$$

Since there is no possible value of $\mathbf{s}_1$ with a negative length, or a length greater than the number of characters in the data, neither of these rules covers the case where [23] requires that we throw an INDEX_SIZE_ERR. Thus, we fault as expected in place of that exception.

The DOMSTRING_SIZE_ERR however, is a class of error that we cannot reason about since the size of a DOMString is implementation-dependant.

This error is in the same class of errors as out of memory errors and arithmetic overflow - it is beyond the scope of this work.

`appendData` This is a simple command, which appends the string `Arg` to the end of the value of the node `N`

$$\frac{\begin{array}{l}\mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}, \mathbf{name}_{[\![\mathtt{N}]\!]_s} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathbf{idref}}_{\mathbf{null}_{\mathbf{txtp}}}[\mathbf{f}]\mathbf{s}) \\ \mathbf{g}' \equiv \mathrm{ap}(\mathbf{cg}, \mathbf{name}_{[\![\mathtt{N}]\!]_s} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathbf{idref}}_{\mathbf{null}_{\mathbf{txtp}}}[\mathbf{f}]\mathbf{s} \otimes_{\mathrm{S}} [\![\mathtt{Arg}]\!]_s)\end{array}}{s, d, \mathbf{g}, \mathtt{appendData}(\mathtt{N}, \mathtt{Arg}) \rightsquigarrow s, d, \mathbf{g}'}$$

`deleteData` This command deletes data from the value of the node `N`. The data to be deleted is specified exactly as for the command `substringData` above, and so we have two analogous cases:

$$\frac{\begin{array}{l}\mathrm{len}(\mathbf{s}_1) = [\![\mathtt{Offset}]\!]_s \wedge \mathrm{len}(\mathbf{s}_2) = [\![\mathtt{Count}]\!]_s \wedge \\ \mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}, \mathbf{name}_{[\![\mathtt{N}]\!]_s} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathbf{idref}}_{\mathbf{null}_{\mathbf{txtp}}}[\mathbf{f}]\mathbf{s}_1 \otimes_{\mathrm{S}} \mathbf{s}_2 \otimes_{\mathrm{S}} \mathbf{s}_3) \\ \mathbf{g}' \equiv \mathrm{ap}(\mathbf{cg}, \mathbf{name}_{[\![\mathtt{N}]\!]_s} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathbf{idref}}_{\mathbf{null}_{\mathbf{txtp}}}[\mathbf{f}]\mathbf{s}_1 \otimes_{\mathrm{S}} \mathbf{s}_3)\end{array}}{s, d, \mathbf{g}, \mathtt{deleteData}(\mathtt{N}, \mathtt{Offset}, \mathtt{Count}) \rightsquigarrow s, d, \mathbf{g}'}$$

$$\frac{\begin{array}{l}\mathrm{len}(\mathbf{s}_1) = [\![\mathtt{Offset}]\!]_s \wedge \mathrm{len}(\mathbf{s}_2) < [\![\mathtt{Count}]\!]_s \wedge \\ \mathbf{g} \equiv \mathrm{ap}(\mathbf{cg}, \mathbf{name}_{[\![\mathtt{N}]\!]_s} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathbf{idref}}_{\mathbf{null}_{\mathbf{txtp}}}[\mathbf{f}]\mathbf{s}_1 \otimes_{\mathrm{S}} \mathbf{s}_2) \\ \mathbf{g}' \equiv \mathrm{ap}(\mathbf{cg}, \mathbf{name}_{[\![\mathtt{N}]\!]_s} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathbf{idref}}_{\mathbf{null}_{\mathbf{txtp}}}[\mathbf{f}]\mathbf{s}_1)\end{array}}{s, d, \mathbf{g}, \mathtt{deleteData}(\mathtt{N}, \mathtt{Offset}, \mathtt{Count}) \rightsquigarrow s, d, \mathbf{g}'}$$

7. Context Logic for DOM Core Level 1

The context logic for DOM Core Level 1 follows the pattern of the logic for Featherweight DOM introduced in Chapter 4.

7.1. Logical Variables

In reasoning about DOM programs, we will make use of logical variables. We therefore introduce a logical environment e to store these variables, and compliment the program store s. For the purpose of writing programs in our DOM language, we have identifier, string, integer and boolean variables. In order to reason about programs, however, we will also require data structure and context variables.

Definition 59 (The Environment)**.** A logical environment e is a finite partial function sending logical variables $\mathrm{Var}_{\mathrm{LOGIC}}$ to their values:

$$e{:}\mathrm{Var}_{\mathrm{LOGIC}} \to (\{\mathbf{null}\} \cup \mathrm{ID} \cup \mathbb{Z} \cup \mathbb{B} \cup \mathrm{D} \cup \mathrm{D}_1{\to}\mathrm{D}_2)$$

where $\mathrm{D}, \mathrm{D}_1, \mathrm{D}_2 \in \mathcal{D}$ (Given in Definition 37) and $\mathrm{VAR}_{\mathrm{LOGIC}}$ denotes the set of logical variables.

We distinguish between program variables and logical variables in our reasoning by writing logical variables in `UPPERCASE`.

7.2. Logical Expressions

Just as in Featherweight DOM, we require logical expressions, in order to compare and calculate with both program and logical variables.

Definition 60 (Logical Expressions)**.** Given the special value **null**, the empty string $\varnothing_{\mathrm{S}}$, characters $\mathbf{c} \in \mathrm{CHAR}$, integers $\mathbf{n} \in \mathbb{Z}$, booleans true and

false, program variables $\texttt{var} \in \text{Var}_{\text{PROG}}$ and logical variables $\texttt{var} \in \text{Var}_{\text{LOGIC}}$, expressions $\texttt{Expr} \in \text{Exp}$ are defined by:

$$\begin{aligned}
\texttt{LExpr} ::= \quad & \textbf{null} \mid \varnothing_{\text{S}} \mid <\textbf{c}>_{\text{S}} \mid \textbf{n} \mid \text{true} \mid \text{false} \mid \\
& \texttt{var} \mid \texttt{VAR} \mid \\
& \texttt{LExpr} = \texttt{LExpr} \mid \\
& \texttt{LExpr} \otimes_{\text{D}} \texttt{LExpr} \mid \\
& \text{len}(\texttt{LExpr}) \mid \\
& \texttt{LExpr} + \texttt{LExpr} \mid \texttt{LExpr} - \texttt{LExpr} \mid \\
& \texttt{LExpr} \times \texttt{LExpr} \mid \texttt{LExpr} \div \texttt{LExpr} \mid \\
& \texttt{LExpr} \wedge \texttt{LExpr} \mid \texttt{LExpr} \vee \texttt{LExpr} \mid \neg\texttt{LExpr}
\end{aligned}$$

Again, we write "abc" to refer to the string $<\text{`a'}>_{\text{S}} \otimes_{\text{S}} <\text{`b'}>_{\text{S}} \otimes_{\text{S}} <\text{`c'}>_{\text{S}}$ and we write logical expression names in `UpperCamelCase`.

Definition 61 (Logical Expression Evaluation)**.** The evaluation of logical expression `LExpr` in store s and environment e is given in Figure 7.1. Note that this evaluation is a partial function, since not all expressions can be successfully evaluated. If such an expression is part of a formula, then that formula is unsatisfiable.

Lemma 62. *Program and Logic Expression Equivalence: For any program expression* ***Expr****, store* s *and environment* e *such that* $[\![\textbf{\textit{Expr}}]\!]_s$ *does not* fault*,* $[\![\textbf{\textit{Expr}}]\!]_s \equiv [\![\textbf{\textit{Expr}}]\!]_{s,e}$

Proof. The proof follows directly from the definitions of program and logical expressions (which are identical except for the addition of environment variables to logical expressions) and the evaluation functions of program and logical expressions (which are identical except for the addition of environment variable evaluation for logical expressions). □

7.3. Logical Formulae

In addition to the sorts of formulae used to reason about Featherweight DOM, DOM Core Level 1 requires formulae for checking the DTD Fragment, for type casting and for flattening trees.

$$
\begin{aligned}
[\![\mathbf{null}]\!]_{s,e} &\triangleq \mathbf{null} \\
[\![\varnothing_{\mathrm{S}}]\!]_{s,e} &\triangleq \varnothing_{\mathrm{S}} \\
[\![<\mathbf{c}>_{\mathrm{S}}]\!]_{s,e} &\triangleq <\mathbf{c}>_{\mathrm{S}} \\
[\![\mathbf{n}]\!]_{s,e} &\triangleq \mathbf{n} \\
[\![\mathrm{true}]\!]_{s,e} &\triangleq \mathrm{true} \\
[\![\mathrm{false}]\!]_{s,e} &\triangleq \mathrm{false} \\
[\![\mathtt{var}]\!]_{s,e} &\triangleq s(\mathtt{var}) \text{ iff } \mathtt{var} \in \mathrm{dom}(s) \\
[\![\mathtt{VAR}]\!]_{s,e} &\triangleq e(\mathtt{VAR}) \text{ iff } \mathtt{VAR} \in \mathrm{dom}(e) \\
[\![\mathtt{LExpr} = \mathtt{LExpr}']\!]_{s,e} &\triangleq [\![\mathtt{LExpr}]\!]_{s,e} = [\![\mathtt{LExpr}']\!]_{s,e} \\
[\![\mathtt{LExpr} \otimes_{\mathrm{D}} \mathtt{LExpr}']\!]_{s,e} &\triangleq [\![\mathtt{LExpr}]\!]_{s,e} \otimes_{\mathrm{D}} [\![\mathtt{LExpr}']\!]_{s,e} \text{ iff } [\![\mathtt{LExpr}]\!]_{s,e}, [\![\mathtt{LExpr}']\!]_{s,e} \in \mathrm{D} \\
[\![\mathrm{len}(\mathtt{LExpr})]\!]_{s,e} &\triangleq \mathrm{len}([\![\mathtt{LExpr}]\!]_{s,e}) \\
[\![\mathtt{LExpr} + \mathtt{LExpr}']\!]_{s,e} &\triangleq [\![\mathtt{LExpr}]\!]_{s,e} + [\![\mathtt{LExpr}']\!]_{s,e} \text{ iff } [\![\mathtt{LExpr}]\!]_{s,e}, [\![\mathtt{LExpr}']\!]_{s,e} \in \mathbb{Z} \\
[\![\mathtt{LExpr} - \mathtt{LExpr}']\!]_{s,e} &\triangleq [\![\mathtt{LExpr}]\!]_{s,e} - [\![\mathtt{LExpr}']\!]_{s,e} \text{ iff } [\![\mathtt{LExpr}]\!]_{s,e}, [\![\mathtt{LExpr}']\!]_{s,e} \in \mathbb{Z} \\
[\![\mathtt{LExpr} \times \mathtt{LExpr}']\!]_{s,e} &\triangleq [\![\mathtt{LExpr}]\!]_{s,e} \times [\![\mathtt{LExpr}']\!]_{s,e} \text{ iff } [\![\mathtt{LExpr}]\!]_{s,e}, [\![\mathtt{LExpr}']\!]_{s,e} \in \mathbb{Z} \\
[\![\mathtt{LExpr} \div \mathtt{LExpr}']\!]_{s,e} &\triangleq [\![\mathtt{LExpr}]\!]_{s,e} \div [\![\mathtt{LExpr}']\!]_{s,e} \text{ iff } [\![\mathtt{LExpr}]\!]_{s,e}, [\![\mathtt{LExpr}']\!]_{s,e} \in \mathbb{Z} \\
[\![\mathtt{LExpr} \wedge \mathtt{LExpr}']\!]_{s,e} &\triangleq [\![\mathtt{LExpr}]\!]_{s,e} \wedge [\![\mathtt{LExpr}']\!]_{s,e} \text{ iff } [\![\mathtt{LExpr}]\!]_{s,e}, [\![\mathtt{LExpr}']\!]_{s,e} \in \mathbb{B} \\
[\![\mathtt{LExpr} \vee \mathtt{LExpr}']\!]_{s,e} &\triangleq [\![\mathtt{LExpr}]\!]_{s,e} \vee [\![\mathtt{LExpr}']\!]_{s,e} \text{ iff } [\![\mathtt{LExpr}]\!]_{s,e}, [\![\mathtt{LExpr}']\!]_{s,e} \in \mathbb{B}
\end{aligned}
$$

Figure 7.1.: Logical Expression Evaluation

Definition 63 (Logical Formulae)**.** The formulae for DOM Core Level 1 are:

$P ::=$	$\neg P \mid P \wedge P \mid P \vee P \mid \mathrm{true}_{\mathrm{A}} \mid \mathrm{false}_{\mathrm{A}}$	Boolean formulae
	$\mid P \circ_{\mathrm{D}} P \mid P \circ\!\!-_{\mathrm{D}_2} P \mid P \multimap P$	structural formulae
	$\ldots$ (see below) $\ldots$	DOM-specific formulae
	$\mid \mathtt{VAR}{:}\mathrm{A}$	logical variables
	$\mid \mathtt{LExpr} \doteq \mathtt{LExpr} \mid \mathtt{LExpr} <\cdot\, \mathtt{LExpr}$	expression equality and inequality
	$\mid \mathtt{LExpr} \in \mathtt{LExpr}$	string exclusion
	$\mid \mathtt{LExpr} \in \mathrm{D}$	type checking
	$\mid (\mathtt{LExpr}, \mathtt{LExpr}) \in \mathrm{dom}(d)$	DTD Fragment Domain Check
	$\mid \mathrm{cast}_{\mathrm{D}_2}(\mathrm{D}_1, P)$	type casting
	$\mid \mathrm{flatten}(\mathtt{LExpr}, P)$	tree flattening
	$\mid \exists \mathtt{VAR}.\, P$	quantification

where VAR is a logical variable, $\mathrm{A} \in \mathcal{A}$, $\mathrm{D}, \mathrm{D}_1, \mathrm{D}_2 \in \mathcal{D}$ and d is a DTD Fragment as introduced in Definition 46.

Definition 64 (DOM-Specific Formulae)**.** The DOM-specific formulae are:

$$
\begin{aligned}
P ::= \ldots \quad & \mid -_{\mathrm{D}} \mid P_{\mathtt{Id}} \{\!\!\{ P \}\!\!\}_{\mathtt{Aid}_{\mathtt{Tp}}}^{\mathtt{IdRef}} [P]_{\mathtt{Fid}} P \mid \\
& P \otimes_{\mathrm{D}} P \mid P \oplus_{\mathrm{G}} P \mid \varnothing_{\mathrm{D}} \mid <P>_{\mathrm{D}} \mid \\
& \ll P_{\mathtt{Id}} \mapsto [P]_{\mathtt{Fid}} \gg_{\mathtt{Specified}}^{\mathtt{IdRef}} \mid {}_{P}^{\mathtt{IdRef}}{}_{\mathtt{Fid}} \mid <P, P, P>_{\mathrm{DF}} \mid \\
& \mathtt{LExpr} \mid d(\mathtt{LExpr}, \mathtt{LExpr})
\end{aligned}
$$

where $\mathrm{D} \in \mathcal{D}$.

7.3.1. Formula Types

Just as in Section 4.4, the type annotations on the formulae enable us to define a simple typing relation $P{:}A$ by induction on the structure of formula P.

Definition 65 (Formula Types)**.** The types for all the formulae are as in Featherweight DOM, except for those of the DOM-specific formulae which have changed slightly and which we give in Figure 7.2.

7.3.2. Satisfaction Relation

Satisfaction for our formulae follows that of Featherweight DOM, with differences in the DOM-specific cases.

Definition 66 (Satisfaction)**.** The satisfaction relation $e, s, d, \mathbf{a} \models_{\mathrm{A}} P$ is defined on environment e, variable store s, DTD fragment d, datum or context $\mathbf{a}$ of type A, and formula P of type A by induction on P. In all the following, A is a data or context type, $\mathbf{d}$ is a datum of type D and $\mathbf{cd}$ is a context of type $\mathrm{D}_1{\to}\mathrm{D}_2$.

We give the cases for the boolean formulae in Figure 7.3, the structural formulae in Figure 7.4, the DOM-specific formulae in Figures 7.5 and 7.6, and the remaining formulae in Figure 7.7.

As with Featherweight DOM, the behaviour of logical negation is subtle. See Section 4.5 for details.

As in Featherweight DOM, we use $\doteq$ and $<\cdot$ to distinguish between logical equality and inequality and expression equality and inequality. As before, expressions are untyped, while formulae are strongly typed, and if an expression appears without an explicit type declaration in a formula it is assumed to be of the string type S.

$$\frac{}{\varnothing_{\mathrm{D}}:\mathrm{D}} \qquad \frac{}{-_{\mathrm{D}}:\mathrm{D}\to\mathrm{D}} \qquad \frac{P_1:\mathrm{F}\wedge P_2:\mathrm{F}}{(P_1\otimes_{\mathrm{F}}P_2):\mathrm{F}} \qquad \frac{(P_1:\mathrm{D}\to\mathrm{F}\wedge P_2:\mathrm{F})\vee(P_1:\mathrm{F}\wedge P_2:\mathrm{D}\to\mathrm{F})}{(P_1\otimes_{\mathrm{F}}P_2):\mathrm{D}\to\mathrm{F}}$$

$$\frac{P:\mathrm{N}}{<P>_{\mathrm{G}}:\mathrm{G}} \qquad \frac{P:\mathrm{D}\to\mathrm{N}}{<P>_{\mathrm{G}}:\mathrm{D}\to\mathrm{G}} \qquad \frac{P_1:\mathrm{G}\wedge P_2:\mathrm{G}}{(P_1\oplus P_2):\mathrm{G}} \qquad \frac{P_1:\mathrm{D}\to\mathrm{G}\wedge P_2:\mathrm{G}}{(P_1\oplus P_2):\mathrm{D}\to\mathrm{G}}$$

$$\frac{P:\mathrm{S}\wedge P':\mathrm{EA}\wedge P'':\mathrm{DF}\wedge P''':\mathrm{S}}{P_{\mathtt{Id}}\{\!\{\ P'\ \}\!\}^{\mathtt{IdRef}}_{\mathtt{Aid}_{\mathtt{Tp}}}[P'']_{\mathtt{Fid}}P''':\mathrm{DOC}} \qquad \frac{P_1:\mathrm{DNEL}\wedge P_2:\mathrm{DE}\wedge P_3:\mathrm{DNEL}}{<P_1,P_2,P_3>_{\mathrm{DF}}:\mathrm{DF}}$$

$$\frac{P:\mathrm{S}\wedge P':\mathrm{EA}\wedge P'':\mathrm{D}\to DF\wedge P''':\mathrm{S}}{P_{\mathtt{Id}}\{\!\{\ P'\ \}\!\}^{\mathtt{IdRef}}_{\mathtt{Aid}_{\mathtt{Tp}}}[P'']_{\mathtt{Fid}}P''':\mathrm{D}\to\mathrm{DOC}} \qquad \frac{\begin{array}{l}(P_1:\mathrm{D}\to\mathrm{DNEL}\wedge P_2:\mathrm{DE}\wedge P_3:\mathrm{DNEL})\vee\\(P_1:\mathrm{DNEL}\wedge P_2:\mathrm{D}\to\mathrm{DE}\wedge P_3:\mathrm{DNEL})\vee\\(P_1:\mathrm{DNEL}\wedge P_2:\mathrm{DE}\wedge P_3:\mathrm{D}\to\mathrm{DNEL})\end{array}}{<P_1,P_2,P_3>_{\mathrm{DF}}:\mathrm{D}\to\mathrm{DF}}$$

$$\frac{P:\mathrm{COMM}}{<P>_{\mathrm{DNEL}}:\mathrm{DNEL}} \qquad \frac{P:\mathrm{D}\to\mathrm{COMM}}{<P>_{\mathrm{DNEL}}:\mathrm{D}\to\mathrm{DNEL}} \qquad \frac{P:\mathrm{ELE}}{<P>_{\mathrm{DE}}:\mathrm{DE}} \qquad \frac{P:\mathrm{D}\to\mathrm{ELE}}{<P>_{\mathrm{DE}}:\mathrm{D}\to\mathrm{DE}}$$

$$\frac{P:\mathrm{S}\wedge P':\mathrm{EA}\wedge P'':FRAGF\wedge P''':\mathrm{S}}{P_{\mathtt{Id}}\{\!\{\ P'\ \}\!\}^{\mathtt{IdRef}}_{\mathtt{Aid}_{\mathtt{Tp}}}[P'']_{\mathtt{Fid}}P''':\mathrm{FRAG}} \qquad \frac{P:\mathrm{S}\wedge P':\mathrm{EA}\wedge P''\mathrm{D}\to:FRAGF\wedge P''':\mathrm{S}}{P_{\mathtt{Id}}\{\!\{\ P'\ \}\!\}^{\mathtt{IdRef}}_{\mathtt{Aid}_{\mathtt{Tp}}}[P'']_{\mathtt{Fid}}P''':\mathrm{D}\to\mathrm{FRAG}}$$

$$\frac{P:\mathrm{ELE}\vee P:\mathrm{TXT}\vee P:\mathrm{COMM}}{<P>_{\mathrm{FRAGF}}:\mathrm{FRAGF}} \qquad \frac{P:\mathrm{D}\to\mathrm{ELE}\vee P:\mathrm{D}\to\mathrm{TXT}\vee P:\mathrm{D}\to\mathrm{COMM}}{<P>_{\mathrm{FRAGF}}:\mathrm{D}\to\mathrm{FRAGF}}$$

$$\frac{P:\mathrm{S}\wedge P':\mathrm{EA}\wedge P'':EF\wedge P''':\mathrm{S}}{P_{\mathtt{Id}}\{\!\{\ P'\ \}\!\}^{\mathtt{IdRef}}_{\mathtt{Aid}_{\mathtt{Tp}}}[P'']_{\mathtt{Fid}}P''':\mathrm{ELE}} \qquad \frac{\begin{array}{l}(P:\mathrm{D}\to\mathrm{S}\wedge P':\mathrm{EA}\wedge P'':\mathrm{EF}\wedge P''':\mathrm{S})\vee\\(P:\mathrm{S}\wedge P':\mathrm{D}\to\mathrm{EA}\wedge P'':\mathrm{EF}\wedge P''':\mathrm{S})\vee\\(P:\mathrm{S}\wedge P':\mathrm{EA}\wedge P'':\mathrm{D}\to\mathrm{EF}\wedge P''':\mathrm{S})\vee\\(P:\mathrm{S}\wedge P':\mathrm{EA}\wedge P'':\mathrm{EF}\wedge P''':\mathrm{D}\to\mathrm{S})\end{array}}{P_{\mathtt{Id}}\{\!\{\ P'\ \}\!\}^{\mathtt{IdRef}}_{\mathtt{Aid}_{\mathtt{Tp}}}[P'']_{\mathtt{Fid}}P''':\mathrm{D}\to\mathrm{ELE}}$$

$$\frac{P:\mathrm{ATTR}}{<P>_{\mathrm{EA}}:\mathrm{EA}} \qquad \frac{P:\mathrm{D}\to\mathrm{ATTR}}{<P>_{\mathrm{EA}}:\mathrm{D}\to\mathrm{EA}}$$

$$\frac{P:\mathrm{ELE}\vee P:\mathrm{TXT}\vee P:\mathrm{COMM}}{<P>_{\mathrm{EF}}:\mathrm{EF}} \qquad \frac{P:\mathrm{D}\to\mathrm{ELE}\vee P:\mathrm{D}\to\mathrm{TXT}\vee P:\mathrm{D}\to\mathrm{COMM}}{<P>_{\mathrm{EF}}:\mathrm{D}\to\mathrm{EF}}$$

$$\frac{P:\mathrm{S}\wedge P':\mathrm{AF}}{\ll P_{\mathtt{Id}}\mapsto[P']_{\mathtt{Fid}}\gg^{\mathtt{IdRef}}_{\mathtt{Specified}}:\mathrm{ATTR}} \qquad \frac{(P:\mathrm{D}\to\mathrm{S}\wedge P':\mathrm{AF})\vee(P:\mathrm{S}\wedge P':\mathrm{D}\to\mathrm{AF})}{\ll P_{\mathtt{Id}}\mapsto[P']_{\mathtt{FId}}\gg^{\mathtt{IdRef}}_{\mathtt{Specified}}:\mathrm{D}\to\mathrm{ATTR}}$$

$$\frac{P:\mathrm{TXT}}{<P>_{\mathrm{AF}}:\mathrm{AF}} \qquad \frac{P:\mathrm{D}\to\mathrm{TXT}}{<P>_{\mathrm{AF}}:\mathrm{D}\to\mathrm{AF}}$$

$$\frac{P:\mathrm{S}\wedge P':\mathrm{EA}\wedge P'':\mathrm{CF}\wedge P''':\mathrm{S}}{P_{\mathtt{Id}}\{\!\{\ P'\ \}\!\}^{\mathtt{IdRef}}_{\mathtt{Aid}_{\mathtt{Tp}}}[P'']_{\mathtt{Fid}}P''':\mathrm{D}\to\mathrm{COMM}} \qquad \frac{P:\mathrm{S}\wedge P':\mathrm{EA}\wedge P'':\mathrm{CF}\wedge P''':\mathrm{D}\to\mathrm{S}}{P_{\mathtt{Id}}\{\!\{\ P'\ \}\!\}^{\mathtt{IdRef}}_{\mathtt{Aid}_{\mathtt{Tp}}}[P'']_{\mathtt{Fid}}P''':\mathrm{D}\to\mathrm{COMM}}$$

$$\frac{P:\mathrm{S}\wedge P':\mathrm{EA}\wedge P'':\mathrm{TF}\wedge P''':\mathrm{S}}{P_{\mathtt{Id}}\{\!\{\ P'\ \}\!\}^{\mathtt{IdRef}}_{\mathtt{Aid}_{\mathtt{Tp}}}[P'']_{\mathtt{Fid}}P''':\mathrm{TXT}} \qquad \frac{P:\mathrm{S}\wedge P':\mathrm{EA}\wedge P'':\mathrm{TF}\wedge P''':\mathrm{D}\to\mathrm{S}}{P_{\mathtt{Id}}\{\!\{\ P'\ \}\!\}^{\mathtt{IdRef}}_{\mathtt{Aid}_{\mathtt{Tp}}}[P'']_{\mathtt{Fid}}P''':\mathrm{D}\to\mathrm{TXT}}$$

$$\frac{P:\mathrm{S}}{{}^{\mathtt{IdRef}}_{P}{}_{\mathtt{Fid}}:\mathrm{ES}} \qquad \frac{}{\mathtt{LExpr}:\mathrm{S}} \qquad \frac{}{d(\mathtt{LExpr},\mathtt{LExpr}'):\mathrm{S}} \qquad \frac{}{(\mathtt{LExpr},\mathtt{LExpr}')\in\mathrm{dom}(d):\mathrm{A}}$$

where $\mathrm{F}\in\mathcal{F},\mathrm{N}\in\mathcal{N},\mathrm{D}\in\mathcal{D},\mathrm{A}\in\mathcal{A}$, $\mathcal{F},\mathcal{N}$ and $\mathcal{D}$ are given in Definition 37 and $\mathcal{A}$ is given in Definition 40.

Figure 7.2.: The Types of DOM-Specific Formulae

$$
\begin{array}{lll}
e,s,d,\mathbf{a} \models_{\mathrm{A}} \neg P & \iff & P{:}\mathrm{A} \wedge e,s,d,\mathbf{a} \not\models_{\mathrm{A}} P \\
e,s,d,\mathbf{a} \models_{\mathrm{A}} P_1 \wedge P_2 & \iff & (e,s,d,\mathbf{a} \models_{\mathrm{A}} P_1) \wedge (e,s,d,\mathbf{a} \models_{\mathrm{A}} P_2) \\
e,s,d,\mathbf{a} \models_{\mathrm{A}} P_1 \vee P_2 & \iff & (e,s,d,\mathbf{a} \models_{\mathrm{A}} P_1) \vee (e,s,d,\mathbf{a} \models_{\mathrm{A}} P_2) \\
e,s,d,\mathbf{a} \models_{\mathrm{A}} \mathrm{true}_{\mathrm{A}} & & \mathit{always} \\
e,s,d,\mathbf{a} \models_{\mathrm{A}} \mathrm{false}_{\mathrm{A}} & & \mathit{never}
\end{array}
$$

Figure 7.3.: Satisfaction for the Boolean Formulae

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

Figure 7.4.: Satisfaction for the Structural Formulae

7.4. Derived Formulae

As with Featherweight DOM, $\cdot>$, $<\doteq$, $\doteq>$ and $\notin$ are derivable in the usual way. Also following Featherweight DOM, we have the following derived formulae.

Definition 67 (Derived Formulae)**.** The notations for expressing 'somewhere, potentially deep down' $\Diamond_{\mathrm{D}_1 \to \mathrm{D}_2} P$ and 'everywhere' $\Box_{\mathrm{D}_1 \to \mathrm{D}_2} P$ are defined by:

$$
\begin{array}{l}
\Diamond_{\mathrm{D}_1 \to \mathrm{D}_2} P \triangleq \mathrm{true}_{\mathrm{D}_1 \to \mathrm{D}_2} \circ_{\mathrm{D}_1} P \\
\Box_{\mathrm{D}_1 \to \mathrm{D}_2} P \triangleq \neg \Diamond_{\mathrm{D}_1 \to \mathrm{D}_2} \neg P
\end{array}
$$

Similarly, formula $\Diamond_{\otimes}(P, \mathrm{TP})$ means "somewhere at this forest level".

$$
\Diamond_{\otimes}(P, \mathrm{TP}) \triangleq \mathrm{true}_{\mathrm{TP}} \otimes_{\mathrm{TP}} <P>_{\mathrm{TP}} \otimes_{\mathrm{TP}} \mathrm{true}
$$

$$e,s,d,\mathbf{cd} \models_{\mathrm{D}\to\mathrm{D}} -_{\mathrm{D}} \iff \mathbf{cd} \equiv -_{\mathrm{D}}$$

$$e,s,d,\mathbf{d} \models_{\mathrm{D}} P_{\mathtt{Id}} \{\!\!\{ P' \}\!\!\}^{\mathtt{Owner}}_{\mathtt{AId}_{\mathtt{Tp}}} [P'']_{\mathtt{Fid}} P''' \iff \exists \mathbf{name}{:}\mathrm{S}, \mathbf{ea}{:}\mathrm{EA}, \mathbf{d}''{:}\mathrm{D}'', \mathbf{s}{:}\mathrm{S}.\ (\mathbf{d} \equiv \mathbf{name}_{[\![\mathtt{Id}]\!]_{s,e}} \{\!\!\{ \mathbf{ea} \}\!\!\}^{[\![\mathtt{Owner}]\!]_{s,e}}_{[\![\mathtt{AId}]\!]_{s,e}\,[\![\mathtt{Tp}]\!]_{s,e}} [\mathbf{d}'']_{[\![\mathtt{Fid}]\!]_{s,e}} \mathbf{s}) \wedge e,s,d,\mathbf{name} \models_{\mathrm{S}} P \wedge e,s,d,\mathbf{ea} \models_{\mathrm{EA}} P' \wedge e,s,d,\mathbf{d}'' \models''_{\mathrm{D}} P'' \wedge e,s,d,\mathbf{s} \models_{\mathrm{S}} P'''$$

$$e,s,d,\mathbf{cd} \models_{\mathrm{D}_1\to\mathrm{D}_2} P_{\mathtt{Id}} \{\!\!\{ P' \}\!\!\}^{\mathtt{Owner}}_{\mathtt{AId}_{\mathtt{Tp}}} [P'']_{\mathtt{Fid}} P''' \iff$$

$$\left(\begin{array}{l} \exists \mathbf{name}{:}\mathrm{S}, \mathbf{c}{:}\mathrm{D}_1\to\mathrm{EA}, \mathbf{d}''{:}\mathrm{D}'', \mathbf{s}{:}\mathrm{S}. \\ (\mathbf{cd} \equiv \mathbf{name}_{[\![\mathtt{Id}]\!]_{s,e}} \{\!\!\{ \mathbf{c} \}\!\!\}^{[\![\mathtt{Owner}]\!]_{s,e}}_{[\![\mathtt{AId}]\!]_{s,e}\,[\![\mathtt{Tp}]\!]_{s,e}} [\mathbf{d}'']_{[\![\mathtt{Fid}]\!]_{s,e}} \mathbf{s}) \wedge \\ e,s,d,\mathbf{name} \models_{\mathrm{S}} P \wedge e,s,d,\mathbf{c} \models_{\mathrm{D}_1\to\mathrm{EA}} P' \wedge \\ e,s,d,\mathbf{d}'' \models''_{\mathrm{D}} P'' \wedge e,s,d,\mathbf{s} \models_{\mathrm{S}} P''' \end{array}\right)$$

$$\vee \left(\begin{array}{l} \exists \mathbf{name}{:}\mathrm{S}, \mathbf{ea}{:}\mathrm{EA}, \mathbf{c}{:}\mathrm{D}_1\to\mathrm{D}'', \mathbf{s}{:}\mathrm{S}. \\ (\mathbf{cd} \equiv \mathbf{name}_{[\![\mathtt{Id}]\!]_{s,e}} \{\!\!\{ \mathbf{ea} \}\!\!\}^{[\![\mathtt{Owner}]\!]_{s,e}}_{[\![\mathtt{AId}]\!]_{s,e}\,[\![\mathtt{Tp}]\!]_{s,e}} [\mathbf{c}]_{[\![\mathtt{Fid}]\!]_{s,e}} \mathbf{s}) \wedge \\ e,s,d,\mathbf{name} \models_{\mathrm{S}} P \wedge e,s,d,\mathbf{ea} \models_{\mathrm{EA}} P' \wedge \\ e,s,d,\mathbf{c} \models_{\mathrm{D}_1\to\mathrm{D}''} P'' \wedge e,s,d,\mathbf{s} \models_{\mathrm{S}} P''' \end{array}\right)$$

$$\vee \left(\begin{array}{l} \exists \mathbf{name}{:}\mathrm{S}, \mathbf{ea}{:}\mathrm{EA}, \mathbf{d}''{:}\mathrm{D}'', \mathbf{c}{:}\mathrm{D}_1\to\mathrm{S}. \\ (\mathbf{cd} \equiv \mathbf{name}_{[\![\mathtt{Id}]\!]_{s,e}} \{\!\!\{ \mathbf{ea} \}\!\!\}^{[\![\mathtt{Owner}]\!]_{s,e}}_{[\![\mathtt{AId}]\!]_{s,e}\,[\![\mathtt{Tp}]\!]_{s,e}} [\mathbf{d}'']_{[\![\mathtt{Fid}]\!]_{s,e}} \mathbf{c}) \wedge \\ e,s,d,\mathbf{name} \models_{\mathrm{S}} P \wedge e,s,d,\mathbf{ea} \models_{\mathrm{EA}} P' \wedge \\ e,s,d,\mathbf{d}'' \models''_{\mathrm{D}} P'' \wedge e,s,d,\mathbf{c} \models_{\mathrm{D}_1\to\mathrm{S}} P''' \end{array}\right)$$

$$e,s,d,\mathbf{d} \models_{\mathrm{D}} P' \otimes_{\mathrm{D}} P'' \iff \exists \mathbf{d}'{:}\mathrm{D}, \mathbf{d}''{:}\mathrm{D}.\ (\mathbf{d} \equiv \mathbf{d}' \otimes_{\mathrm{D}} \mathbf{d}'') \wedge e,s,d,\mathbf{d}' \models_{\mathrm{D}} P' \wedge e,s,d,\mathbf{d}'' \models_{\mathrm{D}} P''$$

$$e,s,d,\mathbf{cd} \models_{\mathrm{D}_1\to\mathrm{D}_2} P' \otimes_{\mathrm{D}_2} P'' \iff \exists \mathbf{cd}'{:}(\mathrm{D}_1\to\mathrm{D}_2), \mathbf{d}{:}\mathrm{D}_2.\ ((\mathbf{cd} \equiv \mathbf{cd}' \otimes_{\mathrm{D}_2} \mathbf{d}) \wedge e,s,d,\mathbf{cd}' \models_{\mathrm{D}_1\to\mathrm{D}_2} P' \wedge e,s,d,\mathbf{d} \models_{\mathrm{D}_2} P'') \vee ((\mathbf{cd} \equiv \mathbf{d} \otimes_{\mathrm{D}_2} \mathbf{cd}') \wedge e,s,d,\mathbf{d} \models_{\mathrm{D}_2} P' \wedge e,s,d,\mathbf{cd}' \models_{\mathrm{D}_1\to\mathrm{D}_2} P'')$$

$$e,s,d,\mathbf{d} \models_{\mathrm{D}} P' \oplus P'' \iff \exists \mathbf{d}'{:}\mathrm{D}, \mathbf{d}''{:}\mathrm{D}.\ (\mathbf{d} \equiv \mathbf{d}' \oplus \mathbf{d}'') \wedge e,s,d,\mathbf{d}' \models_{\mathrm{D}} P' \wedge e,s,d,\mathbf{d}'' \models_{\mathrm{D}} P''$$

$$e,s,d,\mathbf{cd} \models_{\mathrm{D}_1\to\mathrm{D}_2} P' \oplus P'' \iff \exists \mathbf{cd}'{:}(\mathrm{D}_1\to\mathrm{D}_2), \mathbf{d}{:}\mathrm{D}_2.\ (\mathbf{cd} \equiv \mathbf{cd}' \oplus \mathbf{d}) \wedge e,s,d,\mathbf{cd}' \models_{\mathrm{D}_1\to\mathrm{D}_2} P' \wedge e,s,d,\mathbf{d} \models_{\mathrm{D}_2} P''$$

Figure 7.5.: Satisfaction for the DOM Formulae (Part One)

$$e, s, d, \mathbf{d} \models_{\mathrm{D}} \varnothing_{\mathrm{D}} \iff \mathbf{d} \equiv \varnothing_{\mathrm{D}}$$

$$e, s, d, \mathbf{d} \models_{\mathrm{D}} <P>_{\mathrm{D}} \iff \exists \mathbf{d}' {:} \mathrm{D}'.\ (\mathbf{d} \equiv <\mathbf{d}'>_{\mathrm{D}}) \wedge e, s, d, \mathbf{d}' \models_{\mathrm{D}'} P$$

$$e, s, d, \mathbf{cd} \models_{\mathrm{D}_1 \to \mathrm{D}_2} <P>_{\mathrm{D}_3} \iff \exists \mathbf{cd}' {:} (\mathrm{D}_1 \to \mathrm{D}_2').\ (\mathbf{cd} \equiv <\mathbf{cd}'>) \wedge e, s, d, \mathbf{cd}' \models_{\mathrm{D}_1 \to \mathrm{D}_2'} P$$

$$e, s, d, \mathbf{attr} \models_{\mathrm{ATTR}} \ll P_{\texttt{Id}} \mapsto [P']_{\texttt{Fid}} \gg^{\texttt{Owner}}_{\texttt{Specified}} \iff \exists \mathbf{s}{:}\mathrm{S}, \mathbf{af}{:}\mathrm{AF}.\ (\mathbf{attr} \equiv \ll \mathbf{s}_{[\![\texttt{Id}]\!]_{s,e}} \mapsto [\mathbf{af}]_{[\![\texttt{Fid}]\!]_{s,e}} \gg^{[\![\texttt{Owner}]\!]_{s,e}}_{[\![\texttt{Specified}]\!]_{s,e}}) \wedge e, s, d, \mathbf{s} \models_{\mathrm{S}} P \wedge e, s, d, \mathbf{af} \models_{\mathrm{AF}} P'$$

$$e, s, d, \mathbf{cattr} \models_{\mathrm{D}_1 \to \mathrm{ATTR}} \ll P_{\texttt{Id}} \mapsto [P']_{\texttt{Fid}} \gg^{\texttt{Owner}}_{\texttt{Specified}} \iff \exists \mathbf{s}{:}\mathrm{S}, \mathbf{caf}{:}\mathrm{D}_1 \to \mathrm{AF}.\ (\mathbf{attr} \equiv \ll \mathbf{s}_{[\![\texttt{Id}]\!]_{s,e}} \mapsto [\mathbf{af}]_{[\![\texttt{Fid}]\!]_{s,e}} \gg^{[\![\texttt{Owner}]\!]_{s,e}}_{[\![\texttt{Specified}]\!]_{s,e}}) \wedge e, s, d, \mathbf{s} \models_{\mathrm{S}} P \wedge e, s, d, \mathbf{caf} \models_{\mathrm{D}_1 \to \mathrm{AF}} P'$$

$$e, s, d, \mathbf{d} \models_{\mathrm{DF}} <P_1, P_2, P_3>_{\mathrm{DF}} \iff \exists \mathbf{dnel}{:}\mathrm{DNEL}, \mathbf{de}{:}\mathrm{DE}, \mathbf{dnel'}{:}\mathrm{DNEL}.\ \mathbf{d} \equiv <\mathbf{dnel}, \mathbf{de}, \mathbf{dnel'}>_{\mathrm{DF}} \wedge \mathbf{dnel} \models_{\mathrm{DNEL}} P_1 \wedge \mathbf{de} \models_{\mathrm{DE}} P_2 \wedge \mathbf{dnel'} \models_{\mathrm{DNEL}} P_3$$

$$e, s, d, \mathbf{cd} \models_{\mathrm{D}_1 \to \mathrm{DF}} <P_1, P_2, P_3>_{\mathrm{DF}} \iff \left(\begin{array}{l} \exists \mathbf{cdnel}{:}D_1 \to \mathrm{DNEL}, \mathbf{de}{:}\mathbf{DE}, \mathbf{dnel'}{:}\mathrm{DNEL}. \\ \mathbf{cd} \equiv <\mathbf{cdnel}, \mathbf{de}, \mathbf{dnel'}>_{\mathrm{DF}} \wedge \\ \mathbf{cdnel} \models_{\mathrm{D}_1 \to \mathrm{DNEL}} P_1 \wedge \\ \mathbf{de} \models_{\mathrm{DE}} P_2 \wedge \\ \mathbf{dnel'} \models_{\mathrm{DNEL}} P_3 \end{array} \right)$$

$$\vee \left(\begin{array}{l} \exists \mathbf{dnel}{:}\mathrm{DNEL}, \mathbf{cd'}{:}D_1 \to \mathrm{DE}, \mathbf{dnel'}{:}\mathrm{DNEL}. \\ \mathbf{cd} \equiv <\mathbf{dnel}, \mathbf{cd'}, \mathbf{dnel'}>_{\mathrm{DF}} \wedge \\ \mathbf{dnel} \models_{\mathrm{DNEL}} P_1 \wedge \\ \mathbf{cd'} \models_{\mathrm{D}_1 \to \mathrm{DE}} P_2 \wedge \\ \mathbf{dnel'} \models_{\mathrm{DNEL}} P_3 \end{array} \right)$$

$$\vee \left(\begin{array}{l} \exists \mathbf{dnel}{:}\mathrm{DNEL}, \mathbf{de}{:}\mathbf{DE}, \mathbf{cdnel}{:}D_1 \to \mathrm{DNEL}. \\ \mathbf{cd} \equiv <\mathbf{dnel}, \mathbf{de}, \mathbf{cdnel}>_{\mathrm{DF}} \wedge \\ \mathbf{dnel} \models_{\mathrm{DNEL}} P_1 \wedge \\ \mathbf{de} \models_{\mathrm{DE}} P_2 \wedge \\ \mathbf{cdnel} \models_{\mathrm{D}_1 \to \mathrm{DNEL}} P_3 \end{array} \right)$$

$$e, s, d, \mathbf{s} \models_{\mathrm{S}} \texttt{LExpr} \iff \mathbf{s} \equiv [\![\texttt{LExpr}]\!]_{s,e} \wedge [\![\texttt{LExpr}]\!]_{s,e} \in \mathrm{S} \cup \{\mathbf{null}\}$$

$$e, s, d, \mathbf{s} \models_{\mathrm{S}} d(\texttt{LExpr}, \texttt{LExpr}') \iff \mathbf{s} \equiv d([\![\texttt{LExpr}]\!]_{s,e}, [\![\texttt{LExpr}']\!]_{s,e}) \wedge [\![\texttt{LExpr}]\!]_{s,e}, [\![\texttt{LExpr}']\!] \in \mathrm{S}$$

Figure 7.6.: Satisfacion for the DOM Formulae (Part Two)

$$
\begin{array}{lcl}
e, s, d, \mathbf{a} \models_{\mathrm{A}} \texttt{VAR}{:}\mathrm{A} & \iff & e(\texttt{VAR}) \in \mathrm{A} \wedge \mathbf{a} \equiv e(\texttt{VAR}) \\
e, s, d, \mathbf{a} \models_{\mathrm{A}} \texttt{LExpr} \doteq \texttt{LExpr}' & \iff & [\![\texttt{LExpr}]\!]_{s,e} = [\![\texttt{LExpr}']\!]_{s,e} \\
e, s, d, \mathbf{a} \models_{\mathrm{A}} \texttt{LExpr} <\cdot\, \texttt{LExpr}' & \iff & [\![\texttt{LExpr}]\!]_{s,e}, [\![\texttt{LExpr}']\!]_{s,e} \in \mathbb{Z} \wedge [\![\texttt{LExpr}]\!]_{s,e} < [\![\texttt{LExpr}']\!]_{s,e} \\
e, s, d, \mathbf{a} \models_{\mathrm{A}} \texttt{LExpr} \in \texttt{LExpr}' & \iff & [\![\texttt{LExpr}]\!]_{s,e}, [\![\texttt{LExpr}']\!]_{s,e} \in \mathrm{S} \wedge [\![\texttt{LExpr}]\!]_{s,e} \in [\![\texttt{LExpr}']\!]_{s,e} \\
e, s, d, \mathbf{a} \models_{\mathrm{A}} (\texttt{LExpr}, \texttt{LExpr}') \in \mathrm{dom}(d) & \iff & [\![\texttt{LExpr}]\!]_{s,e}, [\![\texttt{LExpr}]\!]_{s,e} \in \mathrm{S} \wedge \\
 & & ([\![\texttt{LExpr}]\!]_{s,e}, [\![\texttt{LExpr}]\!]_{s,e}) \in \mathrm{dom}(d) \\
e, s, d, \mathbf{d} \models_{\mathrm{D}_1} \mathrm{cast}_{\mathrm{D}_2}(\mathrm{D}_1, P) & \iff & e, s, d, (\mathrm{cast}(\mathrm{D}_1, \mathbf{d})) \models_{\mathrm{D}_2} P \\
e, s, d, \mathbf{d} \models_{\mathrm{D}} \mathrm{flatten}(\texttt{LExpr}, P) & \iff & [\![\texttt{LExpr}]\!]_{s,e} \in \mathrm{S} \wedge e, s, d, (\mathrm{dosearch}([\![\texttt{LExpr}]\!]_{s,e}, \mathbf{d})) \models_{\mathrm{EF}} P \\
e, s, d, \mathbf{a} \models_{\mathrm{A}} \exists \texttt{VAR}.\, P & \iff & \exists \mathbf{a}'.\, e[\texttt{VAR} \mapsto \mathbf{a}'], s, \mathbf{a} \models_{\mathrm{A}} P
\end{array}
$$

Figure 7.7.: Satisfaction for the Remaining Formulae

The formula $\Box_{\otimes}(P, \mathrm{TP})$ means "everywhere at this forest level".

$$\Box_{\otimes}(P, \mathrm{TP}) \triangleq \neg(\Diamond_{\otimes}(\neg P, \mathrm{TP}))$$

Notice that because DOM Core Level One contains many more forest types than Featherweight DOM, we have to specify the type of the forest level that we're talking about when we use $\Diamond_{\otimes}$ and $\Box_{\otimes}$.

As with Featherweight DOM, it is also sometimes convenient to write formula without IDs, so we introduce the following shorthands:

$$
\begin{array}{rcl}
P_{\texttt{Id}} \{\!\!\{ P' \}\!\!\}_{\texttt{Tp}} [P'']P''' & \triangleq & \exists \texttt{AID}, \texttt{OWNER}, \texttt{FID}.\, P_{\texttt{Id}} \{\!\!\{ P' \}\!\!\}_{\mathrm{AID}}^{\mathrm{OWNER}}{}_{\texttt{Tp}} [P'']_{\texttt{FID}} P''' \\
P \{\!\!\{ P' \}\!\!\}_{\texttt{Tp}} [P'']P''' & \triangleq & \exists \texttt{ID}, \texttt{AID}, \texttt{OWNER}, \texttt{FID}.\, P_{\texttt{ID}} \{\!\!\{ P' \}\!\!\}_{\mathrm{AID}}^{\mathrm{OWNER}}{}_{\texttt{Tp}} [P'']_{\texttt{FID}} P''' \\
\ll P_{\texttt{Id}} \mapsto [P'] \gg & \triangleq & \exists \texttt{FID}, \texttt{OWNER}, \texttt{SPECIFIED}.\, \ll P_{\texttt{Id}} \mapsto [P']_{\texttt{FID}} \gg^{\texttt{OWNER}}_{\texttt{SPECIFIED}} \\
\ll P \mapsto [P'] \gg & \triangleq & \exists \texttt{ID}, \texttt{FID}, \texttt{OWNER}, \texttt{SPECIFIED}.\, \ll P_{\texttt{ID}} \mapsto [P']_{\texttt{FID}} \gg^{\texttt{OWNER}}_{\texttt{SPECIFIED}}
\end{array}
$$

7.5. Program Reasoning

The Program Reasoning for DOM Core Level 1 is similar to that presented for Featherweight DOM in Chapter 4.7.

Definition 68 (Local Hoare Triples)**.** Recall the evaluation relation $\rightsquigarrow$ relating configuration tuples $s, d, \mathbf{g}, \mathtt{C}$, terminal states $s, d, \mathbf{g}$, and faults given in Definition 48 and Section 6.3.2. The fault-avoiding partial correctness

interpretation of local Hoare Triples is given by:

$$\{P\}\mathtt{C}\{Q\} \iff$$

$$\begin{array}{l}(P{:}G \wedge Q{:}G \wedge \forall e, s, d, \mathbf{g}.\ e, s, d, \mathbf{g} \models_{\mathrm{G}} P \Rightarrow \\ s, d, \mathbf{g}, \mathtt{C} \not\rightsquigarrow \text{fault} \wedge \forall s', d', \mathbf{g}'.\, s, d, \mathbf{g}, \mathtt{C} \rightsquigarrow s', d', \mathbf{g}' \Rightarrow e, s', d', \mathbf{g}' \models_{\mathrm{G}} Q) \\ \vee \\ (P{:}\mathrm{D} \wedge Q{:}\mathrm{D} \wedge \forall e, s, d, \mathbf{g}.\ e, s, d, \mathbf{g} \models_{\mathrm{G}} <P>_{\mathrm{G}} \Rightarrow \\ s, d, \mathbf{g}, \mathtt{C} \not\rightsquigarrow \text{fault} \wedge \forall s', d', \mathbf{g}'.\, s, d, \mathbf{g}, \mathtt{C} \rightsquigarrow s', d', \mathbf{g}' \Rightarrow e, s', d', \mathbf{g}' \models_{\mathrm{G}} <Q>_{\mathrm{G}})\end{array}$$

where $\mathrm{D} \in \mathcal{N}$.

Notice that as with Featherweight DOM (Definition 23) our interpretation of the Hoare triples on trees coerces those trees to groves using $<->_{\mathrm{G}}$. This is necessary as $\rightsquigarrow$ is defined for configuration triples containing groves.

7.6. Command Axioms

For each command given in Section 6.3.2, we give command axioms here. These axioms follow the pattern of those given for Featherweight DOM in Definition 24.

7.6.1. Document

All the Document Interface commands simply introduce new structures to the grove. The axioms are correspondingly simple.

$$\begin{array}{c}\{<\text{“\#document”}_{\mathtt{Doc}} \not\{\ \varnothing_{\mathrm{EA}}\ \}\!\!\!>^{\mathbf{null}}_{\mathbf{null}_9}[\mathtt{F}{:}\mathrm{DF}]_{\mathtt{FID}}\mathbf{null}>_{\mathrm{G}} \wedge \mathtt{x} \doteq \mathtt{Y} \wedge \mathrm{eltName}(\mathtt{Name})\} \\ \mathtt{x} := \mathtt{createElement}(\mathtt{Doc}, \mathtt{Name}) \\ \left\{ \begin{array}{c} <\text{“\#document”}_{\mathtt{Doc}\{\mathtt{Y}/\mathtt{x}\}} \not\{\ \varnothing_{\mathrm{EA}}\ \}\!\!\!>^{\mathbf{null}}_{\mathbf{null}_9}[\mathtt{F}{:}\mathrm{DF}]_{\mathtt{FID}}\mathbf{null}>_{\mathrm{G}} \oplus \\ <\mathtt{Name}\{\mathtt{Y}/\mathtt{x}\}_{\mathtt{x}} \not\{\ \varnothing_{\mathrm{EA}}\ \}\!\!\!>^{\mathtt{Doc}\{\mathtt{Y}/\mathtt{x}\}}_{\mathtt{AID}'_1}[\varnothing_{\mathrm{EF}}]_{\mathtt{FID}'}\mathbf{null}>_{\mathrm{G}} \end{array} \right\}\end{array}$$

$$\begin{array}{c}\{<\text{“\#document”}_{\mathtt{Doc}} \not\{\ \varnothing_{\mathrm{EA}}\ \}\!\!\!>^{\mathbf{null}}_{\mathbf{null}_9}[\mathtt{F}{:}\mathrm{DF}]_{\mathtt{FID}}\mathbf{null}>_{\mathrm{G}} \wedge \mathtt{x} \doteq \mathtt{Y}\} \\ \mathtt{x} := \mathtt{createDocumentFragment}(\mathtt{Doc}) \\ \left\{ \begin{array}{c} <\text{“\#document”}_{\mathtt{Doc}\{\mathtt{Y}/\mathtt{x}\}} \not\{\ \varnothing_{\mathrm{EA}}\ \}\!\!\!>^{\mathbf{null}}_{\mathbf{null}_9}[\mathtt{F}{:}\mathrm{DF}]_{\mathtt{FID}}\mathbf{null}>_{\mathrm{G}} \oplus \\ <\text{“\#document-fragment”}_{\mathtt{x}} \not\{\ \varnothing_{\mathrm{EA}}\ \}\!\!\!>^{\mathtt{Doc}\{\mathtt{Y}/\mathtt{x}\}}_{\mathbf{null}_{11}}[\varnothing_{\mathrm{FRAGF}}]_{\mathtt{FID}'}\mathbf{null}>_{\mathrm{G}} \end{array} \right\}\end{array}$$

$$\{<\text{“\#document”}_{\texttt{Doc}}\{\!\!\{\ \varnothing_{\mathrm{EA}}\ \}\!\!\}^{\mathbf{null}}_{\mathbf{null}_9}[\texttt{F}{:}\mathrm{DF}]_{\texttt{FID}}\mathbf{null}>_{\mathrm{G}} \wedge \texttt{x} \doteq \texttt{Y}\}$$
$$\texttt{x} := \texttt{createTextNode}(\texttt{Doc}, \texttt{Data})$$
$$\left\{\begin{array}{c} <\text{“\#document”}_{\texttt{Doc}\{\texttt{Y/x}\}}\{\!\!\{\ \varnothing_{\mathrm{EA}}\ \}\!\!\}^{\mathbf{null}}_{\mathbf{null}_9}[\texttt{F}{:}\mathrm{DF}]_{\texttt{FID}}\mathbf{null}>_{\mathrm{G}} \oplus \\ <\text{“\#text”}_{\texttt{x}}\{\!\!\{\ \varnothing_{\mathrm{EA}}\ \}\!\!\}^{\texttt{Doc}\{\texttt{Y/x}\}}_{\mathbf{null}_3}[\varnothing_{\mathrm{TF}}]_{\texttt{FID}'}\texttt{Data}\{\texttt{Y/x}\}>_{\mathrm{G}} \end{array}\right\}$$

$$\{<\text{“\#document”}_{\texttt{Doc}}\{\!\!\{\ \varnothing_{\mathrm{EA}}\ \}\!\!\}^{\mathbf{null}}_{\mathbf{null}_9}[\texttt{F}{:}\mathrm{DF}]_{\texttt{FID}}\mathbf{null}>_{\mathrm{G}} \wedge \texttt{x} \doteq \texttt{Y}\}$$
$$\texttt{x} := \texttt{createComment}(\texttt{Doc}, \texttt{Data})$$
$$\left\{\begin{array}{c} <\text{“\#document”}_{\texttt{Doc}\{\texttt{Y/x}\}}\{\!\!\{\ \varnothing_{\mathrm{EA}}\ \}\!\!\}^{\mathbf{null}}_{\mathbf{null}_9}[\texttt{F}{:}\mathrm{DF}]_{\texttt{FID}}\mathbf{null}>_{\mathrm{G}} \oplus \\ <\text{“\#comment”}_{\texttt{x}}\{\!\!\{\ \varnothing_{\mathrm{EA}}\ \}\!\!\}^{\texttt{Doc}\{\texttt{Y/x}\}}_{\mathbf{null}_8}[\varnothing_{\mathrm{CF}}]_{\texttt{FID}'}\texttt{Data}\{\texttt{Y/x}\}>_{\mathrm{G}} \end{array}\right\}$$

$$\{<\text{“\#document”}_{\texttt{Doc}}\{\!\!\{\ \varnothing_{\mathrm{EA}}\ \}\!\!\}^{\mathbf{null}}_{\mathbf{null}_9}[\texttt{F}{:}\mathrm{DF}]_{\texttt{FID}}\mathbf{null}>_{\mathrm{G}} \wedge \texttt{x} \doteq \texttt{Y}\}$$
$$\texttt{x} := \texttt{createAttribute}(\texttt{Doc}, \texttt{Name})$$
$$\left\{\begin{array}{c} <\text{“\#document”}_{\texttt{Doc}\{\texttt{Y/x}\}}\{\!\!\{\ \varnothing_{\mathrm{EA}}\ \}\!\!\}^{\mathbf{null}}_{\mathbf{null}_9}[\texttt{F}{:}\mathrm{DF}]_{\texttt{FID}}\mathbf{null}>_{\mathrm{G}} \oplus \\ <\!\!\ll\texttt{Name}\{\texttt{Y/x}\}_{\texttt{x}} \mapsto [\varnothing_{\mathrm{AF}}]_{\texttt{FID}'} \gg^{\texttt{Doc}\{\texttt{Y/x}\}}_{\texttt{false}}>_{\mathrm{G}} \end{array}\right\}$$

$$\{\varnothing_{\mathrm{G}}\}$$
$$\texttt{x} := \texttt{createDocument}()$$
$$\{<\text{“\#document”}_{\texttt{x}}\{\!\!\{\ \varnothing_{\mathrm{EA}}\ \}\!\!\}^{\mathbf{null}}_{\mathbf{null}_9}[\texttt{F}{:}\mathrm{DF}]_{\texttt{FID}}\mathbf{null}>_{\mathrm{G}}\}$$

$$\{<\text{“\#document”}_{\texttt{Doc}}\{\!\!\{\ \varnothing_{\mathrm{EA}}\ \}\!\!\}^{\mathbf{null}}_{\mathbf{null}_9}[\texttt{F}{:}\mathrm{DF}]_{\texttt{FID}}\mathbf{null}>_{\mathrm{G}} \wedge \texttt{x} \doteq \texttt{Y}\}$$
$$\texttt{x} := \texttt{getElementsByTagName}(\texttt{Doc}, \texttt{TagName})$$
$$\left\{\begin{array}{c} <\text{“\#document”}_{\texttt{Doc}\{\texttt{Y/x}\}}\{\!\!\{\ \varnothing_{\mathrm{EA}}\ \}\!\!\}^{\mathbf{null}}_{\mathbf{null}_9}[\texttt{F}{:}\mathrm{DF}]_{\texttt{FID}}\mathbf{null}>_{\mathrm{G}} \oplus \\ <^{\texttt{Doc}\{\texttt{Y/x}\}}_{\texttt{TagName}\{\texttt{Y/x}\}_{\texttt{x}}}>_{\mathrm{G}} \end{array}\right\}$$

7.6.2. Node

The Node Interface is the most substantial of the interfaces in DOM Core Level 1 and provides the core functionality studied in Featherweight DOM.

Getters and Setters

The Node interface specifies several object-attributes which we represent as pairs of getter and setter commands. These commands are all similar.

$$\{\mathtt{NAME_N} \lessdot \mathtt{EA}{:}\mathrm{EA} \gtrdot_{\mathtt{AIDN}}^{\mathtt{IRN}}{}_{\mathtt{TP}} [\mathtt{F}{:}\mathrm{D}]_{\mathtt{FID}} \mathtt{VAL} \wedge \mathtt{nm} \doteq \mathtt{Y}\}$$
$$\mathtt{nm} := \mathtt{getNodeName}(\mathtt{N})$$
$$\{\mathtt{NAME}_{\mathtt{N}\{\mathtt{Y}/\mathtt{nm}\}} \lessdot \mathtt{EA}{:}\mathrm{EA} \gtrdot_{\mathtt{AIDN}}^{\mathtt{IRN}}{}_{\mathtt{TP}} [\mathtt{F}{:}\mathrm{D}]_{\mathtt{FID}} \mathtt{VAL} \wedge \mathtt{nm} \doteq \mathtt{NAME}\}$$

$$\{\ll \mathtt{NAME_N} \mapsto [\mathtt{AF}{:}\mathrm{AF}]_{\mathtt{FID}} \gg_{\mathtt{SPECIFIED}}^{\mathtt{DOC}} \wedge \mathtt{nm} \doteq \mathtt{Y}\}$$
$$\mathtt{nm} := \mathtt{getNodeName}(\mathtt{N})$$
$$\{\ll \mathtt{NAME}_{\mathtt{N}\{\mathtt{Y}/\mathtt{nm}\}} \mapsto [\mathtt{AF}{:}\mathrm{AF}]_{\mathtt{FID}} \gg_{\mathtt{SPECIFIED}}^{\mathtt{DOC}} \wedge \mathtt{nm} \doteq \mathtt{NAME}\}$$

$$\{\mathtt{NAME_N} \lessdot \mathtt{EA}{:}\mathrm{EA} \gtrdot_{\mathtt{AIDN}}^{\mathtt{IRN}}{}_{\mathtt{TP}} [\mathtt{F}{:}\mathrm{D}]_{\mathtt{FID}} \mathtt{VAL} \wedge \mathtt{v} \doteq \mathtt{Y}\}$$
$$\mathtt{v} := \mathtt{getNodeValueHelper}(\mathtt{N})$$
$$\{\mathtt{NAME}_{\mathtt{N}\{\mathtt{Y}/\mathtt{v}\}} \lessdot \mathtt{EA}{:}\mathrm{EA} \gtrdot_{\mathtt{AIDN}}^{\mathtt{IRN}}{}_{\mathtt{TP}} [\mathtt{F}{:}\mathrm{D}]_{\mathtt{FID}} \mathtt{VAL} \wedge \mathtt{v} \doteq \mathtt{VAL}\}$$

$$\{\mathtt{NAME_N} \lessdot \mathtt{EA}{:}\mathrm{EA} \gtrdot_{\mathtt{AIDN}}^{\mathtt{DOC}}{}_{\mathtt{TP}} [\mathtt{F}{:}\mathrm{D}]_{\mathtt{FID}} \mathtt{VAL} \wedge \mathtt{TP} \in \{2,3,4,7,8\}\}$$
$$\mathtt{setNodeValueHelper}(\mathtt{N},\mathtt{S})$$
$$\{\mathtt{NAME_N} \lessdot \mathtt{EA}{:}\mathrm{EA} \gtrdot_{\mathtt{AIDN}}^{\mathtt{DOC}}{}_{\mathtt{TP}} [\mathtt{F}{:}\mathrm{D}]_{\mathtt{FID}} \mathtt{S}\}$$

$$\{\mathtt{NAME_N} \lessdot \mathtt{EA}{:}\mathrm{EA} \gtrdot_{\mathtt{AIDN}}^{\mathtt{IRN}}{}_{\mathtt{TP}} [\mathtt{F}{:}\mathrm{D}]_{\mathtt{FID}} \mathtt{VAL} \wedge \mathtt{i} \doteq \mathtt{Y}\}$$
$$\mathtt{i} := \mathtt{getNodeType}(\mathtt{N})$$
$$\{\mathtt{NAME}_{\mathtt{N}\{\mathtt{Y}/\mathtt{i}\}} \lessdot \mathtt{EA}{:}\mathrm{EA} \gtrdot_{\mathtt{AIDN}}^{\mathtt{IRN}}{}_{\mathtt{TP}} [\mathtt{F}{:}\mathrm{D}]_{\mathtt{FID}} \mathtt{VAL} \wedge \mathtt{i} \doteq \mathtt{TP}\}$$

$$\{\ll \mathtt{NAME_N} \mapsto [\mathtt{AF}{:}\mathrm{AF}]_{\mathtt{FID}} \gg_{\mathtt{SPECIFIED}}^{\mathtt{DOC}} \wedge \mathtt{i} \doteq \mathtt{Y}\}$$
$$\mathtt{i} := \mathtt{getNodeType}(\mathtt{N})$$
$$\{\ll \mathtt{NAME}_{\mathtt{N}\{\mathtt{Y}/\mathtt{i}\}} \mapsto [\mathtt{AF}{:}\mathrm{AF}]_{\mathtt{FID}} \gg_{\mathtt{SPECIFIED}}^{\mathtt{DOC}} \wedge \mathtt{i} \doteq 2\}$$

$$\left\{\begin{array}{l} \mathtt{NAME_{ID}} \lessdot \mathtt{EA}{:}\mathrm{EA} \gtrdot_{\mathtt{AIDN}}^{\mathtt{IRN}}{}_{\mathtt{TP}} [\\ \qquad \mathtt{F}_1{:}\mathrm{D}_2 \otimes_{\mathrm{D}_2} \\ \qquad < \mathtt{NAME}'_{\mathtt{N}} \lessdot \mathtt{EA}'{:}\mathrm{EA} \gtrdot_{\mathtt{AIDN}'}^{\mathtt{DOC}}{}_{\mathtt{TP}'} [\mathtt{F}'{:}\mathrm{D}]_{\mathtt{FID}'} \mathtt{VAL}' >_{\mathrm{D}_2} \\ \qquad \otimes_{\mathrm{D}_2} \mathtt{F}_2{:}\mathrm{D}_2 \\ ]_{\mathtt{FID}} \mathtt{VAL} \\ \wedge \mathtt{p} \doteq \mathtt{Y} \end{array}\right\}$$
$$\mathtt{p} := \mathtt{getParentNode}(\mathtt{N});$$
$$\left\{\begin{array}{l} \mathtt{NAME_{ID}} \lessdot \mathtt{EA}{:}\mathrm{EA} \gtrdot_{\mathtt{AIDN}}^{\mathtt{IRN}}{}_{\mathtt{TP}} [\\ \qquad \mathtt{F}_1{:}\mathrm{D}_2 \otimes_{\mathrm{D}_2} \\ \qquad < \mathtt{NAME}'_{\mathtt{N}\{\mathtt{Y}/\mathtt{p}\}} \lessdot \mathtt{EA}'{:}\mathrm{EA} \gtrdot_{\mathtt{AIDN}'}^{\mathtt{DOC}}{}_{\mathtt{TP}'} [\mathtt{F}'{:}\mathrm{D}]_{\mathtt{FID}'} \mathtt{VAL}' >_{\mathrm{D}_2}]_{\mathtt{FID}} \mathtt{VAL} \\ \qquad \otimes_{\mathrm{D}_2} \mathtt{F}_2{:}\mathrm{D}_2 \\ \wedge \mathtt{p} \doteq \mathtt{ID} \end{array}\right\}$$

$$\{<\mathtt{NAME_N}\{\!\!\{\ \mathtt{EA}{:}\mathrm{EA}\ \}\!\!\}^{\mathtt{IRN}}_{\mathtt{AIDN_{TP}}}[\mathtt{F}{:}\mathrm{D}]_{\mathtt{FID}}\mathtt{VAL}>_{\mathrm{G}} \wedge \mathtt{p} \doteq \mathtt{Y}\}$$
$$\mathtt{p} := \mathtt{getParentNode(N)};$$
$$\{<\mathtt{NAME_{N\{Y/p\}}}\{\!\!\{\ \mathtt{EA}{:}\mathrm{EA}\ \}\!\!\}^{\mathtt{IRN}}_{\mathtt{AIDN_{TP}}}[\mathtt{F}{:}\mathrm{D}]_{\mathtt{FID}}\mathtt{VAL}>_{\mathrm{G}} \wedge \mathtt{p} \doteq \mathbf{null}\}$$

$$\{\ll\mathtt{NAME_N} \mapsto [\mathtt{AF}{:}\mathrm{AF}]_{\mathtt{FID}} \gg^{\mathtt{DOC}}_{\mathtt{SPECIFIED}} \wedge \mathtt{p} \doteq \mathtt{Y}\}$$
$$\mathtt{p} := \mathtt{getParentNode(N)};$$
$$\{\ll\mathtt{NAME_{N\{Y/p\}}} \mapsto [\mathtt{AF}{:}\mathrm{AF}]_{\mathtt{FID}} \gg^{\mathtt{DOC}}_{\mathtt{SPECIFIED}} \wedge \mathtt{p} \doteq \mathbf{null}\}$$

$$\left\{\begin{array}{l} \ll\mathtt{NAME_{ID}} \mapsto [\mathtt{AF_1}{:}\mathrm{AF} \otimes_{\mathrm{AF}} < \\ \quad \text{“\#text”}_{\mathtt{N}}\{\!\!\{\ \varnothing_{\mathrm{EA}}\ \}\!\!\}^{\mathtt{DOC}}_{\mathbf{null}_3}[\varnothing_{\mathrm{TF}}]_{\mathtt{FID'}}\mathtt{VAL'} \\ >_{\mathrm{AF}} \otimes_{\mathrm{AF}} \mathtt{AF_2}{:}\mathrm{AF}]_{\mathtt{FID}} \gg^{\mathtt{DOC}}_{\mathtt{SPECIFIED}} \wedge \mathtt{p} \doteq \mathtt{Y} \end{array}\right\}$$
$$\mathtt{p} := \mathtt{getParentNode(N)};$$
$$\left\{\begin{array}{l} \ll\mathtt{NAME_{ID}} \mapsto [\mathtt{AF_1}{:}\mathrm{AF} \otimes_{\mathrm{AF}} < \\ \quad \text{“\#text”}_{\mathtt{N\{Y/p\}}}\{\!\!\{\ \varnothing_{\mathrm{EA}}\ \}\!\!\}^{\mathtt{DOC}}_{\mathbf{null}_3}[\varnothing_{\mathrm{TF}}]_{\mathtt{FID'}}\mathtt{VAL'} \\ >_{\mathrm{AF}} \otimes_{\mathrm{AF}} \mathtt{AF_2}{:}\mathrm{AF}]_{\mathtt{FID}} \gg^{\mathtt{DOC}}_{\mathtt{SPECIFIED}} \wedge \mathtt{p} \doteq \mathtt{ID} \end{array}\right\}$$

$$\left\{\begin{array}{l} \mathtt{T}{:}\mathrm{DOC} \wedge \\ \left(\begin{array}{l} \text{“\#document”}_{\mathtt{ID}}\{\!\!\{\ \varnothing_{\mathrm{EA}}\ \}\!\!\}^{\mathbf{null}}_{\mathbf{null_9}}[\\ \quad <\mathtt{DNEL_1}{:}\mathrm{DNEL} \otimes_{\mathrm{DNEL}} \\ \quad <\text{“\#comment”}_{\mathtt{N}}\{\!\!\{\ \varnothing_{\mathrm{EA}}\ \}\!\!\}^{\mathtt{ID}}_{\mathbf{null}_8}[\varnothing_{\mathrm{CF}}]_{\mathtt{FID'}}\mathtt{VAL'}>_{\mathrm{DNEL}} \\ \quad \otimes_{\mathrm{DNEL}} \mathtt{DNEL_2}{:}\mathrm{DNEL}, \mathtt{DE}{:}\mathrm{DE}, \mathtt{DNEL}{:}\mathrm{DNEL}>_{\mathrm{DF}} \\ ]_{\mathtt{FID}}\mathbf{null} \vee \\ \text{“\#document”}_{\mathtt{ID}}\{\!\!\{\ \varnothing_{\mathrm{EA}}\ \}\!\!\}^{\mathbf{null}}_{\mathbf{null_9}}[\\ \quad <\mathtt{DNEL_1}{:}\mathrm{DNEL}, \\ \quad <\mathtt{NAME'_N}\{\!\!\{\ \mathtt{EA'}{:}\mathrm{EA}\ \}\!\!\}^{\mathtt{ID}}_{\mathtt{AID'}_1}[\mathtt{F'}{:}\mathrm{EF}]_{\mathtt{FID'}}\mathbf{null}>_{\mathrm{DE}}, \\ \quad \mathtt{DNEL_2}{:}\mathrm{DNEL}>_{\mathrm{DF}} \\ ]_{\mathtt{FID}}\mathbf{null} \vee \\ \text{“\#document”}_{\mathtt{ID}}\{\!\!\{\ \varnothing_{\mathrm{EA}}\ \}\!\!\}^{\mathbf{null}}_{\mathbf{null_9}}[\\ \quad <\mathtt{DNEL}{:}\mathrm{DNEL}, \mathtt{DE}{:}\mathrm{DE}, \mathtt{DNEL_1}{:}\mathrm{DNEL} \otimes_{\mathrm{DNEL}} \\ \quad <\text{“\#comment”}_{\mathtt{N}}\{\!\!\{\ \varnothing_{\mathrm{EA}}\ \}\!\!\}^{\mathtt{ID}}_{\mathbf{null}_9}[\varnothing_{\mathrm{CF}}]_{\mathtt{FID'}}\mathtt{VAL'}>_{\mathrm{DNEL}} \\ \quad \otimes_{\mathrm{DNEL}} \mathtt{DNEL_2}{:}\mathrm{DNEL}>_{\mathrm{DF}} \\ ]_{\mathtt{FID}}\mathbf{null} \end{array}\right) \end{array}\right\}$$
$$\mathtt{p} := \mathtt{getParentNode(N)};$$
$$\{\mathtt{T}{:}\mathrm{DOC} \wedge \mathtt{p} \doteq \mathtt{ID}\}$$

$$\{\mathtt{NAME_N}\{\!\!\{\ \mathtt{EA}{:}\mathrm{EA}\ \}\!\!\}^{\mathtt{IRN}}_{\mathtt{AIDN_{TP}}}[\mathtt{F}{:}\mathrm{D}]_{\mathtt{FID}}\mathtt{VAL} \wedge \mathtt{kids} \doteq \mathtt{Y}\}$$
$$\mathtt{kids} := \mathtt{getChildNodes(N)};$$
$$\{\mathtt{NAME_{N\{Y/kids\}}}\{\!\!\{\ \mathtt{EA}{:}\mathrm{EA}\ \}\!\!\}^{\mathtt{IRN}}_{\mathtt{AIDN_{TP}}}[\mathtt{F}{:}\mathrm{D}]_{\mathtt{FID}}\mathtt{VAL} \wedge \mathtt{kids} \doteq \mathtt{FID}\}$$

$$\{\ll \mathtt{NAME_N} \mapsto [\mathtt{AF}{:}\mathrm{AF}]_{\mathtt{FID}} \gg^{\mathtt{DOC}}_{\mathtt{SPECIFIED}} \wedge\ \mathtt{kids} \doteq \mathtt{Y}\}$$
$$\mathtt{kids} := \mathtt{getChildNodes}(\mathtt{N});$$
$$\{\ll \mathtt{NAME}_{\mathtt{N}\{\mathtt{Y}/\mathtt{kids}\}} \mapsto [\mathtt{AF}{:}\mathrm{AF}]_{\mathtt{FID}} \gg^{\mathtt{DOC}}_{\mathtt{SPECIFIED}} \wedge\ \mathtt{kids} \doteq \mathtt{FID}\}$$

$$\{\mathtt{NAME_N} \{\!\!\{\ \mathtt{EA}{:}\mathrm{EA}\ \}\!\!\}^{\mathtt{IRN}}_{\mathtt{AIDN_{TP}}} [\mathtt{F}{:}\mathrm{D}]_{\mathtt{FID}} \mathtt{VAL} \wedge\ \mathtt{ats} \doteq \mathtt{Y}\}$$
$$\mathtt{ats} := \mathtt{getAttributes}(\mathtt{N});$$
$$\{\mathtt{NAME}_{\mathtt{N}\{\mathtt{Y}/\mathtt{ats}\}} \{\!\!\{\ \mathtt{EA}{:}\mathrm{EA}\ \}\!\!\}^{\mathtt{IRN}}_{\mathtt{AIDN_{TP}}} [\mathtt{F}{:}\mathrm{D}]_{\mathtt{FID}} \mathtt{VAL} \wedge\ \mathtt{ats} \doteq \mathtt{AIDN}\}$$

$$\{\ll \mathtt{NAME_N} \mapsto [\mathtt{AF}{:}\mathrm{AF}]_{\mathtt{FID}} \gg^{\mathtt{DOC}}_{\mathtt{SPECIFIED}} \wedge\ \mathtt{ats} \doteq \mathtt{Y}\}$$
$$\mathtt{ats} := \mathtt{getAttributes}(\mathtt{N});$$
$$\{\ll \mathtt{NAME}_{\mathtt{N}\{\mathtt{Y}/\mathtt{ats}\}} \mapsto [\mathtt{AF}{:}\mathrm{AF}]_{\mathtt{FID}} \gg^{\mathtt{DOC}}_{\mathtt{SPECIFIED}} \wedge\ \mathtt{ats} \doteq \mathbf{null}\}$$

$$\{\mathtt{NAME_N} \{\!\!\{\ \mathtt{EA}{:}\mathrm{EA}\ \}\!\!\}^{\mathtt{IRN}}_{\mathtt{AIDN_{TP}}} [\mathtt{F}{:}\mathrm{D}]_{\mathtt{FID}} \mathtt{VAL} \wedge\ \mathtt{od} \doteq \mathtt{Y}\}$$
$$\mathtt{od} := \mathtt{getOwnerDocument}(\mathtt{N});$$
$$\{\mathtt{NAME}_{\mathtt{N}\{\mathtt{Y}/\mathtt{od}\}} \{\!\!\{\ \mathtt{EA}{:}\mathrm{EA}\ \}\!\!\}^{\mathtt{IRN}}_{\mathtt{AIDN_{TP}}} [\mathtt{F}{:}\mathrm{D}]_{\mathtt{FID}} \mathtt{VAL} \wedge\ \mathtt{od} \doteq \mathtt{IRN}\}$$

$$\{\ll \mathtt{NAME_N} \mapsto [\mathtt{AF}{:}\mathrm{AF}]_{\mathtt{FID}} \gg^{\mathtt{DOC}}_{\mathtt{SPECIFIED}} \wedge\ \mathtt{od} \doteq \mathtt{Y}\}$$
$$\mathtt{od} := \mathtt{getOwnerDocument}(\mathtt{N});$$
$$\{\ll \mathtt{NAME}_{\mathtt{N}\{\mathtt{Y}/\mathtt{od}\}} \mapsto [\mathtt{AF}{:}\mathrm{AF}]_{\mathtt{FID}} \gg^{\mathtt{DOC}}_{\mathtt{SPECIFIED}} \wedge\ \mathtt{od} \doteq \mathtt{DOC}\}$$

appendChild

In the case of the appendChild command, there are several cases to consider. First the case in which `Parent` is either an Element or a Document Fragment node, and `NewChild` is either an Element, a Text node or a Comment node.

$$\left\{\begin{array}{l}(\varnothing_{\mathrm{D}_1} \multimap (\mathtt{CG}{:}\mathrm{D}_2 \to \mathrm{G} \circ_{\mathrm{D}_2} \mathtt{NAME_{Parent}} \{\!\!\{\ \mathtt{EA}{:}\mathrm{EA}\ \}\!\!\}^{\mathtt{DOC}}_{\mathtt{AIDN_{TP}}} [\mathtt{F}{:}\mathrm{D}_3]_{\mathtt{FID}} \mathbf{null})) \\ \circ_{\mathrm{D}_1} (<\mathtt{NAME}'_{\mathtt{NewChild}} \{\!\!\{\ \mathtt{EA}'{:}\mathrm{EA}\ \}\!\!\}^{\mathtt{DOC}}_{\mathtt{AIDN}'_{\mathtt{TP}'}} [\mathtt{F}'{:}\mathrm{D}_4]_{\mathtt{FID}'} \mathtt{VAL}'>_{\mathrm{D}_1}) \\ \wedge\ \mathtt{n} \doteq \mathtt{Y} \wedge \mathtt{TP} \in \{1, 11\} \wedge \mathtt{TP}' \in \{1, 3, 8\}\end{array}\right\}$$
$$\mathtt{n} := \mathtt{appendChild}(\mathtt{Parent}, \mathtt{NewChild});$$
$$\left\{\begin{array}{l}(\mathtt{CG}{:}\mathrm{D}_2 \to \mathrm{G} \circ_{\mathrm{D}_2} \\ \mathtt{NAME}_{\mathtt{Parent}\{\mathtt{Y}/\mathtt{n}\}} \{\!\!\{\ \mathtt{EA}{:}\mathrm{EA}\ \}\!\!\}^{\mathtt{DOC}}_{\mathtt{AIDN_{TP}}} [\\ \qquad \mathtt{F}{:}\mathrm{D}_3 \otimes_{\mathrm{D}_3} \\ \qquad <\mathtt{NAME}'_{\mathtt{NewChild}\{\mathtt{Y}/\mathtt{n}\}} \{\!\!\{\ \mathtt{EA}'{:}\mathrm{EA}\ \}\!\!\}^{\mathtt{DOC}}_{\mathtt{AIDN}'_{\mathtt{TP}'}} [\mathtt{F}'{:}\mathrm{D}_4]_{\mathtt{FID}'} \mathtt{VAL}'>_{\mathrm{D}_3} \\ ]_{\mathtt{FID}} \mathbf{null}) \\ \wedge\ \mathtt{n} \doteq \mathtt{NewChild}\{\mathtt{Y}/\mathtt{n}\}\end{array}\right\}$$

Next, the case in which `Parent` is a Document node, `NewChild` is a Comment node and `Parent` has no document element.

$$\left\{\begin{array}{l}(\varnothing_{D_1} \multimap (\texttt{CG}{:}\mathrm{DOC}{\to}\mathrm{G} \circ_{\mathrm{DOC}} \text{“\#document”}_{\texttt{Parent}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathbf{null}}_{\mathbf{null_9}}[\\ \qquad <\texttt{F}{:}\mathrm{DNEL}, \varnothing_{\mathrm{DE}}, \varnothing_{\mathrm{DNEL}}>_{\mathrm{DF}} \\ ]_{\texttt{FID}}\mathbf{null})) \\ \circ_{D_1} (<\text{“\#comment”}_{\texttt{NewChild}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\texttt{Parent}}_{\mathbf{null_8}}[\varnothing_{\mathrm{CF}}]_{\texttt{FID}'}\texttt{VAL}'>_{D_1}) \\ \wedge \texttt{n} \doteq \texttt{Y}\end{array}\right\}$$

$$\texttt{n} := \texttt{appendChild}(\texttt{Parent}, \texttt{NewChild});$$

$$\left\{\begin{array}{l}(\texttt{CG}{:}\mathrm{DOC}{\to}\mathrm{G} \circ_{\mathrm{DOC}} (\text{“\#document”}_{\texttt{Parent}\{\texttt{Y/n}\}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathbf{null}}_{\mathbf{null_9}}[\\ \qquad <\texttt{F}{:}\mathrm{DNEL} \otimes_{\mathrm{DNEL}} \\ \qquad <\text{“\#comment”}_{\texttt{NewChild}\{\texttt{Y/n}\}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\texttt{Parent}\{\texttt{Y/n}\}}_{\mathbf{null_8}}[\varnothing_{\mathrm{CF}}]_{\texttt{FID}'}\texttt{VAL}>_{\mathrm{DNEL}}, \\ \qquad \varnothing_{\mathrm{DE}}, \varnothing_{\mathrm{DNEL}}>_{\mathrm{DF}} \\ ]_{\texttt{FID}}\mathbf{null})) \\ \wedge \texttt{n} \doteq \texttt{NewChild}\{\texttt{Y/n}\}\end{array}\right\}$$

Next, the case in which `Parent` is a Document node and `NewChild` is an Element.

$$\left\{\begin{array}{l}(\varnothing_{D_1} \multimap (\texttt{CG}{:}\mathrm{DOC}{\to}\mathrm{G} \circ_{\mathrm{DOC}} \text{“\#document”}_{\texttt{Parent}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathbf{null}}_{\mathbf{null_9}}[\\ \qquad <\texttt{F}{:}\mathrm{DNEL}, \varnothing_{\mathrm{DE}}, \varnothing_{\mathrm{DNEL}}>_{\mathrm{DF}} \ ]_{\texttt{FID}}\mathbf{null})) \\ \circ_{D_1} (<\texttt{S}_{\texttt{NewChild}} \{\!\!\{ \texttt{EA}{:}\mathrm{EA} \}\!\!\}^{\texttt{Parent}}_{\texttt{AID}'\mathbf{1}}[\texttt{F}{:}\mathrm{EF}]_{\texttt{FID}'}\mathbf{null}>_{D_1}) \\ \wedge \texttt{n} \doteq \texttt{Y}\end{array}\right\}$$

$$\texttt{n} := \texttt{appendChild}(\texttt{Parent}, \texttt{NewChild});$$

$$\left\{\begin{array}{l}(\texttt{CG}{:}\mathrm{DOC}{\to}\mathrm{G} \circ_{\mathrm{DOC}} (\text{“\#document”}_{\texttt{Parent}\{\texttt{Y/n}\}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathbf{null}}_{\mathbf{null_9}}[\\ \qquad <\texttt{F}{:}\mathrm{DNEL}, \\ \qquad <\texttt{S}_{\texttt{NewChild}\{\texttt{Y/n}\}} \{\!\!\{ \texttt{EA}{:}\mathrm{EA} \}\!\!\}^{\texttt{Parent}\{\texttt{Y/n}\}}_{\texttt{AID}'\mathbf{1}}[\texttt{F}{:}\mathrm{EF}]_{\texttt{FID}'}\mathbf{null}>_{\mathrm{DE}}, \\ \qquad \varnothing_{\mathrm{DNEL}}>_{\mathrm{DF}} \\ ]_{\texttt{FID}}\mathbf{null})) \\ \wedge \texttt{n} \doteq \texttt{NewChild}\{\texttt{Y/n}\}\end{array}\right\}$$

Next, the case in which `Parent` is a Document node, `NewChild` is a comment node and `Parent` does have a document element.

$$\left\{\begin{array}{l}(\varnothing_{\mathrm{D}_1} \multimap (\mathtt{CG}{:}\mathrm{DOC}{\to}\mathrm{G} \circ_{\mathrm{DOC}} \text{“\#document”}_{\mathtt{Parent}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathbf{null}}_{\mathbf{null_9}} [\\ \quad <\mathtt{F}_1{:}\mathrm{DNEL}, <\mathtt{ELE}{:}\mathrm{ELE}>_{\mathrm{DE}}, \mathtt{F}_2{:}\mathrm{DNEL}>_{\mathrm{DF}} \\ ]_{\mathtt{FID}}\mathbf{null})) \\ \circ_{\mathrm{D}_1} (<\text{“\#comment”}_{\mathtt{NewChild}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathtt{Parent}}_{\mathbf{null_8}} [\varnothing_{\mathrm{CF}}]_{\mathtt{FID'}}\mathtt{VAL'}>_{\mathrm{D}_1}) \\ \wedge\, \mathtt{n} \doteq \mathtt{Y}\end{array}\right\}$$

$$\mathtt{n} := \mathtt{appendChild}(\mathtt{Parent}, \mathtt{NewChild});$$

$$\left\{\begin{array}{l}(\mathtt{CG}{:}\mathrm{DOC}{\to}\mathrm{G} \circ_{\mathrm{DOC}} \\ (\text{“\#document”}_{\mathtt{Parent\{Y/n\}}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathbf{null}}_{\mathbf{null_9}} [\\ \quad <\mathtt{F}_1{:}\mathrm{DNEL}, <\mathtt{ELE}{:}\mathrm{ELE}>_{\mathrm{DE}}, \mathtt{F}_2{:}\mathrm{DNEL} \otimes_{\mathrm{DNEL}} \\ \quad <\text{“\#comment”}_{\mathtt{NewChild\{Y/n\}}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathtt{Parent\{Y/n\}}}_{\mathbf{null_8}} [\varnothing_{\mathrm{CF}}]_{\mathtt{FID'}}\mathtt{VAL}>_{\mathrm{DNEL}}, \\ \quad >_{\mathrm{DF}} \\ ]_{\mathtt{FID}}\mathbf{null})) \\ \wedge\, \mathtt{n} \doteq \mathtt{NewChild\{Y/n\}}\end{array}\right\}$$

Next, the case in which `Parent` is an Attr node, and `NewChild` is a Text node.

$$\left\{\begin{array}{l}(\varnothing_{\mathrm{D}_1} \multimap (\mathtt{CG}{:}\mathrm{ATTR}{\to}\mathrm{G} \circ_{\mathrm{ATTR}} \ll \mathrm{S}_{\mathtt{Parent}} \mapsto [\mathtt{F}{:}\mathrm{AF}]_{\mathtt{FID}} \gg^{\mathtt{DOC}}_{\mathtt{SPECIFIED}})) \\ \circ_{\mathrm{D}_1} (<\text{“\#text”}_{\mathtt{NewChild}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathtt{DOC}}_{\mathbf{null_3}} [\varnothing_{\mathrm{TF}}]_{\mathtt{FID'}}\mathtt{VAL}>_{\mathrm{D}_1}) \\ \wedge\, \mathtt{n} \doteq \mathtt{Y}\end{array}\right\}$$

$$\mathtt{n} := \mathtt{appendChild}(\mathtt{Parent}, \mathtt{NewChild});$$

$$\left\{\begin{array}{l}\mathtt{CG}{:}\mathrm{ATTR}{\to}\mathrm{G} \circ_{\mathrm{ATTR}} \\ \ll \mathrm{S}_{\mathtt{Parent}} \mapsto [\\ \quad \mathtt{F}{:}\mathrm{AF} \otimes_{\mathrm{AF}} <\text{“\#text”}_{\mathtt{NewChild}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathtt{DOC}}_{\mathbf{null_3}} [\varnothing_{\mathrm{TF}}]_{\mathtt{FID'}}\mathtt{VAL}>_{\mathrm{AF}} \\ ]_{\mathtt{FID}} \gg^{\mathtt{DOC}}_{\mathtt{SPECIFIED}} \\ \wedge\, \mathtt{n} \doteq \mathtt{NewChild\{Y/n\}}\end{array}\right\}$$

Now, we repeat all the above cases, but with a Document Fragment as `NewChild`. Recall from Section 6.3.2 that the effect of `appendChild` when `NewChild` is a Document Fragment is to move all of `NewChild`'s children to the end of `Parent`'s child list.

First, the case in which `Parent` is an Element node, and all the children

of NewChild are Elements, Text nodes and Commend nodes.

$$\left\{\begin{array}{l}(\mathtt{CG}{:}\mathrm{D}_2{\to}\mathrm{G} \circ_{\mathrm{D}_2} \mathrm{S}_{\mathtt{Parent}} \{\!\!\{ \mathtt{EA}{:}\mathrm{EA} \}\!\!\}^{\mathtt{DOC}}_{\mathtt{AID}_{\mathbf{1}}} [\mathtt{F}{:}\mathrm{EF}]_{\mathtt{FID}}\mathbf{null}) \\ \oplus <\text{“\#document-fragment”}_{\mathtt{NewChild}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathtt{DOC}}_{\mathbf{null}_{\mathbf{11}}} [\\ \quad \mathtt{F}'{:}\mathrm{FRAGF} \wedge \\ \quad \Box_{\otimes}((\exists \mathtt{NAME}'', \mathtt{ID}'', \mathtt{EA}'', \mathtt{AIDN}'', \mathtt{TP}'', \mathtt{F}'', \mathtt{FID}'', \mathtt{VAL}''. \\ \qquad \mathtt{NAME}''_{\mathtt{ID}''} \{\!\!\{ \mathtt{EA}''{:}\mathrm{EA} \}\!\!\}^{\mathtt{DOC}}_{\mathtt{AIDN}''\mathtt{TP}''} [\mathtt{F}''{:}\mathrm{D}]_{\mathtt{FID}''}\mathtt{VAL}'' \\ \qquad \wedge \mathtt{TP}'' \in \{1,3,8\}), \mathrm{FRAGF}) \\ ]_{\mathtt{FID}'}\mathbf{null}>_{\mathrm{G}} \\ \wedge \mathtt{n} \doteq \mathtt{Y}\end{array}\right\}$$

$$\mathtt{n} := \mathtt{appendChild}(\mathtt{Parent}, \mathtt{NewChild});$$

$$\left\{\begin{array}{l}(\mathtt{CG}{:}\mathrm{D}_2{\to}\mathrm{G} \circ_{\mathrm{D}_2} \\ \mathrm{S}_{\mathtt{Parent}\{\mathtt{Y/n}\}} \{\!\!\{ \mathtt{EA}{:}\mathrm{EA} \}\!\!\}^{\mathtt{DOC}}_{\mathtt{AID}_{\mathbf{1}}} [\\ \quad \mathtt{F}{:}\mathrm{EF} \otimes_{\mathrm{EF}} \mathrm{cast}_{\mathrm{FRAGF}}(\mathrm{EF}, \mathtt{F}'{:}\mathrm{FRAGF}) \\ ]_{\mathtt{FID}}\mathbf{null}) \oplus \\ <\text{“\#document-fragment”}_{\mathtt{NewChild}\{\mathtt{Y/n}\}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathtt{DOC}}_{\mathbf{null}_{\mathbf{11}}} [\varnothing_{\mathrm{FRAGF}}]_{\mathtt{FID}'}\mathbf{null}>_{\mathrm{G}} \\ \wedge \mathtt{n} \doteq \mathtt{NewChild}\{\mathtt{Y/n}\}\end{array}\right\}$$

Next, the case in which Parent is also a Document Fragment.

$$\left\{\begin{array}{l}(\mathtt{CG}{:}\mathrm{D}_2{\to}\mathrm{G} \circ_{\mathrm{D}_2} \text{“\#document-fragment”}_{\mathtt{Parent}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathtt{DOC}}_{\mathbf{null}_{\mathbf{11}}} [\mathtt{F}{:}\mathrm{FRAGF}]_{\mathtt{FID}}\mathbf{null}) \\ \oplus <\text{“\#document-fragment”}_{\mathtt{NewChild}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathtt{DOC}}_{\mathbf{null}_{\mathbf{11}}} [\mathtt{F}'{:}\mathrm{FRAGF}]_{\mathtt{FID}'}\mathbf{null}>_{\mathrm{G}} \\ \wedge \mathtt{n} \doteq \mathtt{Y}\end{array}\right\}$$

$$\mathtt{n} := \mathtt{appendChild}(\mathtt{Parent}, \mathtt{NewChild});$$

$$\left\{\begin{array}{l}(\mathtt{CG}{:}\mathrm{D}_2{\to}\mathrm{G} \circ_{\mathrm{D}_2} \\ \text{“\#document-fragment”}_{\mathtt{Parent}\{\mathtt{Y/n}\}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathtt{DOC}}_{\mathbf{null}_{\mathbf{11}}} [\\ \quad \mathtt{F}{:}\mathrm{FRAGF} \otimes_{\mathrm{FRAGF}} \mathtt{F}'{:}\mathrm{FRAGF} \\ ]_{\mathtt{FID}}\mathbf{null}) \\ \oplus <\text{“\#document-fragment”}_{\mathtt{NewChild}\{\mathtt{Y/n}\}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathtt{DOC}}_{\mathbf{null}_{\mathbf{11}}} [\varnothing_{\mathrm{FRAGF}}]_{\mathtt{FID}'}\mathtt{VAL}'>_{\mathrm{G}} \\ \wedge \mathtt{n} \doteq \mathtt{NewChild}\{\mathtt{Y/n}\}\end{array}\right\}$$

There are three cases in which Parent is a Document node. First, the case in which Parent is a Document node with no document element and all the children to append are Comment nodes.

$$
\left\{
\begin{array}{l}
<\text{“\#document”}_{\texttt{Parent}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathbf{null}}_{\mathbf{null_9}} [<\texttt{F}{:}\mathrm{DNEL}, \varnothing_{\mathrm{DE}}, \varnothing_{\mathrm{DNEL}}>_{\mathrm{DF}}]_{\texttt{FID}} \mathbf{null}>_{\mathrm{G}} \\
\oplus (<\text{“\#document-fragment”}_{\texttt{NewChild}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\texttt{Parent}}_{\mathbf{null_{11}}} [\\
\quad \texttt{F}'{:}\mathrm{FRAGF} \wedge \\
\quad \Box_{\otimes}((\exists \texttt{ID}'', \texttt{FID}'', \texttt{VAL}''. \\
\qquad \text{“\#comment”}_{\texttt{ID}''} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\texttt{Parent}}_{\mathbf{null_8}} [\varnothing_{\mathbf{CF}}]_{\texttt{FID}''\texttt{VAL}''}), \mathrm{FRAGF}) \\
]_{\texttt{FID}'} \mathbf{null}>_{\mathrm{G}}) \\
\wedge \texttt{n} \doteq \texttt{Y}
\end{array}
\right\}
$$

$$
\texttt{n} := \texttt{appendChild}(\texttt{Parent}, \texttt{NewChild});
$$

$$
\left\{
\begin{array}{l}
<\text{“\#document”}_{\texttt{Parent\{Y/n\}}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathbf{null}}_{\mathbf{null_9}} [\\
\quad <\texttt{F}{:}\mathrm{DNEL} \otimes_{\mathrm{DNEL}} \mathrm{cast}_{\mathrm{FRAGF}}(\mathrm{DNEL}, \texttt{F}'{:}\mathrm{FRAGF}), \\
\quad \varnothing_{\mathrm{DE}}, \varnothing_{\mathrm{DNEL}}>_{\mathrm{DF}} \\
]_{\texttt{FID}} \mathbf{null}>_{\mathrm{G}} \oplus \\
<\text{“\#document-fragment”}_{\texttt{NewChild\{Y/n\}}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\texttt{Parent\{Y/n\}}}_{\mathbf{null_{11}}} [\varnothing_{\mathrm{FRAGF}}]_{\texttt{FID}'} \mathbf{null}>_{\mathrm{G}} \\
\wedge \texttt{n} \doteq \texttt{NewChild\{Y/n\}}
\end{array}
\right\}
$$

Next, the case in which `Parent` is a Document node with no document element and one of the children to append is an Element node.

$$
\left\{
\begin{array}{l}
<\text{“\#document”}_{\texttt{Parent}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathbf{null}}_{\mathbf{null_9}} [<\texttt{F}{:}\mathrm{DNEL}, \varnothing_{\mathrm{DE}}, \varnothing_{\mathrm{DNEL}}>_{\mathrm{DF}}]_{\texttt{FID}} \mathbf{null}>_{\mathrm{G}} \\
\oplus (<\text{“\#document-fragment”}_{\texttt{NewChild}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\texttt{Parent}}_{\mathbf{null_{11}}} [\\
\quad (\texttt{F}'{:}\mathrm{FRAGF} \wedge \\
\quad \Box_{\otimes}((\exists \texttt{ID}'', \texttt{FID}'', \texttt{VAL}''. \\
\qquad \text{“\#comment”}_{\texttt{ID}''} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\texttt{Parent}}_{\mathbf{null_8}} [\varnothing_{\mathbf{CF}}]_{\texttt{FID}''\texttt{VAL}''}), \mathrm{FRAGF})) \\
\quad \otimes_{\mathrm{FRAGF}} <\mathrm{S}_{\texttt{id}'''} \{\!\!\{ \texttt{EA}'''{:}\mathrm{EA} \}\!\!\}^{\texttt{Parent}}_{\texttt{AID}'''_{\mathbf{1}}} [\texttt{EF}'''{:}\mathrm{EF}]_{\texttt{FID}''} \mathbf{null}>_{\mathrm{FRAGF}} \otimes_{\mathrm{FRAGF}} \\
\quad (\texttt{F}''''{:}\mathrm{FRAGF} \wedge \\
\quad \Box_{\otimes}(((\exists \texttt{ID}'''', \texttt{FID}'''', \texttt{VAL}''''. \\
\qquad \text{“\#comment”}_{\texttt{ID}''''} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\texttt{Parent}}_{\mathbf{null_8}} [\varnothing_{\mathbf{CF}}]_{\texttt{FID}''''\texttt{VAL}''''}), \mathrm{FRAGF})) \\
]_{\texttt{FID}'} \mathbf{null}>_{\mathrm{G}}) \\
\wedge \texttt{n} \doteq \texttt{Y}
\end{array}
\right\}
$$

$$
\texttt{n} := \texttt{appendChild}(\texttt{Parent}, \texttt{NewChild});
$$

$$
\left\{
\begin{array}{l}
<\text{“\#document”}_{\texttt{Parent\{Y/n\}}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathbf{null}}_{\mathbf{null_9}} [\\
\quad <\texttt{F}{:}\mathrm{DNEL} \otimes_{\mathrm{DNEL}} \mathrm{cast}_{\mathrm{FRAGF}}(\mathrm{DNEL}, \texttt{F}'{:}\mathrm{FRAGF}), \\
\quad <\mathrm{S}_{\texttt{id}'''} \{\!\!\{ \texttt{EA}'''{:}\mathrm{EA} \}\!\!\}^{\texttt{Parent\{Y/n\}}}_{\texttt{AID}'''_{\mathbf{1}}} [\texttt{EF}'''{:}\mathrm{EF}]_{\texttt{FID}'''} \mathbf{null}>_{\mathrm{DE}}, \\
\quad \mathrm{cast}_{\mathrm{FRAGF}}(\mathrm{DNEL}, \texttt{F}''''{:}\mathrm{FRAGF})>_{\mathrm{DF}} \\
]_{\texttt{FID}} \mathbf{null}>_{\mathrm{G}} \oplus \\
<\text{“\#document-fragment”}_{\texttt{NewChild\{Y/n\}}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\texttt{Parent\{Y/n\}}}_{\mathbf{null_{11}}} [\varnothing_{\mathrm{FRAGF}}]_{\texttt{FID}'} \mathbf{null}>_{\mathrm{G}} \\
\wedge \texttt{n} \doteq \texttt{NewChild\{Y/n\}}
\end{array}
\right\}
$$

Next, the case in which `Parent` is a Document node which already has a document element, and all the nodes to append are Comment nodes.

$$\left\{\begin{array}{l} <\text{“\#document”}_{\texttt{Parent}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathbf{null}}_{\mathbf{null_9}}[\\ \quad <\mathtt{F}_1{:}\mathrm{DNEL}, <\mathtt{DE}{:}\mathrm{DE}>_{\mathrm{DE}}, \mathtt{F}_2{:}\mathrm{DNEL}>_{\mathrm{DF}} \ ]_{\mathtt{FID}}\mathbf{null}>_{\mathrm{G}} \\ \oplus (<\text{“\#document-fragment”}_{\texttt{NewChild}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\texttt{Parent}}_{\mathbf{null_{11}}}[\\ \quad \mathtt{F}'{:}\mathrm{FRAGF} \wedge \\ \quad \Box_{\otimes}(((\exists \mathtt{ID}'', \mathtt{FID}'', \mathtt{VAL}''. \\ \qquad \text{“\#comment”}_{\mathtt{ID}''} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\texttt{Parent}}_{\mathbf{null_8}}[\varnothing_{\mathbf{CF}}]_{\mathtt{FID}'' \mathtt{VAL}''}), \mathrm{FRAGF}) \\ ]_{\mathtt{FID}'}\mathbf{null}>_{\mathrm{G}}) \\ \wedge \mathtt{n} \doteq \mathtt{Y} \end{array}\right\}$$

$$\mathtt{n} := \mathtt{appendChild}(\mathtt{Parent}, \mathtt{NewChild});$$

$$\left\{\begin{array}{l} <\text{“\#document”}_{\texttt{Parent}\{\texttt{Y/n}\}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathbf{null}}_{\mathbf{null_9}}[\\ \quad <\mathtt{F}_1{:}\mathrm{DNEL}, \\ \quad <\mathtt{DE}{:}\mathrm{DE}>_{\mathrm{DE}}, \mathtt{F}_2{:}\mathrm{DNEL} \\ \quad \otimes_{\mathrm{DNEL}} \mathrm{cast}_{\mathrm{FRAGF}}(\mathrm{DNEL}, \mathtt{F}'{:}\mathrm{FRAGF}) \varnothing_{\mathrm{DNEL}}>_{\mathrm{DF}} \\ ]_{\mathtt{FID}}\mathbf{null}>_{\mathrm{G}} \oplus \\ <\text{“\#document-fragment”}_{\texttt{NewChild}\{\texttt{Y/n}\}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\texttt{Parent}\{\texttt{Y/n}\}}_{\mathbf{null_{11}}}[\varnothing_{\mathrm{FRAGF}}]_{\mathtt{FID}'}\mathbf{null}>_{\mathrm{G}} \\ \wedge \mathtt{n} \doteq \mathtt{NewChild}\{\mathtt{Y/n}\} \end{array}\right\}$$

Finally, the case in which `Parent` is an Attr node, and all the children to append are Text nodes.

$$\left\{\begin{array}{l} (\mathtt{CG}{:}\mathrm{ATTR}{\to}\mathrm{G} \circ_{\mathrm{ATTR}} \ll \mathtt{S}_{\texttt{Parent}} \mapsto [\mathtt{F}{:}\mathrm{AF}]_{\mathtt{FID}} \gg^{\texttt{DOC}}_{\texttt{SPECIFIED}}) \\ \oplus (<\text{“\#document-fragment”}_{\texttt{NewChild}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\texttt{DOC}}_{\mathbf{null_{11}}}[\\ \quad \mathtt{F}'{:}\mathrm{FRAGF} \wedge \\ \quad \Box_{\otimes}((\exists \mathtt{ID}'', \mathtt{FID}'', \mathtt{VAL}''. \\ \qquad \text{“\#text”}_{\mathtt{id}''} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\texttt{DOC}}_{\mathbf{null_3}}[\varnothing_{\mathrm{TF}}]_{\mathtt{FID}'' \mathtt{VAL}''}, \mathrm{FRAGF}) \\ ]_{\mathtt{FID}'}\mathbf{null}>_{\mathrm{G}}) \\ \wedge \mathtt{n} \doteq \mathtt{Y} \end{array}\right\}$$

$$\mathtt{n} := \mathtt{appendChild}(\mathtt{Parent}, \mathtt{NewChild});$$

$$\left\{\begin{array}{l} \mathtt{CG}{:}\mathrm{ATTR}{\to}\mathrm{G} \circ_{\mathrm{ATTR}} \\ \ll \mathtt{S}_{\texttt{Parent}\{\texttt{Y/n}\}} \mapsto [\mathtt{F}{:}\mathrm{AF} \otimes_{\mathrm{AF}} \mathrm{cast}_{\mathrm{FRAGF}}(\mathrm{AF}, \mathtt{F}'{:}\mathrm{FRAGF})]_{\mathtt{FID}} \gg^{\texttt{DOC}}_{\texttt{SPECIFIED}} \\ \oplus <\text{“\#document-fragment”}_{\texttt{NewChild}\{\texttt{Y/n}\}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\texttt{DOC}}_{\mathbf{null_{11}}}[\varnothing_{\mathrm{FRAGF}}]_{\mathtt{FID}'}\mathbf{null}>_{\mathrm{G}} \\ \wedge \mathtt{n} \doteq \mathtt{NewChild}\{\mathtt{Y/n}\} \end{array}\right\}$$

removeChild

The cases for the `removeChild` command mirror the cases for `appendChild` when `NewChild` is not a Document Fragment.

First, the usual case in which `Parent` is neither a Document nor an Attr node:

$$\left\{\begin{array}{l}(\mathtt{CG}{:}\mathrm{D}_1{\to}\mathrm{G} \circ_{\mathrm{D}_1} \mathtt{NAME}_{\mathtt{Parent}} \{\!\!\{ \mathtt{EA}{:}\mathrm{EA} \}\!\!\}^{\mathtt{DOC}}_{\mathtt{AIDN}_{\mathtt{TP}}} [\\ \quad \mathtt{F}_1{:}\mathrm{D}_2 \otimes_{\mathrm{D}_2} \\ \quad <\mathtt{NAME}_{\mathtt{OldChild}} \{\!\!\{ \mathtt{EA}'{:}\mathrm{EA} \}\!\!\}^{\mathtt{DOC}}_{\mathtt{AIDN}'_{\mathtt{TP}}} [\mathtt{F}'{:}\mathrm{D}_3]_{\mathtt{FID}'} \mathtt{VAL}' >_{\mathrm{D}_2} \\ \quad \otimes_{\mathrm{D}_2} \mathtt{F}_2{:}\mathrm{D}_2 \\ ]_{\mathtt{FID}} \mathbf{null}) \\ \wedge\, \mathtt{n} \doteq \mathtt{Y}\end{array}\right\}$$

$$\mathtt{n} := \mathtt{removeChild}(\mathtt{Parent}, \mathtt{OldChild})$$

$$\left\{\begin{array}{l}(\mathtt{CG}{:}\mathrm{D}_1{\to}\mathrm{G} \circ_{\mathrm{D}_1} \mathtt{NAME}_{\mathtt{Parent}\{\mathtt{Y/n}\}} \{\!\!\{ \mathtt{EA}{:}\mathrm{EA} \}\!\!\}^{\mathtt{DOC}}_{\mathtt{AIDN}_{\mathtt{TP}}} [\\ \quad \mathtt{F}_1{:}\mathrm{D}_2 \otimes_{\mathrm{D}_2} \mathtt{F}_2{:}\mathrm{D}_2 \\ ]_{\mathtt{FID}} \mathbf{null}) \oplus \\ <\mathtt{NAME}_{\mathtt{OldChild}\{\mathtt{Y/n}\}} \{\!\!\{ \mathtt{EA}'{:}\mathrm{EA} \}\!\!\}^{\mathtt{DOC}}_{\mathtt{AIDN}'_{\mathtt{TP}}} [\mathtt{F}'{:}\mathrm{D}_3]_{\mathtt{FID}'} \mathtt{VAL}' >_{\mathrm{G}} \\ \wedge\, \mathtt{n} \doteq \mathtt{OldChild}\{\mathtt{Y/n}\}\end{array}\right\}$$

Next, the case in which `Parent` is an Attr node:

$$\left\{\begin{array}{l}(\mathtt{CG}{:}\mathrm{ATTR}{\to}\mathrm{G} \circ_{\mathrm{ATTR}} \ll \mathtt{S}_{\mathtt{Parent}} \mapsto [\\ \quad \mathtt{F}_1{:}\mathrm{AF} \otimes_{\mathrm{AF}} \\ \quad <\text{“\#text”}_{\mathtt{OldChild}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathtt{DOC}}_{\mathbf{null_3}} [\varnothing_{\mathrm{TF}}]_{\mathtt{FID}'} \mathtt{VAL}' >_{\mathrm{AF}} \\ \quad \otimes_{\mathrm{AF}} \mathtt{F}_2{:}\mathrm{AF} \\ ]_{\mathtt{FID}} \gg^{\mathtt{DOC}}_{\mathtt{SPECIFIED}}) \\ \wedge\, \mathtt{n} \doteq \mathtt{Y}\end{array}\right\}$$

$$\mathtt{n} := \mathtt{removeChild}(\mathtt{Parent}, \mathtt{OldChild})$$

$$\left\{\begin{array}{l}(\mathtt{CG}{:}\mathrm{ATTR}{\to}\mathrm{G} \circ_{\mathrm{ATTR}} \ll \mathtt{S}_{\mathtt{Parent}\{\mathtt{Y/n}\}} \mapsto [\\ \quad \mathtt{F}_1{:}\mathrm{AF} \otimes_{\mathrm{AF}} \mathtt{F}_2{:}\mathrm{AF} \\ ]_{\mathtt{FID}} \gg^{\mathtt{DOC}}_{\mathtt{SPECIFIED}}) \oplus \\ <\text{“\#text”}_{\mathtt{OldChild}\{\mathtt{Y/n}\}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathtt{DOC}}_{\mathtt{AIDN}'_{\mathtt{TP}}} [\varnothing_{\mathrm{TF}}]_{\mathtt{FID}'} \mathtt{VAL}' >_{\mathrm{G}} \\ \wedge\, \mathtt{n} \doteq \mathtt{OldChild}\{\mathtt{Y/n}\}\end{array}\right\}$$

Finally, we present the three cases in which `Parent` is a Document node.

First, the case in which `OldChild` is a comment from the leftmost DNEL structure.

$$\left\{\begin{array}{l} <\text{“\#document”}_{\texttt{Parent}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathbf{null}}_{\mathbf{null_9}}[< \\ \quad \mathtt{F}_1{:}\mathrm{DNEL} \otimes_{\mathrm{DNEL}} \\ \quad <\text{“\#comment”}_{\texttt{OldChild}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\texttt{Parent}}_{\mathbf{null_8}}[\varnothing_{\mathrm{CF}}]_{\texttt{FID}'\texttt{VAL}'} >_{\mathrm{DNEL}} \\ \quad \otimes_{\mathrm{DNEL}} \mathtt{F}_2{:}\mathrm{DNEL}, \mathtt{F}_3{:}\mathrm{DE}, \mathtt{F}_4{:}\mathrm{DNEL} \\ >_{\mathrm{DF}}]_{\texttt{FID}}\mathbf{null} >_{\mathrm{G}} \\ \wedge\, \mathtt{n} \doteq \mathtt{Y} \end{array}\right\}$$

$$\mathtt{n} := \mathtt{removeChild}(\mathtt{Parent}, \mathtt{OldChild})$$

$$\left\{\begin{array}{l} <\text{“\#document”}_{\texttt{Parent}\{\texttt{Y/n}\}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathbf{null}}_{\mathbf{null_9}}[< \\ \quad \mathtt{F}_1{:}\mathrm{DNEL} \otimes_{\mathrm{DNEL}} \mathtt{F}_2{:}\mathrm{DNEL}, \mathtt{F}_3{:}\mathrm{DE}, \mathtt{F}_4{:}\mathrm{DNEL} \\ >_{\mathrm{DF}}]_{\texttt{FID}}\mathbf{null}) \oplus \\ <\text{“\#comment”}_{\texttt{OldChild}\{\texttt{Y/n}\}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\texttt{Parent}\{\texttt{Y/n}\}}_{\mathbf{null_8}}[\varnothing_{\mathrm{CF}}]_{\texttt{FID}'\texttt{VAL}'} >_{\mathrm{G}} \\ \wedge\, \mathtt{n} \doteq \mathtt{OldChild}\{\mathtt{Y/n}\} \end{array}\right\}$$

Next, the case in which `Parent` is a Document node and `OldChild` is the document element.

$$\left\{\begin{array}{l} <\text{“\#document”}_{\texttt{Parent}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathbf{null}}_{\mathbf{null_9}}[< \\ \quad \mathtt{F}_1{:}\mathrm{DNEL}, \\ \quad <\mathtt{NAME}_{\texttt{OldChild}} \{\!\!\{ \mathtt{EA}{:}\mathrm{EA} \}\!\!\}^{\texttt{Parent}}_{\texttt{AID}'\mathbf{1}}[\mathtt{F}{:}\mathrm{EF}]_{\texttt{FID}'}\mathbf{null} >_{\mathrm{DE}}, \\ \quad \mathtt{F}_2{:}\mathrm{DNEL} \\ >_{\mathrm{DF}}]_{\texttt{FID}}\mathbf{null} >_{\mathrm{G}} \\ \wedge\, \mathtt{n} \doteq \mathtt{Y} \end{array}\right\}$$

$$\mathtt{n} := \mathtt{removeChild}(\mathtt{Parent}, \mathtt{OldChild})$$

$$\left\{\begin{array}{l} <\text{“\#document”}_{\texttt{Parent}\{\texttt{Y/n}\}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathbf{null}}_{\mathbf{null_9}}[< \\ \quad \mathtt{F}_1{:}\mathrm{DNEL} \otimes_{\mathrm{DNEL}} \mathtt{F}_2{:}\mathrm{DNEL}, \varnothing_{\mathrm{DE}}, \varnothing_{\mathrm{DNEL}} \\ >_{\mathrm{DF}}]_{\texttt{FID}}\mathbf{null}) \oplus \\ <\mathtt{NAME}_{\texttt{OldChild}\{\texttt{Y/n}\}} \{\!\!\{ \mathtt{EA}{:}\mathrm{EA} \}\!\!\}^{\texttt{Parent}\{\texttt{Y/n}\}}_{\texttt{AID}'\mathbf{1}}[\mathtt{F}{:}\mathrm{EF}]_{\texttt{FID}'}\mathbf{null} >_{\mathrm{G}} \\ \wedge\, \mathtt{n} \doteq \mathtt{OldChild}\{\mathtt{Y/n}\} \end{array}\right\}$$

Finally, the case in which `Parent` is a Document node and `OldChild` is a Comment node from the rightmost DNEL structure.

$$\left\{\begin{array}{l} <\text{“\#document”}_{\texttt{Parent}}\{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathbf{null}}_{\mathbf{null_9}}[< \\ \quad \mathtt{F}_1{:}\mathrm{DNEL}, \mathtt{F}_2{:}\mathrm{DE}, \mathtt{F}_3{:}\mathrm{DNEL} \otimes_{\mathrm{DNEL}} \\ \quad <\text{“\#comment”}_{\texttt{OldChild}}\{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\texttt{Parent}}_{\mathbf{null_8}}[\varnothing_{\mathrm{CF}}]_{\texttt{FID}'}\texttt{VAL}'>_{\mathrm{DNEL}} \\ \quad \otimes_{\mathrm{DNEL}} \mathtt{F}_4{:}\mathrm{DNEL} \\ >_{\mathrm{DF}}]_{\texttt{FID}}\mathbf{null}>_{\mathrm{G}} \\ \wedge\, \mathtt{n} \doteq \mathtt{Y} \end{array}\right\}$$

$$\mathtt{n} := \mathtt{removeChild}(\mathtt{Parent}, \mathtt{OldChild})$$

$$\left\{\begin{array}{l} <\text{“\#document”}_{\texttt{Parent}\{\texttt{Y/n}\}}\{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\mathbf{null}}_{\mathbf{null_9}}[< \\ \quad \mathtt{F}_1{:}\mathrm{DNEL}, \mathtt{F}_2{:}\mathrm{DE}, \mathtt{F}_3{:}\mathrm{DNEL} \otimes_{\mathrm{DNEL}} \mathtt{F}_4{:}\mathrm{DNEL} \\ >_{\mathrm{DF}}]_{\texttt{FID}}\mathbf{null}) \oplus \\ <\text{“\#comment”}_{\texttt{OldChild}\{\texttt{Y/n}\}}\{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}^{\texttt{Parent}\{\texttt{Y/n}\}}_{\mathbf{null_8}}[\varnothing_{\mathrm{CF}}]_{\texttt{FID}'}\texttt{VAL}'>_{\mathrm{G}} \\ \wedge\, \mathtt{n} \doteq \mathtt{OldChild}\{\mathtt{Y/n}\} \end{array}\right\}$$

7.6.3. NodeList

The NodeList interface contains only one essential command, `item`, but that command has a large number of cases to consider. First we present the two possible cases in which the parent node is not a Document node an Attr node or an Element Search structure.

$$\left\{\begin{array}{l} (\mathtt{NAME}_{\mathtt{ID}}\{\!\!\{ \mathtt{EA}{:}\mathrm{EA} \}\!\!\}^{\mathtt{DOC}}_{\mathtt{TP}}[\\ \quad \mathtt{F}_1{:}\mathrm{D}_2 \otimes_{\mathrm{D}_2} \\ \quad <\mathtt{NAME}_{\mathtt{ID}'}\{\!\!\{ \mathtt{EA}'{:}\mathrm{EA} \}\!\!\}^{\mathtt{DOC}}_{\mathtt{TP}'}[\mathtt{F}'{:}\mathrm{D}]_{\mathtt{FID}'}\mathtt{VAL}'>_{\mathrm{D}_2} \\ \quad \otimes_{\mathrm{D}_2} \mathtt{F}_2{:}\mathrm{D}_2 \\ ]_{\mathtt{List}}\mathbf{null}) \\ \wedge\, \mathtt{n} \doteq \mathtt{Y} \wedge \mathrm{len}(\mathtt{F}_1) \doteq \mathtt{Int} \end{array}\right\}$$

$$\mathtt{n} := \mathtt{item}(\mathtt{List}, \mathtt{Int})$$

$$\left\{\begin{array}{l} (\mathtt{NAME}_{\mathtt{ID}}\{\!\!\{ \mathtt{EA}{:}\mathrm{EA} \}\!\!\}^{\mathtt{DOC}}_{\mathtt{TP}}[\\ \quad \mathtt{F}_1{:}\mathrm{D}_2 \otimes_{\mathrm{D}_2} \\ \quad <\mathtt{NAME}_{\mathtt{ID}'}\{\!\!\{ \mathtt{EA}'{:}\mathrm{EA} \}\!\!\}^{\mathtt{DOC}}_{\mathtt{TP}'}[\mathtt{F}'{:}\mathrm{D}]_{\mathtt{FID}'}\mathtt{VAL}'>_{\mathrm{D}_2} \\ \quad \otimes_{\mathrm{D}_2} \mathtt{F}_2{:}\mathrm{D}_2 \\ ]_{\mathtt{List}\{\mathtt{Y/n}\}}\mathbf{null}) \\ \wedge\, \mathtt{n} \doteq \mathtt{ID}' \end{array}\right\}$$

$$\left\{\begin{array}{l}(\mathtt{NAME_{ID}}\{\!\!\{\ \mathtt{EA}{:}\mathrm{EA}\ \}\!\!\}_{\mathtt{AIDN_{TP}}}^{\mathtt{DOC}}[\mathtt{F}{:}\mathrm{D}]_{\mathtt{List}}\mathbf{null}) \\ \wedge\, \mathtt{n} \doteq \mathtt{Y} \wedge (\mathrm{len}(\mathtt{F}) <\!\doteq \mathtt{Int} \vee \mathtt{Int} <\!\cdot\, 0)\end{array}\right\}$$

$$\mathtt{n} := \mathtt{item}(\mathtt{List}, \mathtt{Int})$$

$$\left\{\begin{array}{l}(\mathtt{NAME_{ID}}\{\!\!\{\ \mathtt{EA}{:}\mathrm{EA}\ \}\!\!\}_{\mathtt{AIDN_{TP}}}^{\mathtt{DOC}}[\mathtt{F}{:}\mathrm{D}]_{\mathtt{List}\{\mathtt{Y/n}\}}\mathbf{null}) \\ \wedge\, \mathtt{n} \doteq \mathbf{null}\end{array}\right\}$$

Next, the two possible cases in which the parent node is an Attr node.

$$\left\{\begin{array}{l}(\ll\mathtt{NAME_{ID}} \mapsto [\\ \quad \mathtt{F_1}{:}\mathrm{AF} \otimes_{\mathrm{AF}} \\ \quad <\text{“\#text”}_{\mathtt{ID'}}\{\!\!\{\ \varnothing_{\mathrm{EA}}\ \}\!\!\}_{\mathbf{null_3}}^{\mathtt{DOC}}[\varnothing_{\mathrm{TF}}]_{\mathtt{FID'}}\mathtt{VAL'}>_{\mathrm{AF}} \\ \quad \otimes_{\mathrm{AF}} \mathtt{F_2}{:}\mathrm{AF} \\ ]_{\mathtt{List}} \gg_{\mathtt{SPECIFIED}}^{\mathtt{DOC}}) \\ \wedge\, \mathtt{n} \doteq \mathtt{Y} \wedge \mathrm{len}(\mathtt{F_1}) \doteq \mathtt{Int}\end{array}\right\}$$

$$\mathtt{n} := \mathtt{item}(\mathtt{List}, \mathtt{Int})$$

$$\left\{\begin{array}{l}(\ll\mathtt{NAME_{ID}} \mapsto [\\ \quad \mathtt{F_1}{:}\mathrm{AF} \otimes_{\mathrm{AF}} \\ \quad <\text{“\#text”}_{\mathtt{ID'}}\{\!\!\{\ \varnothing_{\mathrm{EA}}\ \}\!\!\}_{\mathtt{AIDN'_3}}^{\mathtt{DOC}}[\varnothing_{\mathrm{TF}}]_{\mathtt{FID'}}\mathtt{VAL'}>_{\mathrm{AF}} \\ \quad \otimes_{\mathrm{AF}} \mathtt{F_2}{:}\mathrm{AF} \\ ]_{\mathtt{List}\{\mathtt{Y/n}\}} \gg_{\mathtt{SPECIFIED}}^{\mathtt{DOC}}) \\ \wedge\, \mathtt{n} \doteq \mathtt{ID'}\end{array}\right\}$$

$$\left\{\begin{array}{l}(\ll\mathtt{NAME_{ID}} \mapsto [\mathtt{F}{:}\mathrm{AF}]_{\mathtt{List}} \gg_{\mathtt{SPECIFIED}}^{\mathtt{DOC}}) \\ \wedge\, \mathtt{n} \doteq \mathtt{Y} \wedge (\mathrm{len}(\mathtt{F}) <\!\doteq \mathtt{Int} \vee \mathtt{Int} <\!\cdot\, 0)\end{array}\right\}$$

$$\mathtt{n} := \mathtt{item}(\mathtt{List}, \mathtt{Int})$$

$$\left\{\begin{array}{l}(\ll\mathtt{NAME_{ID}} \mapsto [\mathtt{F}{:}\mathrm{AF}]_{\mathtt{List}\{\mathtt{Y/n}\}} \gg_{\mathtt{SPECIFIED}}^{\mathtt{DOC}}) \\ \wedge\, \mathtt{n} \doteq \mathbf{null}\end{array}\right\}$$

There are four possible cases in which the parent is a Document node. First, the case in which the parent is a Document node and the index points into the leftmost DNEL structure.

$$\left\{\begin{array}{l}\text{“\#document”}_{\texttt{ID}}\{\!\!\{\ \varnothing_{\text{EA}}\ \}\!\!\}^{\textbf{null}}_{\textbf{null}_{\textbf{9}}}[< \\ \quad \texttt{F}_1\text{:DNEL} \otimes_{\text{DNEL}} \\ \quad <\text{“\#comment”}_{\texttt{ID}'}\{\!\!\{\ \varnothing_{\text{EA}}\ \}\!\!\}^{\texttt{ID}}_{\textbf{null}_{\textbf{8}}}[\varnothing_{\text{CF}}]_{\texttt{FID}'}\texttt{VAL}'>_{\text{DNEL}} \\ \quad \otimes_{\text{DNEL}} \texttt{F}_2\text{:DNEL}, \texttt{F}_3\text{:DE}, \texttt{F}_4\text{:DNEL} \\ >_{\text{DF}}]_{\texttt{List}}\textbf{null} \\ \wedge\, \texttt{n} \doteq \texttt{Y} \wedge \text{len}(\texttt{F}_1) \doteq \texttt{Int}\end{array}\right\}$$

$$\texttt{n} := \texttt{item}(\texttt{List}, \texttt{Int})$$

$$\left\{\begin{array}{l}\text{“\#document”}_{\texttt{ID}}\{\!\!\{\ \varnothing_{\text{EA}}\ \}\!\!\}^{\textbf{null}}_{\textbf{null}_{\textbf{9}}}[< \\ \quad \texttt{F}_1\text{:DNEL} \otimes_{\text{DNEL}} \\ \quad <\text{“\#comment”}_{\texttt{ID}'}\{\!\!\{\ \varnothing_{\text{EA}}\ \}\!\!\}^{\texttt{ID}}_{\textbf{null}_{\textbf{8}}}[\varnothing_{\text{CF}}]_{\texttt{FID}'}\texttt{VAL}'>_{\text{DNEL}} \\ \quad \otimes_{\text{DNEL}} \texttt{F}_2\text{:DNEL}, \texttt{F}_3\text{:DE}, \texttt{F}_4\text{:DNEL} \\ >_{\text{DF}}]_{\texttt{List}\{\texttt{Y/n}\}}\textbf{null} \\ \wedge\, \texttt{n} \doteq \texttt{ID}'\end{array}\right\}$$

Next, the case in which the parent is a Document node and the index points to the document element.

$$\left\{\begin{array}{l}\text{“\#document”}_{\texttt{ID}}\{\!\!\{\ \varnothing_{\text{EA}}\ \}\!\!\}^{\textbf{null}}_{\textbf{null}_{\textbf{9}}}[< \\ \quad \texttt{F}_1\text{:DNEL}, \\ \quad <\texttt{NAME}_{\texttt{ID}'}\{\!\!\{\ \texttt{EA}'\text{:EA}\ \}\!\!\}^{\texttt{ID}}_{\texttt{AID}'_{\textbf{1}}}[\texttt{F}'\text{:EF}]_{\texttt{FID}'}\textbf{null}>_{\text{DE}}, \\ \quad \texttt{F}_2\text{:DNEL} \\ >_{\text{DF}}]_{\texttt{List}}\textbf{null} \\ \wedge\, \texttt{n} \doteq \texttt{Y} \wedge \text{len}(\texttt{F}_1) \doteq \texttt{Int}\end{array}\right\}$$

$$\texttt{n} := \texttt{item}(\texttt{List}, \texttt{Int})$$

$$\left\{\begin{array}{l}\text{“\#document”}_{\texttt{ID}}\{\!\!\{\ \varnothing_{\text{EA}}\ \}\!\!\}^{\textbf{null}}_{\textbf{null}_{\textbf{9}}}[< \\ \quad \texttt{F}_1\text{:DNEL}, \\ \quad <\texttt{NAME}_{\texttt{ID}'}\{\!\!\{\ \texttt{EA}'\text{:EA}\ \}\!\!\}^{\texttt{ID}}_{\texttt{AID}'_{\textbf{1}}}[\texttt{F}'\text{:EF}]_{\texttt{FID}'}\textbf{null}>_{\text{DE}}, \\ \quad \texttt{F}_2\text{:DNEL} \\ >_{\text{DF}}]_{\texttt{List}\{\texttt{Y/n}\}}\textbf{null} \\ \wedge\, \texttt{n} \doteq \texttt{ID}'\end{array}\right\}$$

Next, the case in which the parent is a Document node and the index points into the rightmost DNEL structure.

$$\left\{\begin{array}{l}\text{“\#document”}_{\texttt{ID}}\{\!\!\{\ \varnothing_{\text{EA}}\ \}\!\!\}^{\textbf{null}}_{\textbf{null}_{\textbf{9}}}[< \\ \quad \texttt{F}_1\text{:DNEL}, \texttt{F}_2\text{:DE}, \texttt{F}_3\text{:DNEL} \otimes_{\text{DNEL}} \\ \quad <\text{“\#comment”}_{\texttt{ID}'}\{\!\!\{\ \varnothing_{\text{EA}}\ \}\!\!\}^{\texttt{ID}}_{\textbf{null}_{\textbf{8}}}[\varnothing_{\text{CF}}]_{\texttt{FID}'}\texttt{VAL}'>_{\text{DNEL}} \\ \quad \otimes_{\text{DNEL}} \texttt{F}_4\text{:DNEL} \\ >_{\text{DF}}]_{\texttt{List}}\textbf{null} \\ \wedge\, \texttt{n} \doteq \texttt{Y} \wedge (\text{len}(\texttt{F}_1) + 1 + \text{len}(\texttt{F}_3)) \doteq \texttt{Int}\end{array}\right\}$$

$$\texttt{n} := \texttt{item}(\texttt{List}, \texttt{Int})$$

$$\left\{\begin{array}{l}\text{“\#document”}_{\texttt{ID}}\{\!\!\{\ \varnothing_{\text{EA}}\ \}\!\!\}^{\textbf{null}}_{\textbf{null}_{\textbf{9}}}[< \\ \quad \texttt{F}_1\text{:DNEL}, \texttt{F}_2\text{:DE}, \texttt{F}_3\text{:DNEL} \otimes_{\text{DNEL}} \\ \quad <\text{“\#comment”}_{\texttt{ID}'}\{\!\!\{\ \varnothing_{\text{EA}}\ \}\!\!\}^{\texttt{ID}}_{\textbf{null}_{\textbf{8}}}[\varnothing_{\text{CF}}]_{\texttt{FID}'}\texttt{VAL}'>_{\text{DNEL}} \\ \quad \otimes_{\text{DNEL}} \texttt{F}_4\text{:DNEL} \\ >_{\text{DF}}]_{\texttt{List}\{\texttt{Y/n}\}}\textbf{null} \\ \wedge\, \texttt{n} \doteq \texttt{ID}'\end{array}\right\}$$

Next, the last Document case, in which the parent is a Document node and the index is out of bounds.

$$\left\{\begin{array}{l}\text{“\#document”}_{\texttt{ID}}\{\!\!\{\ \varnothing_{\text{EA}}\ \}\!\!\}^{\textbf{null}}_{\textbf{null}_{\textbf{9}}}[< \\ \quad \texttt{F}_1\text{:DNEL}, \texttt{F}_2\text{:DE}, \texttt{F}_3\text{:DNEL} \\ >_{\text{DF}}]_{\texttt{List}}\textbf{null} \\ \wedge\, \texttt{n} \doteq \texttt{Y} \wedge ((\text{len}(\texttt{F}_1) + \text{len}(\texttt{F}_2) + \text{len}(\texttt{F}_3)) <\!\doteq \texttt{Int} \vee \texttt{Int} <\cdot\, 0)\end{array}\right\}$$

$$\texttt{n} := \texttt{item}(\texttt{List}, \texttt{Int})$$

$$\left\{\begin{array}{l}\text{“\#document”}_{\texttt{ID}}\{\!\!\{\ \varnothing_{\text{EA}}\ \}\!\!\}^{\textbf{null}}_{\textbf{null}_{\textbf{9}}}[< \\ \quad \texttt{F}_1\text{:DNEL}, \texttt{F}_2\text{:DE}, \texttt{F}_3\text{:DNEL} \\ >_{\text{DF}}]_{\texttt{List}\{\texttt{Y/n}\}}\textbf{null} \\ \wedge\, \texttt{n} \doteq \textbf{null}\end{array}\right\}$$

Finally we present the case in which the parent is an element search structure. Recall from Section 6.3.2 that in the case of an element search structure, the `item` command builds a forest on the fly using the dosearch function, and then gets the appropriate item from that forest. Recall from Definition 66 that the flatten$(\texttt{S}, \texttt{F})$ predicate is satisfied by precisely those structures d which produce a forest equal to $\texttt{F}$ when searched with dosearch$(\texttt{S}, d)$.

$$\left\{\begin{array}{l}\mathtt{S}^{\mathtt{ID}}_{\mathtt{List}} \oplus \\ (\mathtt{CG}{:}\mathrm{D}_1{\to}\mathrm{G} \circ_{\mathrm{D}_1} \mathtt{NAME}_{\mathtt{ID}} \lessdot \mathtt{EA}{:}\mathrm{EA} \gtrdot_{\mathtt{AIDN}_{\mathtt{TP}}}^{\mathtt{DOCN}} [\mathtt{F}{:}\mathrm{D}_2 \wedge \\ \quad \mathrm{flatten}(\mathtt{S}, \mathtt{F}_1{:}\mathrm{EF} \otimes_{\mathrm{EF}} < \\ \quad\quad \mathtt{S}'_{\mathtt{ID}'} \lessdot \mathtt{EA}'{:}\mathrm{EA} \gtrdot_{\mathtt{AID}'_{\mathbf{1}}}^{\mathtt{DOC}'} [\mathtt{F}'{:}\mathrm{EF}]_{\mathtt{FID}'}\mathbf{null} \\ \quad > _{\mathrm{EF}} \otimes_{\mathrm{EF}} \mathtt{F}_2{:}\mathrm{EF}) \\ ]_{\mathtt{FID}}\mathtt{VAL}) \\ \wedge \mathrm{len}(\mathtt{F}_1) \doteq \mathtt{Int} \wedge \mathtt{n} \doteq \mathtt{Y}\end{array}\right\}$$

$$\mathtt{n} := \mathtt{item}(\mathtt{List}, \mathtt{Int})$$

$$\left\{\begin{array}{l}\mathtt{S}^{\mathtt{ID}}_{\mathtt{List}\{\mathtt{Y}/\mathtt{n}\}} \oplus \\ (\mathtt{CG}{:}\mathrm{D}_1{\to}\mathrm{G} \circ_{\mathrm{D}_1} \mathtt{NAME}_{\mathtt{ID}} \lessdot \mathtt{EA}{:}\mathrm{EA} \gtrdot_{\mathtt{AIDN}_{\mathtt{TP}}}^{\mathtt{DOCN}} [\mathtt{F}{:}\mathrm{D}_2]_{\mathtt{FID}}\mathtt{VAL}) \\ \wedge \mathtt{n} \doteq \mathtt{ID}'\end{array}\right\}$$

$$\left\{\begin{array}{l}\mathtt{S}^{\mathtt{ID}}_{\mathtt{List}} \oplus \\ (\mathtt{CG}{:}\mathrm{D}_1{\to}\mathrm{G} \circ_{\mathrm{D}_1} \mathtt{NAME}_{\mathtt{ID}} \lessdot \mathtt{EA}{:}\mathrm{EA} \gtrdot_{\mathtt{AIDN}_{\mathtt{TP}}}^{\mathtt{DOCN}} [\mathtt{F}{:}\mathrm{D}_2 \wedge \\ \quad \mathrm{flatten}(\mathtt{S}, \mathtt{F}'{:}\mathrm{EF}) \\ ]_{\mathtt{FID}}\mathtt{VAL}) \\ \wedge (\mathrm{len}(\mathtt{F}'{:}\mathrm{EF}) <\doteq \mathtt{Int} \vee \mathtt{Int} <\cdot 0) \wedge \mathtt{n} \doteq \mathtt{Y}\end{array}\right\}$$

$$\mathtt{n} := \mathtt{item}(\mathtt{List}, \mathtt{Int})$$

$$\left\{\begin{array}{l}\mathtt{S}^{\mathtt{ID}}_{\mathtt{List}\{\mathtt{Y}/\mathtt{n}\}} \oplus \\ (\mathtt{CG}{:}\mathrm{D}_1{\to}\mathrm{G} \circ_{\mathrm{D}_1} \mathtt{NAME}_{\mathtt{ID}} \lessdot \mathtt{EA}{:}\mathrm{EA} \gtrdot_{\mathtt{AIDN}_{\mathtt{TP}}}^{\mathtt{DOCN}} [\mathtt{F}{:}\mathrm{D}_2]_{\mathtt{FID}}\mathtt{VAL}) \\ \wedge \mathtt{n} \doteq \mathbf{null}\end{array}\right\}$$

7.6.4. Element

The Element interface contains only one essential command: `getElementsByTagName`.

$$\{\mathtt{CG}{:}\mathrm{ELE}{\to}\mathrm{G} \circ_{\mathrm{ELE}} \mathtt{NAME}_{\mathtt{Ele}} \lessdot \mathtt{EA}{:}\mathrm{EA} \gtrdot_{\mathtt{AID}_1}^{\mathtt{DOC}} [\mathtt{F}{:}\mathrm{EF}]_{\mathtt{FID}}\mathbf{null} \wedge \mathtt{x} \doteq \mathtt{Y}\}$$

$$\mathtt{x} := \mathtt{getElementsByTagName}(\mathtt{Ele}, \mathtt{TagName})$$

$$\left\{\begin{array}{c}(\mathtt{CG}{:}\mathrm{ELE}{\to}\mathrm{G} \circ_{\mathrm{ELE}} \mathtt{NAME}_{\mathtt{Ele}\{\mathtt{Y}/\mathtt{x}\}} \lessdot \mathtt{EA}{:}\mathrm{EA} \gtrdot_{\mathtt{AID}_1}^{\mathtt{DOC}} [\mathtt{F}{:}\mathrm{EF}]_{\mathtt{FID}}\mathbf{null}) \oplus \\ <_{\mathtt{TagName}\{\mathtt{Y}/\mathtt{x}\}_{\mathtt{x}}}^{\mathtt{DOC}} >_{\mathrm{G}}\end{array}\right\}$$

7.6.5. Attr

The Attr interface contains only one essential command: the getter command for the "specified" object-attribute.

$$\{\ll \texttt{NAME}_{\texttt{N}} \mapsto [\texttt{F}{:}\mathrm{AF}]_{\texttt{FID}} \gg^{\texttt{DOC}}_{\texttt{SPECIFIED}} \wedge\ \texttt{sp} \doteq \texttt{Y}\}$$
$$\texttt{sp} := \texttt{getSpecified}(\texttt{N})$$
$$\{\ll \texttt{NAME}_{\texttt{N}\{\texttt{Y/sp}\}} \mapsto [\texttt{F}{:}\mathrm{AF}]_{\texttt{FID}} \gg^{\texttt{DOC}}_{\texttt{SPECIFIED}} \wedge\ \texttt{sp} \doteq \texttt{SPECIFIED}\}$$

7.6.6. NamedNodeMap

The NamedNodeMap interface contains three essential commands. We present each in turn.

item

The item command behaves exactly the same as in the Nodelist interface.

$$\left\{\begin{array}{l}\texttt{NAME}_{\texttt{ID}} \not\in \\ \quad \texttt{EA}_1{:}\mathrm{EA} \otimes_{\mathrm{EA}} < \\ \qquad \ll \texttt{NAME}'_{\texttt{ID}'} \mapsto [\texttt{F}'{:}\mathrm{AF}]_{\texttt{FID}'} \gg^{\texttt{DOC}}_{\texttt{SPECIFIED}'} > \otimes_{\mathrm{EA}} \texttt{EA}_2{:}\mathrm{EA} \\ \not\ni_{\texttt{Map}_\texttt{1}}^{\texttt{DOC}} [\texttt{F}{:}\mathrm{EF}]_{\texttt{FID}} \textbf{null} \\ \wedge\ \mathrm{len}(\texttt{EA}_1{:}\mathrm{EA}) \doteq \texttt{Int} \wedge \texttt{n} \doteq \texttt{Y}\end{array}\right\}$$
$$\texttt{n} := \texttt{item}(\texttt{Map}, \texttt{Int})$$
$$\left\{\begin{array}{l}\texttt{NAME}_{\texttt{ID}} \not\in \\ \quad \texttt{EA}_1{:}\mathrm{EA} \otimes_{\mathrm{EA}} < \\ \qquad \ll \texttt{NAME}'_{\texttt{ID}'} \mapsto [\texttt{F}'{:}\mathrm{AF}]_{\texttt{FID}'} \gg^{\texttt{DOC}}_{\texttt{SPECIFIED}'} > \otimes_{\mathrm{EA}} \texttt{EA}_2{:}\mathrm{EA} \\ \not\ni_{\texttt{Map}\{\texttt{Y/n}\}_\texttt{1}}^{\texttt{DOC}} [\texttt{F}{:}\mathrm{EF}]_{\texttt{FID}} \textbf{null} \\ \wedge\ \texttt{n} \doteq \texttt{ID}'\end{array}\right\}$$

$$\left\{\begin{array}{l}\texttt{NAME}_{\texttt{ID}} \not\in \texttt{EA}{:}\mathrm{EA} \not\ni_{\texttt{Map}_\texttt{1}}^{\texttt{DOC}} [\texttt{F}{:}\mathrm{EF}]_{\texttt{FID}} \textbf{null} \\ \wedge\ (\mathrm{len}(\texttt{EA}{:}\mathrm{EA}) <\doteq \texttt{Int} \vee \texttt{Int} <\cdot\, 0) \wedge \texttt{n} \doteq \texttt{Y}\end{array}\right\}$$
$$\texttt{n} := \texttt{item}(\texttt{Map}, \texttt{Int})$$
$$\left\{\begin{array}{l}\texttt{NAME}_{\texttt{ID}} \not\in \texttt{EA}{:}\mathrm{EA} \not\ni_{\texttt{Map}\{\texttt{Y/n}\}_\texttt{1}}^{\texttt{DOC}} [\texttt{F}{:}\mathrm{EF}]_{\texttt{FID}} \textbf{null} \\ \wedge\ \texttt{n} \doteq \textbf{null}\end{array}\right\}$$

setNamedItem

The setNamedItem command, as described in Section 6.3.2, moves a particular node into a NamedNodeMap, possibly replacing a code of the same name. There are two possible cases. First, the case where there is no item of that name in the map.

$$\left\{\begin{array}{l}(\varnothing_{\mathrm{D}_1} \multimap (\mathtt{CG}{:}\mathrm{ELE}{\to}\mathrm{G} \circ_{\mathrm{ELE}} \mathtt{NAME}_{\mathtt{ID}} \{\!\!\{ \\ \quad \mathtt{EA}{:}\mathrm{EA} \wedge \\ \quad \neg\Diamond_{\otimes}(\exists \mathtt{ID}'', \mathtt{F}'', \mathtt{FID}'', \mathtt{SPECIFIED}''. \ll \mathtt{NAME}'_{\mathtt{ID}''} \mapsto [\mathtt{F}'{:}\mathrm{AF}]_{\mathtt{FID}''} \gg^{\mathtt{DOC}}_{\mathtt{SPECIFIED}''}, \mathrm{EA}) \\ \}\!\!\}_{\mathtt{Map}_{\mathbf{1}}}^{\mathtt{DOC}}[\mathtt{F}{:}\mathrm{EF}]_{\mathtt{FID}}\mathbf{null})) \\ \circ_{\mathrm{D}_1} < \ll \mathtt{NAME}'_{\mathtt{Arg}} \mapsto [\mathtt{F}'{:}\mathrm{AF}]_{\mathtt{FID}'} \gg^{\mathtt{DOC}}_{\mathtt{SPECIFIED}'} >_{\mathrm{D}_1} \\ \wedge\, \mathtt{n} \doteq \mathtt{Y}\end{array}\right\}$$

$$\mathtt{n} := \mathtt{setNamedItem}(\mathtt{Map}, \mathtt{Arg})$$

$$\left\{\begin{array}{l}\mathtt{CG}{:}\mathrm{ELE}{\to}\mathrm{G} \circ_{\mathrm{ELE}} \mathtt{NAME}_{\mathtt{ID}} \{\!\!\{ \\ \quad \mathtt{EA}{:}\mathrm{EA} \otimes_{\mathrm{EA}} < \ll \mathtt{NAME}'_{\mathtt{Arg}\{\mathtt{Y}/\mathtt{n}\}} \mapsto [\mathtt{F}'{:}\mathrm{AF}]_{\mathtt{FID}'} \gg^{\mathtt{DOC}}_{\mathtt{SPECIFIED}'} >_{\mathrm{EA}} \\ \}\!\!\}_{\mathtt{Map}\{\mathtt{Y}/\mathtt{n}\}_{\mathbf{1}}}^{\mathtt{DOC}}[\mathtt{F}{:}\mathrm{EF}]_{\mathtt{FID}}\mathbf{null} \\ \wedge\, \mathtt{n} \doteq \mathbf{null}\end{array}\right\}$$

Next, the case in which there is an existing node of that name in the map.

$$\left\{\begin{array}{l}(\varnothing_{\mathrm{D}_1} \multimap (\mathtt{CG}{:}\mathrm{ELE}{\to}\mathrm{G} \circ_{\mathrm{ELE}} \mathtt{NAME}_{\mathtt{ID}} \{\!\!\{ \\ \quad \mathtt{EA}_1{:}\mathrm{EA} \otimes_{\mathrm{EA}} < \ll \mathtt{NAME}'_{\mathtt{ID}''} \mapsto [\mathtt{F}''{:}\mathrm{AF}]_{\mathtt{FID}''} \gg^{\mathtt{DOC}}_{\mathtt{SPECIFIED}''} >_{\mathrm{EA}} \otimes_{\mathrm{EA}} \mathtt{EA}_2{:}\mathrm{EA} \\ \}\!\!\}_{\mathtt{Map}_{\mathbf{1}}}^{\mathtt{DOC}}[\mathtt{F}{:}\mathrm{EF}]_{\mathtt{FID}}\mathbf{null})) \\ \circ_{\mathrm{D}_1} < \ll \mathtt{NAME}'_{\mathtt{Arg}} \mapsto [\mathtt{F}'{:}\mathrm{AF}]_{\mathtt{FID}'} \gg^{\mathtt{DOC}}_{\mathtt{SPECIFIED}'} >_{\mathrm{D}_1} \\ \wedge\, \mathtt{n} \doteq \mathtt{Y}\end{array}\right\}$$

$$\mathtt{n} := \mathtt{setNamedItem}(\mathtt{Map}, \mathtt{Arg})$$

$$\left\{\begin{array}{l}\mathtt{CG}{:}\mathrm{ELE}{\to}\mathrm{G} \circ_{\mathrm{ELE}} \mathtt{NAME}_{\mathtt{ID}} \{\!\!\{ \\ \quad \mathtt{EA}_1{:}\mathrm{EA} \otimes_{\mathrm{EA}} \mathtt{EA}_2{:}\mathrm{EA} \otimes_{\mathrm{EA}} < \ll \mathtt{NAME}'_{\mathtt{Arg}\{\mathtt{Y}/\mathtt{n}\}} \mapsto [\mathtt{F}'{:}\mathrm{AF}]_{\mathtt{FID}'} \gg^{\mathtt{DOC}}_{\mathtt{SPECIFIED}'} >_{\mathrm{EA}} \\ \}\!\!\}_{\mathtt{Map}\{\mathtt{Y}/\mathtt{n}\}_{\mathbf{1}}}^{\mathtt{DOC}}[\mathtt{F}{:}\mathrm{EF}]_{\mathtt{FID}}\mathbf{null} \\ \oplus < \ll \mathtt{NAME}'_{\mathtt{ID}''} \mapsto [\mathtt{F}''{:}\mathrm{AF}]_{\mathtt{FID}''} \gg^{\mathtt{DOC}}_{\mathtt{SPECIFIED}''} >_{\mathrm{G}} \wedge\, \mathtt{n} \doteq \mathbf{null}\end{array}\right\}$$

removeNamedItem

The `removeNamedItem` command is move command that removes an Attr from a NamedNodeMap. The Attr node to remove it determined by its

NodeName. In particular, recall from Section 6.3.2 that if the Attr to be removed is mentioned by the DTD Fragment, then the act of removing it from the NamedNodeMap will also create a new Attr node with a default value given by the DTD Fragment.

$$\left\{\begin{array}{l}\texttt{CG}{:}\mathrm{ELE}{\rightarrow}\mathrm{G} \circ_{\mathrm{ELE}} \texttt{NAME}_{\texttt{ID}} \{\!\!\{ \\ \quad \texttt{EA}_1{:}\mathrm{EA} \otimes_{\mathrm{EA}} \\ \quad <\!\!\ll \texttt{Name}_{\texttt{ID}'} \mapsto [\texttt{F}'{:}\mathrm{AF}]_{\texttt{FID}'} \gg^{\texttt{DOC}}_{\texttt{SPECIFIED}} >_{\mathrm{EA}} \\ \quad \otimes_{\mathrm{EA}} \texttt{EA}_2{:}\mathrm{EA} \\ \}\!\!\}_{\texttt{Map}_\mathbf{1}}^{\texttt{DOC}}[\texttt{F}{:}\mathrm{EF}]_{\texttt{FID}}\mathbf{null} \\ \wedge\, \texttt{n} \doteq \texttt{Y}\end{array}\right\}$$

$$\texttt{n} := \texttt{removeNamedItem}(\texttt{Map}, \texttt{Name})$$

$$\left\{\begin{array}{l}\texttt{CG}{:}\mathrm{ELE}{\rightarrow}\mathrm{G} \circ_{\mathrm{ELE}} \texttt{NAME}_{\texttt{ID}} \{\!\!\{ \\ \quad \texttt{EA}_1{:}\mathrm{EA} \otimes_{\mathrm{EA}} \\ \quad \left(\begin{array}{l}((\texttt{NAME}, \texttt{Name}\{\texttt{Y/n}\}) \notin \mathrm{dom}(d) \wedge \varnothing_{\mathrm{EA}}) \vee \\ (<\text{“\#text”}_{\texttt{ID}''} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{null_3}}^{\texttt{DOC}}[\varnothing_{\mathrm{TF}}]_{\texttt{FID}''} d(\texttt{NAME}, \texttt{Name}\{\texttt{Y/n}\}))\end{array}\right) \\ \quad \otimes_{\mathrm{EA}} \texttt{EA}_2{:}\mathrm{EA} \\ \}\!\!\}_{\texttt{Map}\{\texttt{Y/n}\}_\mathbf{1}}^{\texttt{DOC}}[\texttt{F}{:}\mathrm{EF}]_{\texttt{FID}}\mathbf{null} \\ \oplus <\!\!\ll \texttt{Name}\{\texttt{Y/n}\}_{\texttt{ID}'} \mapsto [\texttt{F}'{:}\mathrm{AF}]_{\texttt{FID}'} \gg^{\texttt{DOC}}_{\texttt{SPECIFIED}} >_{\mathrm{G}} \\ \wedge\, \texttt{n} \doteq \texttt{ID}'\end{array}\right\}$$

7.6.7. Character Data

The CharacterData interface contains three essential commands. We present them each in turn.

substringData

This command extracts a range of data from a Text or Comment node.

$$\left\{\begin{array}{l}\texttt{NAME}_{\texttt{Node}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{null}_{\texttt{TP}}}^{\texttt{DOC}}[\varnothing_{\mathrm{D}}](\mathrm{S}_1 \otimes_{\mathrm{S}} \mathrm{S}_2 \otimes_{\mathrm{S}} \mathrm{S}_3) \\ \wedge\, \mathrm{len}(\mathrm{S}_1) \doteq \texttt{Offset} \wedge \mathrm{len}(\mathrm{S}_2) \doteq \texttt{Count} \wedge \texttt{str} \doteq \texttt{Y}\end{array}\right\}$$

$$\texttt{str} := \texttt{substringData}(\texttt{Node}, \texttt{Offset}, \texttt{Count})$$

$$\left\{\begin{array}{l}\texttt{NAME}_{\texttt{Node}\{\texttt{Y/str}\}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{null}_{\texttt{TP}}}^{\texttt{DOC}}[\varnothing_{\mathrm{D}}](\mathrm{S}_1 \otimes_{\mathrm{S}} \mathrm{S}_2 \otimes_{\mathrm{S}} \mathrm{S}_3) \\ \wedge\, \texttt{str} \doteq \mathrm{S}_2\end{array}\right\}$$

$$\left\{ \begin{array}{l} \mathtt{NAME_{Nd}} \lessdot \varnothing_{\mathrm{EA}} \gtrdot^{\mathtt{DOC}}_{\mathbf{null}_{\mathtt{TP}}} [\varnothing_{\mathrm{D}}](\mathtt{S}_1 \otimes_{\mathrm{S}} \mathtt{S}_2) \\ \quad \wedge \mathrm{len}(\mathtt{S}_1) \doteq \mathtt{Offset} \wedge \mathrm{len}(\mathtt{S}_2) <\cdot \mathtt{Count} \wedge \mathtt{str} \doteq \mathtt{Y} \end{array} \right\}$$
$$\mathtt{str} := \mathtt{substringData}(\mathtt{Nd}, \mathtt{Offset}, \mathtt{Count})$$
$$\left\{ \begin{array}{l} \mathtt{NAME_{Nd\{Y/str\}}} \lessdot \varnothing_{\mathrm{EA}} \gtrdot^{\mathtt{DOC}}_{\mathbf{null}_{\mathtt{TP}}} [\varnothing_{\mathrm{D}}](\mathtt{S}_1 \otimes_{\mathrm{S}} \mathtt{S}_2) \\ \quad \wedge \mathtt{str} \doteq \mathtt{S}_2 \end{array} \right\}$$

appendData

The `appendData` command appends new data to an existing Text or Comment node. Recall that we can check the type of an expression in a formula by using $\mathtt{Expr} \in \mathrm{D}$ for any type D.

$$\left\{ \mathtt{NAME_{Nd}} \lessdot \varnothing_{\mathrm{EA}} \gtrdot^{\mathtt{DOC}}_{\mathbf{null}_{\mathtt{TP}}} [\varnothing_{\mathrm{D}}]\mathtt{S} \wedge \mathtt{Arg} \in \mathrm{S} \right\}$$
$$\mathtt{appendData}(\mathtt{Nd}, \mathtt{Arg})$$
$$\left\{ \mathtt{NAME_{Nd}} \lessdot \varnothing_{\mathrm{EA}} \gtrdot^{\mathtt{DOC}}_{\mathbf{null}_{\mathtt{TP}}} [\varnothing_{\mathrm{D}}](\mathtt{S} \otimes_{\mathrm{S}} \mathtt{Arg}) \right\}$$

deleteData

The `deleteData` command removes a substring from the value of a Text or Comment node.

$$\left\{ \begin{array}{l} \mathtt{NAME_{Nd}} \lessdot \varnothing_{\mathrm{EA}} \gtrdot^{\mathtt{DOC}}_{\mathbf{null}_{\mathtt{TP}}} [\varnothing_{\mathrm{D}}](\mathtt{S}_1 \otimes_{\mathrm{S}} \mathtt{S}_2 \otimes_{\mathrm{S}} \mathtt{S}_3) \\ \quad \wedge \mathrm{len}(\mathtt{S}_1) \doteq \mathtt{Offset} \wedge \mathrm{len}(\mathtt{S}_2) \doteq \mathtt{Count} \end{array} \right\}$$
$$\mathtt{deleteData}(\mathtt{Nd}, \mathtt{Offset}, \mathtt{Count})$$
$$\left\{ \mathtt{NAME_{Nd}} \lessdot \varnothing_{\mathrm{EA}} \gtrdot^{\mathtt{DOC}}_{\mathbf{null}_{\mathtt{TP}}} [\varnothing_{\mathrm{D}}](\mathtt{S}_1 \otimes_{\mathrm{S}} \mathtt{S}_3) \right\}$$

$$\left\{ \begin{array}{l} \mathtt{NAME_{Nd}} \lessdot \varnothing_{\mathrm{EA}} \gtrdot^{\mathtt{DOC}}_{\mathbf{null}_{\mathtt{TP}}} [\varnothing_{\mathrm{D}}](\mathtt{S}_1 \otimes_{\mathrm{S}} \mathtt{S}_2) \\ \quad \wedge \mathrm{len}(\mathtt{S}_1) \doteq \mathtt{Offset} \wedge \mathrm{len}(\mathtt{S}_2) <\cdot \mathtt{Count} \end{array} \right\}$$
$$\mathtt{deleteData}(\mathtt{Nd}, \mathtt{Offset}, \mathtt{Count})$$
$$\left\{ \mathtt{NAME_{Nd}} \lessdot \varnothing_{\mathrm{EA}} \gtrdot^{\mathtt{DOC}}_{\mathbf{null}_{\mathtt{TP}}} [\varnothing_{\mathrm{D}}]\mathtt{S}_1 \right\}$$

7.7. Inference Rules

The inference rules are the same as those presented for Featherweight DOM in Definition 25.

```
<students>
  <current>
    <student startDate="Jan 2006">
      <name>Gareth Smith</name>
      <address>Over the rainbow</address>
      <subject>Computer Science</subject>
      <finalYear />
    </student>
  </current>
  <alumni>
    <student startDate="Sept 2002" finishDate="March 2007">
      <name>Uri Zarfaty</name>
      <address>Where the wild things are</address>
      <subject>Computer Science</subject>
    </student>
  </alumni>
</students>
```

Figure 7.8.: Contacts Document

7.8. Example

Consider a document which represents the contact information for a university development office. A snippit of such a document is presented in Figure 7.8

We can describe the schema of documents such as this with the XML Schema in Figure 7.9 or with the formula S given in Figure 7.10.

At the end of the academic year we may wish to graduate the final year students, and move them into the "alumni" section of the data file, where they can be fruitfully harvested. Assuming that some previous admin process has tagged the current final year students with the "finalYear" element, this operation of graduating the final year students can be performed by the procedure `graduateStudents(doc, currentDate)`[1] given in Figure 7.11.

We can prove that this procedure maintains the schema of the document it operates over. As with our Featherweight DOM examples we require an additional safety precondition to say the the "`currentDate`" parameter is

[1] In a real system it is of course quite likely that there would be a system call for determining the current date. Since such system calls are beyond the scope of this work, we satisfy ourselves with the "currentDate" parameter.

```
<?xml version='1.0' encoding='UTF-8'?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           elementFormDefault="qualified">
  <xs:element name="students">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="current" minOccurs="1" maxOccurs="1">
          <xs:element name="student" minOccurs="0" maxOccurs="unbounded">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="name" minOccurs="1"
                            maxOccurs="1" type="xs:string" />
                <xs:element name="address" minOccurs="1"
                            maxOccurs="1" type="xs:string" />
                <xs:element name="subject" minOccurs="1"
                            maxOccurs="1" type="xs:string" />
                <xs:element name="finalYear" minOccurs="0"
                            maxOccurs="1" />
              </xs:sequence>
              <xs:attribute name="startYear" type="xs:string"/>
            </xs:complexType>
          </xs:element>
        </xs:element>
        <xs:element name="alumni" minOccurs="1" maxOccurs="1">
          <xs:element name="student" minOccurs="0" maxOccurs="unbounded">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="name" minOccurs="1"
                            maxOccurs="1" type="string" />
                <xs:element name="address" minOccurs="1"
                            maxOccurs="1" type="string" />
                <xs:element name="subject" minOccurs="1"
                            maxOccurs="1" type="string" />
              </xs:sequence>
              <xs:attribute name="startYear" type="xs:string"/>
              <xs:attribute name="startYear" type="xs:string"/>
            </xs:complexType>
          </xs:element>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Figure 7.9.: Contacts Schema

$$
\begin{aligned}
S \triangleq\ & \\
& <\text{“\#document”} \lessdot \varnothing_{\mathrm{EA}} \gtrdot_{\mathbf{9}} [<\varnothing_{\mathrm{DNEL}}, < \\
& \quad \text{“students”} \lessdot \varnothing_{\mathrm{EA}} \gtrdot_{\mathbf{1}} [\\
& \quad\quad \text{“current”} \lessdot \varnothing_{\mathrm{EA}} \gtrdot_{\mathbf{1}} [\\
& \quad\quad\quad \Box_{\otimes}(\text{“student”} \lessdot < \\
& \quad\quad\quad\quad \ll \text{“startDate”} \mapsto [\mathrm{true}_{\mathrm{AF}}] \gg \\
& \quad\quad\quad\quad >_{\mathrm{EA}} \gtrdot_{\mathbf{1}} [\\
& \quad\quad\quad\quad\quad < \text{“name”} \lessdot \varnothing_{\mathrm{EA}} \gtrdot_{\mathbf{1}} [TXTS]\mathbf{null} >_{\mathrm{EF}} \otimes_{\mathrm{EF}} \\
& \quad\quad\quad\quad\quad < \text{“address”} \lessdot \varnothing_{\mathrm{EA}} \gtrdot_{\mathbf{1}} [TXTS]\mathbf{null} >_{\mathrm{EF}} \otimes_{\mathrm{EF}} \\
& \quad\quad\quad\quad\quad < \text{“subject”} \lessdot \varnothing_{\mathrm{EA}} \gtrdot_{\mathbf{1}} [TXTS]\mathbf{null} >_{\mathrm{EF}} \otimes_{\mathrm{EF}} \\
& \quad\quad\quad\quad\quad (\varnothing_{\mathrm{EF}} \vee < \text{“finalYear”} \lessdot \varnothing_{\mathrm{EA}} \gtrdot_{\mathbf{1}} [\varnothing_{\mathrm{EF}}]\mathbf{null} >_{\mathrm{EF}}) \\
& \quad\quad\quad]\mathbf{null}, \mathrm{EF}) \\
& \quad\quad]\mathbf{null} \\
& \quad\quad \text{“alumni”} \lessdot \varnothing_{\mathrm{EA}} \gtrdot_{\mathbf{1}} [\\
& \quad\quad\quad \Box_{\otimes}(\text{“student”} \lessdot < \\
& \quad\quad\quad\quad \ll \text{“startDate”} \mapsto [\mathrm{true}_{\mathrm{AF}}] \gg \otimes_{\mathrm{AF}} \\
& \quad\quad\quad\quad \ll \text{“finishDate”} \mapsto [\mathrm{true}_{\mathrm{AF}}] \gg \\
& \quad\quad\quad\quad >_{\mathrm{EA}} \gtrdot_{\mathbf{1}} [\\
& \quad\quad\quad\quad\quad < \text{“name”} \lessdot \varnothing_{\mathrm{EA}} \gtrdot_{\mathbf{1}} [TXTS]\mathbf{null} >_{\mathrm{EF}} \otimes_{\mathrm{EF}} \\
& \quad\quad\quad\quad\quad < \text{“address”} \lessdot \varnothing_{\mathrm{EA}} \gtrdot_{\mathbf{1}} [TXTS]\mathbf{null} >_{\mathrm{EF}} \otimes_{\mathrm{EF}} \\
& \quad\quad\quad\quad\quad < \text{“subject”} \lessdot \varnothing_{\mathrm{EA}} \gtrdot_{\mathbf{1}} [TXTS]\mathbf{null} >_{\mathrm{EF}} \otimes_{\mathrm{EF}} \\
& \quad\quad\quad]\mathbf{null}, \mathrm{EF}) \\
& \quad\quad]\mathbf{null} \\
& \quad]\mathbf{null} \\
& >_{\mathrm{DE}}, \varnothing_{\mathrm{DNEL}} >_{\mathrm{DF}}]\mathbf{null} >_{\mathrm{G}}
\end{aligned}
$$

$$
TXTS \triangleq \Box_{\otimes}(\text{“\#text”} \lessdot \varnothing_{\mathrm{EA}} \gtrdot_{\mathbf{3}} [\varnothing_{\mathrm{TF}}]\mathrm{true}_{\mathrm{S}}, \mathrm{EF})
$$

$$
P \triangleq \texttt{currentDate} \in \mathrm{S} \wedge \Diamond_{\mathrm{DOC}\rightarrow\mathrm{G}} \text{“\#document”}_{\texttt{doc}} \lessdot \varnothing_{\mathrm{EA}} \gtrdot_{\mathbf{9}} [\mathrm{true}_{\mathrm{DF}}]\mathbf{null}
$$

Figure 7.10.: Contacts Predicate and Safety Precondition

```
graduateStudents(doc, currentDate) ≜
    local kids, de, alumni, tags, currentStudentTag,
          currentStudent, finishDateAt, txt, ats :
        kids := getChildNodes(doc) ;
        de := item(kids, 0) ;
        kids := getChildNodes(de) ;
        alumni := item(kids, 1) ;
        tags := getElementsByTagName(doc, "finalYear") ;
        currentStudentTag := item(tags, 0) ;
        while currentStudentTag ≠ null do
            currentStudent := getParentNode(currentStudentTag) ;
            removeChild(currentStudent, currentStudentTag) ;
            finishDateAt := createAttribute(doc, "finishDate") ;
            txt := createTextNode(doc, currentDate) ;
            appendChild(finishDateAt, txt) ;
            ats := getAttributes(currentStudent) ;
            setNamedItem(ats, finishDateAt) ;
            appendChild(alumni, currentStudent) ;
            currentStudentTag := item(tags, 0)
        od
    endloc
```

Figure 7.11.: The graduateStudents Procedure

a String and that the "doc" parameter refers to a document in the grove. This safety precondition S is also given in Figure 7.10. Finally, we may also leave garbage in the grove which the garbage collector will dispose of for us. The local Hoare triple for the procedure that we wish to prove is therefore:

$$\{P \wedge S\}\texttt{graduateStudents}(\texttt{doc}, \texttt{currentDate})\{S \oplus \text{true}_{\text{G}}\}$$

The proof of this triple is a little lengthy but surprisingly readable, and given in Appendix B.2. Notice that the footprints of many of our commands are quite large, and that this adds significantly to the length of the proof. We discuss this issue further in Chapter 9.

8. DOM in the Wild

DOM implementations can be found in every modern web browser, and in every major modern programming language. During the course of this research we have tested the behaviour of a number of DOM implementations. In particular, our experiments into how web browsers handle default attributes and normalize operations are recorded in Appendix C. Recall the potentially confusing specification for the Attr node object attribute "specified" first mentioned in Section 1.1. Every web browser tested seemed to treat this attribute differently. It is tempting to conclude that the W3C specification has failed to clearly communicate its intent to browser developers – or at the very least to convince them of the wisdom of that intent. On the issue of the normalize command, which was definitely not completely specified (as discussed in Sections 1.1 and B.1), all web browsers apart from Internet Explorer are in agreement. Our specification in Section B.1 follows this consensus.

The W3C have provided a series of tests designed to judge the compliance of an implementation with the specification [24]. To our knowledge, no web browser has ever passed all these tests, though many perform well in the specialised HTML sections. It should be noted that it may not be desirable for a web browser to pass all tests, since some of them test failure modes which we may not want to see in a practical browser. If the specification requires that a common coding error result in a fault, is it better to insist that implementations must fault on encountering that error, or should they be free to try to detect the error, and compensate for it?

Historically, the market pressures on web browsers have tended to favour those that tried to detect common errors and compensate for them. Recently however, there have been grass-roots movements which have lobbied for greater standards compliance in web browsers. These movements argue that attempting to compensate for poor code will result in a de-facto standard that encourages poor code, to the detriment of all. Furthermore,

they are wary of large companies making use of large market shares to warp previously open web standards to their advantage, forcing more and more people to use their technology exclusively.

A notable grass-roots movement is the Web Standards Project [67], who have produced the infamous Acid Tests [1], which test browser compliance to DOM, HTML, CSS and ECMAScript standards. According to those tests, the only stable public release of a mainstream browser to be completely compliant is Safari 4.0. The next stable release of Chrome is expected to follow suit. Firefox does not perform as well as the two front runners, and Internet Explorer lags further behind.

This formalisation promises a more precise way of measuring if a particular DOM implementation satisfies the specification. Gardner, Dinsdale-Young and Wheelhouse have begun the work of linking high level reasoning about specifications to low level reasoning about implementations in their paper "Abstraction and Refinement for Local Reasoning"[30].

8.1. Python minidom

Recall that in Chapter 1.1, we asserted that presenting information about the precondition of the Node command "appendChild" in the documentation of that command's errors was potentially confusing. In this Section, we present a real DOM implementation which missed that precondition, and speculate on how it may have been overlooked.

The documentation for Python minidom[50] states "DOMException is currently not supported in xml.dom.minidom. Instead, xml.dom.minidom uses standard Python exceptions such as TypeError and AttributeError". This seems like a perfectly sensible design decision, if the standard Python exceptions duplicate the functionality of DOMException. The definition of "DOMExceptoion" in the DOM Level 1 spec says:

Begin Quote

> DOM operations only raise exceptions in "exceptional" circumstances, i.e., when an operation is impossible to perform (either for logical reasons, because data is lost, or because the implementation has become unstable). In general, DOM methods re-

turn specific error values in ordinary processing situation, such as out-of-bound errors when using NodeList.

Implementations may raise other exceptions under other circumstances. For example, implementations may raise an implementation-dependent exception if a null argument is passed.

Some languages and object systems do not support the concept of exceptions. For such systems, *error conditions may be indicated using native error reporting mechanisms.* For some bindings, for example, methods may return error codes similar to those listed in the corresponding method descriptions.

End Quote

This looks encouraging – Python has a strong dynamic typing system with associated mechanisms for detecting and reporting runtime errors. The decision to use that native mechanism to handle any errors seems like a good one. The DOM specification gives more detail in this section about the specific sorts of DOMException which might be raised. There is one in particular which interests us:

Begin Quote

HIERARCHY_REQUEST_ERR If any node is inserted somewhere it doesn't belong

End Quote

Indeed Python minidom does make use of the type system to catch any attempted insertion of a node into "somewhere it doesn't belong". Attempting to insert an Element as a child of an Attribute Node does raise a type error. Python minidom certainly does seem to handle errors in a way that is compliant with the description of "DOMException" in section 1.2 of the core level 1 DOM spec.

Since the behaviour of DOMException is sufficiently handled globally by existing Python error reporting mechanisms, is there any need for the implementor to care about what DOMExceptions might be thrown by the "insertBefore" method in particular?

Begin Quote

HIERARCHY_REQUEST_ERR: Raised if this node is of a type that does not allow children of the type of the newChild node, *or if the node to insert is one of this node's ancestors.*

End Quote

In fact, prior to 2007, Python minidom did not throw an error if you attempted to insert a node as a child of one of its own children. Instead, the operation silently went ahead, creating a structure with a loop where we expected there to be a tree. Many other methods in Python minidom also expected the data structure they operate over to be a tree, and would diverge on encountering this loop. This independently discovered issue was independently discovered and fixed by Orendorff in 2007 [54].

9. Conclusions and Future Work

In Chapter 3 we presented Featherweight DOM, which provided the conceptual core of this thesis and introduced our strategies for describing DOM. We used an abstract data structure and accompanying context structure with many different types of hole. We described essential commands from DOM using that data and we used a simple imperative programming language to describe the remaining commands. This chapter also introduced our philosophy of working with a dynamic programming language, with dynamically typed variables and expressions, and dynamic scope. Our specification of Featherweight DOM is precise and compositional, in contrast to the W3C specification.

In Chapter 4 we presented context logic for Featherweight DOM which allows us to reason about Featherweight DOM programs. This logic is necessarily different from previous work, since Featherweight DOM contexts have many types of hole. We also presented a novel proof of soundness for our reasoning, making use of a new definition for command locality. Where previous work gives us reasoning about local commands, we provide local reasoning about DOM commands, which are not all local by the old definition. Our new definition has potential future applications in reasoning about JavaScript, concurrent monitors and other programming constructs of limited locality.

In Chapter 6 we presented the fundamental interfaces of DOM Core Level 1. In this chapter we described in rigorous detail all the quirks and corner cases of that portion of the DOM Specification. We showed cases such as the behaviour of the "specified" object-attribute which are confusing, and cases such as the `removeNamedItem` command of the NamedNodeMap interface and the `normalize` command of the Element interface which are inconsistent or incomplete in the W3C specification. Where there was inconsistency or incompleteness, we made decisions based on the behaviour of existing implementations and on robustness of behaviour. As with Chapter 3, our

specification is precise and compositional.

In Chapter 7 we presented context logic for DOM Core Level 1, which allows us to reason about programs written using the fundamental interfaces of DOM Core Level 1. This logic followed the logic of Chapter 4 with only small alterations to allow reasoning about structures such as DTD Fragments and Element Searches.

In Chapters 5 and 7 we presented example programs written using Featherweight DOM and DOM Core Level 1 respectively. These examples go a long way towards demonstrating the practicality of this style of reasoning about web programs, and the example in Chapter 7 in particular demonstrates some of the current limits of this work. The proof of that example program, while conceptually straightforward, is textually unwieldy. This unwieldyness is a symptom of the same problem that inspired local reasoning with separation logic in the first place. The problem is that we are forced to mention many superfluous structures in every step of our reasoning. The promise of local reasoning is that we should be able to use the frame rule to discard these structures when we are not explicitly interested in them, and thus ensure that all our Hoare triples succinctly mention only the footprints of the programs they describe. In the case of DOM is it clear that our footprints are too large. This problem is particularly evident in the specification of `appendChild`: when moving a node from a source to a target, context logic requires that the footprint of the command be the smallest subtree of the whole structure which contains both the source and the target points. This tree may contain a large number of common ancestors of both nodes, which are totally unaffected by the command, and which we would like to safely ignore.

One solution to the problem of unwieldyness in our program proofs is to fix our context logic so as to reduce the size of our footprints. This challenge has been met by Gardner and Wheelhouse who, in response to this work on DOM, have developed a new logic called "Segment Logic" [35]. This logic allows us to build contexts with many labelled holes, which in turn allows for a much smaller description of the `appendChild` command. Future work on reasoning about web programs should certainly take advantage of this development.

Another possible solution to the problem of unwieldyness is to introduce automation to the process of reasoning about programs. All the program

proofs given in this thesis have been enticingly mechanical, which gives us hope that such proofs may be automated entirely. Local reasoning using separation logic has enjoyed a great deal of success in reasoning about low-level C programs. Tools such as smallfoot [62] and space invader [63] have been used in the wild to find memory faults in real programs, and look set to become an important tool for the system programmers of the future. In comparison, local reasoning using context logic is in its infancy, and this work represents its first claim of practical utility. The schema preservation examples given in Chapters 5.4 and 7.8 hint at one possible route towards the automated success of the separation logic community: it may be possible to automatically generate predicates from traditional schema descriptions written in subsets of languages such as XML Schema or Relax NG. This would remove the need for training end-users in understanding context logic notation. It may then be possible to develop a tool which uses symbolic execution techniques mirroring those developed in the separation logic community to attempt to prove that the generated schema formula was maintained by the program. A system such as this could be used alongside programs like the W3C HTML Validator [66]: while the W3C tool will validate static HTML, our tool would test whether the JavaScript in the page was capable of invalidating that HTML at runtime.

In joint work with Gardner and Wright [32], the author has begun the task of adapting this work to reasoning about mash-ups. Mashups may acquire data and code from remote sources at runtime, and incorporate those data and code into their own data and control structures. This example-driven work highlights another possible avenue for the exploitation of local reasoning about DOM. The vast majority of mashup programs are extremely simple, written by large numbers of minimally trained programmers in order to enhance the appearance or UI of websites. However, most of these programs make use of a small number of extremely complex libraries, written by a small number of highly trained programmers. One of the aims of the work presented in [32] was to make easy programs easy to reason about, and hard programs possible to reason about. There is enormous benefit to be reaped in automating the relatively simple process of reasoning about simple programs, and expending some human effort in providing trusted specifications for the libraries which those simple programs use.

In joint work with Sergio Maffeis, Gardner and the author have recently

begun the task of reasoning about the full JavaScript language using a variant of separation logic. Maffeis has recently completed an operational semantics for JavaScript with Mitchell and Taly [49], and it is this operational semantics which is forming the core of this new work. Local reasoning, of the sort practised with separation logic and context logic, seems particularly well suited to JavaScript, since that language is so very dependent on its heap. For example, JavaScript has no variable stack, preferring instead to emulate one as a linked list in the heap. This is how it is possible to include such features as the infamous "with" statement. However, early work suggests that simple operations such as assignment in JavaScript are not "local" in the traditional sense. Fortunately, through the use of the new definition of locality introduced in this thesis in Chapter 4, a separation logic for reasoning about JavaScript programs still seems possible.

The possibility of reasoning about languages as challenging as JavaScript is exciting. In general, scripting languages are challenging to reason about, since they tend to embrace a number of features that are traditionally considered "unsafe" - such as reflection, dynamic weak typing, implicit type coercion and so on. These features are not present by accident, but because they solve particular problems faced by programmers on the web. The author believes that local reasoning is uniquely suited to reasoning about these "difficult" mechanisms.

Bibliography

[1] Acid Tests. http://www.acidtests.org/. 8

[2] Ajax: A new approach to web applications. http://www.adaptivepath.com/ideas/essays/archives/000385.php. 1.1

[3] Christopher Anderson, Sophia Drossopoulou, and Paola Giannini. Towards Type Inference for JavaScript. In *19th European Conference on Object-Oriented Programming (ECOOP 2005), Glasgow, Scotland*, pages 428–452, July 2005. 1.1

[4] Josh Berdine, Cristiano Calcagno, and Peter W. Ohearn. Smallfoot: Modular automatic assertion checking with separation logic. In *International Symposium on Formal Methods for Components and Objects*, pages 115–137. Springer, 2005. 1.1, 2.2.3

[5] G. M. Bierman, M. J. Parkinson, and A. M. Pitts. Mj: An imperative core calculus for java and java with effects. Technical report, Cambridge, 2003. 1.1

[6] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. *Proceedings of the 32nd ACM SIGPLAN symposium on Principles of Programming Languages*, 40(1):259–270, 2005. 2.2.3

[7] Chandrasekhar Boyapati, Alexandru Salcianu, William Beebee, Jr., and Martin Rinard. Ownership types for safe region-based memory management in real-time java. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 324–337, New York, NY, USA, 2003. ACM. 1.1

[8] James Brotherston, Richard Bornat, and Cristiano Calcagno. Cyclic proofs of program termination in separation logic. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on*

Principles of programming languages, pages 101–112, New York, NY, USA, 2008. ACM. 2.2.3

[9] R Burstall. Some techniques for proving correctness of programs which alter data structures. In *Machine Intelligence 7*, pages 23–50. Edinburgh University Press, 1972. 2.1

[10] C. Calcagno, P. Gardner, and U. Zarfaty. Context logic as modal logic: Completeness and parametric inexpressivity. In *POPL '07: Proceedings of the 34th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, 2007. 2.3, 2.3.3

[11] Cristiano Calcagno, Thomas Dinsdale-Young, and Philippa Gardner. Adjunct elimination in context logic for trees. In Zhong Shao, editor, *Programming Languages and Systems*, volume 4807 of *Lecture Notes in Computer Science*, pages 255–270. Springer Berlin / Heidelberg, 2007. 2.3.3

[12] Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 289–300, New York, NY, USA, 2009. ACM. 1.1, 2.2.3

[13] 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. 1.1, 2.3, 4

[14] Cristiano Calcagno, Philippa Gardner, and Uri Zarfaty. Local reasoning about data update. *Computation, Meaning and Logic: Articles dedicated to Gordon Plotkin, Electronic Notes in Theoretical Computer Science*, 2007. 2.3, 2.3.3

[15] Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local action and abstract separation logic. In *LICS '07: Proceedings of the 22nd Annual IEEE Symposium on Logic in Computer Science*, pages 366–378, Washington, DC, USA, 2007. IEEE Computer Society. 2.2.3

[16] Luca Cardelli and Andrew D. Gordon. Anytime, anywhere: modal logics for mobile ambients. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 365–377, New York, NY, USA, 2000. ACM. 1.1, 2.3

[17] Rfc 3875: The common gateway interface (cgi). http://tools.ietf.org/html/rfc3875. 1.1

[18] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *In Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 48–64. ACM Press, 1998. 1.1

[19] Ryan Dewsbury. *Google Web Toolkit Applications*. Prentice Hall, 2008. 1.1

[20] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975. 2.1

[21] Dino Distefano and Matthew J. Parkinson. jStar: Towards practical verification for Java. In Gail E. Harris, editor, *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA*, pages 213–226. ACM, 2008. 2.2.3

[22] Dino Distefano and Matthew J. Parkinson J. jstar: towards practical verification for java. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 213–226, New York, NY, USA, 2008. ACM. 1.1, 2.2.3

[23] Document Object Model Level 1 specification. http://www.w3.org/TR/REC-DOM-Level-1/. 1.1, 3.1, 3.1, 3.3, 5.1, 6.2, 6.2.1, 6.2.6, 6.2.7, 6.2.8, 6.2.9, 6.3, 6.3.1, 6.3.2, 6.3.2, 6.3.2, 6.3.2, 6.3.2, 6.3.2, 6.3.2, 6.3.2, 6.3.2, 6.3.2, 6.3.2, 6.3.2, 6.3.2, A, A.1, A.1, A.1, A.1, A.1, B.1, B.1, B.1, B.1, B.1, B.1, B.1, B.1, B.1, B.1, B.1, B.1

[24] DOM tests. http://www.w3.org/DOM/Test/. 5.3, 8

[25] XML Specification Section 2.8. http://www.w3.org/TR/REC-xml/#dt-markupdecl. The definition of a DTD is given in this Section of the W3C XML Specification. 6.3.2

[26] Ecmascript language overview. http://www.ecmascript.org/es4/spec/overview.pdf. 1.1

[27] ECMAScript language specification, June 1997. 1.1

[28] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. A programmer's reduction semantics for classes and mixins. In *Formal Syntax and Semantics of Java*, pages 241–269, London, UK, 1999. Springer-Verlag. 1.1

[29] R. W. Floyd. Assigning meanings to programs. *Mathematical aspects of computer science*, 19(19-32):1, 1967. 2.1

[30] Philippa Gardner, Thomas Dinsdale-Young, and Mark Wheelhouse. Abstraction and refinement for local reasoning. In *VSTTE '10: Proceedings of the 3rd International Conference on Verified Software: Theories, Tools and Experiments*, 2010. 2.3.3, 8

[31] Philippa Gardner, Gareth Smith, Mark J. Wheelhouse, and Uri Zarfaty. DOM: Towards a formal specification. In *PLAN-X, a Principles of Programming Languages Workshop*, 2008. 1.3

[32] Philippa Gardner, Gareth Smith, and Adam Wright. Local reasoning about mashups. In *Theory workshop at the 3rd International Conference on Verified Software: Theories, Tools and Experiments*, 2010. 1.3, 9

[33] Philippa Gardner and Uri Zarfaty. Reasoning about high-level tree update and its low-level implementation, 2008. 2.3.3, 5.3

[34] 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. 1.3, 2.3.3

[35] Philippa A. Gardner and Mark J. Wheelhouse. Small specifications for tree update. In *WSFM-09: The 6th International Workshop on Web Services and Formal Methods.*, 2009. 2.3.3, 9

[36] A default attribute test. http://www.doc.ic.ac.uk/~gds/webtests/defaultAttribute.html. C.2

[37] A normalize test. http://www.doc.ic.ac.uk/~gds/webtests/normalize.html. C.1

[38] A specified test. http://www.doc.ic.ac.uk/~gds/webtests/specified.html. C.3

[39] P. Heidegger and P. Thiemann. Recency types for dynamically-typed, object-based languages. In *Foundations of Object-Oriented Languages (FOOL)*, 2009. 1.1

[40] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969. 2.1

[41] C. A. R. Hoare. Proof of a program: Find. *Communications of the ACM*, 14(1):39–45, 1971. 2.1

[42] Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-based distributed programming in java. In *ECOOP '08: Proceedings of the 22nd European conference on Object-Oriented Programming*, pages 516–541, Berlin, Heidelberg, 2008. Springer-Verlag. 1.1

[43] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001. 1.1

[44] 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. 6.3.1

[45] jquery: The write less, do more, javascript library. http://jquery.com. 1.1

[46] Saul Kripke. Semantical analysis of intuitionistic logic. In *Formal Systems and Recursive Functions: Proceedings of the Eighth Logic Colloquium*, pages 92–130, Amsterdam, 1963. North-Holland. 2.2

[47] Lightweight java. http://www.cl.cam.ac.uk/research/pls/javasem/lj/. 1.1

[48] Should Document.cloneNode() work in Level 1? http://markmail.org/message/hanji7cokhvs7xbo. B.1

[49] Sergio Maffeis, John C. Mitchell, and Ankur Taly. An operational semantics for javascript. In G. Ramalingam, editor, *Programming Languages and Systems, 6th Asian Symposium, APLAS 2008, Bangalore, India, December 9-11, 2008. Proceedings*, volume 5356 of *Lecture Notes in Computer Science*, pages 307–325. Springer, 2008. 1.1, 9

[50] Python minidom. http://docs.python.org/lib/module-xml.dom.minidom.html. 8.1

[51] Peter W. OHearn. Resources, concurrency, and local reasoning. *Theoretical Computer Science*, 375(1-3):271–307, 2007. 2.2.3

[52] Peter W. O'Hearn and David J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5:215–244, 1999. 2.2

[53] Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 268–280, New York, NY, USA, 2004. ACM. 2.2.3

[54] Jason Orendorff. Compliance Patches for minidom. Included with Python, 2007. See http://bugs.python.org/issue1704134. 8.1

[55] Peter OHearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In Laurent Fribourg, editor, *Computer Science Logic*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer Berlin / Heidelberg, 2001. 1.1, 2.2, 2.2.3, 4.7, 4.8.2

[56] Matthew J. Parkinson. Local reasoning for Java. Technical Report 654, University of Cambridge Computer Laboratory, November 2005. Ph.D. dissertation. 1.1, 2.2.3

[57] John C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Millennial Perspectives in Computer Science*, pages 303–321. Palgrave, 2000. 1, 1.1, 2.2

[58] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science - LICS*, pages 55–74. IEEE Computer Society, 2002. 1.1, 2.2

[59] 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. 1.1, 2.2

[60] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of javascript programs. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, New York, NY, USA, 2010. ACM. 1.1

[61] Slayer. http://www.eastlondonmassive.org/East_London_Massive/Invader_Home.html. 2.2.3

[62] Smallfoot. http://www.dcs.qmul.ac.uk/research/logic/theory/projects/smallfoot/index.html. 9

[63] Space invader. http://www.eastlondonmassive.org/East_London_Massive/Invader_Home.html. 1.1, 2.2.3, 9

[64] Peter Thiemann. Towards a type system for analyzing JavaScript programs. In Mooly Sagiv, editor, *Programming Languages and Systems*, volume 3444 of *Lecture Notes in Computer Science*, pages 140–140. Springer Berlin / Heidelberg, 2005. 1.1

[65] Viktor Vafeiadis and Matthew Parkinson. A marriage of rely/guarantee and separation logic. In *The 18th International Conference on Concurrency Theory*. Springer, 2007. 2.2.3

[66] The W3C HTML Validator. http://validator.w3.org/. 5.4, 9

[67] Web Standards Project. http://www.webstandards.org/. 8

[68] Information management: A proposal. http://www.w3.org/History/1989/proposal.html. 1.1

[69] WorldWideWeb: Proposal for a hypertext project. http://www.w3.org/Proposal.html, 1990. 1.1

[70] Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter O'Hearn. Scalable shape analysis for systems code. In *CAV '08: Proceedings of the 20th international conference on Computer Aided Verification*, pages 385–398, Berlin, Heidelberg, 2008. Springer-Verlag. 1, 1.1, 2.2.3

[71] Hongseok Yang and Peter OHearn. A semantic basis for local reasoning. In Mogens Nielsen and Uffe Engberg, editors, *Foundations of Software Science and Computation Structures*, volume 2303 of *Lecture Notes in Computer Science*, pages 281–335. Springer Berlin / Heidelberg, 2002. 2.2.3

[72] Uri Zarfaty. *Context Logic and Tree Update*. PhD thesis, Imperial College, 2007. 2.3, 2.3.3, 4.7, 4.8.2, 4.8.3, 4.8.4, 5.2

A. Featherweight DOM

This appendix contains additional material pertinent to Featherweight DOM, as described in Chapters 3 and 4. Recall that Section 3.3 presents the commands essential to the Node interface of [23]. Section A.1 contains implementations of the remaining commands, written in terms of the commands given in Section 3.3. Recall further that Section 5.2 presents the derivation of the weakest precondition for the `getNodeName` command. Section A.2 presents the derivations of the preconditions of all the remaining commands.

A.1. The Remaining Commands

In this section we present the commands from the Node Interface of [23] not covered in Section 3.3.

getNodeValue This command returns the text contained in a text node, or **null** if its argument is an element. First we define a helpful subroutine "getDataLength":

$$
\begin{array}{l}
\texttt{length} := \texttt{getDataLength}(\texttt{node}) \triangleq \\
\quad \texttt{local str} : \\
\quad\quad \texttt{length} := 0 \ ; \\
\quad\quad \texttt{str} := \texttt{substringData}(\texttt{node}, \texttt{length}, 1) \ ; \\
\quad\quad \texttt{while str} \neq \varnothing_{\mathrm{S}} \ \texttt{do} \\
\quad\quad\quad \texttt{length} := \texttt{length} + 1 \ ; \\
\quad\quad\quad \texttt{str} := \texttt{substringData}(\texttt{node}, \texttt{length}, 1) \\
\quad\quad \texttt{od} \\
\quad \texttt{endloc}
\end{array}
$$

Using this command, it is simple to define "getNodeValue":

```
v:=getNodeValue(n) ≜
    local name, len :
        name:=getNodeName(n) ;
        if name = "#text" then
            len := getDataLength(n)v := substringData(n, 0, len)
        else
            v := null
        fi
    endloc
```

setNodeValue This command sets the text of a text node. If called on an element node, it faults.

```
setNodeValue(node, Str) ≜
    local len :
        len := getDataLength(node) ;
        if len = 0 then
            appendData(node, Str)
        else
            replaceData(node, 0, len − 1, Str)
        fi
    endloc
```

getFirstChild and getLastChild These commands are convenient ways to access the first and last child of a node.

```
child:=getFirstChild(n) ≜
    local kids :
        kids:=getChildNodes(n) ;
        child:=item(kids, 0)
    endloc
```

```
child:=getLastChild(n) ≜
    local kids, i, guard, currentNode :
        kids:=getChildNodes(n) ;
        i:=0 ;
        guard:=false ;
        child:=null ;
        while guard do
            currentNode:=item(kids, i) ;
            if currentNode = null then
                guard:=false
            else
                child:=currentNode
            fi ;
            i:=i + 1
        od
    endloc
```

getPreviousSibling and getNextSibling These commands are convenient ways to access a node's immediate siblings.

```
sibling:=getPreviousSibling(n) ≜
    local parent, kids, i, guard, currentNode :
        parent:=getParentNode(n) ;
        if parent = null then
            sibling:=null
        else
            kids:=getChildNodes(parent) ;
            i:=0 ;
            guard:=true ;
            sibling:=null ;
            while guard do
                currentNode:=item(kids, i) ;
                if (currentNode = n) then
                    guard:=false
                else
                    sibling:=currentNode
                fi ;
                i:=i + 1
            od
        fi
    endloc
```

```
sibling:=getNextSibling(n) ≜
    local parent, kids, i, guard, currentNode :
        parent:=getParentNode(n) ;
        if parent = null then
            sibling:=null
        else
            kids:=getChildNodes(parent) ;
            i:=0 ;
            guard:=true ;
            while guard do
                currentNode:=item(kids, i) ;
                if (currentNode = n) then
                    guard:=false ;
                    sibling:=item(kids, i + 1)
                else
                    skip
                fi ;
                i:=i + 1
            od
        fi
    endloc
```

insertBefore As [23] says, this command:

Begin Quote

> Inserts the node newChild before the existing child node refChild. If refChild is null, insert newChild at the end of the list of children.

End Quote

```
insertBefore(parent, newChild, refChild) ≜
    local kids, i, currentNode, foundNewChild, fragKids :
        if refChild = null then
            appendChild(parent, newChild)
        else
            kids:=getChildNodes(parent) ;
            i:=0 ;
            currentNode:=item(kids, i) ;
            foundNewChild:=false ;
            while currentNode ≠ null ∧ currentNode ≠ refChild do
                if currentNode = newChild then
                    foundNewChild:=true
                fi ;
                i:=i + 1 ;
                currentNode:=item(kids, i)
            od ;
            if currentNode = null then
                fault
            fi ;
            appendChild(parent, newChild) ;
            if foundNewChild then
                i:=i − 1
            fi ;
            appendChild(parent, refChild) ;
            currentNode:=item(kids) ;
            while currentNode ≠ newChild do
                appendChild(parent, currentNode) ;
                currentNode:=item(kids, i)
            od
        fi
    endloc
```

replaceChild According to [23], this command:

Begin Quote

> Replaces the child node oldChild with newChild in the list of children, and returns the oldChild node. If the newChild is already in the tree, it is first removed.

End Quote

```
replaceChild(parent, newChild, oldChild) ≜
    insertBefore(parent, newChild, oldChild) ;
    removeChild(parent, oldChild)
```

hasChildNodes [23] describes this method as "a convenience method to allow easy determination of whether a node has any children" which returns "true if the node has any children, false if the node has no children".

```
v:=hasChildNodes(node) ≜
    local name, kids, n :
        name:=getNodeName(node) ;
        if name = "#text" then
            v:=false
        else
            kids:=getChildNodes(node) ;
            n:=item(kids, 0) ;
            if n = null then
                v:=false
            else
                v:=true
            fi
        fi
    endloc
```

cloneNode According to [23], this command:

Begin Quote

> Returns a duplicate of this node, i.e., serves as a generic copy constructor for nodes. The duplicate node has no parent (parentNode returns **null**.).
>
> Cloning an Element copies all attributes and their values, including those generated by the XML processor to represent defaulted attributes, but this method does not copy any text it contains unless it is a deep clone, since the text is contained in a

child Text node. Cloning any other type of node simply returns a copy of this node.

End Quote

The command returns "The duplicate node.", and takes the parameter "deep": "If true, recursively clone the subtree under the specified node; if false, clone only the node itself (and its attributes, if it is an Element)."

```
newNode:=cloneNode(node, deep) ≙
    local name, valstr, kids, i, kid, newKid :
        name:=getNodeName(node) ;
        if name = "#text" then
            valstr:=getNodeValue(node) ;
            newNode:=createTextNode(valstr)
        else
            newNode:=createElement(name) ;
            if deep then
                kids:=getChildNodes(node) ;
                i:=0 ;
                kid:=item(kids, i) ;
                while kid ≠ null do
                    newKid:=cloneNode(kid, true) ;
                    appendChild(newNode, newKid) ;
                    i:=i + 1 ;
                    kid:=item(kids, i)
                od
            fi
        fi
    endloc
```

A.2. Weakest Preconditions

In this section, we present the derivations of the weakest preconditions for Featherweight DOM.

A.2.1. appendChild

$\{(\varnothing_{\mathrm{D}} \multimap (\mathtt{C}{:}\mathrm{ELE}{\to}\mathrm{G} \circ_{\mathrm{ELE}} (\mathtt{NAME}_{\mathtt{parent}}[\mathtt{F}{:}\mathrm{F}]_{\mathtt{FID}}))) \circ_{\mathrm{D}} {<}(\mathtt{NAME}'_{\mathtt{newChild}}[\mathtt{F}'{:}\mathrm{F}]_{\mathtt{FID}'} \vee \text{“\#text”}_{\mathtt{newChild}}\mathtt{VAL}) \wedge \mathtt{T}{:}\mathrm{D}'{>}_{\mathrm{D}}\}$
$\mathtt{appendChild}(\mathtt{parent}, \mathtt{newChild})$
$\{\mathtt{C}{:}\mathrm{ELE}{\to}\mathrm{G} \circ_{\mathrm{ELE}} (\mathtt{NAME}_{\mathtt{parent}}[\mathtt{F}{:}\mathrm{F} \otimes_{\mathrm{F}} {<}\mathtt{T}{:}\mathrm{D}'{>}_{\mathrm{F}}]_{\mathtt{FID}})\}$

FRAME

$$\left\{\begin{array}{l}((\mathtt{C}{:}\mathrm{ELE}{\to}\mathrm{G} \circ_{\mathrm{ELE}} (\mathtt{NAME}_{\mathtt{parent}}[\mathtt{F}{:}\mathrm{F} \otimes_{\mathrm{F}} {<}\mathtt{T}{:}\mathrm{D}'{>}_{\mathrm{F}}]_{\mathtt{FID}})) \multimap P) \circ_{\mathrm{G}} \\ ((\varnothing_{\mathrm{D}} \multimap (\mathtt{C}{:}\mathrm{ELE}{\to}\mathrm{G} \circ_{\mathrm{ELE}} (\mathtt{NAME}_{\mathtt{parent}}[\mathtt{F}{:}\mathrm{F}]_{\mathtt{FID}}))) \circ_{\mathrm{D}} {<}(\mathtt{NAME}'_{\mathtt{newChild}}[\mathtt{F}'{:}\mathrm{F}]_{\mathtt{FID}'} \vee \text{“\#text”}_{\mathtt{newChild}}\mathtt{VAL}) \wedge \mathtt{T}{:}\mathrm{D}'{>}_{\mathrm{D}})\end{array}\right\}$$
$\mathtt{appendChild}(\mathtt{parent}, \mathtt{newChild})$
$$\left\{\begin{array}{l}((\mathtt{C}{:}\mathrm{ELE}{\to}\mathrm{G} \circ_{\mathrm{ELE}} (\mathtt{NAME}_{\mathtt{parent}}[\mathtt{F}{:}\mathrm{F} \otimes_{\mathrm{F}} {<}\mathtt{T}{:}\mathrm{D}'{>}_{\mathrm{F}}]_{\mathtt{FID}})) \multimap P) \circ_{\mathrm{G}} \\ (\mathtt{C}{:}\mathrm{ELE}{\to}\mathrm{G} \circ_{\mathrm{ELE}} (\mathtt{NAME}_{\mathtt{parent}}[\mathtt{F}{:}\mathrm{F} \otimes_{\mathrm{F}} {<}\mathtt{T}{:}\mathrm{D}'{>}_{\mathrm{F}}]_{\mathtt{FID}}))\end{array}\right\}$$

CONS

$$\left\{\begin{array}{l}((\mathtt{C}{:}\mathrm{ELE}{\to}\mathrm{G} \circ_{\mathrm{ELE}} (\mathtt{NAME}_{\mathtt{parent}}[\mathtt{F}{:}\mathrm{F} \otimes_{\mathrm{F}} {<}\mathtt{T}{:}\mathrm{D}'{>}_{\mathrm{F}}]_{\mathtt{FID}})) \multimap P) \circ_{\mathrm{G}} \\ ((\varnothing_{\mathrm{D}} \multimap (\mathtt{C}{:}\mathrm{ELE}{\to}\mathrm{G} \circ_{\mathrm{ELE}} (\mathtt{NAME}_{\mathtt{parent}}[\mathtt{F}{:}\mathrm{F}]_{\mathtt{FID}}))) \circ_{\mathrm{D}} {<}(\mathtt{NAME}'_{\mathtt{newChild}}[\mathtt{F}'{:}\mathrm{F}]_{\mathtt{FID}'} \vee \text{“\#text”}_{\mathtt{newChild}}\mathtt{VAL}) \wedge \mathtt{T}{:}\mathrm{D}'{>}_{\mathrm{D}})\end{array}\right\}$$
$\mathtt{appendChild}(\mathtt{parent}, \mathtt{newChild})$
$\{P\}$

ELIM

$$\left\{\begin{array}{l}\exists \mathtt{C}, \mathtt{NAME}, \mathtt{F}, \mathtt{T}, \mathtt{FID}, \mathtt{NAME}', \mathtt{F}', \mathtt{FID}', \mathtt{VAL}. \\ ((\mathtt{C}{:}\mathrm{ELE}{\to}\mathrm{G} \circ_{\mathrm{ELE}} (\mathtt{NAME}_{\mathtt{parent}}[\mathtt{F}{:}\mathrm{F} \otimes_{\mathrm{F}} {<}\mathtt{T}{:}\mathrm{D}'{>}_{\mathrm{F}}]_{\mathtt{FID}})) \multimap P) \circ_{\mathrm{G}} \\ ((\varnothing_{\mathrm{D}} \multimap (\mathtt{C}{:}\mathrm{ELE}{\to}\mathrm{G} \circ_{\mathrm{ELE}} (\mathtt{NAME}_{\mathtt{parent}}[\mathtt{F}{:}\mathrm{F}]_{\mathtt{FID}}))) \circ_{\mathrm{D}} {<}(\mathtt{NAME}'_{\mathtt{newChild}}[\mathtt{F}'{:}\mathrm{F}]_{\mathtt{FID}'} \vee \text{“\#text”}_{\mathtt{newChild}}\mathtt{VAL}) \wedge \mathtt{T}{:}\mathrm{D}'{>}_{\mathrm{D}})\end{array}\right\}$$
$\mathtt{appendChild}(\mathtt{parent}, \mathtt{newChild})$
$\{P\}$

where $\mathrm{D} \in \{\mathrm{F}, \mathrm{G}\}, \mathrm{D}' \in \{\mathrm{ELE}, \mathrm{TXT}\}$

A.2.2. removeChild

$$\left\{<\texttt{C}:\text{ELE}\rightarrow\text{G} \circ_{\text{ELE}} (\texttt{NAME}_{\texttt{parent}}[\texttt{F}_1:\text{F} \otimes_{\text{F}} <(\texttt{NAME}'_{\texttt{oldChild}}[\texttt{F}':\text{F}]_{\texttt{FID}'} \vee \text{“\#text”}_{\texttt{oldChild}}\texttt{VAL}) \wedge \texttt{T}:\text{D}>_{\text{F}} \otimes_{\text{F}} \texttt{F}_2:\text{F}]_{\texttt{FID}})>_{\text{G}}\right\}$$
$$\texttt{removeChild}(\texttt{parent}, \texttt{oldChild})$$
$$\left\{<\texttt{C}:\text{ELE}\rightarrow\text{G} \circ_{\text{ELE}} (\texttt{NAME}_{\texttt{parent}}[\texttt{F}_1:\text{F} \otimes_{\text{F}} \texttt{F}_2:\text{F}]_{\texttt{FID}})>_{\text{G}} \oplus <\texttt{T}:\text{D}>_{\text{G}}\right\}$$

FRAME

$$\left\{\begin{array}{l}((<\texttt{C}:\text{ELE}\rightarrow\text{G} \circ_{\text{ELE}} (\texttt{NAME}_{\texttt{parent}}[\texttt{F}_1:\text{F} \otimes_{\text{F}} \texttt{F}_2:\text{F}]_{\texttt{FID}})>_{\text{G}} \oplus <\texttt{T}:\text{D}>_{\text{G}}) \multimap P)\circ_{\text{G}} \\ (<\texttt{C}:\text{ELE}\rightarrow\text{G} \circ_{\text{ELE}} (\texttt{NAME}_{\texttt{parent}}[\texttt{F}_1:\text{F} \otimes_{\text{F}} <(\texttt{NAME}'_{\texttt{oldChild}}[\texttt{F}':\text{F}]_{\texttt{FID}'} \vee \text{“\#text”}_{\texttt{oldChild}}\texttt{VAL}) \wedge \texttt{T}:\text{D}>_{\text{F}} \otimes_{\text{F}} \texttt{F}_2:\text{F}]_{\texttt{FID}})>_{\text{G}})\end{array}\right\}$$
$$\texttt{removeChild}(\texttt{parent}, \texttt{oldChild})$$
$$\left\{\begin{array}{l}((<\texttt{C}:\text{ELE}\rightarrow\text{G} \circ_{\text{ELE}} (\texttt{NAME}_{\texttt{parent}}[\texttt{F}_1:\text{F} \otimes_{\text{F}} \texttt{F}_2:\text{F}]_{\texttt{FID}})>_{\text{G}} \oplus <\texttt{T}:\text{D}>_{\text{G}}) \multimap P)\circ_{\text{G}} \\ (<\texttt{C}:\text{ELE}\rightarrow\text{G} \circ_{\text{ELE}} (\texttt{NAME}_{\texttt{parent}}[\texttt{F}_1:\text{F} \otimes_{\text{F}} \texttt{F}_2:\text{F}]_{\texttt{FID}})>_{\text{G}} \oplus <\texttt{T}:\text{D}>_{\text{G}})\end{array}\right\}$$

CONS

$$\left\{\begin{array}{l}((<\texttt{C}:\text{ELE}\rightarrow\text{G} \circ_{\text{ELE}} (\texttt{NAME}_{\texttt{parent}}[\texttt{F}_1:\text{F} \otimes_{\text{F}} \texttt{F}_2:\text{F}]_{\texttt{FID}})>_{\text{G}} \oplus <\texttt{T}:\text{D}>_{\text{G}}) \multimap P)\circ_{\text{G}} \\ (<\texttt{C}:\text{ELE}\rightarrow\text{G} \circ_{\text{ELE}} (\texttt{NAME}_{\texttt{parent}}[\texttt{F}_1:\text{F} \otimes_{\text{F}} <(\texttt{NAME}'_{\texttt{oldChild}}[\texttt{F}':\text{F}]_{\texttt{FID}'} \vee \text{“\#text”}_{\texttt{oldChild}}\texttt{VAL}) \wedge \texttt{T}:\text{D}>_{\text{F}} \otimes_{\text{F}} \texttt{F}_2:\text{F}]_{\texttt{FID}})>_{\text{G}})\end{array}\right\}$$
$$\texttt{removeChild}(\texttt{parent}, \texttt{oldChild})$$
$$\left\{P\right\}$$

ELIM

$$\left\{\begin{array}{l}\exists \texttt{C}, \texttt{NAME}, \texttt{F}_1, \texttt{F}_2, \texttt{FID}, \texttt{T}, \texttt{NAME}', \texttt{F}', \texttt{FID}', \texttt{VAL}.\ ((<\texttt{C}:\text{ELE}\rightarrow\text{G} \circ_{\text{ELE}} (\texttt{NAME}_{\texttt{parent}}[\texttt{F}_1:\text{F} \otimes_{\text{F}} \texttt{F}_2:\text{F}]_{\texttt{FID}})>_{\text{G}} \oplus <\texttt{T}:\text{D}>_{\text{G}}) \multimap P)\circ_{\text{G}} \\ (<\texttt{C}:\text{ELE}\rightarrow\text{G} \circ_{\text{ELE}} (\texttt{NAME}_{\texttt{parent}}[\texttt{F}_1:\text{F} \otimes_{\text{F}} <(\texttt{NAME}'_{\texttt{oldChild}}[\texttt{F}':\text{F}]_{\texttt{FID}'} \vee \text{“\#text”}_{\texttt{oldChild}}\texttt{VAL}) \wedge \texttt{T}:\text{D}>_{\text{F}} \otimes_{\text{F}} \texttt{F}_2:\text{F}]_{\texttt{FID}})>_{\text{G}})\end{array}\right\}$$
$$\texttt{removeChild}(\texttt{parent}, \texttt{oldChild})$$
$$\left\{P\right\}$$

where $\text{D} \in \{\text{ELE}, \text{TXT}\}$

A.2.3. getNodeName

$$\Big\{(\texttt{NAME}_{\texttt{node}}[\texttt{F:F}]_{\texttt{FID}} \vee \text{“\#text”}_{\texttt{node}}\texttt{VAL}) \wedge \texttt{T:D}\Big\}$$
$$\texttt{var} := \texttt{getNodeName}(\texttt{node})$$
$$\Big\{\texttt{T:D} \wedge (\texttt{var} \doteq \texttt{NAME})\Big\}$$

FRAME

$$\left\{\begin{array}{l}(\texttt{T:D} \multimap P\{\texttt{NAME}/\texttt{var}\}) \circ_{\texttt{D}} \\ ((\texttt{NAME}_{\texttt{node}}[\texttt{F:F}]_{\texttt{FID}} \vee \text{“\#text”}_{\texttt{node}}\texttt{VAL}) \wedge \texttt{T:D})\end{array}\right\}$$
$$\texttt{var} := \texttt{getNodeName}(\texttt{node})$$
$$\left\{\begin{array}{l}(\texttt{T:D} \multimap P\{\texttt{NAME}/\texttt{var}\}) \circ_{\texttt{D}} \\ (\texttt{T:D} \wedge (\texttt{var} \doteq \texttt{NAME}))\end{array}\right\}$$

CONS

$$\Big\{\Diamond_{\texttt{D}\to\texttt{D}'}((\texttt{NAME}_{\texttt{node}}[\texttt{F:F}]_{\texttt{FID}} \vee \text{“\#text”}_{\texttt{node}}\texttt{VAL}) \wedge \texttt{T:D}) \wedge P\{\texttt{NAME}/\texttt{var}\}\Big\}$$
$$\texttt{var} := \texttt{getNodeName}(\texttt{node})$$
$$\Big\{P\{\texttt{NAME}/\texttt{var}\} \wedge (\texttt{var} \doteq \texttt{NAME})\Big\}$$

CONS/ELIM

$$\left\{\begin{array}{l}\exists \texttt{NAME}, \texttt{F}, \texttt{FID}, \texttt{VAL}, \texttt{T}. \\ \Diamond_{\texttt{D}\to\texttt{D}'}((\texttt{NAME}_{\texttt{node}}[\texttt{F:F}]_{\texttt{FID}} \vee \text{“\#text”}_{\texttt{node}}\texttt{VAL}) \wedge \texttt{T:D}) \wedge P\{\texttt{NAME}/\texttt{var}\}\end{array}\right\}$$
$$\texttt{var} := \texttt{getNodeName}(\texttt{node})$$
$$\Big\{P\Big\}$$

where $\text{D} \in \{\text{ELE}, \text{TXT}\}, \text{D}' \in \{\text{ELE}, \text{TXT}, \text{F}, \text{G}\}$

A.2.4. getParentNode

$$\left\{\mathtt{NAME}'_{\mathtt{ID}'}[\mathtt{F}_1{:}\mathrm{F} \otimes_\mathrm{F} {<}(\mathtt{NAME}_{\mathtt{node}}[\mathtt{F}{:}\mathrm{F}]_{\mathrm{FID}} \vee \text{“\#text”}_{\mathtt{node}}\mathtt{VAL}) \wedge \mathtt{T}{:}\mathrm{D}{>}_\mathrm{F} \otimes_\mathrm{F} \mathtt{F}_2{:}\mathrm{F}]_{\mathrm{FID}'}\right\}$$
$$\mathtt{var} := \mathtt{getParentNode}(\mathtt{node})$$
$$\left\{\mathtt{NAME}'_{\mathtt{ID}'}[\mathtt{F}_1{:}\mathrm{F} \otimes_\mathrm{F} {<}\mathtt{T}{:}\mathrm{D}{>}_\mathrm{F} \otimes_\mathrm{F} \mathtt{F}_2{:}\mathrm{F}]_{\mathrm{FID}'} \wedge (\mathtt{var} \doteq \mathtt{ID}')\right\}$$

FRAME

$$\left\{\begin{array}{l}((\mathtt{NAME}'_{\mathtt{ID}'}[\mathtt{F}_1{:}\mathrm{F} \otimes_\mathrm{F} {<}\mathtt{T}{:}\mathrm{D}{>}_\mathrm{F} \otimes_\mathrm{F} \mathtt{F}_2{:}\mathrm{F}]_{\mathrm{FID}'}) \multimap P\{\mathtt{ID}'/\mathtt{var}\}) \circ_{\mathrm{ELE}} \\ (\mathtt{NAME}'_{\mathtt{ID}'}[\mathtt{F}_1{:}\mathrm{F} \otimes_\mathrm{F} {<}(\mathtt{NAME}_{\mathtt{node}}[\mathtt{F}{:}\mathrm{F}]_{\mathrm{FID}} \vee \text{“\#text”}_{\mathtt{node}}\mathtt{VAL}) \wedge \mathtt{T}{:}\mathrm{D}{>}_\mathrm{F} \otimes_\mathrm{F} \mathtt{F}_2{:}\mathrm{F}]_{\mathrm{FID}'})\end{array}\right\}$$
$$\mathtt{var} := \mathtt{getParentNode}(\mathtt{node})$$
$$\left\{\begin{array}{l}((\mathtt{NAME}'_{\mathtt{ID}'}[\mathtt{F}_1{:}\mathrm{F} \otimes_\mathrm{F} {<}\mathtt{T}{:}\mathrm{D}{>}_\mathrm{F} \otimes_\mathrm{F} \mathtt{F}_2{:}\mathrm{F}]_{\mathrm{FID}'}) \multimap P\{\mathtt{ID}'/\mathtt{var}\}) \circ_{\mathrm{ELE}} \\ (\mathtt{NAME}'_{\mathtt{ID}'}[\mathtt{F}_1{:}\mathrm{F} \otimes_\mathrm{F} {<}\mathtt{T}{:}\mathrm{D}{>}_\mathrm{F} \otimes_\mathrm{F} \mathtt{F}_2{:}\mathrm{F}]_{\mathrm{FID}'} \wedge (\mathtt{var} \doteq \mathtt{ID}'))\end{array}\right\}$$

CONS

$$\left\{\begin{array}{l}\Diamond_{\mathrm{ELE}\to\mathrm{D}'}\mathtt{NAME}'_{\mathtt{ID}'}[\mathtt{F}_1{:}\mathrm{F} \otimes_\mathrm{F} {<}(\mathtt{NAME}_{\mathtt{node}}[\mathtt{F}{:}\mathrm{F}]_{\mathrm{FID}} \vee \text{“\#text”}_{\mathtt{node}}\mathtt{VAL}) \wedge \mathtt{T}{:}\mathrm{D}{>}_\mathrm{F} \otimes_\mathrm{F} \mathtt{F}_2{:}\mathrm{F}]_{\mathrm{FID}'} \\ \wedge P\{\mathtt{ID}'/\mathtt{var}\} \circ_{\mathrm{ELE}}\end{array}\right\}$$
$$\mathtt{var} := \mathtt{getParentNode}(\mathtt{node})$$
$$\left\{P\{\mathtt{ID}'/\mathtt{var}\} \wedge (\mathtt{var} \doteq \mathtt{ID}')\right\}$$

CONS/ELIM

$$\left\{\begin{array}{l}\exists \mathtt{NAME}', \mathtt{ID}', \mathtt{F}_1, \mathtt{NAME}, \mathtt{F}, \mathtt{FID}, \mathtt{VAL}, \mathtt{T}, \mathtt{F}_2, \mathtt{FID}'. \\ \Diamond_{\mathrm{ELE}\to\mathrm{D}'}\mathtt{NAME}'_{\mathtt{ID}'}[\mathtt{F}_1{:}\mathrm{F} \otimes_\mathrm{F} {<}(\mathtt{NAME}_{\mathtt{node}}[\mathtt{F}{:}\mathrm{F}]_{\mathrm{FID}} \vee \text{“\#text”}_{\mathtt{node}}\mathtt{VAL}) \wedge \mathtt{T}{:}\mathrm{D}{>}_\mathrm{F} \otimes_\mathrm{F} \mathtt{F}_2{:}\mathrm{F}]_{\mathrm{FID}'} \\ \wedge P\{\mathtt{ID}'/\mathtt{var}\} \circ_{\mathrm{ELE}}\end{array}\right\}$$
$$\mathtt{var} := \mathtt{getParentNode}(\mathtt{node})$$
$$\left\{P\right\}$$

where $\mathrm{D} \in \{\mathrm{ELE}, \mathrm{TXT}\}, \mathrm{D}' \in \{\mathrm{ELE}, \mathrm{F}, \mathrm{G}\}$

$$\left\{ <(\texttt{NAME}_{\texttt{node}}[\texttt{F:F}]_{\texttt{FID}} \vee \text{“\#text”}_{\texttt{node}}\texttt{VAL}) \wedge \texttt{T:D}>_{\texttt{G}} \right\}$$
$$\texttt{var} := \texttt{getParentNode}(\texttt{node})$$
$$\left\{ <\texttt{T:D}>_{\texttt{G}} \wedge (\texttt{var} \doteq \textbf{null}) \right\}$$

FRAME

$$\left\{ \begin{array}{l} (<\texttt{T:D}>_{\texttt{G}} \multimap P\{\textbf{null}/\texttt{var}\}) \circ_{\texttt{D}} \\ (<(\texttt{NAME}_{\texttt{node}}[\texttt{F:F}]_{\texttt{FID}} \vee \text{“\#text”}_{\texttt{node}}\texttt{VAL}) \wedge \texttt{T:D}>_{\texttt{G}}) \end{array} \right\}$$
$$\texttt{var} := \texttt{getParentNode}(\texttt{node})$$
$$\left\{ \begin{array}{l} (<\texttt{T:D}>_{\texttt{G}} \multimap P\{\textbf{null}/\texttt{var}\}) \circ_{\texttt{D}} \\ (<\texttt{T:D}>_{\texttt{G}} \wedge (\texttt{var} \doteq \textbf{null})) \end{array} \right\}$$

CONS

$$\left\{ \Diamond_{\texttt{G}\to\texttt{G}} <(\texttt{NAME}_{\texttt{node}}[\texttt{F:F}]_{\texttt{FID}} \vee \text{“\#text”}_{\texttt{node}}\texttt{VAL}) \wedge \texttt{T:D}>_{\texttt{G}} \wedge P\{\textbf{null}/\texttt{var}\} \right\}$$
$$\texttt{var} := \texttt{getParentNode}(\texttt{node})$$
$$\left\{ P\{\textbf{null}/\texttt{var}\} \wedge (\texttt{var} \doteq \textbf{null})) \right\}$$

CONS/ELIM

$$\left\{ \begin{array}{l} \exists \texttt{NAME}, \texttt{F}, \texttt{FID}, \texttt{VAL}, \texttt{T}. \\ \Diamond_{\texttt{G}\to\texttt{G}} <(\texttt{NAME}_{\texttt{node}}[\texttt{F:F}]_{\texttt{FID}} \vee \text{“\#text”}_{\texttt{node}}\texttt{VAL}) \wedge \texttt{T:D}>_{\texttt{G}} \wedge P\{\textbf{null}/\texttt{var}\} \end{array} \right\}$$
$$\texttt{var} := \texttt{getParentNode}(\texttt{node})$$
$$\left\{ P \right\}$$

where $\texttt{D} \in \{\text{ELE}, \text{TXT}\}$

A.2.5. getChildNodes

$$\left\{ \texttt{NAME}_{\texttt{Y}}[\texttt{F:F}]_{\texttt{FID}} \wedge \texttt{node} \doteq \texttt{Y} \right\}$$
$$\texttt{list} := \texttt{getChildNodes}(\texttt{node})$$
$$\left\{ \texttt{NAME}_{\texttt{Y}}[\texttt{F:F}]_{\texttt{FID}} \wedge (\texttt{list} \doteq \texttt{FID}) \right\}$$

FRAME

$$\left\{ \begin{array}{l} (\texttt{NAME}_{\texttt{Y}}[\texttt{F:F}]_{\texttt{FID}} \multimap P\{\texttt{FID}/\texttt{list}\}) \circ_{\text{ELE}} \\ (\texttt{NAME}_{\texttt{Y}}[\texttt{F:F}]_{\texttt{FID}} \wedge \texttt{node} \doteq \texttt{Y}) \end{array} \right\}$$
$$\texttt{list} := \texttt{getChildNodes}(\texttt{node})$$
$$\left\{ \begin{array}{l} (\texttt{NAME}_{\texttt{Y}}[\texttt{F:F}]_{\texttt{FID}} \multimap P\{\texttt{FID}/\texttt{list}\}) \circ_{\text{ELE}} \\ (\texttt{NAME}_{\texttt{Y}}[\texttt{F:F}]_{\texttt{FID}} \wedge (\texttt{list} \doteq \texttt{FID})) \end{array} \right\}$$

CONS

$$\left\{ \Diamond_{\text{ELE}\to\texttt{D}} \texttt{NAME}_{\texttt{Y}}[\texttt{F:F}]_{\texttt{FID}} \wedge P\{\texttt{FID}/\texttt{list}\} \wedge (\texttt{node} \doteq \texttt{Y}) \right\}$$
$$\texttt{list} := \texttt{getChildNodes}(\texttt{node})$$
$$\left\{ P\{\texttt{FID}/\texttt{list}\} \wedge (\texttt{list} \doteq \texttt{FID}) \right\}$$

CONS/ELIM

$$\left\{ \begin{array}{l} \exists \texttt{NAME}, \texttt{Y}, \texttt{F}, \texttt{FID}. \\ \Diamond_{\text{ELE}\to\texttt{D}} \texttt{NAME}_{\texttt{Y}}[\texttt{F:F}]_{\texttt{FID}} \wedge P\{\texttt{FID}/\texttt{list}\} \wedge (\texttt{node} \doteq \texttt{Y}) \end{array} \right\}$$
$$\texttt{list} := \texttt{getChildNodes}(\texttt{node})$$
$$\left\{ P \right\}$$

A.2.6. createElement

$$\Big\{\varnothing_{\mathrm{G}} \wedge \mathtt{Name} \in \mathrm{S} \wedge \text{`}\#' \notin \mathtt{Name} \wedge \mathtt{node} \doteq \mathtt{Y}\Big\}$$
$$\mathtt{node} := \mathtt{createElement}(\mathtt{Name})$$
$$\Big\{<\mathtt{Name}\{\mathtt{Y}/\mathtt{node}\}_{\mathtt{node}}[\varnothing_{\mathrm{F}}]_{\mathtt{FID}}>_{\mathrm{G}}\Big\}$$

FRAME/ELIM

$$\left\{\begin{array}{l}\exists \mathtt{Y}.\ (\forall \mathtt{NODE}, \mathtt{FID}.\\ <\mathtt{Name}\{\mathtt{Y}/\mathtt{node}\}_{\mathtt{NODE}}[\varnothing_{\mathrm{F}}]_{\mathtt{FID}}>_{\mathrm{G}} \multimap P\{\mathtt{NODE}/\mathtt{node}\}) \circ_{\mathrm{G}}\\ (\varnothing_{\mathrm{G}} \wedge \mathtt{Name} \in \mathrm{S} \wedge \text{`}\#' \notin \mathtt{Name} \wedge \mathtt{node} \doteq \mathtt{Y})\end{array}\right\}$$
$$\mathtt{node} := \mathtt{createElement}(\mathtt{Name})$$
$$\left\{\begin{array}{l}\exists \mathtt{Y}.\ (\forall \mathtt{NODE}, \mathtt{FID}.\\ <\mathtt{Name}\{\mathtt{Y}/\mathtt{node}\}_{\mathtt{NODE}}[\varnothing_{\mathrm{F}}]_{\mathtt{FID}}>_{\mathrm{G}} \multimap P\{\mathtt{NODE}/\mathtt{node}\}) \circ_{\mathrm{G}}\\ <\mathtt{Name}\{\mathtt{Y}/\mathtt{node}\}_{\mathtt{node}}[\varnothing_{\mathrm{F}}]_{\mathtt{FID}}>_{\mathrm{G}}\end{array}\right\}$$

CONS

$$\left\{\begin{array}{l}\forall \mathtt{NODE}, \mathtt{FID}.\\ (<\mathtt{Name}_{\mathtt{NODE}}[\varnothing_{\mathrm{F}}]_{\mathtt{FID}}>_{\mathrm{G}} \multimap P\{\mathtt{NODE}/\mathtt{node}\}) \circ_{\mathrm{G}}\\ (\varnothing_{\mathrm{G}} \wedge \mathtt{Name} \in \mathrm{S} \wedge \text{`}\#' \notin \mathtt{Name})\end{array}\right\}$$
$$\mathtt{node} := \mathtt{createElement}(\mathtt{Name})$$
$$\Big\{P\Big\}$$

A.2.7. item

$$\left\{\begin{array}{l}\mathtt{NAME}_{\mathtt{ID}}[\mathtt{F}_1{:}\mathrm{F} \otimes_{\mathrm{F}} <(\mathtt{NAME}'_{\mathtt{ID}'}[\mathtt{F}'{:}\mathrm{F}]_{\mathrm{FID}'} \vee \text{“}\#\mathrm{text}\text{”}_{\mathtt{ID}'}\mathtt{VAL}') \wedge \mathtt{T}{:}\mathrm{D}>_{\mathrm{F}} \otimes_{\mathrm{F}} \mathtt{F}_2{:}\mathrm{F}]_{\mathtt{list}}\\ \wedge (\mathtt{Int} \doteq \mathrm{len}(\mathtt{F}_1)) \wedge \mathtt{Y} \doteq \mathtt{list}\end{array}\right\}$$
$$\mathtt{node} := \mathtt{item}(\mathtt{list}, \mathtt{Int})$$
$$\Big\{\mathtt{NAME}_{\mathtt{ID}}[\mathtt{F}_1{:}\mathrm{F} \otimes_{\mathrm{F}} <\mathtt{T}{:}\mathrm{D}>_{\mathrm{F}} \otimes_{\mathrm{F}} \mathtt{F}_2{:}\mathrm{F}]_{\mathtt{Y}} \wedge (\mathtt{node} \doteq \mathtt{ID}')\Big\}$$

FRAME

$$\left\{\begin{array}{l}(\mathtt{NAME}_{\mathtt{ID}}[\mathtt{F}_1{:}\mathrm{F} \otimes_{\mathrm{F}} <\mathtt{T}{:}\mathrm{D}>_{\mathrm{F}} \otimes_{\mathrm{F}} \mathtt{F}_2{:}\mathrm{F}]_{\mathtt{Y}} \multimap P\{\mathtt{ID}'/\mathtt{node}\}) \circ_{\mathrm{ELE}}\\ (\mathtt{NAME}_{\mathtt{ID}}[\mathtt{F}_1{:}\mathrm{F} \otimes_{\mathrm{F}} <(\mathtt{NAME}'_{\mathtt{ID}'}[\mathtt{F}'{:}\mathrm{F}]_{\mathrm{FID}'} \vee \text{“}\#\mathrm{text}\text{”}_{\mathtt{ID}'}\mathtt{VAL}') \wedge \mathtt{T}{:}\mathrm{D}>_{\mathrm{F}} \otimes_{\mathrm{F}} \mathtt{F}_2{:}\mathrm{F}]_{\mathtt{list}}\\ \wedge (\mathtt{Int} \doteq \mathrm{len}(\mathtt{F}_1)) \wedge \mathtt{Y} \doteq \mathtt{list})\end{array}\right\}$$
$$\mathtt{node} := \mathtt{item}(\mathtt{list}, \mathtt{Int})$$
$$\left\{\begin{array}{l}(\mathtt{NAME}_{\mathtt{ID}}[\mathtt{F}_1{:}\mathrm{F} \otimes_{\mathrm{F}} <\mathtt{T}{:}\mathrm{D}>_{\mathrm{F}} \otimes_{\mathrm{F}} \mathtt{F}_2{:}\mathrm{F}]_{\mathtt{Y}} \multimap P\{\mathtt{ID}'/\mathtt{node}\}) \circ_{\mathrm{ELE}}\\ (\mathtt{NAME}_{\mathtt{ID}}[\mathtt{F}_1{:}\mathrm{F} \otimes_{\mathrm{F}} <\mathtt{T}{:}\mathrm{D}>_{\mathrm{F}} \otimes_{\mathrm{F}} \mathtt{F}_2{:}\mathrm{F}]_{\mathtt{Y}} \wedge (\mathtt{node} \doteq \mathtt{ID}'))\end{array}\right\}$$

CONS

$$\left\{\begin{array}{l}\Diamond_{\mathrm{ELE}\to\mathrm{D}'}(\mathtt{NAME}_{\mathtt{ID}}[\mathtt{F}_1{:}\mathrm{F} \otimes_{\mathrm{F}} <(\mathtt{NAME}'_{\mathtt{ID}'}[\mathtt{F}'{:}\mathrm{F}]_{\mathrm{FID}'} \vee \text{“}\#\mathrm{text}\text{”}_{\mathtt{ID}'}\mathtt{VAL}') \wedge \mathtt{T}{:}\mathrm{D}>_{\mathrm{F}} \otimes_{\mathrm{F}} \mathtt{F}_2{:}\mathrm{F}]_{\mathtt{Y}})\\ \wedge P\{\mathtt{ID}'/\mathtt{node}\} \wedge (\mathtt{Int} \doteq \mathrm{len}(\mathtt{F}_1)) \wedge (\mathtt{Y} \doteq \mathtt{list})\end{array}\right\}$$
$$\mathtt{node} := \mathtt{item}(\mathtt{list}, \mathtt{Int})$$
$$\Big\{P\{\mathtt{ID}'/\mathtt{node}\} \wedge (\mathtt{node} \doteq \mathtt{ID}'))\Big\}$$

CONS/ELIM

$$\left\{\begin{array}{l}\exists \mathtt{NAME}, \mathtt{ID}, \mathtt{F}_1, \mathtt{NAME}', \mathtt{ID}', \mathtt{F}', \mathtt{FID}', \mathtt{VAL}'\mathtt{T}, \mathtt{F}_2.\\ \Diamond_{\mathrm{ELE}\to\mathrm{D}'}(\mathtt{NAME}_{\mathtt{ID}}[\mathtt{F}_1{:}\mathrm{F} \otimes_{\mathrm{F}} <(\mathtt{NAME}'_{\mathtt{ID}'}[\mathtt{F}'{:}\mathrm{F}]_{\mathrm{FID}'} \vee \text{“}\#\mathrm{text}\text{”}_{\mathtt{ID}'}\mathtt{VAL}') \wedge \mathtt{T}{:}\mathrm{D}>_{\mathrm{F}} \otimes_{\mathrm{F}} \mathtt{F}_2{:}\mathrm{F}]_{\mathtt{list}})\\ \wedge P\{\mathtt{ID}'/\mathtt{node}\} \wedge (\mathtt{Int} \doteq \mathrm{len}(\mathtt{F}_1))\end{array}\right\}$$
$$\mathtt{node} := \mathtt{item}(\mathtt{list}, \mathtt{Int})$$
$$\Big\{P\Big\}$$

where $\mathrm{D} \in \{\mathrm{ELE}, \mathrm{TXT}\}, \mathrm{D}' \in \{\mathrm{ELE}, \mathrm{F}, \mathrm{G}\}$

$$\left\{\texttt{NAME}_{\texttt{ID}}[\texttt{F}{:}\mathrm{F}]_{\texttt{list}} \wedge \texttt{list} \doteq \texttt{Y} \wedge (\texttt{Int} <\cdot\, 0 \vee \texttt{Int} \doteq\!> \mathrm{len}(\texttt{F}))\right\}$$
$$\texttt{node} := \texttt{item}(\texttt{list}, \texttt{Int})$$
$$\left\{\texttt{NAME}_{\texttt{ID}}[\texttt{F}{:}\mathrm{F}]_{\texttt{Y}} \wedge (\texttt{node} \doteq \textbf{null})\right\}$$

FRAME

$$\left\{\begin{array}{l}(\texttt{NAME}_{\texttt{ID}}[\texttt{F}{:}\mathrm{F}]_{\texttt{Y}} \multimap P\{\textbf{null}/\texttt{node}\}) \circ_{\mathrm{ELE}} \\ (\texttt{NAME}_{\texttt{ID}}[\texttt{F}{:}\mathrm{F}]_{\texttt{list}} \wedge \texttt{list} \doteq \texttt{Y} \wedge (\texttt{Int} <\cdot\, 0 \vee \texttt{Int} \doteq\!> \mathrm{len}(\texttt{F})))\end{array}\right\}$$
$$\texttt{node} := \texttt{item}(\texttt{list}, \texttt{Int})$$
$$\left\{\begin{array}{l}(\texttt{NAME}_{\texttt{ID}}[\texttt{F}{:}\mathrm{F}]_{\texttt{Y}} \multimap P\{\textbf{null}/\texttt{node}\}) \circ_{\mathrm{ELE}} \\ (\texttt{NAME}_{\texttt{ID}}[\texttt{F}{:}\mathrm{F}]_{\texttt{Y}} \wedge (\texttt{node} \doteq \textbf{null}))\end{array}\right\}$$

CONS

$$\left\{\begin{array}{l}\Diamond_{\mathrm{ELE}\to\mathrm{D}}(\texttt{NAME}_{\texttt{ID}}[\texttt{F}{:}\mathrm{F}]_{\texttt{list}}) \wedge P\{\textbf{null}/\texttt{node}\} \\ \wedge\, (\texttt{Int} <\cdot\, 0 \vee \texttt{Int} \doteq\!> \mathrm{len}(\texttt{F})))\end{array}\right\}$$
$$\texttt{node} := \texttt{item}(\texttt{list}, \texttt{Int})$$
$$\left\{P\{\textbf{null}/\texttt{node}\} \wedge (\texttt{node} \doteq \textbf{null})\right\}$$

CONS/ELIM

$$\left\{\begin{array}{l}\exists \texttt{NAME}, \texttt{ID}, \texttt{F}. \\ \Diamond_{\mathrm{ELE}\to\mathrm{D}}(\texttt{NAME}_{\texttt{ID}}[\texttt{F}{:}\mathrm{F}]_{\texttt{list}}) \wedge P\{\textbf{null}/\texttt{node}\} \\ \wedge\, (\texttt{Int} <\cdot\, 0 \vee \texttt{Int} \doteq\!> \mathrm{len}(\texttt{F})))\end{array}\right\}$$
$$\texttt{node} := \texttt{item}(\texttt{list}, \texttt{Int})$$
$$\left\{P\right\}$$

where $\mathrm{D} \in \{\mathrm{ELE}, \mathrm{F}, \mathrm{G}\}$

A.2.8. substringData

$$\left\{\begin{array}{l}\text{“\#text”}_{\mathtt{Y}}\mathtt{STR}_1 \otimes_{\mathrm{S}} \mathtt{STR}' \otimes_{\mathrm{S}} \mathtt{STR}_2 \wedge (\mathtt{Offset} \doteq \mathrm{len}(\mathtt{STR}_1)) \\ \wedge\, (\mathtt{Count} \doteq \mathrm{len}(\mathtt{STR}') \wedge \mathtt{node} \doteq \mathtt{Y})\end{array}\right\}$$

$\mathtt{str}{:=}\mathtt{substringData}(\mathtt{node},\mathtt{Offset},\mathtt{Count})$

$$\left\{\text{“\#text”}_{\mathtt{Y}}\mathtt{STR}_1 \otimes_{\mathrm{S}} \mathtt{STR}' \otimes_{\mathrm{S}} \mathtt{STR}_2 \wedge (\mathtt{str} \doteq \mathtt{STR}')\right\}$$

FRAME

$$\left\{\begin{array}{l}(\text{“\#text”}_{\mathtt{Y}}\mathtt{STR}_1 \otimes_{\mathrm{S}} \mathtt{STR}' \otimes_{\mathrm{S}} \mathtt{STR}_2 \multimap P\{\mathtt{STR}'/\mathtt{str}\}) \circ_{\mathrm{TXT}} \\ (\text{“\#text”}_{\mathtt{Y}}\mathtt{STR}_1 \otimes_{\mathrm{S}} \mathtt{STR}' \otimes_{\mathrm{S}} \mathtt{STR}_2 \wedge (\mathtt{Offset} \doteq \mathrm{len}(\mathtt{STR}_1)) \\ \wedge\, (\mathtt{Count} \doteq \mathrm{len}(\mathtt{STR}') \wedge \mathtt{node} \doteq \mathtt{Y}))\end{array}\right\}$$

$\mathtt{str}{:=}\mathtt{substringData}(\mathtt{node},\mathtt{Offset},\mathtt{Count})$

$$\left\{\begin{array}{l}(\text{“\#text”}_{\mathtt{Y}}\mathtt{STR}_1 \otimes_{\mathrm{S}} \mathtt{STR}' \otimes_{\mathrm{S}} \mathtt{STR}_2 \multimap P\{\mathtt{STR}'/\mathtt{str}\}) \circ_{\mathrm{TXT}} \\ (\text{“\#text”}_{\mathtt{Y}}\mathtt{STR}_1 \otimes_{\mathrm{S}} \mathtt{STR}' \otimes_{\mathrm{S}} \mathtt{STR}_2 \wedge (\mathtt{str} \doteq \mathtt{STR}'))\end{array}\right\}$$

CONS

$$\left\{\begin{array}{l}\Diamond_{\mathrm{TXT}\to\mathrm{D}}(\text{“\#text”}_{\mathtt{node}}\mathtt{STR}_1 \otimes_{\mathrm{S}} \mathtt{STR}' \otimes_{\mathrm{S}} \mathtt{STR}_2 \wedge P\{\mathtt{STR}'/\mathtt{str}\}) \\ \wedge\, (\mathtt{Offset} \doteq \mathrm{len}(\mathtt{STR}_1)) \\ \wedge\, (\mathtt{Count} \doteq \mathrm{len}(\mathtt{STR}'))\end{array}\right\}$$

$\mathtt{str}{:=}\mathtt{substringData}(\mathtt{node},\mathtt{Offset},\mathtt{Count})$

$$\left\{P\{\mathtt{STR}'/\mathtt{str}\} \wedge (\mathtt{str} \doteq \mathtt{STR}')\right\}$$

CONS/ELIM

$$\left\{\begin{array}{l}\exists \mathtt{STR}_1, \mathtt{STR}', \mathtt{STR}_2. \\ \Diamond_{\mathrm{TXT}\to\mathrm{D}}(\text{“\#text”}_{\mathtt{node}}\mathtt{STR}_1 \otimes_{\mathrm{S}} \mathtt{STR}' \otimes_{\mathrm{S}} \mathtt{STR}_2 \wedge P\{\mathtt{STR}'/\mathtt{str}\}) \\ \wedge\, (\mathtt{Offset} \doteq \mathrm{len}(\mathtt{STR}_1)) \\ \wedge\, (\mathtt{Count} \doteq \mathrm{len}(\mathtt{STR}'))\end{array}\right\}$$

$\mathtt{str}{:=}\mathtt{substringData}(\mathtt{node},\mathtt{Offset},\mathtt{Count})$

$$\left\{P\right\}$$

where $\mathrm{D} \in \{\mathrm{ELE}, \mathrm{TXT}, \mathrm{F}, \mathrm{G}\}$

$$\left\{\begin{array}{l}\text{“\#text”}_{\mathtt{Y}}\mathtt{STR}_1 \otimes_{\mathrm{S}} \mathtt{STR}' \wedge (\mathtt{Offset} \doteq \mathrm{len}(\mathtt{STR}_1)) \\ \wedge\, (\mathtt{Count} \cdot > \mathrm{len}(\mathtt{STR}')) \wedge \mathtt{node} \doteq \mathtt{Y}\end{array}\right\}$$
$$\mathtt{str} := \mathtt{substringData}(\mathtt{node}, \mathtt{Offset}, \mathtt{Count})$$
$$\left\{\text{“\#text”}_{\mathtt{Y}}\mathtt{STR}_1 \otimes_{\mathrm{S}} \mathtt{STR}' \wedge (\mathtt{str} \doteq \mathtt{STR}')\right\}$$

FRAME

$$\left\{\begin{array}{l}(\text{“\#text”}_{\mathtt{Y}}\mathtt{STR}_1 \otimes_{\mathrm{S}} \mathtt{STR}' \multimap P\{\mathtt{STR}'/\mathtt{str}\}) \circ_{\mathrm{TXT}} \\ (\text{“\#text”}_{\mathtt{Y}}\mathtt{STR}_1 \otimes_{\mathrm{S}} \mathtt{STR}' \wedge (\mathtt{Offset} \doteq \mathrm{len}(\mathtt{STR}_1)) \\ \wedge\, (\mathtt{Count} \cdot > \mathrm{len}(\mathtt{STR}')) \wedge \mathtt{node} \doteq \mathtt{Y})\end{array}\right\}$$
$$\mathtt{str} := \mathtt{substringData}(\mathtt{node}, \mathtt{Offset}, \mathtt{Count})$$
$$\left\{\begin{array}{l}(\text{“\#text”}_{\mathtt{Y}}\mathtt{STR}_1 \otimes_{\mathrm{S}} \mathtt{STR}' \multimap P\{\mathtt{STR}'/\mathtt{str}\}) \circ_{\mathrm{TXT}} \\ (\text{“\#text”}_{\mathtt{Y}}\mathtt{STR}_1 \otimes_{\mathrm{S}} \mathtt{STR}' \wedge (\mathtt{str} \doteq \mathtt{STR}'))\end{array}\right\}$$

CONS

$$\left\{\begin{array}{l}\Diamond_{\mathrm{TXT}\to\mathrm{D}}(\text{“\#text”}_{\mathtt{node}}\mathtt{STR}_1 \otimes_{\mathrm{S}} \mathtt{STR}') \wedge P\{\mathtt{STR}'/\mathtt{str}\} \\ \wedge\, (\mathtt{Offset} \doteq \mathrm{len}(\mathtt{STR}_1)) \wedge (\mathtt{Count} \cdot > \mathrm{len}(\mathtt{STR}'))\end{array}\right\}$$
$$\mathtt{str} := \mathtt{substringData}(\mathtt{node}, \mathtt{Offset}, \mathtt{Count})$$
$$\left\{P\{\mathtt{STR}'/\mathtt{str}\} \wedge (\mathtt{str} \doteq \mathtt{STR}')\right\}$$

CONS/ELIM

$$\left\{\begin{array}{l}\exists \mathtt{STR}_1, \mathtt{STR}'. \\ \Diamond_{\mathrm{TXT}\to\mathrm{D}}(\text{“\#text”}_{\mathtt{node}}\mathtt{STR}_1 \otimes_{\mathrm{S}} \mathtt{STR}') \wedge P\{\mathtt{STR}'/\mathtt{str}\} \\ \wedge\, (\mathtt{Offset} \doteq \mathrm{len}(\mathtt{STR}_1)) \wedge (\mathtt{Count} \cdot > \mathrm{len}(\mathtt{STR}'))\end{array}\right\}$$
$$\mathtt{str} := \mathtt{substringData}(\mathtt{node}, \mathtt{Offset}, \mathtt{Count})$$
$$\left\{P\right\}$$

where $\mathrm{D} \in \{\mathrm{ELE}, \mathrm{TXT}, \mathrm{F}, \mathrm{G}\}$

A.2.9. appendData

$$\frac{\begin{array}{l}\left\{\text{“\#text”}_{\texttt{node}}\texttt{STR} \wedge \texttt{Arg} \in \mathrm{S}\right\} \\ \texttt{appendData}(\texttt{node}, \texttt{Arg}) \\ \left\{\text{“\#text”}_{\texttt{node}}\texttt{STR} \otimes_{\mathrm{S}} \texttt{Arg}\right\}\end{array}}{\begin{array}{l}\left\{\begin{array}{l}(\text{“\#text”}_{\texttt{node}}\texttt{STR} \otimes_{\mathrm{S}} \texttt{Arg} \multimap P) \circ_{\mathrm{TXT}} \\ (\text{“\#text”}_{\texttt{node}}\texttt{STR} \wedge \texttt{Arg} \in \mathrm{S})\end{array}\right\} \\ \texttt{appendData}(\texttt{node}, \texttt{Arg}) \\ \left\{\begin{array}{l}(\text{“\#text”}_{\texttt{node}}\texttt{STR} \otimes_{\mathrm{S}} \texttt{Arg} \multimap P) \circ_{\mathrm{TXT}} \\ (\text{“\#text”}_{\texttt{node}}\texttt{STR} \otimes_{\mathrm{S}} \texttt{Arg})\end{array}\right\}\end{array}}\ \textsc{Frame}$$

$$\frac{\begin{array}{l}\left\{\begin{array}{l}(\text{“\#text”}_{\texttt{node}}\texttt{STR} \otimes_{\mathrm{S}} \texttt{Arg} \multimap P) \circ_{\mathrm{TXT}} \\ (\text{“\#text”}_{\texttt{node}}\texttt{STR} \wedge \texttt{Arg} \in \mathrm{S})\end{array}\right\} \\ \texttt{appendData}(\texttt{node}, \texttt{Arg}) \\ \left\{P\right\}\end{array}}{\begin{array}{l}\left\{\begin{array}{l}\exists \texttt{STR}. \\ (\text{“\#text”}_{\texttt{node}}\texttt{STR} \otimes_{\mathrm{S}} \texttt{Arg} \multimap P) \circ_{\mathrm{TXT}} \\ (\text{“\#text”}_{\texttt{node}}\texttt{STR} \wedge \texttt{Arg} \in \mathrm{S})\end{array}\right\} \\ \texttt{appendData}(\texttt{node}, \texttt{Arg}) \\ \left\{P\right\}\end{array}}\ \textsc{Cons}$$

$$\textsc{Elim}$$

A.2.10. deleteData

$$\left\{\begin{array}{l}\text{“\#text”}_{\texttt{node}}\texttt{STR}_1 \otimes_{\mathrm{S}} \texttt{STR} \otimes_{\mathrm{S}} \texttt{STR}_2 \wedge (\texttt{Offset} \doteq \mathrm{len}(\texttt{STR}_1)) \\ \wedge\, (\texttt{Count} \doteq \mathrm{len}(\texttt{STR}))\end{array}\right\}$$
$$\texttt{deleteData}(\texttt{node}, \texttt{Offset}, \texttt{Count})$$
$$\left\{\text{“\#text”}_{\texttt{node}}\texttt{STR}_1 \otimes_{\mathrm{S}} \texttt{STR}_2\right\}$$

FRAME

$$\left\{\begin{array}{l}(\text{“\#text”}_{\texttt{node}}\texttt{STR}_1 \otimes_{\mathrm{S}} \texttt{STR}_2 \multimap P) \circ_{\mathrm{TXT}} \\ (\text{“\#text”}_{\texttt{node}}\texttt{STR}_1 \otimes_{\mathrm{S}} \texttt{STR} \otimes_{\mathrm{S}} \texttt{STR}_2 \wedge (\texttt{Offset} \doteq \mathrm{len}(\texttt{STR}_1)) \\ \wedge\, (\texttt{Count} \doteq \mathrm{len}(\texttt{STR})))\end{array}\right\}$$
$$\texttt{deleteData}(\texttt{node}, \texttt{Offset}, \texttt{Count})$$
$$\left\{\begin{array}{l}(\text{“\#text”}_{\texttt{node}}\texttt{STR}_1 \otimes_{\mathrm{S}} \texttt{STR}_2 \multimap P) \circ_{\mathrm{TXT}} \\ (\text{“\#text”}_{\texttt{node}}\texttt{STR}_1 \otimes_{\mathrm{S}} \texttt{STR}_2)\end{array}\right\}$$

CONS/ELIM

$$\left\{\begin{array}{l}\exists \texttt{STR}_1, \texttt{STR}_2, \texttt{STR}. \\ (\text{“\#text”}_{\texttt{node}}\texttt{STR}_1 \otimes_{\mathrm{S}} \texttt{STR}_2 \multimap P) \circ_{\mathrm{TXT}} \\ (\text{“\#text”}_{\texttt{node}}\texttt{STR}_1 \otimes_{\mathrm{S}} \texttt{STR} \otimes_{\mathrm{S}} \texttt{STR}_2 \wedge (\texttt{Offset} \doteq \mathrm{len}(\texttt{STR}_1)) \\ \wedge\, (\texttt{Count} \doteq \mathrm{len}(\texttt{STR})))\end{array}\right\}$$
$$\texttt{deleteData}(\texttt{node}, \texttt{Offset}, \texttt{Count})$$
$$\left\{P\right\}$$

$$\left\{\begin{array}{l}\text{“\#text”}_{\texttt{node}}\texttt{STR}_1 \otimes_{\mathrm{S}} \texttt{STR} \wedge (\texttt{Offset} \doteq \mathrm{len}(\texttt{STR}_1)) \\ \wedge\, (\texttt{Count} \cdot > \mathrm{len}(\texttt{STR}))\end{array}\right\}$$
$$\texttt{deleteData}(\texttt{node}, \texttt{Offset}, \texttt{Count})$$
$$\left\{\text{“\#text”}_{\texttt{node}}\texttt{STR}_1\right\}$$

FRAME

$$\left\{\begin{array}{l}(\text{“\#text”}_{\texttt{node}}\texttt{STR}_1 \multimap P) \circ_{\mathrm{TXT}} \\ (\text{“\#text”}_{\texttt{node}}\texttt{STR}_1 \otimes_{\mathrm{S}} \texttt{STR} \wedge (\texttt{Offset} \doteq \mathrm{len}(\texttt{STR}_1)) \\ \wedge\, (\texttt{Count} \cdot > \mathrm{len}(\texttt{STR})))\end{array}\right\}$$
$$\texttt{deleteData}(\texttt{node}, \texttt{Offset}, \texttt{Count})$$
$$\left\{\begin{array}{l}(\text{“\#text”}_{\texttt{node}}\texttt{STR}_1 \multimap P) \circ_{\mathrm{TXT}} \\ (\text{“\#text”}_{\texttt{node}}\texttt{STR}_1)\end{array}\right\}$$

CONS/ELIM

$$\left\{\begin{array}{l}\exists \texttt{STR}_1, \texttt{STR}. \\ (\text{“\#text”}_{\texttt{node}}\texttt{STR}_1 \multimap P) \circ_{\mathrm{TXT}} \\ (\text{“\#text”}_{\texttt{node}}\texttt{STR}_1 \otimes_{\mathrm{S}} \texttt{STR} \wedge (\texttt{Offset} \doteq \mathrm{len}(\texttt{STR}_1)) \\ \wedge\, (\texttt{Count} \cdot > \mathrm{len}(\texttt{STR})))\end{array}\right\}$$
$$\texttt{deleteData}(\texttt{node}, \texttt{Offset}, \texttt{Count})$$
$$\left\{P\right\}$$

A.2.11. createTextNode

$$\left\{\varnothing_{\mathrm{G}} \wedge \texttt{node} \doteq \texttt{Y} \wedge \texttt{Str} \in \mathrm{S}\right\}$$
$$\texttt{node} := \texttt{createTextNode}(\texttt{Str})$$
$$\left\{<\text{“text”}_{\texttt{node}}\texttt{Str}\{\texttt{Y}/\texttt{node}\}>_{\mathrm{G}}\right\}$$

FRAME

$$\left\{\begin{array}{l}(\forall \texttt{NODE}. \\ <\text{“text”}_{\texttt{NODE}}\texttt{Str}\{\texttt{Y}/\texttt{node}\}>_{\mathrm{G}} \multimap P\{\texttt{NODE}/\texttt{node}\}) \circ_{\mathrm{G}} \\ \varnothing_{\mathrm{G}} \wedge \texttt{node} \doteq \texttt{Y} \wedge \texttt{Str} \in \mathrm{S}\end{array}\right\}$$
$$\texttt{node} := \texttt{createTextNode}(\texttt{Str})$$
$$\left\{\begin{array}{l}(\forall \texttt{NODE}. \\ <\text{“text”}_{\texttt{NODE}}\texttt{Str}\{\texttt{Y}/\texttt{node}\}>_{\mathrm{G}} \multimap P\{\texttt{NODE}/\texttt{node}\}) \circ_{\mathrm{G}} \\ <\text{“text”}_{\texttt{node}}\texttt{Str}\{\texttt{Y}/\texttt{node}\}>_{\mathrm{G}}\end{array}\right\}$$

CONS/ELIM

$$\left\{\begin{array}{l}\exists \texttt{Y}. \\ (\forall \texttt{NODE}. \\ <\text{“text”}_{\texttt{NODE}}\texttt{Str}\{\texttt{Y}/\texttt{node}\}>_{\mathrm{G}} \multimap P\{\texttt{NODE}/\texttt{node}\}) \circ_{\mathrm{G}} \\ \varnothing_{\mathrm{G}} \wedge \texttt{node} \doteq \texttt{Y} \wedge \texttt{Str} \in \mathrm{S}\end{array}\right\}$$
$$\texttt{node} := \texttt{createTextNode}(\texttt{Str})$$
$$\left\{P\right\}$$

A.2.12. assignment

$$\left\{\varnothing_{\mathrm{D}} \wedge (\texttt{Y} \doteq \texttt{Exp})\right\}$$
$$\texttt{var} := \texttt{Exp}$$
$$\left\{\varnothing_{\mathrm{D}} \wedge (\texttt{var} \doteq \texttt{Y})\right\}$$

FRAME

$$\left\{\begin{array}{l}(\varnothing_{\mathrm{D}} \multimap P\{\texttt{Y}/\texttt{var}\}) \circ_{\mathrm{D}} \\ \varnothing_{\mathrm{D}} \wedge (\texttt{Y} \doteq \texttt{Exp})\end{array}\right\}$$
$$\texttt{var} := \texttt{Exp}$$
$$\left\{\begin{array}{l}(\varnothing_{\mathrm{D}} \multimap P\{\texttt{Y}/\texttt{var}\}) \circ_{\mathrm{D}} \\ (\varnothing_{\mathrm{D}} \wedge (\texttt{var} \doteq \texttt{Y}))\end{array}\right\}$$

CONS

$$\left\{P\{\texttt{Exp}/\texttt{var}\}\right\}$$
$$\texttt{var} := \texttt{Exp}$$
$$\left\{P\right\}$$
where $\mathrm{D} \in \{\mathrm{F}, \mathrm{G}, \mathrm{S}\}$

A.2.13. skip

$$\frac{}{\begin{array}{l}\left\{\varnothing_{\mathrm{D}}\right\}\\ \texttt{skip}\\ \left\{\varnothing_{\mathrm{D}}\right\}\end{array}}\ \textsc{Frame}$$

$$\frac{}{\begin{array}{l}\left\{(\varnothing_{\mathrm{D}} \multimap P) \circ_{\mathrm{D}} \varnothing_{\mathrm{D}}\right\}\\ \texttt{skip}\\ \left\{(\varnothing_{\mathrm{D}} \multimap P) \circ_{\mathrm{D}} \varnothing_{\mathrm{D}}\right\}\end{array}}\ \textsc{Cons}$$

$$\begin{array}{l}\left\{P\right\}\\ \texttt{skip}\\ \left\{P\right\}\end{array}$$

where $\mathrm{D} \in \{\mathrm{F}, \mathrm{G}, \mathrm{S}\}$

B. DOM Core Level 1

This appendix contains additional material pertinent to DOM Core Level 1, as described in Chapters 6 and 7. Recall that Section 6.3 presents the commands essential to the fundamental interfaces of DOM Core Level 1. Section B.1 contains implementations of the remaining commands, written in terms of the commands given in Section 6.3. Further recall that Section 7.8 presents reasoning about an example program that manipulates a student data document. Section B.2 presents the full proof of that program.

B.1. The Remaining Commands

In this section we take each interface specified by [23] in turn, and reproduce the IDL definition given for that interface by [23]. We annotate each object attribute and method in these IDL definitions with a "*" if that object attribute or method has been provided in Section 6.3.2. For every behaviour not explicitly provided in Section 6.3.2 we provide a composite command. In these composite commands, we often make use of constants such as **ELEMENT_NODE** and **DOCUMENT_NODE** to refer to the type numbers of particular node types, in this case, 1 and 9 respectively.

DOMImplementation

This interface provides only one behaviour – the method "hasFeature". This method returns true if this particular implementation has a given feature, and false otherwise. This is necessarily implementation dependant, and so we do not reason about it here.

DocumentFragment

This interface contains no behaviours that are not already covered by the Node interface:

Begin Quote

```
interface DocumentFragment : Node {
};
```

End Quote

Document

The IDL Definition given in [23] for this Interface is:

Begin Quote

```
interface Document : Node {
  readonly attribute  DocumentType         doctype;
  readonly attribute  DOMImplementation    implementation;
  readonly attribute  Element              documentElement;
  Element                createElement*(in DOMString tagName)
                                       raises(DOMException);
  DocumentFragment       createDocumentFragment*();
  Text                   createTextNode*(in DOMString data);
  Comment                createComment*(in DOMString data);
  CDATASection           createCDATASection(in DOMString data)
                                          raises(DOMException);
  ProcessingInstruction  createProcessingInstruction(
                                           in DOMString target,
                                           in DOMString data)
                                           raises(DOMException);
  Attr                   createAttribute*(in DOMString name)
                                       raises(DOMException);
  EntityReference        createEntityReference(in DOMString name)
                                            raises(DOMException);
  NodeList               getElementsByTagName*(in DOMString tagname);
};
```

End Quote

We deal with each attribute and method in turn.

doctype The DocumentType interface is one of the "Extended Interfaces" of DOM Core Level 1, and so is beyond the scope of this work, as explained in the opening paragraphs of this chapter.

implementation As explained above, the DOMImplementation interface is entirely implementation dependant, and so we do not reason about it.

documentElement Recall that Document nodes have zero or one Element node children, and may have any number of Comment node children. The "documentElement" attribute is a read-only object attribute, which provides access to the Element child of a Document node. We provide a composite getter command which returns the Element child of a Document, or **null** if the Document has no element child. If this command is called on a node of any other type, it faults.

```
n:=getDocumentElement(doc) ≙
    local tp, kids, i, guard, currentNode, tp :
        tp:=getNodeType(doc) ;
        if tp = DOCUMENT_NODE then
            kids:=getChildNodes(doc) ;
            i:=0 ;
            n:=null ;
            guard:=true ;
            while guard do
                currentNode:=item(kids, i) ;
                if currentNode = null then
                    guard:=false
                else
                    tp:=getNodeType(currentNode) ;
                    if tp = ELEMENT_NODE then
                        n:=currentNode ;
                        guard:=false
                    else
                        skip
                    fi
                fi ;
                i:=i + 1
            od
        else
            fault
        fi
    endloc
```

createCDATASection, createProcessingInstruction and createEntityReference Since each of these methods exist to create objects from the Extended Interfaces, they are beyond the scope of this work.

Node

The IDL definition of this interface is:

Begin Quote

```
interface Node {
  // NodeType
  const unsigned short      ELEMENT_NODE       = 1;
  const unsigned short      ATTRIBUTE_NODE     = 2;
  const unsigned short      TEXT_NODE          = 3;
  const unsigned short      CDATA_SECTION_NODE = 4;
  const unsigned short      ENTITY_REFERENCE_NODE = 5;
  const unsigned short      ENTITY_NODE        = 6;
  const unsigned short      PROCESSING_INSTRUCTION_NODE = 7;
  const unsigned short      COMMENT_NODE       = 8;
  const unsigned short      DOCUMENT_NODE      = 9;
  const unsigned short      DOCUMENT_TYPE_NODE = 10;
  const unsigned short      DOCUMENT_FRAGMENT_NODE = 11;
  const unsigned short      NOTATION_NODE      = 12;

  readonly attribute  DOMString       nodeName*;
           attribute  DOMString       nodeValue;
                                      // raises(DOMException) on setting
                                      // raises(DOMException) on retrieval
  readonly attribute  unsigned short  nodeType*;
  readonly attribute  Node            parentNode*;
  readonly attribute  NodeList        childNodes*;
  readonly attribute  Node            firstChild;
  readonly attribute  Node            lastChild;
  readonly attribute  Node            previousSibling;
  readonly attribute  Node            nextSibling;
  readonly attribute  NamedNodeMap    attributes*;
  readonly attribute  Document        ownerDocument*;
  Node                      insertBefore(in Node newChild,
                                         in Node refChild)
                                         raises(DOMException);
  Node                      replaceChild(in Node newChild,
                                         in Node oldChild)
                                         raises(DOMException);
  Node                      removeChild*(in Node oldChild)
                                        raises(DOMException);
  Node                      appendChild*(in Node newChild)
                                        raises(DOMException);
  boolean                   hasChildNodes();
  Node                      cloneNode(in boolean deep);
};
```

End Quote

We represent the const unsigned short values “ELEMENT_NODE”, “ATTRIBUTE_NODE” and so on with the constants **ELEMENT_NODE**, **ATTRIBUTE_NODE** and so on.

nodeValue In Section 6.3.2 we provided helper commands, which are equivalent to the getter and setter commands `getNodeValue` and `setNodeValue` in the non-Attr cases. Here we provide the full getter and setter commands, which make use of those helpers, and also accurately handle the interesting Attr cases.

The Attr cases are interesting because the object attribute named "value" of an Attr node is only one of two representations of the value of the XML attribute's that the Attr object represents. The other representation is the list of TextNodes that are children of the Attr. This state of affairs is suggested by the following quotes from [23]:

Begin Quote

```
Note that the nodeValue attribute on the Attr instance can also
be used to retrieve the string version of the attribute's value(s).

In XML, where the value of an attribute can contain entity
references, the child nodes of the Attr node provide a
representation in which entity references are not expanded.
These child nodes may be either Text or EntityReference nodes.
```

End Quote

Begin Quote

```
Attributes

...

    value
        On retrieval, the value of the attribute is returned as a
        string. Character and general entity references are replaced
        with their values.

        On setting, this creates a Text node with the unparsed
        contents of the string.
```

End Quote

It seems clear that the "value" of an Attr node must be accessible using *both* the object-attribute "nodeValue" *and* by reading the "nodeValue" object-attributes of all the Text node children of the Attr node. These two values must remain consistent at all times.

Since the object attribute representation can be calculated from the TextNode representation but not vice-versa, we choose the TextNode representation as the canonical form, and calculate the object attribute representation on the fly, with the following procedures:

```
v:=getNodeValue(n) ≜
    local tp, kids, i, guard, currentNode, s :
        tp:=getNodeType(n) ;
        if tp = ATTRIBUTE_NODE then
            v:="" ;
            kids:=getChildNodes(n) ;
            i:=0 ;
            guard:=true ;
            while guard do
                currentNode:=item(kids, i) ;
                if currentNode = null then
                    guard:=false
                else
                    s:=getNodeValueHelper(currentNode) ;
                    v:=v ⊗S s
                fi ;
                i:=i + 1
            od
        else
            v := getNodeValueHelper(n)
        fi
    endloc
```

```
setNodeValue(n, v) ≜
    local tp, kids, guard, currentNode, owner, newContent :
        tp:=getNodeType(n) ;
        if tp = ATTRIBUTE_NODE then
            kids:=getChildNodes(n) ;
            guard:=true ;
            while guard do
                currentNode:=item(kids, 0) ;
                if currentNode = null then
                    guard:=false
                else
                    removeChild(n, currentNode)
                fi
            od ;
            owner:=getOwnerDocument(n) ;
            newContent:=createTextNode(owner, v) ;
            appendChild(n, newContent) ;
        else
            setNodeValueHelper(n, v)
        fi
    endloc
```

firstChild and lastChild These read-only object attributes are convenient ways to access the first and last child of a node. We provide the following composite getter commands.

```
child:=getFirstChild(n) ≜
    local kids :
        kids:=getChildNodes(n) ;
        child:=item(kids, 0)
    endloc
```

```
child:=getLastChild(n) ≜
    local kids, i, guard, currentNode :
        kids:=getChildNodes(n) ;
        i:=0 ;
        guard:=false ;
        child:=null ;
        while guard do
            currentNode:=item(kids, i) ;
            if currentNode = null then
                guard:=false
            else
                child:=currentNode
            fi ;
            i:=i + 1
        od
    endloc
```

previousSibling and nextSibling These read-only object attributes are convenient ways to access a node's immediate siblings. We provide the following composite getter commands.

```
sibling:=getPreviousSibling(n) ≜
    local parent, kids, i, guard, currentNode :
        parent:=getParentNode(n) ;
        if parent = null then
            sibling:=null
        else
            kids:=getChildNodes(parent) ;
            i:=0 ;
            guard:=true ;
            sibling:=null ;
            while guard do
                currentNode:=item(kids, i) ;
                if (currentNode = n) then
                    guard:=false
                else
                    sibling:=currentNode
                fi ;
                i:=i + 1
            od
        fi
    endloc
```

```
sibling:=getNextSibling(n) ≜
    local parent, kids, i, guard, currentNode :
        parent:=getParentNode(n) ;
        if parent = null then
            sibling:=null
        else
            kids:=getChildNodes(parent) ;
            i:=0 ;
            guard:=true ;
            while guard do
                currentNode:=item(kids, i) ;
                if (currentNode = n) then
                    guard:=false ;
                    sibling:=item(kids, i + 1)
                else
                    skip
                fi ;
                i:=i + 1
            od
        fi
    endloc
```

insertBefore As [23] says, this command:

Begin Quote

Inserts the node newChild before the existing child node refChild. If refChild is null, insert newChild at the end of the list of children.

End Quote

```
n:=insertBefore(parent, newChild, refChild) ≙
    local kids, i, currentNode, foundNewChild, firstNewChild, tp, fragKids :
        if refChild = null then
            n:=appendChild(parent, newChild)
        else
            kids:=getChildNodes(parent) ;
            i:=0 ;
            currentNode:=item(kids, i) ;
            foundNewChild:=false ;
            while currentNode ≠ null ∧ currentNode ≠ refChild do
                if currentNode = newChild then
                    foundNewChild:=true
                fi ;
                i:=i + 1 ;
                currentNode:=item(kids, i)
            od ;
            if currentNode = null then
                fault
            fi ;
            firstNewChild:=newChild ;
            tp:=getNodeType(firstNewChild) ;
            if tp = DOCUMENT_FRAGMENT then
                fragKids:=getChildNodes(firstNewChild) ;
                firstNewChild:=item(fragKids, 0) ;
                if firstNewChild = null then
                    firstNewChild:=refChild
                fi
            fi ;
            n:=appendChild(parent, newChild) ;
            if foundNewChild then
                i:=i − 1
            fi ;
            n:=appendChild(parent, refChild) ;
            currentNode:=item(kids) ;
            while currentNode ≠ firstNewChild do
                n:=appendChild(parent, currentNode) ;
                currentNode:=item(kids, i)
            od ;
            n:=newChild
        fi
    endloc
```

replaceChild According to [23], this command:

Begin Quote

> Replaces the child node oldChild with newChild in the list of children, and returns the oldChild node. If the newChild is already in the tree, it is first removed.

End Quote

```
n:=replaceChild(parent, newChild, oldChild) ≜
    n:=insertBefore(parent, newChild, oldChild) ;
    n:=removeChild(parent, oldChild)
```

hasChildNodes [23] describes this method as "a convenience method to allow easy determination of whether a node has any children" which returns "true if the node has any children, false if the node has no children".

```
v:=hasChildNodes(node) ≜
    local kids, n :
        kids:=getChildNodes(node) ;
        n:=item(kids, 0) ;
        if n = null then
            v:=false
        else
            v:=true
        fi
    endloc
```

cloneNode According to [23], this command:

Begin Quote

> Returns a duplicate of this node, i.e., serves as a generic copy constructor for nodes. The duplicate node has no parent (parentNode returns **null**.).

> Cloning an Element copies all attributes and their values, including those generated by the XML processor to represent defaulted attributes, but this method does not copy any text it contains unless it is a deep clone, since the text is contained in a child Text node. Cloning any other type of node simply returns a copy of this node.

End Quote

The command returns "The duplicate node.", and takes the parameter "`deep`": "If true, recursively clone the subtree under the specified node; if false, clone only the node itself (and its attributes, if it is an Element)."

```
newNode:=cloneNode(node, deep) ≙
    local doc, tp, name, ats, i, oldat, newat, valstr, valnode, newats, kids, kid, newKid :
        doc:=getOwnerDocument(node) ; tp:=getNodeType(node) ; name:=getNodeName(node) ;
        if tp = 1 then
            // node is an Element node
            newNode:=createElement(doc, name) ; newats:=getAttributes(newNode) ;
            ats:=getAttributes(node) ; i:=0 ; oldat:=item(ats, i) ;
            while oldat ≠ null do
                name:=getNodeName(oldat) ; newat:=createAttribute(doc, name) ;
                valstr:=getNodeValue(oldat) ; valnode:=createTextNode(doc, valstr) ;
                appendChild(newat, valnode) ; setNamedItem(newats, newat) ;
                i:=i + 1 ; oldat:=item(ats, i)
            od
        else
            if tp = 2 then
                // node is an Attribute node
                newNode:=createAttribute(doc, name) ; valstr:=getNodeValue(node) ;
                valnode:=createTextNode(doc, valstr) ; appendChild(newNode, valnode)
            else
                if tp = 3 then
                    // node is a Text node
                    valstr:=getNodeValue(node) ; newNode:=createTextNode(doc, valstr)
                else
                    if tp = 8 then
                        // node is a Comment node
                        valstr:=getNodeValue(node) ; newNode:=createComment(doc, valstr)
                    else
                        if tp = 9 then
                            // node is a Document node
                            newNode:=createDocument()
                        else
                            // node is a Document Fragment node
                            newNode:=createDocumentFragment(doc)
                        fi
                    fi
                fi
            fi
        fi ;
        if deep then
            kids:=getChildNodes(node) ; i:=0 ; kid:=item(kids, i) ;
            while kid ≠ null do
                newKid:=cloneNode(kid, true) ; appendChild(newNode, newKid) ;
                i:=i + 1 ; kid:=item(kids, i)
            od
        fi
    endloc
```

Note that [23] specifies that "Cloning an Element copies all attributes and their values". This is subtly different from copying the attributes and *all their children*, even though the concatenation of the values of the children of an attribute does represent the value of the attribute. We copy the value as a single string which is used to instantiate a new TextNode in the cloned attr node. We do not copy the text nodes which happen to correspond to the value of the attribute. The specification is admittedly unclear as to which of these two behaviours is preferred. We have chosen the one which we believe is preferable, but it is easy to see how the implementation of cloneNode could be changed to provide the alternate behaviour.

Note also that attempting a deep clone of a document node which has any children will fault. This is because of an interaction between cloneNode and and the object attribute "ownerDocument" of Node objects. When deep cloning a Document node, [23] says cloneNode should return "a duplicate of this node", which is to say, another distinct Document node, and should "recursively clone the subtree". When we clone the first child node of the Document node in question, we take heed of the child node's ownerDocument, which is "the Document object used to create new nodes". The new clone of the child node is therefore owned by the same Document node as the old child node, and therefore cannot be made a child of the newly cloned Document node. When our cloneNode procedure attempts to add the new child to the new Document node, it faults.

The crux of the matter here is that the clone of a child of a Document node cannot both be a faithful clone of the original child (and therefore owned by the original Document), and also be owned by the clone of the Document. This interpretation of the specification may be controversial in some quarters, so we further justify our decision here by noting that our chosen interpretation is the more conservative one in the context of a fault-avoiding reasoning. If a program does not fault under our interpretation of the specification, it certainly will not fault or behave unpredictably under the competing (non-faulting) interpretation. Finally, this command was discussed on the W3C DOM mailing list in 1998, and can be read in the archive here: [48].

NodeList

The IDL definition of this interface is:

Begin Quote

```
interface NodeList {
  Node                    item*(in unsigned long index);
  readonly attribute  unsigned long        length;
};
```

End Quote

length The readonly object attribute "length" is described in [23] as representing "The number of nodes in the list. The range of valid child node indices is 0 to length-1 inclusive.".

We provide the getter command `getLength`:

```
length:=getLength(list) ≙
    local n :
        length:=0 ;
        n:=item(list, length) ;
        while n ≠ null do
            length:=length + 1 ;
            n:=item(list, length)
        od
    endloc
```

Note that, since the only DOM command used by this procedure is "`item`", this command will also function perfectly well as a getter command for the NamedNodeMap object attribute "length", mentioned below.

NamedNodeMap

The IDL definition of this interface is:

Begin Quote

```
interface NamedNodeMap {
  Node                        getNamedItem(in DOMString name);
  Node                        setNamedItem*(in Node arg)
                                          raises(DOMException);
  Node                        removeNamedItem*(in DOMString name)
                                            raises(DOMException);
  Node                        item*(in unsigned long index);
  readonly attribute  unsigned long        length;
};
```

End Quote

getNamedItem According to [23] this method "Retrieves a node specified by name.". It takes the parameter `name` which represents the "Name of a node to retrieve." and returns "A Node (of any type) with the specified name, or null if the specified name did not identify any node in the map.".

```
n:=getNamedItem(list, name) ≙
    local i, currentName :
        i:=0 ;
        n:=item(list, i) ;
        if n ≠ null then
            currentName:=getNodeName(n)
        fi ;
        while currentName ≠ name ∧ n ≠ null do
            i:=i + 1 ;
            n:=item(list, i) ;
            if n ≠ null then
                currentName:=getNodeName(n)
            fi
        od
    endloc
```

length This readonly object attribute demonstrates behaviour identical to that of the "item" attribute of the NodeList interface. Our getter command `getLength` given earlier in this section requires only one DOM command in order to function: "`item`", which also operates on NamedNodeMaps. That

implementation of therefore functions perfectly well as both a NodeList command and a NamedNodeMap command.

Character Data

The IDL definition of this interface is:

Begin Quote

```
interface CharacterData : Node {
           attribute  DOMString            data;
                                 // raises(DOMException) on setting
                                 // raises(DOMException) on retrieval
  readonly attribute  unsigned long        length;
  DOMString                 substringData*(in unsigned long offset,
                                           in unsigned long count)
                                           raises(DOMException);
  void                      appendData*(in DOMString arg)
                                       raises(DOMException);
  void                      insertData(in unsigned long offset,
                                       in DOMString arg)
                                       raises(DOMException);
  void                      deleteData*(in unsigned long offset,
                                       in unsigned long count)
                                       raises(DOMException);
  void                      replaceData(in unsigned long offset,
                                        in unsigned long count,
                                        in DOMString arg)
                                        raises(DOMException);
};
```

End Quote

data The object attribute "data" of the CharacterData interface is functionally equivalent to the "value" object attribute of the Node interface.

```
val:=getData(n) ≙
    local tp :
        tp:=getNodeType(n) ;
        if tp = 3 ∨ tp = 8 then
            val:=getNodeValue(n)
        else
            fault
        fi
    endloc
```

```
setData(n, newval) ≜
    local tp :
        tp:=getNodeType(n) ;
        if tp = 3 ∨ tp = 8 then
            setNodeValue(n, newval)
        else
            fault
        fi
    endloc
```

length This readonly object attribute represents "The number of characters that are available through data and the substringData method below. This may have the value zero, i.e., CharacterData nodes may be empty.". We provide the getter command "`getTextLength`", which we have named in order to avoid a name clash with the "`getLength`" which operates on NodeLists and NamedNodeMaps.

```
l:=getTextLength(n) ≜
    local tp, val :
        tp:=getNodeType(n) ;
        if tp = 3 ∨ tp = 8 then
            val:=getNodeValue(n) ;
            l:= len(val)
        else
            fault
        fi
    endloc
```

insertData According to [23], this method will "Insert a string at the specified character offset." taking arguments `offset` "The character offset at which to insert." and `arg` "The DOMString to insert.".

```
insertData(n, offset, arg) ≜
    local len, s :
        len:=getTextLength() ;
        s:=substringData(offset, len) ;
        deleteData(n, offset, len) ;
        appendData(n, arg) ;
        appendData(n, s)
    endloc
```

replaceData According to [23], this method will "Replace the characters starting at the specified character offset with the specified string." taking arguments `offset` "The offset from which to start replacing", `count` "The number of characters to replace" and `arg` "The DOMString with which the range must be replaced".

```
replaceData(n, offset, count, arg) ≜
    deleteData(n, offset, count) ;
    insertData(n, offset, arg)
```

Attr

The IDL definition of this interface is:

Begin Quote

```
interface Attr : Node {
  readonly attribute  DOMString           name;
  readonly attribute  boolean             specified*;
           attribute  DOMString           value;
};
```

End Quote

name The readonly object attribute "name" of the Attr interface is functionally equivalent to the "nodeName" object attribute of the Node interface.

```
nm:=getName(n) ≜
    local tp :
        tp:=getNodeType(n) ;
        if tp = 2 then
            nm:=getNodeName(n)
        else
            fault
        fi
    endloc
```

value The object attribute "value" of the Attr interface represents the same data as the concatenation of all that Attr's children. As with other object attributes we provide getter and setter commands.

[23] says: "On retrieval, the value of the attribute is returned as a string."

```
val := getValue(n) ≜
    local tp, kids, i, len, kid, s :
        tp := getNodeType(n) ;
        if tp ≠ 2 then
            fault
        fi ;
        val := "" ;
        kids := getChildNodes(n) ;
        i := 0 ;
        len := getLength(kids) ;
        while i < len do
            kid := item(kids, i) ;
            s := getNodeValue(kid) ;
            val := val ⊗S s ;
            i := i + 1
        od
    endloc
```

[23] also says: "On setting, this creates a Text node with the unparsed contents of the string".

```
setValue(n, val) ≜
    local tp, kids, i, currentNode, doc, newNode :
        tp := getNodeType(n) ;
        if tp ≠ 2 then
            fault
        fi ;
        kids := getChildNodes(n) ;
        i := getLength(kids) ;
        while i > 0 do
            currentNode := item(kids, 0) ;
            removeChild(n, currentNode) ;
            i := i − 1
        od ;
        doc := getOwnerDocument(n) ;
        newNode := createTextNode(doc, val) ;
        appendChild(n, newNode)
    endloc
```

Element

The IDL definition of this interface is:

Begin Quote

```
interface Element : Node {
  readonly attribute  DOMString           tagName;
  DOMString                getAttribute(in DOMString name);
  void                     setAttribute(in DOMString name,
                                        in DOMString value)
                                        raises(DOMException);
  void                     removeAttribute(in DOMString name)
                                          raises(DOMException);
  Attr                     getAttributeNode(in DOMString name);
  Attr                     setAttributeNode(in Attr newAttr)
                                           raises(DOMException);
  Attr                     removeAttributeNode(in Attr oldAttr)
                                              raises(DOMException);
  NodeList                 getElementsByTagName*(in DOMString name);
  void                     normalize();
};
```

End Quote

tagName This readonly object attribute is functionally equivalent to the "nodeName" object attribute of the Node interface.

```
nm:=getTagName(n) ≜
    local tp :
        tp:=getNodeType(n) ;
        if tp = 1 then
            nm:=getNodeName(n)
        else
            fault
        fi
    endloc
```

getAttribute This method "Retrieves an attribute value by name", returning "The Attr value as a string, or the empty string if that attribute does not have a specified or default value."

```
str := getAttribute(n, name) ≜
    local ats, at :
        ats := getAttributes(n) ;
        at := getNamedItem(ats, name) ;
        if at = null then
            str = ""
        else
            str := getNodeValue(at)
        fi
    endloc
```

setAttribute This method "Adds a new attribute. If an attribute with that name is already present in the element, its value is changed to be that of the value parameter."

```
setAttribute(n, name, value) ≜
    local ats, at, doc :
        ats := getAttributes(n) ;
        at := getNamedItem(ats, name) ;
        if at = null then
            doc := getOwnerDocument(n) ;
            at := createAttribute(doc, name) ;
            setNodeValue(at, value) ;
            setNamedItem(ats, at)
        else
            setNodeValue(at, value)
        fi
    endloc
```

removeAttribute This method "Removes an attribute by name. If the removed attribute has a default value it is immediately replaced". Recall that the immediate replacement if the attribute has a default name is handled by our `removeNamedItem` command.

```
removeAttribute(n, name) ≜
    local ats :
        ats := getAttributes(n) ;
        removeNamedItem(ats, name)
    endloc
```

getAttributeNode This method "Retrieves an Attr node by name."

```
at := getAttributeNode(n, name) ≜
    local ats :
        ats := getAttributes(n) ;
        at := getNamedItem(ats, name)
    endloc
```

setAttributeNode This method "Adds a new attribute. If an attribute with that name is already present in the element, it is replaced by the new one." ... "If the newAttr attribute replaces an existing attribute with the

same name, the previously existing Attr node is returned, otherwise null is returned."

```
oldAttr := setAttributeNode(n, newAttr) ≜
    local ats :
        ats := getAttributes(n) ;
        oldAttr := setNamedItem(ats, newAttr)
    endloc
```

removeAttributeNode This method "Removes the specified attribute", returning "The Attr node that was removed".

```
at := removeAttributeNode(n, oldAttr) ≜
    local ats, i, name :
        ats := getAttributes(n) ;
        i := 0 ;
        at := item(ats, i) ;
        while at ≠ null ∧ at ≠ oldAttr do
            i := i + 1 ;
            at := item(ats, i)
        od ;
        if at ≠ null then
            name := getNodeName(at) ;
            removeNamedItem(ats, name)
        else
            skip
        fi
    endloc
```

normalize This method "Puts all Text nodes in the full depth of the sub-tree underneath this Element into a "normal" form where only markup (e.g., tags, comments, processing instructions, CDATA sections, and entity references) separates Text nodes, i.e., there are no adjacent Text nodes."

As we indicated in Chapter 1.1, this specification is not complete since it does not specify whether any of the text nodes in the pre-normalized structure should be re-used in the normalized structure. Our experiments with DOM implementations (See Chapter 8 and Appendix C) show that

there is a surprising consensus among web browsers that the first Text node in any consecutive sequence should be re-used. The following procedure follows that example.

```
normalize(n) ≜
    local tp, kids, lastNode, i, currentNode, lasttp, text :
        tp:=getNodeType(n) ;
        if tp = 1 then
            kids := getChildNodes(n) ;
            lastNode := n ;
            i := 0 ;
            currentNode := item(kids, i) ;
            while currentNode ≠ null do
                tp := getNodeType(currentNode) ;
                if tp = 3 then
                    lasttp := getNodeType(lastNode) ;
                    if lasttp = 3 then
                        text := getNodeValue(currentNode) ;
                        appendData(lastNode, text) ;
                        removeChild(n, currentNode) ;
                        i := i − 1
                    else
                        lastNode := currentNode
                    fi
                else
                    lastNode := currentNode ;
                    if tp = 1 then
                        normalize(currentNode)
                    else
                        skip
                    fi
                fi ;
                i := i + 1 ;
                currentNode := item(kids, i)
            od
        else
            fault
        fi
    endloc
```

Text

The Text interface inherits from "CharacterData" interface, which provides most of its functionality. There is one text-specific method though, and that is "splitText". The IDL definition of this interface is:

Begin Quote

```
interface Text : CharacterData {
  Text                        splitText(in unsigned long offset)
                                        raises(DOMException);
};
```

End Quote

splitText This method "Breaks this Text node into two Text nodes at the specified offset, keeping both in the tree as siblings. This node then only contains all the content up to the offset point. And a new Text node, which is inserted as the next sibling of this node, contains all the content at and after the offset point" and returns "The new Text node".

```
newText := splitText(n, offset) ≜
    local tp, length, text, doc, parent :
        tp := getNodeType(n) ;
        if tp = 3 then
            length := getTextLength(n) ;
            text := substringData(n, offset, length) ;
            deleteData(n, offset, length) ;
            doc := getOwnerDocument(n) ;
            newText := createTextNode(doc, text) ;
            parent := getParentNode(n) ;
            insertBefore(parent, newText, n) ;
            insertBefore(parent, n, newText) ;
        else
            fault
        fi
    endloc
```

Comment

The Comment interface inherits from the CharacterData interface, and provides no additional functionality. These nodes represent comments in the XML data, which are handled by DOM in the same way as other character data.

B.2. Proving Schema Preservation of the graduateStudents Procedure

In this section, we present the full proof of the example program given in Section 7.8.

$\{P \wedge S\}$

$\texttt{graduateStudents}(\texttt{doc}, \texttt{currentDate}) \triangleq$

$\texttt{localkids}, \texttt{de}, \texttt{alumni}, \texttt{tags}, \texttt{currentStudentTag},$

$\texttt{currentStudent}, \texttt{finishDateAt}, \texttt{txt}, \texttt{ats}:$

$\texttt{kids} = \texttt{getChildNodes}(\texttt{doc})\ ;$

$\texttt{de} = \texttt{item}(\texttt{kids}, 0)\ ;$

$\texttt{kids} = \texttt{getChildNodes}(\texttt{de})\ ;$

$\texttt{alumniitem}(\texttt{kids}, 1)\ ;$

$\texttt{tags} := \texttt{getElementsByTagName}(\texttt{doc}, \text{“finalYear”})\ ;$

$$
\left\{
\begin{array}{l}
{<}\text{“\#document”}_{\texttt{doc}} \lessdot \varnothing_{\mathrm{EA}} \gtrdot_{\mathbf{9}} [{<}\varnothing_{\mathrm{DNEL}}, < \\
\quad \text{“students”}_{\texttt{de}} \lessdot \varnothing_{\mathrm{EA}} \gtrdot_{\mathbf{1}} [\\
\quad\quad \text{“current”} \lessdot \varnothing_{\mathrm{EA}} \gtrdot_{\mathbf{1}} [\\
\quad\quad\quad \Box_{\otimes}(\text{“student”} \lessdot < \\
\quad\quad\quad\quad \ll \text{“startDate”} \mapsto [\mathrm{true}_{\mathrm{AF}}] \gg \\
\quad\quad\quad {>}_{\mathrm{EA}} \gtrdot_{\mathbf{1}} [\\
\quad\quad\quad\quad {<}\text{“name”} \lessdot \varnothing_{\mathrm{EA}} \gtrdot_{\mathbf{1}} [TXTS]\mathbf{null}{>}_{\mathrm{EF}} \otimes_{\mathrm{EF}} \\
\quad\quad\quad\quad {<}\text{“address”} \lessdot \varnothing_{\mathrm{EA}} \gtrdot_{\mathbf{1}} [TXTS]\mathbf{null}{>}_{\mathrm{EF}} \otimes_{\mathrm{EF}} \\
\quad\quad\quad\quad {<}\text{“subject”} \lessdot \varnothing_{\mathrm{EA}} \gtrdot_{\mathbf{1}} [TXTS]\mathbf{null}{>}_{\mathrm{EF}} \otimes_{\mathrm{EF}} \\
\quad\quad\quad\quad (\varnothing_{\mathrm{EF}} \vee {<}\text{“finalYear”} \lessdot \varnothing_{\mathrm{EA}} \gtrdot_{\mathbf{1}} [\varnothing_{\mathrm{EF}}]\mathbf{null}{>}_{\mathrm{EF}}) \\
\quad\quad\quad]\mathbf{null}, \mathrm{EF}) \\
\quad\quad]\mathbf{null} \\
\quad\quad \text{“alumni”}_{\texttt{alumni}} \lessdot \varnothing_{\mathrm{EA}} \gtrdot_{\mathbf{1}} [\\
\quad\quad\quad \Box_{\otimes}(\text{“student”} \lessdot < \\
\quad\quad\quad\quad \ll \text{“startDate”} \mapsto [\mathrm{true}_{\mathrm{AF}}] \gg \otimes_{\mathrm{AF}} \\
\quad\quad\quad\quad \ll \text{“finishDate”} \mapsto [\mathrm{true}_{\mathrm{AF}}] \gg \\
\quad\quad\quad {>}_{\mathrm{EA}} \gtrdot_{\mathbf{1}} [\\
\quad\quad\quad\quad {<}\text{“name”} \lessdot \varnothing_{\mathrm{EA}} \gtrdot_{\mathbf{1}} [TXTS]\mathbf{null}{>}_{\mathrm{EF}} \otimes_{\mathrm{EF}} \\
\quad\quad\quad\quad {<}\text{“address”} \lessdot \varnothing_{\mathrm{EA}} \gtrdot_{\mathbf{1}} [TXTS]\mathbf{null}{>}_{\mathrm{EF}} \otimes_{\mathrm{EF}} \\
\quad\quad\quad\quad {<}\text{“subject”} \lessdot \varnothing_{\mathrm{EA}} \gtrdot_{\mathbf{1}} [TXTS]\mathbf{null}{>}_{\mathrm{EF}} \otimes_{\mathrm{EF}} \\
\quad\quad\quad]\mathbf{null}, \mathrm{EF}) \\
\quad\quad]\mathbf{null} \\
\quad]_{\texttt{kids}}\mathbf{null} \\
{>}_{\mathrm{DE}}, \varnothing_{\mathrm{DNEL}}{>}_{\mathrm{DF}} \wedge \mathrm{flatten}(\text{“finalYear”}, \texttt{FINALS}{:}\mathrm{EF})]\mathbf{null}{>}_{\mathrm{G}} \oplus \\
{<}\text{“finalYear”}^{\texttt{doc}}_{\texttt{tags}}{>}_{\mathrm{G}}
\end{array}
\right\}
$$

$$
\begin{array}{l}
\texttt{currentStudentTag} := \texttt{item}(\texttt{tags}, \mathbf{0}) \ ; \\
\textbf{while } \texttt{currentStudentTag} \neq \textbf{null } \textbf{do} \\
\quad \texttt{currentStudent} := \texttt{getParentNode}(\texttt{currentStudentTag}) \ ; \\
\left\{ \begin{array}{l}
\quad <\text{“\#document”}_{\texttt{doc}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{9}} \, [<\varnothing_{\mathrm{DNEL}}, < \\
\qquad \text{“students”}_{\texttt{de}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{1}} \, [\\
\qquad\quad \text{“current”} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{1}} \, [\\
\qquad\qquad \Box_{\otimes}(\text{“student”} \{\!\!\{ < \\
\qquad\qquad\quad \ll \text{“startDate”} \mapsto [\mathrm{true}_{\mathrm{AF}}] \gg \\
\qquad\qquad >_{\mathrm{EA}} \}\!\!\}_{\mathbf{1}} \, [\\
\qquad\qquad\quad < \text{“name”} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{1}} \, [TXTS]\mathbf{null} >_{\mathrm{EF}} \otimes_{\mathrm{EF}} \\
\qquad\qquad\quad < \text{“address”} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{1}} \, [TXTS]\mathbf{null} >_{\mathrm{EF}} \otimes_{\mathrm{EF}} \\
\qquad\qquad\quad < \text{“subject”} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{1}} \, [TXTS]\mathbf{null} >_{\mathrm{EF}} \otimes_{\mathrm{EF}} \\
\qquad\qquad\quad (\varnothing_{\mathrm{EF}} \vee < \text{“finalYear”} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{1}} \, [\varnothing_{\mathrm{EF}}]\mathbf{null} >_{\mathrm{EF}}) \\
\qquad\qquad]\mathbf{null}, \mathrm{EF}) \otimes_{\mathrm{EF}} \\
\qquad\qquad \text{“student”}_{\texttt{currentStudent}} \{\!\!\{ < \\
\qquad\qquad\quad \ll \text{“startDate”} \mapsto [\mathrm{true}_{\mathrm{AF}}] \gg \\
\qquad\qquad >_{\mathrm{EA}} \}\!\!\}_{\mathbf{1}} \, [\\
\qquad\qquad\quad < \text{“name”} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{1}} \, [TXTS]\mathbf{null} >_{\mathrm{EF}} \otimes_{\mathrm{EF}} \\
\qquad\qquad\quad < \text{“address”} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{1}} \, [TXTS]\mathbf{null} >_{\mathrm{EF}} \otimes_{\mathrm{EF}} \\
\qquad\qquad\quad < \text{“subject”} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{1}} \, [TXTS]\mathbf{null} >_{\mathrm{EF}} \otimes_{\mathrm{EF}} \\
\qquad\qquad\quad < \text{“finalYear”}_{\texttt{currentStudentTag}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{1}} \, [\varnothing_{\mathrm{EF}}]\mathbf{null} >_{\mathrm{EF}} \\
\qquad\qquad]\mathbf{null} \otimes_{\mathrm{EF}} \\
\qquad\qquad \Box_{\otimes}(\text{“student”} \{\!\!\{ < \\
\qquad\qquad\quad \ll \text{“startDate”} \mapsto [\mathrm{true}_{\mathrm{AF}}] \gg \\
\qquad\qquad >_{\mathrm{EA}} \}\!\!\}_{\mathbf{1}} \, [\\
\qquad\qquad\quad < \text{“name”} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{1}} \, [TXTS]\mathbf{null} >_{\mathrm{EF}} \otimes_{\mathrm{EF}} \\
\qquad\qquad\quad < \text{“address”} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{1}} \, [TXTS]\mathbf{null} >_{\mathrm{EF}} \otimes_{\mathrm{EF}} \\
\qquad\qquad\quad < \text{“subject”} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{1}} \, [TXTS]\mathbf{null} >_{\mathrm{EF}} \otimes_{\mathrm{EF}} \\
\qquad\qquad\quad (\varnothing_{\mathrm{EF}} \vee < \text{“finalYear”} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{1}} \, [\varnothing_{\mathrm{EF}}]\mathbf{null} >_{\mathrm{EF}}) \\
\qquad\qquad]\mathbf{null}, \mathrm{EF}) \\
\qquad\quad]\mathbf{null} \\
\qquad\quad \text{“alumni”}_{\texttt{alumni}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{1}} \, [\\
\qquad\qquad \Box_{\otimes}(\text{“student”} \{\!\!\{ < \\
\qquad\qquad\quad \ll \text{“startDate”} \mapsto [\mathrm{true}_{\mathrm{AF}}] \gg \otimes_{\mathrm{AF}} \\
\qquad\qquad\quad \ll \text{“finishDate”} \mapsto [\mathrm{true}_{\mathrm{AF}}] \gg \\
\qquad\qquad >_{\mathrm{EA}} \}\!\!\}_{\mathbf{1}} \, [\\
\qquad\qquad\quad < \text{“name”} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{1}} \, [TXTS]\mathbf{null} >_{\mathrm{EF}} \otimes_{\mathrm{EF}} \\
\qquad\qquad\quad < \text{“address”} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{1}} \, [TXTS]\mathbf{null} >_{\mathrm{EF}} \otimes_{\mathrm{EF}} \\
\qquad\qquad\quad < \text{“subject”} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{1}} \, [TXTS]\mathbf{null} >_{\mathrm{EF}} \otimes_{\mathrm{EF}} \\
\qquad\qquad]\mathbf{null}, \mathrm{EF}) \\
\qquad\quad]\mathbf{null} \\
\qquad]_{\texttt{kids}}\mathbf{null} \\
\quad >_{\mathrm{DE}}, \varnothing_{\mathrm{DNEL}} >_{\mathrm{DF}} \wedge \\
\quad \mathrm{flatten}(\text{“finalYear”}, \\
\quad\quad <\text{“finalYear”}_{\texttt{currentStudentTag}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{1}} \, [\varnothing_{\mathrm{EF}}]\mathbf{null} >_{\mathrm{EF}} \\
\quad\quad \otimes_{\mathrm{EF}} \texttt{FINALS}' : \mathrm{EF} \\
\quad)]\mathbf{null} >_{\mathrm{G}} \oplus < \text{“finalYear”}^{\texttt{doc}}_{\texttt{tags}} >_{\mathrm{G}}
\end{array} \right.
\end{array}
$$

$\texttt{removeChild}(\texttt{currentStudent}, \texttt{currentStudentTag})$;
$\texttt{finishDateAt} := \texttt{createAttribute}(\texttt{doc}, \text{“finishDate”})$;
$\texttt{txt} = \texttt{createTextNode}(\texttt{doc}, \texttt{currentDate})$; $\texttt{appendChild}(\texttt{finishDateAt}, \texttt{txt})$;
$\texttt{ats} := \texttt{getAttributes}(\texttt{currentStudent})$; $\texttt{setNamedItem}(\texttt{ats}, \texttt{finishDateAt})$;
$\texttt{appendChild}(\texttt{alumni}, \texttt{currentStudent})$; $\texttt{currentStudentTag} := \texttt{item}(\texttt{tags}, \mathbf{0})$

$$
\left\{
\begin{array}{l}
< \text{“\#document”}_{\texttt{doc}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{9}} [< \varnothing_{\mathrm{DNEL}}, < \text{“students”}_{\texttt{de}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{1}} [\\
\quad \text{“current”} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{1}} [\\
\quad\quad \Box_{\otimes} (\text{“student”} \{\!\!\{ << \text{“startDate”} \mapsto [\mathrm{true}_{\mathrm{AF}}] \gg >_{\mathrm{EA}} \}\!\!\}_{\mathbf{1}} [\\
\quad\quad\quad < \text{“name”} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{1}} [TXTS] \mathbf{null} >_{\mathrm{EF}} \otimes_{\mathrm{EF}} \\
\quad\quad\quad < \text{“address”} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{1}} [TXTS] \mathbf{null} >_{\mathrm{EF}} \otimes_{\mathrm{EF}} \\
\quad\quad\quad < \text{“subject”} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{1}} [TXTS] \mathbf{null} >_{\mathrm{EF}} \otimes_{\mathrm{EF}} \\
\quad\quad\quad (\varnothing_{\mathrm{EF}} \vee < \text{“finalYear”} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{1}} [\varnothing_{\mathrm{EF}}] \mathbf{null} >_{\mathrm{EF}}) \\
\quad\quad] \mathbf{null}, \mathrm{EF}) \otimes_{\mathrm{EF}} \\
\quad\quad \varnothing_{\mathrm{EF}} \otimes_{\mathrm{EF}} \\
\quad\quad \Box_{\otimes} (\text{“student”} \{\!\!\{ << \text{“startDate”} \mapsto [\mathrm{true}_{\mathrm{AF}}] \gg >_{\mathrm{EA}} \}\!\!\}_{\mathbf{1}} [\\
\quad\quad\quad < \text{“name”} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{1}} [TXTS] \mathbf{null} >_{\mathrm{EF}} \otimes_{\mathrm{EF}} \\
\quad\quad\quad < \text{“address”} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{1}} [TXTS] \mathbf{null} >_{\mathrm{EF}} \otimes_{\mathrm{EF}} \\
\quad\quad\quad < \text{“subject”} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{1}} [TXTS] \mathbf{null} >_{\mathrm{EF}} \otimes_{\mathrm{EF}} \\
\quad\quad\quad (\varnothing_{\mathrm{EF}} \vee < \text{“finalYear”} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{1}} [\varnothing_{\mathrm{EF}}] \mathbf{null} >_{\mathrm{EF}}) \\
\quad\quad] \mathbf{null}, \mathrm{EF}) \\
\quad] \mathbf{null} \\
\quad \text{“alumni”}_{\texttt{alumni}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{1}} [\Box_{\otimes} (\text{“student”} \{\!\!\{ < \\
\quad\quad << \text{“startDate”} \mapsto [\mathrm{true}_{\mathrm{AF}}] \gg \otimes_{\mathrm{AF}} \\
\quad\quad << \text{“finishDate”} \mapsto [\mathrm{true}_{\mathrm{AF}}] \gg \\
\quad >_{\mathrm{EA}} \}\!\!\}_{\mathbf{1}} [\\
\quad\quad < \text{“name”} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{1}} [TXTS] \mathbf{null} >_{\mathrm{EF}} \otimes_{\mathrm{EF}} \\
\quad\quad < \text{“address”} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{1}} [TXTS] \mathbf{null} >_{\mathrm{EF}} \otimes_{\mathrm{EF}} \\
\quad\quad < \text{“subject”} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{1}} [TXTS] \mathbf{null} >_{\mathrm{EF}} \otimes_{\mathrm{EF}} \\
\quad] \mathbf{null}, \mathrm{EF}) \otimes_{\mathrm{EF}} \\
\quad \text{“student”}_{\texttt{currentStudent}} \{\!\!\{ < \\
\quad\quad << \text{“startDate”} \mapsto [\mathrm{true}_{\mathrm{AF}}] \gg \otimes_{\mathrm{AF}} \\
\quad\quad << \text{“finishDate”} \mapsto [\mathrm{true}_{\mathrm{AF}}] \gg \\
\quad >_{\mathrm{EA}} \}\!\!\}_{\mathbf{1}} [\\
\quad\quad < \text{“name”} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{1}} [TXTS] \mathbf{null} >_{\mathrm{EF}} \otimes_{\mathrm{EF}} \\
\quad\quad < \text{“address”} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{1}} [TXTS] \mathbf{null} >_{\mathrm{EF}} \otimes_{\mathrm{EF}} \\
\quad\quad < \text{“subject”} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{1}} [TXTS] \mathbf{null} >_{\mathrm{EF}} \otimes_{\mathrm{EF}} \\
\quad\quad \varnothing_{\mathrm{EF}} \\
\quad] \mathbf{null} \otimes_{\mathrm{EF}} \\
\quad \Box_{\otimes} (\text{“student”} \{\!\!\{ < \\
\quad\quad << \text{“startDate”} \mapsto [\mathrm{true}_{\mathrm{AF}}] \gg \otimes_{\mathrm{AF}} \\
\quad\quad << \text{“finishDate”} \mapsto [\mathrm{true}_{\mathrm{AF}}] \gg \\
\quad >_{\mathrm{EA}} \}\!\!\}_{\mathbf{1}} [\\
\quad\quad < \text{“name”} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{1}} [TXTS] \mathbf{null} >_{\mathrm{EF}} \otimes_{\mathrm{EF}} \\
\quad\quad < \text{“address”} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{1}} [TXTS] \mathbf{null} >_{\mathrm{EF}} \otimes_{\mathrm{EF}} \\
\quad\quad < \text{“subject”} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{1}} [TXTS] \mathbf{null} >_{\mathrm{EF}} \otimes_{\mathrm{EF}} \\
\quad] \mathbf{null}, \mathrm{EF})] \mathbf{null} \\
] _{\texttt{kids}} \mathbf{null} >_{\mathrm{DE}}, \varnothing_{\mathrm{DNEL}} >_{\mathrm{DF}} \wedge \\
\mathrm{flatten}(\text{“finalYear”}, \texttt{FINALS}' \text{:EF})] \mathbf{null} >_{\mathrm{G}} \oplus \\
< \text{“finalYear”}^{\texttt{doc}}_{\texttt{tags}} >_{\mathrm{G}} \oplus \\
< \text{“finalYear”}_{\texttt{currentStudentTag}} \{\!\!\{ \varnothing_{\mathrm{EA}} \}\!\!\}_{\mathbf{1}} [\varnothing_{\mathrm{EF}}] \mathbf{null} >_{\mathrm{G}}
\end{array}
\right\}
$$

od
endloc
$\{S \oplus \mathrm{true}_{\mathrm{G}}\}$

C. Implementation Behaviour

This appendix documents several experiments performed to ascertain the behaviour of various real world DOM implementations. Section C.1 investigates the behaviour of the normalize command, Section C.2 investigates the behaviour of default attributes and Section C.3 investigates the behaviour of the "specified" object-attribute.

C.1. The normalize Command

We investigated the behaviour of browsers when executing a "normalize" operation with a series of JavaScript tests [37]. The source for these tests is reproduced here:

```
<html>
  <head>
    <title>An experiment with DOM normalize</title>
<script>
text = null;
oldText = null;
function addmorewords() {
para = document.getElementById("words")
        oldText = para.childNodes.item(0);
        text = document.createTextNode("... and now there are more words.");
        para.appendChild(text);
}
function viewtext() {
        if (text==null) {
                alert("That node is null - you can't view the contents");
        }
        else { alert("The contents of that node are: "+(text.nodeValue)); }
}
```

```
function viewoldtext() {
        if (oldText==null) {
                alert("That node is null - you can't view the contents");
        }
        else { alert("The contents of that node are: "+(oldText.nodeValue)); }
}
function normexp() {
para = document.getElementById("words");
        para.normalize();
}
function moveexp() {
        para = document.getElementById("sandbox");
        para.appendChild(text);
}
function moveoldexp() {
        para = document.getElementById("sandbox");
        para.appendChild(oldText);
}
</script>
  </head>
  <body>
          <h1>An experiment with DOM normalize</h1>
<p id="words">
This paragraph is the one we will experiment with - it contains some words.
</p>
<p>
You can add some more words to the previous paragraph by clicking this button:
<input type="button" value="Add More Words" onclick="addmorewords()"/>
</p>
<p>
If you just clicked on that button, then the paragraph now has more words in
it. These are distinct from the original words - they are a separate text node
which was added by the javascript in the button (if you're using firefox, you
can verify this with the DOM Inspector). I've kept a reference to that text
node. You can see the contents of that new separate text node by clicking this
button:
```

```
<input type="button" value="View Text Node Contents" onclick="viewtext()"/>
</p>
<p>
When you added more words, I also kept a reference to the text that
  was already in the paragraph. You can see the contents of that node
  by clicking here:
<input type="button" value="View Old Text Contents" onclick="viewoldtext()"/>
</p>
<p>
If we don't want there to be two different text nodes inside that paragraph, we
can <em>normalize</em> the paragraph. DOM specifies normalize like
<a href="http://www.w3.org/TR/REC-DOM-Level-1/level-one-core.html#ID-162CF083">so</
        <blockquote>
                Puts all Text nodes in the full depth of the sub-tree
                underneath this Element into a &quot;normal&quot; form where
                only markup (e.g., tags, comments, processing instructions,
                CDATA sections, and entity references) separates Text nodes,
                i.e., there are no adjacent Text nodes. This can be used to
                ensure that the DOM view of a document is the same as if it
                were saved and re-loaded, and is useful when operations (such
                as XPointer lookups) that depend on a particular document tree
                structure are to be used.<br />
                This method has no parameters.<br />
                This method returns nothing.<br />
                This method raises no exceptions.
        </blockquote>
</p>
<p>
You can normalize our experimental paragraph with this button:
<input type="button" value="Normalize!" onclick="normexp()"/>
</p>
<p>
So now there should be only one text node in our paragraph - but it
should render just the same (again - look in the DOM Inspector). So
far so good. But what about the reference we had to the second text
node? Given what the DOM spec said - what should the value of that
```

```
variable be? What about the old text node? You can click the
&quot;View Text Node Contents&quot; and &quot;View Old Text
Contents&quot; buttons to find out...
</p>
<p>
If you're using firefox, chrome or safari, then what's happened here
is that the contents of the second text node were copied, and appended
to the first text node. The second text node was then removed as if
with &quot;removeChild&quot; - so the reference to that node is still
valid, and it can in fact be moved back into the document tree. Try
it:
        <input type="button"
               value="Move text to sandbox"
               onclick="moveexp()" />
Similarly, we can try to move the old text node (which now contains
all the text) into the sand box:
        <input type="button"
               value="Move old text to sandbox"
               onclick="moveoldexp()" />
If, on the other hand, you were using Internet Explorer, then both
text nodes were removed, and a brand new one was created containing
the concatenation of their contents.
</p>
<p id="sandbox">
Here's a sandbox you can move a text node into.
</p>
  </body>
</html>
```

C.2. Default Attributes

We investigated the behaviour of browsers when interacting with default attributes with a series of JavaScript tests [36]. The source for these tests is reproduced here:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
```

```
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html> <head>
<title>Default Attribute Test</title>
<SCRIPT TYPE="text/javascript">
function bingFirst() {
    ele = document.getElementsByTagName("a").item(0);
    at = ele.attributes.getNamedItem("shape")
    if(at==null) {
        alert("It doesn't exist");
    } else {
        val = at.value;
        alert("The value is: `"+val+"'");
    }
}

function bingSecond() {
    ele = document.getElementsByTagName("a").item(1);
    at = ele.attributes.getNamedItem("shape")
    if(at==null) {
        alert("It doesn't exist");
    } else {
        val = at.value;
        alert("The value is: `"+val+"'");
    }
}

function bingThird() {
    ele = document.getElementsByTagName("a").item(2);
    at = ele.attributes.getNamedItem("href")
    if(at==null) {
        alert("It doesn't exist");
    } else {
        val = at.value;
        alert("The value is: `"+val+"'");
    }
}
```

```
function bingFourth() {
    ele = document.getElementsByTagName("a").item(3);
    at = ele.attributes.getNamedItem("href")
    if(at==null) {
        alert("It doesn't exist");
    } else {
        val = at.value;
        alert("The value is: '"+val+"'");
    }
}

function ieTest() {
    ele = document.getElementsByTagName("a").item(3);
    at = ele.attributes.getNamedItem("fraggle");
    if(at==null) {
        alert("It doesn't exist. And that's good, because it's a magic fraggle.");
    } else {
        val = at.value;
        alert("Fraggles exist! With value: '"+val+"'");
    }
}

function ieSecondTest() {
    para = document.getElementById("iePara");
    a = document.createElement("a");
    a.appendChild(document.createTextNode("Link"));

    at = a.attributes.getNamedItem("href");
    if(at==null) {
        alert("It doesn't exist. Despite being specified in the DTD.");
    } else {
        val = at.value;
        alert("It does exist. With value: '"+val+"'");
    }
```

```
    para.appendChild(a);
    ele = document.getElementsByTagName("a").item(4);
    at = ele.attributes.getNamedItem("href")
    if(at==null) {
        alert("It still doesn't exist. Despite being specified in the DTD.");
    } else {
        val = at.value;
        alert("It does exist now. With value: '"+val+"'");
    }
}


</SCRIPT>


</head>

<body>
<h1>Default Attribute Test</h1>

<p>
This is a test of how the DOM deals with default attributes. All test
results mentioned in the prose were gotten using Firefox 2.0.0.14,
Firefox 3.5.5, Safari 4.0.4, IE
6.0.2900.2180.xpsp_sp2_gdr.070227-2254, IE 8.0.6001.18702 and Chrome
4.0.249.30. Here's the bit of DOM spec (with broken internal links)
that caused me to ask these questions:</p>

<blockquote>


<dt><b>Interface <i>Attr</i></b></dt>
<dd>
<p />
The <code>Attr</code> interface represents an attribute in an
<code>Element</code> object. Typically the allowable values for
```

```
the attribute are defined in a document type definition.
  <p />
  ...
  <p />
  The attribute's effective value is determined as follows: if this
   attribute has been explicitly assigned any value, that value is the
   attribute's effective value; otherwise, if there is a declaration for
   this attribute, and that declaration includes a default value, then
   that default value is the attribute's effective value; otherwise, the
   attribute does not exist on this element in the structure model until
   it has been explicitly added.  Note that the <code>nodeValue</code>
   attribute on the <code>Attr</code> instance can also be used to
   retrieve the string version of the attribute's value(s).
  <p />
  ...
  <p />

<dl>
<dt><b>Attributes</b></dt>
<dd>

...

<dl>
<dt><code class='attribute-name'>specified</code></dt>
<dd>
If this attribute was explicitly given a value in the original
    document, this is <code>true</code>; otherwise, it is <code>false</code>.
    Note that the implementation is in charge of this attribute, not the
    user. If the user changes the value of the attribute (even if it ends up
    having the same value as the default value) then the <code>specified</code>
    flag is automatically flipped to <code>true</code>.  To re-specify the
    attribute as the default value from the DTD, the user must delete the
    attribute. The implementation will then make a new attribute available
    with <code>specified</code> set to <code>false</code> and the default value
    (if one exists).<p>
```

```
In summary:
<ul>
<li> If the attribute has an assigned value in the document then
    <code>specified</code> is <code>true</code>, and the value is the
    assigned value.
</li>
<li> If the attribute has no assigned value in the document and has
    a default value in the DTD, then  <code>specified</code> is
    <code>false</code>, and the value is the default value in the DTD.
</li>
<li> If the attribute has no assigned value in the document and has
    a value of #IMPLIED in the DTD, then the  attribute does not appear
    in the structure model of the document.
</li>
</ul>

</dd>
</dl>

</dd>
</dl>
</dd>
</dl>



</blockquote>

<p>I want to understand how DTD specified default-values actually
work. First, I'm going to create an element node with an attribute
which has a default value in the DTD. Here's a section of the xhtml
DTD we're using:
</p>
<pre>
&lt;!--================== The Anchor Element ================================--&gt;
```

```
&lt;!-- content is %Inline; except that anchors shouldn't be nested --&gt;

&lt;!ELEMENT a %a.content;&gt;
&lt;!ATTLIST a
  %attrs;
  %focus;
  charset     %Charset;       #IMPLIED
  type        %ContentType;   #IMPLIED
  name        NMTOKEN         #IMPLIED
  href        %URI;           #IMPLIED
  hreflang    %LanguageCode;  #IMPLIED
  rel         %LinkTypes;     #IMPLIED
  rev         %LinkTypes;     #IMPLIED
  shape       %Shape;         &quot;rect&quot;
  coords      %Coords;        #IMPLIED
  &gt;
</pre>

<p>
Since the anchor element has a default shape, I'm going to experiment
with a few anchor elements. Here's one
to <a href="http://google.com">Google</a>. And
here's <a href="http://www.google.com" shape>one</a> with an empty
shape attribute (if such a thing exists). Next we'll have a couple of
buttons to bing the contents of those shape attributes up in our
faces.
</p>
<p>
<input type="button" value="Bing first value" onclick="bingFirst()" />
<input type="button" value="Bing second value" onclick="bingSecond()" />
</p>

<p>
Notice how in chrome, safari and firefox, the first value simply
doesn't exist, and the second is assumed to be &quot;&quot;. IE
returns the latter value for both. Neither of these are the default
```

```
value specified in the DTD.
</p>
<p>
Since the specified attribute values don't seem to behave entirely as
specced, there's probably nothing useful to be learned from #IMPLIED
attribute values. Still, we might as well try. Two links with none
existent and empty anchors are <a>here</a> and <a href>here</a>.
</p>
<input type="button" value="Bing third value" onclick="bingThird()" />
<input type="button" value="Bing fourth value" onclick="bingFourth()" />
<p>
In firefox, chrome, safari and IE 6 these second two buttons behave
exactly like the first two. In IE 8 they both report that the value
doesn't exist.
</p>
<p>Given the results from IE, there's one more test that needs
performing. Does IE magically call any attribute you like into
existence whenever you ask for it in code?
<input type="button" value="IE test" onclick="ieTest()" />
</p>
<p id="iePara">On all platforms, that attribute doesn't exist. As well
it shouldn't. But this suggests that IE was only calling unmentioned
attributes into existence when they were mentioned in the DTD - not
indiscriminately. Does it do this the whole time, or only at parse
time? Let's try to create an anchor node without an href, and add it
to this paragraph.
<input type="button" value="IE Test take 2" onclick="ieSecondTest()" />
</p>
<p>
It does seem to be impossible to create anchor tags in IE 6 without
href attributes. Like it actually is enforcing the DTD in the
DOM. Interestingly, IE 8 doesn't do this.
</p>
<p>
Experiment with the &quot;specified&quot; property <a href="specified.html">here</a
</p>
```

```
</body> </html>
```

C.3. The "specified" Attribute

We investigated the behaviour of browsers when interacting with the "specified" object attribute of Attr nodes with a series of JavaScript tests [38]. The source for these tests is reproduced here:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html> <head>
<title>Attribute &quot;specified&quot; Test</title>
<SCRIPT TYPE="text/javascript">
function bingFirst() {
    ele = document.getElementsByTagName("a").item(1);
    at = ele.attributes.getNamedItem("shape")
    if(at==null) {
        alert("It doesn't exist");
    } else {
        val = at.specified;
        alert("The value is: `"+val+"'");
    }
}

function bingSecond() {
    ele = document.getElementsByTagName("a").item(2);
    at = ele.attributes.getNamedItem("shape")
    if(at==null) {
        alert("It doesn't exist");
    } else {
        val = at.specified;
        alert("The value is: `"+val+"'");
    }
}

function bingThird() {
```

```
    ele = document.getElementsByTagName("a").item(3);
    at = ele.attributes.getNamedItem("href")
    if(at==null) {
        alert("It doesn't exist");
    } else {
        val = at.specified;
        alert("The value is: '"+val+"'");
    }
}

function bingFourth() {
    ele = document.getElementsByTagName("a").item(4);
    at = ele.attributes.getNamedItem("href")
    if(at==null) {
        alert("It doesn't exist");
    } else {
        val = at.specified;
        alert("The value is: '"+val+"'");
    }
}

function tweakAttr() {
    ele=document.getElementsByTagName("a").item(4);
    at = ele.attributes.getNamedItem("href");
    if(at==null) {
        alert("It doesn't exist");
    } else {
        at.value = "http://www.doc.ic.ac.uk";
        alert("Changed");
    }
}

function gdsSpecified() {
    ele=document.getElementsByTagName("a").item(5);
    at = ele.attributes.getNamedItem("gds");
    if(at==null) {
```

```
        alert("It doesn't exist");
    } else {
        val = at.specified;
        alert("The value is: '"+val+"'");
    }
}

function changeGds() {
    ele=document.getElementsByTagName("a").item(5);
    at = ele.attributes.getNamedItem("gds");
    if(at==null) {
        alert("It doesn't exist");
    } else {
        at.value = "red";
        alert("Changed");
    }
}

function newShapeAttr() {
    at = document.createAttribute("shape");
    alert("The value is: '"+at.specified+"'");
}
function newHrefAttr() {
    at = document.createAttribute("href");
    alert("The value is: '"+at.specified+"'");
}
function newGdsAttr() {
    at = document.createAttribute("gdsish");
    alert("The value is: '"+at.specified+"'");
}

</SCRIPT>


</head>
```

```
<body>
<h1>Attribute &quot;specified&quot; Test</h1>
<p>
I experimented with the default values of
attributes <a href="defaultAttribute.html">here</a>. Here, we learn
about the &quot;specified&quot; property. Here's the relevent bit of
the DOM spec again:
</p>
<blockquote>
<dl>
<dt><code class='attribute-name'>specified</code></dt>
<dd>
If this attribute was explicitly given a value in the original
    document, this is <code>true</code>; otherwise, it is <code>false</code>.
    Note that the implementation is in charge of this attribute, not the
    user. If the user changes the value of the attribute (even if it ends up
    having the same value as the default value) then the <code>specified</code>
    flag is automatically flipped to <code>true</code>.  To re-specify the
    attribute as the default value from the DTD, the user must delete the
    attribute. The implementation will then make a new attribute available
    with <code>specified</code> set to <code>false</code> and the default value
    (if one exists).<p>
In summary:
<ul>
<li> If the attribute has an assigned value in the document then
    <code>specified</code> is <code>true</code>, and the value is the
    assigned value.
</li>
<li> If the attribute has no assigned value in the document and has
    a default value in the DTD, then  <code>specified</code> is
    <code>false</code>, and the value is the default value in the DTD.
</li>
<li> If the attribute has no assigned value in the document and has
    a value of #IMPLIED in the DTD, then the  attribute does not appear
    in the structure model of the document.
</li>
```

```
</ul>

</dd>
</dl>
</blockquote>
<p>
You may be forgiven for thinking that one of the instances of the
word <q>true</q> in there must be a typo. How does that attribute ever
get set to false? Let's find out.
</p>
<p>
Since the anchor element has a default shape, we experimented with a
few anchor elements. Here's one
to <a href="http://google.com">Google</a>. And
here's <a href="http://www.google.com" shape>one</a> with an empty
shape attribute (if such a thing exists). Next we'll have a couple of
buttons to bing the specified attribute of those shape attrs up in our
faces.</p>
<p>
<input type="button" value="Bing first value" onclick="bingFirst()" />
<input type="button" value="Bing second value" onclick="bingSecond()" />
</p>

<p>
In chrome, safari and firefox, the first one doesn't exist - as we
noticed in the previous experiment. The second one has a specified of
true. IE 6 agrees with the second but disagrees with the first, setting
that specified to false. IE 8 sets both to false.</p>
<p>
We perform the same experiment with #IMPLIED attribute values. Two
links with none existent and empty anchors are <a>here</a>
and <a href>here</a>.
</p>
<input type="button" value="Bing third value" onclick="bingThird()" />
<input type="button" value="Bing fourth value" onclick="bingFourth()" />
```

```
<p>As in the default attribute test, in firefox, chrome, safari and IE
6 these second two buttons behave exactly like the first two. In IE 8
they both report that the value doesn't exist.</p>

<p>
We can tweak the value of the last link, and see the difference it makes.
<input type="button" value="Tweak the attr" onclick="tweakAttr()" />
</p>
<p>It makes no difference on any platform.</a>
<p>
So far, we've discovered no false specifieds in chrome, safari or
firefox. Let's try two penultimate things to do that. Here's
a <a href="http://news.bbc.co.uk" gds="blue">link</a> with a weird
&quot;gds&quot; attribute.
</p>
<input type="button" value="gds specified?" onclick="gdsSpecified()" />
<input type="button" value="change gds" onclick="changeGds()" />
<p>The gds attr has a specified attribute of true whether it's changed
in code or not, on all platforms.</p>
<p>
Finally, what happens when we create a brand new attribute? With a
default value, implied, and unmentioned in the DTD?
<input type="button" value="new shape one" onclick="newShapeAttr()" />
<input type="button" value="new href one" onclick="newHrefAttr()" />
<input type="button" value="new gds one" onclick="newGdsAttr()" />
</p>

<p>
In firefox, those last three are all false. In chrome and safari, it
does seem to be impossible to find a false value for specified. IE
agrees with firefox.
</p>

</body> </html>
```