Abstract

We present technical details and proofs too lengthy to fit into a conference paper.

1 Operational Semantics

1.1 Core Language

Semantics for the standard imperative language features.

\[
\begin{align*}
[\text{Id}]_s & = \text{id} \\
\implies s, \text{FT}, g. \text{id} & = \text{Id} \leadsto [s] \text{id} \mapsto \text{id}, \text{FT}, g \\
\end{align*}
\]

\[
\begin{align*}
[\text{Str}]_s & = \text{s} \\
\implies s, \text{FT}, g. \text{str} & = \text{Str} \leadsto [s] \text{str} \mapsto \text{s}, \text{FT}, g \\
\end{align*}
\]

\[
\begin{align*}
[\text{Int}]_s & = \text{n} \\
\implies s, \text{FT}, g. \text{int} & = \text{Int} \leadsto [s] \text{int} \mapsto \text{n}, \text{FT}, g \\
\end{align*}
\]

\[
\begin{align*}
[\text{Bool}]_s & = \text{b} \\
\implies s, \text{FT}, g. \text{bool} = \text{Bool} \leadsto [s] \text{bool} \mapsto \text{b}, \text{FT}, g \\
\end{align*}
\]

\[
\begin{align*}
\text{J} & \\
\end{align*}
\]

\[
\begin{align*}
\text{FT} & \\
\end{align*}
\]

\[
\begin{align*}
\text{g} & \\
\end{align*}
\]

\[
\begin{align*}
\text{id} & \\
\end{align*}
\]

\[
\begin{align*}
\text{FT}, \text{g} & \\
\end{align*}
\]

\[
\begin{align*}
\text{s} & \\
\end{align*}
\]

\[
\begin{align*}
\text{\{C\}} & \\
\end{align*}
\]

\[
\begin{align*}
\text{\text{skip}} & \\
\end{align*}
\]

\[
\begin{align*}
\text{\text{fault}} & \\
\end{align*}
\]

Any unmentioned state

\[
\begin{align*}
s, \text{FT}, g. \text{C} & \leadsto \text{fault} \\
\end{align*}
\]

The functions and web rules are given in the main paper body.
1.2 Document

From the “Document” interface, we define the commands:

\[ g \equiv g' \oplus <\text{Document}[^{null|\text{doc}}]|\text{[id]}_{\text{did}} |_{\text{[name]}} |_{\text{[text|\text{data}]}}, f_{\text{nd}}, g \]
\[ g' \equiv g \oplus <\text{[name]}_{\text{[doc]}}, |_{\text{[doc]}}, f_{\text{nd}}, g \]

\[ s, FT, g, x := \text{createElement}(\text{doc}, \text{name}) \prec [s|x \leftarrow \text{id}], FT, g' \]

\[ g \equiv g' \oplus <\text{Document}[^{null|\text{doc}}], f_{\text{nd}}, g \]
\[ g' \equiv g \oplus <\text{[text|\text{data}]}_{\text{[doc]}}, f_{\text{nd}}, g \]

\[ s, FT, g, x := \text{createTextNode}(\text{doc}, \text{data}) \prec [s|x \leftarrow \text{id}], FT, g' \]

1.3 Node

From the “Node” interface, we define the commands:

\[ g \equiv \text{ap}(cg, \text{name}_{\text{[did]}, [f]_{\text{nd}}} \]
\[ s, FT, g, nm := \text{getnodeName}(n) \prec [s|nm \leftarrow \text{name}], FT, g \]

\[ g \equiv g' \oplus \#\text{Document}[^{null|\text{doc}}], f_{\text{nd}} \]
\[ s, FT, g, nm := \text{getnodeName}(n) \prec [s|nm \leftarrow \#\text{Document}], FT, g \]

\[ g \equiv \text{ap}(cg, \text{name}_{\text{[did]}, [f]_{\text{nd}}}, f'_{\text{nd}} \oplus f_{\text{nd}}) \]
\[ s, FT, g, p := \text{getParentNode}(n) \prec [s|p \leftarrow \text{id}], FT, g \]

\[ g \equiv g' \oplus <\text{name}_{\text{[did]}, [f]_{\text{nd}}}, g \]
\[ s, FT, g, p := \text{getParentNode}(n) \prec [s|p \leftarrow \text{null}], FT, g \]

\[ g \equiv g' \oplus <\#\text{Document}[^{null|\text{doc}}], f_{\text{nd}}, g \]
\[ s, FT, g, p := \text{getParentNode}(n) \prec [s|p \leftarrow \text{null}], FT, g \]

\[ g \equiv \text{ap}(cg, \text{name}_{\text{[did]}, [f]_{\text{nd}}} \]
\[ s, FT, g, kids := \text{getChildren}(n) \prec [s|kids \leftarrow \text{fid}], FT, g \]

\[ g \equiv g' \oplus \#\text{Document}[^{null|\text{doc}}], f_{\text{nd}} \]
\[ s, FT, g, kids := \text{getChildren}(n) \prec [s|kids \leftarrow \text{fid}], FT, g \]

\[ g \equiv \text{ap}(cg, \text{name}_{\text{[did]}, [f]_{\text{nd}}} \]
\[ s, FT, g, od := \text{getOwnerDocument}(n) \prec [s|od \leftarrow \text{did}], FT, g \]

\[ g \equiv g' \oplus \#\text{Document}[^{null|\text{doc}}], f_{\text{nd}} \]
\[ s, FT, g, od := \text{getOwnerDocument}(n) \prec [s|od \leftarrow \text{null}], FT, g \]

We now give the rules for structural manipulation commands.

\[ g \equiv \text{ap}(cg', <\text{name}_{\text{[newChild]}, [f]_{\text{nd}}}, g'_{\text{nd}} >_{\text{id}}) \]
\[ \text{ap}(cg', \Theta_{\text{id}}) \equiv \text{ap}(cg, s'_{\text{did}} \text{parent}, [\text{ef}]_{\text{nd}}) \]

\[ g' \equiv \text{ap}(cg, s'_{\text{did}} \text{parent}, [\text{ef}] \oplus <\text{name}_{\text{[newChild]}, [f]_{\text{nd}}}, g'_{\text{nd}} >_{\text{id}}) \]
\[ s, FT, g, n := \text{appendChild}(\text{parent}, \text{newChild}) \prec [s|n \leftarrow \text{newChild}], FT, g' \]

\[ g \equiv \text{ap}(cg, <\text{[parent]}, [f]_{\text{nd}}>, g'_{\text{nd}}) \oplus <\text{Document}[^{null|\text{doc}}], [\text{ef}]_{\text{nd}}, g \]

\[ g' \equiv \text{ap}(cg, \Theta_{\text{id}}) \oplus <\text{Document}[^{null|\text{doc}}], [\text{parent}], s'_{\text{did}} \text{val}, [\text{[newChild]}, [f]_{\text{nd}}], g'_{\text{nd}} >_{\text{id}} \]
\[ s, FT, g, n := \text{appendChild}(\text{parent}, \text{newChild}) \prec [s|n \leftarrow \text{newChild}], FT, g' \]
1.4 NodeList

We require a function, len, that will determine the length of a forest.

\[
\begin{align*}
\text{len}(f_1 \otimes f_2) & \triangleq \text{len}(f_1) + \text{len}(f_2) \\
\text{len}(\varnothing_{D_1}) & \triangleq 0 \\
\text{len}(\langle \text{name}_{id} \rangle \text{ea}_{\text{null}} \text{\text{詹姆}} \text{\text{null}} \text{\text{詹姆}} \text{\text{null}} \triangleright \text{\text{詹姆}}) & \triangleq 1 \\
\end{align*}
\]

From the “NodeList” interface, we define the commands:

\[
\begin{align*}
l & \quad \text{len}(f_1) = [\text{int}]_a \land \\
g & \quad \text{ap} \left( \text{cg}, \text{name}_{id}^{\text{did}} \cdot f_1 \otimes \langle \text{\text{name}}_{id}^{\text{did}} \rangle \cdot \text{\text{詹姆}}_{\text{did}} > \text{\text{詹姆}}_{\text{did}} \otimes f_2 \cdot \text{\text{詹姆}}_{\text{did}} > \right) \\
& \quad \text{s}, \text{FT}, g, n := \text{item(list, int)} \leadsto [s|n \leftarrow \text{\text{詹姆}}], \text{FT}, g \\
& \quad (\text{len}(f) \leq [\text{int}]_a \lor [\text{int}]_a < 0) \land \\
& \quad g \equiv \text{ap} \left( \text{cg}, \text{name}_{id}^{\text{did}} \langle f \rangle_{\text{詹姆}}_{\text{null}} \right) \\
& \quad \text{s}, \text{FT}, g, n := \text{item(list, int)} \leadsto [s|n \leftarrow \text{\text{詹姆}}], \text{FT}, g \\
& \quad [\text{int}]_a = 0 \land \\
& \quad g \equiv g' \odot \# \text{\text{document}}_{\text{null}} \langle \text{\text{name}}_{id}^{\text{did}} \rangle \cdot \text{\text{詹姆}}_{\text{null}} \rangle \text{\text{詹姆}}_{\text{null}} > \text{\text{詹姆}}_{\text{null}} \rangle \\
& \quad \text{s}, \text{FT}, g, n := \text{item(list, int)} \leadsto [s|n \leftarrow \text{\text{詹姆}}], \text{FT}, g \\
& \quad (\text{len}(f) \leq [\text{int}]_a \lor [\text{int}]_a < 0) \land \\
& \quad g \equiv g' \odot \# \text{\text{document}}_{\text{null}} \langle f \rangle_{\text{詹姆}}_{\text{null}} \\
& \quad \text{s}, \text{FT}, g, n := \text{item(list, int)} \leadsto [s|n \leftarrow \text{\text{詹姆}}], \text{FT}, g
\end{align*}
\]

1.5 Text Data

From the textual manipulation interfaces, we define the following command. We assume the changes to the function len above such that it finds the lengths of strings.

\[
\begin{align*}
l & \quad \text{len}(s_1) = [\text{offset}]_a \land \text{len}(s_2) = [\text{count}]_a \land \\
g & \quad \text{ap} \left( \text{cg}, \text{name}_{\text{node}}^{\text{did}} \cdot [s_1 \otimes s_2 \otimes s_3] \right) \\
& \quad \text{s}, \text{FT}, g, \text{str} := \text{\text{substring}}_{\text{node}, \text{offset}, \text{count}} \leadsto [s|\text{str} \leftarrow \text{\text{詹姆}}], \text{FT}, g \\
& \quad \text{len}(s_1) = [\text{offset}]_a \land \text{len}(s_2) = [\text{count}]_a \land \\
g & \quad \text{ap} \left( \text{cg}, \text{name}_{\text{node}}^{\text{did}} \cdot [s_1 \otimes s_2 \otimes s_3] \right) \\
& \quad \text{s}, \text{FT}, g, \text{str} := \text{\text{substring}}_{\text{node}, \text{offset}, \text{count}} \leadsto [s|\text{str} \leftarrow \text{\text{詹姆}}], \text{FT}, g \\
l & \quad \text{len}(s_1) = [\text{offset}]_a \land \text{len}(s_2) = [\text{count}]_a \land \\
g & \quad \text{ap} \left( \text{cg}, \text{name}_{\text{node}}^{\text{did}} \cdot [s_1 \otimes s_2 \otimes s_3] \right) \\
g' & \quad \text{ap} \left( \text{cg}, \text{name}_{\text{node}}^{\text{did}} \cdot [s_1 \otimes s_2 \otimes s_3] \right) \\
& \quad \text{s}, \text{FT}, g, \text{\text{delete}}_{\text{node}, \text{offset}, \text{count}} \leadsto [s|\text{FT}, g] \\
\end{align*}
\]
2 Complete Command Axioms

\[
\begin{align*}
\text{len}(s_1) &= [\text{offset}]_s \land \text{len}(s_2) < [\text{count}]_s \land \\
g &\equiv \text{ap}(\text{gc}\cdot\text{name}\cdot\text{lid}([\text{node}], s_1 \otimes s_2)) \\
g' &\equiv \text{ap}(\text{gc}\cdot\text{name}\cdot\text{lid}([\text{node}], s_1)) \\
&\text{s, FT, g, deleteData(node, offset, count)} \leadsto s, \text{FT, g} \\
g &\equiv \text{ap}(\text{gc}\cdot\text{name}\cdot\text{lid}([\text{node}], [s])) \\
g' &\equiv \text{ap}(\text{gc}\cdot\text{name}\cdot\text{lid}([\text{node}], s \otimes [\text{arg}], [\text{lid}]))) \\
&\text{s, FT, g, appendData(node, arg)} \leadsto s, \text{FT, g}
\end{align*}
\]

For the Node interface, we give:

\[
\begin{align*}
\{&\text{name}^\text{did}[f]\mid \text{fid} > G \land x = Y \\
\text{nm} &= \text{createElement}(\text{doc}, \text{name}) \\
\{&\text{name}^\text{null}[\text{doc}[\text{fid} > G \oplus \text{name}^{\text{doc}}\{\text{y} / \text{x}\}][\text{doc}[\text{fid} > G] \\
\{&\text{name}^\text{null}[\text{doc}][\text{y} / \text{x}]\mid \text{fid} > G \land x = Y \\
\text{x} &= \text{createTextNode}(\text{doc}, \text{data}) \\
\{&\text{name}^\text{null}[\text{doc}[\text{y} / \text{x}]\mid \text{fid} > G \oplus \text{#text}^{\text{doc}}\{\text{y} / \text{x}\}[\text{data}[\text{y} / \text{x}]][\text{fid} > G]
\end{align*}
\]
For the NodeList interface, we give.

\[
\{ \text{gc} \circ D_2 \text{parent} \mid f_1 \otimes <\text{name}\text{oldChild} [f']_{\text{fid}} \circ D_3 \otimes f_2]_{\text{fid}} \wedge n = \text{null} \}
\]

For the Text Data interfaces, we give.

\[
\{ \text{#text}\text{node} \mid s_1 \otimes s_2 \otimes s_3]_{\text{fid}} \wedge \text{offset} = ||s_1|| \wedge \text{count} = ||s_2|| \wedge \text{str} = \text{Y} \}
\]

\[
\{ \text{#text}\text{node} \mid s_1 \otimes s_2 \otimes s_3]_{\text{fid}} \wedge \text{offset} = ||s_1|| \wedge \text{count} > ||s_2|| \wedge \text{str} = \text{Y} \}
\]

\[
\{ \text{#text}\text{node} \mid s_1 \otimes s_2 \otimes s_3]_{\text{fid}} \wedge \text{offset} = ||s_1|| \wedge \text{count} = ||s_2|| \}
\]

\[
\{ \text{#text}\text{node} \mid s_1 \otimes s_2 \otimes s_3]_{\text{fid}} \wedge \text{offset} = ||s_1|| \wedge \text{count} > ||s_2|| \}
\]

\[
\{ \text{#text}\text{node} \mid s_1 \otimes s_2 \otimes s_3]_{\text{fid}} \wedge \text{offset} = ||s_1|| \wedge \text{count} = ||s_2|| \}
\]

\[
\{ \text{#text}\text{node} \mid s_1 \otimes s_2 \otimes s_3]_{\text{fid}} \wedge \text{offset} = ||s_1|| \wedge \text{count} > ||s_2|| \}
\]

\[
\{ \text{#text}\text{node} \mid s_1 \otimes s_2 \otimes s_3]_{\text{fid}} \wedge \text{offset} = ||s_1|| \wedge \text{count} = ||s_2|| \}
\]

\[
\{ \text{#text}\text{node} \mid s_1 \otimes s_2 \otimes s_3]_{\text{fid}} \wedge \text{offset} = ||s_1|| \wedge \text{count} > ||s_2|| \}
\]

\[
\{ \text{#text}\text{node} \mid s_1 \otimes s_2 \otimes s_3]_{\text{fid}} \wedge \text{offset} = ||s_1|| \wedge \text{count} = ||s_2|| \}
\]

\[
\{ \text{#text}\text{node} \mid s_1 \otimes s_2 \otimes s_3]_{\text{fid}} \wedge \text{offset} = ||s_1|| \wedge \text{count} > ||s_2|| \}
\]

3 Definitions

We ease of reading, we repeat some key definitions from the paper that we will use in our proofs.

**Definition 1** (Fault avoiding interpretation). We use a fault avoiding interpretation for our Hoare Triples, such that \{P\}C\{Q\} means “When C is executed in an environment satisfying P, the code cannot reduce to fault state. If the execution does not diverge, the environment of any terminal state satisfies Q.”
Definition 2 (Satisfaction Relation). Satisfaction for our assertion language is defined by induction on the structure of assertions, as below.

\[
\begin{align*}
e, s, FT, a \models_A P & \Rightarrow P' \iff e, s, FT, a \models_A P \Rightarrow e, s, FT, a \models_A P' \\
e, s, FT, d \models_D \text{false}_D & \text{ never} \\
e, s, FT, cd \models_{D_1 \rightarrow D_2} \text{false}_D & \text{ never} \\
e, s, FT, a \models_A P & \Rightarrow e, s, FT, a \models_A P' \\
e, s, FT, d \models_D \text{false}_D & \text{ never} \\
\end{align*}
\]

4 Proofs

We now give proofs of our results. Many elements are similar to existing work on Separation Logic and Hoare reasoning; we refer the reader to such existing work where possible. We focus mostly on our mashup specific additions.

Lemma 13. The small axioms are sound with respect to the operational semantics.
Proof. The proofs of the small axioms follow directly from the operational semantics in the following ways.

**Data Retrieval:** The proof for all the data retrieval commands are similar. We illustrate using the “getNodeName” command. This command has the following operational semantics:

\[
g \equiv \text{ap}(cg, \text{name}^{\text{did}}_{[m]}, [f]_{\text{fid}})\]

\[
s, FT, g, \text{nm} := \text{getNodeName}(n) \rightarrow [s][\text{nm} \leftarrow \text{name}], FT, g\]

\[
g \equiv g' \oplus \#\text{document}^{\text{null}}_{[m]}, [f]_{\text{fid}}\]

\[
s, FT, g, \text{nm} := \text{getNodeName}(n) \rightarrow [s][\text{nm} \leftarrow \#\text{document}], FT, g\]

And the axiom:

\[
\{\text{name}^{\text{did}}_{n}[f]_{\text{fid}} \land \text{nm} = Y\}
\]

\[
\text{nm} = \text{getNodeName}(n)
\]

\[
\{\text{name}^{\text{did}}_{n[Y/\text{nm}]}[f]_{\text{fid}} \land \text{nm} = \text{name}\}
\]

We consider this command in two cases. The case in which the node in question is a Document node, and the case in which it isn’t. Both these cases are similar. We deal first with the case in which the node is not a Document node.

The precondition for getNodeName is:

\[
\text{name}^{\text{did}}_{n}[f]_{\text{fid}} \land \text{nm} = Y
\]

By the interpretation of Hoare Triples in Definition 1 (given that this is not a Document Node), we consider program states satisfying the condition:

\[
<\text{name}^{\text{did}}_{n}[f]_{\text{fid}} > G \land \text{nm} = Y
\]

By the satisfaction relation in definition 2, any concrete state \(s, FT, g\) satisfying this condition in a logical environment \(e\) must be described by:

\[
\exists s, d'. (g \equiv <s^{\text{did}}, [d']_{s(t(fid))} > G \land e, s, FT, s \models s \text{name} \land e, s, FT, d' \models D' f) \land s(\text{nm}) = s(Y)
\]

This is equivalent to:

\[
\exists f : D'. (g \equiv \text{ap}(G, <\text{name}^{\text{did}}_{n[Y/\text{nm}]}[f]_{\text{fid}} > G)) \land s(\text{name}) = \text{name} \land [d]_{s} = \text{did} \land e(f) = f \land s(\text{fid}) = \text{fid} \land s(\text{nm}) = s(Y)
\]

If we apply the operational semantic for \(\text{nm} = \text{getNodeName}(n)\), then \(FT, g\) remain unchanged, but \(s\) is updated to become \([s][\text{nm} \leftarrow \text{name}]\), and so we have:

\[
\exists f : D'. (g \equiv \text{ap}(G, <\text{name}^{\text{did}}_{n[Y/\text{nm}]}[f]_{\text{fid}} > G)) \land s(\text{name}) = \text{name} \land [d]_{s} = \text{did} \land e(f) = f \land s(\text{fid}) = \text{fid} \land s(\text{nm}) = s(\text{name})
\]

This state satisfies the formula:

\[
<\text{name}^{\text{did}}_{n[Y/\text{nm}]}[f]_{\text{fid}} > G \land \text{nm} = \text{name}
\]

And by the interpretation of Hoare Triples, this gives us the postcondition:

\[
\text{name}^{\text{did}}_{n[Y/\text{nm}]}[f]_{\text{fid}} \land \text{nm} = \text{name}
\]

The case for Document nodes is similar, diverging only in the detail of the interpretation of the Hoare triple, and the use of the (similar) Document case of the operational semantics. The precondition for getNodeName is:

\[
\text{name}^{\text{did}}_{n}[f]_{\text{fid}} \land \text{nm} = Y
\]

By the interpretation of Hoare Triples in Definition 1 (given that this is Document Node), we consider program states satisfying this condition. By the satisfaction relation in Definition 2, any concrete state \(s, FT, g\) satisfying this condition in a logical environment \(e\) must be described by:

\[
\exists s, d'. (g \equiv s^{\text{did}}, [d']_{s(t(fid))} \land e, s, FT, s \models s \text{name} \land e, s, FT, d' \models D' f) \land s(\text{nm}) = s(Y)
\]
This is equivalent to:

\[ \exists f \cdot (g \equiv \text{zero} \wedge \#document \ni \text{null} \wedge \|\text{did}\|_s = \text{null} \wedge \varepsilon(f) = f \wedge s(\text{fid}) = \text{fid} \wedge s(\text{nm}) = s(Y) \]

If we apply the operational semantic for \( \text{nm} = \text{getNodeName}(n) \), then \( \text{FT}, g \) remain unchanged, but \( s \) is updated to become \( [s|\text{nm} \leftarrow \#document] \), and so we have:

\[ \exists f \cdot (g \equiv \emptyset \oplus \#document, [f|\text{fid}] \wedge \|\text{did}\|_s = \text{null} \wedge \varepsilon(f) = f \wedge s(\text{fid}) = \text{fid} \wedge s(\text{nm}) = s(\text{name}) \]

This state satisfies the formula:

\[ \text{name} \text{did} \ni (f|\text{fid}) \wedge \text{nm} = \text{name} \]

...which is the postcondition of the command.

**Theorem 19 (Component Soundness).** Let \( W \) contain only correct specifications. Then, the proof of any component \( u \) with respect to \( W \) is sound.

*Proof.* The standard inference rules of Hoare Logic remain sound in our presentation. Each of our axioms is sound (by lemma 13), and is both safety and termination monotone; that is, if \( C, s, \text{FT}, g \) does not fault and terminates identically to the same grove extended by an arbitrary context. The proof of these, and the frame rule are then similar to those in Local Hoare reasoning about DOM (Gardner et al). The interesting cases are those handling functions and remote web data. Many of these rules have quite simple antecedent structure, and act almost as axioms. We describe each in turn.

1. **Function introduction:** Recall the function introduction operational semantic and inference rule.

\[
\begin{align*}
\forall P, Q, \text{Mods}. & f(p_1, \ldots, p_m) \{ C \} \leadsto s, \text{FT}[f \leftarrow (p^n, C)], g \\
\implies & \{ P \wedge \gamma(g_1) \wedge \ldots \wedge \gamma(g_n) \} \text{C}(Q) \\
\wedge & \text{mods}(C) \subseteq \text{Mods} \\
\text{\{empty\}function f(p_1, \ldots, p_n) \{ C \}\{ \gamma(f) \wedge \emptyset \}}
\end{align*}
\]

The operational effect of introducing a function is the entry of the parameters and code into the function table. This operation cannot fault, always terminates, and neither reads nor alters any store variables or heap structure. The precondition required in the conclusion of the rule is satisfied only by states with an empty heap, and arbitrary store and function table; this follows directly from the satisfaction relation. The postcondition is satisfied by the same state, except that the function table must contain an entry for the introduced function - an effect ensured by the operational rule.

The antecedent of this rule checks every specification in the specification web for the function name \( f \) against the code that is being introduced for that name; in effect, using this rule requires that a subproof be generated to show that the concrete code given for the function satisfies every specification that could be used for it. This requires a check of the Hoare triple given, and the modification set stated.

Notice that in the check, we conjoin \( \gamma(g_1) \ldots \gamma(g_n) \) to the specified precondition. This asserts that any local functions to the component have been introduced. It is sound by the construction of programs - all local functions are introduced before any function body can be called (see the language grammar), functions cannot be removed once introduced (by direct appeal to the semantics), and so any execution of \( C \) must occur in an environment where the functions exist.
2. **Function call**: Recall the function call operational semantic and inference rule.

\[
\begin{align*}
\text{FT}(f) &= (\mathcal{P}^n, \mathcal{C}) \\
\text{s, FT, g, } \Theta([c, s])[e_1/p_1, \ldots, e_m/p_m] &\rightarrow s', \text{FT}', \text{g}' \\
\text{s, FT, g, v := f(e_1, \ldots, e_n) } &\rightarrow s'\{\text{v/ret}\}, \text{FT}', \text{g}' \\

f(p_1, \ldots, p_m)_{\text{spec}} &: (P, Q, \text{Mods}) \in \mathcal{F} \\
\{\gamma(f) \land \theta(P)\}v &= f(e_1, \ldots, e_m)\{\gamma(f) \land \theta(Q)[v/\text{ret}]\}
\end{align*}
\]

The operational effect of calling a function is to look up the code associated with a name and arity within the function table, then to reduce that code to either a terminal state (i.e., store, function table and heap), or to diverge (if the code does). This process can fault if the function table does not have a entry for the given function name and arity, or if the code the function table contains for the function faults during execution; it terminates under the same conditions as the code given for the function body. Argument expressions are passed by direct substitution for the parameter names. No assignment is permitted to formal parameter names, hence this cannot introduce faults.

The definition of the \( \Theta \) substitution operation are such that any variable in the function code declared `local` will be renamed so as to not clash with any variable currently in the environment, that will be introduced to the environment, or is mentioned anywhere on the web (practically, this will be ensured by prefixing some unique identifier based on the URI). This gives the correct interpretation of locality, as though these local variables continue to exist in the store, no other code can possibly access them unless they are passed as parameters, as it does not know their names.

The associated inference rule obtains a function specification for the call from the specification web, which we assume to be true (i.e., provable with respect to any function body introduced under the name being called) in this lemma. The precondition for the Hoare triple in the conclusion is the conjunction of \( \gamma(f) \) and \( \theta(P) \). This is satisfied by all states where the function table contains (at least) code for the function being called, and so the operational rule cannot fault when finding the function. Moreover, the environment satisfies the specified precondition for the function body code, and so the code execution cannot fault by assumption. The substitution \( \theta \) handles the locality renaming and parameter substitutions as in the operational rule, and is standard. The postcondition uses \( Q \) from the specification, again renamed by \( \theta \) to avoid local names clashing with other bound names; \( Q \) describes any environment that results from executing the code by the assumption of specification correctness w.r.t our triple interpretation. The return value is assigned to the correct variable, and exists in the postcondition by the syntactic construction of function bodies.

3. **Function return**: Note the function return operational semantic and recall the inference rule.

\[
\begin{align*}
\text{s, FT, g, return } E &\rightarrow s[\text{ret } \leftarrow E], \text{FT, g} \\
\{\Theta_G\} &\text{return } E[\text{ret } = E \land \Theta_G]
\end{align*}
\]

The operational effect is to assign the return expression to the reserved variable name `ret`. This cannot fault, requires no heap, store or function entries and must always terminate. The axiom mirrors the operational rule; states satisfying the precondition are any in which the heap is empty; the resultant state maintains this empty heap, but certainly has a store entry for `ret`. The `ret` variable is reserved, and so cannot appear in program code. Moreover, by the construction of programs, the `return` operational rule is always followed by the conclusion of a function call rule which substitutes `ret` for the variable it will be assigned to.

4. **Document fetch**: Recall the `fetchDocument` operational semantic and inference rule.

\[
\begin{align*}
W(\text{uri}) &= (\#\text{document}^\text{null}[\text{de}]_{\text{id}}, \mathcal{C}) \\
g' &= g \oplus \langle \#\text{document}^\text{null}[\text{de}]_{\text{id}} > G \\
\text{s, FT, g, n := fetchDocument(\text{uri}) } &\rightarrow s[n \leftarrow \text{id}], \text{FT, g'}
\end{align*}
\]
fetchDocument faults only if the given component URI is not present in the abstract web. Otherwise, it requires no heap or store, and results in a heap containing the data available at the URI, with the document element identifier stored within the named variable.

All states satisfying the precondition of the conclusion must satisfy $\omega(\text{uri})$. By the satisfaction relation, this is true iff the URI exists in the abstract web, hence the command cannot fault. The data imported satisfies the schema predicate in the specification by assumption, and contains no clashing identifiers by the definition of $W$. We substitute the assignee name for reserved variable name this to ensure the name does not appear as a program variable. Hence $Q$ describes all environments that result from the command.

5. **Script execution:** Recall the runScript operational semantic and inference rule.

$$
W(\text{uri}) = (\text{doc}, C)
$$

$$
\frac{s, FT, g, \Theta(C, s) \Rightarrow s', FT', g'}{s, FT, g, \text{runScript(\text{uri})} \Rightarrow s', FT', g'}
$$

runScript has a similar operational effect to function calling, passing control to code specified elsewhere. As such, the proof is similar to function calling, except the requirement that a given function be in the environment is replaced by that of a given component being in the abstract web.

Each of our new inference rules is then sound, thus is our logic.

**Lemma 21** (Specification set extension). Let a component be proven correct with respect to its specification using a specification web $W$ and let $W' \supseteq W$. If the elements of $W'$ are positive specifications; and each function specification added by the extension $W' - W$ is mentioned in the function specifications of $W$; then original proof is also correct with respect to $W'$.

**Proof.** Consider a valid proof of a component $u$ with respect to $W$. Extending $W$ could invalidate this proof only if a rule used in the proof became unapplicable by the extension. Examining the rules, this can occur only if

1. The function introduction rule was used for a function name $f$ that has been given a new specification in the extension (and would not have been able to generate a sub-proof for the new specification)

2. A precondition that was shown to be true w.r.t $W$ stopped being demonstrably true w.r.t. $W'$

The first cannot occur by the requirement that any function specification in $W'$ for a name already mentioned in $W$ not be different from one already seen in $W$. For the second, only one assertion has a satisfaction definition that could be altered by the structure of the web: $\omega$, the definition of which an existential check for the existence of a URI. Extending the web could make a previously true usage of $\omega$ untrue only if that usage occurred negatively; the restriction to positive specifications eliminates this case. 

**Lemma 23.** Let $u$ be a component with specification $S$, and let there be a proof of that $u$ satisfies $S$ using a specification web $W$. If the specifications in $W$ are correct, then the abstract web with domain $\text{deps}(u)$ is closed.
Proof. By definition, a closed web is one in which no execution of any script within the web will use resources via \texttt{runScript} and \texttt{fetchDocument} at a URI not present within the web. We assume that there is a proof of \( u \) with specification \( S \); let \( P \) be the precondition of the component code \( S \) specifies. The only two commands that access the web are \texttt{fetchDocument} and \texttt{runScript}. If the code of \( u \) uses neither, any web is closed with respect to \( u \). If it uses these commands, the proof of \( u \) must use the associated inference rules, which both demand states satisfying \( \omega(u') \) for some \( u' \).

No inference rule or axiom introduces assertions of the form \( \omega(u') \). We show that \( P \) must mention \( \omega(u') \) by induction on the structure of proofs. Either \( \omega(u') \) is

1. asserted by \( P \), and are hence in \texttt{deps}
2. part of the postcondition for some component \( v \) that was mashed in via \texttt{runScript}. By the same argument as above, it must have been in the precondition for \( v \), i.e. part of \texttt{deps}(\( v \)), and hence must have also been asserted by \( P \); otherwise the precondition of \( v \) for \texttt{runScript} would not have been satisfiable.

Hence \( P \) mentions every \( u' \) that is used by \texttt{runScript} or \texttt{fetchDocument}, and \texttt{deps}(\( u \)) is closed.

N.B: This is not the same as syntactically finding each use of the web commands, which will over-approximate the used resources (e.g. consider functions that are introduced by never called)

\[ \square \]

**Theorem 25** (Reasonable Web soundness). Let \( W \) be a Reasonable Web and fix some specification web \( W \). If all components within \( W \) have specifications in \( W \) that can be proven w.r.t. \( W \), the execution of any one cannot fault.

**Proof.** From Theorem 19 and Lemma 23, we know that \( W \) has sufficient specifications to prove each component in \( S \), and that if we assume the specifications are correct, each proof is sound and by Definition 1 means the code cannot fault. It remains to show that we need not make this assumption, and that each specification will be checked.

We have to check each function specification, each component code specification, and each component data specification. The later two are checked by the statement of the theorem - we assume each component can be proven with respect to \( W \), which involves finding a proof of the data and code specifications. Function specifications are checked by the function introduction rule. Notice that we cannot call a function until \( \gamma(f) \) is true. The only rule or axiom which introduces \( \gamma(f) \) to be true is function introduction; this rule checks the body of the function being introduced against all the specifications.

Recursion is sound by minor adaptations of standard techniques (e.g. see *Local reasoning for Java* (Parkinson, PhD Thesis, 2005))

\[ \square \]

5 Function bodies

We used several function bodies without giving their details. These are below.

```plaintext
local function getFirstChild(n)
{
    local children;
    local first;

    children = getChildNodes(n);
    first = item(children, 0);

    return first;
};

local function getNodeValue(n)
{
    local first;
    local text;
```
int i;
local last;

first = getFirstChild(n);
i = 0;
text = "";
last = " ";

while (text != last)
{
    last = text;
    text = substringdata(first, 0, i);
    i = i + 1;
}

return text;