Core DOM

A Formal Model of the Document Object Model

Achim D. Brucker Michael Herzberg

March 19, 2025

Department of Computer Science
The University of Sheffield
Sheffield, UK
{a.brucker, msherzberg1}@sheffield.ac.uk

Abstract

In this AFP entry, we formalize the core of the Document Object Model (DOM). At its core, the DOM defines a tree-like data structure for representing documents in general and HTML documents in particular. It is the heart of any modern web browser.

Formalizing the key concepts of the DOM is a prerequisite for the formal reasoning over client-side JavaScript programs and for the analysis of security concepts in modern web browsers.

We present a formalization of the core DOM, with focus on the *node-tree* and the operations defined on node-trees, in Isabelle/HOL. We use the formalization to verify the functional correctness of the most important functions defined in the DOM standard. Moreover, our formalization is 1. *extensible*, i.e., can be extended without the need of re-proving already proven properties and 2. *executable*, i.e., we can generate executable code from our specification.

Keywords: Document Object Model, DOM, Formal Semantics, Isabelle/HOL

Contents

1 Introduction 7

2 Preliminaries 9
- 2.1 Hiding Type Variables (Hiding_Type_Variables) . . . 9
- 2.2 The Heap Error Monad (Heap_Error_Monad) . . . 10

3 References and Pointers 23
- 3.1 References (Ref) . . . 23
- 3.2 Object (ObjectPointer) . . . 23
- 3.3 Node (NodePointer) . . . 24
- 3.4 Element (ElementPointer) . . . 25
- 3.5 CharacterData (CharacterDataPointer) . . . 27
- 3.6 Document (DocumentPointer) . . . 30
- 3.7 ShadowRoot (ShadowRootPointer) . . . 32

4 Classes 35
- 4.1 The Class Infrastructure (BaseClass) . . . 35
- 4.2 Object (ObjectClass) . . . 35
- 4.3 Node (NodeClass) . . . 38
- 4.4 Element (ElementClass) . . . 41
- 4.5 CharacterData (CharacterDataClass) . . . 45
- 4.6 Document (DocumentClass) . . . 50

5 Monadic Object Constructors and Accessors 57
- 5.1 The Monad Infrastructure (BaseMonad) . . . 57
- 5.2 Object (ObjectMonad) . . . 60
- 5.3 Node (NodeMonad) . . . 63
- 5.4 Element (ElementMonad) . . . 65
- 5.5 CharacterData (CharacterDataMonad) . . . 70
- 5.6 Document (DocumentMonad) . . . 75

6 The Core DOM 83
- 6.1 Basic Data Types (Core_DOM_Basic_Datatypes) . . . 83
- 6.2 Querying and Modifying the DOM (Core_DOM_Functions) . . . 83
- 6.3 Wellformedness (Core_DOM_Heap_WF) . . . 130
- 6.4 The Core DOM (Core_DOM) . . . 158

7 Test Suite 159
- 7.1 Common Test Setup (Core_DOM_BaseTest) . . . 159
- 7.2 Testing Document_adoptNode (Document_adoptNode) . . . 163
- 7.3 Testing Document_getElementById (Document_getElementById) . . . 164
- 7.4 Testing Node_insertBefore (Node_insertBefore) . . . 168
- 7.5 Testing Node_removeChild (Node_removeChild) . . . 170
- 7.6 Core DOM Test Cases (Core_DOM_Tests) . . . 172

1 Introduction

In a world in which more and more applications are offered as services on the internet, web browsers start to take on a similarly central role in our daily IT infrastructure as operating systems. Thus, web browsers should be developed as rigidly and formally as operating systems. While formal methods are a well-established technique in the development of operating systems (see, e. g., Klein [12] for an overview of formal verification of operating systems), there are few proposals for improving the development of web browsers using formal approaches [2, 9, 10, 13].

As a first step towards a verified client-side web application stack, we model and formally verify the Document Object Model (DOM) in Isabelle/HOL. The DOM [14, 15] is *the* central data structure of all modern web browsers. At its core, the Document Object Model (DOM), defines a tree-like data structure for representing documents in general and HTML documents in particular. Thus, the correctness of a DOM implementation is crucial for ensuring that a web browser displays web pages correctly. Moreover, the DOM is the core data structure underlying client-side JavaScript programs, i. e., client-side JavaScript programs are mostly programs that read, write, and update the DOM.

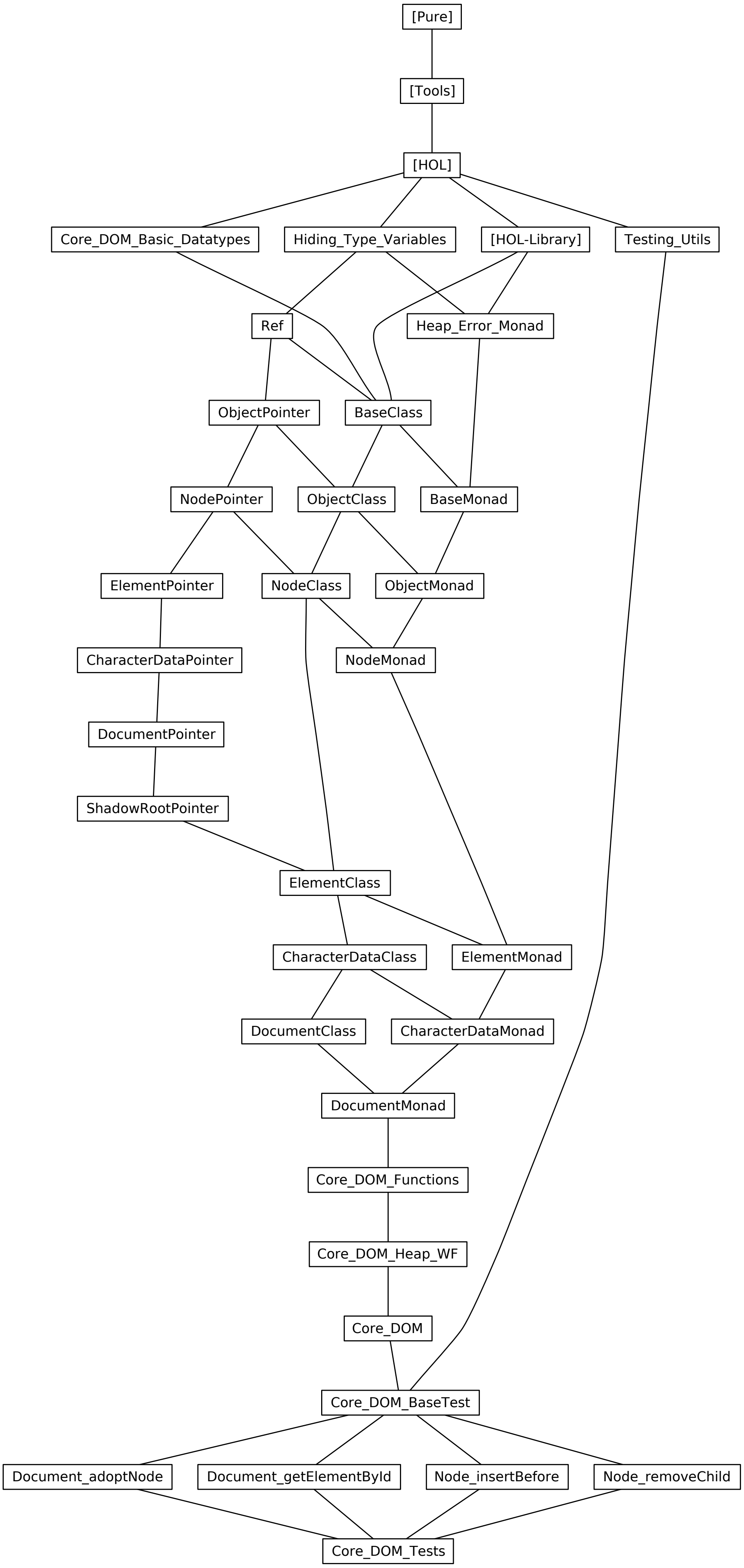
In more detail, we formalize the core DOM as a shallow embedding [11] in Isabelle/HOL. Our formalization is based on a typed data model for the *node-tree*, i. e., a data structure for representing XML-like documents in a tree structure. Furthermore, we formalize a typed heap for storing (partial) node-trees together with the necessary consistency constraints. Finally, we formalize the operations (as described in the DOM standard [15]) on this heap that allow manipulating node-trees.

Our machine-checked formalization of the DOM node tree [15] has the following desirable properties:

- It provides a *consistency guarantee.* Since all definitions in our formal semantics are conservative and all rules are derived, the logical consistency of the DOM node-tree is reduced to the consistency of HOL.
- It serves as a *technical basis for a proof system.* Based on the derived rules and specific setup of proof tactics over node-trees, our formalization provides a generic proof environment for the verification of programs manipulating node-trees.
- It is *executable*, which allows to validate its compliance to the standard by evaluating the compliance test suite on the formal model and
- It is *extensible* in the sense of [3, 7], i. e., properties proven over the core DOM do not need to be re-proven for object-oriented extensions such as the HTML document model.

The rest of this document is automatically generated from the formalization in Isabelle/HOL, i.e., all content is checked by Isabelle.[1] The structure follows the theory dependencies (see Figure 1.1): we start with introducing the technical preliminaries of our formalization (chapter 2). Next, we introduce the concepts of pointers (chapter 3) and classes (chapter 4), i.e., the core object-oriented datatypes of the DOM. On top of this data model, we define the functional behavior of the DOM classes, i.e., their methods (chapter 5). In chapter 6, we introduce the formalization of the functionality of the core DOM, i.e., the *main entry point for users* that want to use this AFP entry. Finally, we formalize the relevant compliance test cases in chapter 7.

[1]For a brief overview of the work, we refer the reader to [4].


[Pure]
[Tools]
[HOL]
Core_DOM_Basic_Datatypes
Hiding_Type_Variables
[HOL-Library]
Testing_Utils
Ref
Heap_Error_Monad
ObjectPointer
BaseClass
NodePointer
ObjectClass
BaseMonad
ElementPointer
NodeClass
ObjectMonad
CharacterDataPointer
NodeMonad
DocumentPointer
ShadowRootPointer
ElementClass
CharacterDataClass
ElementMonad
DocumentClass
CharacterDataMonad
DocumentMonad
Core_DOM_Functions
Core_DOM_Heap_WF
Core_DOM
Core_DOM_BaseTest
Document_adoptNode
Document_getElementById
Node_insertBefore
Node_removeChild
Core_DOM_Tests

2 Preliminaries

In this chapter, we introduce the technical preliminaries of our formalization of the core DOM, namely a mechanism for hiding type variables and the heap error monad.

2.1 Hiding Type Variables (Hiding_Type_Variables)

This theory[1] implements a mechanism for declaring default type variables for data types. This comes handy for complex data types with many type variables.

```
theory
  "Hiding_Type_Variables"
imports
  Main
keywords
    "register_default_tvars"
    "update_default_tvars_mode"::thy_decl
begin⟨ML⟩⟨ML⟩⟨ML⟩
```

2.1.1 Introduction

When modelling object-oriented data models in HOL with the goal of preserving *extensibility* (e.g., as described in [3, 7]) one needs to define type constructors with a large number of type variables. This can reduce the readability of the overall formalization. Thus, we use a short-hand notation in cases were the names of the type variables are known from the context. In more detail, this theory sets up both configurable print and parse translations that allows for replacing *all* type variables by `(_)`, e.g., a five-ary constructor `('a, 'b, 'c, 'd, 'e) hide_tvar_foo` can be shorted to `(_) hide_tvar_foo`. The use of this shorthand in output (printing) and input (parsing) is, on a per-type basis, user-configurable using the top-level commands `register_default_tvars` (for registering the names of the default type variables and the print/parse mode) and `update_default_tvars_mode` (for changing the print/parse mode dynamically).

The input also supports short-hands for declaring default sorts (e.g., `(_::linorder)` specifies that all default variables need to be instances of the sort (type class) `linorder` and short-hands of overriding a suffice (or prefix) of the default type variables. For example, `('state) hide_tvar_foo _.` is a short-hand for `('a, 'b, 'c, 'd, 'state) hide_tvar_foo`. In this document, we omit the implementation details (we refer the interested reader to theory file) and continue directly with a few examples.

2.1.2 Example

Given the following type definition:

```
datatype ('a, 'b) hide_tvar_foobar = hide_tvar_foo 'a | hide_tvar_bar 'b
type_synonym ('a, 'b, 'c, 'd) hide_tvar_baz = "('a+'b, 'a × 'b) hide_tvar_foobar"
```

We can register default values for the type variables for the abstract data type as well as the type synonym:

```
register_default_tvars  "('alpha, 'beta) hide_tvar_foobar" (print_all,parse)
register_default_tvars  "('alpha, 'beta, 'gamma, 'delta) hide_tvar_baz" (print_all,parse)
```

This allows us to write

```
definition hide_tvar_f::"(_) hide_tvar_foobar ⇒ (_) hide_tvar_foobar ⇒ (_) hide_tvar_foobar"
     where "hide_tvar_f a b = a"
definition hide_tvar_g::"(_) hide_tvar_baz ⇒ (_) hide_tvar_baz ⇒ (_) hide_tvar_baz"
  where "hide_tvar_g a b = a"
```

[1]This theory can be used "stand-alone," i.e., this theory is not specific to the DOM formalization. The latest version is part of the "Isabelle Hacks" repository: https://git.logicalhacking.com/adbrucker/isabelle-hacks/.

Instead of specifying the type variables explicitely. This makes, in particular for type constructors with a large number of type variables, definitions much more concise. This syntax is also used in the output of antiquotations, e.g., *(x::(_) hide_tvar_baz ⇒ (_) hide_tvar_baz ⇒ (_) hide_tvar_baz) = hide_tvar_g*. Both the print translation and the parse translation can be disabled for each type individually:

update_default_tvars_mode *"_ hide_tvar_foobar" (noprint,noparse)*
update_default_tvars_mode *"_ hide_tvar_foobar" (noprint,noparse)*

Now, Isabelle's interactive output and the antiquotations will show all type variables, e.g., *(x::('a + 'b, 'a × 'b) hide_tvar_foobar ⇒ ('a + 'b, 'a × 'b) hide_tvar_foobar ⇒ ('a + 'b, 'a × 'b) hide_tvar_foobar) = hide_tvar_g*.

end

2.2 The Heap Error Monad (Heap_Error_Monad)

In this theory, we define a heap and error monad for modeling exceptions. This allows us to define composite methods similar to stateful programming in Haskell, but also to stay close to the official DOM specification.

```
theory
  Heap_Error_Monad
  imports
    Hiding_Type_Variables
    "HOL-Library.Monad_Syntax"
begin
```

2.2.1 The Program Data Type

```
datatype ('heap, 'e, 'result) prog = Prog (the_prog: "'heap ⇒ 'e + 'result × 'heap")
register_default_tvars "('heap, 'e, 'result) prog" (print, parse)
```

2.2.2 Basic Functions

```
definition
  bind :: "(_, 'result) prog ⇒ ('result ⇒ (_, 'result2) prog) ⇒ (_, 'result2) prog"
  where
    "bind f g = Prog (λh. (case (the_prog f) h of Inr (x, h') ⇒ (the_prog (g x)) h'
                                                    | Inl exception ⇒ Inl exception))"

adhoc_overloading Monad_Syntax.bind ⇌ bind

definition
  execute :: "'heap ⇒ ('heap, 'e, 'result) prog ⇒ ('e + 'result × 'heap)"
  (‹((_)/ ⊢ (_))› [51, 52] 55)
  where
    "execute h p = (the_prog p) h"

definition
  returns_result :: "'heap ⇒ ('heap, 'e, 'result) prog ⇒ 'result ⇒ bool"
  (‹((_)/ ⊢ (_)/ →ᵣ (_))› [60, 35, 61] 65)
  where
    "returns_result h p r ⟷ (case h ⊢ p of Inr (r', _) ⇒ r = r' | Inl _ ⇒ False)"

fun select_result (‹|(_)|ᵣ›)
  where
    "select_result (Inr (r, _)) = r"
  | "select_result (Inl _) = undefined"

lemma returns_result_eq [elim]: "h ⊢ f →ᵣ y ⟹ h ⊢ f →ᵣ y' ⟹ y = y'"
  ⟨proof⟩

definition
  returns_heap :: "'heap ⇒ ('heap, 'e, 'result) prog ⇒ 'heap ⇒ bool"
  (‹((_)/ ⊢ (_)/ →ₕ (_))› [60, 35, 61] 65)
  where
    "returns_heap h p h' ⟷ (case h ⊢ p of Inr (_ , h'') ⇒ h' = h'' | Inl _ ⇒ False)"

fun select_heap (<|(_)|h>)
  where
    "select_heap (Inr ( _, h)) = h"
  | "select_heap (Inl _) = undefined"

lemma returns_heap_eq [elim]: "h ⊢ f →h h' ⟹ h ⊢ f →h h'' ⟹ h' = h''"
  ⟨proof⟩

definition
  returns_result_heap :: "'heap ⇒ ('heap, 'e, 'result) prog ⇒ 'result ⇒ 'heap ⇒ bool"
  (<((_)/ ⊢ (_)/ →r (_) →h (_))> [60, 35, 61, 62] 65)
  where
    "returns_result_heap h p r h' ⟷ h ⊢ p →r r ∧ h ⊢ p →h h'"

lemma return_result_heap_code [code]:
  "returns_result_heap h p r h' ⟷ (case h ⊢ p of Inr (r', h'') ⇒ r = r' ∧ h' = h'' | Inl _ ⇒ False)"
  ⟨proof⟩

fun select_result_heap (<|(_)|rh>)
  where
    "select_result_heap (Inr (r, h)) = (r, h)"
  | "select_result_heap (Inl _) = undefined"

definition
  returns_error :: "'heap ⇒ ('heap, 'e, 'result) prog ⇒ 'e ⇒ bool"
  (<((_)/ ⊢ (_)/ →e (_))> [60, 35, 61] 65)
  where
    "returns_error h p e = (case h ⊢ p of Inr _ ⇒ False | Inl e' ⇒ e = e')"

definition is_OK :: "'heap ⇒ ('heap, 'e, 'result) prog ⇒ bool" (<((_)/ ⊢ ok (_))> [75, 75])
  where
    "is_OK h p = (case h ⊢ p of Inr _ ⇒ True | Inl _ ⇒ False)"

lemma is_OK_returns_result_I [intro]: "h ⊢ f →r y ⟹ h ⊢ ok f"
  ⟨proof⟩

lemma is_OK_returns_result_E [elim]:
  assumes "h ⊢ ok f"
  obtains x where "h ⊢ f →r x"
  ⟨proof⟩

lemma is_OK_returns_heap_I [intro]: "h ⊢ f →h h' ⟹ h ⊢ ok f"
  ⟨proof⟩

lemma is_OK_returns_heap_E [elim]:
  assumes "h ⊢ ok f"
  obtains h' where "h ⊢ f →h h'"
  ⟨proof⟩

lemma select_result_I:
  assumes "h ⊢ ok f"
    and "⋀x. h ⊢ f →r x ⟹ P x"
  shows "P |h ⊢ f|r"
  ⟨proof⟩

lemma select_result_I2 [simp]:
  assumes "h ⊢ f →r x"
  shows "|h ⊢ f|r = x"
  ⟨proof⟩
```

lemma returns_result_select_result [simp]:
  assumes "h $\vdash$ ok f"
  shows "h $\vdash$ f $\rightarrow_r$ |h $\vdash$ f|$_r$"
  ⟨*proof*⟩

lemma select_result_E:
  assumes "P |h $\vdash$ f|$_r$" **and** "h $\vdash$ ok f"
  obtains x **where** "h $\vdash$ f $\rightarrow_r$ x" **and** "P x"
  ⟨*proof*⟩

lemma select_result_eq: "($\bigwedge$x .h $\vdash$ f $\rightarrow_r$ x = h' $\vdash$ f $\rightarrow_r$ x) $\Longrightarrow$ |h $\vdash$ f|$_r$ = |h' $\vdash$ f|$_r$"
  ⟨*proof*⟩

definition error :: "'e $\Rightarrow$ ('heap, 'e, 'result) prog"
  where
    "error exception = Prog (λh. Inl exception)"

lemma error_bind [iff]: "(error e $\gg=$ g) = error e"
  ⟨*proof*⟩

lemma error_returns_result [simp]: "$\neg$ (h $\vdash$ error e $\rightarrow_r$ y)"
  ⟨*proof*⟩

lemma error_returns_heap [simp]: "$\neg$ (h $\vdash$ error e $\rightarrow_h$ h')"
  ⟨*proof*⟩

lemma error_returns_error [simp]: "h $\vdash$ error e $\rightarrow_e$ e"
  ⟨*proof*⟩

definition return :: "'result $\Rightarrow$ ('heap, 'e, 'result) prog"
  where
    "return result = Prog (λh. Inr (result, h))"

lemma return_ok [simp]: "h $\vdash$ ok (return x)"
  ⟨*proof*⟩

lemma return_bind [iff]: "(return x $\gg=$ g) = g x"
  ⟨*proof*⟩

lemma return_id [simp]: "f $\gg=$ return = f"
  ⟨*proof*⟩

lemma return_returns_result [iff]: "(h $\vdash$ return x $\rightarrow_r$ y) = (x = y)"
  ⟨*proof*⟩

lemma return_returns_heap [iff]: "(h $\vdash$ return x $\rightarrow_h$ h') = (h = h')"
  ⟨*proof*⟩

lemma return_returns_error [iff]: "$\neg$ h $\vdash$ return x $\rightarrow_e$ e"
  ⟨*proof*⟩

definition noop :: "('heap, 'e, unit) prog"
  where
    "noop = return ()"

lemma noop_returns_heap [simp]: "h $\vdash$ noop $\rightarrow_h$ h' $\longleftrightarrow$ h = h'"
  ⟨*proof*⟩

definition get_heap :: "('heap, 'e, 'heap) prog"
  where
    "get_heap = Prog (λh. h $\vdash$ return h)"

lemma get_heap_ok [simp]: "h $\vdash$ ok (get_heap)"

⟨*proof*⟩

lemma *get_heap_returns_result [simp]: "(h ⊢ get_heap ⪢= (λh'. f h') $\rightarrow_r$ x) = (h ⊢ f h $\rightarrow_r$ x)"*
⟨*proof*⟩

lemma *get_heap_returns_heap [simp]: "(h ⊢ get_heap ⪢= (λh'. f h') $\rightarrow_h$ h'') = (h ⊢ f h $\rightarrow_h$ h'')"*
⟨*proof*⟩

lemma *get_heap_is_OK [simp]: "(h ⊢ ok (get_heap ⪢= (λh'. f h'))) = (h ⊢ ok (f h))"*
⟨*proof*⟩

lemma *get_heap_E [elim]: "(h ⊢ get_heap $\rightarrow_r$ x) ⟹ x = h"*
⟨*proof*⟩

definition *return_heap :: "'heap ⇒ ('heap, 'e, unit) prog"*
where
"return_heap h = Prog (λ_. h ⊢ return ())"

lemma *return_heap_E [iff]: "(h ⊢ return_heap h' $\rightarrow_h$ h'') = (h'' = h')"*
⟨*proof*⟩

lemma *return_heap_returns_result [simp]: "h ⊢ return_heap h' $\rightarrow_r$ ()"*
⟨*proof*⟩

2.2.3 Pure Heaps

definition *pure :: "('heap, 'e, 'result) prog ⇒ 'heap ⇒ bool"*
where *"pure f h ⟷ h ⊢ ok f ⟶ h ⊢ f $\rightarrow_h$ h"*

lemma *return_pure [simp]: "pure (return x) h"*
⟨*proof*⟩

lemma *error_pure [simp]: "pure (error e) h"*
⟨*proof*⟩

lemma *noop_pure [simp]: "pure (noop) h"*
⟨*proof*⟩

lemma *get_pure [simp]: "pure get_heap h"*
⟨*proof*⟩

lemma *pure_returns_heap_eq:*
"h ⊢ f $\rightarrow_h$ h' ⟹ pure f h ⟹ h = h'"
⟨*proof*⟩

lemma *pure_eq_iff:*
"(∀h' x. h ⊢ f $\rightarrow_r$ x ⟶ h ⊢ f $\rightarrow_h$ h' ⟶ h = h') ⟷ pure f h"
⟨*proof*⟩

2.2.4 Bind

lemma *bind_assoc [simp]:*
"((bind f g) ⪢= h) = (f ⪢= (λx. (g x ⪢= h)))"
⟨*proof*⟩

lemma *bind_returns_result_E:*
assumes *"h ⊢ f ⪢= g $\rightarrow_r$ y"*
obtains *x h'* **where** *"h ⊢ f $\rightarrow_r$ x"* **and** *"h ⊢ f $\rightarrow_h$ h'"* **and** *"h' ⊢ g x $\rightarrow_r$ y"*
⟨*proof*⟩

lemma *bind_returns_result_E2:*
assumes *"h ⊢ f ⪢= g $\rightarrow_r$ y"* **and** *"pure f h"*
obtains *x* **where** *"h ⊢ f $\rightarrow_r$ x"* **and** *"h ⊢ g x $\rightarrow_r$ y"*

⟨*proof*⟩

lemma *bind_returns_result_E3:*
assumes *"h ⊢ f ≫= g $\rightarrow_r$ y"* **and** *"h ⊢ f $\rightarrow_r$ x"* **and** *"pure f h"*
shows *"h ⊢ g x $\rightarrow_r$ y"*
⟨*proof*⟩

lemma *bind_returns_result_E4:*
assumes *"h ⊢ f ≫= g $\rightarrow_r$ y"* **and** *"h ⊢ f $\rightarrow_r$ x"*
obtains *h'* **where** *"h ⊢ f $\rightarrow_h$ h'"* **and** *"h' ⊢ g x $\rightarrow_r$ y"*
⟨*proof*⟩

lemma *bind_returns_heap_E:*
assumes *"h ⊢ f ≫= g $\rightarrow_h$ h''"*
obtains *x h'* **where** *"h ⊢ f $\rightarrow_r$ x"* **and** *"h ⊢ f $\rightarrow_h$ h'"* **and** *"h' ⊢ g x $\rightarrow_h$ h''"*
⟨*proof*⟩

lemma *bind_returns_heap_E2 [elim]:*
assumes *"h ⊢ f ≫= g $\rightarrow_h$ h'"* **and** *"pure f h"*
obtains *x* **where** *"h ⊢ f $\rightarrow_r$ x"* **and** *"h ⊢ g x $\rightarrow_h$ h'"*
⟨*proof*⟩

lemma *bind_returns_heap_E3 [elim]:*
assumes *"h ⊢ f ≫= g $\rightarrow_h$ h'"* **and** *"h ⊢ f $\rightarrow_r$ x"* **and** *"pure f h"*
shows *"h ⊢ g x $\rightarrow_h$ h'"*
⟨*proof*⟩

lemma *bind_returns_heap_E4:*
assumes *"h ⊢ f ≫= g $\rightarrow_h$ h''"* **and** *"h ⊢ f $\rightarrow_h$ h'"*
obtains *x* **where** *"h ⊢ f $\rightarrow_r$ x"* **and** *"h' ⊢ g x $\rightarrow_h$ h''"*
⟨*proof*⟩

lemma *bind_returns_error_I [intro]:*
assumes *"h ⊢ f $\rightarrow_e$ e"*
shows *"h ⊢ f ≫= g $\rightarrow_e$ e"*
⟨*proof*⟩

lemma *bind_returns_error_I3:*
assumes *"h ⊢ f $\rightarrow_r$ x"* **and** *"h ⊢ f $\rightarrow_h$ h'"* **and** *"h' ⊢ g x $\rightarrow_e$ e"*
shows *"h ⊢ f ≫= g $\rightarrow_e$ e"*
⟨*proof*⟩

lemma *bind_returns_error_I2 [intro]:*
assumes *"pure f h"* **and** *"h ⊢ f $\rightarrow_r$ x"* **and** *"h ⊢ g x $\rightarrow_e$ e"*
shows *"h ⊢ f ≫= g $\rightarrow_e$ e"*
⟨*proof*⟩

lemma *bind_is_OK_E [elim]:*
assumes *"h ⊢ ok (f ≫= g)"*
obtains *x h'* **where** *"h ⊢ f $\rightarrow_r$ x"* **and** *"h ⊢ f $\rightarrow_h$ h'"* **and** *"h' ⊢ ok (g x)"*
⟨*proof*⟩

lemma *bind_is_OK_E2:*
assumes *"h ⊢ ok (f ≫= g)"* **and** *"h ⊢ f $\rightarrow_r$ x"*
obtains *h'* **where** *"h ⊢ f $\rightarrow_h$ h'"* **and** *"h' ⊢ ok (g x)"*
⟨*proof*⟩

lemma *bind_returns_result_I [intro]:*
assumes *"h ⊢ f $\rightarrow_r$ x"* **and** *"h ⊢ f $\rightarrow_h$ h'"* **and** *"h' ⊢ g x $\rightarrow_r$ y"*
shows *"h ⊢ f ≫= g $\rightarrow_r$ y"*
⟨*proof*⟩

lemma *bind_pure_returns_result_I [intro]:*

```
  assumes "pure f h" and "h ⊢ f →r x" and "h ⊢ g x →r y"
  shows "h ⊢ f ⤜ g →r y"
  ⟨proof⟩

lemma bind_pure_returns_result_I2 [intro]:
  assumes "pure f h" and "h ⊢ ok f" and "⋀x. h ⊢ f →r x ⟹ h ⊢ g x →r y"
  shows "h ⊢ f ⤜ g →r y"
  ⟨proof⟩

lemma bind_returns_heap_I [intro]:
  assumes "h ⊢ f →r x" and "h ⊢ f →h h'" and "h' ⊢ g x →h h''"
  shows "h ⊢ f ⤜ g →h h''"
  ⟨proof⟩

lemma bind_returns_heap_I2 [intro]:
  assumes "h ⊢ f →h h'" and "⋀x. h ⊢ f →r x ⟹ h' ⊢ g x →h h''"
  shows "h ⊢ f ⤜ g →h h''"
  ⟨proof⟩

lemma bind_is_OK_I [intro]:
  assumes "h ⊢ f →r x" and "h ⊢ f →h h'" and "h' ⊢ ok (g x)"
  shows "h ⊢ ok (f ⤜ g)"
  ⟨proof⟩

lemma bind_is_OK_I2 [intro]:
  assumes "h ⊢ ok f" and "⋀x h'. h ⊢ f →r x ⟹ h ⊢ f →h h' ⟹ h' ⊢ ok (g x)"
  shows "h ⊢ ok (f ⤜ g)"
  ⟨proof⟩

lemma bind_is_OK_pure_I [intro]:
  assumes "pure f h" and "h ⊢ ok f" and "⋀x. h ⊢ f →r x ⟹ h ⊢ ok (g x)"
  shows "h ⊢ ok (f ⤜ g)"
  ⟨proof⟩

lemma bind_pure_I:
  assumes "pure f h" and "⋀x. h ⊢ f →r x ⟹ pure (g x) h"
  shows "pure (f ⤜ g) h"
  ⟨proof⟩

lemma pure_pure:
  assumes "h ⊢ ok f" and "pure f h"
  shows "h ⊢ f →h h"
  ⟨proof⟩

lemma bind_returns_error_eq:
  assumes "h ⊢ f →e e"
    and "h ⊢ g →e e"
  shows "h ⊢ f = h ⊢ g"
  ⟨proof⟩
```

2.2.5 Map

```
fun map_M :: "('x ⇒ ('heap, 'e, 'result) prog) ⇒ 'x list ⇒ ('heap, 'e, 'result list) prog"
  where
    "map_M f [] = return []"
  | "map_M f (x#xs) = do {
      y ← f x;
      ys ← map_M f xs;
      return (y # ys)
    }"

lemma map_M_ok_I [intro]:
  "(⋀x. x ∈ set xs ⟹ h ⊢ ok (f x)) ⟹ (⋀x. x ∈ set xs ⟹ pure (f x) h) ⟹ h ⊢ ok (map_M f xs)"
```

⟨*proof*⟩

lemma *map_M_pure_I : "⋀h. (⋀x. x ∈ set xs ⟹ pure (f x) h) ⟹ pure (map_M f xs) h"*
⟨*proof*⟩

lemma *map_M_pure_E :*
assumes *"h ⊢ map_M g xs* $\to_r$ *ys"* **and** *"x ∈ set xs"* **and** *"⋀x h. x ∈ set xs ⟹ pure (g x) h"*
obtains *y* **where** *"h ⊢ g x* $\to_r$ *y"* **and** *"y ∈ set ys"*
⟨*proof*⟩

lemma *map_M_pure_E2:*
assumes *"h ⊢ map_M g xs* $\to_r$ *ys"* **and** *"y ∈ set ys"* **and** *"⋀x h. x ∈ set xs ⟹ pure (g x) h"*
obtains *x* **where** *"h ⊢ g x* $\to_r$ *y"* **and** *"x ∈ set xs"*
⟨*proof*⟩

2.2.6 Forall

```
fun forall_M :: "('y ⇒ ('heap, 'e, 'result) prog) ⇒ 'y list ⇒ ('heap, 'e, unit) prog"
  where
    "forall_M P [] = return ()"
  | "forall_M P (x # xs) = do {
      P x;
      forall_M P xs
    }"
```

lemma *pure_forall_M_I: "(⋀x. x ∈ set xs ⟹ pure (P x) h) ⟹ pure (forall_M P xs) h"*
⟨*proof*⟩

2.2.7 Fold

```
fun fold_M :: "('result ⇒ 'y ⇒ ('heap, 'e, 'result) prog) ⇒ 'result ⇒ 'y list
  ⇒ ('heap, 'e, 'result) prog"
  where
    "fold_M f d [] = return d" |
    "fold_M f d (x # xs) = do { y ← f d x; fold_M f y xs }"
```

lemma *fold_M_pure_I : "(⋀d x. pure (f d x) h) ⟹ (⋀d. pure (fold_M f d xs) h)"*
⟨*proof*⟩

2.2.8 Filter

```
fun filter_M :: "('x ⇒ ('heap, 'e, bool) prog) ⇒ 'x list ⇒ ('heap, 'e, 'x list) prog"
  where
    "filter_M P [] = return []"
  | "filter_M P (x#xs) = do {
      p ← P x;
      ys ← filter_M P xs;
      return (if p then x # ys else ys)
    }"
```

lemma *filter_M_pure_I [intro]: "(⋀x. x ∈ set xs ⟹ pure (P x) h) ⟹ pure (filter_M P xs)h"*
⟨*proof*⟩

lemma *filter_M_is_OK_I [intro]:*
"(⋀x. x ∈ set xs ⟹ h ⊢ ok (P x)) ⟹ (⋀x. x ∈ set xs ⟹ pure (P x) h) ⟹ h ⊢ ok (filter_M P xs)"
⟨*proof*⟩

lemma *filter_M_not_more_elements:*
assumes *"h ⊢ filter_M P xs* $\to_r$ *ys"* **and** *"⋀x. x ∈ set xs ⟹ pure (P x) h"* **and** *"x ∈ set ys"*
shows *"x ∈ set xs"*
⟨*proof*⟩

```
lemma filter_M_in_result_if_ok:
  assumes "h ⊢ filter_M P xs →r ys" and "⋀h x. x ∈ set xs ⟹ pure (P x) h" and "x ∈ set xs" and
    "h ⊢ P x →r True"
  shows "x ∈ set ys"
  ⟨proof⟩

lemma filter_M_holds_for_result:
  assumes "h ⊢ filter_M P xs →r ys" and "x ∈ set ys" and "⋀x h. x ∈ set xs ⟹ pure (P x) h"
  shows "h ⊢ P x →r True"
  ⟨proof⟩

lemma filter_M_empty_I:
  assumes "⋀x. pure (P x) h"
    and "∀x ∈ set xs. h ⊢ P x →r False"
  shows "h ⊢ filter_M P xs →r []"
  ⟨proof⟩

lemma filter_M_subset_2: "h ⊢ filter_M P xs →r ys ⟹ h' ⊢ filter_M P xs →r ys'
                          ⟹ (⋀x. pure (P x) h) ⟹ (⋀x. pure (P x) h')
                          ⟹ (∀b. ∀x ∈ set xs. h ⊢ P x →r True ⟶ h' ⊢ P x →r b ⟶ b)
                          ⟹ set ys ⊆ set ys'"
⟨proof⟩

lemma filter_M_subset: "h ⊢ filter_M P xs →r ys ⟹ set ys ⊆ set xs"
  ⟨proof⟩

lemma filter_M_distinct: "h ⊢ filter_M P xs →r ys ⟹ distinct xs ⟹ distinct ys"
  ⟨proof⟩

lemma filter_M_filter: "h ⊢ filter_M P xs →r ys ⟹ (⋀x. x ∈ set xs ⟹ pure (P x) h)
                       ⟹ (∀x ∈ set xs. h ⊢ ok P x) ∧ ys = filter (λx. |h ⊢ P x|r) xs"
  ⟨proof⟩

lemma filter_M_filter2: "(⋀x. x ∈ set xs ⟹ pure (P x) h ∧  h ⊢ ok P x)
                       ⟹ filter (λx. |h ⊢ P x|r) xs = ys ⟹ h ⊢ filter_M P xs →r ys"
  ⟨proof⟩

lemma filter_ex1: "∃!x ∈ set xs. P x ⟹ P x ⟹ x ∈ set xs ⟹ distinct xs
                  ⟹ filter P xs = [x]"
  ⟨proof⟩

lemma filter_M_ex1:
  assumes "h ⊢ filter_M P xs →r ys"
    and "x ∈ set xs"
    and "∃!x ∈ set xs. h ⊢ P x →r True"
    and "⋀x. x ∈ set xs ⟹ pure (P x) h"
    and "distinct xs"
    and "h ⊢ P x →r True"
  shows "ys = [x]"
⟨proof⟩

lemma filter_M_eq:
  assumes "⋀x. pure (P x) h" and "⋀x. pure (P x) h'"
    and "⋀b x. x ∈ set xs ⟹ h ⊢ P x →r b = h' ⊢ P x →r b"
  shows "h ⊢ filter_M P xs →r ys ⟷ h' ⊢ filter_M P xs →r ys"
  ⟨proof⟩
```

2.2.9 Map Filter

```
definition map_filter_M :: "('x ⇒ ('heap, 'e, 'y option) prog) ⇒ 'x list
  ⇒ ('heap, 'e, 'y list) prog"
  where
    "map_filter_M f xs = do {
```

```
    ys_opts ← map_M f xs;
    ys_no_opts ← filter_M (λx. return (x ≠ None)) ys_opts;
    map_M (λx. return (the x)) ys_no_opts
  }"
```

lemma *map_filter_M_pure:* "(⋀x h. x ∈ set xs ⟹ pure (f x) h) ⟹ pure (map_filter_M f xs) h"
  ⟨*proof*⟩

lemma *map_filter_M_pure_E:*
  assumes "h ⊢ (map_filter_M::('x ⇒ ('heap, 'e, 'y option) prog) ⇒ 'x list
  ⇒ ('heap, 'e, 'y list) prog) f xs $\rightarrow_r$ ys" **and** "y ∈ set ys" **and** "⋀x h. x ∈ set xs ⟹ pure (f x) h"
  obtains x **where** "h ⊢ f x $\rightarrow_r$ Some y" **and** "x ∈ set xs"
⟨*proof*⟩

2.2.10 Iterate

fun *iterate_M* :: "('heap, 'e, 'result) prog list ⇒ ('heap, 'e, 'result) prog"
  where
    "iterate_M [] = return undefined"
  | "iterate_M (x # xs) = x ⤜ (λ_. iterate_M xs)"

lemma *iterate_M_concat:*
  assumes "h ⊢ iterate_M xs $\rightarrow_h$ h'"
    and "h' ⊢ iterate_M ys $\rightarrow_h$ h''"
  shows "h ⊢ iterate_M (xs @ ys) $\rightarrow_h$ h''"
  ⟨*proof*⟩

2.2.11 Miscellaneous Rules

lemma *execute_bind_simp:*
  assumes "h ⊢ f $\rightarrow_r$ x" **and** "h ⊢ f $\rightarrow_h$ h'"
  shows "h ⊢ f ⤜ g = h' ⊢ g x"
  ⟨*proof*⟩

lemma *bind_cong [fundef_cong]:*
  fixes f1 f2 :: "('heap, 'e, 'result) prog"
    and g1 g2 :: "'result ⇒ ('heap, 'e, 'result2) prog"
  assumes "h ⊢ f1 = h ⊢ f2"
    and "⋀y h'. h ⊢ f1 $\rightarrow_r$ y ⟹ h ⊢ f1 $\rightarrow_h$ h' ⟹ h' ⊢ g1 y = h' ⊢ g2 y"
  shows "h ⊢ (f1 ⤜ g1) = h ⊢ (f2 ⤜ g2)"
  ⟨*proof*⟩

lemma *bind_cong_2:*
  assumes "pure f h" **and** "pure f h'"
    and "⋀x. h ⊢ f $\rightarrow_r$ x = h' ⊢ f $\rightarrow_r$ x"
    and "⋀x. h ⊢ f $\rightarrow_r$ x ⟹ h ⊢ g x $\rightarrow_r$ y = h' ⊢ g x $\rightarrow_r$ y'"
  shows "h ⊢ f ⤜ g $\rightarrow_r$ y = h' ⊢ f ⤜ g $\rightarrow_r$ y'"
  ⟨*proof*⟩

lemma *bind_case_cong [fundef_cong]:*
  assumes "x = x'" **and** "⋀a. x = Some a ⟹ f a h = f' a h"
  shows "(case x of Some a ⇒ f a | None ⇒ g) h = (case x' of Some a ⇒ f' a | None ⇒ g) h"
  ⟨*proof*⟩

2.2.12 Reasoning About Reads and Writes

definition *preserved* :: "('heap, 'e, 'result) prog ⇒ 'heap ⇒ 'heap ⇒ bool"
  where
    "preserved f h h' ⟷ (∀x. h ⊢ f $\rightarrow_r$ x ⟷ h' ⊢ f $\rightarrow_r$ x)"

lemma *preserved_code [code]:*
  "preserved f h h' = (((h ⊢ ok f) ∧ (h' ⊢ ok f) ∧ $|$h ⊢ f$|_r$ = $|$h' ⊢ f$|_r$) ∨ ((¬h ⊢ ok f) ∧ (¬h' ⊢ ok f)))"

```
  ⟨proof⟩

lemma reflp_preserved_f [simp]: "reflp (preserved f)"
  ⟨proof⟩
lemma transp_preserved_f [simp]: "transp (preserved f)"
  ⟨proof⟩


definition
  all_args :: "('a ⇒ ('heap, 'e, 'result) prog) ⇒ ('heap, 'e, 'result) prog set"
  where
    "all_args f = (⋃arg. {f arg})"


definition
  reads :: "('heap ⇒ 'heap ⇒ bool) set ⇒ ('heap, 'e, 'result) prog ⇒ 'heap
            ⇒ 'heap ⇒ bool"
  where
    "reads S getter h h' ⟷ (∀P ∈ S. reflp P ∧ transp P) ∧ ((∀P ∈ S. P h h')
                                 ⟶ preserved getter h h')"

lemma reads_singleton [simp]: "reads {preserved f} f h h'"
  ⟨proof⟩

lemma reads_bind_pure:
  assumes "pure f h" and "pure f h'"
    and "reads S f h h'"
    and "⋀x. h ⊢ f →r x ⟹ reads S (g x) h h'"
  shows "reads S (f ≫= g) h h'"
  ⟨proof⟩

lemma reads_insert_writes_set_left:
  "∀P ∈ S. reflp P ∧ transp P ⟹ reads {getter} f h h' ⟹ reads (insert getter S) f h h'"
  ⟨proof⟩

lemma reads_insert_writes_set_right:
  "reflp getter ⟹ transp getter ⟹ reads S f h h' ⟹ reads (insert getter S) f h h'"
  ⟨proof⟩

lemma reads_subset:
  "reads S f h h' ⟹ ∀P ∈ S' - S. reflp P ∧ transp P ⟹ S ⊆ S' ⟹ reads S' f h h'"
  ⟨proof⟩

lemma return_reads [simp]: "reads {} (return x) h h'"
  ⟨proof⟩

lemma error_reads [simp]: "reads {} (error e) h h'"
  ⟨proof⟩

lemma noop_reads [simp]: "reads {} noop h h'"
  ⟨proof⟩

lemma filter_M_reads:
  assumes "⋀x. x ∈ set xs ⟹ pure (P x) h" and "⋀x. x ∈ set xs ⟹ pure (P x) h'"
    and "⋀x. x ∈ set xs ⟹ reads S (P x) h h'"
    and "∀P ∈ S. reflp P ∧ transp P"
  shows "reads S (filter_M P xs) h h'"
  ⟨proof⟩

definition writes ::
  "('heap, 'e, 'result) prog set ⇒ ('heap, 'e, 'result2) prog ⇒ 'heap ⇒ 'heap ⇒ bool"
  where
    "writes S setter h h'
    ⟷ (h ⊢ setter →_h h' ⟶ (∃progs. set progs ⊆ S ∧ h ⊢ iterate_M progs →_h h'))"

lemma writes_singleton [simp]: "writes (all_args f) (f a) h h'"
  ⟨proof⟩

lemma writes_singleton2 [simp]: "writes {f} f h h'"
  ⟨proof⟩

lemma writes_union_left_I:
  assumes "writes S f h h'"
  shows "writes (S ∪ S') f h h'"
  ⟨proof⟩

lemma writes_union_right_I:
  assumes "writes S' f h h'"
  shows "writes (S ∪ S') f h h'"
  ⟨proof⟩

lemma writes_union_minus_split:
  assumes "writes (S - S2) f h h'"
    and "writes (S' - S2) f h h'"
  shows "writes ((S ∪ S') - S2) f h h'"
  ⟨proof⟩

lemma writes_subset: "writes S f h h' ⟹ S ⊆ S' ⟹ writes S' f h h'"
  ⟨proof⟩

lemma writes_error [simp]: "writes S (error e) h h'"
  ⟨proof⟩

lemma writes_not_ok [simp]: "¬h ⊢ ok f ⟹ writes S f h h'"
  ⟨proof⟩

lemma writes_pure [simp]:
  assumes "pure f h"
  shows "writes S f h h'"
  ⟨proof⟩

lemma writes_bind:
  assumes "⋀h2. writes S f h h2"
  assumes "⋀x h2. h ⊢ f →_r x ⟹ h ⊢ f →_h h2 ⟹ writes S (g x) h2 h'"
  shows "writes S (f ⤜ g) h h'"
  ⟨proof⟩

lemma writes_bind_pure:
  assumes "pure f h"
  assumes "⋀x. h ⊢ f →_r x ⟹ writes S (g x) h h'"
  shows "writes S (f ⤜ g) h h'"
  ⟨proof⟩

lemma writes_small_big:
  assumes "writes SW setter h h'"
  assumes "h ⊢ setter →_h h'"
  assumes "⋀h h' w. w ∈ SW ⟹  h ⊢ w →_h h' ⟹ P h h'"
  assumes "reflp P"
  assumes "transp P"
  shows "P h h'"
⟨proof⟩

lemma reads_writes_preserved:
  assumes "reads SR getter h h'"
  assumes "writes SW setter h h'"
  assumes "h ⊢ setter →_h h'"
  assumes "⋀h h'. ∀w ∈ SW. h ⊢ w →h h' ⟶ (∀r ∈ SR. r h h')"
  shows "h ⊢ getter →r x ⟷ h' ⊢ getter →r x"
⟨proof⟩

lemma reads_writes_separate_forwards:
  assumes "reads SR getter h h'"
  assumes "writes SW setter h h'"
  assumes "h ⊢ setter →h h'"
  assumes "h ⊢ getter →r x"
  assumes "⋀h h'. ∀w ∈ SW. h ⊢ w →h h' ⟶ (∀r ∈ SR. r h h')"
  shows "h' ⊢ getter →r x"
  ⟨proof⟩

lemma reads_writes_separate_backwards:
  assumes "reads SR getter h h'"
  assumes "writes SW setter h h'"
  assumes "h ⊢ setter →h h'"
  assumes "h' ⊢ getter →r x"
  assumes "⋀h h'. ∀w ∈ SW. h ⊢ w →h h' ⟶ (∀r ∈ SR. r h h')"
  shows "h ⊢ getter →r x"
  ⟨proof⟩

end
```

3 References and Pointers

In this chapter, we introduce a generic type for object-oriented references and typed pointers for each class type defined in the DOM standard.

3.1 References (Ref)

This theory, we introduce a generic reference. All our typed pointers include such a reference, which allows us to distinguish pointers of the same type, but also to iterate over all pointers in a set.

```
theory
  Ref
  imports
    "../preliminaries/Hiding_Type_Variables"
begin

instantiation sum :: (linorder, linorder) linorder
begin
definition less_eq_sum :: "'a + 'b ⇒ 'a + 'b ⇒ bool"
  where
    "less_eq_sum t t' = (case t of
      Inl l ⇒ (case t' of
        Inl l' ⇒ l ≤ l'
      | Inr r' ⇒ True)
    | Inr r ⇒ (case t' of
        Inl l' ⇒ False
      | Inr r' ⇒ r ≤ r'))"
definition less_sum :: "'a + 'b ⇒ 'a + 'b ⇒ bool"
  where
    "less_sum t t' ≡ t ≤ t' ∧ ¬ t' ≤ t"
instance ⟨proof⟩
end

type_synonym ref = nat
consts cast :: 'a

end
```

3.2 Object (ObjectPointer)

In this theory, we introduce the typed pointer for the class Object. This class is the common superclass of our class model.

```
theory ObjectPointer
  imports
    Ref
begin

datatype 'object_ptr object_ptr = Ext 'object_ptr
register_default_tvars "'object_ptr object_ptr"

instantiation object_ptr :: (linorder) linorder
begin
definition less_eq_object_ptr :: "'object_ptr::linorder object_ptr ⇒ 'object_ptr object_ptr ⇒ bool"
  where "less_eq_object_ptr x y ≡ (case x of Ext i ⇒ (case y of Ext j ⇒ i ≤ j))"
definition less_object_ptr :: "'object_ptr::linorder object_ptr ⇒ 'object_ptr object_ptr ⇒ bool"
```

```
  where "less_object_ptr x y ≡ x ≤ y ∧ ¬ y ≤ x"
instance ⟨proof⟩
end

end
```

3.3 Node (NodePointer)

In this theory, we introduce the typed pointers for the class Node.

```
theory NodePointer
  imports
    ObjectPointer
begin

datatype 'node_ptr node_ptr = Ext 'node_ptr
register_default_tvars "'node_ptr node_ptr"

type_synonym ('object_ptr, 'node_ptr) object_ptr = "('node_ptr node_ptr + 'object_ptr) object_ptr"
register_default_tvars "('object_ptr, 'node_ptr) object_ptr"

definition cast$_{node\_ptr2object\_ptr}$ :: "(_) node_ptr  ⇒ (_) object_ptr"
  where
    "cast$_{node\_ptr2object\_ptr}$ ptr = object_ptr.Ext (Inl ptr)"

definition cast$_{object\_ptr2node\_ptr}$ :: "(_) object_ptr ⇒ (_) node_ptr option"
  where
    "cast$_{object\_ptr2node\_ptr}$ object_ptr = (case object_ptr of object_ptr.Ext (Inl node_ptr)
                                                ⇒ Some node_ptr | _ ⇒ None)"

adhoc_overloading cast ⇌ cast$_{node\_ptr2object\_ptr}$ cast$_{object\_ptr2node\_ptr}$

definition is_node_ptr_kind :: "(_) object_ptr ⇒ bool"
  where
    "is_node_ptr_kind ptr = (cast$_{object\_ptr2node\_ptr}$ ptr ≠ None)"

instantiation node_ptr :: (linorder) linorder
begin
definition less_eq_node_ptr :: "(_::linorder) node_ptr ⇒ (_) node_ptr ⇒ bool"
  where "less_eq_node_ptr x y ≡ (case x of Ext i ⇒ (case y of Ext j ⇒ i ≤ j))"
definition less_node_ptr :: "(_::linorder) node_ptr ⇒ (_) node_ptr ⇒ bool"
  where "less_node_ptr x y ≡ x ≤ y ∧ ¬ y ≤ x"
instance
  ⟨proof⟩
end

lemma node_ptr_casts_commute [simp]:
  "cast$_{object\_ptr2node\_ptr}$ ptr = Some node_ptr ⟷ cast$_{node\_ptr2object\_ptr}$ node_ptr = ptr"
  ⟨proof⟩

lemma node_ptr_casts_commute2 [simp]:
  "cast$_{object\_ptr2node\_ptr}$ (cast$_{node\_ptr2object\_ptr}$ node_ptr) = Some node_ptr"
  ⟨proof⟩

lemma node_ptr_casts_commute3 [simp]:
  assumes "is_node_ptr_kind ptr"
  shows "cast$_{node\_ptr2object\_ptr}$ (the (cast$_{object\_ptr2node\_ptr}$ ptr)) = ptr"
  ⟨proof⟩

lemma is_node_ptr_kind_obtains:
  assumes "is_node_ptr_kind ptr"
  obtains node_ptr where "cast$_{object\_ptr2node\_ptr}$ ptr = Some node_ptr"
  ⟨proof⟩
```

```
lemma is_node_ptr_kind_none:
  assumes "¬is_node_ptr_kind ptr"
  shows "cast_{object_ptr2node_ptr} ptr = None"
  ⟨proof⟩

lemma is_node_ptr_kind_cast [simp]: "is_node_ptr_kind (cast_{node_ptr2object_ptr} node_ptr)"
  ⟨proof⟩

lemma cast_{node_ptr2object_ptr}_inject [simp]:
  "cast_{node_ptr2object_ptr} x = cast_{node_ptr2object_ptr} y ⟷ x = y"
  ⟨proof⟩

lemma cast_{object_ptr2node_ptr}_ext_none [simp]:
  "cast_{object_ptr2node_ptr} (object_ptr.Ext (Inr (Inr (Inr object_ext_ptr)))) = None"
  ⟨proof⟩

lemma node_ptr_inclusion [simp]:
  "cast_{node_ptr2object_ptr} node_ptr ∈ cast_{node_ptr2object_ptr} ` node_ptrs ⟷ node_ptr ∈ node_ptrs"
  ⟨proof⟩
end
```

3.4 Element (ElementPointer)

In this theory, we introduce the typed pointers for the class Element.

```
theory ElementPointer
  imports
    NodePointer
begin

datatype 'element_ptr element_ptr = Ref (the_ref: ref) | Ext 'element_ptr
register_default_tvars "'element_ptr element_ptr"

type_synonym ('node_ptr, 'element_ptr) node_ptr
  = "('element_ptr element_ptr + 'node_ptr) node_ptr"
register_default_tvars "('node_ptr, 'element_ptr) node_ptr"
type_synonym ('object_ptr, 'node_ptr, 'element_ptr) object_ptr
  = "('object_ptr, 'element_ptr element_ptr + 'node_ptr) object_ptr"
register_default_tvars "('object_ptr, 'node_ptr, 'element_ptr) object_ptr"


definition cast_{element_ptr2element_ptr} :: "(_) element_ptr ⇒ (_) element_ptr"
  where
    "cast_{element_ptr2element_ptr} = id"

definition cast_{element_ptr2node_ptr} :: "(_) element_ptr ⇒ (_) node_ptr"
  where
    "cast_{element_ptr2node_ptr} ptr = node_ptr.Ext (Inl ptr)"

abbreviation cast_{element_ptr2object_ptr} :: "(_) element_ptr ⇒ (_) object_ptr"
  where
    "cast_{element_ptr2object_ptr} ptr ≡ cast_{node_ptr2object_ptr} (cast_{element_ptr2node_ptr} ptr)"

definition cast_{node_ptr2element_ptr} :: "(_) node_ptr ⇒ (_) element_ptr option"
  where
    "cast_{node_ptr2element_ptr} node_ptr = (case node_ptr of node_ptr.Ext (Inl element_ptr)
                                        ⇒ Some element_ptr | _ ⇒ None)"

abbreviation cast_{object_ptr2element_ptr} :: "(_) object_ptr ⇒ (_) element_ptr option"
  where
    "cast_{object_ptr2element_ptr} ptr ≡ (case cast_{object_ptr2node_ptr} ptr of
                                    Some node_ptr ⇒ cast_{node_ptr2element_ptr} node_ptr
```

```
                              | None ⇒ None)"

adhoc_overloading cast ⇌ cast_element_ptr2node_ptr cast_element_ptr2object_ptr
  cast_node_ptr2element_ptr cast_object_ptr2element_ptr cast_element_ptr2element_ptr

consts is_element_ptr_kind :: 'a
definition is_element_ptr_kind_node_ptr :: "(_) node_ptr ⇒ bool"
  where
    "is_element_ptr_kind_node_ptr ptr = (case cast_node_ptr2element_ptr ptr of Some _ ⇒ True | _ ⇒ False)"

abbreviation is_element_ptr_kind_object_ptr :: "(_) object_ptr ⇒ bool"
  where
    "is_element_ptr_kind_object_ptr ptr ≡ (case cast ptr of
                                            Some node_ptr ⇒ is_element_ptr_kind_node_ptr node_ptr
                                          | None ⇒ False)"

adhoc_overloading is_element_ptr_kind ⇌ is_element_ptr_kind_object_ptr is_element_ptr_kind_node_ptr
lemmas is_element_ptr_kind_def = is_element_ptr_kind_node_ptr_def

consts is_element_ptr :: 'a
definition is_element_ptr_element_ptr :: "(_) element_ptr ⇒ bool"
  where
    "is_element_ptr_element_ptr ptr = (case ptr of element_ptr.Ref _ ⇒ True | _ ⇒ False)"

abbreviation is_element_ptr_node_ptr :: "(_) node_ptr ⇒ bool"
  where
    "is_element_ptr_node_ptr ptr ≡ (case cast ptr of
                                Some element_ptr ⇒ is_element_ptr_element_ptr element_ptr
                              | _ ⇒ False)"

abbreviation is_element_ptr_object_ptr :: "(_) object_ptr ⇒ bool"
  where
    "is_element_ptr_object_ptr ptr ≡ (case cast ptr of
                                Some node_ptr ⇒ is_element_ptr_node_ptr node_ptr
                              | None ⇒ False)"

adhoc_overloading is_element_ptr ⇌ is_element_ptr_object_ptr is_element_ptr_node_ptr is_element_ptr_element_ptr
lemmas is_element_ptr_def = is_element_ptr_element_ptr_def

consts is_element_ptr_ext :: 'a
abbreviation "is_element_ptr_ext_element_ptr ptr ≡ ¬ is_element_ptr_element_ptr ptr"

abbreviation "is_element_ptr_ext_node_ptr ptr ≡ is_element_ptr_kind ptr ∧ (¬ is_element_ptr_node_ptr ptr)"

abbreviation "is_element_ptr_ext_object_ptr ptr ≡ is_element_ptr_kind ptr ∧ (¬ is_element_ptr_object_ptr ptr)"
adhoc_overloading is_element_ptr_ext ⇌ is_element_ptr_ext_object_ptr is_element_ptr_ext_node_ptr


instantiation element_ptr :: (linorder) linorder
begin
definition
  less_eq_element_ptr :: "(_::linorder) element_ptr ⇒ (_)element_ptr ⇒ bool"
  where
    "less_eq_element_ptr x y ≡ (case x of Ext i ⇒ (case y of Ext j ⇒ i ≤ j | Ref _ ⇒ False)
                                     | Ref i ⇒ (case y of Ext _ ⇒ True | Ref j ⇒ i ≤ j))"
definition
  less_element_ptr :: "(_::linorder) element_ptr ⇒ (_) element_ptr ⇒ bool"
  where "less_element_ptr x y ≡ x ≤ y ∧ ¬ y ≤ x"
instance
  ⟨proof⟩
end

lemma is_element_ptr_ref [simp]: "is_element_ptr (element_ptr.Ref n)"
```

```
  ⟨proof⟩

lemma element_ptr_casts_commute [simp]:
  "cast_{node_ptr2element_ptr} node_ptr = Some element_ptr ⟷ cast_{element_ptr2node_ptr} element_ptr = node_ptr"
  ⟨proof⟩

lemma element_ptr_casts_commute2 [simp]:
  "(cast_{node_ptr2element_ptr} (cast_{element_ptr2node_ptr} element_ptr) = Some element_ptr)"
  ⟨proof⟩

lemma element_ptr_casts_commute3 [simp]:
  assumes "is_element_ptr_kind_{node_ptr} node_ptr"
  shows "cast_{element_ptr2node_ptr} (the (cast_{node_ptr2element_ptr} node_ptr)) = node_ptr"
  ⟨proof⟩

lemma is_element_ptr_kind_obtains:
  assumes "is_element_ptr_kind node_ptr"
  obtains element_ptr where "node_ptr = cast_{element_ptr2node_ptr} element_ptr"
  ⟨proof⟩

lemma is_element_ptr_kind_none:
  assumes "¬is_element_ptr_kind node_ptr"
  shows "cast_{node_ptr2element_ptr} node_ptr = None"
  ⟨proof⟩

lemma is_element_ptr_kind_cast [simp]:
  "is_element_ptr_kind (cast_{element_ptr2node_ptr} element_ptr)"
  ⟨proof⟩

lemma cast_{element_ptr2node_ptr}_inject [simp]:
  "cast_{element_ptr2node_ptr} x = cast_{element_ptr2node_ptr} y ⟷ x = y"
  ⟨proof⟩

lemma cast_{node_ptr2element_ptr}_ext_none [simp]:
  "cast_{node_ptr2element_ptr} (node_ptr.Ext (Inr (Inr node_ext_ptr))) = None"
  ⟨proof⟩

lemma is_element_ptr_implies_kind [dest]: "is_element_ptr_{node_ptr} ptr ⟹ is_element_ptr_kind_{node_ptr} ptr"
  ⟨proof⟩

end
```

3.5 CharacterData (CharacterDataPointer)

In this theory, we introduce the typed pointers for the class CharacterData.

```
theory CharacterDataPointer
  imports
    ElementPointer
begin

datatype 'character_data_ptr character_data_ptr = Ref (the_ref: ref) | Ext 'character_data_ptr
register_default_tvars "'character_data_ptr character_data_ptr"
type_synonym ('node_ptr, 'element_ptr, 'character_data_ptr) node_ptr
  = "('character_data_ptr character_data_ptr + 'node_ptr, 'element_ptr) node_ptr"
register_default_tvars "('node_ptr, 'element_ptr, 'character_data_ptr) node_ptr"
type_synonym ('object_ptr, 'node_ptr, 'element_ptr, 'character_data_ptr) object_ptr
  = "('object_ptr, 'character_data_ptr character_data_ptr + 'node_ptr, 'element_ptr) object_ptr"
register_default_tvars "('object_ptr, 'node_ptr, 'element_ptr, 'character_data_ptr) object_ptr"

definition cast_{character_data_ptr2node_ptr} :: "(_) character_data_ptr ⇒ (_) node_ptr"
  where
    "cast_{character_data_ptr2node_ptr} ptr = node_ptr.Ext (Inr (Inl ptr))"
```

```
abbreviation cast_character_data_ptr2object_ptr :: "(_) character_data_ptr ⇒ (_) object_ptr"
  where
    "cast_character_data_ptr2object_ptr ptr ≡ cast_node_ptr2object_ptr (cast_character_data_ptr2node_ptr ptr)"

definition cast_node_ptr2character_data_ptr :: "(_) node_ptr ⇒ (_) character_data_ptr option"
  where
    "cast_node_ptr2character_data_ptr node_ptr = (case node_ptr of
                         node_ptr.Ext (Inr (Inl character_data_ptr)) ⇒ Some character_data_ptr
                       | _ ⇒ None)"

abbreviation cast_object_ptr2character_data_ptr :: "(_) object_ptr ⇒ (_) character_data_ptr option"
  where
    "cast_object_ptr2character_data_ptr ptr ≡ (case cast_object_ptr2node_ptr ptr of
                                   Some node_ptr ⇒ cast_node_ptr2character_data_ptr node_ptr
                                 | None ⇒ None)"

adhoc_overloading cast ⇌ cast_character_data_ptr2node_ptr cast_character_data_ptr2object_ptr
  cast_node_ptr2character_data_ptr cast_object_ptr2character_data_ptr

consts is_character_data_ptr_kind :: 'a
definition is_character_data_ptr_kind_node_ptr :: "(_) node_ptr ⇒ bool"
  where
    "is_character_data_ptr_kind_node_ptr ptr = (case cast_node_ptr2character_data_ptr ptr
                                               of Some _ ⇒ True | _ ⇒ False)"

abbreviation is_character_data_ptr_kind_object_ptr :: "(_) object_ptr ⇒ bool"
  where
    "is_character_data_ptr_kind_object_ptr ptr ≡ (case cast ptr of
                         Some node_ptr ⇒ is_character_data_ptr_kind_node_ptr node_ptr
                       | None ⇒ False)"

adhoc_overloading is_character_data_ptr_kind ⇌ is_character_data_ptr_kind_object_ptr
  is_character_data_ptr_kind_node_ptr
lemmas is_character_data_ptr_kind_def = is_character_data_ptr_kind_node_ptr_def

consts is_character_data_ptr :: 'a
definition is_character_data_ptr_character_data_ptr :: "(_) character_data_ptr ⇒ bool"
  where
    "is_character_data_ptr_character_data_ptr ptr = (case ptr
                                      of character_data_ptr.Ref _ ⇒ True | _ ⇒ False)"

abbreviation is_character_data_ptr_node_ptr :: "(_) node_ptr ⇒ bool"
  where
    "is_character_data_ptr_node_ptr ptr ≡ (case cast ptr of
      Some character_data_ptr ⇒ is_character_data_ptr_character_data_ptr character_data_ptr
    | _ ⇒ False)"

abbreviation is_character_data_ptr_object_ptr :: "(_) object_ptr ⇒ bool"
  where
    "is_character_data_ptr_object_ptr ptr ≡ (case cast_object_ptr2node_ptr ptr of
      Some node_ptr ⇒ is_character_data_ptr_node_ptr node_ptr
    | None ⇒ False)"

adhoc_overloading is_character_data_ptr ⇌
  is_character_data_ptr_object_ptr is_character_data_ptr_node_ptr is_character_data_ptr_character_data_ptr
lemmas is_character_data_ptr_def = is_character_data_ptr_character_data_ptr_def

consts is_character_data_ptr_ext :: 'a
abbreviation
  "is_character_data_ptr_ext_character_data_ptr ptr ≡ ¬ is_character_data_ptr_character_data_ptr ptr"

abbreviation "is_character_data_ptr_ext_node_ptr ptr ≡ (case cast_node_ptr2character_data_ptr ptr of
  Some character_data_ptr ⇒ is_character_data_ptr_ext_{character_data_ptr} character_data_ptr
| None ⇒ False)"

abbreviation "is_character_data_ptr_ext_{object_ptr} ptr ≡ (case cast_{object_ptr2node_ptr} ptr of
  Some node_ptr ⇒ is_character_data_ptr_ext_{node_ptr} node_ptr
| None ⇒ False)"

adhoc_overloading is_character_data_ptr_ext ⇌
  is_character_data_ptr_ext_{object_ptr} is_character_data_ptr_ext_{node_ptr} is_character_data_ptr_ext_{character_data_ptr}

instantiation character_data_ptr :: (linorder) linorder
begin
definition
  less_eq_character_data_ptr :: "(_::linorder) character_data_ptr ⇒ (_) character_data_ptr ⇒ bool"
  where
    "less_eq_character_data_ptr x y ≡ (case x of Ext i ⇒ (case y of Ext j ⇒ i ≤ j | Ref _ ⇒ False)
                                              | Ref i ⇒ (case y of Ext _ ⇒ True | Ref j ⇒ i ≤ j))"
definition
  less_character_data_ptr :: "(_::linorder) character_data_ptr ⇒ (_) character_data_ptr ⇒ bool"
  where "less_character_data_ptr x y ≡ x ≤ y ∧ ¬ y ≤ x"
instance
  ⟨proof⟩
end

lemma is_character_data_ptr_ref [simp]: "is_character_data_ptr (character_data_ptr.Ref n)"
  ⟨proof⟩

lemma cast_element_ptr_not_character_data_ptr [simp]:
  "(cast_{element_ptr2node_ptr} element_ptr ≠ cast_{character_data_ptr2node_ptr} character_data_ptr)"
  "(cast_{character_data_ptr2node_ptr} character_data_ptr ≠ cast_{element_ptr2node_ptr} element_ptr)"
  ⟨proof⟩

lemma is_character_data_ptr_kind_not_element_ptr [simp]:
  "¬ is_character_data_ptr_kind (cast_{element_ptr2node_ptr} element_ptr)"
  ⟨proof⟩
lemma is_element_ptr_kind_not_character_data_ptr [simp]:
  "¬ is_element_ptr_kind (cast_{character_data_ptr2node_ptr} character_data_ptr)"
  ⟨proof⟩

lemma is_character_data_ptr_kind_cast [simp]:
  "is_character_data_ptr_kind (cast_{character_data_ptr2node_ptr} character_data_ptr)"
  ⟨proof⟩

lemma character_data_ptr_casts_commute [simp]:
  "cast_{node_ptr2character_data_ptr} node_ptr = Some character_data_ptr
  ⟷ cast_{character_data_ptr2node_ptr} character_data_ptr = node_ptr"
  ⟨proof⟩

lemma character_data_ptr_casts_commute2 [simp]:
  "(cast_{node_ptr2character_data_ptr} (cast_{character_data_ptr2node_ptr} character_data_ptr) = Some character_data_ptr)"
  ⟨proof⟩

lemma character_data_ptr_casts_commute3 [simp]:
  assumes "is_character_data_ptr_kind_{node_ptr} node_ptr"
  shows "cast_{character_data_ptr2node_ptr} (the (cast_{node_ptr2character_data_ptr} node_ptr)) = node_ptr"
  ⟨proof⟩

lemma is_character_data_ptr_kind_obtains:
  assumes "is_character_data_ptr_kind_{node_ptr} node_ptr"
  obtains character_data_ptr where "cast_{character_data_ptr2node_ptr} character_data_ptr = node_ptr"
  ⟨proof⟩

lemma is_character_data_ptr_kind_none:
  assumes "¬is_character_data_ptr_kind$_{node\_ptr}$ node_ptr"
  shows "cast$_{node\_ptr2character\_data\_ptr}$ node_ptr = None"
  ⟨proof⟩

lemma cast$_{character\_data\_ptr2node\_ptr}$_inject [simp]:
  "cast$_{character\_data\_ptr2node\_ptr}$ x = cast$_{character\_data\_ptr2node\_ptr}$ y ⟷ x = y"
  ⟨proof⟩

lemma cast$_{node\_ptr2character\_data\_ptr}$_ext_none [simp]:
  "cast$_{node\_ptr2character\_data\_ptr}$ (node_ptr.Ext (Inr (Inr node_ext_ptr))) = None"
  ⟨proof⟩

end
```

3.6 Document (DocumentPointer)

In this theory, we introduce the typed pointers for the class Document.

```
theory DocumentPointer
  imports
    CharacterDataPointer
begin

datatype 'document_ptr document_ptr = Ref (the_ref: ref) | Ext 'document_ptr
register_default_tvars "'document_ptr document_ptr"
type_synonym ('object_ptr, 'node_ptr, 'element_ptr, 'character_data_ptr, 'document_ptr) object_ptr
  = "('document_ptr document_ptr + 'object_ptr, 'node_ptr, 'element_ptr, 'character_data_ptr) object_ptr"
register_default_tvars "('object_ptr, 'node_ptr, 'element_ptr, 'character_data_ptr, 'document_ptr) object_ptr"


definition cast$_{document\_ptr2object\_ptr}$ :: "(_)document_ptr ⇒ (_) object_ptr"
  where
    "cast$_{document\_ptr2object\_ptr}$ ptr = object_ptr.Ext (Inr (Inl ptr))"

definition cast$_{object\_ptr2document\_ptr}$ :: "(_) object_ptr ⇒ (_) document_ptr option"
  where
    "cast$_{object\_ptr2document\_ptr}$ ptr = (case ptr of
                object_ptr.Ext (Inr (Inl document_ptr)) ⇒ Some document_ptr
              | _ ⇒ None)"

adhoc_overloading cast ⇌ cast$_{document\_ptr2object\_ptr}$ cast$_{object\_ptr2document\_ptr}$


definition is_document_ptr_kind :: "(_) object_ptr ⇒ bool"
  where
    "is_document_ptr_kind ptr = (case cast$_{object\_ptr2document\_ptr}$ ptr of
                                 Some _ ⇒ True | None ⇒ False)"

consts is_document_ptr :: 'a
definition is_document_ptr$_{document\_ptr}$ :: "(_) document_ptr ⇒ bool"
  where
    "is_document_ptr$_{document\_ptr}$ ptr = (case ptr of document_ptr.Ref _ ⇒ True | _ ⇒ False)"

abbreviation is_document_ptr$_{object\_ptr}$ :: "(_) object_ptr ⇒ bool"
  where
    "is_document_ptr$_{object\_ptr}$ ptr ≡ (case cast$_{object\_ptr2document\_ptr}$ ptr of
      Some document_ptr ⇒ is_document_ptr$_{document\_ptr}$ document_ptr
    | None ⇒ False)"
adhoc_overloading is_document_ptr ⇌ is_document_ptr$_{object\_ptr}$ is_document_ptr$_{document\_ptr}$
lemmas is_document_ptr_def = is_document_ptr$_{document\_ptr}$_def

consts is_document_ptr_ext :: 'a
abbreviation "is_document_ptr_ext$_{document\_ptr}$ ptr ≡ ¬ is_document_ptr$_{document\_ptr}$ ptr"
```

abbreviation *"is_document_ptr_ext*$_{object\_ptr}$ *ptr ≡ (case cast*$_{object\_ptr2document\_ptr}$ *ptr of*
  Some document_ptr ⇒ is_document_ptr_ext$_{document\_ptr}$ *document_ptr*
| None ⇒ False)"
adhoc_overloading *is_document_ptr_ext ⇌ is_document_ptr_ext*$_{object\_ptr}$ *is_document_ptr_ext*$_{document\_ptr}$

instantiation *document_ptr :: (linorder) linorder*
begin
definition *less_eq_document_ptr :: "(_::linorder) document_ptr ⇒ (_) document_ptr ⇒ bool"*
  where *"less_eq_document_ptr x y ≡ (case x of Ext i ⇒ (case y of Ext j ⇒ i ≤ j | Ref _ ⇒ False)*
                                                     | Ref i ⇒ (case y of Ext _ ⇒ True | Ref j ⇒ i ≤ j))"
definition *less_document_ptr :: "(_::linorder) document_ptr ⇒ (_) document_ptr ⇒ bool"*
  where *"less_document_ptr x y ≡ x ≤ y ∧ ¬ y ≤ x"*
instance
  ⟨*proof*⟩
end

lemma *is_document_ptr_ref [simp]: "is_document_ptr (document_ptr.Ref n)"*
  ⟨*proof*⟩

lemma *cast_document_ptr_not_node_ptr [simp]:*
  "cast$_{document\_ptr2object\_ptr}$ *document_ptr ≠ cast*$_{node\_ptr2object\_ptr}$ *node_ptr"*
  "cast$_{node\_ptr2object\_ptr}$ *node_ptr ≠ cast*$_{document\_ptr2object\_ptr}$ *document_ptr"*
  ⟨*proof*⟩

lemma *document_ptr_no_node_ptr_cast [simp]:*
  "¬ is_document_ptr_kind (cast$_{node\_ptr2object\_ptr}$ *node_ptr)"*
  ⟨*proof*⟩
lemma *node_ptr_no_document_ptr_cast [simp]:*
  "¬ is_node_ptr_kind (cast$_{document\_ptr2object\_ptr}$ *document_ptr)"*
  ⟨*proof*⟩

lemma *document_ptr_document_ptr_cast [simp]:*
  "is_document_ptr_kind (cast$_{document\_ptr2object\_ptr}$ *document_ptr)"*
  ⟨*proof*⟩

lemma *document_ptr_casts_commute [simp]:*
  "cast$_{object\_ptr2document\_ptr}$ *ptr = Some document_ptr ⟷ cast*$_{document\_ptr2object\_ptr}$ *document_ptr = ptr"*
  ⟨*proof*⟩

lemma *document_ptr_casts_commute2 [simp]:*
  "(cast$_{object\_ptr2document\_ptr}$ *(cast*$_{document\_ptr2object\_ptr}$ *document_ptr) = Some document_ptr)"*
  ⟨*proof*⟩

lemma *document_ptr_casts_commute3 [simp]:*
  assumes *"is_document_ptr_kind ptr"*
  shows *"cast*$_{document\_ptr2object\_ptr}$ *(the (cast*$_{object\_ptr2document\_ptr}$ *ptr)) = ptr"*
  ⟨*proof*⟩

lemma *is_document_ptr_kind_obtains:*
  assumes *"is_document_ptr_kind ptr"*
  obtains *document_ptr* **where** *"ptr = cast*$_{document\_ptr2object\_ptr}$ *document_ptr"*
  ⟨*proof*⟩

lemma *is_document_ptr_kind_none:*
  assumes *"¬is_document_ptr_kind ptr"*
  shows *"cast*$_{object\_ptr2document\_ptr}$ *ptr = None"*
  ⟨*proof*⟩

lemma *cast*$_{document\_ptr2object\_ptr}$*_inject [simp]:*
  "cast$_{document\_ptr2object\_ptr}$ *x = cast*$_{document\_ptr2object\_ptr}$ *y ⟷ x = y"*
  ⟨*proof*⟩

```
lemma cast_{object_ptr2document_ptr}_ext_none [simp]:
  "cast_{object_ptr2document_ptr} (object_ptr.Ext (Inr (Inr (Inr object_ext_ptr)))) = None"
  ⟨proof⟩

lemma is_document_ptr_kind_not_element_ptr_kind [dest]:
  "is_document_ptr_kind ptr ⟹ ¬ is_element_ptr_kind ptr"
  ⟨proof⟩
end
```

3.7 ShadowRoot (ShadowRootPointer)

In this theory, we introduce the typed pointers for the class ShadowRoot. Note that, in this document, we will not make use of ShadowRoots nor will we discuss their particular properties. We only include them here, as they are required for future work and they cannot be added alter following the object-oriented extensibility of our data model.

```
theory ShadowRootPointer
  imports
    "DocumentPointer"
begin

datatype 'shadow_root_ptr shadow_root_ptr = Ref (the_ref: ref) | Ext 'shadow_root_ptr
register_default_tvars "'shadow_root_ptr shadow_root_ptr"
type_synonym ('object_ptr, 'node_ptr, 'element_ptr, 'character_data_ptr,
    'document_ptr, 'shadow_root_ptr) object_ptr
  = "('shadow_root_ptr shadow_root_ptr + 'object_ptr, 'node_ptr, 'element_ptr,
      'character_data_ptr, 'document_ptr) object_ptr"
register_default_tvars "('object_ptr, 'node_ptr, 'element_ptr, 'character_data_ptr,
                         'document_ptr, 'shadow_root_ptr) object_ptr"

definition cast_{shadow_root_ptr2shadow_root_ptr} :: "(_) shadow_root_ptr ⇒ (_) shadow_root_ptr"
  where
    "cast_{shadow_root_ptr2shadow_root_ptr} = id"

definition cast_{shadow_root_ptr2object_ptr} :: "(_)shadow_root_ptr ⇒ (_) object_ptr"
  where
    "cast_{shadow_root_ptr2object_ptr} ptr = object_ptr.Ext (Inr (Inr (Inl ptr)))"

definition cast_{object_ptr2shadow_root_ptr} :: "(_) object_ptr ⇒ (_) shadow_root_ptr option"
  where
    "cast_{object_ptr2shadow_root_ptr} ptr = (case ptr of
              object_ptr.Ext (Inr (Inr (Inl shadow_root_ptr))) ⇒ Some shadow_root_ptr
              | _ ⇒ None)"

adhoc_overloading cast ⇌ cast_{shadow_root_ptr2object_ptr} cast_{object_ptr2shadow_root_ptr} cast_{shadow_root_ptr2shadow_root_pt


definition is_shadow_root_ptr_kind :: "(_) object_ptr ⇒ bool"
  where
    "is_shadow_root_ptr_kind ptr = (case cast_{object_ptr2shadow_root_ptr} ptr of Some _ ⇒ True
                                                                   | None ⇒ False)"

consts is_shadow_root_ptr :: 'a
definition is_shadow_root_ptr_{shadow_root_ptr} :: "(_) shadow_root_ptr ⇒ bool"
  where
    "is_shadow_root_ptr_{shadow_root_ptr} ptr = (case ptr of shadow_root_ptr.Ref _ ⇒ True
                                                      | _ ⇒ False)"

abbreviation is_shadow_root_ptr_{object_ptr} :: "(_) object_ptr ⇒ bool"
  where
    "is_shadow_root_ptr_{object_ptr} ptr ≡ (case cast_{object_ptr2shadow_root_ptr} ptr of
      Some shadow_root_ptr ⇒ is_shadow_root_ptr_{shadow_root_ptr} shadow_root_ptr
    | None ⇒ False)"
adhoc_overloading is_shadow_root_ptr ⇌ is_shadow_root_ptr$_{object\_ptr}$ is_shadow_root_ptr$_{shadow\_root\_ptr}$
lemmas is_shadow_root_ptr_def = is_shadow_root_ptr$_{shadow\_root\_ptr}$_def

consts is_shadow_root_ptr_ext :: 'a
abbreviation "is_shadow_root_ptr_ext$_{shadow\_root\_ptr}$ ptr ≡ ¬ is_shadow_root_ptr$_{shadow\_root\_ptr}$ ptr"

abbreviation "is_shadow_root_ptr_ext$_{object\_ptr}$ ptr ≡ (case cast$_{object\_ptr2shadow\_root\_ptr}$ ptr of
  Some shadow_root_ptr ⇒ is_shadow_root_ptr_ext$_{shadow\_root\_ptr}$ shadow_root_ptr
| None ⇒ False)"
adhoc_overloading is_shadow_root_ptr_ext ⇌ is_shadow_root_ptr_ext$_{object\_ptr}$ is_shadow_root_ptr_ext$_{shadow\_root\_ptr}$

instantiation shadow_root_ptr :: (linorder) linorder
begin
definition
  less_eq_shadow_root_ptr :: "(_::linorder) shadow_root_ptr ⇒ (_) shadow_root_ptr ⇒ bool"
  where
    "less_eq_shadow_root_ptr x y ≡ (case x of Ext i ⇒ (case y of Ext j ⇒ i ≤ j | Ref _ ⇒ False)
                                           | Ref i ⇒ (case y of Ext _ ⇒ True | Ref j ⇒ i ≤ j))"
definition less_shadow_root_ptr :: "(_::linorder) shadow_root_ptr ⇒ (_) shadow_root_ptr ⇒ bool"
  where "less_shadow_root_ptr x y ≡ x ≤ y ∧ ¬ y ≤ x"
instance
  ⟨proof⟩
end


lemma is_shadow_root_ptr_ref [simp]: "is_shadow_root_ptr (shadow_root_ptr.Ref n)"
  ⟨proof⟩

lemma is_shadow_root_ptr_not_node_ptr[simp]: "¬is_shadow_root_ptr (cast$_{node\_ptr2object\_ptr}$ node_ptr)"
  ⟨proof⟩

lemma cast_shadow_root_ptr_not_node_ptr [simp]:
  "cast$_{shadow\_root\_ptr2object\_ptr}$ shadow_root_ptr ≠ cast$_{node\_ptr2object\_ptr}$ node_ptr"
  "cast$_{node\_ptr2object\_ptr}$ node_ptr ≠ cast$_{shadow\_root\_ptr2object\_ptr}$ shadow_root_ptr"
  ⟨proof⟩

lemma cast_shadow_root_ptr_not_document_ptr [simp]:
  "cast$_{shadow\_root\_ptr2object\_ptr}$ shadow_root_ptr ≠ cast$_{document\_ptr2object\_ptr}$ document_ptr"
  "cast$_{document\_ptr2object\_ptr}$ document_ptr ≠ cast$_{shadow\_root\_ptr2object\_ptr}$ shadow_root_ptr"
  ⟨proof⟩

lemma shadow_root_ptr_no_node_ptr_cast [simp]:
  "¬ is_shadow_root_ptr_kind (cast$_{node\_ptr2object\_ptr}$ node_ptr)"
  ⟨proof⟩
lemma node_ptr_no_shadow_root_ptr_cast [simp]:
  "¬ is_node_ptr_kind (cast$_{shadow\_root\_ptr2object\_ptr}$ shadow_root_ptr)"
  ⟨proof⟩

lemma shadow_root_ptr_no_document_ptr_cast [simp]:
  "¬ is_shadow_root_ptr_kind (cast$_{document\_ptr2object\_ptr}$ document_ptr)"
  ⟨proof⟩
lemma document_ptr_no_shadow_root_ptr_cast [simp]:
  "¬ is_document_ptr_kind (cast$_{shadow\_root\_ptr2object\_ptr}$ shadow_root_ptr)"
  ⟨proof⟩

lemma shadow_root_ptr_shadow_root_ptr_cast [simp]:
  "is_shadow_root_ptr_kind (cast$_{shadow\_root\_ptr2object\_ptr}$ shadow_root_ptr)"
  ⟨proof⟩

lemma shadow_root_ptr_casts_commute [simp]:
  "cast$_{object\_ptr2shadow\_root\_ptr}$ ptr = Some shadow_root_ptr ⟷ cast$_{shadow\_root\_ptr2object\_ptr}$ shadow_root_ptr
= ptr"
```

⟨*proof*⟩

lemma *shadow_root_ptr_casts_commute2 [simp]:*
"(cast$_{object\_ptr2shadow\_root\_ptr}$ (cast$_{shadow\_root\_ptr2object\_ptr}$ shadow_root_ptr) = Some shadow_root_ptr)"
⟨*proof*⟩

lemma *shadow_root_ptr_casts_commute3 [simp]:*
assumes "is_shadow_root_ptr_kind ptr"
shows "cast$_{shadow\_root\_ptr2object\_ptr}$ (the (cast$_{object\_ptr2shadow\_root\_ptr}$ ptr)) = ptr"
⟨*proof*⟩

lemma *is_shadow_root_ptr_kind_obtains:*
assumes "is_shadow_root_ptr_kind ptr"
obtains shadow_root_ptr **where** "ptr = cast$_{shadow\_root\_ptr2object\_ptr}$ shadow_root_ptr"
⟨*proof*⟩

lemma *is_shadow_root_ptr_kind_none:*
assumes "¬is_shadow_root_ptr_kind ptr"
shows "cast$_{object\_ptr2shadow\_root\_ptr}$ ptr = None"
⟨*proof*⟩

lemma *cast$_{shadow_root_ptr2object_ptr}$_inject [simp]:*
"cast$_{shadow\_root\_ptr2object\_ptr}$ x = cast$_{shadow\_root\_ptr2object\_ptr}$ y ⟷ x = y"
⟨*proof*⟩

lemma *cast$_{object_ptr2shadow_root_ptr}$_ext_none [simp]:*
"cast$_{object\_ptr2shadow\_root\_ptr}$ (object_ptr.Ext (Inr (Inr (Inr object_ext_ptr)))) = None"
⟨*proof*⟩

lemma *is_shadow_root_ptr_kind_simp1 [dest]:* "is_document_ptr_kind ptr ⟹ ¬is_shadow_root_ptr_kind ptr"
⟨*proof*⟩

lemma *is_shadow_root_ptr_kind_simp2 [dest]:* "is_node_ptr_kind ptr ⟹ ¬is_shadow_root_ptr_kind ptr"
⟨*proof*⟩

end

4 Classes

In this chapter, we introduce the classes of our DOM model. The definition of the class types follows closely the one of the pointer types. Instead of datatypes, we use records for our classes. a generic type for object-oriented references and typed pointers for each class type defined in the DOM standard.

4.1 The Class Infrastructure (BaseClass)

In this theory, we introduce the basic infrastructure for our encoding of classes.

```
theory BaseClass
  imports
    "HOL-Library.Finite_Map"
    "../pointers/Ref"
    "../Core_DOM_Basic_Datatypes"
begin

named_theorems instances

consts get :: 'a
consts put :: 'a
consts delete :: 'a
```

Overall, the definition of the class types follows closely the one of the pointer types. Instead of datatypes, we use records for our classes. This allows us to, first, make use of record inheritance, which is, in addition to the type synonyms of previous class types, the second place where the inheritance relationship of our types manifest. Second, we get a convenient notation to define classes, in addition to automatically generated getter and setter functions.

Along with our class types, we also develop our heap type, which is a finite map at its core. It is important to note that while the map stores a mapping from `object_ptr` to `Object`, we restrict the type variables of the record extension slot of `Object` in such a way that allows down-casting, but requires a bit of taking-apart and re-assembling of our records before they are stored in the heap.

Throughout the theory files, we will use underscore case to reference pointer types, and camel case for class types.

Every class type contains at least one attribute; nothing. This is used for two purposes: first, the record package does not allow records without any attributes. Second, we will use the getter of nothing later to check whether a class of the correct type could be retrieved, for which we will be able to use our infrastructure regarding the behaviour of getters across different heaps.

```
locale l_type_wf = fixes type_wf :: "'heap ⇒ bool"

locale l_known_ptr = fixes known_ptr :: "'ptr ⇒ bool"

end
```

4.2 Object (ObjectClass)

In this theory, we introduce the definition of the class Object. This class is the common superclass of our class model.

```
theory ObjectClass
  imports
    BaseClass
    "../pointers/ObjectPointer"
```

```
begin

record RObject =
  nothing :: unit
register_default_tvars "'Object RObject_ext"
type_synonym 'Object Object = "'Object RObject_scheme"
register_default_tvars "'Object Object"

datatype ('object_ptr, 'Object) heap = Heap (the_heap: "((_) object_ptr, (_) Object) fmap")
register_default_tvars "('object_ptr, 'Object) heap"
type_synonym heap$_{final}$ = "(unit, unit) heap"

definition object_ptr_kinds :: "(_) heap ⇒ (_) object_ptr fset"
  where
    "object_ptr_kinds = fmdom ∘ the_heap"

lemma object_ptr_kinds_simp [simp]:
  "object_ptr_kinds (Heap (fmupd object_ptr object (the_heap h)))
          = {|object_ptr|} |∪| object_ptr_kinds h"
  ⟨proof⟩

definition get$_{Object}$ :: "(_) object_ptr ⇒ (_) heap ⇒ (_) Object option"
  where
    "get$_{Object}$ ptr h = fmlookup (the_heap h) ptr"
adhoc_overloading get ⇌ get$_{Object}$

locale l_type_wf_def$_{Object}$
begin
definition a_type_wf :: "(_) heap ⇒ bool"
  where
    "a_type_wf h = True"
end
global_interpretation l_type_wf_def$_{Object}$ defines type_wf = a_type_wf ⟨proof⟩
lemmas type_wf_defs = a_type_wf_def

locale l_type_wf$_{Object}$ = l_type_wf type_wf for type_wf :: "((_) heap ⇒ bool)" +
  assumes type_wf$_{Object}$: "type_wf h ⟹ ObjectClass.type_wf h"

locale l_get$_{Object}$_lemmas = l_type_wf$_{Object}$
begin
lemma get$_{Object}$_type_wf:
  assumes "type_wf h"
  shows "object_ptr |∈| object_ptr_kinds h ⟷ get$_{Object}$ object_ptr h ≠ None"
  ⟨proof⟩
end

global_interpretation l_get$_{Object}$_lemmas type_wf
  ⟨proof⟩

definition put$_{Object}$ :: "(_) object_ptr ⇒ (_) Object ⇒ (_) heap ⇒ (_) heap"
  where
    "put$_{Object}$ ptr obj h = Heap (fmupd ptr obj (the_heap h))"
adhoc_overloading put ⇌ put$_{Object}$

lemma put$_{Object}$_ptr_in_heap:
  assumes "put$_{Object}$ object_ptr object h = h'"
  shows "object_ptr |∈| object_ptr_kinds h'"
  ⟨proof⟩

lemma put$_{Object}$_put_ptrs:
  assumes "put$_{Object}$ object_ptr object h = h'"
  shows "object_ptr_kinds h' = object_ptr_kinds h |∪| {|object_ptr|}"
  ⟨proof⟩
```

```
lemma object_more_extend_id [simp]: "more (extend x y) = y"
  ⟨proof⟩

lemma object_empty [simp]: "⦇nothing = (), ... = more x⦈ = x"
  ⟨proof⟩

locale l_known_ptr$_{Object}$
begin
definition a_known_ptr :: "(_) object_ptr ⇒ bool"
  where
    "a_known_ptr ptr = False"

lemma known_ptr_not_object_ptr:
  "a_known_ptr ptr ⟹ ¬is_object_ptr ptr ⟹ known_ptr ptr"
  ⟨proof⟩
end
global_interpretation l_known_ptr$_{Object}$ defines known_ptr = a_known_ptr ⟨proof⟩
lemmas known_ptr_defs = a_known_ptr_def

locale l_known_ptrs = l_known_ptr known_ptr for known_ptr :: "(_) object_ptr ⇒ bool" +
  fixes known_ptrs :: "(_) heap ⇒ bool"
  assumes known_ptrs_known_ptr: "known_ptrs h ⟹ ptr |∈| object_ptr_kinds h ⟹ known_ptr ptr"
  assumes known_ptrs_preserved:
    "object_ptr_kinds h = object_ptr_kinds h' ⟹ known_ptrs h = known_ptrs h'"
  assumes known_ptrs_subset:
    "object_ptr_kinds h' |⊆| object_ptr_kinds h ⟹ known_ptrs h ⟹ known_ptrs h'"
  assumes known_ptrs_new_ptr:
    "object_ptr_kinds h' = object_ptr_kinds h |∪| {|new_ptr|} ⟹ known_ptr new_ptr ⟹
known_ptrs h ⟹ known_ptrs h'"

locale l_known_ptrs$_{Object}$ = l_known_ptr known_ptr for known_ptr :: "(_) object_ptr ⇒ bool"
begin
definition a_known_ptrs :: "(_) heap ⇒ bool"
  where
    "a_known_ptrs h = (∀ptr ∈ fset (object_ptr_kinds h). known_ptr ptr)"

lemma known_ptrs_known_ptr:
  "a_known_ptrs h ⟹ ptr |∈| object_ptr_kinds h ⟹ known_ptr ptr"
  ⟨proof⟩

lemma known_ptrs_preserved:
  "object_ptr_kinds h = object_ptr_kinds h' ⟹ a_known_ptrs h = a_known_ptrs h'"
  ⟨proof⟩
lemma known_ptrs_subset:
  "object_ptr_kinds h' |⊆| object_ptr_kinds h ⟹ a_known_ptrs h ⟹ a_known_ptrs h'"
  ⟨proof⟩
lemma known_ptrs_new_ptr:
  "object_ptr_kinds h' = object_ptr_kinds h |∪| {|new_ptr|} ⟹ known_ptr new_ptr ⟹
a_known_ptrs h ⟹ a_known_ptrs h'"
  ⟨proof⟩
end
global_interpretation l_known_ptrs$_{Object}$ known_ptr defines known_ptrs = a_known_ptrs ⟨proof⟩
lemmas known_ptrs_defs = a_known_ptrs_def

lemma known_ptrs_is_l_known_ptrs: "l_known_ptrs known_ptr known_ptrs"
  ⟨proof⟩


lemma get_object_ptr_simp1 [simp]: "get$_{Object}$ object_ptr (put$_{Object}$ object_ptr object h) = Some object"
  ⟨proof⟩
lemma get_object_ptr_simp2 [simp]:
  "object_ptr ≠ object_ptr'
  ⟹ get_Object object_ptr (put_Object object_ptr' object h) = get_Object object_ptr h"
  ⟨proof⟩
```

4.2.1 Limited Heap Modifications

```
definition heap_unchanged_except :: "(_) object_ptr set ⇒ (_) heap ⇒ (_) heap ⇒ bool"
  where
    "heap_unchanged_except S h h' = (∀ptr ∈ (fset (object_ptr_kinds h)
                                     ∪ (fset (object_ptr_kinds h'))) - S. get ptr h = get ptr h')"

definition delete_Object :: "(_) object_ptr ⇒ (_) heap ⇒ (_) heap option" where
  "delete_Object ptr h = (if ptr |∈| object_ptr_kinds h then Some (Heap (fmdrop ptr (the_heap h)))
                                                           else None)"

lemma delete_Object_pointer_removed:
  assumes "delete_Object ptr h = Some h'"
  shows "ptr |∉| object_ptr_kinds h'"
  ⟨proof⟩

lemma delete_Object_pointer_ptr_in_heap:
  assumes "delete_Object ptr h = Some h'"
  shows "ptr |∈| object_ptr_kinds h"
  ⟨proof⟩

lemma delete_Object_ok:
  assumes "ptr |∈| object_ptr_kinds h"
  shows "delete_Object ptr h ≠ None"
  ⟨proof⟩
```

4.2.2 Code Generator Setup

```
definition "create_heap xs = Heap (fmap_of_list xs)"

code_datatype ObjectClass.heap.Heap create_heap

lemma object_ptr_kinds_code3 [code]:
  "fmlookup (the_heap (create_heap xs)) x = map_of xs x"
  ⟨proof⟩

lemma object_ptr_kinds_code4 [code]:
  "the_heap (create_heap xs) = fmap_of_list xs"
  ⟨proof⟩

lemma object_ptr_kinds_code5 [code]:
  "the_heap (Heap x) = x"
  ⟨proof⟩


end
```

4.3 Node (NodeClass)

In this theory, we introduce the types for the Node class.

```
theory NodeClass
  imports
    ObjectClass
    "../pointers/NodePointer"
begin
```

Node

```
record RNode = RObject
```

```
  + nothing :: unit
register_default_tvars "'Node RNode_ext"
type_synonym 'Node Node = "'Node RNode_scheme"
register_default_tvars "'Node Node"
type_synonym ('Object, 'Node) Object = "('Node RNode_ext + 'Object) Object"
register_default_tvars "('Object, 'Node) Object"

type_synonym ('object_ptr, 'node_ptr, 'Object, 'Node) heap
  = "('node_ptr node_ptr + 'object_ptr, 'Node RNode_ext + 'Object) heap"
register_default_tvars
  "('object_ptr, 'node_ptr, 'Object, 'Node) heap"
type_synonym heapfinal = "(unit, unit, unit, unit) heap"


definition node_ptr_kinds :: "(_) heap ⇒ (_) node_ptr fset"
  where
    "node_ptr_kinds heap =
    (the |`| (castobject_ptr2node_ptr |`| (ffilter is_node_ptr_kind (object_ptr_kinds heap))))"

lemma node_ptr_kinds_simp [simp]:
  "node_ptr_kinds (Heap (fmupd (cast node_ptr) node (the_heap h)))
       = {|node_ptr|} |∪| node_ptr_kinds h"
  ⟨proof⟩

definition castObject2Node :: "(_) Object ⇒ (_) Node option"
  where
    "castObject2Node obj = (case RObject.more obj of Inl node
    ⇒ Some (RObject.extend (RObject.truncate obj) node) | _ ⇒ None)"
adhoc_overloading cast ⇌ castObject2Node

definition castNode2Object:: "(_) Node ⇒ (_) Object"
  where
    "castNode2Object node = (RObject.extend (RObject.truncate node) (Inl (RObject.more node)))"
adhoc_overloading cast ⇌ castNode2Object

definition is_node_kind :: "(_) Object ⇒ bool"
  where
    "is_node_kind ptr ⟷ castObject2Node ptr ≠ None"

definition getNode :: "(_) node_ptr ⇒ (_) heap ⇒ (_) Node option"
  where
    "getNode node_ptr h = Option.bind (get (cast node_ptr) h) cast"
adhoc_overloading get ⇌ getNode

locale l_type_wf_defNode
begin
definition a_type_wf :: "(_) heap ⇒ bool"
  where
    "a_type_wf h = (ObjectClass.type_wf h
                  ∧ (∀node_ptr ∈ fset( node_ptr_kinds h). getNode node_ptr h ≠ None))"
end
global_interpretation l_type_wf_defNode defines type_wf = a_type_wf ⟨proof⟩
lemmas type_wf_defs = a_type_wf_def

locale l_type_wfNode = l_type_wf type_wf for type_wf :: "((_) heap ⇒ bool)" +
  assumes type_wfNode: "type_wf h ⟹ NodeClass.type_wf h"

sublocale l_type_wfNode ⊆ l_type_wfObject
  ⟨proof⟩

locale l_getNode_lemmas = l_type_wfNode
begin
sublocale l_getObject_lemmas ⟨proof⟩
```

```
lemma get_{Node}_type_wf:
  assumes "type_wf h"
  shows "node_ptr |∈| node_ptr_kinds h ⟷ get_{Node} node_ptr h ≠ None"
  ⟨proof⟩
end

global_interpretation l_get_{Node}_lemmas type_wf
  ⟨proof⟩

definition put_{Node} :: "(_) node_ptr ⇒ (_) Node ⇒ (_) heap ⇒ (_) heap"
  where
    "put_{Node} node_ptr node = put (cast node_ptr) (cast node)"
adhoc_overloading put ⇌ put_{Node}

lemma put_{Node}_ptr_in_heap:
  assumes "put_{Node} node_ptr node h = h'"
  shows "node_ptr |∈| node_ptr_kinds h'"
  ⟨proof⟩

lemma put_{Node}_put_ptrs:
  assumes "put_{Node} node_ptr node h = h'"
  shows "object_ptr_kinds h' = object_ptr_kinds h |∪| {|cast node_ptr|}"
  ⟨proof⟩

lemma node_ptr_kinds_commutes [simp]:
  "cast node_ptr |∈| object_ptr_kinds h ⟷ node_ptr |∈| node_ptr_kinds h"
⟨proof⟩

lemma node_empty [simp]:
  "⦇RObject.nothing = (), RNode.nothing = (), ... = RNode.more node⦈ = node"
  ⟨proof⟩

lemma cast_{Node2Object}_inject [simp]: "cast_{Node2Object} x = cast_{Node2Object} y ⟷ x = y"
  ⟨proof⟩

lemma cast_{Object2Node}_none [simp]:
  "cast_{Object2Node} obj = None ⟷ ¬ (∃node. cast_{Node2Object} node = obj)"
  ⟨proof⟩

lemma cast_{Object2Node}_some [simp]: "cast_{Object2Node} obj = Some node ⟷ cast node = obj"
  ⟨proof⟩

lemma cast_{Object2Node}_inv [simp]: "cast_{Object2Node} (cast_{Node2Object} node) = Some node"
  ⟨proof⟩

locale l_known_ptr_{Node}
begin
definition a_known_ptr :: "(_) object_ptr ⇒ bool"
  where
    "a_known_ptr ptr = False"
end
global_interpretation l_known_ptr_{Node} defines known_ptr = a_known_ptr ⟨proof⟩
lemmas known_ptr_defs = a_known_ptr_def


locale l_known_ptrs_{Node} = l_known_ptr known_ptr for known_ptr :: "(_) object_ptr ⇒ bool"
begin
definition a_known_ptrs :: "(_) heap ⇒ bool"
  where
    "a_known_ptrs h = (∀ptr ∈ fset (object_ptr_kinds h). known_ptr ptr)"

lemma known_ptrs_known_ptr: "a_known_ptrs h ⟹ ptr |∈| object_ptr_kinds h ⟹ known_ptr ptr"
  ⟨proof⟩
```

```
lemma known_ptrs_preserved:
  "object_ptr_kinds h = object_ptr_kinds h' ⟹ a_known_ptrs h = a_known_ptrs h'"
  ⟨proof⟩
lemma known_ptrs_subset:
  "object_ptr_kinds h' |⊆| object_ptr_kinds h ⟹ a_known_ptrs h ⟹ a_known_ptrs h'"
  ⟨proof⟩
lemma known_ptrs_new_ptr:
  "object_ptr_kinds h' = object_ptr_kinds h |∪| {|new_ptr|} ⟹ known_ptr new_ptr ⟹
a_known_ptrs h ⟹ a_known_ptrs h'"
  ⟨proof⟩
end
global_interpretation l_known_ptrs_Node known_ptr defines known_ptrs = a_known_ptrs ⟨proof⟩
lemmas known_ptrs_defs = a_known_ptrs_def

lemma known_ptrs_is_l_known_ptrs: "l_known_ptrs known_ptr known_ptrs"
  ⟨proof⟩

lemma get_node_ptr_simp1 [simp]: "get_Node node_ptr (put_Node node_ptr node h) = Some node"
  ⟨proof⟩
lemma get_node_ptr_simp2 [simp]:
  "node_ptr ≠ node_ptr' ⟹ get_Node node_ptr (put_Node node_ptr' node h) = get_Node node_ptr h"
  ⟨proof⟩

end
```

4.4 Element (ElementClass)

In this theory, we introduce the types for the Element class.

```
theory ElementClass
  imports
    "NodeClass"
    "ShadowRootPointer"
begin
```

The type *DOMString* is a type synonym for *string*, define in section 6.

```
type_synonym attr_key = DOMString
type_synonym attr_value = DOMString
type_synonym attrs = "(attr_key, attr_value) fmap"
type_synonym tag_name = DOMString
record ('node_ptr, 'element_ptr, 'character_data_ptr, 'shadow_root_ptr) RElement = RNode +
  nothing :: unit
  tag_name :: tag_name
  child_nodes :: "('node_ptr, 'element_ptr, 'character_data_ptr) node_ptr list"
  attrs :: attrs
  shadow_root_opt :: "'shadow_root_ptr shadow_root_ptr option"
type_synonym
  ('node_ptr, 'element_ptr, 'character_data_ptr, 'shadow_root_ptr, 'Element) Element
  = "('node_ptr, 'element_ptr, 'character_data_ptr, 'shadow_root_ptr, 'Element option)
RElement_scheme"
register_default_tvars
  "('node_ptr, 'element_ptr, 'character_data_ptr, 'shadow_root_ptr, 'Element) Element"
type_synonym
  ('node_ptr, 'element_ptr, 'character_data_ptr, 'shadow_root_ptr, 'Node, 'Element) Node
  = "(('node_ptr, 'element_ptr, 'character_data_ptr, 'shadow_root_ptr, 'Element option) RElement_ext
+ 'Node) Node"
register_default_tvars
  "('node_ptr, 'element_ptr, 'character_data_ptr, 'shadow_root_ptr, 'Node, 'Element) Node"
type_synonym
  ('node_ptr, 'element_ptr, 'character_data_ptr, 'shadow_root_ptr, 'Object, 'Node, 'Element) Object
  = "('Object, ('node_ptr, 'element_ptr, 'character_data_ptr, 'shadow_root_ptr, 'Element option)
RElement_ext + 'Node) Object"
```

register_default_tvars
  "('node_ptr, 'element_ptr, 'character_data_ptr, 'shadow_root_ptr, 'Object, 'Node, 'Element) Object"

type_synonym
  ('object_ptr, 'node_ptr, 'element_ptr, 'character_data_ptr, 'document_ptr, 'shadow_root_ptr,
    'Object, 'Node, 'Element) heap
  = "('document_ptr document_ptr + 'shadow_root_ptr shadow_root_ptr + 'object_ptr,
'element_ptr element_ptr + 'character_data_ptr character_data_ptr + 'node_ptr, 'Object,
('node_ptr, 'element_ptr, 'character_data_ptr, 'shadow_root_ptr, 'Element option) RElement_ext +
'Node) heap"
register_default_tvars
  "('object_ptr, 'node_ptr, 'element_ptr, 'character_data_ptr, 'document_ptr, 'shadow_root_ptr,
'Object, 'Node, 'Element) heap"
type_synonym heap$_{final}$ = "(unit, unit, unit, unit, unit, unit, unit, unit, unit) heap"

definition element_ptr_kinds :: "(_) heap ⇒ (_) element_ptr fset"
  where
    "element_ptr_kinds heap =
the |'| (cast$_{node\_ptr2element\_ptr}$ |'| (ffilter is_element_ptr_kind (node_ptr_kinds heap)))"

lemma element_ptr_kinds_simp [simp]:
  "element_ptr_kinds (Heap (fmupd (cast element_ptr) element (the_heap h))) =
{|element_ptr|} |∪| element_ptr_kinds h"
  ⟨*proof*⟩

definition element_ptrs :: "(_) heap ⇒ (_) element_ptr fset"
  where
    "element_ptrs heap = ffilter is_element_ptr (element_ptr_kinds heap)"

definition cast$_{Node2Element}$ :: "(_) Node ⇒ (_) Element option"
  where
    "cast$_{Node2Element}$ node =
(case RNode.more node of Inl element ⇒ Some (RNode.extend (RNode.truncate node) element) | _ ⇒ None)"
adhoc_overloading cast ⇌ cast$_{Node2Element}$

abbreviation cast$_{Object2Element}$ :: "(_) Object ⇒ (_) Element option"
  where
    "cast$_{Object2Element}$ obj ≡ (case cast$_{Object2Node}$ obj of Some node ⇒ cast$_{Node2Element}$ node | None ⇒
None)"
adhoc_overloading cast ⇌ cast$_{Object2Element}$

definition cast$_{Element2Node}$ :: "(_) Element ⇒ (_) Node"
  where
    "cast$_{Element2Node}$ element = RNode.extend (RNode.truncate element) (Inl (RNode.more element))"
adhoc_overloading cast ⇌ cast$_{Element2Node}$

abbreviation cast$_{Element2Object}$ :: "(_) Element ⇒ (_) Object"
  where
    "cast$_{Element2Object}$ ptr ≡ cast$_{Node2Object}$ (cast$_{Element2Node}$ ptr)"
adhoc_overloading cast ⇌ cast$_{Element2Object}$

consts is_element_kind :: 'a
definition is_element_kind$_{Node}$ :: "(_) Node ⇒ bool"
  where
    "is_element_kind$_{Node}$ ptr ⟷ cast$_{Node2Element}$ ptr ≠ None"

adhoc_overloading is_element_kind ⇌ is_element_kind$_{Node}$
lemmas is_element_kind_def = is_element_kind$_{Node}$_def

abbreviation is_element_kind$_{Object}$ :: "(_) Object ⇒ bool"
  where
    "is_element_kind$_{Object}$ ptr ≡ cast$_{Object2Element}$ ptr ≠ None"
adhoc_overloading is_element_kind ⇌ is_element_kind$_{Object}$

```
lemma element_ptr_kinds_commutes [simp]:
  "cast element_ptr |∈| node_ptr_kinds h ⟷ element_ptr |∈| element_ptr_kinds h"
⟨proof⟩

definition get_{Element} :: "(_) element_ptr ⇒ (_) heap ⇒ (_) Element option"
  where
    "get_{Element} element_ptr h = Option.bind (get_{Node} (cast element_ptr) h) cast"
adhoc_overloading get ⇌ get_{Element}

locale l_type_wf_def_{Element}
begin
definition a_type_wf :: "(_) heap ⇒ bool"
  where
    "a_type_wf h = (NodeClass.type_wf h ∧ (∀element_ptr ∈ fset (element_ptr_kinds h).
                                                        get_{Element} element_ptr h ≠ None))"
end
global_interpretation l_type_wf_def_{Element} defines type_wf = a_type_wf ⟨proof⟩
lemmas type_wf_defs = a_type_wf_def

locale l_type_wf_{Element} = l_type_wf type_wf for type_wf :: "((_) heap ⇒ bool)" +
  assumes type_wf_{Element}: "type_wf h ⟹ ElementClass.type_wf h"

sublocale l_type_wf_{Element} ⊆ l_type_wf_{Node}
  ⟨proof⟩

locale l_get_{Element}_lemmas = l_type_wf_{Element}
begin
sublocale l_get_{Node}_lemmas ⟨proof⟩

lemma get_{Element}_type_wf:
  assumes "type_wf h"
  shows "element_ptr |∈| element_ptr_kinds h ⟷ get_{Element} element_ptr h ≠ None"
⟨proof⟩

end

global_interpretation l_get_{Element}_lemmas type_wf
  ⟨proof⟩

definition put_{Element} :: "(_) element_ptr ⇒ (_) Element ⇒ (_) heap ⇒ (_) heap"
  where
    "put_{Element} element_ptr element = put_{Node} (cast element_ptr) (cast element)"
adhoc_overloading put ⇌ put_{Element}

lemma put_{Element}_ptr_in_heap:
  assumes "put_{Element} element_ptr element h = h'"
  shows "element_ptr |∈| element_ptr_kinds h'"
  ⟨proof⟩

lemma put_{Element}_put_ptrs:
  assumes "put_{Element} element_ptr element h = h'"
  shows "object_ptr_kinds h' = object_ptr_kinds h |∪| {|cast element_ptr|}"
  ⟨proof⟩



lemma cast_{Element2Node}_inject [simp]:
  "cast_{Element2Node} x = cast_{Element2Node} y ⟷ x = y"
  ⟨proof⟩

lemma cast_{Node2Element}_none [simp]:
  "cast_{Node2Element} node = None ⟷ ¬ (∃element. cast_{Element2Node} element = node)"
```

⟨proof⟩

lemma cast$_{Node2Element}$_some [simp]:
"cast$_{Node2Element}$ node = Some element ⟷ cast$_{Element2Node}$ element = node"
⟨proof⟩

lemma cast$_{Node2Element}$_inv [simp]: "cast$_{Node2Element}$ (cast$_{Element2Node}$ element) = Some element"
⟨proof⟩

lemma get_elment_ptr_simp1 [simp]:
"get$_{Element}$ element_ptr (put$_{Element}$ element_ptr element h) = Some element"
⟨proof⟩
lemma get_elment_ptr_simp2 [simp]:
"element_ptr ≠ element_ptr'
⟹ get$_{Element}$ element_ptr (put$_{Element}$ element_ptr' element h) = get$_{Element}$ element_ptr h"
⟨proof⟩

```
abbreviation "create_element_obj tag_name_arg child_nodes_arg attrs_arg shadow_root_opt_arg
  ≡ ⦇ RObject.nothing = (), RNode.nothing = (), RElement.nothing = (),
      tag_name = tag_name_arg, Element.child_nodes = child_nodes_arg, attrs = attrs_arg,
      shadow_root_opt = shadow_root_opt_arg, ... = None ⦈"
```

definition new$_{Element}$ :: "(_) heap ⇒ ((_) element_ptr × (_) heap)"
where

```
    "newElement h =
      (let new_element_ptr = element_ptr.Ref (Suc (fMax (finsert 0 (element_ptr.the_ref
                                  |`| (element_ptrs h)))))
       in
      (new_element_ptr, put new_element_ptr (create_element_obj '''' [] fmempty None) h))"
```

lemma new$_{Element}$_ptr_in_heap:
assumes "new$_{Element}$ h = (new_element_ptr, h')"
shows "new_element_ptr |∈| element_ptr_kinds h'"
⟨proof⟩

lemma new_element_ptr_new:
"element_ptr.Ref (Suc (fMax (finsert 0 (element_ptr.the_ref |`| element_ptrs h)))) |∉| element_ptrs h"
⟨proof⟩

lemma new$_{Element}$_ptr_not_in_heap:
assumes "new$_{Element}$ h = (new_element_ptr, h')"
shows "new_element_ptr |∉| element_ptr_kinds h"
⟨proof⟩

lemma new$_{Element}$_new_ptr:
assumes "new$_{Element}$ h = (new_element_ptr, h')"
shows "object_ptr_kinds h' = object_ptr_kinds h |∪| {|cast new_element_ptr|}"
⟨proof⟩

lemma new$_{Element}$_is_element_ptr:
assumes "new$_{Element}$ h = (new_element_ptr, h')"
shows "is_element_ptr new_element_ptr"
⟨proof⟩

lemma new$_{Element}$_get$_{Object}$ [simp]:
assumes "new$_{Element}$ h = (new_element_ptr, h')"
assumes "ptr ≠ cast new_element_ptr"
shows "get$_{Object}$ ptr h = get$_{Object}$ ptr h'"
⟨proof⟩

lemma new$_{Element}$_get$_{Node}$ [simp]:
assumes "new$_{Element}$ h = (new_element_ptr, h')"

```
  assumes "ptr ≠ cast new_element_ptr"
  shows "get_Node ptr h = get_Node ptr h'"
  ⟨proof⟩

lemma new_Element_get_Element [simp]:
  assumes "new_Element h = (new_element_ptr, h')"
  assumes "ptr ≠ new_element_ptr"
  shows "get_Element ptr h = get_Element ptr h'"
  ⟨proof⟩

locale l_known_ptr_Element
begin
definition a_known_ptr :: "(_) object_ptr ⇒ bool"
  where
    "a_known_ptr ptr = (known_ptr ptr ∨ is_element_ptr ptr)"

lemma known_ptr_not_element_ptr: "¬is_element_ptr ptr ⟹ a_known_ptr ptr ⟹ known_ptr ptr"
  ⟨proof⟩
end
global_interpretation l_known_ptr_Element defines known_ptr = a_known_ptr ⟨proof⟩
lemmas known_ptr_defs = a_known_ptr_def


locale l_known_ptrs_Element = l_known_ptr known_ptr for known_ptr :: "(_) object_ptr ⇒ bool"
begin
definition a_known_ptrs :: "(_) heap ⇒ bool"
  where
    "a_known_ptrs h = (∀ptr ∈ fset (object_ptr_kinds h). known_ptr ptr)"

lemma known_ptrs_known_ptr:
  "ptr |∈| object_ptr_kinds h ⟹ a_known_ptrs h ⟹ known_ptr ptr"
  ⟨proof⟩

lemma known_ptrs_preserved:
  "object_ptr_kinds h = object_ptr_kinds h' ⟹ a_known_ptrs h = a_known_ptrs h'"
  ⟨proof⟩
lemma known_ptrs_subset:
  "object_ptr_kinds h' |⊆| object_ptr_kinds h ⟹ a_known_ptrs h ⟹ a_known_ptrs h'"
  ⟨proof⟩
lemma known_ptrs_new_ptr:
  "object_ptr_kinds h' = object_ptr_kinds h |∪| {|new_ptr|} ⟹ known_ptr new_ptr ⟹
a_known_ptrs h ⟹ a_known_ptrs h'"
  ⟨proof⟩
end
global_interpretation l_known_ptrs_Element known_ptr defines known_ptrs = a_known_ptrs ⟨proof⟩
lemmas known_ptrs_defs = a_known_ptrs_def

lemma known_ptrs_is_l_known_ptrs: "l_known_ptrs known_ptr known_ptrs"
  ⟨proof⟩

end
```

4.5 CharacterData (CharacterDataClass)

In this theory, we introduce the types for the CharacterData class.

```
theory CharacterDataClass
  imports
    ElementClass
begin
```

CharacterData

The type *DOMString* is a type synonym for *string*, defined section 6.

```
record RCharacterData = RNode +
  nothing :: unit
  val :: DOMString
register_default_tvars "'CharacterData RCharacterData_ext"
type_synonym 'CharacterData CharacterData = "'CharacterData option RCharacterData_scheme"
register_default_tvars "'CharacterData CharacterData"
type_synonym ('node_ptr, 'element_ptr, 'character_data_ptr, 'shadow_root_ptr, 'Node,
    'Element, 'CharacterData) Node
  = "('node_ptr, 'element_ptr, 'character_data_ptr, 'shadow_root_ptr,
      'CharacterData option RCharacterData_ext + 'Node, 'Element) Node"
register_default_tvars "('node_ptr, 'element_ptr, 'character_data_ptr, 'shadow_root_ptr, 'Node,
                        'Element, 'CharacterData) Node"
type_synonym ('node_ptr, 'element_ptr, 'character_data_ptr, 'shadow_root_ptr, 'Object, 'Node,
    'Element, 'CharacterData) Object
  = "('node_ptr, 'element_ptr, 'character_data_ptr, 'shadow_root_ptr, 'Object,
      'CharacterData option RCharacterData_ext + 'Node,
      'Element) Object"
register_default_tvars "('node_ptr, 'element_ptr, 'character_data_ptr, 'shadow_root_ptr, 'Object,
                        'Node, 'Element, 'CharacterData) Object"

type_synonym ('object_ptr, 'node_ptr, 'element_ptr, 'character_data_ptr, 'document_ptr,
    'shadow_root_ptr, 'Object, 'Node, 'Element, 'CharacterData) heap
  = "('object_ptr, 'node_ptr, 'element_ptr, 'character_data_ptr, 'document_ptr, 'shadow_root_ptr,
      'Object, 'CharacterData option RCharacterData_ext + 'Node, 'Element) heap"
register_default_tvars "('object_ptr, 'node_ptr, 'element_ptr, 'character_data_ptr, 'document_ptr,
                        'shadow_root_ptr, 'Object, 'Node, 'Element, 'CharacterData) heap"
type_synonym heap_final = "(unit, unit, unit, unit, unit, unit, unit, unit, unit, unit) heap"


definition character_data_ptr_kinds :: "(_) heap ⇒ (_) character_data_ptr fset"
  where
    "character_data_ptr_kinds heap = the |`| (cast |`| (ffilter is_character_data_ptr_kind
                                                         (node_ptr_kinds heap)))"

lemma character_data_ptr_kinds_simp [simp]:
  "character_data_ptr_kinds (Heap (fmupd (cast character_data_ptr) character_data (the_heap h)))
           = {|character_data_ptr|} |∪| character_data_ptr_kinds h"
  ⟨proof⟩

definition character_data_ptrs :: "(_) heap ⇒ _ character_data_ptr fset"
  where
    "character_data_ptrs heap = ffilter is_character_data_ptr (character_data_ptr_kinds heap)"

abbreviation "character_data_ptr_exts heap ≡ character_data_ptr_kinds heap - character_data_ptrs heap"

definition cast_Node2CharacterData :: "(_) Node ⇒ (_) CharacterData option"
  where
    "cast_Node2CharacterData node = (case RNode.more node of
          Inr (Inl character_data) ⇒ Some (RNode.extend (RNode.truncate node) character_data)
        | _ ⇒ None)"
adhoc_overloading cast ⇌ cast_Node2CharacterData

abbreviation cast_Object2CharacterData :: "(_) Object ⇒ (_) CharacterData  option"
  where
    "cast_Object2CharacterData obj ≡ (case cast_Object2Node obj of Some node ⇒ cast_Node2CharacterData node

                                                                | None ⇒ None)"
adhoc_overloading cast ⇌ cast_Object2CharacterData

definition cast_CharacterData2Node :: "(_) CharacterData ⇒ (_) Node"
```

```
  where
    "cast_{CharacterData2Node} character_data = RNode.extend (RNode.truncate character_data)
                                                   (Inr (Inl (RNode.more character_data)))"
adhoc_overloading cast ⇌ cast_{CharacterData2Node}

abbreviation cast_{CharacterData2Object} :: "(_) CharacterData ⇒ (_) Object"
  where
    "cast_{CharacterData2Object} ptr ≡ cast_{Node2Object} (cast_{CharacterData2Node} ptr)"
adhoc_overloading cast ⇌ cast_{CharacterData2Object}

consts is_character_data_kind :: 'a
definition is_character_data_kind_{Node} :: "(_) Node ⇒ bool"
  where
    "is_character_data_kind_{Node} ptr ⟷ cast_{Node2CharacterData} ptr ≠ None"

adhoc_overloading is_character_data_kind ⇌ is_character_data_kind_{Node}
lemmas is_character_data_kind_def = is_character_data_kind_{Node}_def

abbreviation is_character_data_kind_{Object} :: "(_) Object ⇒ bool"
  where
    "is_character_data_kind_{Object} ptr ≡ cast_{Object2CharacterData} ptr ≠ None"
adhoc_overloading is_character_data_kind ⇌ is_character_data_kind_{Object}

lemma character_data_ptr_kinds_commutes [simp]:
  "cast character_data_ptr |∈| node_ptr_kinds h
      ⟷ character_data_ptr |∈| character_data_ptr_kinds h"
⟨proof⟩

definition get_{CharacterData} :: "(_) character_data_ptr ⇒ (_) heap ⇒ (_) CharacterData option"
  where
    "get_{CharacterData} character_data_ptr h = Option.bind (get_{Node} (cast character_data_ptr) h) cast"
adhoc_overloading get ⇌ get_{CharacterData}

locale l_type_wf_def_{CharacterData}
begin
definition a_type_wf :: "(_) heap ⇒ bool"
  where
    "a_type_wf h = (ElementClass.type_wf h
                ∧ (∀character_data_ptr ∈ fset (character_data_ptr_kinds h).
                    get_{CharacterData} character_data_ptr h ≠ None))"
end
global_interpretation l_type_wf_def_{CharacterData} defines type_wf = a_type_wf ⟨proof⟩
lemmas type_wf_defs = a_type_wf_def

locale l_type_wf_{CharacterData} = l_type_wf type_wf for type_wf :: "((_) heap ⇒ bool)" +
  assumes type_wf_{CharacterData}: "type_wf h ⟹ CharacterDataClass.type_wf h"

sublocale l_type_wf_{CharacterData} ⊆ l_type_wf_{Element}
  ⟨proof⟩

locale l_get_{CharacterData}_lemmas = l_type_wf_{CharacterData}
begin
sublocale l_get_{Element}_lemmas ⟨proof⟩

lemma get_{CharacterData}_type_wf:
  assumes "type_wf h"
  shows "character_data_ptr |∈| character_data_ptr_kinds h
          ⟷ get_{CharacterData} character_data_ptr h ≠ None"
  ⟨proof⟩
end

global_interpretation l_get_{CharacterData}_lemmas type_wf
  ⟨proof⟩
```

definition $put_{CharacterData}$ :: "(_) character_data_ptr ⇒ (_) CharacterData ⇒ (_) heap ⇒ (_) heap"
  where
    "$put_{CharacterData}$ character_data_ptr character_data = put_{Node} (cast character_data_ptr)
                    (cast character_data)"
adhoc_overloading put ⇌ $put_{CharacterData}$

lemma $put_{CharacterData}$_ptr_in_heap:
  assumes "$put_{CharacterData}$ character_data_ptr character_data h = h'"
  shows "character_data_ptr |∈| character_data_ptr_kinds h'"
  ⟨proof⟩

lemma $put_{CharacterData}$_put_ptrs:
  assumes "$put_{CharacterData}$ character_data_ptr character_data h = h'"
  shows "object_ptr_kinds h' = object_ptr_kinds h |∪| {|cast character_data_ptr|}"
  ⟨proof⟩

lemma $cast_{CharacterData2Node}$_inject [simp]: "$cast_{CharacterData2Node}$ x = $cast_{CharacterData2Node}$ y ⟷ x = y"
  ⟨proof⟩

lemma $cast_{Node2CharacterData}$_none [simp]:
  "$cast_{Node2CharacterData}$ node = None ⟷ ¬ (∃character_data. $cast_{CharacterData2Node}$ character_data = node)"
  ⟨proof⟩

lemma $cast_{Node2CharacterData}$_some [simp]:
  "$cast_{Node2CharacterData}$ node = Some character_data ⟷ $cast_{CharacterData2Node}$ character_data = node"
  ⟨proof⟩

lemma $cast_{Node2CharacterData}$_inv [simp]:
  "$cast_{Node2CharacterData}$ ($cast_{CharacterData2Node}$ character_data) = Some character_data"
  ⟨proof⟩

lemma cast_element_not_character_data [simp]:
  "($cast_{Element2Node}$ element ≠ $cast_{CharacterData2Node}$ character_data)"
  "($cast_{CharacterData2Node}$ character_data ≠ $cast_{Element2Node}$ element)"
  ⟨proof⟩

lemma get_CharacterData_simp1 [simp]:
  "$get_{CharacterData}$ character_data_ptr ($put_{CharacterData}$ character_data_ptr character_data h)
      = Some character_data"
  ⟨proof⟩
lemma get_CharacterData_simp2 [simp]:
  "character_data_ptr ≠ character_data_ptr' ⟹ $get_{CharacterData}$ character_data_ptr
        ($put_{CharacterData}$ character_data_ptr' character_data h) = $get_{CharacterData}$ character_data_ptr h"
  ⟨proof⟩

lemma get_CharacterData_simp3 [simp]:
  "$get_{Element}$ element_ptr ($put_{CharacterData}$ character_data_ptr f h) = $get_{Element}$ element_ptr h"
  ⟨proof⟩
lemma get_CharacterData_simp4 [simp]:
  "$get_{CharacterData}$ element_ptr ($put_{Element}$ character_data_ptr f h) = $get_{CharacterData}$ element_ptr h"
  ⟨proof⟩

lemma $new_{Element}$_$get_{CharacterData}$ [simp]:
  assumes "$new_{Element}$ h = (new_element_ptr, h')"
  shows "$get_{CharacterData}$ ptr h = $get_{CharacterData}$ ptr h'"
  ⟨proof⟩

abbreviation "create_character_data_obj val_arg

```
  ≡ (| RObject.nothing = (), RNode.nothing = (), RCharacterData.nothing = (), val = val_arg, ... = None |)"
```

definition $\mathtt{new}_{CharacterData}$ `:: "(_) heap ⇒ ((_) character_data_ptr × (_) heap)"`
  where
```
    "newCharacterData h =
      (let new_character_data_ptr = character_data_ptr.Ref (Suc (fMax (character_data_ptr.the_ref
            |`| (character_data_ptrs h)))) in
        (new_character_data_ptr, put new_character_data_ptr (create_character_data_obj '''') h))"
```

lemma $\mathtt{new}_{CharacterData}$`_ptr_in_heap:`
  assumes "$\mathtt{new}_{CharacterData}$ `h = (new_character_data_ptr, h')"`
  shows `"new_character_data_ptr |∈| character_data_ptr_kinds h'"`
  ⟨*proof*⟩

lemma `new_character_data_ptr_new:`
```
  "character_data_ptr.Ref (Suc (fMax (finsert 0 (character_data_ptr.the_ref |`| character_data_ptrs h))))

              |∉| character_data_ptrs h"
```
  ⟨*proof*⟩

lemma $\mathtt{new}_{CharacterData}$`_ptr_not_in_heap:`
  assumes "$\mathtt{new}_{CharacterData}$ `h = (new_character_data_ptr, h')"`
  shows `"new_character_data_ptr |∉| character_data_ptr_kinds h"`
  ⟨*proof*⟩

lemma $\mathtt{new}_{CharacterData}$`_new_ptr:`
  assumes "$\mathtt{new}_{CharacterData}$ `h = (new_character_data_ptr, h')"`
  shows `"object_ptr_kinds h' = object_ptr_kinds h |∪| {|cast new_character_data_ptr|}"`
  ⟨*proof*⟩

lemma $\mathtt{new}_{CharacterData}$`_is_character_data_ptr:`
  assumes "$\mathtt{new}_{CharacterData}$ `h = (new_character_data_ptr, h')"`
  shows `"is_character_data_ptr new_character_data_ptr"`
  ⟨*proof*⟩

lemma $\mathtt{new}_{CharacterData}$`_`$\mathtt{get}_{Object}$ `[simp]:`
  assumes "$\mathtt{new}_{CharacterData}$ `h = (new_character_data_ptr, h')"`
  assumes `"ptr ≠ cast new_character_data_ptr"`
  shows "$\mathtt{get}_{Object}$ `ptr h =` $\mathtt{get}_{Object}$ `ptr h'"`
  ⟨*proof*⟩

lemma $\mathtt{new}_{CharacterData}$`_`$\mathtt{get}_{Node}$ `[simp]:`
  assumes "$\mathtt{new}_{CharacterData}$ `h = (new_character_data_ptr, h')"`
  assumes `"ptr ≠ cast new_character_data_ptr"`
  shows "$\mathtt{get}_{Node}$ `ptr h =` $\mathtt{get}_{Node}$ `ptr h'"`
  ⟨*proof*⟩

lemma $\mathtt{new}_{CharacterData}$`_`$\mathtt{get}_{Element}$ `[simp]:`
  assumes "$\mathtt{new}_{CharacterData}$ `h = (new_character_data_ptr, h')"`
  shows "$\mathtt{get}_{Element}$ `ptr h =` $\mathtt{get}_{Element}$ `ptr h'"`
  ⟨*proof*⟩

lemma $\mathtt{new}_{CharacterData}$`_`$\mathtt{get}_{CharacterData}$ `[simp]:`
  assumes "$\mathtt{new}_{CharacterData}$ `h = (new_character_data_ptr, h')"`
  assumes `"ptr ≠ new_character_data_ptr"`
  shows "$\mathtt{get}_{CharacterData}$ `ptr h =` $\mathtt{get}_{CharacterData}$ `ptr h'"`
  ⟨*proof*⟩

locale `l_known_ptr`$_{CharacterData}$
begin
definition `a_known_ptr :: "(_) object_ptr ⇒ bool"`
  where

```
    "a_known_ptr ptr = (known_ptr ptr ∨ is_character_data_ptr ptr)"

lemma known_ptr_not_character_data_ptr:
  "¬is_character_data_ptr ptr ⟹ a_known_ptr ptr ⟹ known_ptr ptr"
  ⟨proof⟩
end
global_interpretation l_known_ptr_CharacterData defines known_ptr = a_known_ptr ⟨proof⟩
lemmas known_ptr_defs = a_known_ptr_def


locale l_known_ptrs_CharacterData = l_known_ptr known_ptr for known_ptr :: "(_) object_ptr ⇒ bool"
begin
definition a_known_ptrs :: "(_) heap ⇒ bool"
  where
    "a_known_ptrs h = (∀ptr ∈ fset (object_ptr_kinds h). known_ptr ptr)"

lemma known_ptrs_known_ptr: "a_known_ptrs h ⟹ ptr |∈| object_ptr_kinds h ⟹ known_ptr ptr"
  ⟨proof⟩

lemma known_ptrs_preserved:
  "object_ptr_kinds h = object_ptr_kinds h' ⟹ a_known_ptrs h = a_known_ptrs h'"
  ⟨proof⟩
lemma known_ptrs_subset:
  "object_ptr_kinds h' |⊆| object_ptr_kinds h ⟹ a_known_ptrs h ⟹ a_known_ptrs h'"
  ⟨proof⟩
lemma known_ptrs_new_ptr:
  "object_ptr_kinds h' = object_ptr_kinds h |∪| {|new_ptr|} ⟹ known_ptr new_ptr ⟹
a_known_ptrs h ⟹ a_known_ptrs h'"
  ⟨proof⟩
end
global_interpretation l_known_ptrs_CharacterData known_ptr defines known_ptrs = a_known_ptrs ⟨proof⟩
lemmas known_ptrs_defs = a_known_ptrs_def

lemma known_ptrs_is_l_known_ptrs: "l_known_ptrs known_ptr known_ptrs"
  ⟨proof⟩

end
```

4.6 Document (DocumentClass)

In this theory, we introduce the types for the Document class.

```
theory DocumentClass
  imports
    CharacterDataClass
begin
```

The type *doctype* is a type synonym for *string*, defined in section 6.

```
record ('node_ptr, 'element_ptr, 'character_data_ptr) RDocument = RObject +
  nothing :: unit
  doctype :: doctype
  document_element :: "(_) element_ptr option"
  disconnected_nodes :: "('node_ptr, 'element_ptr, 'character_data_ptr) node_ptr list"
type_synonym
  ('node_ptr, 'element_ptr, 'character_data_ptr, 'Document) Document
  = "('node_ptr, 'element_ptr, 'character_data_ptr, 'Document option) RDocument_scheme"
register_default_tvars
  "('node_ptr, 'element_ptr, 'character_data_ptr, 'Document) Document"
type_synonym
  ('node_ptr, 'element_ptr, 'character_data_ptr, 'shadow_root_ptr, 'Object, 'Node,
    'Element, 'CharacterData, 'Document) Object
  = "('node_ptr, 'element_ptr, 'character_data_ptr, 'shadow_root_ptr,
       ('node_ptr, 'element_ptr, 'character_data_ptr, 'Document option)
    RDocument_ext + 'Object, 'Node, 'Element, 'CharacterData) Object"
register_default_tvars "('node_ptr, 'element_ptr, 'character_data_ptr, 'shadow_root_ptr,
     'Object, 'Node, 'Element, 'CharacterData, 'Document) Object"

type_synonym ('object_ptr, 'node_ptr, 'element_ptr, 'character_data_ptr, 'document_ptr,
    'shadow_root_ptr, 'Object, 'Node, 'Element, 'CharacterData, 'Document) heap
  = "('object_ptr, 'node_ptr, 'element_ptr, 'character_data_ptr, 'document_ptr,
      'shadow_root_ptr,
      ('node_ptr, 'element_ptr, 'character_data_ptr, 'Document option) RDocument_ext + 'Object, 'Node,
      'Element, 'CharacterData) heap"
register_default_tvars
  "('object_ptr, 'node_ptr, 'element_ptr, 'character_data_ptr, 'document_ptr,
  'shadow_root_ptr, 'Object, 'Node, 'Element, 'CharacterData, 'Document) heap"
type_synonym heap_final = "(unit, unit, unit, unit, unit, unit, unit, unit, unit, unit, unit) heap"


definition document_ptr_kinds :: "(_) heap ⇒ (_) document_ptr fset"
  where
    "document_ptr_kinds heap = the |`| (cast_object_ptr2document_ptr |`|
                              (ffilter is_document_ptr_kind (object_ptr_kinds heap)))"

definition document_ptrs :: "(_) heap ⇒ (_) document_ptr fset"
  where
    "document_ptrs heap = ffilter is_document_ptr (document_ptr_kinds heap)"

definition cast_Object2Document :: "(_) Object ⇒ (_) Document option"
  where
    "cast_Object2Document obj = (case RObject.more obj of
      Inr (Inl document) ⇒ Some (RObject.extend (RObject.truncate obj) document)
    | _ ⇒ None)"
adhoc_overloading cast ⇌ cast_Object2Document

definition cast_Document2Object:: "(_) Document ⇒ (_) Object"
  where
    "cast_Document2Object document = (RObject.extend (RObject.truncate document)
                                          (Inr (Inl (RObject.more document))))"
adhoc_overloading cast ⇌ cast_Document2Object

definition is_document_kind :: "(_) Object ⇒ bool"
  where
    "is_document_kind ptr ⟷ cast_Object2Document ptr ≠ None"

lemma document_ptr_kinds_simp [simp]:
  "document_ptr_kinds (Heap (fmupd (cast document_ptr) document (the_heap h)))
        = {|document_ptr|} |∪| document_ptr_kinds h"
  ⟨proof⟩

lemma document_ptr_kinds_commutes [simp]:
  "cast document_ptr |∈| object_ptr_kinds h ⟷ document_ptr |∈| document_ptr_kinds h"
⟨proof⟩

definition get_Document :: "(_) document_ptr ⇒ (_) heap ⇒ (_) Document option"
  where
    "get_Document document_ptr h = Option.bind (get (cast document_ptr) h) cast"
adhoc_overloading get ⇌ get_Document

locale l_type_wf_def_Document
begin
definition a_type_wf :: "(_) heap ⇒ bool"
  where
    "a_type_wf h = (CharacterDataClass.type_wf h ∧
    (∀document_ptr ∈ fset (document_ptr_kinds h). get_Document document_ptr h ≠ None))"
end
```

```
global_interpretation l_type_wf_def_{Document} defines type_wf = a_type_wf ⟨proof⟩
lemmas type_wf_defs = a_type_wf_def

locale l_type_wf_{Document} = l_type_wf type_wf for type_wf :: "((_) heap ⇒ bool)" +
  assumes type_wf_{Document}: "type_wf h ⟹ DocumentClass.type_wf h"

sublocale l_type_wf_{Document} ⊆ l_type_wf_{CharacterData}
  ⟨proof⟩

locale l_get_{Document}_lemmas = l_type_wf_{Document}
begin
sublocale l_get_{CharacterData}_lemmas ⟨proof⟩
lemma get_{Document}_type_wf:
  assumes "type_wf h"
  shows "document_ptr |∈| document_ptr_kinds h ⟷ get_{Document} document_ptr h ≠ None"
  ⟨proof⟩
end

global_interpretation l_get_{Document}_lemmas type_wf ⟨proof⟩

definition put_{Document} :: "(_) document_ptr ⇒ (_) Document ⇒ (_) heap ⇒ (_) heap"
  where
    "put_{Document} document_ptr document = put (cast document_ptr) (cast document)"
adhoc_overloading put ⇌ put_{Document}

lemma put_{Document}_ptr_in_heap:
  assumes "put_{Document} document_ptr document h = h'"
  shows "document_ptr |∈| document_ptr_kinds h'"
  ⟨proof⟩

lemma put_{Document}_put_ptrs:
  assumes "put_{Document} document_ptr document h = h'"
  shows "object_ptr_kinds h' = object_ptr_kinds h |∪| {|cast document_ptr|}"
  ⟨proof⟩


lemma cast_{Document2Object}_inject [simp]: "cast_{Document2Object} x = cast_{Document2Object} y ⟷ x = y"
  ⟨proof⟩

lemma cast_{Object2Document}_none [simp]:
  "cast_{Object2Document} obj = None ⟷ ¬ (∃document. cast_{Document2Object} document = obj)"
  ⟨proof⟩

lemma cast_{Object2Document}_some [simp]:
  "cast_{Object2Document} obj = Some document ⟷ cast document = obj"
  ⟨proof⟩

lemma cast_{Object2Document}_inv [simp]: "cast_{Object2Document} (cast_{Document2Object} document) = Some document"
  ⟨proof⟩

lemma cast_document_not_node [simp]:
  "cast_{Document2Object} document ≠ cast_{Node2Object} node"
  "cast_{Node2Object} node ≠ cast_{Document2Object} document"
  ⟨proof⟩

lemma get_document_ptr_simp1 [simp]:
  "get_{Document} document_ptr (put_{Document} document_ptr document h) = Some document"
  ⟨proof⟩
lemma get_document_ptr_simp2 [simp]:
  "document_ptr ≠ document_ptr'
   ⟹ get_{Document} document_ptr (put_{Document} document_ptr' document h) = get_{Document} document_ptr h"
  ⟨proof⟩
```

lemma *get_document_ptr_simp3 [simp]:*
  "get$_{Element}$ element_ptr (put$_{Document}$ document_ptr f h) = get$_{Element}$ element_ptr h"
  ⟨*proof*⟩
lemma *get_document_ptr_simp4 [simp]:*
  "get$_{Document}$ document_ptr (put$_{Element}$ element_ptr f h) = get$_{Document}$ document_ptr h"
  ⟨*proof*⟩
lemma *get_document_ptr_simp5 [simp]:*
  "get$_{CharacterData}$ character_data_ptr (put$_{Document}$ document_ptr f h) = get$_{CharacterData}$ character_data_ptr h"
  ⟨*proof*⟩
lemma *get_document_ptr_simp6 [simp]:*
  "get$_{Document}$ document_ptr (put$_{CharacterData}$ character_data_ptr f h) = get$_{Document}$ document_ptr h"
  ⟨*proof*⟩

lemma *new$_{Element}$_get$_{Document}$ [simp]:*
  assumes *"new$_{Element}$ h = (new_element_ptr, h')"*
  shows *"get$_{Document}$ ptr h = get$_{Document}$ ptr h'"*
  ⟨*proof*⟩

lemma *new$_{CharacterData}$_get$_{Document}$ [simp]:*
  assumes *"new$_{CharacterData}$ h = (new_character_data_ptr, h')"*
  shows *"get$_{Document}$ ptr h = get$_{Document}$ ptr h'"*
  ⟨*proof*⟩

abbreviation
  create_document_obj :: "char list ⇒ (_) element_ptr option ⇒ (_) node_ptr list ⇒ (_) Document"
  where
    "create_document_obj doctype_arg document_element_arg disconnected_nodes_arg
  ≡ ⦇ RObject.nothing = (), RDocument.nothing = (), doctype = doctype_arg,
      document_element = document_element_arg,
      disconnected_nodes = disconnected_nodes_arg, ... = None ⦈"

definition *new$_{Document}$ :: "(_)heap ⇒ ((_) document_ptr × (_) heap)"*
  where
    "new$_{Document}$ h =
     (let new_document_ptr = document_ptr.Ref (Suc (fMax (finsert 0 (document_ptr.the_ref |`| (document_ptrs h)))))
        in
         (new_document_ptr, put new_document_ptr (create_document_obj '''' None []) h))"

lemma *new$_{Document}$_ptr_in_heap:*
  assumes *"new$_{Document}$ h = (new_document_ptr, h')"*
  shows *"new_document_ptr |∈| document_ptr_kinds h'"*
  ⟨*proof*⟩

lemma *new_document_ptr_new:*
  "document_ptr.Ref (Suc (fMax (finsert 0 (document_ptr.the_ref |`| document_ptrs h))))
      |∉| document_ptrs h"
  ⟨*proof*⟩

lemma *new$_{Document}$_ptr_not_in_heap:*
  assumes *"new$_{Document}$ h = (new_document_ptr, h')"*
  shows *"new_document_ptr |∉| document_ptr_kinds h"*
  ⟨*proof*⟩

lemma *new$_{Document}$_new_ptr:*
  assumes *"new$_{Document}$ h = (new_document_ptr, h')"*
  shows *"object_ptr_kinds h' = object_ptr_kinds h |∪| {|cast new_document_ptr|}"*
  ⟨*proof*⟩

```
lemma new_Document_is_document_ptr:
  assumes "new_Document h = (new_document_ptr, h')"
  shows "is_document_ptr new_document_ptr"
  ⟨proof⟩

lemma new_Document_get_Object [simp]:
  assumes "new_Document h = (new_document_ptr, h')"
  assumes "ptr ≠ cast new_document_ptr"
  shows "get_Object ptr h = get_Object ptr h'"
  ⟨proof⟩

lemma new_Document_get_Node [simp]:
  assumes "new_Document h = (new_document_ptr, h')"
  shows "get_Node ptr h = get_Node ptr h'"
  ⟨proof⟩

lemma new_Document_get_Element [simp]:
  assumes "new_Document h = (new_document_ptr, h')"
  shows "get_Element ptr h = get_Element ptr h'"
  ⟨proof⟩

lemma new_Document_get_CharacterData [simp]:
  assumes "new_Document h = (new_document_ptr, h')"
  shows "get_CharacterData ptr h = get_CharacterData ptr h'"
  ⟨proof⟩

lemma new_Document_get_Document [simp]:
  assumes "new_Document h = (new_document_ptr, h')"
  assumes "ptr ≠ new_document_ptr"
  shows "get_Document ptr h = get_Document ptr h'"
  ⟨proof⟩


locale l_known_ptr_Document
begin
definition a_known_ptr :: "(_) object_ptr ⇒ bool"
  where
    "a_known_ptr ptr = (known_ptr ptr ∨ is_document_ptr ptr)"

lemma known_ptr_not_document_ptr: "¬is_document_ptr ptr ⟹ a_known_ptr ptr ⟹ known_ptr ptr"
  ⟨proof⟩
end
global_interpretation l_known_ptr_Document defines known_ptr = a_known_ptr ⟨proof⟩
lemmas known_ptr_defs = a_known_ptr_def


locale l_known_ptrs_Document = l_known_ptr known_ptr for known_ptr :: "(_) object_ptr ⇒ bool"
begin
definition a_known_ptrs :: "(_) heap ⇒ bool"
  where
    "a_known_ptrs h = (∀ptr ∈ fset (object_ptr_kinds h). known_ptr ptr)"

lemma known_ptrs_known_ptr: "a_known_ptrs h ⟹ ptr |∈| object_ptr_kinds h ⟹ known_ptr ptr"
  ⟨proof⟩

lemma known_ptrs_preserved:
  "object_ptr_kinds h = object_ptr_kinds h' ⟹ a_known_ptrs h = a_known_ptrs h'"
  ⟨proof⟩
lemma known_ptrs_subset:
  "object_ptr_kinds h' |⊆| object_ptr_kinds h ⟹ a_known_ptrs h ⟹ a_known_ptrs h'"
  ⟨proof⟩
lemma known_ptrs_new_ptr:
  "object_ptr_kinds h' = object_ptr_kinds h |∪| {|new_ptr|} ⟹ known_ptr new_ptr ⟹
```

a_known_ptrs h $\Longrightarrow$ a_known_ptrs h'"
  ⟨*proof*⟩
end
global_interpretation l_known_ptrs$_{Document}$ known_ptr **defines** known_ptrs = a_known_ptrs ⟨*proof*⟩
lemmas known_ptrs_defs = a_known_ptrs_def

lemma known_ptrs_is_l_known_ptrs [instances]: "l_known_ptrs known_ptr known_ptrs"
  ⟨*proof*⟩

end

5 Monadic Object Constructors and Accessors

In this chapter, we introduce the moandic method definitions for the classes of our DOM formalization. Again the overall structure follows the same structure as for the class types and the pointer types.

5.1 The Monad Infrastructure (BaseMonad)

In this theory, we introduce the basic infrastructure for our monadic class encoding.

```
theory BaseMonad
  imports
    "../classes/BaseClass"
    "../preliminaries/Heap_Error_Monad"
begin
```

5.1.1 Datatypes

```
datatype exception = NotFoundError | HierarchyRequestError | NotSupportedError | SegmentationFault
  | AssertException | NonTerminationException | InvokeError | TypeError

consts put_M :: 'a
consts get_M :: 'a
consts delete_M :: 'a

lemma sorted_list_of_set_eq [dest]:
  "sorted_list_of_set (fset x) = sorted_list_of_set (fset y) ⟹ x = y"
  ⟨proof⟩


locale l_ptr_kinds_M =
  fixes ptr_kinds :: "'heap ⇒ 'ptr::linorder fset"
begin
definition a_ptr_kinds_M :: "('heap, exception, 'ptr list) prog"
  where
    "a_ptr_kinds_M = do {
      h ← get_heap;
      return (sorted_list_of_set (fset (ptr_kinds h)))
    }"

lemma ptr_kinds_M_ok [simp]: "h ⊢ ok a_ptr_kinds_M"
  ⟨proof⟩

lemma ptr_kinds_M_pure [simp]: "pure a_ptr_kinds_M h"
  ⟨proof⟩

lemma ptr_kinds_ptr_kinds_M [simp]: "ptr ∈ set |h ⊢ a_ptr_kinds_M|ᵣ ⟷ ptr |∈| ptr_kinds h"
  ⟨proof⟩

lemma ptr_kinds_M_ptr_kinds [simp]:
  "h ⊢ a_ptr_kinds_M →ᵣ xa ⟷ xa = sorted_list_of_set (fset (ptr_kinds h))"
  ⟨proof⟩
lemma ptr_kinds_M_ptr_kinds_returns_result [simp]:
  "h ⊢ a_ptr_kinds_M ≫= f →ᵣ x ⟷ h ⊢ f (sorted_list_of_set (fset (ptr_kinds h))) →ᵣ x"
  ⟨proof⟩
lemma ptr_kinds_M_ptr_kinds_returns_heap [simp]:
  "h ⊢ a_ptr_kinds_M ≫= f →ₕ h' ⟷ h ⊢ f (sorted_list_of_set (fset (ptr_kinds h))) →ₕ h'"
  ⟨proof⟩
```

```
end

locale l_get_M =
  fixes get :: "'ptr ⇒ 'heap ⇒ 'obj option"
  fixes type_wf :: "'heap ⇒ bool"
  fixes ptr_kinds :: "'heap ⇒ 'ptr fset"
  assumes "type_wf h ⟹ ptr |∈| ptr_kinds h ⟹ get ptr h ≠ None"
  assumes "get ptr h ≠ None ⟹ ptr |∈| ptr_kinds h"
begin

definition a_get_M :: "'ptr ⇒ ('obj ⇒ 'result) ⇒  ('heap, exception, 'result) prog"
  where
    "a_get_M ptr getter = (do {
      h ← get_heap;
      (case get ptr h of
        Some res ⇒ return (getter res)
      | None ⇒ error SegmentationFault)
    })"

lemma get_M_pure [simp]: "pure (a_get_M ptr getter) h"
  ⟨proof⟩

lemma get_M_ok:
  "type_wf h ⟹ ptr |∈| ptr_kinds h ⟹ h ⊢ ok (a_get_M ptr getter)"
  ⟨proof⟩
lemma get_M_ptr_in_heap:
  "h ⊢ ok (a_get_M ptr getter) ⟹ ptr |∈| ptr_kinds h"
  ⟨proof⟩

end

locale l_put_M = l_get_M get for get :: "'ptr ⇒ 'heap ⇒ 'obj option" +
  fixes put :: "'ptr ⇒ 'obj ⇒ 'heap ⇒ 'heap"
begin
definition a_put_M :: "'ptr ⇒ (('v ⇒ 'v) ⇒ 'obj ⇒ 'obj) ⇒ 'v ⇒ ('heap, exception, unit) prog"
  where
    "a_put_M ptr setter v = (do {
      obj ← a_get_M ptr id;
      h ← get_heap;
      return_heap (put ptr (setter (λ_. v) obj) h)
    })"

lemma put_M_ok:
  "type_wf h ⟹ ptr |∈| ptr_kinds h ⟹ h ⊢ ok (a_put_M ptr setter v)"
  ⟨proof⟩

lemma put_M_ptr_in_heap:
  "h ⊢ ok (a_put_M ptr setter v) ⟹ ptr |∈| ptr_kinds h"
  ⟨proof⟩

end
```

5.1.2 Setup for Defining Partial Functions

```
lemma execute_admissible:
  "ccpo.admissible (fun_lub (flat_lub (Inl (e::'e)))) (fun_ord (flat_ord (Inl e)))
     ((λa. ∀(h::'heap) h2 (r::'result). h ⊢ a = Inr (r, h2) ⟶ P h h2 r) ∘ Prog)"
⟨proof⟩

lemma execute_admissible2:
  "ccpo.admissible (fun_lub (flat_lub (Inl (e::'e)))) (fun_ord (flat_ord (Inl e)))
     ((λa. ∀(h::'heap) h' h2 h2' (r::'result) r'.
                   h ⊢ a = Inr (r, h2) ⟶ h' ⊢ a = Inr (r', h2') ⟶ P h h' h2 h2' r r') ∘ Prog)"
```

⟨proof⟩

```
definition dom_prog_ord ::
  "('heap, exception, 'result) prog ⇒ ('heap, exception, 'result) prog ⇒ bool" where
  "dom_prog_ord = img_ord (λa b. execute b a) (fun_ord (flat_ord (Inl NonTerminationException)))"

definition dom_prog_lub ::
  "('heap, exception, 'result) prog set ⇒ ('heap, exception, 'result) prog" where
  "dom_prog_lub = img_lub (λa b. execute b a) Prog (fun_lub (flat_lub (Inl NonTerminationException)))"

lemma dom_prog_lub_empty: "dom_prog_lub {} = error NonTerminationException"
  ⟨proof⟩

lemma dom_prog_interpretation: "partial_function_definitions dom_prog_ord dom_prog_lub"
⟨proof⟩

interpretation dom_prog: partial_function_definitions dom_prog_ord dom_prog_lub
  rewrites "dom_prog_lub {} ≡ error NonTerminationException"
  ⟨proof⟩

lemma admissible_dom_prog:
  "dom_prog.admissible (λf. ∀x h h' r. h ⊢ f x →_r r ⟶ h ⊢ f x →_h h' ⟶ P x h h' r)"
⟨proof⟩

lemma admissible_dom_prog2:
  "dom_prog.admissible (λf. ∀x h h2 h' h2' r r2. h ⊢ f x →_r r ⟶ h ⊢ f x →_h h'
           ⟶ h2 ⊢ f x →_r r2 ⟶ h2 ⊢ f x →_h h2' ⟶ P x h h2 h' h2' r r2)"
⟨proof⟩

lemma fixp_induct_dom_prog:
  fixes F :: "'c ⇒ 'c" and
    U :: "'c ⇒ 'b ⇒ ('heap, exception, 'result) prog" and
    C :: "('b ⇒ ('heap, exception, 'result) prog) ⇒ 'c" and
    P :: "'b ⇒ 'heap ⇒ 'heap ⇒ 'result ⇒ bool"
  assumes mono: "⋀x. monotone (fun_ord dom_prog_ord) dom_prog_ord (λf. U (F (C f)) x)"
  assumes eq: "f ≡ C (ccpo.fixp (fun_lub dom_prog_lub) (fun_ord dom_prog_ord) (λf. U (F (C f))))"
  assumes inverse2: "⋀f. U (C f) = f"
  assumes step: "⋀f x h h' r. (⋀x h h' r. h ⊢ (U f x) →_r r ⟹ h ⊢ (U f x) →_h h' ⟹ P x h h' r)
    ⟹ h ⊢ (U (F f) x) →_r r ⟹ h ⊢ (U (F f) x) →_h h' ⟹ P x h h' r"
  assumes defined: "h ⊢ (U f x) →_r r" and "h ⊢ (U f x) →_h h'"
  shows "P x h h' r"
  ⟨proof⟩

⟨ML⟩


abbreviation "mono_dom_prog ≡ monotone (fun_ord dom_prog_ord) dom_prog_ord"

lemma dom_prog_ordI:
  assumes "⋀h. h ⊢ f →_e NonTerminationException ∨ h ⊢ f = h ⊢ g"
  shows "dom_prog_ord f g"
⟨proof⟩

lemma dom_prog_ordE:
  assumes "dom_prog_ord x y"
  obtains "h ⊢ x →_e NonTerminationException" | " h ⊢ x = h ⊢ y"
  ⟨proof⟩


lemma bind_mono [partial_function_mono]:
  fixes B :: "('a ⇒ ('heap, exception, 'result) prog) ⇒ ('heap, exception, 'result2) prog"
  assumes mf: "mono_dom_prog B" and mg: "⋀y. mono_dom_prog (λf. C y f)"
  shows "mono_dom_prog (λf. B f ⤜ (λy. C y f))"
```

```
⟨proof⟩

lemma mono_dom_prog1 [partial_function_mono]:
  fixes g ::  "('a ⇒ ('heap, exception, 'result) prog) ⇒ 'b ⇒ ('heap, exception, 'result) prog"
  assumes "⋀x. (mono_dom_prog (λf. g f x))"
  shows "mono_dom_prog (λf. map_M (g f) xs)"
  ⟨proof⟩

lemma mono_dom_prog2 [partial_function_mono]:
  fixes g ::  "('a ⇒ ('heap, exception, 'result) prog) ⇒ 'b ⇒ ('heap, exception, 'result) prog"
  assumes "⋀x. (mono_dom_prog (λf. g f x))"
  shows "mono_dom_prog (λf. forall_M (g f) xs)"
  ⟨proof⟩

lemma sorted_list_set_cong [simp]:
  "sorted_list_of_set (fset FS) = sorted_list_of_set (fset FS') ⟷ FS = FS'"
  ⟨proof⟩

end
```

5.2 Object (ObjectMonad)

In this theory, we introduce the monadic method setup for the Object class.

```
theory ObjectMonad
  imports
    BaseMonad
    "../classes/ObjectClass"
begin

type_synonym ('object_ptr, 'Object, 'result) dom_prog
  = "((_) heap, exception, 'result) prog"
register_default_tvars "('object_ptr, 'Object, 'result) dom_prog"

global_interpretation l_ptr_kinds_M object_ptr_kinds defines object_ptr_kinds_M = a_ptr_kinds_M ⟨proof⟩
lemmas object_ptr_kinds_M_defs = a_ptr_kinds_M_def


global_interpretation l_dummy defines get_M_Object = "l_get_M.a_get_M get_Object" ⟨proof⟩
lemma get_M_is_l_get_M: "l_get_M get_Object type_wf object_ptr_kinds"
  ⟨proof⟩
lemmas get_M_defs = get_M_Object_def[unfolded l_get_M.a_get_M_def[OF get_M_is_l_get_M]]

adhoc_overloading get_M ⇌ get_M_Object

locale l_get_M_Object_lemmas = l_type_wf_Object
begin
interpretation l_get_M get_Object type_wf object_ptr_kinds
  ⟨proof⟩
lemmas get_M_Object_ok = get_M_ok[folded get_M_Object_def]
lemmas get_M_Object_ptr_in_heap = get_M_ptr_in_heap[folded get_M_Object_def]
end

global_interpretation l_get_M_Object_lemmas type_wf
  ⟨proof⟩

lemma object_ptr_kinds_M_reads:
  "reads (⋃object_ptr. {preserved (get_M_Object object_ptr RObject.nothing)}) object_ptr_kinds_M h h'"
  ⟨proof⟩


global_interpretation l_put_M type_wf object_ptr_kinds get_Object put_Object
  rewrites "a_get_M = get_M_Object"
  defines put_M_Object = a_put_M
    ⟨proof⟩
lemmas put_M_defs = a_put_M_def
adhoc_overloading put_M ⇌ put_M_Object


locale l_put_M_Object_lemmas = l_type_wf_Object
begin
interpretation l_put_M  type_wf object_ptr_kinds get_Object put_Object
  ⟨proof⟩
lemmas put_M_Object_ok = put_M_ok[folded put_M_Object_def]
lemmas put_M_Object_ptr_in_heap = put_M_ptr_in_heap[folded put_M_Object_def]
end

global_interpretation l_put_M_Object_lemmas type_wf
  ⟨proof⟩


definition check_in_heap :: "(_) object_ptr ⇒ (_, unit) dom_prog"
  where
    "check_in_heap ptr = do {
      h ← get_heap;
      (if ptr |∈| object_ptr_kinds h then
        return ()
      else
        error SegmentationFault
      )}"

lemma check_in_heap_ptr_in_heap: "ptr |∈| object_ptr_kinds h ⟷ h ⊢ ok (check_in_heap ptr)"
  ⟨proof⟩
lemma check_in_heap_pure [simp]: "pure (check_in_heap ptr) h"
  ⟨proof⟩
lemma check_in_heap_is_OK [simp]:
  "ptr |∈| object_ptr_kinds h ⟹ h ⊢ ok (check_in_heap ptr ≫= f) = h ⊢ ok (f ())"
  ⟨proof⟩
lemma check_in_heap_returns_result [simp]:
  "ptr |∈| object_ptr_kinds h ⟹ h ⊢ (check_in_heap ptr ≫= f) →_r x = h ⊢ f () →_r x"
  ⟨proof⟩
lemma check_in_heap_returns_heap [simp]:
  "ptr |∈| object_ptr_kinds h ⟹ h ⊢ (check_in_heap ptr ≫= f) →_h h' = h ⊢ f () →_h h'"
  ⟨proof⟩

lemma check_in_heap_reads:
  "reads {preserved (get_M object_ptr nothing)} (check_in_heap object_ptr) h h'"
  ⟨proof⟩
```

5.2.1 Invoke

```
fun invoke_rec :: "(((_) object_ptr ⇒ bool) × ((_) object_ptr ⇒ 'args
                ⇒ (_, 'result) dom_prog)) list ⇒ (_) object_ptr ⇒ 'args
                ⇒ (_, 'result) dom_prog"
  where
    "invoke_rec ((P, f)#xs) ptr args = (if P ptr then f ptr args else invoke_rec xs ptr args)"
  | "invoke_rec [] ptr args = error InvokeError"

definition invoke :: "(((_) object_ptr ⇒ bool) × ((_) object_ptr ⇒ 'args
                    ⇒ (_, 'result) dom_prog)) list
                    ⇒ (_) object_ptr ⇒ 'args ⇒ (_, 'result) dom_prog"
  where
    "invoke xs ptr args = do { check_in_heap ptr; invoke_rec xs ptr args}"

lemma invoke_split: "P (invoke ((Pred, f) # xs) ptr args) =
    ((¬(Pred ptr) ⟶ P (invoke xs ptr args))
  ∧ (Pred ptr ⟶ P (do {check_in_heap ptr; f ptr args})))"
  ⟨proof⟩

lemma invoke_split_asm: "P (invoke ((Pred, f) # xs) ptr args) =
    (¬((¬(Pred ptr) ∧ (¬ P (invoke xs ptr args)))
    ∨ (Pred ptr ∧ (¬ P (do {check_in_heap ptr; f ptr args})))))"
  ⟨proof⟩
lemmas invoke_splits = invoke_split invoke_split_asm

lemma invoke_ptr_in_heap: "h ⊢ ok (invoke xs ptr args) ⟹ ptr |∈| object_ptr_kinds h"
  ⟨proof⟩

lemma invoke_pure [simp]: "pure (invoke [] ptr args) h"
  ⟨proof⟩

lemma invoke_is_OK [simp]:
  "ptr |∈| object_ptr_kinds h ⟹ Pred ptr
    ⟹ h ⊢ ok (invoke ((Pred, f) # xs) ptr args) = h ⊢ ok (f ptr args)"
  ⟨proof⟩
lemma invoke_returns_result [simp]:
  "ptr |∈| object_ptr_kinds h ⟹ Pred ptr
   ⟹ h ⊢ (invoke ((Pred, f) # xs) ptr args) →r x = h ⊢ f ptr args →r x"
  ⟨proof⟩
lemma invoke_returns_heap [simp]:
  "ptr |∈| object_ptr_kinds h ⟹ Pred ptr
   ⟹ h ⊢ (invoke ((Pred, f) # xs) ptr args) →h h' = h ⊢ f ptr args →h h'"
  ⟨proof⟩

lemma invoke_not [simp]: "¬Pred ptr ⟹ invoke ((Pred, f) # xs) ptr args = invoke xs ptr args"
  ⟨proof⟩

lemma invoke_empty [simp]: "¬h ⊢ ok (invoke [] ptr args)"
  ⟨proof⟩

lemma invoke_empty_reads [simp]: "∀P ∈ S. reflp P ∧ transp P ⟹ reads S (invoke [] ptr args) h h'"
  ⟨proof⟩
```

5.2.2 Modified Heaps

```
lemma get_object_ptr_simp [simp]:
  "get_Object object_ptr (put_Object ptr obj h) = (if ptr = object_ptr then Some obj else get object_ptr h)"
  ⟨proof⟩

lemma object_ptr_kinds_simp [simp]: "object_ptr_kinds (put_Object ptr obj h) = object_ptr_kinds h |∪| {|ptr|}"
  ⟨proof⟩

lemma type_wf_put_I:
  assumes "type_wf h"
  shows "type_wf (put_Object ptr obj h)"
  ⟨proof⟩

lemma type_wf_put_ptr_not_in_heap_E:
  assumes "type_wf (put_Object ptr obj h)"
  assumes "ptr |∉| object_ptr_kinds h"
  shows "type_wf h"
  ⟨proof⟩

lemma type_wf_put_ptr_in_heap_E:
  assumes "type_wf (put_Object ptr obj h)"
  assumes "ptr |∈| object_ptr_kinds h"
  shows "type_wf h"
  ⟨proof⟩
```

5.2.3 Preserving Types

```
lemma type_wf_preserved: "type_wf h = type_wf h'"
  ⟨proof⟩


lemma object_ptr_kinds_preserved_small:
  assumes "⋀object_ptr. preserved (get_M_Object object_ptr RObject.nothing) h h'"
  shows "object_ptr_kinds h = object_ptr_kinds h'"
  ⟨proof⟩

lemma object_ptr_kinds_preserved:
  assumes "writes SW setter h h'"
  assumes "h ⊢ setter →_h h'"
  assumes "⋀h h' w object_ptr. w ∈ SW ⟹ h ⊢ w →_h h'
          ⟹ preserved (get_M_Object object_ptr RObject.nothing) h h'"
  shows "object_ptr_kinds h = object_ptr_kinds h'"
⟨proof⟩


lemma reads_writes_preserved2:
  assumes "writes SW setter h h'"
  assumes "h ⊢ setter →_h h'"
  assumes "⋀h h' x. ∀w ∈ SW. h ⊢ w →_h h' ⟶ preserved (get_M_Object ptr getter) h h'"
  shows "preserved (get_M ptr getter) h h'"
  ⟨proof⟩
end
```

5.3 Node (NodeMonad)

In this theory, we introduce the monadic method setup for the Node class.

```
theory NodeMonad
  imports
    ObjectMonad
    "../classes/NodeClass"
begin

type_synonym ('object_ptr, 'node_ptr, 'Object, 'Node, 'result) dom_prog
  = "((_) heap, exception, 'result) prog"
register_default_tvars "('object_ptr, 'node_ptr, 'Object, 'Node, 'result) dom_prog"


global_interpretation l_ptr_kinds_M node_ptr_kinds defines node_ptr_kinds_M = a_ptr_kinds_M ⟨proof⟩
lemmas node_ptr_kinds_M_defs = a_ptr_kinds_M_def

lemma node_ptr_kinds_M_eq:
  assumes "|h ⊢ object_ptr_kinds_M|_r = |h' ⊢ object_ptr_kinds_M|_r"
  shows "|h ⊢ node_ptr_kinds_M|_r = |h' ⊢ node_ptr_kinds_M|_r"
  ⟨proof⟩


global_interpretation l_dummy defines get_M_Node = "l_get_M.a_get_M get_Node" ⟨proof⟩
lemma get_M_is_l_get_M: "l_get_M get_Node type_wf node_ptr_kinds"
  ⟨proof⟩
lemmas get_M_defs = get_M_Node_def[unfolded l_get_M.a_get_M_def[OF get_M_is_l_get_M]]

adhoc_overloading get_M ⇌ get_M_Node

locale l_get_M_Node_lemmas = l_type_wf_Node
begin
sublocale l_get_M_Object_lemmas ⟨proof⟩
```

interpretation *l_get_M* get_{Node} *type_wf node_ptr_kinds*
  ⟨*proof*⟩
lemmas $get\_M_{Node}\_ok$ = *get_M_ok[folded* $get\_M_{Node}\_def$*]*
end

global_interpretation $l\_get\_M_{Node}\_lemmas$ *type_wf* ⟨*proof*⟩

lemma *node_ptr_kinds_M_reads:*
  "reads (⋃object_ptr. {preserved (get_M_{Object} *object_ptr RObject.nothing)}) node_ptr_kinds_M h h'"*
  ⟨*proof*⟩

global_interpretation *l_put_M type_wf node_ptr_kinds* get_{Node} put_{Node}
  rewrites *"a_get_M =* get_M_{Node}*"*
  defines put_M_{Node} *= a_put_M*
   ⟨*proof*⟩

lemmas *put_M_defs = a_put_M_def*
adhoc_overloading *put_M ⇌* put_M_{Node}

locale $l\_put\_M_{Node}\_lemmas$ *=* $l\_type\_wf_{Node}$
begin
sublocale $l\_put\_M_{Object}\_lemmas$ ⟨*proof*⟩

interpretation *l_put_M type_wf node_ptr_kinds* get_{Node} put_{Node}
  ⟨*proof*⟩
lemmas $put\_M_{Node}\_ok$ = *put_M_ok[folded* $put\_M_{Node}\_def$*]*
end

global_interpretation $l\_put\_M_{Node}\_lemmas$ *type_wf* ⟨*proof*⟩

lemma *get_M_Object_preserved1 [simp]:*
  "(⋀x. getter (cast (setter (λ_. v) x)) = getter (cast x)) ⟹ h ⊢ put_M_{Node} *node_ptr setter v* $\rightarrow_h$ *h'*

    ⟹ preserved (get_M_{Object} *object_ptr getter) h h'"*
  ⟨*proof*⟩

lemma *get_M_Object_preserved2 [simp]:*
  "cast node_ptr ≠ object_ptr ⟹ h ⊢ put_M_{Node} *node_ptr setter v* $\rightarrow_h$ *h'*
    ⟹ preserved (get_M_{Object} *object_ptr getter) h h'"*
  ⟨*proof*⟩
lemma *get_M_Object_preserved3 [simp]:*
  "h ⊢ put_M_{Node} *node_ptr setter v* $\rightarrow_h$ *h' ⟹ (⋀x. getter (cast (setter (λ_. v) x)) = getter (cast x))*

    ⟹ preserved (get_M_{Object} *object_ptr getter) h h'"*
  ⟨*proof*⟩

lemma *get_M_Object_preserved4 [simp]:*
  "cast node_ptr ≠ object_ptr ⟹ h ⊢ put_M_{Object} *object_ptr setter v* $\rightarrow_h$ *h'*
     ⟹ preserved (get_M_{Node} *node_ptr getter) h h'"*
  ⟨*proof*⟩

5.3.1 Modified Heaps

lemma *get_node_ptr_simp [simp]:*
  "get_{Node} *node_ptr (*put_{Object} *ptr obj h) = (if ptr = cast node_ptr then cast obj else get node_ptr h)"*
  ⟨*proof*⟩

lemma *node_ptr_kinds_simp [simp]:*
  "node_ptr_kinds (put_{Object} *ptr obj h)*
                = node_ptr_kinds h |∪| (if is_node_ptr_kind ptr then {|the (cast ptr)|} else {||})"
  ⟨*proof*⟩

```
lemma type_wf_put_I:
  assumes "type_wf h"
  assumes "ObjectClass.type_wf (put_Object ptr obj h)"
  assumes "is_node_ptr_kind ptr ⟹ is_node_kind obj"
  shows "type_wf (put_Object ptr obj h)"
  ⟨proof⟩

lemma type_wf_put_ptr_not_in_heap_E:
  assumes "type_wf (put_Object ptr obj h)"
  assumes "ptr |∉| object_ptr_kinds h"
  shows "type_wf h"
  ⟨proof⟩

lemma type_wf_put_ptr_in_heap_E:
  assumes "type_wf (put_Object ptr obj h)"
  assumes "ptr |∈| object_ptr_kinds h"
  assumes "ObjectClass.type_wf h"
  assumes "is_node_ptr_kind ptr ⟹ is_node_kind (the (get ptr h))"
  shows "type_wf h"
  ⟨proof⟩
```

5.3.2 Preserving Types

```
lemma node_ptr_kinds_small:
  assumes "⋀object_ptr. preserved (get_M_Object object_ptr RObject.nothing) h h'"
  shows "node_ptr_kinds h = node_ptr_kinds h'"
  ⟨proof⟩

lemma node_ptr_kinds_preserved:
  assumes "writes SW setter h h'"
  assumes "h ⊢ setter →_h h'"
  assumes "⋀h h'. ∀w ∈ SW. h ⊢ w →_h h'
          ⟶ (∀object_ptr. preserved (get_M_Object object_ptr RObject.nothing) h h')"
  shows "node_ptr_kinds h = node_ptr_kinds h'"
  ⟨proof⟩


lemma type_wf_preserved_small:
  assumes "⋀object_ptr. preserved (get_M_Object object_ptr RObject.nothing) h h'"
  assumes "⋀node_ptr. preserved (get_M_Node node_ptr RNode.nothing) h h'"
  shows "type_wf h = type_wf h'"
  ⟨proof⟩

lemma type_wf_preserved:
  assumes "writes SW setter h h'"
  assumes "h ⊢ setter →_h h'"
  assumes "⋀h h' w. w ∈ SW ⟹ h ⊢ w →_h h'
          ⟹ ∀object_ptr. preserved (get_M_Object object_ptr RObject.nothing) h h'"
  assumes "⋀h h' w. w ∈ SW ⟹ h ⊢ w →_h h'
          ⟹ ∀node_ptr. preserved (get_M_Node node_ptr RNode.nothing) h h'"
  shows "type_wf h = type_wf h'"
⟨proof⟩
end
```

5.4 Element (ElementMonad)

In this theory, we introduce the monadic method setup for the Element class.

```
theory ElementMonad
  imports
    NodeMonad
    "ElementClass"
begin
```

```
type_synonym ('object_ptr, 'node_ptr, 'element_ptr, 'character_data_ptr, 'document_ptr,
    'shadow_root_ptr, 'Object, 'Node, 'Element,'result) dom_prog
  = "((_) heap, exception, 'result) prog"
register_default_tvars "('object_ptr, 'node_ptr, 'element_ptr, 'character_data_ptr,
                        'document_ptr, 'shadow_root_ptr, 'Object, 'Node, 'Element,'result) dom_prog"


global_interpretation l_ptr_kinds_M element_ptr_kinds defines element_ptr_kinds_M = a_ptr_kinds_M ⟨proof⟩
lemmas element_ptr_kinds_M_defs = a_ptr_kinds_M_def


lemma element_ptr_kinds_M_eq:
  assumes "|h ⊢ node_ptr_kinds_M|$_r$ = |h' ⊢ node_ptr_kinds_M|$_r$"
  shows "|h ⊢ element_ptr_kinds_M|$_r$ = |h' ⊢ element_ptr_kinds_M|$_r$"
  ⟨proof⟩

lemma element_ptr_kinds_M_reads:
  "reads (⋃element_ptr. {preserved (get_M$_{Object}$ element_ptr RObject.nothing)}) element_ptr_kinds_M h h'"
  ⟨proof⟩

global_interpretation l_dummy defines get_M$_{Element}$ = "l_get_M.a_get_M get$_{Element}$" ⟨proof⟩
lemma get_M_is_l_get_M: "l_get_M get$_{Element}$ type_wf element_ptr_kinds"
  ⟨proof⟩
lemmas get_M_defs = get_M$_{Element}$_def[unfolded l_get_M.a_get_M_def[OF get_M_is_l_get_M]]

adhoc_overloading get_M ⇌ get_M$_{Element}$

locale l_get_M$_{Element}$_lemmas = l_type_wf$_{Element}$
begin
sublocale l_get_M$_{Node}$_lemmas ⟨proof⟩

interpretation l_get_M get$_{Element}$ type_wf element_ptr_kinds
  ⟨proof⟩
lemmas get_M$_{Element}$_ok = get_M_ok[folded get_M$_{Element}$_def]
lemmas get_M$_{Element}$_ptr_in_heap = get_M_ptr_in_heap[folded get_M$_{Element}$_def]
end

global_interpretation l_get_M$_{Element}$_lemmas type_wf ⟨proof⟩


global_interpretation l_put_M type_wf element_ptr_kinds get$_{Element}$ put$_{Element}$
  rewrites "a_get_M = get_M$_{Element}$"
  defines put_M$_{Element}$ = a_put_M
   ⟨proof⟩

lemmas put_M_defs = a_put_M_def
adhoc_overloading put_M ⇌ put_M$_{Element}$


locale l_put_M$_{Element}$_lemmas = l_type_wf$_{Element}$
begin
sublocale l_put_M$_{Node}$_lemmas ⟨proof⟩

interpretation l_put_M type_wf element_ptr_kinds get$_{Element}$ put$_{Element}$
  ⟨proof⟩

lemmas put_M$_{Element}$_ok = put_M_ok[folded put_M$_{Element}$_def]
end

global_interpretation l_put_M$_{Element}$_lemmas type_wf ⟨proof⟩
```

lemma *element_put_get [simp]:*
  "h ⊢ put_M$_{Element}$ element_ptr setter v →$_h$ h' ⟹ (⋀x. getter (setter (λ_. v) x) = v)
  ⟹ h' ⊢ get_M$_{Element}$ element_ptr getter →$_r$ v"
  ⟨proof⟩
lemma *get_M_Element_preserved1 [simp]:*
  "element_ptr ≠ element_ptr' ⟹ h ⊢ put_M$_{Element}$ element_ptr setter v →$_h$ h'
    ⟹ preserved (get_M$_{Element}$ element_ptr' getter) h h'"
  ⟨proof⟩
lemma *element_put_get_preserved [simp]:*
  "(⋀x. getter (setter (λ_. v) x) = getter x) ⟹ h ⊢ put_M$_{Element}$ element_ptr setter v →$_h$ h'
    ⟹ preserved (get_M$_{Element}$ element_ptr' getter) h h'"
  ⟨proof⟩
lemma *get_M_Element_preserved3 [simp]:*
  "(⋀x. getter (cast (setter (λ_. v) x)) = getter (cast x))
    ⟹ h ⊢ put_M$_{Element}$ element_ptr setter v →$_h$ h' ⟹ preserved (get_M$_{Object}$ object_ptr getter) h h'"
  ⟨proof⟩
lemma *get_M_Element_preserved4 [simp]:*
  "(⋀x. getter (cast (setter (λ_. v) x)) = getter (cast x))
    ⟹ h ⊢ put_M$_{Element}$ element_ptr setter v →$_h$ h' ⟹ preserved (get_M$_{Node}$ node_ptr getter) h h'"
  ⟨proof⟩

lemma *get_M_Element_preserved5 [simp]:*
  "cast element_ptr ≠ node_ptr ⟹ h ⊢ put_M$_{Element}$ element_ptr setter v →$_h$ h'
  ⟹ preserved (get_M$_{Node}$ node_ptr getter) h h'"
  ⟨proof⟩
lemma *get_M_Element_preserved6 [simp]:*
  "h ⊢ put_M$_{Element}$ element_ptr setter v →$_h$ h'
    ⟹ (⋀x. getter (cast (setter (λ_. v) x)) = getter (cast x))
    ⟹ preserved (get_M$_{Node}$ node_ptr getter) h h'"
  ⟨proof⟩

lemma *get_M_Element_preserved7 [simp]:*
  "cast element_ptr ≠ node_ptr ⟹ h ⊢ put_M$_{Node}$ node_ptr setter v →$_h$ h'
    ⟹ preserved (get_M$_{Element}$ element_ptr getter) h h'"
  ⟨proof⟩

lemma *get_M_Element_preserved8 [simp]:*
  "cast element_ptr ≠ object_ptr ⟹ h ⊢ put_M$_{Element}$ element_ptr setter v →$_h$ h'
    ⟹ preserved (get_M$_{Object}$ object_ptr getter) h h'"
  ⟨proof⟩
lemma *get_M_Element_preserved9 [simp]:*
  "h ⊢ put_M$_{Element}$ element_ptr setter v →$_h$ h'
    ⟹ (⋀x. getter (cast (setter (λ_. v) x)) = getter (cast x))
   ⟹ preserved (get_M$_{Object}$ object_ptr getter) h h'"
  ⟨proof⟩

lemma *get_M_Element_preserved10 [simp]:*
  "cast element_ptr ≠ object_ptr ⟹ h ⊢ put_M$_{Object}$ object_ptr setter v →$_h$ h'
    ⟹ preserved (get_M$_{Element}$ element_ptr getter) h h'"
  ⟨proof⟩

5.4.1 Creating Elements

definition *new_element :: "(_, (_) element_ptr) dom_prog"*
  where
    "new_element = do {
      h ← get_heap;
      (new_ptr, h') ← return (new$_{Element}$ h);
      return_heap h';
      return new_ptr
    }"

lemma *new_element_ok [simp]:*

"h ⊢ ok new_element"
⟨proof⟩

lemma new_element_ptr_in_heap:
assumes "h ⊢ new_element $\rightarrow_h$ h'"
and "h ⊢ new_element $\rightarrow_r$ new_element_ptr"
shows "new_element_ptr |∈| element_ptr_kinds h'"
⟨proof⟩

lemma new_element_ptr_not_in_heap:
assumes "h ⊢ new_element $\rightarrow_h$ h'"
and "h ⊢ new_element $\rightarrow_r$ new_element_ptr"
shows "new_element_ptr |∉| element_ptr_kinds h"
⟨proof⟩

lemma new_element_new_ptr:
assumes "h ⊢ new_element $\rightarrow_h$ h'"
and "h ⊢ new_element $\rightarrow_r$ new_element_ptr"
shows "object_ptr_kinds h' = object_ptr_kinds h |∪| {|cast new_element_ptr|}"
⟨proof⟩

lemma new_element_is_element_ptr:
assumes "h ⊢ new_element $\rightarrow_r$ new_element_ptr"
shows "is_element_ptr new_element_ptr"
⟨proof⟩

lemma new_element_child_nodes:
assumes "h ⊢ new_element $\rightarrow_h$ h'"
assumes "h ⊢ new_element $\rightarrow_r$ new_element_ptr"
shows "h' ⊢ get_M new_element_ptr child_nodes $\rightarrow_r$ []"
⟨proof⟩

lemma new_element_tag_name:
assumes "h ⊢ new_element $\rightarrow_h$ h'"
assumes "h ⊢ new_element $\rightarrow_r$ new_element_ptr"
shows "h' ⊢ get_M new_element_ptr tag_name $\rightarrow_r$ ''''"
⟨proof⟩

lemma new_element_attrs:
assumes "h ⊢ new_element $\rightarrow_h$ h'"
assumes "h ⊢ new_element $\rightarrow_r$ new_element_ptr"
shows "h' ⊢ get_M new_element_ptr attrs $\rightarrow_r$ fmempty"
⟨proof⟩

lemma new_element_shadow_root_opt:
assumes "h ⊢ new_element $\rightarrow_h$ h'"
assumes "h ⊢ new_element $\rightarrow_r$ new_element_ptr"
shows "h' ⊢ get_M new_element_ptr shadow_root_opt $\rightarrow_r$ None"
⟨proof⟩

lemma new_element_get_M_{Object}:
"h ⊢ new_element $\rightarrow_h$ h' ⟹ h ⊢ new_element $\rightarrow_r$ new_element_ptr ⟹ ptr ≠ cast new_element_ptr
⟹ preserved (get_M_{Object} ptr getter) h h'"
⟨proof⟩
lemma new_element_get_M_{Node}:
"h ⊢ new_element $\rightarrow_h$ h' ⟹ h ⊢ new_element $\rightarrow_r$ new_element_ptr ⟹ ptr ≠ cast new_element_ptr
⟹ preserved (get_M_{Node} ptr getter) h h'"
⟨proof⟩
lemma new_element_get_$M_{Element}$:
"h ⊢ new_element $\rightarrow_h$ h' ⟹ h ⊢ new_element $\rightarrow_r$ new_element_ptr ⟹ ptr ≠ new_element_ptr
⟹ preserved (get_$M_{Element}$ ptr getter) h h'"
⟨proof⟩

5.4.2 Modified Heaps

lemma *get_Element_ptr_simp [simp]:*
  "get$_{Element}$ element_ptr (put$_{Object}$ ptr obj h)
    = (if ptr = cast element_ptr then cast obj else get element_ptr h)"
  ⟨*proof*⟩

lemma *element_ptr_kinds_simp [simp]:*
  "element_ptr_kinds (put$_{Object}$ ptr obj h)
    = element_ptr_kinds h |∪| (if is_element_ptr_kind ptr then {|the (cast ptr)|} else {||})"
  ⟨*proof*⟩

lemma *type_wf_put_I:*
  assumes *"type_wf h"*
  assumes *"NodeClass.type_wf (put$_{Object}$ ptr obj h)"*
  assumes *"is_element_ptr_kind ptr ⟹ is_element_kind obj"*
  shows *"type_wf (put$_{Object}$ ptr obj h)"*
  ⟨*proof*⟩

lemma *type_wf_put_ptr_not_in_heap_E:*
  assumes *"type_wf (put$_{Object}$ ptr obj h)"*
  assumes *"ptr |∉| object_ptr_kinds h"*
  shows *"type_wf h"*
  ⟨*proof*⟩

lemma *type_wf_put_ptr_in_heap_E:*
  assumes *"type_wf (put$_{Object}$ ptr obj h)"*
  assumes *"ptr |∈| object_ptr_kinds h"*
  assumes *"NodeClass.type_wf h"*
  assumes *"is_element_ptr_kind ptr ⟹ is_element_kind (the (get ptr h))"*
  shows *"type_wf h"*
  ⟨*proof*⟩

5.4.3 Preserving Types

lemma *new_element_type_wf_preserved [simp]: "h ⊢ new_element →$_h$ h' ⟹ type_wf h = type_wf h'"*
  ⟨*proof*⟩

locale *l_new_element = l_type_wf +*
  assumes *new_element_types_preserved: "h ⊢ new_element →$_h$ h' ⟹ type_wf h = type_wf h'"*

lemma *new_element_is_l_new_element: "l_new_element type_wf"*
  ⟨*proof*⟩

lemma *put_M$_{Element}$_tag_name_type_wf_preserved [simp]:*
  "h ⊢ put_M element_ptr tag_name_update v →$_h$ h' ⟹ type_wf h = type_wf h'"
  ⟨*proof*⟩

lemma *put_M$_{Element}$_child_nodes_type_wf_preserved [simp]:*
  "h ⊢ put_M element_ptr child_nodes_update v →$_h$ h' ⟹ type_wf h = type_wf h'"
  ⟨*proof*⟩

lemma *put_M$_{Element}$_attrs_type_wf_preserved [simp]:*
  "h ⊢ put_M element_ptr attrs_update v →$_h$ h' ⟹ type_wf h = type_wf h'"
  ⟨*proof*⟩

lemma *put_M$_{Element}$_shadow_root_opt_type_wf_preserved [simp]:*
  "h ⊢ put_M element_ptr shadow_root_opt_update v →$_h$ h' ⟹ type_wf h = type_wf h'"
  ⟨*proof*⟩

lemma *put_M_pointers_preserved:*
  assumes *"h ⊢ put_M$_{Element}$ element_ptr setter v →$_h$ h'"*

```
  shows "object_ptr_kinds h = object_ptr_kinds h'"
  ⟨proof⟩

lemma element_ptr_kinds_preserved:
  assumes "writes SW setter h h'"
  assumes "h ⊢ setter →h h'"
  assumes "⋀h h'. ∀w ∈ SW. h ⊢ w →h h'
                 ⟶ (∀object_ptr. preserved (get_M_Object object_ptr RObject.nothing) h h')"
  shows "element_ptr_kinds h = element_ptr_kinds h'"
  ⟨proof⟩


lemma element_ptr_kinds_small:
  assumes "⋀object_ptr. preserved (get_M_Object object_ptr RObject.nothing) h h'"
  shows "element_ptr_kinds h = element_ptr_kinds h'"
  ⟨proof⟩

lemma type_wf_preserved_small:
  assumes "⋀object_ptr. preserved (get_M_Object object_ptr RObject.nothing) h h'"
  assumes "⋀node_ptr. preserved (get_M_Node node_ptr RNode.nothing) h h'"
  assumes "⋀element_ptr. preserved (get_M_Element element_ptr RElement.nothing) h h'"
  shows "type_wf h = type_wf h'"
  ⟨proof⟩

lemma type_wf_preserved:
  assumes "writes SW setter h h'"
  assumes "h ⊢ setter →h h'"
  assumes "⋀h h' w. w ∈ SW ⟹ h ⊢ w →h h'
          ⟹ ∀object_ptr. preserved (get_M_Object object_ptr RObject.nothing) h h'"
  assumes "⋀h h' w. w ∈ SW ⟹ h ⊢ w →h h'
          ⟹ ∀node_ptr. preserved (get_M_Node node_ptr RNode.nothing) h h'"
  assumes "⋀h h' w. w ∈ SW ⟹ h ⊢ w →h h'
          ⟹ ∀element_ptr. preserved (get_M_Element element_ptr RElement.nothing) h h'"
  shows "type_wf h = type_wf h'"
⟨proof⟩

lemma type_wf_drop: "type_wf h ⟹ type_wf (Heap (fmdrop ptr (the_heap h)))"
  ⟨proof⟩

end
```

5.5 CharacterData (CharacterDataMonad)

In this theory, we introduce the monadic method setup for the CharacterData class.

```
theory CharacterDataMonad
  imports
    ElementMonad
    "../classes/CharacterDataClass"
begin

type_synonym ('object_ptr, 'node_ptr, 'element_ptr, 'character_data_ptr, 'document_ptr,
    'shadow_root_ptr, 'Object, 'Node, 'Element, 'CharacterData, 'result) dom_prog
  = "((_) heap, exception, 'result) prog"
register_default_tvars
  "('object_ptr, 'node_ptr, 'element_ptr, 'character_data_ptr, 'document_ptr, 'shadow_root_ptr,
                             'Object, 'Node, 'Element, 'CharacterData, 'result) dom_prog"


global_interpretation l_ptr_kinds_M character_data_ptr_kinds
  defines character_data_ptr_kinds_M = a_ptr_kinds_M ⟨proof⟩
lemmas character_data_ptr_kinds_M_defs = a_ptr_kinds_M_def
```

lemma *character_data_ptr_kinds_M_eq:*
  assumes *"$|h \vdash$ node_ptr_kinds_M$|_r$ = $|h' \vdash$ node_ptr_kinds_M$|_r$"*
  shows *"$|h \vdash$ character_data_ptr_kinds_M$|_r$ = $|h' \vdash$ character_data_ptr_kinds_M$|_r$"*
  ⟨*proof*⟩

lemma *character_data_ptr_kinds_M_reads:*
  "reads ($\bigcup$node_ptr. {preserved (get_M$_{Object}$ node_ptr RObject.nothing)}) character_data_ptr_kinds_M h h'"
  ⟨*proof*⟩

global_interpretation *l_dummy* **defines** *get_M$_{CharacterData}$ = "l_get_M.a_get_M get$_{CharacterData}$"* ⟨*proof*⟩
lemma *get_M_is_l_get_M: "l_get_M get$_{CharacterData}$ type_wf character_data_ptr_kinds"*
  ⟨*proof*⟩
lemmas *get_M_defs = get_M$_{CharacterData}$_def[unfolded l_get_M.a_get_M_def[OF get_M_is_l_get_M]]*

adhoc_overloading *get_M $\rightleftharpoons$ get_M$_{CharacterData}$*

locale *l_get_M$_{CharacterData}$_lemmas = l_type_wf$_{CharacterData}$*
begin
sublocale *l_get_M$_{Element}$_lemmas* ⟨*proof*⟩

interpretation *l_get_M get$_{CharacterData}$ type_wf character_data_ptr_kinds*
  ⟨*proof*⟩
lemmas *get_M$_{CharacterData}$_ok = get_M_ok[folded get_M$_{CharacterData}$_def]*
end

global_interpretation *l_get_M$_{CharacterData}$_lemmas type_wf* ⟨*proof*⟩

global_interpretation *l_put_M type_wf character_data_ptr_kinds get$_{CharacterData}$ put$_{CharacterData}$*
  rewrites *"a_get_M = get_M$_{CharacterData}$"* **defines** *put_M$_{CharacterData}$ = a_put_M*
   ⟨*proof*⟩

lemmas *put_M_defs = a_put_M_def*
adhoc_overloading *put_M $\rightleftharpoons$ put_M$_{CharacterData}$*

locale *l_put_M$_{CharacterData}$_lemmas = l_type_wf$_{CharacterData}$*
begin
sublocale *l_put_M$_{Element}$_lemmas* ⟨*proof*⟩

interpretation *l_put_M type_wf character_data_ptr_kinds get$_{CharacterData}$ put$_{CharacterData}$*
  ⟨*proof*⟩
lemmas *put_M$_{CharacterData}$_ok = put_M_ok[folded put_M$_{CharacterData}$_def]*
end

global_interpretation *l_put_M$_{CharacterData}$_lemmas type_wf* ⟨*proof*⟩

lemma *CharacterData_simp1 [simp]:*
  "($\bigwedge$x. getter (setter (λ_. v) x) = v) $\Longrightarrow$ h $\vdash$ put_M$_{CharacterData}$ character_data_ptr setter v $\rightarrow_h$ h'
   $\Longrightarrow$ h' $\vdash$ get_M$_{CharacterData}$ character_data_ptr getter $\rightarrow_r$ v"
  ⟨*proof*⟩
lemma *CharacterData_simp2 [simp]:*
  "character_data_ptr $\neq$ character_data_ptr'
   $\Longrightarrow$ h $\vdash$ put_M$_{CharacterData}$ character_data_ptr setter v $\rightarrow_h$ h'
  $\Longrightarrow$ preserved (get_M$_{CharacterData}$ character_data_ptr' getter) h h'"
  ⟨*proof*⟩
lemma *CharacterData_simp3 [simp]: "*
  ($\bigwedge$x. getter (setter (λ_. v) x) = getter x)
  $\Longrightarrow$ h $\vdash$ put_M$_{CharacterData}$ character_data_ptr setter v $\rightarrow_h$ h'
  $\Longrightarrow$ preserved (get_M$_{CharacterData}$ character_data_ptr' getter) h h'"
  ⟨*proof*⟩

lemma CharacterData_simp4 [simp]:
  "h ⊢ put_$M_{CharacterData}$ character_data_ptr setter v $\rightarrow_h$ h'
  ⟹ preserved (get_$M_{Element}$ element_ptr getter) h h'"
  ⟨*proof*⟩
lemma CharacterData_simp5 [simp]:
  "h ⊢ put_$M_{Element}$ element_ptr setter v $\rightarrow_h$ h'
  ⟹ preserved (get_$M_{CharacterData}$ character_data_ptr getter) h h'"
  ⟨*proof*⟩
lemma CharacterData_simp6 [simp]:
  "(⋀x. getter (cast (setter (λ_. v) x)) = getter (cast x))
   ⟹ h ⊢ put_$M_{CharacterData}$ character_data_ptr setter v $\rightarrow_h$ h'
  ⟹ preserved (get_M_{Object} object_ptr getter) h h'"
  ⟨*proof*⟩
lemma CharacterData_simp7 [simp]:
  "(⋀x. getter (cast (setter (λ_. v) x)) = getter (cast x))
  ⟹ h ⊢ put_$M_{CharacterData}$ character_data_ptr setter v $\rightarrow_h$ h'
  ⟹ preserved (get_M_{Node} node_ptr getter) h h'"
  ⟨*proof*⟩

lemma CharacterData_simp8 [simp]:
  "cast character_data_ptr ≠ node_ptr
   ⟹ h ⊢ put_$M_{CharacterData}$ character_data_ptr setter v $\rightarrow_h$ h'
   ⟹ preserved (get_M_{Node} node_ptr getter) h h'"
  ⟨*proof*⟩
lemma CharacterData_simp9 [simp]:
  "h ⊢ put_$M_{CharacterData}$ character_data_ptr setter v $\rightarrow_h$ h'
   ⟹ (⋀x. getter (cast (setter (λ_. v) x)) = getter (cast x))
   ⟹ preserved (get_M_{Node} node_ptr getter) h h'"
  ⟨*proof*⟩
lemma CharacterData_simp10 [simp]:
  "cast character_data_ptr ≠ node_ptr
   ⟹ h ⊢ put_M_{Node} node_ptr setter v $\rightarrow_h$ h'
   ⟹ preserved (get_$M_{CharacterData}$ character_data_ptr getter) h h'"
  ⟨*proof*⟩

lemma CharacterData_simp11 [simp]:
  "cast character_data_ptr ≠ object_ptr
  ⟹ h ⊢ put_$M_{CharacterData}$ character_data_ptr setter v $\rightarrow_h$ h'
  ⟹ preserved (get_M_{Object} object_ptr getter) h h'"
  ⟨*proof*⟩

lemma CharacterData_simp12 [simp]:
  "h ⊢ put_$M_{CharacterData}$ character_data_ptr setter v $\rightarrow_h$ h'
   ⟹ (⋀x. getter (cast (setter (λ_. v) x)) = getter (cast x))
   ⟹ preserved (get_M_{Object} object_ptr getter) h h'"
  ⟨*proof*⟩

lemma CharacterData_simp13 [simp]:
  "cast character_data_ptr ≠ object_ptr ⟹ h ⊢ put_M_{Object} object_ptr setter v $\rightarrow_h$ h'
    ⟹ preserved (get_$M_{CharacterData}$ character_data_ptr getter) h h'"
  ⟨*proof*⟩

lemma new_element_get_$M_{CharacterData}$:
  "h ⊢ new_element $\rightarrow_h$ h' ⟹ preserved (get_$M_{CharacterData}$ ptr getter) h h'"
  ⟨*proof*⟩

5.5.1 Creating CharacterData

definition new_character_data :: "(_, (_) character_data_ptr) dom_prog"
  where
    "new_character_data = do {
      h ← get_heap;
      (new_ptr, h') ← return (new$_{CharacterData}$ h);

```
    return_heap h';
    return new_ptr
  }"
```

lemma *new_character_data_ok [simp]:*
"h ⊢ ok new_character_data"
⟨*proof*⟩

lemma *new_character_data_ptr_in_heap:*
assumes *"h ⊢ new_character_data* $\rightarrow_h$ *h'"*
and *"h ⊢ new_character_data* $\rightarrow_r$ *new_character_data_ptr"*
shows *"new_character_data_ptr |∈| character_data_ptr_kinds h'"*
⟨*proof*⟩

lemma *new_character_data_ptr_not_in_heap:*
assumes *"h ⊢ new_character_data* $\rightarrow_h$ *h'"*
and *"h ⊢ new_character_data* $\rightarrow_r$ *new_character_data_ptr"*
shows *"new_character_data_ptr |∉| character_data_ptr_kinds h"*
⟨*proof*⟩

lemma *new_character_data_new_ptr:*
assumes *"h ⊢ new_character_data* $\rightarrow_h$ *h'"*
and *"h ⊢ new_character_data* $\rightarrow_r$ *new_character_data_ptr"*
shows *"object_ptr_kinds h' = object_ptr_kinds h |∪| {|cast new_character_data_ptr|}"*
⟨*proof*⟩

lemma *new_character_data_is_character_data_ptr:*
assumes *"h ⊢ new_character_data* $\rightarrow_r$ *new_character_data_ptr"*
shows *"is_character_data_ptr new_character_data_ptr"*
⟨*proof*⟩

lemma *new_character_data_child_nodes:*
assumes *"h ⊢ new_character_data* $\rightarrow_h$ *h'"*
assumes *"h ⊢ new_character_data* $\rightarrow_r$ *new_character_data_ptr"*
shows *"h' ⊢ get_M new_character_data_ptr val* $\rightarrow_r$ *''''"*
⟨*proof*⟩

lemma *new_character_data_get_M*$_{Object}$*:*
"h ⊢ new_character_data $\rightarrow_h$ *h'* ⟹ *h ⊢ new_character_data* $\rightarrow_r$ *new_character_data_ptr*
⟹ *ptr ≠ cast new_character_data_ptr* ⟹ *preserved (get_M*$_{Object}$ *ptr getter) h h'"*
⟨*proof*⟩
lemma *new_character_data_get_M*$_{Node}$*:*
"h ⊢ new_character_data $\rightarrow_h$ *h'* ⟹ *h ⊢ new_character_data* $\rightarrow_r$ *new_character_data_ptr*
⟹ *ptr ≠ cast new_character_data_ptr* ⟹ *preserved (get_M*$_{Node}$ *ptr getter) h h'"*
⟨*proof*⟩
lemma *new_character_data_get_M*$_{Element}$*:*
"h ⊢ new_character_data $\rightarrow_h$ *h'* ⟹ *h ⊢ new_character_data* $\rightarrow_r$ *new_character_data_ptr*
⟹ *preserved (get_M*$_{Element}$ *ptr getter) h h'"*
⟨*proof*⟩
lemma *new_character_data_get_M*$_{CharacterData}$*:*
"h ⊢ new_character_data $\rightarrow_h$ *h'* ⟹ *h ⊢ new_character_data* $\rightarrow_r$ *new_character_data_ptr*
⟹ *ptr ≠ new_character_data_ptr* ⟹ *preserved (get_M*$_{CharacterData}$ *ptr getter) h h'"*
⟨*proof*⟩

5.5.2 Modified Heaps

lemma *get_CharacterData_ptr_simp [simp]:*
"get$_{CharacterData}$ *character_data_ptr (put*$_{Object}$ *ptr obj h)*
= (if ptr = cast character_data_ptr then cast obj else get character_data_ptr h)"
⟨*proof*⟩

lemma *Character_data_ptr_kinds_simp [simp]:*
"character_data_ptr_kinds (put$_{Object}$ *ptr obj h) = character_data_ptr_kinds h |∪|*

```
                          (if is_character_data_ptr_kind ptr then {|the (cast ptr)|} else {||})"
  ⟨proof⟩
```

lemma *type_wf_put_I:*
  assumes *"type_wf h"*
  assumes *"ElementClass.type_wf (put$_{Object}$ ptr obj h)"*
  assumes *"is_character_data_ptr_kind ptr ⟹ is_character_data_kind obj"*
  shows *"type_wf (put$_{Object}$ ptr obj h)"*
  ⟨*proof*⟩

lemma *type_wf_put_ptr_not_in_heap_E:*
  assumes *"type_wf (put$_{Object}$ ptr obj h)"*
  assumes *"ptr |∉| object_ptr_kinds h"*
  shows *"type_wf h"*
  ⟨*proof*⟩

lemma *type_wf_put_ptr_in_heap_E:*
  assumes *"type_wf (put$_{Object}$ ptr obj h)"*
  assumes *"ptr |∈| object_ptr_kinds h"*
  assumes *"ElementClass.type_wf h"*
  assumes *"is_character_data_ptr_kind ptr ⟹ is_character_data_kind (the (get ptr h))"*
  shows *"type_wf h"*
  ⟨*proof*⟩

5.5.3 Preserving Types

lemma *new_element_type_wf_preserved [simp]:*
  assumes *"h ⊢ new_element →$_h$ h'"*
  shows *"type_wf h = type_wf h'"*
  ⟨*proof*⟩

lemma *new_element_is_l_new_element: "l_new_element type_wf"*
  ⟨*proof*⟩

lemma *put_M$_{Element}$_tag_name_type_wf_preserved [simp]:*
  "h ⊢ put_M element_ptr tag_name_update v →$_h$ h' ⟹ type_wf h = type_wf h'"
  ⟨*proof*⟩

lemma *put_M$_{Element}$_child_nodes_type_wf_preserved [simp]:*
  "h ⊢ put_M element_ptr child_nodes_update v →$_h$ h' ⟹ type_wf h = type_wf h'"
  ⟨*proof*⟩

lemma *put_M$_{Element}$_attrs_type_wf_preserved [simp]:*
  "h ⊢ put_M element_ptr attrs_update v →$_h$ h' ⟹ type_wf h = type_wf h'"
  ⟨*proof*⟩

lemma *put_M$_{Element}$_shadow_root_opt_type_wf_preserved [simp]:*
  "h ⊢ put_M element_ptr shadow_root_opt_update v →$_h$ h' ⟹ type_wf h = type_wf h'"
  ⟨*proof*⟩

lemma *new_character_data_type_wf_preserved [simp]:*
  "h ⊢ new_character_data →$_h$ h' ⟹ type_wf h = type_wf h'"
  ⟨*proof*⟩

locale *l_new_character_data = l_type_wf +*
  assumes *new_character_data_types_preserved: "h ⊢ new_character_data →$_h$ h' ⟹ type_wf h = type_wf h'"*

lemma *new_character_data_is_l_new_character_data: "l_new_character_data type_wf"*
  ⟨*proof*⟩

lemma *put_M$_{CharacterData}$_val_type_wf_preserved [simp]:*

```
  "h ⊢ put_M character_data_ptr val_update v →h h' ⟹ type_wf h = type_wf h'"
  ⟨proof⟩

lemma character_data_ptr_kinds_small:
  assumes "⋀object_ptr. preserved (get_M_Object object_ptr RObject.nothing) h h'"
  shows "character_data_ptr_kinds h = character_data_ptr_kinds h'"
  ⟨proof⟩

lemma character_data_ptr_kinds_preserved:
  assumes "writes SW setter h h'"
  assumes "h ⊢ setter →h h'"
  assumes "⋀h h'. ∀w ∈ SW. h ⊢ w →h h'
          ⟶ (∀object_ptr. preserved (get_M_Object object_ptr RObject.nothing) h h')"
  shows "character_data_ptr_kinds h = character_data_ptr_kinds h'"
  ⟨proof⟩


lemma type_wf_preserved_small:
  assumes "⋀object_ptr. preserved (get_M_Object object_ptr RObject.nothing) h h'"
  assumes "⋀node_ptr. preserved (get_M_Node node_ptr RNode.nothing) h h'"
  assumes "⋀element_ptr. preserved (get_M_Element element_ptr RElement.nothing) h h'"
  assumes "⋀character_data_ptr. preserved (get_M_CharacterData character_data_ptr
                                            RCharacterData.nothing) h h'"
  shows "type_wf h = type_wf h'"
  ⟨proof⟩

lemma type_wf_preserved:
  assumes "writes SW setter h h'"
  assumes "h ⊢ setter →h h'"
  assumes "⋀h h' w. w ∈ SW ⟹ h ⊢ w →h h'
          ⟹ ∀object_ptr. preserved (get_M_Object object_ptr RObject.nothing) h h'"
  assumes "⋀h h' w. w ∈ SW ⟹ h ⊢ w →h h'
          ⟹ ∀node_ptr. preserved (get_M_Node node_ptr RNode.nothing) h h'"
  assumes "⋀h h' w. w ∈ SW ⟹ h ⊢ w →h h'
          ⟹ ∀element_ptr. preserved (get_M_Element element_ptr RElement.nothing) h h'"
  assumes "⋀h h' w. w ∈ SW ⟹ h ⊢ w →h h'
          ⟹ ∀character_data_ptr. preserved (get_M_CharacterData character_data_ptr
                                              RCharacterData.nothing) h h'"
  shows "type_wf h = type_wf h'"
⟨proof⟩

lemma type_wf_drop: "type_wf h ⟹ type_wf (Heap (fmdrop ptr (the_heap h)))"
  ⟨proof⟩

end
```

5.6 Document (DocumentMonad)

In this theory, we introduce the monadic method setup for the Document class.

```
theory DocumentMonad
  imports
    CharacterDataMonad
    "../classes/DocumentClass"
begin

type_synonym ('object_ptr, 'node_ptr, 'element_ptr, 'character_data_ptr, 'document_ptr,
    'shadow_root_ptr, 'Object, 'Node, 'Element, 'CharacterData, 'Document, 'result) dom_prog
  = "((_) heap, exception, 'result) prog"
register_default_tvars "('object_ptr, 'node_ptr, 'element_ptr, 'character_data_ptr, 'document_ptr,
           'shadow_root_ptr, 'Object, 'Node, 'Element, 'CharacterData, 'Document, 'result) dom_prog"
```

global_interpretation *l_ptr_kinds_M document_ptr_kinds* **defines** *document_ptr_kinds_M = a_ptr_kinds_M* ⟨*proof*⟩
lemmas *document_ptr_kinds_M_defs = a_ptr_kinds_M_def*

lemma *document_ptr_kinds_M_eq:*
  assumes "$|h \vdash$ *object_ptr_kinds_M*$|_r$ = $|h' \vdash$ *object_ptr_kinds_M*$|_r$"
  shows "$|h \vdash$ *document_ptr_kinds_M*$|_r$ = $|h' \vdash$ *document_ptr_kinds_M*$|_r$"
  ⟨*proof*⟩

lemma *document_ptr_kinds_M_reads:*
  "reads ($\bigcup$object_ptr. {preserved (get_M_{Object} object_ptr RObject.nothing)}) document_ptr_kinds_M h h'"
  ⟨*proof*⟩

global_interpretation *l_dummy* **defines** *get_$M_{Document}$ = "l_get_M.a_get_M get$_{Document}$"* ⟨*proof*⟩
lemma *get_M_is_l_get_M: "l_get_M get$_{Document}$ type_wf document_ptr_kinds"*
  ⟨*proof*⟩
lemmas *get_M_defs = get_$M_{Document}$_def[unfolded l_get_M.a_get_M_def[OF get_M_is_l_get_M]]*

adhoc_overloading *get_M* ⇌ *get_$M_{Document}$*

locale *l_get_$M_{Document}$_lemmas = l_type_wf$_{Document}$*
begin
sublocale *l_get_$M_{CharacterData}$_lemmas* ⟨*proof*⟩

interpretation *l_get_M get$_{Document}$ type_wf document_ptr_kinds*
  ⟨*proof*⟩
lemmas *get_$M_{Document}$_ok = get_M_ok[folded get_$M_{Document}$_def]*
end

global_interpretation *l_get_$M_{Document}$_lemmas type_wf* ⟨*proof*⟩

global_interpretation *l_put_M type_wf document_ptr_kinds get$_{Document}$ put$_{Document}$*
  rewrites *"a_get_M = get_$M_{Document}$"* **defines** *put_$M_{Document}$ = a_put_M*
   ⟨*proof*⟩

lemmas *put_M_defs = a_put_M_def*
adhoc_overloading *put_M* ⇌ *put_$M_{Document}$*

locale *l_put_$M_{Document}$_lemmas = l_type_wf$_{Document}$*
begin
sublocale *l_put_$M_{CharacterData}$_lemmas* ⟨*proof*⟩

interpretation *l_put_M type_wf document_ptr_kinds get$_{Document}$ put$_{Document}$*
  ⟨*proof*⟩
lemmas *put_$M_{Document}$_ok = put_M_ok[folded put_$M_{Document}$_def]*
end

global_interpretation *l_put_$M_{Document}$_lemmas type_wf* ⟨*proof*⟩

lemma *document_put_get [simp]:*
  "h ⊢ put_$M_{Document}$ document_ptr setter v $\rightarrow_h$ h'
     $\Longrightarrow$ ($\bigwedge$x. getter (setter (λ_. v) x) = v)
     $\Longrightarrow$ h' ⊢ get_$M_{Document}$ document_ptr getter $\rightarrow_r$ v"
  ⟨*proof*⟩
lemma *get_M_Mdocument_preserved1 [simp]:*
  "document_ptr $\neq$ document_ptr'
    $\Longrightarrow$ h ⊢ put_$M_{Document}$ document_ptr setter v $\rightarrow_h$ h'
    $\Longrightarrow$ preserved (get_$M_{Document}$ document_ptr' getter) h h'"
  ⟨*proof*⟩
lemma *document_put_get_preserved [simp]:*
  "h ⊢ put_$M_{Document}$ document_ptr setter v $\rightarrow_h$ h'

```
   ⟹ (⋀x. getter (setter (λ_. v) x) = getter x)
   ⟹ preserved (get_M$_{Document}$ document_ptr' getter) h h'"
  ⟨proof⟩

lemma get_M_Mdocument_preserved2 [simp]:
  "h ⊢ put_M$_{Document}$ document_ptr setter v →$_h$ h' ⟹ preserved (get_M$_{Node}$ node_ptr getter) h h'"
  ⟨proof⟩

lemma get_M_Mdocument_preserved3 [simp]:
  "cast document_ptr ≠ object_ptr
   ⟹ h ⊢ put_M$_{Document}$ document_ptr setter v →$_h$ h'
   ⟹ preserved (get_M$_{Object}$ object_ptr getter) h h'"
  ⟨proof⟩
lemma get_M_Mdocument_preserved4  [simp]:
  "h ⊢ put_M$_{Document}$ document_ptr setter v →$_h$ h'
   ⟹ (⋀x. getter (cast (setter (λ_. v) x)) = getter (cast x))
   ⟹ preserved (get_M$_{Object}$ object_ptr getter) h h'"
  ⟨proof⟩

lemma get_M_Mdocument_preserved5 [simp]:
  "cast document_ptr ≠ object_ptr
  ⟹ h ⊢ put_M$_{Object}$ object_ptr setter v →$_h$ h'
  ⟹ preserved (get_M$_{Document}$ document_ptr getter) h h'"
  ⟨proof⟩

lemma get_M_Mdocument_preserved6 [simp]:
  "h ⊢ put_M$_{Document}$ document_ptr setter v →$_h$ h' ⟹ preserved (get_M$_{Element}$ element_ptr getter) h h'"
  ⟨proof⟩
lemma get_M_Mdocument_preserved7 [simp]:
  "h ⊢ put_M$_{Element}$ element_ptr setter v →$_h$ h' ⟹ preserved (get_M$_{Document}$ document_ptr getter) h h'"
  ⟨proof⟩
lemma get_M_Mdocument_preserved8 [simp]:
  "h ⊢ put_M$_{Document}$ document_ptr setter v →$_h$ h'
    ⟹ preserved (get_M$_{CharacterData}$ character_data_ptr getter) h h'"
  ⟨proof⟩
lemma get_M_Mdocument_preserved9 [simp]:
  "h ⊢ put_M$_{CharacterData}$ character_data_ptr setter v →$_h$ h'
    ⟹ preserved (get_M$_{Document}$ document_ptr getter) h h'"
  ⟨proof⟩
lemma get_M_Mdocument_preserved10 [simp]:
  "(⋀x. getter (cast (setter (λ_. v) x)) = getter (cast x))
    ⟹ h ⊢ put_M$_{Document}$ document_ptr setter v →$_h$ h' ⟹ preserved (get_M$_{Object}$ object_ptr getter) h
h'"
  ⟨proof⟩

lemma new_element_get_M$_{Document}$:
  "h ⊢ new_element →$_h$ h' ⟹ preserved (get_M$_{Document}$ ptr getter) h h'"
  ⟨proof⟩

lemma new_character_data_get_M$_{Document}$:
  "h ⊢ new_character_data →$_h$ h' ⟹ preserved (get_M$_{Document}$ ptr getter) h h'"
  ⟨proof⟩
```

5.6.1 Creating Documents

```
definition new_document :: "(_, (_) document_ptr) dom_prog"
  where
    "new_document = do {
      h ← get_heap;
      (new_ptr, h') ← return (new$_{Document}$ h);
      return_heap h';
      return new_ptr
    }"
```

lemma *new_document_ok [simp]:*
  "h ⊢ ok new_document"
  ⟨*proof*⟩

lemma *new_document_ptr_in_heap:*
  assumes *"h ⊢ new_document* $\rightarrow_h$ *h'"*
    and *"h ⊢ new_document* $\rightarrow_r$ *new_document_ptr"*
  shows *"new_document_ptr |∈| document_ptr_kinds h'"*
  ⟨*proof*⟩

lemma *new_document_ptr_not_in_heap:*
  assumes *"h ⊢ new_document* $\rightarrow_h$ *h'"*
    and *"h ⊢ new_document* $\rightarrow_r$ *new_document_ptr"*
  shows *"new_document_ptr |∉| document_ptr_kinds h"*
  ⟨*proof*⟩

lemma *new_document_new_ptr:*
  assumes *"h ⊢ new_document* $\rightarrow_h$ *h'"*
    and *"h ⊢ new_document* $\rightarrow_r$ *new_document_ptr"*
  shows *"object_ptr_kinds h' = object_ptr_kinds h |∪| {|cast new_document_ptr|}"*
  ⟨*proof*⟩

lemma *new_document_is_document_ptr:*
  assumes *"h ⊢ new_document* $\rightarrow_r$ *new_document_ptr"*
  shows *"is_document_ptr new_document_ptr"*
  ⟨*proof*⟩

lemma *new_document_doctype:*
  assumes *"h ⊢ new_document* $\rightarrow_h$ *h'"*
  assumes *"h ⊢ new_document* $\rightarrow_r$ *new_document_ptr"*
  shows *"h' ⊢ get_M new_document_ptr doctype* $\rightarrow_r$ *''''"*
  ⟨*proof*⟩

lemma *new_document_document_element:*
  assumes *"h ⊢ new_document* $\rightarrow_h$ *h'"*
  assumes *"h ⊢ new_document* $\rightarrow_r$ *new_document_ptr"*
  shows *"h' ⊢ get_M new_document_ptr document_element* $\rightarrow_r$ *None"*
  ⟨*proof*⟩

lemma *new_document_disconnected_nodes:*
  assumes *"h ⊢ new_document* $\rightarrow_h$ *h'"*
  assumes *"h ⊢ new_document* $\rightarrow_r$ *new_document_ptr"*
  shows *"h' ⊢ get_M new_document_ptr disconnected_nodes* $\rightarrow_r$ *[]"*
  ⟨*proof*⟩

lemma *new_document_get_*M_{Object}*:*
  "h ⊢ new_document $\rightarrow_h$ *h'* ⟹ *h ⊢ new_document* $\rightarrow_r$ *new_document_ptr*
    ⟹ *ptr ≠ cast new_document_ptr* ⟹ *preserved (get_*M_{Object} *ptr getter) h h'"*
  ⟨*proof*⟩
lemma *new_document_get_*M_{Node}*:*
  "h ⊢ new_document $\rightarrow_h$ *h'* ⟹ *h ⊢ new_document* $\rightarrow_r$ *new_document_ptr*
    ⟹ *preserved (get_*M_{Node} *ptr getter) h h'"*
  ⟨*proof*⟩
lemma *new_document_get_*$M_{Element}$*:*
  "h ⊢ new_document $\rightarrow_h$ *h'* ⟹ *h ⊢ new_document* $\rightarrow_r$ *new_document_ptr*
    ⟹ *preserved (get_*$M_{Element}$ *ptr getter) h h'"*
  ⟨*proof*⟩
lemma *new_document_get_*$M_{CharacterData}$*:*
  "h ⊢ new_document $\rightarrow_h$ *h'* ⟹ *h ⊢ new_document* $\rightarrow_r$ *new_document_ptr*
    ⟹ *preserved (get_*$M_{CharacterData}$ *ptr getter) h h'"*
  ⟨*proof*⟩

lemma *new_document_get_$M_{Document}$:*
  "h ⊢ new_document →$_h$ h'
    ⟹ h ⊢ new_document →$_r$ new_document_ptr ⟹ ptr ≠ new_document_ptr
    ⟹ preserved (get_$M_{Document}$ ptr getter) h h'"
  ⟨proof⟩

5.6.2 Modified Heaps

lemma *get_document_ptr_simp [simp]:*
  "get$_{Document}$ document_ptr (put$_{Object}$ ptr obj h)
    = (if ptr = cast document_ptr then cast obj else get document_ptr h)"
  ⟨proof⟩

lemma *document_ptr_kidns_simp [simp]:*
  "document_ptr_kinds (put$_{Object}$ ptr obj h)
    = document_ptr_kinds h |∪| (if is_document_ptr_kind ptr then {|the (cast ptr)|} else {||})"
  ⟨proof⟩

lemma *type_wf_put_I:*
  assumes *"type_wf h"*
  assumes *"CharacterDataClass.type_wf (put$_{Object}$ ptr obj h)"*
  assumes *"is_document_ptr_kind ptr ⟹ is_document_kind obj"*
  shows *"type_wf (put$_{Object}$ ptr obj h)"*
  ⟨proof⟩

lemma *type_wf_put_ptr_not_in_heap_E:*
  assumes *"type_wf (put$_{Object}$ ptr obj h)"*
  assumes *"ptr |∉| object_ptr_kinds h"*
  shows *"type_wf h"*
  ⟨proof⟩

lemma *type_wf_put_ptr_in_heap_E:*
  assumes *"type_wf (put$_{Object}$ ptr obj h)"*
  assumes *"ptr |∈| object_ptr_kinds h"*
  assumes *"CharacterDataClass.type_wf h"*
  assumes *"is_document_ptr_kind ptr ⟹ is_document_kind (the (get ptr h))"*
  shows *"type_wf h"*
  ⟨proof⟩

5.6.3 Preserving Types

lemma *new_element_type_wf_preserved [simp]:*
  "h ⊢ new_element →$_h$ h' ⟹ type_wf h = type_wf h'"
  ⟨proof⟩

lemma *new_element_is_l_new_element [instances]:*
  "l_new_element type_wf"
  ⟨proof⟩

lemma *put_$M_{Element}$_tag_name_type_wf_preserved [simp]:*
  "h ⊢ put_M element_ptr tag_name_update v →$_h$ h' ⟹ type_wf h = type_wf h'"
  ⟨proof⟩

lemma *put_$M_{Element}$_child_nodes_type_wf_preserved [simp]:*
  "h ⊢ put_M element_ptr child_nodes_update v →$_h$ h' ⟹ type_wf h = type_wf h'"
  ⟨proof⟩

lemma *put_$M_{Element}$_attrs_type_wf_preserved [simp]:*
  "h ⊢ put_M element_ptr attrs_update v →$_h$ h' ⟹ type_wf h = type_wf h'"
  ⟨proof⟩

lemma *put_$M_{Element}$_shadow_root_opt_type_wf_preserved [simp]:*
  "h ⊢ put_M element_ptr shadow_root_opt_update v →$_h$ h' ⟹ type_wf h = type_wf h'"

⟨*proof*⟩

lemma *new_character_data_type_wf_preserved [simp]:*
  "h ⊢ new_character_data $\rightarrow_h$ h' ⟹ type_wf h = type_wf h'"
  ⟨*proof*⟩

lemma *new_character_data_is_l_new_character_data [instances]:*
  "l_new_character_data type_wf"
  ⟨*proof*⟩

lemma *put_$M_{CharacterData}$_val_type_wf_preserved [simp]:*
  "h ⊢ put_M character_data_ptr val_update v $\rightarrow_h$ h' ⟹ type_wf h = type_wf h'"
  ⟨*proof*⟩

lemma *new_document_type_wf_preserved [simp]: "h ⊢ new_document $\rightarrow_h$ h' ⟹ type_wf h = type_wf h'"*
  ⟨*proof*⟩

locale *l_new_document = l_type_wf +*
  assumes *new_document_types_preserved: "h ⊢ new_document $\rightarrow_h$ h' ⟹ type_wf h = type_wf h'"*

lemma *new_document_is_l_new_document [instances]: "l_new_document type_wf"*
  ⟨*proof*⟩

lemma *put_$M_{Document}$_doctype_type_wf_preserved [simp]:*
  "h ⊢ put_M document_ptr doctype_update v $\rightarrow_h$ h' ⟹ type_wf h = type_wf h'"
  ⟨*proof*⟩

lemma *put_$M_{Document}$_document_element_type_wf_preserved [simp]:*
  "h ⊢ put_M document_ptr document_element_update v $\rightarrow_h$ h' ⟹ type_wf h = type_wf h'"
  ⟨*proof*⟩

lemma *put_$M_{Document}$_disconnected_nodes_type_wf_preserved [simp]:*
  "h ⊢ put_M document_ptr disconnected_nodes_update v $\rightarrow_h$ h' ⟹ type_wf h = type_wf h'"
  ⟨*proof*⟩

lemma *document_ptr_kinds_small:*
  assumes *"⋀object_ptr. preserved (get_M_{Object} object_ptr RObject.nothing) h h'"*
  shows *"document_ptr_kinds h = document_ptr_kinds h'"*
  ⟨*proof*⟩

lemma *document_ptr_kinds_preserved:*
  assumes *"writes SW setter h h'"*
  assumes *"h ⊢ setter $\rightarrow_h$ h'"*
  assumes *"⋀h h'. ∀w ∈ SW. h ⊢ w $\rightarrow_h$ h'*
          ⟶ (∀object_ptr. preserved (get_M_{Object} object_ptr RObject.nothing) h h')"
  shows *"document_ptr_kinds h = document_ptr_kinds h'"*
  ⟨*proof*⟩

lemma *type_wf_preserved_small:*
  assumes *"⋀object_ptr. preserved (get_M_{Object} object_ptr RObject.nothing) h h'"*
  assumes *"⋀node_ptr. preserved (get_M_{Node} node_ptr RNode.nothing) h h'"*
  assumes *"⋀element_ptr. preserved (get_$M_{Element}$ element_ptr RElement.nothing) h h'"*
  assumes *"⋀character_data_ptr. preserved*
                      (get_$M_{CharacterData}$ character_data_ptr RCharacterData.nothing) h h'"
  assumes *"⋀document_ptr. preserved (get_$M_{Document}$ document_ptr RDocument.nothing) h h'"*
  shows *"DocumentClass.type_wf h = DocumentClass.type_wf h'"*
  ⟨*proof*⟩

lemma *type_wf_preserved:*
  assumes *"writes SW setter h h'"*
  assumes *"h ⊢ setter $\rightarrow_h$ h'"*
  assumes *"⋀h h' w. w ∈ SW ⟹ h ⊢ w $\rightarrow_h$ h'*

```
        ⟹ ∀object_ptr. preserved (get_M_Object object_ptr RObject.nothing) h h'"
  assumes "⋀h h' w. w ∈ SW ⟹ h ⊢ w →_h h'
        ⟹ ∀node_ptr. preserved (get_M_Node node_ptr RNode.nothing) h h'"
  assumes "⋀h h' w. w ∈ SW ⟹ h ⊢ w →_h h'
        ⟹ ∀element_ptr. preserved (get_M_Element element_ptr RElement.nothing) h h'"
  assumes "⋀h h' w. w ∈ SW ⟹ h ⊢ w →_h h'
        ⟹ ∀character_data_ptr. preserved
                 (get_M_CharacterData character_data_ptr RCharacterData.nothing) h h'"
  assumes "⋀h h' w. w ∈ SW ⟹ h ⊢ w →_h h'
        ⟹ ∀document_ptr. preserved (get_M_Document document_ptr RDocument.nothing) h h'"
  shows "DocumentClass.type_wf h = DocumentClass.type_wf h'"
⟨proof⟩

lemma type_wf_drop: "type_wf h ⟹ type_wf (Heap (fmdrop ptr (the_heap h)))"
  ⟨proof⟩
end
```

6 The Core DOM

In this chapter, we introduce the formalization of the core DOM, i.e., the most important algorithms for querying or modifying the DOM, as defined in the standard. For more details, we refer the reader to [4].

6.1 Basic Data Types (Core_DOM_Basic_Datatypes)

This theory formalizes the primitive data types used by the DOM standard [1].

```
theory Core_DOM_Basic_Datatypes
  imports
    Main
begin
```

```
type_synonym USVString = string
```

In the official standard, the type *USVString* corresponds to the set of all possible sequences of Unicode scalar values. As we are not interested in analyzing the specifics of Unicode strings, we just model *USVString* using the standard type *string* of Isabelle/HOL.

```
type_synonym DOMString = string
```

In the official standard, the type *DOMString* corresponds to the set of all possible sequences of code units, commonly interpreted as UTF-16 encoded strings. Again, as we are not interested in analyzing the specifics of Unicode strings, we just model *DOMString* using the standard type *string* of Isabelle/HOL.

```
type_synonym doctype = DOMString
```

Examples
```
definition html :: doctype
  where "html = ''<!DOCTYPE html>''"
```

```
hide_const id
```

This dummy locale is used to create scoped definitions by using global interpretations and defines.

```
locale l_dummy
end
```

6.2 Querying and Modifying the DOM (Core_DOM_Functions)

In this theory, we are formalizing the functions for querying and modifying the DOM.

```
theory Core_DOM_Functions
imports
  "monads/DocumentMonad"
begin
```

If we do not declare show_variants, then all abbreviations that contain constants that are overloaded by using adhoc_overloading get immediately unfolded.

```
declare [[show_variants]]
```

6.2.1 Various Functions

```
lemma insort_split: "x ∈ set (insort y xs) ⟷ (x = y ∨ x ∈ set xs)"
  ⟨proof⟩
```

```
lemma concat_map_distinct:
  "distinct (concat (map f xs)) ⟹ y ∈ set (concat (map f xs)) ⟹ ∃!x ∈ set xs. y ∈ set (f x)"
  ⟨proof⟩

lemma concat_map_all_distinct: "distinct (concat (map f xs)) ⟹ x ∈ set xs ⟹ distinct (f x)"
  ⟨proof⟩

lemma distinct_concat_map_I:
  assumes "distinct xs"
    and "⋀x. x ∈ set xs ⟹ distinct (f x)"
and "⋀x y. x ∈ set xs ⟹ y ∈ set xs ⟹ x ≠ y ⟹ (set (f x)) ∩ (set (f y)) = {}"
shows "distinct (concat ((map f xs)))"
  ⟨proof⟩

lemma distinct_concat_map_E:
  assumes "distinct (concat ((map f xs)))"
  shows "⋀x y. x ∈ set xs ⟹ y ∈ set xs ⟹ x ≠ y ⟹ (set (f x)) ∩ (set (f y)) = {}"
    and "⋀x. x ∈ set xs ⟹ distinct (f x)"
  ⟨proof⟩

lemma bind_is_OK_E3 [elim]:
  assumes "h ⊢ ok (f ≫= g)" and "pure f h"
  obtains x where "h ⊢ f →_r x"  and "h ⊢ ok (g x)"
  ⟨proof⟩
```

6.2.2 Basic Functions

get_child_nodes

```
locale l_get_child_nodes_{Core_DOM}_defs
begin

definition get_child_nodes_{element_ptr} :: "(_) element_ptr ⇒ unit ⇒ (_, (_) node_ptr list) dom_prog"
  where
    "get_child_nodes_{element_ptr} element_ptr _ = get_M element_ptr RElement.child_nodes"

definition get_child_nodes_{character_data_ptr} :: "(_) character_data_ptr ⇒ unit ⇒ (_, (_) node_ptr list) dom_prog"
  where
    "get_child_nodes_{character_data_ptr} _ _ = return []"

definition get_child_nodes_{document_ptr} :: "(_) document_ptr ⇒ unit ⇒ (_, (_) node_ptr list) dom_prog"
  where
    "get_child_nodes_{document_ptr} document_ptr _ = do {
    doc_elem ← get_M document_ptr document_element;
    (case doc_elem of
      Some element_ptr ⇒ return [cast element_ptr]
    | None ⇒ return [])
  }"

definition a_get_child_nodes_tups :: "(((_) object_ptr ⇒ bool) × ((_) object_ptr ⇒ unit
  ⇒ (_, (_) node_ptr list) dom_prog)) list"
  where
    "a_get_child_nodes_tups = [
        (is_element_ptr, get_child_nodes_{element_ptr} ∘ the ∘ cast),
        (is_character_data_ptr, get_child_nodes_{character_data_ptr} ∘ the ∘ cast),
        (is_document_ptr, get_child_nodes_{document_ptr} ∘ the ∘ cast)
     ]"

definition a_get_child_nodes :: "(_) object_ptr ⇒ (_, (_) node_ptr list) dom_prog"
  where
    "a_get_child_nodes ptr = invoke a_get_child_nodes_tups ptr ()"

definition a_get_child_nodes_locs :: "(_) object_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"
  where
    "a_get_child_nodes_locs ptr ≡
```

```
      (if is_element_ptr_kind ptr then {preserved (get_M (the (cast ptr)) RElement.child_nodes)} else {})
∪
      (if is_document_ptr_kind ptr then {preserved (get_M (the (cast ptr)) RDocument.document_element)}
else {}) ∪
      {preserved (get_M ptr RObject.nothing)}"

definition first_child :: "(_) object_ptr ⇒ (_, (_) node_ptr option) dom_prog"
  where
    "first_child ptr = do {
      children ← a_get_child_nodes ptr;
      return (case children of [] ⇒ None | child#_ ⇒ Some child)}"
end

locale l_get_child_nodes_defs =
  fixes get_child_nodes :: "(_) object_ptr ⇒ (_, (_) node_ptr list) dom_prog"
  fixes get_child_nodes_locs :: "(_) object_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"

locale l_get_child_nodes_Core_DOM =
  l_type_wf type_wf +
  l_known_ptr known_ptr +
  l_get_child_nodes_defs get_child_nodes get_child_nodes_locs +
  l_get_child_nodes_Core_DOM_defs
  for type_wf :: "(_) heap ⇒ bool"
  and known_ptr :: "(_) object_ptr ⇒ bool"
  and get_child_nodes :: "(_) object_ptr ⇒ (_, (_) node_ptr list) dom_prog"
  and get_child_nodes_locs :: "(_) object_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set" +
  assumes known_ptr_impl: "known_ptr = DocumentClass.known_ptr"
  assumes type_wf_impl: "type_wf = DocumentClass.type_wf"
  assumes get_child_nodes_impl: "get_child_nodes = a_get_child_nodes"
  assumes get_child_nodes_locs_impl: "get_child_nodes_locs = a_get_child_nodes_locs"
begin
lemmas get_child_nodes_def = get_child_nodes_impl[unfolded a_get_child_nodes_def]
lemmas get_child_nodes_locs_def = get_child_nodes_locs_impl[unfolded a_get_child_nodes_locs_def]

lemma get_child_nodes_split:
  "P (invoke (a_get_child_nodes_tups @ xs) ptr ()) =
    ((known_ptr ptr ⟶ P (get_child_nodes ptr))
    ∧ (¬(known_ptr ptr) ⟶ P (invoke xs ptr ())))"
  ⟨proof⟩

lemma get_child_nodes_split_asm:
  "P (invoke (a_get_child_nodes_tups @ xs) ptr ()) =
    (¬((known_ptr ptr ∧ ¬P (get_child_nodes ptr))
    ∨ (¬(known_ptr ptr) ∧ ¬P (invoke xs ptr ()))))"
  ⟨proof⟩

lemmas get_child_nodes_splits = get_child_nodes_split get_child_nodes_split_asm

lemma get_child_nodes_ok [simp]:
  assumes "known_ptr ptr"
  assumes "type_wf h"
  assumes "ptr |∈| object_ptr_kinds h"
  shows "h ⊢ ok (get_child_nodes ptr)"
  ⟨proof⟩

lemma get_child_nodes_ptr_in_heap [simp]:
  assumes "h ⊢ get_child_nodes ptr →r children"
  shows "ptr |∈| object_ptr_kinds h"
  ⟨proof⟩

lemma get_child_nodes_pure [simp]:
  "pure (get_child_nodes ptr) h"
  ⟨proof⟩
```

```
lemma get_child_nodes_reads: "reads (get_child_nodes_locs ptr) (get_child_nodes ptr) h h'"
  ⟨proof⟩
end

locale l_get_child_nodes = l_type_wf + l_known_ptr + l_get_child_nodes_defs +
  assumes get_child_nodes_reads: "reads (get_child_nodes_locs ptr) (get_child_nodes ptr) h h'"
  assumes get_child_nodes_ok: "type_wf h ⟹ known_ptr ptr ⟹ ptr |∈| object_ptr_kinds h
                                         ⟹ h ⊢ ok (get_child_nodes ptr)"
  assumes get_child_nodes_ptr_in_heap: "h ⊢ ok (get_child_nodes ptr) ⟹ ptr |∈| object_ptr_kinds h"
  assumes get_child_nodes_pure [simp]: "pure (get_child_nodes ptr) h"
```

global_interpretation $l_get_child_nodes_{Core_DOM}$_defs defines
  get_child_nodes = $l_get_child_nodes_{Core_DOM}$_defs.a_get_child_nodes and
  get_child_nodes_locs = $l_get_child_nodes_{Core_DOM}$_defs.a_get_child_nodes_locs
  ⟨proof⟩

interpretation
  i_get_child_nodes?: $l\_get\_child\_nodes_{Core\_DOM}$ type_wf known_ptr get_child_nodes get_child_nodes_locs
  ⟨proof⟩
declare $l_get_child_nodes_{Core_DOM}$_axioms[instances]

```
lemma get_child_nodes_is_l_get_child_nodes [instances]:
  "l_get_child_nodes type_wf known_ptr get_child_nodes get_child_nodes_locs"
  ⟨proof⟩
```

new_element locale $l\_new\_element\_get\_child\_nodes_{Core\_DOM}$ =
  $l\_get\_child\_nodes_{Core\_DOM}$ type_wf known_ptr get_child_nodes get_child_nodes_locs

```
  for type_wf :: "(_) heap ⇒ bool"
  and known_ptr :: "(_) object_ptr ⇒ bool"
  and get_child_nodes :: "(_) object_ptr ⇒ ((_) heap, exception, (_) node_ptr list) prog"
  and get_child_nodes_locs :: "(_) object_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"
begin
lemma get_child_nodes_new_element:
  "ptr' ≠ cast new_element_ptr ⟹ h ⊢ new_element →r new_element_ptr ⟹ h ⊢ new_element →h h'
     ⟹ r ∈ get_child_nodes_locs ptr' ⟹ r h h'"
  ⟨proof⟩

lemma new_element_no_child_nodes:
  "h ⊢ new_element →r new_element_ptr ⟹ h ⊢ new_element →h h'
   ⟹ h' ⊢ get_child_nodes (cast new_element_ptr) →r []"
  ⟨proof⟩
end

locale l_new_element_get_child_nodes = l_new_element + l_get_child_nodes +
  assumes get_child_nodes_new_element:
          "ptr' ≠ cast new_element_ptr ⟹ h ⊢ new_element →r new_element_ptr
            ⟹ h ⊢ new_element →h h' ⟹ r ∈ get_child_nodes_locs ptr' ⟹ r h h'"
        assumes new_element_no_child_nodes:
          "h ⊢ new_element →r new_element_ptr ⟹ h ⊢ new_element →h h'
            ⟹ h' ⊢ get_child_nodes (cast new_element_ptr) →r []"
```

interpretation i_new_element_get_child_nodes?:
  $l\_new\_element\_get\_child\_nodes_{Core\_DOM}$ type_wf known_ptr get_child_nodes get_child_nodes_locs
  ⟨proof⟩
declare $l_new_element_get_child_nodes_{Core_DOM}$_axioms[instances]

```
lemma new_element_get_child_nodes_is_l_new_element_get_child_nodes [instances]:
  "l_new_element_get_child_nodes type_wf known_ptr get_child_nodes get_child_nodes_locs"
  ⟨proof⟩
```

new_character_data locale $l\_new\_character\_data\_get\_child\_nodes_{Core\_DOM}$ =
  $l\_get\_child\_nodes_{Core\_DOM}$ type_wf known_ptr get_child_nodes get_child_nodes_locs

```
  for type_wf :: "(_) heap ⇒ bool"
  and known_ptr :: "(_) object_ptr ⇒ bool"
  and get_child_nodes :: "(_) object_ptr ⇒ ((_) heap, exception, (_) node_ptr list) prog"
  and get_child_nodes_locs :: "(_) object_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"
begin
lemma get_child_nodes_new_character_data:
  "ptr' ≠ cast new_character_data_ptr ⟹ h ⊢ new_character_data →r new_character_data_ptr
   ⟹ h ⊢ new_character_data →h h' ⟹ r ∈ get_child_nodes_locs ptr' ⟹ r h h'"
  ⟨proof⟩

lemma new_character_data_no_child_nodes:
  "h ⊢ new_character_data →r new_character_data_ptr ⟹ h ⊢ new_character_data →h h'
  ⟹ h' ⊢ get_child_nodes (cast new_character_data_ptr) →r []"
  ⟨proof⟩
end

locale l_new_character_data_get_child_nodes = l_new_character_data + l_get_child_nodes +
  assumes get_child_nodes_new_character_data:
   "ptr' ≠ cast new_character_data_ptr ⟹ h ⊢ new_character_data →r new_character_data_ptr
       ⟹ h ⊢ new_character_data →h h' ⟹ r ∈ get_child_nodes_locs ptr' ⟹ r h h'"
 assumes new_character_data_no_child_nodes:
  "h ⊢ new_character_data →r new_character_data_ptr ⟹ h ⊢ new_character_data →h h'
       ⟹ h' ⊢ get_child_nodes (cast new_character_data_ptr) →r []"

interpretation i_new_character_data_get_child_nodes?:
  l_new_character_data_get_child_nodesCore_DOM type_wf known_ptr get_child_nodes get_child_nodes_locs
  ⟨proof⟩
declare l_new_character_data_get_child_nodesCore_DOM_axioms[instances]

lemma new_character_data_get_child_nodes_is_l_new_character_data_get_child_nodes [instances]:
  "l_new_character_data_get_child_nodes type_wf known_ptr get_child_nodes get_child_nodes_locs"
  ⟨proof⟩
```

new_document locale l_new_document_get_child_nodes$_{Core\_DOM}$ =

```
  l_get_child_nodesCore_DOM type_wf known_ptr get_child_nodes get_child_nodes_locs
  for type_wf :: "(_) heap ⇒ bool"
  and known_ptr :: "(_) object_ptr ⇒ bool"
  and get_child_nodes :: "(_) object_ptr ⇒ ((_) heap, exception, (_) node_ptr list) prog"
  and get_child_nodes_locs :: "(_) object_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"
begin
lemma get_child_nodes_new_document:
  "ptr' ≠ cast new_document_ptr ⟹ h ⊢ new_document →r new_document_ptr
     ⟹ h ⊢ new_document →h h' ⟹ r ∈ get_child_nodes_locs ptr' ⟹ r h h'"
  ⟨proof⟩

lemma new_document_no_child_nodes:
  "h ⊢ new_document →r new_document_ptr ⟹ h ⊢ new_document →h h'
     ⟹ h' ⊢ get_child_nodes (cast new_document_ptr) →r []"
  ⟨proof⟩
end

locale l_new_document_get_child_nodes = l_new_document + l_get_child_nodes +
  assumes get_child_nodes_new_document:
   "ptr' ≠ cast new_document_ptr ⟹ h ⊢ new_document →r new_document_ptr
     ⟹ h ⊢ new_document →h h' ⟹ r ∈ get_child_nodes_locs ptr' ⟹ r h h'"
 assumes new_document_no_child_nodes:
   "h ⊢ new_document →r new_document_ptr ⟹ h ⊢ new_document →h h'
     ⟹ h' ⊢ get_child_nodes (cast new_document_ptr) →r []"

interpretation i_new_document_get_child_nodes?:
  l_new_document_get_child_nodesCore_DOM type_wf known_ptr get_child_nodes get_child_nodes_locs
  ⟨proof⟩
declare l_new_document_get_child_nodesCore_DOM_axioms[instances]
```

```
lemma new_document_get_child_nodes_is_l_new_document_get_child_nodes [instances]:
  "l_new_document_get_child_nodes type_wf known_ptr get_child_nodes get_child_nodes_locs"
  ⟨proof⟩
```

set_child_nodes

```
locale l_set_child_nodes_Core_DOM_defs
begin
definition set_child_nodes_element_ptr ::
  "(_) element_ptr ⇒ (_) node_ptr list ⇒ (_, unit) dom_prog"
  where
  "set_child_nodes_element_ptr element_ptr children = put_M element_ptr RElement.child_nodes_update children"

definition set_child_nodes_character_data_ptr ::
  "(_) character_data_ptr ⇒ (_) node_ptr list ⇒ (_, unit) dom_prog"
  where
    "set_child_nodes_character_data_ptr _ _ =  error HierarchyRequestError"

definition set_child_nodes_document_ptr :: "(_) document_ptr ⇒ (_) node_ptr list ⇒ (_, unit) dom_prog"
  where
    "set_child_nodes_document_ptr document_ptr children = do {
      (case children of
        [] ⇒ put_M document_ptr document_element_update None
      | child # [] ⇒ (case cast child of
          Some element_ptr ⇒ put_M document_ptr document_element_update (Some element_ptr)
        | None ⇒ error HierarchyRequestError)
      | _ ⇒ error HierarchyRequestError)
    }"

definition a_set_child_nodes_tups ::
  "(((_) object_ptr ⇒ bool) × ((_) object_ptr ⇒ (_) node_ptr list ⇒ (_, unit) dom_prog)) list"
  where
    "a_set_child_nodes_tups ≡ [
        (is_element_ptr, set_child_nodes_element_ptr ∘ the ∘ cast),
        (is_character_data_ptr, set_child_nodes_character_data_ptr ∘ the ∘ cast),
        (is_document_ptr, set_child_nodes_document_ptr ∘ the ∘ cast)
     ]"

definition a_set_child_nodes :: "(_) object_ptr ⇒ (_) node_ptr list ⇒ (_, unit) dom_prog"
  where
    "a_set_child_nodes ptr children = invoke a_set_child_nodes_tups ptr (children)"
lemmas set_child_nodes_defs = a_set_child_nodes_def

definition a_set_child_nodes_locs :: "(_) object_ptr ⇒ (_, unit) dom_prog set"
  where
    "a_set_child_nodes_locs ptr ≡
      (if is_element_ptr_kind ptr
          then all_args (put_M_Element (the (cast ptr)) RElement.child_nodes_update) else {}) ∪
      (if is_document_ptr_kind ptr
          then all_args (put_M_Document (the (cast ptr)) document_element_update) else {})"
end

locale l_set_child_nodes_defs =
  fixes set_child_nodes :: "(_) object_ptr ⇒ (_) node_ptr list ⇒ (_, unit) dom_prog"
  fixes set_child_nodes_locs :: "(_) object_ptr ⇒ (_, unit) dom_prog set"

locale l_set_child_nodes_Core_DOM =
  l_type_wf type_wf +
  l_known_ptr known_ptr +
  l_set_child_nodes_defs set_child_nodes set_child_nodes_locs +
  l_set_child_nodes_Core_DOM_defs
  for type_wf :: "(_) heap ⇒ bool"
  and known_ptr :: "(_) object_ptr ⇒ bool"
  and set_child_nodes :: "(_) object_ptr ⇒ (_) node_ptr list ⇒ (_, unit) dom_prog"
  and set_child_nodes_locs :: "(_) object_ptr ⇒ (_, unit) dom_prog set" +
  assumes known_ptr_impl: "known_ptr = DocumentClass.known_ptr"
  assumes type_wf_impl: "type_wf = DocumentClass.type_wf"
  assumes set_child_nodes_impl: "set_child_nodes = a_set_child_nodes"
  assumes set_child_nodes_locs_impl: "set_child_nodes_locs = a_set_child_nodes_locs"
begin
lemmas set_child_nodes_def = set_child_nodes_impl[unfolded a_set_child_nodes_def]
lemmas set_child_nodes_locs_def = set_child_nodes_locs_impl[unfolded a_set_child_nodes_locs_def]

lemma set_child_nodes_split:
  "P (invoke (a_set_child_nodes_tups @ xs) ptr (children)) =
    ((known_ptr ptr ⟶ P (set_child_nodes ptr children))
    ∧ (¬(known_ptr ptr) ⟶ P (invoke xs ptr (children))))"
  ⟨proof⟩

lemma set_child_nodes_split_asm:
  "P (invoke (a_set_child_nodes_tups @ xs) ptr (children)) =
    (¬((known_ptr ptr ∧ ¬P (set_child_nodes ptr children))
    ∨ (¬(known_ptr ptr) ∧ ¬P (invoke xs ptr (children)))))"
  ⟨proof⟩
lemmas set_child_nodes_splits = set_child_nodes_split set_child_nodes_split_asm

lemma set_child_nodes_writes: "writes (set_child_nodes_locs ptr) (set_child_nodes ptr children) h h'"
  ⟨proof⟩

lemma set_child_nodes_pointers_preserved:
  assumes "w ∈ set_child_nodes_locs object_ptr"
  assumes "h ⊢ w →_h h'"
  shows "object_ptr_kinds h = object_ptr_kinds h'"
  ⟨proof⟩

lemma set_child_nodes_typess_preserved:
  assumes "w ∈ set_child_nodes_locs object_ptr"
  assumes "h ⊢ w →_h h'"
  shows "type_wf h = type_wf h'"
  ⟨proof⟩
end

locale l_set_child_nodes = l_type_wf + l_set_child_nodes_defs +
  assumes set_child_nodes_writes:
   "writes (set_child_nodes_locs ptr) (set_child_nodes ptr children) h h'"
 assumes set_child_nodes_pointers_preserved:
   "w ∈ set_child_nodes_locs object_ptr ⟹ h ⊢ w →_h h' ⟹ object_ptr_kinds h = object_ptr_kinds h'"
 assumes set_child_nodes_types_preserved:
   "w ∈ set_child_nodes_locs object_ptr ⟹ h ⊢ w →_h h' ⟹ type_wf h = type_wf h'"

global_interpretation l_set_child_nodes_Core_DOM_defs defines
  set_child_nodes = l_set_child_nodes_Core_DOM_defs.a_set_child_nodes and
  set_child_nodes_locs = l_set_child_nodes_Core_DOM_defs.a_set_child_nodes_locs ⟨proof⟩

interpretation
  i_set_child_nodes?: l_set_child_nodes_Core_DOM type_wf known_ptr set_child_nodes set_child_nodes_locs
  ⟨proof⟩
declare l_set_child_nodes_Core_DOM_axioms[instances]


lemma set_child_nodes_is_l_set_child_nodes [instances]:
  "l_set_child_nodes type_wf set_child_nodes set_child_nodes_locs"
  ⟨proof⟩
```

get_child_nodes locale l_set_child_nodes_get_child_nodes_Core_DOM = l_get_child_nodes_Core_DOM + l_set_child_nod

```
begin

lemma set_child_nodes_get_child_nodes:
  assumes "known_ptr ptr"
  assumes "type_wf h"
  assumes "h ⊢ set_child_nodes ptr children →_h h'"
  shows "h' ⊢ get_child_nodes ptr →_r children"
⟨proof⟩

lemma set_child_nodes_get_child_nodes_different_pointers:
  assumes "ptr ≠ ptr'"
  assumes "w ∈ set_child_nodes_locs ptr"
  assumes "h ⊢ w →_h h'"
  assumes "r ∈ get_child_nodes_locs ptr'"
  shows "r h h'"
  ⟨proof⟩

lemma set_child_nodes_element_ok [simp]:
  assumes "known_ptr ptr"
  assumes "type_wf h"
  assumes "ptr |∈| object_ptr_kinds h"
  assumes "is_element_ptr_kind ptr"
  shows "h ⊢ ok (set_child_nodes ptr children)"
⟨proof⟩

lemma set_child_nodes_document1_ok [simp]:
  assumes "known_ptr ptr"
  assumes "type_wf h"
  assumes "ptr |∈| object_ptr_kinds h"
  assumes "is_document_ptr_kind ptr"
  assumes "children = []"
  shows "h ⊢ ok (set_child_nodes ptr children)"
⟨proof⟩

lemma set_child_nodes_document2_ok [simp]:
  assumes "known_ptr ptr"
  assumes "type_wf h"
  assumes "ptr |∈| object_ptr_kinds h"
  assumes "is_document_ptr_kind ptr"
  assumes "children = [child]"
  assumes "is_element_ptr_kind child"
  shows "h ⊢ ok (set_child_nodes ptr children)"
⟨proof⟩
end

locale l_set_child_nodes_get_child_nodes = l_get_child_nodes + l_set_child_nodes +
  assumes set_child_nodes_get_child_nodes:
    "type_wf h ⟹ known_ptr ptr
               ⟹ h ⊢ set_child_nodes ptr children →_h h' ⟹ h' ⊢ get_child_nodes ptr →_r children"
  assumes set_child_nodes_get_child_nodes_different_pointers:
    "ptr ≠ ptr' ⟹ w ∈ set_child_nodes_locs ptr ⟹ h ⊢ w →_h h'
                ⟹ r ∈ get_child_nodes_locs ptr' ⟹ r h h'"

interpretation
  i_set_child_nodes_get_child_nodes?: l_set_child_nodes_get_child_nodes_Core_DOM type_wf
  known_ptr get_child_nodes get_child_nodes_locs set_child_nodes set_child_nodes_locs
  ⟨proof⟩
declare l_set_child_nodes_get_child_nodes_Core_DOM_axioms[instances]

lemma set_child_nodes_get_child_nodes_is_l_set_child_nodes_get_child_nodes [instances]:
  "l_set_child_nodes_get_child_nodes type_wf known_ptr get_child_nodes get_child_nodes_locs
                                     set_child_nodes set_child_nodes_locs"
  ⟨proof⟩
```

get_attribute

```
locale l_get_attribute$_{Core\_DOM}$_defs
begin
definition a_get_attribute :: "(_) element_ptr ⇒ attr_key ⇒ (_, attr_value option) dom_prog"
  where
    "a_get_attribute ptr k = do {m ← get_M ptr attrs; return (fmlookup m k)}"
lemmas get_attribute_defs = a_get_attribute_def

definition a_get_attribute_locs :: "(_) element_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"
  where
    "a_get_attribute_locs element_ptr = {preserved (get_M element_ptr attrs)}"
end

locale l_get_attribute_defs =
  fixes get_attribute :: "(_) element_ptr ⇒ attr_key ⇒ (_, attr_value option) dom_prog"
  fixes get_attribute_locs :: "(_) element_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"

locale l_get_attribute$_{Core\_DOM}$ =
  l_type_wf type_wf +
  l_get_attribute_defs get_attribute get_attribute_locs +
  l_get_attribute$_{Core\_DOM}$_defs
  for type_wf :: "(_) heap ⇒ bool"
  and get_attribute :: "(_) element_ptr ⇒ attr_key ⇒ (_, attr_value option) dom_prog"
  and get_attribute_locs :: "(_) element_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set" +
  assumes type_wf_impl: "type_wf = DocumentClass.type_wf"
  assumes get_attribute_impl: "get_attribute = a_get_attribute"
  assumes get_attribute_locs_impl: "get_attribute_locs = a_get_attribute_locs"
begin
lemma get_attribute_pure [simp]: "pure (get_attribute ptr k) h"
  ⟨proof⟩

lemma get_attribute_ok:
  "type_wf h ⟹ element_ptr |∈| element_ptr_kinds h ⟹ h ⊢ ok (get_attribute element_ptr k)"
  ⟨proof⟩

lemma get_attribute_ptr_in_heap:
  "h ⊢ ok (get_attribute element_ptr k) ⟹ element_ptr |∈| element_ptr_kinds h"
  ⟨proof⟩

lemma get_attribute_reads:
  "reads (get_attribute_locs element_ptr) (get_attribute element_ptr k) h h'"
  ⟨proof⟩
end

locale l_get_attribute = l_type_wf + l_get_attribute_defs +
assumes get_attribute_reads:
  "reads (get_attribute_locs element_ptr) (get_attribute element_ptr k) h h'"
assumes get_attribute_ok:
  "type_wf h ⟹ element_ptr |∈| element_ptr_kinds h ⟹ h ⊢ ok (get_attribute element_ptr k)"
assumes get_attribute_ptr_in_heap:
  "h ⊢ ok (get_attribute element_ptr k) ⟹ element_ptr |∈| element_ptr_kinds h"
assumes get_attribute_pure [simp]: "pure (get_attribute element_ptr k) h"

global_interpretation l_get_attribute$_{Core\_DOM}$_defs defines
  get_attribute = l_get_attribute$_{Core\_DOM}$_defs.a_get_attribute and
  get_attribute_locs = l_get_attribute$_{Core\_DOM}$_defs.a_get_attribute_locs ⟨proof⟩

interpretation
  i_get_attribute?: l_get_attribute$_{Core\_DOM}$ type_wf get_attribute get_attribute_locs
  ⟨proof⟩
declare l_get_attribute$_{Core\_DOM}$_axioms[instances]
```

lemma *get_attribute_is_l_get_attribute [instances]:*
  "l_get_attribute type_wf get_attribute get_attribute_locs"
  ⟨*proof*⟩

set_attribute

locale *l_set_attribute*$_{Core\_DOM}$*_defs*
begin

definition
  a_set_attribute :: "(_) element_ptr ⇒ attr_key ⇒ attr_value option ⇒ (_, unit) dom_prog"
  where
    "a_set_attribute ptr k v = do {
      m ← get_M ptr attrs;
      put_M ptr attrs_update (if v = None then fmdrop k m else fmupd k (the v) m)
    }"

definition *a_set_attribute_locs :: "(_) element_ptr ⇒ (_, unit) dom_prog set"*
  where
    "a_set_attribute_locs element_ptr ≡ all_args (put_M element_ptr attrs_update)"
end

locale *l_set_attribute_defs =*
  fixes *set_attribute :: "(_) element_ptr ⇒ attr_key ⇒ attr_value option ⇒ (_, unit) dom_prog"*
  fixes *set_attribute_locs :: "(_) element_ptr ⇒ (_, unit) dom_prog set"*

locale *l_set_attribute*$_{Core\_DOM}$ *=*
  l_type_wf type_wf +
  l_set_attribute_defs set_attribute set_attribute_locs +
  l_set_attribute$_{Core\_DOM}$*_defs*
  for *type_wf :: "(_) heap ⇒ bool"*
  and *set_attribute :: "(_) element_ptr ⇒ attr_key ⇒ attr_value option ⇒ (_, unit) dom_prog"*
  and *set_attribute_locs :: "(_) element_ptr ⇒ (_, unit) dom_prog set" +*
  assumes *type_wf_impl: "type_wf = DocumentClass.type_wf"*
  assumes *set_attribute_impl: "set_attribute = a_set_attribute"*
  assumes *set_attribute_locs_impl: "set_attribute_locs = a_set_attribute_locs"*
begin
lemmas *set_attribute_def = set_attribute_impl[folded a_set_attribute_def]*
lemmas *set_attribute_locs_def = set_attribute_locs_impl[unfolded a_set_attribute_locs_def]*

lemma *set_attribute_ok: "type_wf h ⟹ element_ptr |∈| element_ptr_kinds h ⟹ h ⊢ ok (set_attribute element_ptr k v)"*
  ⟨*proof*⟩

lemma *set_attribute_writes:*
  "writes (set_attribute_locs element_ptr) (set_attribute element_ptr k v) h h'"
  ⟨*proof*⟩
end

locale *l_set_attribute = l_type_wf + l_set_attribute_defs +*
  assumes *set_attribute_writes:*
    "writes (set_attribute_locs element_ptr) (set_attribute element_ptr k v) h h'"
  assumes *set_attribute_ok:*
    "type_wf h ⟹ element_ptr |∈| element_ptr_kinds h ⟹ h ⊢ ok (set_attribute element_ptr k v)"

global_interpretation *l_set_attribute*$_{Core\_DOM}$*_defs* **defines**
  set_attribute = l_set_attribute$_{Core\_DOM}$*_defs.a_set_attribute* **and**
  set_attribute_locs = l_set_attribute$_{Core\_DOM}$*_defs.a_set_attribute_locs* ⟨*proof*⟩
interpretation
  i_set_attribute?: l_set_attribute$_{Core\_DOM}$ *type_wf set_attribute set_attribute_locs*
  ⟨*proof*⟩
declare *l_set_attribute*$_{Core\_DOM}$*_axioms[instances]*

```
lemma set_attribute_is_l_set_attribute [instances]:
  "l_set_attribute type_wf set_attribute set_attribute_locs"
  ⟨proof⟩
```

get_attribute locale $l\_set\_attribute\_get\_attribute_{Core\_DOM}$ =

```
  l_get_attribute_Core_DOM +
  l_set_attribute_Core_DOM
begin

lemma set_attribute_get_attribute:
  "h ⊢ set_attribute ptr k v →_h h' ⟹ h' ⊢ get_attribute ptr k →_r v"
  ⟨proof⟩
end

locale l_set_attribute_get_attribute = l_get_attribute + l_set_attribute +
  assumes set_attribute_get_attribute:
    "h ⊢ set_attribute ptr k v →_h h' ⟹ h' ⊢ get_attribute ptr k →_r v"

interpretation
  i_set_attribute_get_attribute?: l_set_attribute_get_attribute_Core_DOM type_wf
                                  get_attribute get_attribute_locs set_attribute set_attribute_locs
  ⟨proof⟩
declare l_set_attribute_get_attribute_Core_DOM_axioms[instances]

lemma set_attribute_get_attribute_is_l_set_attribute_get_attribute [instances]:
  "l_set_attribute_get_attribute type_wf get_attribute get_attribute_locs set_attribute set_attribute_locs"
  ⟨proof⟩
```

get_child_nodes locale $l\_set\_attribute\_get\_child\_nodes_{Core\_DOM}$ =

```
  l_set_attribute_Core_DOM +
  l_get_child_nodes_Core_DOM
begin
lemma set_attribute_get_child_nodes:
  "∀w ∈ set_attribute_locs ptr. (h ⊢ w →_h h' ⟶ (∀r ∈ get_child_nodes_locs ptr'. r h h'))"
  ⟨proof⟩
end

locale l_set_attribute_get_child_nodes =
  l_set_attribute +
  l_get_child_nodes +
  assumes set_attribute_get_child_nodes:
        "∀w ∈ set_attribute_locs ptr. (h ⊢ w →_h h' ⟶ (∀r ∈ get_child_nodes_locs ptr'. r h h'))"

interpretation
  i_set_attribute_get_child_nodes?: l_set_attribute_get_child_nodes_Core_DOM type_wf
                  set_attribute set_attribute_locs known_ptr get_child_nodes get_child_nodes_locs
  ⟨proof⟩
declare l_set_attribute_get_child_nodes_Core_DOM_axioms[instances]

lemma set_attribute_get_child_nodes_is_l_set_attribute_get_child_nodes [instances]:
  "l_set_attribute_get_child_nodes type_wf set_attribute set_attribute_locs known_ptr
                                   get_child_nodes get_child_nodes_locs"
  ⟨proof⟩
```

get_disconnected_nodes

```
locale l_get_disconnected_nodes_Core_DOM_defs
begin
definition a_get_disconnected_nodes :: "(_) document_ptr
    ⇒ (_, (_) node_ptr list) dom_prog"
  where
    "a_get_disconnected_nodes document_ptr = get_M document_ptr disconnected_nodes"
lemmas get_disconnected_nodes_defs = a_get_disconnected_nodes_def
```

```
definition a_get_disconnected_nodes_locs :: "(_) document_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"
  where
    "a_get_disconnected_nodes_locs document_ptr = {preserved (get_M document_ptr disconnected_nodes)}"
end

locale l_get_disconnected_nodes_defs =
  fixes get_disconnected_nodes :: "(_) document_ptr ⇒ (_, (_) node_ptr list) dom_prog"
  fixes get_disconnected_nodes_locs :: "(_) document_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"

locale l_get_disconnected_nodes$_{Core\_DOM}$ =
  l_type_wf type_wf +
  l_get_disconnected_nodes_defs get_disconnected_nodes get_disconnected_nodes_locs +
  l_get_disconnected_nodes$_{Core\_DOM}$_defs
  for type_wf :: "(_) heap ⇒ bool"
  and get_disconnected_nodes :: "(_) document_ptr ⇒ ((_) heap, exception, (_) node_ptr list) prog"
  and get_disconnected_nodes_locs :: "(_) document_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set" +
  assumes type_wf_impl: "type_wf = DocumentClass.type_wf"
  assumes get_disconnected_nodes_impl: "get_disconnected_nodes = a_get_disconnected_nodes"
  assumes get_disconnected_nodes_locs_impl: "get_disconnected_nodes_locs = a_get_disconnected_nodes_locs"
begin
lemmas
  get_disconnected_nodes_def = get_disconnected_nodes_impl[unfolded a_get_disconnected_nodes_def]
lemmas
  get_disconnected_nodes_locs_def = get_disconnected_nodes_locs_impl[unfolded a_get_disconnected_nodes_locs_def]

lemma get_disconnected_nodes_ok:
  "type_wf h ⟹ document_ptr |∈| document_ptr_kinds h ⟹ h ⊢ ok (get_disconnected_nodes document_ptr)"
  ⟨proof⟩

lemma get_disconnected_nodes_ptr_in_heap:
  "h ⊢ ok (get_disconnected_nodes document_ptr) ⟹ document_ptr |∈| document_ptr_kinds h"
  ⟨proof⟩

lemma get_disconnected_nodes_pure [simp]: "pure (get_disconnected_nodes document_ptr) h"
  ⟨proof⟩

lemma get_disconnected_nodes_reads:
  "reads (get_disconnected_nodes_locs document_ptr) (get_disconnected_nodes document_ptr) h h'"
  ⟨proof⟩
end

locale l_get_disconnected_nodes = l_type_wf + l_get_disconnected_nodes_defs +
  assumes get_disconnected_nodes_reads:
    "reads (get_disconnected_nodes_locs document_ptr) (get_disconnected_nodes document_ptr) h h'"
  assumes get_disconnected_nodes_ok:
    "type_wf h ⟹ document_ptr |∈| document_ptr_kinds h ⟹ h ⊢ ok (get_disconnected_nodes document_ptr)"
  assumes get_disconnected_nodes_ptr_in_heap:
    "h ⊢ ok (get_disconnected_nodes document_ptr) ⟹ document_ptr |∈| document_ptr_kinds h"
  assumes get_disconnected_nodes_pure [simp]:
   "pure (get_disconnected_nodes document_ptr) h"

global_interpretation l_get_disconnected_nodes$_{Core\_DOM}$_defs defines
  get_disconnected_nodes = l_get_disconnected_nodes$_{Core\_DOM}$_defs.a_get_disconnected_nodes and
  get_disconnected_nodes_locs = l_get_disconnected_nodes$_{Core\_DOM}$_defs.a_get_disconnected_nodes_locs ⟨proof⟩
interpretation
  i_get_disconnected_nodes?: l_get_disconnected_nodes$_{Core\_DOM}$ type_wf get_disconnected_nodes
                             get_disconnected_nodes_locs
  ⟨proof⟩
declare l_get_disconnected_nodes$_{Core\_DOM}$_axioms[instances]

lemma get_disconnected_nodes_is_l_get_disconnected_nodes [instances]:
  "l_get_disconnected_nodes type_wf get_disconnected_nodes get_disconnected_nodes_locs"
  ⟨proof⟩

set_child_nodes  locale l_set_child_nodes_get_disconnected_nodes_Core_DOM =
  l_set_child_nodes_Core_DOM +
  CD: l_get_disconnected_nodes_Core_DOM
begin
lemma set_child_nodes_get_disconnected_nodes:
  "∀w ∈ a_set_child_nodes_locs ptr. (h ⊢ w →_h h' ⟶ (∀r ∈ a_get_disconnected_nodes_locs ptr'. r h h'))"
  ⟨proof⟩
end

locale l_set_child_nodes_get_disconnected_nodes = l_set_child_nodes + l_get_disconnected_nodes +
  assumes set_child_nodes_get_disconnected_nodes:
  "∀w ∈ set_child_nodes_locs ptr. (h ⊢ w →_h h' ⟶ (∀r ∈ get_disconnected_nodes_locs ptr'. r h h'))"

interpretation
  i_set_child_nodes_get_disconnected_nodes?: l_set_child_nodes_get_disconnected_nodes_Core_DOM type_wf
                                              known_ptr set_child_nodes set_child_nodes_locs
                                              get_disconnected_nodes get_disconnected_nodes_locs
  ⟨proof⟩
declare l_set_child_nodes_get_disconnected_nodes_Core_DOM_axioms[instances]

lemma set_child_nodes_get_disconnected_nodes_is_l_set_child_nodes_get_disconnected_nodes [instances]:
  "l_set_child_nodes_get_disconnected_nodes type_wf set_child_nodes set_child_nodes_locs
                                             get_disconnected_nodes get_disconnected_nodes_locs"
  ⟨proof⟩

set_attribute  locale l_set_attribute_get_disconnected_nodes_Core_DOM =
  l_set_attribute_Core_DOM +
  l_get_disconnected_nodes_Core_DOM
begin
lemma set_attribute_get_disconnected_nodes:
  "∀w ∈ a_set_attribute_locs ptr. (h ⊢ w →_h h' ⟶ (∀r ∈ a_get_disconnected_nodes_locs ptr'. r h h'))"
  ⟨proof⟩
end

locale l_set_attribute_get_disconnected_nodes = l_set_attribute + l_get_disconnected_nodes +
  assumes set_attribute_get_disconnected_nodes:
  "∀w ∈ set_attribute_locs ptr. (h ⊢ w →_h h' ⟶ (∀r ∈ get_disconnected_nodes_locs ptr'. r h h'))"

interpretation
  i_set_attribute_get_disconnected_nodes?: l_set_attribute_get_disconnected_nodes_Core_DOM type_wf
              set_attribute set_attribute_locs get_disconnected_nodes get_disconnected_nodes_locs
  ⟨proof⟩
declare l_set_attribute_get_disconnected_nodes_Core_DOM_axioms[instances]

lemma set_attribute_get_disconnected_nodes_is_l_set_attribute_get_disconnected_nodes [instances]:
  "l_set_attribute_get_disconnected_nodes type_wf set_attribute set_attribute_locs
                                           get_disconnected_nodes get_disconnected_nodes_locs"
  ⟨proof⟩

new_element  locale l_new_element_get_disconnected_nodes_Core_DOM =
  l_get_disconnected_nodes_Core_DOM type_wf get_disconnected_nodes get_disconnected_nodes_locs
  for type_wf :: "(_) heap ⇒ bool"
  and get_disconnected_nodes :: "(_) document_ptr ⇒ ((_) heap, exception, (_) node_ptr list) prog"
  and get_disconnected_nodes_locs :: "(_) document_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"
begin
lemma get_disconnected_nodes_new_element:
   "h ⊢ new_element →_r new_element_ptr ⟹ h ⊢ new_element →_h h'
       ⟹ r ∈ get_disconnected_nodes_locs ptr' ⟹ r h h'"
  ⟨proof⟩
end
```

```
locale l_new_element_get_disconnected_nodes = l_get_disconnected_nodes_defs +
  assumes get_disconnected_nodes_new_element:
   "h ⊢ new_element →r new_element_ptr ⟹ h ⊢ new_element →h h'
        ⟹ r ∈ get_disconnected_nodes_locs ptr' ⟹ r h h'"

interpretation i_new_element_get_disconnected_nodes?:
  l_new_element_get_disconnected_nodesCore_DOM type_wf get_disconnected_nodes
  get_disconnected_nodes_locs
  ⟨proof⟩
declare l_new_element_get_disconnected_nodesCore_DOM_axioms[instances]

lemma new_element_get_disconnected_nodes_is_l_new_element_get_disconnected_nodes [instances]:
  "l_new_element_get_disconnected_nodes get_disconnected_nodes_locs"
  ⟨proof⟩
```

new_character_data

```
locale l_new_character_data_get_disconnected_nodesCore_DOM =
  l_get_disconnected_nodesCore_DOM type_wf get_disconnected_nodes get_disconnected_nodes_locs
  for type_wf :: "(_) heap ⇒ bool"
  and get_disconnected_nodes :: "(_) document_ptr ⇒ ((_) heap, exception, (_) node_ptr list) prog"
  and get_disconnected_nodes_locs :: "(_) document_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"
begin
lemma get_disconnected_nodes_new_character_data:
  "h ⊢ new_character_data →r new_character_data_ptr ⟹ h ⊢ new_character_data →h h'
       ⟹ r ∈ get_disconnected_nodes_locs ptr' ⟹ r h h'"
  ⟨proof⟩
end

locale l_new_character_data_get_disconnected_nodes = l_get_disconnected_nodes_defs +
  assumes get_disconnected_nodes_new_character_data:
  "h ⊢ new_character_data →r new_character_data_ptr ⟹ h ⊢ new_character_data →h h'
       ⟹ r ∈ get_disconnected_nodes_locs ptr' ⟹ r h h'"

interpretation i_new_character_data_get_disconnected_nodes?:
  l_new_character_data_get_disconnected_nodesCore_DOM type_wf get_disconnected_nodes
                                                      get_disconnected_nodes_locs
  ⟨proof⟩
declare l_new_character_data_get_disconnected_nodesCore_DOM_axioms[instances]

lemma new_character_data_get_disconnected_nodes_is_l_new_character_data_get_disconnected_nodes [instances]:
  "l_new_character_data_get_disconnected_nodes get_disconnected_nodes_locs"
  ⟨proof⟩
```

new_document

```
locale l_new_document_get_disconnected_nodesCore_DOM =
  l_get_disconnected_nodesCore_DOM type_wf get_disconnected_nodes get_disconnected_nodes_locs
  for type_wf :: "(_) heap ⇒ bool"
  and get_disconnected_nodes :: "(_) document_ptr ⇒ ((_) heap, exception, (_) node_ptr list) prog"
  and get_disconnected_nodes_locs :: "(_) document_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"
begin
lemma get_disconnected_nodes_new_document_different_pointers:
  "new_document_ptr ≠ ptr' ⟹ h ⊢ new_document →r new_document_ptr ⟹ h ⊢ new_document →h h'
    ⟹ r ∈ get_disconnected_nodes_locs ptr' ⟹ r h h'"
  ⟨proof⟩

lemma new_document_no_disconnected_nodes:
  "h ⊢ new_document →r new_document_ptr ⟹ h ⊢ new_document →h h'
    ⟹ h' ⊢ get_disconnected_nodes new_document_ptr →r []"
  ⟨proof⟩

end

interpretation i_new_document_get_disconnected_nodes?:
  l_new_document_get_disconnected_nodesCore_DOM type_wf get_disconnected_nodes get_disconnected_nodes_locs
  ⟨proof⟩
declare l_new_document_get_disconnected_nodes$_{Core\_DOM}$_axioms[instances]

locale l_new_document_get_disconnected_nodes = l_get_disconnected_nodes_defs +
  assumes get_disconnected_nodes_new_document_different_pointers:
  "new_document_ptr ≠ ptr' ⟹ h ⊢ new_document →r new_document_ptr ⟹ h ⊢ new_document →h h'
    ⟹ r ∈ get_disconnected_nodes_locs ptr' ⟹ r h h'"
  assumes new_document_no_disconnected_nodes:
  "h ⊢ new_document →r new_document_ptr ⟹ h ⊢ new_document →h h'
    ⟹ h' ⊢ get_disconnected_nodes new_document_ptr →r []"

lemma new_document_get_disconnected_nodes_is_l_new_document_get_disconnected_nodes [instances]:
  "l_new_document_get_disconnected_nodes get_disconnected_nodes get_disconnected_nodes_locs"
  ⟨proof⟩
```

set_disconnected_nodes

```
locale l_set_disconnected_nodes$_{Core\_DOM}$_defs
begin

definition a_set_disconnected_nodes :: "(_) document_ptr ⇒ (_) node_ptr list ⇒ (_, unit) dom_prog"
  where
    "a_set_disconnected_nodes document_ptr disc_nodes =
put_M document_ptr disconnected_nodes_update disc_nodes"
lemmas set_disconnected_nodes_defs = a_set_disconnected_nodes_def

definition a_set_disconnected_nodes_locs :: "(_) document_ptr ⇒ (_, unit) dom_prog set"
  where
    "a_set_disconnected_nodes_locs document_ptr ≡ all_args (put_M document_ptr disconnected_nodes_update)"
end

locale l_set_disconnected_nodes_defs =
  fixes set_disconnected_nodes :: "(_) document_ptr ⇒ (_) node_ptr list ⇒ (_, unit) dom_prog"
  fixes set_disconnected_nodes_locs :: "(_) document_ptr ⇒ (_, unit) dom_prog set"

locale l_set_disconnected_nodes$_{Core\_DOM}$ =
  l_type_wf type_wf +
  l_set_disconnected_nodes_defs set_disconnected_nodes set_disconnected_nodes_locs +
  l_set_disconnected_nodes$_{Core\_DOM}$_defs
  for type_wf :: "(_) heap ⇒ bool"
  and set_disconnected_nodes :: "(_) document_ptr ⇒ (_) node_ptr list ⇒ (_, unit) dom_prog"
  and set_disconnected_nodes_locs :: "(_) document_ptr ⇒ (_, unit) dom_prog set" +
  assumes type_wf_impl: "type_wf = DocumentClass.type_wf"
  assumes set_disconnected_nodes_impl: "set_disconnected_nodes = a_set_disconnected_nodes"
  assumes set_disconnected_nodes_locs_impl: "set_disconnected_nodes_locs = a_set_disconnected_nodes_locs"
begin
lemmas set_disconnected_nodes_def = set_disconnected_nodes_impl[unfolded a_set_disconnected_nodes_def]
lemmas set_disconnected_nodes_locs_def =
  set_disconnected_nodes_locs_impl[unfolded a_set_disconnected_nodes_locs_def]
lemma set_disconnected_nodes_ok:
  "type_wf h ⟹ document_ptr |∈| document_ptr_kinds h ⟹
h ⊢ ok (set_disconnected_nodes document_ptr node_ptrs)"
  ⟨proof⟩

lemma set_disconnected_nodes_ptr_in_heap:
  "h ⊢ ok (set_disconnected_nodes document_ptr disc_nodes) ⟹ document_ptr |∈| document_ptr_kinds h"
  ⟨proof⟩

lemma set_disconnected_nodes_writes:
  "writes (set_disconnected_nodes_locs document_ptr) (set_disconnected_nodes document_ptr disc_nodes) h
h'"
  ⟨proof⟩
```

```
lemma set_disconnected_nodes_pointers_preserved:
  assumes "w ∈ set_disconnected_nodes_locs object_ptr"
  assumes "h ⊢ w →_h h'"
  shows "object_ptr_kinds h = object_ptr_kinds h'"
  ⟨proof⟩

lemma set_disconnected_nodes_typess_preserved:
  assumes "w ∈ set_disconnected_nodes_locs object_ptr"
  assumes "h ⊢ w →_h h'"
  shows "type_wf h = type_wf h'"
  ⟨proof⟩
end

locale l_set_disconnected_nodes = l_type_wf + l_set_disconnected_nodes_defs +
  assumes set_disconnected_nodes_writes:
    "writes (set_disconnected_nodes_locs document_ptr)
(set_disconnected_nodes document_ptr disc_nodes) h h'"
  assumes set_disconnected_nodes_ok:
    "type_wf h ⟹ document_ptr |∈| document_ptr_kinds h ⟹
h ⊢ ok (set_disconnected_nodes document_ptr disc_noded)"
  assumes set_disconnected_nodes_ptr_in_heap:
    "h ⊢ ok (set_disconnected_nodes document_ptr disc_noded) ⟹
document_ptr |∈| document_ptr_kinds h"
  assumes set_disconnected_nodes_pointers_preserved:
    "w ∈ set_disconnected_nodes_locs document_ptr ⟹ h ⊢ w →_h h' ⟹
object_ptr_kinds h = object_ptr_kinds h'"
 assumes set_disconnected_nodes_types_preserved:
   "w ∈ set_disconnected_nodes_locs document_ptr ⟹ h ⊢ w →_h h' ⟹ type_wf h = type_wf h'"

global_interpretation l_set_disconnected_nodes_{Core_DOM}_defs defines
  set_disconnected_nodes = l_set_disconnected_nodes_{Core_DOM}_defs.a_set_disconnected_nodes and
  set_disconnected_nodes_locs = l_set_disconnected_nodes_{Core_DOM}_defs.a_set_disconnected_nodes_locs ⟨proof⟩
interpretation
  i_set_disconnected_nodes?: l_set_disconnected_nodes_{Core_DOM} type_wf set_disconnected_nodes
                             set_disconnected_nodes_locs
  ⟨proof⟩
declare l_set_disconnected_nodes_{Core_DOM}_axioms[instances]

lemma set_disconnected_nodes_is_l_set_disconnected_nodes [instances]:
  "l_set_disconnected_nodes type_wf set_disconnected_nodes set_disconnected_nodes_locs"
  ⟨proof⟩
```

get_disconnected_nodes

```
locale l_set_disconnected_nodes_get_disconnected_nodes_{Core_DOM} = l_get_disconnected_nod

                                                  + l_set_disconnected_nodes_{Core_DOM}
begin
lemma set_disconnected_nodes_get_disconnected_nodes:
  assumes "h ⊢ a_set_disconnected_nodes document_ptr disc_nodes →_h h'"
  shows "h' ⊢ a_get_disconnected_nodes document_ptr →_r disc_nodes"
  ⟨proof⟩

lemma set_disconnected_nodes_get_disconnected_nodes_different_pointers:
  assumes "ptr ≠ ptr'"
  assumes "w ∈ a_set_disconnected_nodes_locs ptr"
  assumes "h ⊢ w →_h h'"
  assumes "r ∈ a_get_disconnected_nodes_locs ptr'"
  shows "r h h'"
  ⟨proof⟩
end

locale l_set_disconnected_nodes_get_disconnected_nodes = l_get_disconnected_nodes
                                                       + l_set_disconnected_nodes +
  assumes set_disconnected_nodes_get_disconnected_nodes:
```

```
  "h ⊢ set_disconnected_nodes document_ptr disc_nodes →h h'
     ⟹ h' ⊢ get_disconnected_nodes document_ptr →r disc_nodes"
  assumes set_disconnected_nodes_get_disconnected_nodes_different_pointers:
  "ptr ≠ ptr' ⟹ w ∈ set_disconnected_nodes_locs ptr ⟹ h ⊢ w →h h'
    ⟹ r ∈ get_disconnected_nodes_locs ptr' ⟹ r h h'"

interpretation i_set_disconnected_nodes_get_disconnected_nodes?:
  l_set_disconnected_nodes_get_disconnected_nodesCore_DOM type_wf get_disconnected_nodes
                    get_disconnected_nodes_locs set_disconnected_nodes set_disconnected_nodes_locs
  ⟨proof⟩
declare l_set_disconnected_nodes_get_disconnected_nodesCore_DOM_axioms[instances]

lemma set_disconnected_nodes_get_disconnected_nodes_is_l_set_disconnected_nodes_get_disconnected_nodes
  [instances]:
  "l_set_disconnected_nodes_get_disconnected_nodes  type_wf get_disconnected_nodes get_disconnected_nodes_locs

                                                    set_disconnected_nodes set_disconnected_nodes_locs"
  ⟨proof⟩
```

get_child_nodes

```
locale l_set_disconnected_nodes_get_child_nodesCore_DOM =
  l_set_disconnected_nodesCore_DOM +
  l_get_child_nodesCore_DOM
begin
lemma set_disconnected_nodes_get_child_nodes:
  "∀w ∈ set_disconnected_nodes_locs ptr. (h ⊢ w →h h' ⟶ (∀r ∈ get_child_nodes_locs ptr'. r h h'))"
  ⟨proof⟩
end

locale l_set_disconnected_nodes_get_child_nodes = l_set_disconnected_nodes_defs + l_get_child_nodes_defs
+
  assumes set_disconnected_nodes_get_child_nodes [simp]:
    "∀w ∈ set_disconnected_nodes_locs ptr. (h ⊢ w →h h' ⟶ (∀r ∈ get_child_nodes_locs ptr'. r h h'))"

interpretation
  i_set_disconnected_nodes_get_child_nodes?: l_set_disconnected_nodes_get_child_nodesCore_DOM
                                             type_wf
                                             set_disconnected_nodes set_disconnected_nodes_locs
                                             known_ptr get_child_nodes get_child_nodes_locs
  ⟨proof⟩
declare l_set_disconnected_nodes_get_child_nodesCore_DOM_axioms[instances]

lemma set_disconnected_nodes_get_child_nodes_is_l_set_disconnected_nodes_get_child_nodes [instances]:
  "l_set_disconnected_nodes_get_child_nodes set_disconnected_nodes_locs get_child_nodes_locs"
  ⟨proof⟩
```

get_tag_name

```
locale l_get_tag_nameCore_DOM_defs
begin
definition a_get_tag_name :: "(_) element_ptr ⇒ (_, tag_name) dom_prog"
  where
    "a_get_tag_name element_ptr = get_M element_ptr tag_name"

definition a_get_tag_name_locs :: "(_) element_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"
  where
    "a_get_tag_name_locs element_ptr ≡ {preserved (get_M element_ptr tag_name)}"
end

locale l_get_tag_name_defs =
  fixes get_tag_name :: "(_) element_ptr ⇒ (_, tag_name) dom_prog"
  fixes get_tag_name_locs :: "(_) element_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"

locale l_get_tag_nameCore_DOM =
  l_type_wf type_wf +
  l_get_tag_name_defs get_tag_name get_tag_name_locs +
  l_get_tag_name_{Core_DOM}_defs
  for type_wf :: "(_) heap ⇒ bool"
    and get_tag_name :: "(_) element_ptr ⇒ (_, tag_name) dom_prog"
  and get_tag_name_locs :: "(_) element_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set" +
  assumes type_wf_impl: "type_wf = DocumentClass.type_wf"
  assumes get_tag_name_impl: "get_tag_name = a_get_tag_name"
  assumes get_tag_name_locs_impl: "get_tag_name_locs = a_get_tag_name_locs"
begin
lemmas get_tag_name_def = get_tag_name_impl[unfolded a_get_tag_name_def]
lemmas get_tag_name_locs_def = get_tag_name_locs_impl[unfolded a_get_tag_name_locs_def]



lemma get_tag_name_ok:
  "type_wf h ⟹ element_ptr |∈| element_ptr_kinds h ⟹ h ⊢ ok (get_tag_name element_ptr)"
  ⟨proof⟩

lemma get_tag_name_pure [simp]: "pure (get_tag_name element_ptr) h"
  ⟨proof⟩

lemma get_tag_name_ptr_in_heap [simp]:
  assumes "h ⊢ get_tag_name element_ptr →_r children"
  shows "element_ptr |∈| element_ptr_kinds h"
  ⟨proof⟩

lemma get_tag_name_reads: "reads (get_tag_name_locs element_ptr) (get_tag_name element_ptr) h h'"
  ⟨proof⟩
end

locale l_get_tag_name = l_type_wf + l_get_tag_name_defs +
  assumes get_tag_name_reads:
    "reads (get_tag_name_locs element_ptr) (get_tag_name element_ptr) h h'"
  assumes get_tag_name_ok:
    "type_wf h ⟹ element_ptr |∈| element_ptr_kinds h ⟹ h ⊢ ok (get_tag_name element_ptr)"
  assumes get_tag_name_ptr_in_heap:
    "h ⊢ ok (get_tag_name element_ptr) ⟹ element_ptr |∈| element_ptr_kinds h"
  assumes get_tag_name_pure [simp]:
    "pure (get_tag_name element_ptr) h"


global_interpretation l_get_tag_name_{Core_DOM}_defs defines
  get_tag_name = l_get_tag_name_{Core_DOM}_defs.a_get_tag_name and
  get_tag_name_locs = l_get_tag_name_{Core_DOM}_defs.a_get_tag_name_locs ⟨proof⟩

interpretation
  i_get_tag_name?: l_get_tag_name_{Core_DOM} type_wf get_tag_name get_tag_name_locs
  ⟨proof⟩
declare l_get_tag_name_{Core_DOM}_axioms[instances]

lemma get_tag_name_is_l_get_tag_name [instances]:
  "l_get_tag_name type_wf get_tag_name get_tag_name_locs"
  ⟨proof⟩
```

set_disconnected_nodes

```
locale l_set_disconnected_nodes_get_tag_name_{Core_DOM} =
  l_set_disconnected_nodes_{Core_DOM} +
  l_get_tag_name_{Core_DOM}
begin
lemma set_disconnected_nodes_get_tag_name:
  "∀w ∈ a_set_disconnected_nodes_locs ptr. (h ⊢ w →_h h' ⟶ (∀r ∈ a_get_tag_name_locs ptr'. r h h'))"
  ⟨proof⟩
end
```

```
locale l_set_disconnected_nodes_get_tag_name = l_set_disconnected_nodes + l_get_tag_name +
  assumes set_disconnected_nodes_get_tag_name:
  "∀w ∈ set_disconnected_nodes_locs ptr. (h ⊢ w →h h' ⟶ (∀r ∈ get_tag_name_locs ptr'. r h h'))"

interpretation
  i_set_disconnected_nodes_get_tag_name?: l_set_disconnected_nodes_get_tag_nameCore_DOM type_wf
                                          set_disconnected_nodes set_disconnected_nodes_locs
                                          get_tag_name get_tag_name_locs
  ⟨proof⟩
declare l_set_disconnected_nodes_get_tag_nameCore_DOM_axioms[instances]

lemma set_disconnected_nodes_get_tag_name_is_l_set_disconnected_nodes_get_tag_name [instances]:
  "l_set_disconnected_nodes_get_tag_name type_wf set_disconnected_nodes set_disconnected_nodes_locs
                                         get_tag_name get_tag_name_locs"
  ⟨proof⟩
```

set_child_nodes

```
locale l_set_child_nodes_get_tag_nameCore_DOM =
  l_set_child_nodesCore_DOM +
  l_get_tag_nameCore_DOM
begin
lemma set_child_nodes_get_tag_name:
  "∀w ∈ set_child_nodes_locs ptr. (h ⊢ w →h h' ⟶ (∀r ∈ get_tag_name_locs ptr'. r h h'))"
  ⟨proof⟩
end

locale l_set_child_nodes_get_tag_name = l_set_child_nodes + l_get_tag_name +
  assumes set_child_nodes_get_tag_name:
    "∀w ∈ set_child_nodes_locs ptr. (h ⊢ w →h h' ⟶ (∀r ∈ get_tag_name_locs ptr'. r h h'))"

interpretation
  i_set_child_nodes_get_tag_name?: l_set_child_nodes_get_tag_nameCore_DOM type_wf known_ptr
                                   set_child_nodes set_child_nodes_locs get_tag_name get_tag_name_locs
  ⟨proof⟩
declare l_set_child_nodes_get_tag_nameCore_DOM_axioms[instances]

lemma set_child_nodes_get_tag_name_is_l_set_child_nodes_get_tag_name [instances]:
  "l_set_child_nodes_get_tag_name type_wf set_child_nodes set_child_nodes_locs get_tag_name get_tag_name_locs"
  ⟨proof⟩
```

set_tag_type

```
locale l_set_tag_nameCore_DOM_defs
begin

definition a_set_tag_name :: "(_) element_ptr ⇒ tag_name ⇒ (_, unit) dom_prog"
  where
    "a_set_tag_name ptr tag = do {
      m ← get_M ptr attrs;
      put_M ptr tag_name_update tag
    }"
lemmas set_tag_name_defs = a_set_tag_name_def

definition a_set_tag_name_locs :: "(_) element_ptr ⇒ (_, unit) dom_prog set"
  where
    "a_set_tag_name_locs element_ptr ≡ all_args (put_M element_ptr tag_name_update)"
end

locale l_set_tag_name_defs =
  fixes set_tag_name :: "(_) element_ptr ⇒ tag_name ⇒ (_, unit) dom_prog"
  fixes set_tag_name_locs :: "(_) element_ptr ⇒ (_, unit) dom_prog set"

locale l_set_tag_nameCore_DOM =
  l_type_wf type_wf +
  l_set_tag_name_defs set_tag_name set_tag_name_locs +
  l_set_tag_name$_{Core\_DOM}$_defs
  for type_wf :: "(_) heap ⇒ bool"
    and set_tag_name :: "(_) element_ptr ⇒ char list ⇒ (_, unit) dom_prog"
    and set_tag_name_locs :: "(_) element_ptr ⇒ (_, unit) dom_prog set" +
  assumes type_wf_impl: "type_wf = DocumentClass.type_wf"
  assumes set_tag_name_impl: "set_tag_name = a_set_tag_name"
  assumes set_tag_name_locs_impl: "set_tag_name_locs = a_set_tag_name_locs"
begin

lemma set_tag_name_ok:
  "type_wf h ⟹ element_ptr |∈| element_ptr_kinds h ⟹ h ⊢ ok (set_tag_name element_ptr tag)"
  ⟨proof⟩

lemma set_tag_name_writes:
  "writes (set_tag_name_locs element_ptr) (set_tag_name element_ptr tag) h h'"
  ⟨proof⟩

lemma set_tag_name_pointers_preserved:
  assumes "w ∈ set_tag_name_locs element_ptr"
  assumes "h ⊢ w →$_h$ h'"
  shows "object_ptr_kinds h = object_ptr_kinds h'"
  ⟨proof⟩

lemma set_tag_name_typess_preserved:
  assumes "w ∈ set_tag_name_locs element_ptr"
  assumes "h ⊢ w →$_h$ h'"
  shows "type_wf h = type_wf h'"
  ⟨proof⟩
end

locale l_set_tag_name = l_type_wf + l_set_tag_name_defs +
  assumes set_tag_name_writes:
    "writes (set_tag_name_locs element_ptr) (set_tag_name element_ptr tag) h h'"
  assumes set_tag_name_ok:
    "type_wf h ⟹ element_ptr |∈| element_ptr_kinds h ⟹ h ⊢ ok (set_tag_name element_ptr tag)"
  assumes set_tag_name_pointers_preserved:
    "w ∈ set_tag_name_locs element_ptr ⟹ h ⊢ w →$_h$ h' ⟹ object_ptr_kinds h = object_ptr_kinds h'"
  assumes set_tag_name_types_preserved:
    "w ∈ set_tag_name_locs element_ptr ⟹ h ⊢ w →$_h$ h' ⟹ type_wf h = type_wf h'"


global_interpretation l_set_tag_name$_{Core\_DOM}$_defs defines
  set_tag_name = l_set_tag_name$_{Core\_DOM}$_defs.a_set_tag_name and
  set_tag_name_locs = l_set_tag_name$_{Core\_DOM}$_defs.a_set_tag_name_locs ⟨proof⟩
interpretation
  i_set_tag_name?: l_set_tag_name$_{Core\_DOM}$ type_wf set_tag_name set_tag_name_locs
  ⟨proof⟩
declare l_set_tag_name$_{Core\_DOM}$_axioms[instances]

lemma set_tag_name_is_l_set_tag_name [instances]:
  "l_set_tag_name type_wf set_tag_name set_tag_name_locs"
  ⟨proof⟩
```

get_child_nodes

```
locale l_set_tag_name_get_child_nodes$_{Core\_DOM}$ =
  l_set_tag_name$_{Core\_DOM}$ +
  l_get_child_nodes$_{Core\_DOM}$
begin
lemma set_tag_name_get_child_nodes:
  "∀w ∈ set_tag_name_locs ptr. (h ⊢ w →$_h$ h' ⟶ (∀r ∈ get_child_nodes_locs ptr'. r h h'))"
  ⟨proof⟩
end
```

locale *l_set_tag_name_get_child_nodes = l_set_tag_name + l_get_child_nodes +*
  assumes *set_tag_name_get_child_nodes:*
    "∀w ∈ set_tag_name_locs ptr. (h ⊢ w $\rightarrow_h$ h' ⟶ (∀r ∈ get_child_nodes_locs ptr'. r h h'))"

interpretation
  i_set_tag_name_get_child_nodes?: l_set_tag_name_get_child_nodes$_{Core\_DOM}$ *type_wf*
  set_tag_name set_tag_name_locs known_ptr
                                    get_child_nodes get_child_nodes_locs
  ⟨*proof*⟩
declare *l_set_tag_name_get_child_nodes*$_{Core\_DOM}$*_axioms[instances]*

lemma *set_tag_name_get_child_nodes_is_l_set_tag_name_get_child_nodes [instances]:*
  "l_set_tag_name_get_child_nodes type_wf set_tag_name set_tag_name_locs known_ptr get_child_nodes
                                    get_child_nodes_locs"
  ⟨*proof*⟩

get_disconnected_nodes **locale** *l_set_tag_name_get_disconnected_nodes*$_{Core\_DOM}$ *=*
  l_set_tag_name$_{Core\_DOM}$ *+*
  l_get_disconnected_nodes$_{Core\_DOM}$
begin
lemma *set_tag_name_get_disconnected_nodes:*
  "∀w ∈ set_tag_name_locs ptr. (h ⊢ w $\rightarrow_h$ h' ⟶ (∀r ∈ get_disconnected_nodes_locs ptr'. r h h'))"
  ⟨*proof*⟩
end

locale *l_set_tag_name_get_disconnected_nodes = l_set_tag_name + l_get_disconnected_nodes +*
  assumes *set_tag_name_get_disconnected_nodes:*
    "∀w ∈ set_tag_name_locs ptr. (h ⊢ w $\rightarrow_h$ h' ⟶ (∀r ∈ get_disconnected_nodes_locs ptr'. r h h'))"

interpretation
  i_set_tag_name_get_disconnected_nodes?: l_set_tag_name_get_disconnected_nodes$_{Core\_DOM}$ *type_wf*
  set_tag_name set_tag_name_locs get_disconnected_nodes
                                    get_disconnected_nodes_locs
  ⟨*proof*⟩
declare *l_set_tag_name_get_disconnected_nodes*$_{Core\_DOM}$*_axioms[instances]*

lemma *set_tag_name_get_disconnected_nodes_is_l_set_tag_name_get_disconnected_nodes [instances]:*
  "l_set_tag_name_get_disconnected_nodes type_wf set_tag_name set_tag_name_locs get_disconnected_nodes
                                    get_disconnected_nodes_locs"
  ⟨*proof*⟩

get_tag_type **locale** *l_set_tag_name_get_tag_name*$_{Core\_DOM}$ *= l_get_tag_name*$_{Core\_DOM}$
  + l_set_tag_name$_{Core\_DOM}$
begin
lemma *set_tag_name_get_tag_name:*
  assumes *"h ⊢ a_set_tag_name element_ptr tag $\rightarrow_h$ h'"*
  shows *"h' ⊢ a_get_tag_name element_ptr $\rightarrow_r$ tag"*
  ⟨*proof*⟩

lemma *set_tag_name_get_tag_name_different_pointers:*
  assumes *"ptr ≠ ptr'"*
  assumes *"w ∈ a_set_tag_name_locs ptr"*
  assumes *"h ⊢ w $\rightarrow_h$ h'"*
  assumes *"r ∈ a_get_tag_name_locs ptr'"*
  shows *"r h h'"*
  ⟨*proof*⟩
end

locale *l_set_tag_name_get_tag_name = l_get_tag_name + l_set_tag_name +*
  assumes *set_tag_name_get_tag_name:*
    "h ⊢ set_tag_name element_ptr tag $\rightarrow_h$ h'
      ⟹ h' ⊢ get_tag_name element_ptr $\rightarrow_r$ tag"

```
  assumes set_tag_name_get_tag_name_different_pointers:
    "ptr ≠ ptr' ⟹ w ∈ set_tag_name_locs ptr ⟹ h ⊢ w →h h'
    ⟹ r ∈ get_tag_name_locs ptr' ⟹ r h h'"

interpretation i_set_tag_name_get_tag_name?:
  l_set_tag_name_get_tag_name_Core_DOM type_wf get_tag_name
  get_tag_name_locs set_tag_name set_tag_name_locs
  ⟨proof⟩
declare l_set_tag_name_get_tag_name_Core_DOM_axioms[instances]

lemma set_tag_name_get_tag_name_is_l_set_tag_name_get_tag_name [instances]:
  "l_set_tag_name_get_tag_name  type_wf get_tag_name get_tag_name_locs
                                                  set_tag_name set_tag_name_locs"
  ⟨proof⟩
```

set_val

```
locale l_set_val_Core_DOM_defs
begin

definition a_set_val :: "(_) character_data_ptr ⇒ DOMString ⇒ (_, unit) dom_prog"
  where
    "a_set_val ptr v = do {
      m ← get_M ptr val;
      put_M ptr val_update v
    }"
lemmas set_val_defs = a_set_val_def

definition a_set_val_locs :: "(_) character_data_ptr ⇒ (_, unit) dom_prog set"
  where
    "a_set_val_locs character_data_ptr ≡ all_args (put_M character_data_ptr val_update)"
end

locale l_set_val_defs =
  fixes set_val :: "(_) character_data_ptr ⇒ DOMString ⇒ (_, unit) dom_prog"
  fixes set_val_locs :: "(_) character_data_ptr ⇒ (_, unit) dom_prog set"

locale l_set_val_Core_DOM =
  l_type_wf type_wf +
  l_set_val_defs set_val set_val_locs +
  l_set_val_Core_DOM_defs
  for type_wf :: "(_) heap ⇒ bool"
  and set_val :: "(_) character_data_ptr ⇒ char list ⇒ (_, unit) dom_prog"
  and set_val_locs :: "(_) character_data_ptr ⇒ (_, unit) dom_prog set"  +
  assumes type_wf_impl: "type_wf = DocumentClass.type_wf"
  assumes set_val_impl: "set_val = a_set_val"
  assumes set_val_locs_impl: "set_val_locs = a_set_val_locs"
begin

lemma set_val_ok:
  "type_wf h ⟹ character_data_ptr |∈| character_data_ptr_kinds h ⟹ h ⊢ ok (set_val character_data_ptr
tag)"
  ⟨proof⟩

lemma set_val_writes: "writes (set_val_locs character_data_ptr) (set_val character_data_ptr tag) h h'"
  ⟨proof⟩

lemma set_val_pointers_preserved:
  assumes "w ∈ set_val_locs character_data_ptr"
  assumes "h ⊢ w →h h'"
  shows "object_ptr_kinds h = object_ptr_kinds h'"
  ⟨proof⟩
```

lemma *set_val_typess_preserved:*
  assumes *"w ∈ set_val_locs character_data_ptr"*
  assumes *"h ⊢ w →$_h$ h'"*
  shows *"type_wf h = type_wf h'"*
  ⟨*proof*⟩
end

locale *l_set_val = l_type_wf + l_set_val_defs +*
  assumes *set_val_writes:*
    "writes (set_val_locs character_data_ptr) (set_val character_data_ptr tag) h h'"
  assumes *set_val_ok:*
    "type_wf h ⟹ character_data_ptr |∈| character_data_ptr_kinds h ⟹ h ⊢ ok (set_val character_data_ptr tag)"
  assumes *set_val_pointers_preserved:*
    "w ∈ set_val_locs character_data_ptr ⟹ h ⊢ w →$_h$ h' ⟹ object_ptr_kinds h = object_ptr_kinds h'"
  assumes *set_val_types_preserved:*
    "w ∈ set_val_locs character_data_ptr ⟹ h ⊢ w →$_h$ h' ⟹ type_wf h = type_wf h'"

global_interpretation *l_set_val$_{Core_DOM}$_defs* **defines**
  set_val = l_set_val$_{Core_DOM}$_defs.a_set_val **and**
  set_val_locs = l_set_val$_{Core_DOM}$_defs.a_set_val_locs ⟨*proof*⟩
interpretation
  i_set_val?: l_set_val$_{Core\_DOM}$ type_wf set_val set_val_locs
  ⟨*proof*⟩
declare *l_set_val$_{Core_DOM}$_axioms[instances]*

lemma *set_val_is_l_set_val [instances]: "l_set_val type_wf set_val set_val_locs"*
  ⟨*proof*⟩

get_child_nodes **locale** *l_set_val_get_child_nodes$_{Core\_DOM}$ =*
  l_set_val$_{Core\_DOM}$ +
  l_get_child_nodes$_{Core\_DOM}$
begin
lemma *set_val_get_child_nodes:*
  "∀w ∈ set_val_locs ptr. (h ⊢ w →$_h$ h' ⟶ (∀r ∈ get_child_nodes_locs ptr'. r h h'))"
  ⟨*proof*⟩
end

locale *l_set_val_get_child_nodes = l_set_val + l_get_child_nodes +*
  assumes *set_val_get_child_nodes:*
    "∀w ∈ set_val_locs ptr. (h ⊢ w →$_h$ h' ⟶ (∀r ∈ get_child_nodes_locs ptr'. r h h'))"

interpretation
  i_set_val_get_child_nodes?: l_set_val_get_child_nodes$_{Core\_DOM}$ type_wf set_val set_val_locs known_ptr

                    get_child_nodes get_child_nodes_locs
  ⟨*proof*⟩
declare *l_set_val_get_child_nodes$_{Core_DOM}$_axioms[instances]*

lemma *set_val_get_child_nodes_is_l_set_val_get_child_nodes [instances]:*
  "l_set_val_get_child_nodes type_wf set_val set_val_locs known_ptr get_child_nodes get_child_nodes_locs"
  ⟨*proof*⟩

get_disconnected_nodes **locale** *l_set_val_get_disconnected_nodes$_{Core\_DOM}$ =*
  l_set_val$_{Core\_DOM}$ +
  l_get_disconnected_nodes$_{Core\_DOM}$
begin
lemma *set_val_get_disconnected_nodes:*
  "∀w ∈ set_val_locs ptr. (h ⊢ w →$_h$ h' ⟶ (∀r ∈ get_disconnected_nodes_locs ptr'. r h h'))"
  ⟨*proof*⟩
end

```
locale l_set_val_get_disconnected_nodes = l_set_val + l_get_disconnected_nodes +
  assumes set_val_get_disconnected_nodes:
   "∀w ∈ set_val_locs ptr. (h ⊢ w →_h h' ⟶ (∀r ∈ get_disconnected_nodes_locs ptr'. r h h'))"

interpretation
  i_set_val_get_disconnected_nodes?: l_set_val_get_disconnected_nodes_Core_DOM type_wf set_val
                                      set_val_locs get_disconnected_nodes get_disconnected_nodes_locs
  ⟨proof⟩
declare l_set_val_get_disconnected_nodes_Core_DOM_axioms[instances]

lemma set_val_get_disconnected_nodes_is_l_set_val_get_disconnected_nodes [instances]:
  "l_set_val_get_disconnected_nodes type_wf set_val set_val_locs get_disconnected_nodes
get_disconnected_nodes_locs"
  ⟨proof⟩
```

get_parent

```
locale l_get_parent_Core_DOM_defs =
  l_get_child_nodes_defs get_child_nodes get_child_nodes_locs
  for get_child_nodes :: "(_::linorder) object_ptr ⇒ (_, (_) node_ptr list) dom_prog"
  and get_child_nodes_locs :: "(_) object_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"
begin
definition a_get_parent :: "(_) node_ptr ⇒ (_, (_::linorder) object_ptr option) dom_prog"
  where
    "a_get_parent node_ptr = do {
      check_in_heap (cast node_ptr);
      parent_ptrs ← object_ptr_kinds_M ≫= filter_M (λptr. do {
        children ← get_child_nodes ptr;
        return (node_ptr ∈ set children)
      });
      (if parent_ptrs = []
        then return None
        else return (Some (hd parent_ptrs)))
    }"

definition
  "a_get_parent_locs ≡ (⋃ptr. get_child_nodes_locs ptr ∪ {preserved (get_M_Object ptr RObject.nothing)})"
end

locale l_get_parent_defs =
  fixes get_parent :: "(_) node_ptr ⇒ (_, (_::linorder) object_ptr option) dom_prog"
  fixes get_parent_locs :: "((_) heap ⇒ (_) heap ⇒ bool) set"

locale l_get_parent_Core_DOM =
  l_get_child_nodes type_wf known_ptr get_child_nodes get_child_nodes_locs +
  l_known_ptrs known_ptr known_ptrs +
  l_get_parent_Core_DOM_defs get_child_nodes get_child_nodes_locs +
  l_get_parent_defs get_parent get_parent_locs
  for known_ptr :: "(_::linorder) object_ptr ⇒ bool"
  and type_wf :: "(_) heap ⇒ bool"
  and get_child_nodes
  and get_child_nodes_locs
  and known_ptrs :: "(_) heap ⇒ bool"
  and get_parent :: "(_) node_ptr ⇒ ((_) heap, exception, (_) object_ptr option) prog"
  and get_parent_locs +
  assumes get_parent_impl: "get_parent = a_get_parent"
  assumes get_parent_locs_impl: "get_parent_locs = a_get_parent_locs"
begin
lemmas get_parent_def = get_parent_impl[unfolded a_get_parent_def]
lemmas get_parent_locs_def = get_parent_locs_impl[unfolded a_get_parent_locs_def]

lemma get_parent_pure [simp]: "pure (get_parent ptr) h"
  ⟨proof⟩
```

```
lemma get_parent_ok [simp]:
  assumes "type_wf h"
  assumes "known_ptrs h"
  assumes "ptr |∈| node_ptr_kinds h"
  shows "h ⊢ ok (get_parent ptr)"
  ⟨proof⟩

lemma get_parent_ptr_in_heap [simp]: "h ⊢ ok (get_parent node_ptr) ⟹ node_ptr |∈| node_ptr_kinds h"
  ⟨proof⟩

lemma get_parent_parent_in_heap:
  assumes "h ⊢ get_parent child_node →r Some parent"
  shows "parent |∈| object_ptr_kinds h"
  ⟨proof⟩

lemma get_parent_child_dual:
  assumes "h ⊢ get_parent child →r Some ptr"
  obtains children where "h ⊢ get_child_nodes ptr →r children" and "child ∈ set children"
  ⟨proof⟩

lemma get_parent_reads: "reads get_parent_locs (get_parent node_ptr) h h'"
  ⟨proof⟩

lemma get_parent_reads_pointers: "preserved (get_MObject ptr RObject.nothing) ∈ get_parent_locs"
  ⟨proof⟩
end

locale l_get_parent = l_type_wf + l_known_ptrs + l_get_parent_defs + l_get_child_nodes +
  assumes get_parent_reads:
    "reads get_parent_locs (get_parent node_ptr) h h'"
  assumes get_parent_ok:
    "type_wf h ⟹ known_ptrs h ⟹ node_ptr |∈| node_ptr_kinds h ⟹ h ⊢ ok (get_parent node_ptr)"
  assumes get_parent_ptr_in_heap:
    "h ⊢ ok (get_parent node_ptr) ⟹ node_ptr |∈| node_ptr_kinds h"
  assumes get_parent_pure [simp]:
    "pure (get_parent node_ptr) h"
  assumes get_parent_parent_in_heap:
    "h ⊢ get_parent child_node →r Some parent ⟹ parent |∈| object_ptr_kinds h"
  assumes get_parent_child_dual:
    "h ⊢ get_parent child →r Some ptr ⟹ (⋀children. h ⊢ get_child_nodes ptr →r children
         ⟹ child ∈ set children ⟹ thesis) ⟹ thesis"
  assumes get_parent_reads_pointers:
    "preserved (get_MObject ptr RObject.nothing) ∈ get_parent_locs"

global_interpretation l_get_parentCore_DOM_defs get_child_nodes get_child_nodes_locs defines
  get_parent = "l_get_parentCore_DOM_defs.a_get_parent get_child_nodes" and
  get_parent_locs = "l_get_parentCore_DOM_defs.a_get_parent_locs get_child_nodes_locs" ⟨proof⟩

interpretation
  i_get_parent?: l_get_parentCore_DOM known_ptr type_wf get_child_nodes get_child_nodes_locs known_ptrs

                  get_parent get_parent_locs
  ⟨proof⟩
declare l_get_parentCore_DOM_axioms[instances]

lemma get_parent_is_l_get_parent [instances]:
  "l_get_parent type_wf known_ptr known_ptrs get_parent get_parent_locs get_child_nodes get_child_nodes_locs"
  ⟨proof⟩
```

set_disconnected_nodes

```
locale l_set_disconnected_nodes_get_parentCore_DOM =
  l_set_disconnected_nodes_get_child_nodes
    set_disconnected_nodes set_disconnected_nodes_locs get_child_nodes get_child_nodes_locs
  + l_set_disconnected_nodes_{Core_DOM}
    type_wf set_disconnected_nodes set_disconnected_nodes_locs
  + l_get_parent_{Core_DOM}
    known_ptr type_wf get_child_nodes get_child_nodes_locs known_ptrs get_parent get_parent_locs
  for known_ptr :: "(_::linorder) object_ptr ⇒ bool"
  and type_wf :: "(_) heap ⇒ bool"
  and set_disconnected_nodes :: "(_) document_ptr ⇒ (_) node_ptr list ⇒ ((_) heap, exception, unit) prog"
  and set_disconnected_nodes_locs :: "(_) document_ptr ⇒ ((_) heap, exception, unit) prog set"
  and get_child_nodes :: "(_) object_ptr ⇒ ((_) heap, exception, (_) node_ptr list) prog"
  and get_child_nodes_locs :: "(_) object_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"
  and known_ptrs :: "(_) heap ⇒ bool"
  and get_parent :: "(_) node_ptr ⇒ ((_) heap, exception, (_) object_ptr option) prog"
  and get_parent_locs :: "((_) heap ⇒ (_) heap ⇒ bool) set"
begin
lemma set_disconnected_nodes_get_parent [simp]:
  "∀w ∈ set_disconnected_nodes_locs ptr. (h ⊢ w →_h h' ⟶ (∀r ∈ get_parent_locs. r h h'))"
  ⟨proof⟩
end

locale l_set_disconnected_nodes_get_parent = l_set_disconnected_nodes_defs + l_get_parent_defs +
  assumes set_disconnected_nodes_get_parent [simp]:
    "∀w ∈ set_disconnected_nodes_locs ptr. (h ⊢ w →_h h' ⟶ (∀r ∈ get_parent_locs. r h h'))"

interpretation i_set_disconnected_nodes_get_parent?:
  l_set_disconnected_nodes_get_parent_{Core_DOM} known_ptr type_wf set_disconnected_nodes
  set_disconnected_nodes_locs get_child_nodes get_child_nodes_locs known_ptrs get_parent get_parent_locs
  ⟨proof⟩
declare l_set_disconnected_nodes_get_parent_{Core_DOM}_axioms[instances]

lemma set_disconnected_nodes_get_parent_is_l_set_disconnected_nodes_get_parent [instances]:
  "l_set_disconnected_nodes_get_parent set_disconnected_nodes_locs get_parent_locs"
  ⟨proof⟩
```

get_root_node

```
locale l_get_root_node_{Core_DOM}_defs =
  l_get_parent_defs get_parent get_parent_locs
  for get_parent :: "(_) node_ptr ⇒ ((_) heap, exception, (_::linorder) object_ptr option) prog"
  and get_parent_locs :: "((_) heap ⇒ (_) heap ⇒ bool) set"
begin
partial_function (dom_prog)
  a_get_ancestors :: "(_::linorder) object_ptr ⇒ (_, (_) object_ptr list) dom_prog"
  where
    "a_get_ancestors ptr = do {
      check_in_heap ptr;
      ancestors ← (case cast_{object_ptr2node_ptr} ptr of
        Some node_ptr ⇒ do {
          parent_ptr_opt ← get_parent node_ptr;
          (case parent_ptr_opt of
            Some parent_ptr ⇒ a_get_ancestors parent_ptr
          | None ⇒ return [])
        }
      | None ⇒ return []);
      return (ptr # ancestors)
    }"

definition "a_get_ancestors_locs = get_parent_locs"

definition a_get_root_node :: "(_) object_ptr ⇒ (_, (_) object_ptr) dom_prog"
  where
    "a_get_root_node ptr = do {
      ancestors ← a_get_ancestors ptr;
      return (last ancestors)
    }"
definition "a_get_root_node_locs = a_get_ancestors_locs"
end

locale l_get_ancestors_defs =
  fixes get_ancestors :: "(_::linorder) object_ptr ⇒ (_, (_) object_ptr list) dom_prog"
  fixes get_ancestors_locs :: "((_) heap ⇒ (_) heap ⇒ bool) set"

locale l_get_root_node_defs =
  fixes get_root_node :: "(_) object_ptr ⇒ (_, (_) object_ptr) dom_prog"
  fixes get_root_node_locs :: "((_) heap ⇒ (_) heap ⇒ bool) set"

locale l_get_root_node_Core_DOM =
  l_get_parent +
  l_get_root_node_Core_DOM_defs +
  l_get_ancestors_defs +
  l_get_root_node_defs +
  assumes get_ancestors_impl: "get_ancestors = a_get_ancestors"
  assumes get_ancestors_locs_impl: "get_ancestors_locs = a_get_ancestors_locs"
  assumes get_root_node_impl: "get_root_node = a_get_root_node"
  assumes get_root_node_locs_impl: "get_root_node_locs = a_get_root_node_locs"
begin
lemmas get_ancestors_def = a_get_ancestors.simps[folded get_ancestors_impl]
lemmas get_ancestors_locs_def = a_get_ancestors_locs_def[folded get_ancestors_locs_impl]
lemmas get_root_node_def = a_get_root_node_def[folded get_root_node_impl get_ancestors_impl]
lemmas get_root_node_locs_def = a_get_root_node_locs_def[folded get_root_node_locs_impl
                                                         get_ancestors_locs_impl]

lemma get_ancestors_pure [simp]:
  "pure (get_ancestors ptr) h"
⟨proof⟩


lemma get_root_node_pure [simp]: "pure (get_root_node ptr) h"
  ⟨proof⟩


lemma get_ancestors_ptr_in_heap:
  assumes "h ⊢ ok (get_ancestors ptr)"
  shows "ptr |∈| object_ptr_kinds h"
  ⟨proof⟩

lemma get_ancestors_ptr:
  assumes "h ⊢ get_ancestors ptr →r ancestors"
  shows "ptr ∈ set ancestors"
  ⟨proof⟩

lemma get_ancestors_not_node:
  assumes "h ⊢ get_ancestors ptr →r ancestors"
  assumes "¬is_node_ptr_kind ptr"
  shows "ancestors = [ptr]"
  ⟨proof⟩

lemma get_root_node_no_parent:
  "h ⊢ get_parent node_ptr →r None ⟹ h ⊢ get_root_node (cast node_ptr) →r cast node_ptr"
  ⟨proof⟩

end

locale l_get_ancestors = l_get_ancestors_defs +
  assumes get_ancestors_pure [simp]: "pure (get_ancestors node_ptr) h"
  assumes get_ancestors_ptr_in_heap: "h ⊢ ok (get_ancestors ptr) ⟹ ptr |∈| object_ptr_kinds h"
  assumes get_ancestors_ptr: "h ⊢ get_ancestors ptr →r ancestors ⟹ ptr ∈ set ancestors"
```

```
locale l_get_root_node = l_get_root_node_defs + l_get_parent_defs +
  assumes get_root_node_pure[simp]:
   "pure (get_root_node ptr) h"
  assumes get_root_node_no_parent:
   "h ⊢ get_parent node_ptr →r None ⟹ h ⊢ get_root_node (cast node_ptr) →r cast node_ptr"

global_interpretation l_get_root_node_Core_DOM_defs get_parent get_parent_locs
  defines get_root_node = "l_get_root_node_Core_DOM_defs.a_get_root_node get_parent"
      and get_root_node_locs = "l_get_root_node_Core_DOM_defs.a_get_root_node_locs get_parent_locs"
      and get_ancestors = "l_get_root_node_Core_DOM_defs.a_get_ancestors get_parent"
      and get_ancestors_locs = "l_get_root_node_Core_DOM_defs.a_get_ancestors_locs get_parent_locs"
  ⟨proof⟩
declare a_get_ancestors.simps [code]

interpretation
  i_get_root_node?: l_get_root_node_Core_DOM type_wf known_ptr known_ptrs get_parent get_parent_locs
                    get_child_nodes get_child_nodes_locs get_ancestors get_ancestors_locs
                    get_root_node get_root_node_locs
  ⟨proof⟩
declare l_get_root_node_Core_DOM_axioms[instances]

lemma get_ancestors_is_l_get_ancestors [instances]: "l_get_ancestors get_ancestors"
  ⟨proof⟩

lemma get_root_node_is_l_get_root_node [instances]: "l_get_root_node get_root_node get_parent"
  ⟨proof⟩
```

set_disconnected_nodes

```
locale l_set_disconnected_nodes_get_ancestors_Core_DOM =
  l_set_disconnected_nodes_get_parent
    set_disconnected_nodes set_disconnected_nodes_locs get_parent get_parent_locs
  + l_get_root_node_Core_DOM
    type_wf known_ptr known_ptrs get_parent get_parent_locs get_child_nodes get_child_nodes_locs
    get_ancestors get_ancestors_locs get_root_node get_root_node_locs
  + l_set_disconnected_nodes_Core_DOM
     type_wf set_disconnected_nodes set_disconnected_nodes_locs
  for known_ptr :: "(_::linorder) object_ptr ⇒ bool"
  and set_disconnected_nodes :: "(_) document_ptr ⇒ (_) node_ptr list ⇒ ((_) heap, exception, unit) prog"
  and set_disconnected_nodes_locs :: "(_) document_ptr ⇒ ((_) heap, exception, unit) prog set"
  and get_child_nodes :: "(_) object_ptr ⇒ ((_) heap, exception, (_) node_ptr list) prog"
  and get_child_nodes_locs :: "(_) object_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"
  and get_parent :: "(_) node_ptr ⇒ ((_) heap, exception, (_) object_ptr option) prog"
  and get_parent_locs :: "((_) heap ⇒ (_) heap ⇒ bool) set"
  and type_wf :: "(_) heap ⇒ bool"
  and known_ptrs :: "(_) heap ⇒ bool"
  and get_ancestors :: "(_) object_ptr ⇒ ((_) heap, exception, (_) object_ptr list) prog"
  and get_ancestors_locs :: "((_) heap ⇒ (_) heap ⇒ bool) set"
  and get_root_node :: "(_) object_ptr ⇒ ((_) heap, exception, (_) object_ptr) prog"
  and get_root_node_locs :: "((_) heap ⇒ (_) heap ⇒ bool) set"
begin
lemma set_disconnected_nodes_get_ancestors:
  "∀w ∈ set_disconnected_nodes_locs ptr. (h ⊢ w →h h' ⟶ (∀r ∈ get_ancestors_locs. r h h'))"
  ⟨proof⟩
end

locale l_set_disconnected_nodes_get_ancestors = l_set_disconnected_nodes_defs + l_get_ancestors_defs +
  assumes set_disconnected_nodes_get_ancestors:
    "∀w ∈ set_disconnected_nodes_locs ptr. (h ⊢ w →h h' ⟶ (∀r ∈ get_ancestors_locs. r h h'))"

interpretation
  i_set_disconnected_nodes_get_ancestors?: l_set_disconnected_nodes_get_ancestors_Core_DOM known_ptr
                                           set_disconnected_nodes set_disconnected_nodes_locs
                                           get_child_nodes get_child_nodes_locs get_parent
```

```
                                          get_parent_locs type_wf known_ptrs get_ancestors
                                          get_ancestors_locs get_root_node get_root_node_locs
  ⟨proof⟩
declare l_set_disconnected_nodes_get_ancestors_Core_DOM_axioms[instances]


lemma set_disconnected_nodes_get_ancestors_is_l_set_disconnected_nodes_get_ancestors [instances]:
  "l_set_disconnected_nodes_get_ancestors set_disconnected_nodes_locs get_ancestors_locs"
  ⟨proof⟩
```

get_owner_document

```
locale l_get_owner_document_Core_DOM_defs =
  l_get_disconnected_nodes_defs get_disconnected_nodes get_disconnected_nodes_locs +
  l_get_root_node_defs get_root_node get_root_node_locs
  for get_root_node :: "(_::linorder) object_ptr ⇒ ((_) heap, exception, (_) object_ptr) prog"
  and get_root_node_locs :: "((_) heap ⇒ (_) heap ⇒ bool) set"
  and get_disconnected_nodes :: "(_) document_ptr ⇒ ((_) heap, exception, (_) node_ptr list) prog"
  and get_disconnected_nodes_locs :: "(_) document_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"
begin

definition a_get_owner_document_node_ptr :: "(_) node_ptr ⇒ unit ⇒ (_, (_) document_ptr) dom_prog"
  where
    "a_get_owner_document_node_ptr node_ptr _ = do {
      root ← get_root_node (cast node_ptr);
      (case cast root of
        Some document_ptr ⇒ return document_ptr
      | None ⇒ do {
        ptrs ← document_ptr_kinds_M;
        candidates ← filter_M (λdocument_ptr. do {
          disconnected_nodes ← get_disconnected_nodes document_ptr;
          return (root ∈ cast ‘ set disconnected_nodes)
        }) ptrs;
        return (hd candidates)
      })
    }"

definition
  a_get_owner_document_document_ptr :: "(_) document_ptr ⇒ unit ⇒ (_, (_) document_ptr) dom_prog"
  where
    "a_get_owner_document_document_ptr document_ptr _ = do {
      document_ptrs ← document_ptr_kinds_M;
      (if document_ptr ∈ set document_ptrs then return document_ptr else error SegmentationFault)}"

definition
  a_get_owner_document_tups :: "(((_) object_ptr ⇒ bool) × ((_) object_ptr ⇒ unit
                                            ⇒ (_, (_) document_ptr) dom_prog)) list"
  where
     "a_get_owner_document_tups = [
        (is_element_ptr, a_get_owner_document_node_ptr ∘ the ∘ cast),
        (is_character_data_ptr, a_get_owner_document_node_ptr ∘ the ∘ cast),
        (is_document_ptr, a_get_owner_document_document_ptr ∘ the ∘ cast)
      ]"

definition a_get_owner_document :: "(_) object_ptr ⇒ (_, (_) document_ptr) dom_prog"
  where
    "a_get_owner_document ptr = invoke a_get_owner_document_tups ptr ()"
end

locale l_get_owner_document_defs =
  fixes get_owner_document :: "(_::linorder) object_ptr ⇒ (_, (_) document_ptr) dom_prog"

locale l_get_owner_document_Core_DOM =
  l_known_ptr known_ptr +
  l_get_disconnected_nodes type_wf get_disconnected_nodes get_disconnected_nodes_locs +
  l_get_root_node get_root_node get_root_node_locs +
  l_get_owner_document_Core_DOM_defs get_root_node get_root_node_locs get_disconnected_nodes
                                     get_disconnected_nodes_locs +
  l_get_owner_document_defs get_owner_document
  for known_ptr :: "(_::linorder) object_ptr ⇒ bool"
  and type_wf :: "(_) heap ⇒ bool"
  and get_disconnected_nodes :: "(_) document_ptr ⇒ ((_) heap, exception, (_) node_ptr list) prog"
  and get_disconnected_nodes_locs :: "(_) document_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"
  and get_root_node :: "(_) object_ptr ⇒ ((_) heap, exception, (_) object_ptr) prog"
  and get_root_node_locs :: "((_) heap ⇒ (_) heap ⇒ bool) set"
  and get_owner_document :: "(_) object_ptr ⇒ ((_) heap, exception, (_) document_ptr) prog" +
  assumes known_ptr_impl: "known_ptr = a_known_ptr"
  assumes get_owner_document_impl: "get_owner_document = a_get_owner_document"
begin
lemmas known_ptr_def = known_ptr_impl[unfolded a_known_ptr_def]
lemmas get_owner_document_def = a_get_owner_document_def[folded get_owner_document_impl]

lemma get_owner_document_split:
  "P (invoke (a_get_owner_document_tups @ xs) ptr ()) =
    ((known_ptr ptr ⟶ P (get_owner_document ptr))
    ∧ (¬(known_ptr ptr) ⟶ P (invoke xs ptr ())))"
  ⟨proof⟩

lemma get_owner_document_split_asm:
  "P (invoke (a_get_owner_document_tups @ xs) ptr ()) =
    (¬((known_ptr ptr ∧ ¬P (get_owner_document ptr))
    ∨ (¬(known_ptr ptr) ∧ ¬P (invoke xs ptr ()))))"
  ⟨proof⟩
lemmas get_owner_document_splits = get_owner_document_split get_owner_document_split_asm

lemma get_owner_document_pure [simp]:
  "pure (get_owner_document ptr) h"
⟨proof⟩

lemma get_owner_document_ptr_in_heap:
  assumes "h ⊢ ok (get_owner_document ptr)"
  shows "ptr |∈| object_ptr_kinds h"
  ⟨proof⟩
end

locale l_get_owner_document = l_get_owner_document_defs +
  assumes get_owner_document_ptr_in_heap:
    "h ⊢ ok (get_owner_document ptr) ⟹ ptr |∈| object_ptr_kinds h"
  assumes get_owner_document_pure [simp]:
    "pure (get_owner_document ptr) h"

global_interpretation l_get_owner_document_Core_DOM_defs get_root_node get_root_node_locs
                                                         get_disconnected_nodes get_disconnected_nodes_locs
  defines get_owner_document_tups =
        "l_get_owner_document_Core_DOM_defs.a_get_owner_document_tups get_root_node get_disconnected_nodes"
     and get_owner_document =
        "l_get_owner_document_Core_DOM_defs.a_get_owner_document get_root_node get_disconnected_nodes"
     and get_owner_document_node_ptr =
        "l_get_owner_document_Core_DOM_defs.a_get_owner_document_node_ptr get_root_node get_disconnected_nodes"
  ⟨proof⟩
interpretation
  i_get_owner_document?: l_get_owner_document_Core_DOM get_parent get_parent_locs known_ptr type_wf
                         get_disconnected_nodes get_disconnected_nodes_locs get_root_node
                         get_root_node_locs get_owner_document
  ⟨proof⟩
declare l_get_owner_document_Core_DOM_axioms[instances]
```

```
lemma get_owner_document_is_l_get_owner_document [instances]:
  "l_get_owner_document get_owner_document"
  ⟨proof⟩
```

remove_child

```
locale l_remove_child_Core_DOM_defs =
  l_get_child_nodes_defs get_child_nodes get_child_nodes_locs +
  l_set_child_nodes_defs set_child_nodes set_child_nodes_locs +
  l_get_parent_defs get_parent get_parent_locs +
  l_get_owner_document_defs get_owner_document +
  l_get_disconnected_nodes_defs get_disconnected_nodes get_disconnected_nodes_locs +
  l_set_disconnected_nodes_defs set_disconnected_nodes set_disconnected_nodes_locs
  for get_child_nodes :: "(_::linorder) object_ptr ⇒ ((_) heap, exception, (_) node_ptr list) prog"
  and get_child_nodes_locs :: "(_) object_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"
  and set_child_nodes :: "(_) object_ptr ⇒ (_) node_ptr list ⇒ ((_) heap, exception, unit) prog"
  and set_child_nodes_locs :: "(_) object_ptr ⇒ ((_) heap, exception, unit) prog set"
  and get_parent :: "(_) node_ptr ⇒ ((_) heap, exception, (_) object_ptr option) prog"
  and get_parent_locs :: "((_) heap ⇒ (_) heap ⇒ bool) set"
  and get_owner_document :: "(_) object_ptr ⇒ ((_) heap, exception, (_) document_ptr) prog"
  and get_disconnected_nodes :: "(_) document_ptr ⇒ ((_) heap, exception, (_) node_ptr list) prog"
  and get_disconnected_nodes_locs :: "(_) document_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"
  and set_disconnected_nodes :: "(_) document_ptr ⇒ (_) node_ptr list ⇒ ((_) heap, exception, unit) prog"
  and set_disconnected_nodes_locs :: "(_) document_ptr ⇒ ((_) heap, exception, unit) prog set"
begin
definition a_remove_child :: "(_) object_ptr ⇒ (_) node_ptr ⇒ (_, unit) dom_prog"
  where
    "a_remove_child ptr child = do {
      children ← get_child_nodes ptr;
      if child ∉ set children then
        error NotFoundError
      else do {
        owner_document ← get_owner_document (cast child);
        disc_nodes ← get_disconnected_nodes owner_document;
        set_disconnected_nodes owner_document (child # disc_nodes);
        set_child_nodes ptr (remove1 child children)
      }
    }"

definition a_remove_child_locs :: "(_) object_ptr ⇒ (_) document_ptr ⇒ (_, unit) dom_prog set"
  where
    "a_remove_child_locs ptr owner_document = set_child_nodes_locs ptr
                                              ∪ set_disconnected_nodes_locs owner_document"

definition a_remove :: "(_) node_ptr ⇒ (_, unit) dom_prog"
  where
    "a_remove node_ptr = do {
      parent_opt ← get_parent node_ptr;
      (case parent_opt of
        Some parent ⇒ do {
          a_remove_child parent node_ptr;
          return ()
        }
      | None ⇒ return ())
    }"
end

locale l_remove_child_defs =
  fixes remove_child :: "(_::linorder) object_ptr ⇒ (_) node_ptr ⇒ (_, unit) dom_prog"
  fixes remove_child_locs :: "(_) object_ptr ⇒ (_) document_ptr ⇒ (_, unit) dom_prog set"

locale l_remove_defs =
  fixes remove :: "(_) node_ptr ⇒ (_, unit) dom_prog"

locale l_remove_child_Core_DOM =
  l_remove_child_Core_DOM_defs +
  l_remove_child_defs +
  l_remove_defs +
  l_get_parent +
  l_get_owner_document +
  l_set_child_nodes_get_child_nodes +
  l_set_child_nodes_get_disconnected_nodes +
  l_set_disconnected_nodes_get_disconnected_nodes +
  l_set_disconnected_nodes_get_child_nodes +
  assumes remove_child_impl: "remove_child = a_remove_child"
  assumes remove_child_locs_impl: "remove_child_locs = a_remove_child_locs"
  assumes remove_impl: "remove = a_remove"
begin
lemmas remove_child_def = a_remove_child_def[folded remove_child_impl]
lemmas remove_child_locs_def = a_remove_child_locs_def[folded remove_child_locs_impl]
lemmas remove_def = a_remove_def[folded remove_child_impl remove_impl]

lemma remove_child_ptr_in_heap:
  assumes "h ⊢ ok (remove_child ptr child)"
  shows "ptr |∈| object_ptr_kinds h"
⟨proof⟩


lemma remove_child_child_in_heap:
  assumes "h ⊢ remove_child ptr' child →_h h'"
  shows "child |∈| node_ptr_kinds h"
  ⟨proof⟩


lemma remove_child_in_disconnected_nodes:

  assumes "h ⊢ remove_child ptr child →_h h'"
  assumes "h ⊢ get_owner_document (cast child) →_r owner_document"
  assumes "h' ⊢ get_disconnected_nodes owner_document →_r disc_nodes"
  shows "child ∈ set disc_nodes"
⟨proof⟩

lemma remove_child_writes [simp]:
  "writes (remove_child_locs ptr |h ⊢ get_owner_document (cast child)|_r) (remove_child ptr child) h h'"
  ⟨proof⟩

lemma remove_writes:
  "writes (remove_child_locs (the |h ⊢ get_parent child|_r) |h ⊢ get_owner_document (cast child)|_r)
(remove child) h h'"
  ⟨proof⟩

lemma remove_child_children_subset:
  assumes "h ⊢ remove_child parent child →_h h'"
    and "h ⊢ get_child_nodes ptr →_r children"
    and "h' ⊢ get_child_nodes ptr →_r children'"
    and known_ptrs: "known_ptrs h"
    and type_wf: "type_wf h"
  shows "set children' ⊆ set children"
⟨proof⟩


lemma remove_child_pointers_preserved:
  assumes "w ∈ remove_child_locs ptr owner_document"
  assumes "h ⊢ w →_h h'"
  shows "object_ptr_kinds h = object_ptr_kinds h'"
  ⟨proof⟩

lemma remove_child_types_preserved:
  assumes "w ∈ remove_child_locs ptr owner_document"
  assumes "h ⊢ w →h h'"
  shows "type_wf h = type_wf h'"
  ⟨proof⟩
end

locale l_remove_child = l_type_wf + l_known_ptrs + l_remove_child_defs + l_get_owner_document_defs
                    + l_get_child_nodes_defs + l_get_disconnected_nodes_defs +
  assumes remove_child_writes:
    "writes (remove_child_locs object_ptr |h ⊢ get_owner_document (cast child)|r)
(remove_child object_ptr child) h h'"
  assumes remove_child_pointers_preserved:
  "w ∈ remove_child_locs ptr owner_document ⟹ h ⊢ w →h h' ⟹ object_ptr_kinds h = object_ptr_kinds
h'"
  assumes remove_child_types_preserved:
  "w ∈ remove_child_locs ptr owner_document ⟹ h ⊢ w →h h' ⟹ type_wf h = type_wf h'"
  assumes remove_child_in_disconnected_nodes:
    "known_ptrs h ⟹ h ⊢ remove_child ptr child →h h'
    ⟹ h ⊢ get_owner_document (cast child) →r owner_document
    ⟹ h' ⊢ get_disconnected_nodes owner_document →r disc_nodes
    ⟹ child ∈ set disc_nodes"
  assumes remove_child_ptr_in_heap: "h ⊢ ok (remove_child ptr child) ⟹ ptr |∈| object_ptr_kinds h"
  assumes remove_child_child_in_heap: "h ⊢ remove_child ptr' child →h h' ⟹ child |∈| node_ptr_kinds
h"
  assumes remove_child_children_subset:
    "known_ptrs h ⟹ type_wf h ⟹ h ⊢ remove_child parent child →h h'
    ⟹ h ⊢ get_child_nodes ptr →r children
    ⟹ h' ⊢ get_child_nodes ptr →r children'
    ⟹ set children' ⊆ set children"

locale l_remove


global_interpretation l_remove_child_Core_DOM_defs get_child_nodes get_child_nodes_locs set_child_nodes

                                        set_child_nodes_locs get_parent get_parent_locs
                                        get_owner_document get_disconnected_nodes
                                        get_disconnected_nodes_locs set_disconnected_nodes
                                        set_disconnected_nodes_locs
  defines remove =
    "l_remove_child_Core_DOM_defs.a_remove get_child_nodes set_child_nodes get_parent get_owner_document

                                    get_disconnected_nodes set_disconnected_nodes"
  and remove_child =
    "l_remove_child_Core_DOM_defs.a_remove_child get_child_nodes set_child_nodes get_owner_document
                                        get_disconnected_nodes set_disconnected_nodes"
  and remove_child_locs =
    "l_remove_child_Core_DOM_defs.a_remove_child_locs set_child_nodes_locs set_disconnected_nodes_locs"
  ⟨proof⟩
interpretation
  i_remove_child?: l_remove_child_Core_DOM get_child_nodes get_child_nodes_locs set_child_nodes
                  set_child_nodes_locs get_parent get_parent_locs get_owner_document
                  get_disconnected_nodes get_disconnected_nodes_locs set_disconnected_nodes
                  set_disconnected_nodes_locs remove_child remove_child_locs remove type_wf
                  known_ptr known_ptrs
  ⟨proof⟩
declare l_remove_child_Core_DOM_axioms[instances]

lemma remove_child_is_l_remove_child [instances]:
  "l_remove_child type_wf known_ptr known_ptrs remove_child remove_child_locs get_owner_document
```

```
                get_child_nodes get_disconnected_nodes"
  ⟨proof⟩
```

adopt_node

```
locale l_adopt_node_Core_DOM_defs =
  l_get_owner_document_defs get_owner_document +
  l_get_parent_defs get_parent get_parent_locs +
  l_remove_child_defs remove_child remove_child_locs +
  l_get_disconnected_nodes_defs get_disconnected_nodes get_disconnected_nodes_locs +
  l_set_disconnected_nodes_defs set_disconnected_nodes set_disconnected_nodes_locs
  for get_owner_document :: "(_::linorder) object_ptr ⇒ ((_) heap, exception, (_) document_ptr) prog"
  and get_parent :: "(_) node_ptr ⇒ ((_) heap, exception, (_) object_ptr option) prog"
  and get_parent_locs :: "((_) heap ⇒ (_) heap ⇒ bool) set"
  and remove_child :: "(_) object_ptr ⇒ (_) node_ptr ⇒ ((_) heap, exception, unit) prog"
  and remove_child_locs :: "(_) object_ptr ⇒ (_) document_ptr ⇒ ((_) heap, exception, unit) prog set"
  and get_disconnected_nodes :: "(_) document_ptr ⇒ ((_) heap, exception, (_) node_ptr list) prog"
  and get_disconnected_nodes_locs :: "(_) document_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"
  and set_disconnected_nodes :: "(_) document_ptr ⇒ (_) node_ptr list ⇒ ((_) heap, exception, unit) prog"
  and set_disconnected_nodes_locs :: "(_) document_ptr ⇒ ((_) heap, exception, unit) prog set"
begin
definition a_adopt_node :: "(_) document_ptr ⇒ (_) node_ptr ⇒ (_, unit) dom_prog"
  where
    "a_adopt_node document node = do {
      old_document ← get_owner_document (cast node);
      parent_opt ← get_parent node;
      (case parent_opt of
        Some parent ⇒ do {
          remove_child parent node
        } | None ⇒ do {
          return ()
        });
      (if document ≠ old_document then do {
        old_disc_nodes ← get_disconnected_nodes old_document;
        set_disconnected_nodes old_document (remove1 node old_disc_nodes);
        disc_nodes ← get_disconnected_nodes document;
        set_disconnected_nodes document (node # disc_nodes)
      } else do {
        return ()
      })
    }"

definition
  a_adopt_node_locs :: "(_) object_ptr option ⇒ (_) document_ptr ⇒ (_) document_ptr ⇒ (_, unit) dom_prog
set"
  where
    "a_adopt_node_locs parent owner_document document_ptr =
    ((if parent = None
      then {}
      else remove_child_locs (the parent) owner_document) ∪ set_disconnected_nodes_locs document_ptr
                                                           ∪ set_disconnected_nodes_locs owner_document)"
end

locale l_adopt_node_defs =
  fixes
  adopt_node :: "(_) document_ptr ⇒ (_) node_ptr ⇒ (_, unit) dom_prog"
  fixes
  adopt_node_locs :: "(_) object_ptr option ⇒ (_) document_ptr ⇒ (_) document_ptr ⇒ (_, unit) dom_prog
set"

global_interpretation l_adopt_node_Core_DOM_defs get_owner_document get_parent get_parent_locs remove_child

                                    remove_child_locs get_disconnected_nodes
```

```
                                        get_disconnected_nodes_locs set_disconnected_nodes
                                        set_disconnected_nodes_locs
  defines adopt_node = "l_adopt_node_Core_DOM_defs.a_adopt_node get_owner_document get_parent remove_child

                                                      get_disconnected_nodes set_disconnected_nodes"
    and adopt_node_locs = "l_adopt_node_Core_DOM_defs.a_adopt_node_locs
                                                 remove_child_locs set_disconnected_nodes_locs"
  ⟨proof⟩

locale l_adopt_node_Core_DOM =
  l_adopt_node_Core_DOM_defs
    get_owner_document get_parent get_parent_locs remove_child remove_child_locs get_disconnected_nodes

                     get_disconnected_nodes_locs set_disconnected_nodes set_disconnected_nodes_locs
  + l_adopt_node_defs
    adopt_node adopt_node_locs
  + l_get_owner_document
    get_owner_document
  + l_get_parent_Core_DOM
    known_ptr type_wf get_child_nodes get_child_nodes_locs known_ptrs get_parent get_parent_locs
  + l_remove_child_Core_DOM
    get_child_nodes get_child_nodes_locs set_child_nodes set_child_nodes_locs get_parent
    get_parent_locs get_owner_document get_disconnected_nodes get_disconnected_nodes_locs
    set_disconnected_nodes set_disconnected_nodes_locs remove_child remove_child_locs remove type_wf
    known_ptr known_ptrs
  + l_set_disconnected_nodes_get_disconnected_nodes
    type_wf get_disconnected_nodes get_disconnected_nodes_locs set_disconnected_nodes
    set_disconnected_nodes_locs
  for get_owner_document :: "(_::linorder) object_ptr ⇒ ((_) heap, exception, (_) document_ptr) prog"
  and get_parent :: "(_) node_ptr ⇒ ((_) heap, exception, (_) object_ptr option) prog"
  and get_parent_locs :: "((_) heap ⇒ (_) heap ⇒ bool) set"
  and remove_child :: "(_) object_ptr ⇒ (_) node_ptr ⇒ ((_) heap, exception, unit) prog"
  and remove_child_locs :: "(_) object_ptr ⇒ (_) document_ptr ⇒ ((_) heap, exception, unit) prog set"
  and get_disconnected_nodes :: "(_) document_ptr ⇒ ((_) heap, exception, (_) node_ptr list) prog"
  and get_disconnected_nodes_locs :: "(_) document_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"
  and set_disconnected_nodes :: "(_) document_ptr ⇒ (_) node_ptr list ⇒ ((_) heap, exception, unit) prog"
  and set_disconnected_nodes_locs :: "(_) document_ptr ⇒ ((_) heap, exception, unit) prog set"
  and adopt_node :: "(_) document_ptr ⇒ (_) node_ptr ⇒ ((_) heap, exception, unit) prog"
  and adopt_node_locs :: "(_) object_ptr option ⇒ (_) document_ptr ⇒ (_) document_ptr
                          ⇒ ((_) heap, exception, unit) prog set"
  and known_ptr :: "(_) object_ptr ⇒ bool"
  and type_wf :: "(_) heap ⇒ bool"
  and get_child_nodes :: "(_) object_ptr ⇒ ((_) heap, exception, (_) node_ptr list) prog"
  and get_child_nodes_locs :: "(_) object_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"
  and known_ptrs :: "(_) heap ⇒ bool"
  and set_child_nodes :: "(_) object_ptr ⇒ (_) node_ptr list ⇒ ((_) heap, exception, unit) prog"
  and set_child_nodes_locs :: "(_) object_ptr ⇒ ((_) heap, exception, unit) prog set"
  and remove :: "(_) node_ptr ⇒ ((_) heap, exception, unit) prog" +
  assumes adopt_node_impl: "adopt_node = a_adopt_node"
  assumes adopt_node_locs_impl: "adopt_node_locs = a_adopt_node_locs"
begin
lemmas adopt_node_def = a_adopt_node_def[folded adopt_node_impl]
lemmas adopt_node_locs_def = a_adopt_node_locs_def[folded adopt_node_locs_impl]

lemma adopt_node_writes:
  shows "writes (adopt_node_locs |h ⊢ get_parent node|_r |h
         ⊢ get_owner_document (cast node)|_r document_ptr) (adopt_node document_ptr node) h h'"
  ⟨proof⟩

lemma adopt_node_children_subset:
  assumes "h ⊢ adopt_node owner_document node →_h h'"
    and "h ⊢ get_child_nodes ptr →_r children"
    and "h' ⊢ get_child_nodes ptr →_r children'"
    and known_ptrs: "known_ptrs h"
    and type_wf: "type_wf h"
  shows "set children' ⊆ set children"
⟨proof⟩

lemma adopt_node_child_in_heap:
  assumes "h ⊢ ok (adopt_node document_ptr child)"
  shows "child |∈| node_ptr_kinds h"
  ⟨proof⟩

lemma adopt_node_pointers_preserved:
  assumes "w ∈ adopt_node_locs parent owner_document document_ptr"
  assumes "h ⊢ w →_h h'"
  shows "object_ptr_kinds h = object_ptr_kinds h'"
  ⟨proof⟩

lemma adopt_node_types_preserved:
  assumes "w ∈ adopt_node_locs parent owner_document document_ptr"
  assumes "h ⊢ w →_h h'"
  shows "type_wf h = type_wf h'"
  ⟨proof⟩
end

locale l_adopt_node = l_type_wf + l_known_ptrs + l_get_parent_defs + l_adopt_node_defs +
  l_get_child_nodes_defs + l_get_owner_document_defs +
  assumes adopt_node_writes:
  "writes (adopt_node_locs |h ⊢ get_parent node|_r
          |h ⊢ get_owner_document (cast node)|_r document_ptr) (adopt_node document_ptr node) h h'"
  assumes adopt_node_pointers_preserved:
  "w ∈ adopt_node_locs parent owner_document document_ptr
   ⟹ h ⊢ w →_h h' ⟹ object_ptr_kinds h = object_ptr_kinds h'"
  assumes adopt_node_types_preserved:
  "w ∈ adopt_node_locs parent owner_document document_ptr
   ⟹ h ⊢ w →_h h' ⟹ type_wf h = type_wf h'"
  assumes adopt_node_child_in_heap:
  "h ⊢ ok (adopt_node document_ptr child) ⟹ child |∈| node_ptr_kinds h"
  assumes adopt_node_children_subset:
  "h ⊢ adopt_node owner_document node →_h h' ⟹ h ⊢ get_child_nodes ptr →_r children
     ⟹ h' ⊢ get_child_nodes ptr →_r children'
     ⟹ known_ptrs h ⟹ type_wf h ⟹ set children' ⊆ set children"

interpretation
  i_adopt_node?: l_adopt_node_Core_DOM get_owner_document get_parent get_parent_locs remove_child
                 remove_child_locs get_disconnected_nodes get_disconnected_nodes_locs
                 set_disconnected_nodes set_disconnected_nodes_locs adopt_node adopt_node_locs
                 known_ptr type_wf get_child_nodes get_child_nodes_locs known_ptrs set_child_nodes
                 set_child_nodes_locs remove
  ⟨proof⟩
declare l_adopt_node_Core_DOM_axioms[instances]


lemma adopt_node_is_l_adopt_node [instances]:
  "l_adopt_node type_wf known_ptr known_ptrs get_parent adopt_node adopt_node_locs get_child_nodes
                get_owner_document"
  ⟨proof⟩
```

insert_before

```
locale l_insert_before_Core_DOM_defs =
  l_get_parent_defs get_parent get_parent_locs
  + l_get_child_nodes_defs get_child_nodes get_child_nodes_locs
  + l_set_child_nodes_defs set_child_nodes set_child_nodes_locs
  + l_get_ancestors_defs get_ancestors get_ancestors_locs
  + l_adopt_node_defs adopt_node adopt_node_locs
  + l_set_disconnected_nodes_defs set_disconnected_nodes set_disconnected_nodes_locs
  + l_get_disconnected_nodes_defs get_disconnected_nodes get_disconnected_nodes_locs
  + l_get_owner_document_defs get_owner_document
  for get_parent :: "(_) node_ptr ⇒ ((_) heap, exception, (_::linorder) object_ptr option) prog"
  and get_parent_locs :: "((_) heap ⇒ (_) heap ⇒ bool) set"
  and get_child_nodes :: "(_) object_ptr ⇒ ((_) heap, exception, (_) node_ptr list) prog"
  and get_child_nodes_locs :: "(_) object_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"
  and set_child_nodes :: "(_) object_ptr ⇒ (_) node_ptr list ⇒ ((_) heap, exception, unit) prog"
  and set_child_nodes_locs :: "(_) object_ptr ⇒ ((_) heap, exception, unit) prog set"
  and get_ancestors :: "(_) object_ptr ⇒ ((_) heap, exception, (_) object_ptr list) prog"
  and get_ancestors_locs :: "((_) heap ⇒ (_) heap ⇒ bool) set"
  and adopt_node :: "(_) document_ptr ⇒ (_) node_ptr ⇒ ((_) heap, exception, unit) prog"
  and adopt_node_locs :: "(_) object_ptr option ⇒ (_) document_ptr ⇒ (_) document_ptr
                                       ⇒ ((_) heap, exception, unit) prog set"
  and set_disconnected_nodes :: "(_) document_ptr ⇒ (_) node_ptr list ⇒ ((_) heap, exception, unit) prog"
  and set_disconnected_nodes_locs :: "(_) document_ptr ⇒ ((_) heap, exception, unit) prog set"
  and get_disconnected_nodes :: "(_) document_ptr ⇒ ((_) heap, exception, (_) node_ptr list) prog"
  and get_disconnected_nodes_locs :: "(_) document_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"
  and get_owner_document :: "(_) object_ptr ⇒ ((_) heap, exception, (_) document_ptr) prog"
begin

definition a_next_sibling :: "(_) node_ptr ⇒ (_, (_) node_ptr option) dom_prog"
  where
    "a_next_sibling node_ptr = do {
      parent_opt ← get_parent node_ptr;
      (case parent_opt of
        Some parent ⇒ do {
          children ← get_child_nodes parent;
          (case (dropWhile (λptr. ptr = node_ptr) (dropWhile (λptr. ptr ≠ node_ptr) children)) of
            x#_ ⇒ return (Some x)
          | [] ⇒ return None)}
      | None ⇒ return None)
    }"

fun insert_before_list :: "'xyz ⇒ 'xyz option ⇒ 'xyz list ⇒ 'xyz list"
  where
    "insert_before_list v (Some reference) (x#xs) = (if reference = x
      then v#x#xs else x # insert_before_list v (Some reference) xs)"
    | "insert_before_list v (Some _) [] = [v]"
    | "insert_before_list v None xs = xs @ [v]"

definition a_insert_node :: "(_) object_ptr ⇒ (_) node_ptr ⇒ (_) node_ptr option
  ⇒ (_, unit) dom_prog"
  where
    "a_insert_node ptr new_child reference_child_opt = do {
      children ← get_child_nodes ptr;
      set_child_nodes ptr (insert_before_list new_child reference_child_opt children)
    }"

definition a_ensure_pre_insertion_validity :: "(_) node_ptr ⇒ (_) object_ptr
  ⇒ (_) node_ptr option ⇒ (_, unit) dom_prog"
  where
    "a_ensure_pre_insertion_validity node parent child_opt = do {
      (if is_character_data_ptr_kind parent
        then error HierarchyRequestError else return ());
      ancestors ← get_ancestors parent;
      (if cast node ∈ set ancestors then error HierarchyRequestError else return ());
      (case child_opt of
        Some child ⇒ do {
          child_parent ← get_parent child;
          (if child_parent ≠ Some parent then error NotFoundError else return ())}
      | None ⇒ return ());
      children ← get_child_nodes parent;
      (if children ≠ [] ∧ is_document_ptr parent
        then error HierarchyRequestError else return ());
      (if is_character_data_ptr node ∧ is_document_ptr parent
        then error HierarchyRequestError else return ())
    }"

definition a_insert_before :: "(_) object_ptr ⇒ (_) node_ptr
  ⇒ (_) node_ptr option ⇒ (_, unit) dom_prog"
  where
    "a_insert_before ptr node child = do {
      a_ensure_pre_insertion_validity node ptr child;
      reference_child ← (if Some node = child
        then a_next_sibling node
        else return child);
      owner_document ← get_owner_document ptr;
      adopt_node owner_document node;
      disc_nodes ← get_disconnected_nodes owner_document;
      set_disconnected_nodes owner_document (remove1 node disc_nodes);
      a_insert_node ptr node reference_child
    }"

definition a_insert_before_locs :: "(_) object_ptr ⇒ (_) object_ptr option ⇒ (_) document_ptr
                                        ⇒ (_) document_ptr ⇒ (_, unit) dom_prog set"
  where
    "a_insert_before_locs ptr old_parent child_owner_document ptr_owner_document =
      adopt_node_locs old_parent child_owner_document ptr_owner_document ∪
      set_child_nodes_locs ptr ∪
      set_disconnected_nodes_locs ptr_owner_document"
end

locale l_insert_before_defs =
  fixes insert_before :: "(_) object_ptr ⇒ (_) node_ptr ⇒ (_) node_ptr option ⇒ (_, unit) dom_prog"
  fixes insert_before_locs :: "(_) object_ptr ⇒ (_) object_ptr option ⇒ (_) document_ptr
                                    ⇒ (_) document_ptr ⇒ (_, unit) dom_prog set"

locale l_append_child_Core_DOM_defs =
  l_insert_before_defs
begin
definition "a_append_child ptr child = insert_before ptr child None"
end

locale l_append_child_defs =
  fixes append_child :: "(_) object_ptr ⇒ (_) node_ptr ⇒ (_, unit) dom_prog"

locale l_insert_before_Core_DOM =
  l_insert_before_Core_DOM_defs
    get_parent get_parent_locs get_child_nodes get_child_nodes_locs set_child_nodes
    set_child_nodes_locs get_ancestors get_ancestors_locs adopt_node adopt_node_locs
    set_disconnected_nodes set_disconnected_nodes_locs get_disconnected_nodes
    get_disconnected_nodes_locs get_owner_document
  + l_insert_before_defs
    insert_before insert_before_locs
  + l_append_child_defs
    append_child
  + l_set_child_nodes_get_child_nodes
    type_wf known_ptr get_child_nodes get_child_nodes_locs set_child_nodes set_child_nodes_locs
  + l_get_ancestors
    get_ancestors get_ancestors_locs
  + l_adopt_node
    type_wf known_ptr known_ptrs get_parent get_parent_locs adopt_node adopt_node_locs
    get_child_nodes get_child_nodes_locs get_owner_document
  + l_set_disconnected_nodes
```

```
    type_wf set_disconnected_nodes set_disconnected_nodes_locs
  + l_get_disconnected_nodes
    type_wf get_disconnected_nodes get_disconnected_nodes_locs
  + l_get_owner_document
    get_owner_document
  + l_get_parent_Core_DOM
    known_ptr type_wf get_child_nodes get_child_nodes_locs known_ptrs get_parent get_parent_locs
  + l_set_disconnected_nodes_get_child_nodes
    set_disconnected_nodes set_disconnected_nodes_locs get_child_nodes get_child_nodes_locs
  for get_parent :: "(_) node_ptr ⇒ ((_) heap, exception, (_::linorder) object_ptr option) prog"
  and get_parent_locs :: "((_) heap ⇒ (_) heap ⇒ bool) set"
  and get_child_nodes :: "(_) object_ptr ⇒ ((_) heap, exception, (_) node_ptr list) prog"
  and get_child_nodes_locs :: "(_) object_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"
  and set_child_nodes :: "(_) object_ptr ⇒ (_) node_ptr list ⇒ ((_) heap, exception, unit) prog"
  and set_child_nodes_locs :: "(_) object_ptr ⇒ ((_) heap, exception, unit) prog set"
  and get_ancestors :: "(_) object_ptr ⇒ ((_) heap, exception, (_) object_ptr list) prog"
  and get_ancestors_locs :: "((_) heap ⇒ (_) heap ⇒ bool) set"
  and adopt_node :: "(_) document_ptr ⇒ (_) node_ptr ⇒ ((_) heap, exception, unit) prog"
  and adopt_node_locs :: "(_) object_ptr option ⇒ (_) document_ptr ⇒ (_) document_ptr
                                        ⇒ ((_) heap, exception, unit) prog set"
  and set_disconnected_nodes :: "(_) document_ptr ⇒ (_) node_ptr list ⇒ ((_) heap, exception, unit) prog"
  and set_disconnected_nodes_locs :: "(_) document_ptr ⇒ ((_) heap, exception, unit) prog set"
  and get_disconnected_nodes :: "(_) document_ptr ⇒ ((_) heap, exception, (_) node_ptr list) prog"
  and get_disconnected_nodes_locs :: "(_) document_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"
  and get_owner_document :: "(_) object_ptr ⇒ ((_) heap, exception, (_) document_ptr) prog"
    and insert_before ::
    "(_) object_ptr ⇒ (_) node_ptr ⇒ (_) node_ptr option ⇒ ((_) heap, exception, unit) prog"
  and insert_before_locs :: "(_) object_ptr ⇒ (_) object_ptr option ⇒ (_) document_ptr
                                    ⇒ (_) document_ptr ⇒ (_, unit) dom_prog set"
  and append_child :: "(_) object_ptr ⇒ (_) node_ptr ⇒ ((_) heap, exception, unit) prog"
  and type_wf :: "(_) heap ⇒ bool"
  and known_ptr :: "(_) object_ptr ⇒ bool"
  and known_ptrs :: "(_) heap ⇒ bool" +
  assumes insert_before_impl: "insert_before = a_insert_before"
  assumes insert_before_locs_impl: "insert_before_locs = a_insert_before_locs"
begin
lemmas insert_before_def = a_insert_before_def[folded insert_before_impl]
lemmas insert_before_locs_def = a_insert_before_locs_def[folded insert_before_locs_impl]

lemma next_sibling_pure [simp]:
  "pure (a_next_sibling new_child) h"
  ⟨proof⟩

lemma insert_before_list_in_set: "x ∈ set (insert_before_list v ref xs) ⟷ x = v ∨ x ∈ set xs"
  ⟨proof⟩

lemma insert_before_list_distinct: "x ∉ set xs ⟹ distinct xs ⟹ distinct (insert_before_list x ref
xs)"
  ⟨proof⟩

lemma insert_before_list_subset: "set xs ⊆ set (insert_before_list x ref xs)"
  ⟨proof⟩

lemma insert_before_list_node_in_set: "x ∈ set (insert_before_list x ref xs)"
  ⟨proof⟩

lemma insert_node_writes:
  "writes (set_child_nodes_locs ptr) (a_insert_node ptr new_child reference_child_opt) h h'"
  ⟨proof⟩

lemma ensure_pre_insertion_validity_pure [simp]:
  "pure (a_ensure_pre_insertion_validity node ptr child) h"
  ⟨proof⟩
```

```
lemma insert_before_reference_child_not_in_children:
  assumes "h ⊢ get_parent child →r Some parent"
    and "ptr ≠ parent"
    and "¬is_character_data_ptr_kind ptr"
    and "h ⊢ get_ancestors ptr →r ancestors"
    and "cast node ∉ set ancestors"
  shows "h ⊢ insert_before ptr node (Some child) →e NotFoundError"
⟨proof⟩

lemma insert_before_ptr_in_heap:
  assumes "h ⊢ ok (insert_before ptr node reference_child)"
  shows "ptr |∈| object_ptr_kinds h"
  ⟨proof⟩

lemma insert_before_child_in_heap:
  assumes "h ⊢ ok (insert_before ptr node reference_child)"
  shows "node |∈| node_ptr_kinds h"
  ⟨proof⟩

lemma insert_node_children_remain_distinct:
    assumes insert_node: "h ⊢ a_insert_node ptr new_child reference_child_opt →h h2"
    and "h ⊢ get_child_nodes ptr →r children"
    and "new_child ∉ set children"
    and "⋀ptr children. h ⊢ get_child_nodes ptr →r children ⟹ distinct children"
    and known_ptr: "known_ptr ptr"
    and type_wf: "type_wf h"
  shows "⋀ptr children. h2 ⊢ get_child_nodes ptr →r children ⟹ distinct children"
⟨proof⟩

lemma insert_before_writes:
  "writes (insert_before_locs ptr |h ⊢ get_parent child|r
      |h ⊢ get_owner_document (cast child)|r |h ⊢ get_owner_document ptr|r) (insert_before ptr child ref)
h h'"
  ⟨proof⟩
end


locale l_append_childCore_DOM =
  l_append_child_defs +
  l_append_childCore_DOM_defs +
  assumes append_child_impl: "append_child = a_append_child"
begin

lemmas append_child_def = a_append_child_def[folded append_child_impl]
end

locale l_insert_before = l_insert_before_defs

locale l_append_child = l_append_child_defs

global_interpretation l_insert_beforeCore_DOM_defs get_parent get_parent_locs get_child_nodes
   get_child_nodes_locs set_child_nodes set_child_nodes_locs get_ancestors get_ancestors_locs
   adopt_node adopt_node_locs set_disconnected_nodes set_disconnected_nodes_locs
   get_disconnected_nodes get_disconnected_nodes_locs get_owner_document
  defines
    next_sibling = "l_insert_beforeCore_DOM_defs.a_next_sibling get_parent get_child_nodes" and
    insert_node = "l_insert_beforeCore_DOM_defs.a_insert_node get_child_nodes set_child_nodes" and
    ensure_pre_insertion_validity = "l_insert_beforeCore_DOM_defs.a_ensure_pre_insertion_validity
                                                get_parent get_child_nodes get_ancestors" and
    insert_before = "l_insert_beforeCore_DOM_defs.a_insert_before get_parent get_child_nodes
                               set_child_nodes get_ancestors adopt_node set_disconnected_nodes
                               get_disconnected_nodes get_owner_document" and
    insert_before_locs = "l_insert_before$_{Core\_DOM}$_defs.a_insert_before_locs set_child_nodes_locs
                                                              adopt_node_locs set_disconnected_nodes_locs"
  ⟨proof⟩

global_interpretation l_append_child$_{Core\_DOM}$_defs insert_before
  defines append_child = "l_append_child$_{Core\_DOM}$_defs.a_append_child insert_before"
  ⟨proof⟩

interpretation
  i_insert_before?: l_insert_before$_{Core\_DOM}$ get_parent get_parent_locs get_child_nodes
  get_child_nodes_locs set_child_nodes set_child_nodes_locs get_ancestors get_ancestors_locs
  adopt_node adopt_node_locs set_disconnected_nodes set_disconnected_nodes_locs get_disconnected_nodes
  get_disconnected_nodes_locs get_owner_document insert_before insert_before_locs append_child
  type_wf known_ptr known_ptrs
  ⟨proof⟩
declare l_insert_before$_{Core\_DOM}$_axioms[instances]

interpretation i_append_child?: l_append_child$_{Core\_DOM}$ append_child insert_before insert_before_locs
  ⟨proof⟩
declare l_append_child$_{Core\_DOM}$_axioms[instances]
```

create_element

```
locale l_create_element$_{Core\_DOM}$_defs =
  l_get_disconnected_nodes_defs get_disconnected_nodes get_disconnected_nodes_locs +
  l_set_disconnected_nodes_defs set_disconnected_nodes set_disconnected_nodes_locs +
  l_set_tag_name_defs set_tag_name set_tag_name_locs
  for get_disconnected_nodes ::
      "(_) document_ptr ⇒ ((_) heap, exception, (_) node_ptr list) prog"
  and get_disconnected_nodes_locs ::
      "(_) document_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"
  and set_disconnected_nodes ::
      "(_) document_ptr ⇒ (_) node_ptr list ⇒ ((_) heap, exception, unit) prog"
  and set_disconnected_nodes_locs ::
      "(_) document_ptr ⇒ ((_) heap, exception, unit) prog set"
    and set_tag_name ::
      "(_) element_ptr ⇒ char list ⇒ ((_) heap, exception, unit) prog"
    and set_tag_name_locs ::
      "(_) element_ptr ⇒ ((_) heap, exception, unit) prog set"
begin
definition a_create_element :: "(_) document_ptr ⇒ tag_name ⇒ (_, (_) element_ptr) dom_prog"
  where
    "a_create_element document_ptr tag = do {
      new_element_ptr ← new_element;
      set_tag_name new_element_ptr tag;
      disc_nodes ← get_disconnected_nodes document_ptr;
      set_disconnected_nodes document_ptr (cast new_element_ptr # disc_nodes);
      return new_element_ptr
    }"
end

locale l_create_element_defs =
  fixes create_element :: "(_) document_ptr ⇒ tag_name ⇒ (_, (_) element_ptr) dom_prog"

global_interpretation l_create_element$_{Core\_DOM}$_defs get_disconnected_nodes get_disconnected_nodes_locs

                                                       set_disconnected_nodes set_disconnected_nodes_locs
  set_tag_name set_tag_name_locs
  defines
  create_element = "l_create_element$_{Core\_DOM}$_defs.a_create_element get_disconnected_nodes
                                                                     set_disconnected_nodes set_tag_name"
  ⟨proof⟩
```

```
locale l_create_element_Core_DOM =
  l_create_element_Core_DOM_defs get_disconnected_nodes get_disconnected_nodes_locs
  set_disconnected_nodes set_disconnected_nodes_locs set_tag_name set_tag_name_locs +
  l_get_disconnected_nodes type_wf get_disconnected_nodes get_disconnected_nodes_locs +
  l_set_tag_name type_wf set_tag_name set_tag_name_locs +
  l_create_element_defs create_element +
  l_known_ptr known_ptr
  for get_disconnected_nodes :: "(_) document_ptr ⇒ ((_) heap, exception, (_) node_ptr list) prog"
  and get_disconnected_nodes_locs :: "(_) document_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"
  and set_disconnected_nodes :: "(_) document_ptr ⇒ (_) node_ptr list ⇒ ((_) heap, exception, unit) prog"
  and set_disconnected_nodes_locs :: "(_) document_ptr ⇒ ((_) heap, exception, unit) prog set"
    and set_tag_name :: "(_) element_ptr ⇒ char list ⇒ ((_) heap, exception, unit) prog"
    and set_tag_name_locs :: "(_) element_ptr ⇒ ((_) heap, exception, unit) prog set"
  and type_wf :: "(_) heap ⇒ bool"
  and create_element :: "(_) document_ptr ⇒ char list ⇒ ((_) heap, exception, (_) element_ptr) prog"
  and known_ptr :: "(_) object_ptr ⇒ bool" +
  assumes known_ptr_impl: "known_ptr = a_known_ptr"
  assumes create_element_impl: "create_element = a_create_element"
begin
lemmas create_element_def = a_create_element_def[folded create_element_impl]

lemma create_element_document_in_heap:
  assumes "h ⊢ ok (create_element document_ptr tag)"
  shows "document_ptr |∈| document_ptr_kinds h"
⟨proof⟩

lemma create_element_known_ptr:
  assumes "h ⊢ create_element document_ptr tag →r new_element_ptr"
  shows "known_ptr (cast new_element_ptr)"
⟨proof⟩
end

locale l_create_element = l_create_element_defs

interpretation
  i_create_element?: l_create_element_Core_DOM get_disconnected_nodes get_disconnected_nodes_locs
  set_disconnected_nodes set_disconnected_nodes_locs set_tag_name set_tag_name_locs type_wf
  create_element known_ptr
  ⟨proof⟩
declare l_create_element_Core_DOM_axioms[instances]
```

create_character_data

```
locale l_create_character_data_Core_DOM_defs =
  l_set_val_defs set_val set_val_locs +
  l_get_disconnected_nodes_defs get_disconnected_nodes get_disconnected_nodes_locs +
  l_set_disconnected_nodes_defs set_disconnected_nodes set_disconnected_nodes_locs
  for set_val :: "(_) character_data_ptr ⇒ char list ⇒ ((_) heap, exception, unit) prog"
  and set_val_locs :: "(_) character_data_ptr ⇒ ((_) heap, exception, unit) prog set"
  and get_disconnected_nodes :: "(_) document_ptr ⇒ ((_) heap, exception, (_) node_ptr list) prog"
  and get_disconnected_nodes_locs :: "(_) document_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"
  and set_disconnected_nodes :: "(_) document_ptr ⇒ (_) node_ptr list ⇒ ((_) heap, exception, unit) prog"
  and set_disconnected_nodes_locs :: "(_) document_ptr ⇒ ((_) heap, exception, unit) prog set"
begin
definition a_create_character_data :: "(_) document_ptr ⇒ string ⇒ (_, (_) character_data_ptr) dom_prog"
  where
    "a_create_character_data document_ptr text = do {
      new_character_data_ptr ← new_character_data;
      set_val new_character_data_ptr text;
      disc_nodes ← get_disconnected_nodes document_ptr;
      set_disconnected_nodes document_ptr (cast new_character_data_ptr # disc_nodes);
      return new_character_data_ptr
    }"
```

```
end

locale l_create_character_data_defs =
  fixes create_character_data :: "(_) document_ptr ⇒ string ⇒ (_, (_) character_data_ptr) dom_prog"

global_interpretation l_create_character_data_Core_DOM_defs set_val set_val_locs get_disconnected_nodes

                    get_disconnected_nodes_locs set_disconnected_nodes set_disconnected_nodes_locs
  defines create_character_data = "l_create_character_data_Core_DOM_defs.a_create_character_data
                                        set_val get_disconnected_nodes set_disconnected_nodes"
  ⟨proof⟩

locale l_create_character_data_Core_DOM =
  l_create_character_data_Core_DOM_defs set_val set_val_locs get_disconnected_nodes
  get_disconnected_nodes_locs set_disconnected_nodes set_disconnected_nodes_locs +
  l_get_disconnected_nodes type_wf get_disconnected_nodes get_disconnected_nodes_locs +
  l_set_val type_wf set_val set_val_locs +
  l_create_character_data_defs create_character_data +
  l_known_ptr known_ptr
  for get_disconnected_nodes :: "(_) document_ptr ⇒ ((_) heap, exception, (_) node_ptr list) prog"
  and get_disconnected_nodes_locs :: "(_) document_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"
  and set_disconnected_nodes :: "(_) document_ptr ⇒ (_) node_ptr list ⇒ ((_) heap, exception, unit) prog"
  and set_disconnected_nodes_locs :: "(_) document_ptr ⇒ ((_) heap, exception, unit) prog set"
  and set_val :: "(_) character_data_ptr ⇒ char list ⇒ ((_) heap, exception, unit) prog"
  and set_val_locs :: "(_) character_data_ptr ⇒ ((_) heap, exception, unit) prog set"
  and type_wf :: "(_) heap ⇒ bool"
  and create_character_data :: "(_) document_ptr ⇒ char list ⇒ ((_) heap, exception, (_) character_data_ptr)
prog"
  and known_ptr :: "(_) object_ptr ⇒ bool" +
  assumes known_ptr_impl: "known_ptr = a_known_ptr"
  assumes create_character_data_impl: "create_character_data = a_create_character_data"
begin
lemmas create_character_data_def = a_create_character_data_def[folded create_character_data_impl]

lemma create_character_data_document_in_heap:
  assumes "h ⊢ ok (create_character_data document_ptr text)"
  shows "document_ptr |∈| document_ptr_kinds h"
⟨proof⟩

lemma create_character_data_known_ptr:
  assumes "h ⊢ create_character_data document_ptr text →r new_character_data_ptr"
  shows "known_ptr (cast new_character_data_ptr)"
⟨proof⟩
end

locale l_create_character_data = l_create_character_data_defs

interpretation
  i_create_character_data?: l_create_character_data_Core_DOM get_disconnected_nodes
  get_disconnected_nodes_locs set_disconnected_nodes set_disconnected_nodes_locs set_val set_val_locs
  type_wf create_character_data known_ptr
  ⟨proof⟩
declare l_create_character_data_Core_DOM_axioms [instances]
```

create_character_data

```
locale l_create_document_Core_DOM_defs
begin
definition a_create_document :: "(_, (_) document_ptr) dom_prog"
  where
    "a_create_document = new_document"
end
```

```
locale l_create_document_defs =
  fixes create_document :: "(_, (_) document_ptr) dom_prog"

global_interpretation l_create_document_Core_DOM_defs
  defines create_document = "l_create_document_Core_DOM_defs.a_create_document"
  ⟨proof⟩

locale l_create_document_Core_DOM =
  l_create_document_Core_DOM_defs +
  l_create_document_defs +
  assumes create_document_impl: "create_document = a_create_document"
begin
lemmas
  create_document_def = create_document_impl[unfolded create_document_def, unfolded a_create_document_def]
end

locale l_create_document = l_create_document_defs

interpretation
  i_create_document?: l_create_document_Core_DOM create_document
  ⟨proof⟩
declare l_create_document_Core_DOM_axioms [instances]
```

tree_order

```
locale l_to_tree_order_Core_DOM_defs =
  l_get_child_nodes_defs get_child_nodes get_child_nodes_locs
  for get_child_nodes :: "(_::linorder) object_ptr ⇒ ((_) heap, exception, (_) node_ptr list) prog"
  and get_child_nodes_locs :: "(_) object_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"
begin
partial_function (dom_prog) a_to_tree_order :: "(_) object_ptr ⇒ (_, (_) object_ptr list) dom_prog"
  where
    "a_to_tree_order ptr = (do {
      children ← get_child_nodes ptr;
      treeorders ← map_M a_to_tree_order (map (cast) children);
      return (ptr # concat treeorders)
    })"
end

locale l_to_tree_order_defs =
  fixes to_tree_order :: "(_) object_ptr ⇒ (_, (_) object_ptr list) dom_prog"

global_interpretation l_to_tree_order_Core_DOM_defs get_child_nodes get_child_nodes_locs defines
  to_tree_order = "l_to_tree_order_Core_DOM_defs.a_to_tree_order get_child_nodes" ⟨proof⟩
declare a_to_tree_order.simps [code]

locale l_to_tree_order_Core_DOM =
  l_get_child_nodes type_wf known_ptr get_child_nodes get_child_nodes_locs +
  l_to_tree_order_Core_DOM_defs get_child_nodes get_child_nodes_locs +
  l_to_tree_order_defs to_tree_order
  for known_ptr :: "(_::linorder) object_ptr ⇒ bool"
  and type_wf :: "(_) heap ⇒ bool"
  and get_child_nodes :: "(_) object_ptr ⇒ ((_) heap, exception, (_) node_ptr list) prog"
  and get_child_nodes_locs :: "(_) object_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"
  and to_tree_order :: "(_) object_ptr ⇒ ((_) heap, exception, (_) object_ptr list) prog" +
  assumes to_tree_order_impl: "to_tree_order = a_to_tree_order"
begin
lemmas to_tree_order_def = a_to_tree_order.simps[folded to_tree_order_impl]

lemma to_tree_order_pure [simp]: "pure (to_tree_order ptr) h"
⟨proof⟩
end
```

```
locale l_to_tree_order =
  fixes to_tree_order :: "(_) object_ptr ⇒ (_, (_) object_ptr list) dom_prog"
  assumes to_tree_order_pure [simp]: "pure (to_tree_order ptr) h"

interpretation
  i_to_tree_order?: l_to_tree_order_Core_DOM known_ptr type_wf get_child_nodes get_child_nodes_locs
                    to_tree_order
  ⟨proof⟩
declare l_to_tree_order_Core_DOM_axioms[instances]

lemma to_tree_order_is_l_to_tree_order [instances]: "l_to_tree_order to_tree_order"
  ⟨proof⟩
```

first_in_tree_order

```
locale l_first_in_tree_order_Core_DOM_defs =
  l_to_tree_order_defs to_tree_order
  for to_tree_order :: "(_) object_ptr ⇒ ((_) heap, exception, (_) object_ptr list) prog"
begin
definition a_first_in_tree_order :: "(_) object_ptr ⇒ ((_) object_ptr
                                   ⇒ (_, 'result option) dom_prog) ⇒ (_, 'result option) dom_prog"
  where
    "a_first_in_tree_order ptr f = (do {
      tree_order ← to_tree_order ptr;
      results ← map_filter_M f tree_order;
      (case results of
        [] ⇒ return None
      | x#_⇒ return (Some x))
    })"
end

locale l_first_in_tree_order_defs =
  fixes first_in_tree_order :: "(_) object_ptr ⇒ ((_) object_ptr ⇒ (_, 'result option) dom_prog)
                                              ⇒ (_, 'result option) dom_prog"

global_interpretation l_first_in_tree_order_Core_DOM_defs to_tree_order defines
  first_in_tree_order = "l_first_in_tree_order_Core_DOM_defs.a_first_in_tree_order to_tree_order" ⟨proof⟩

locale l_first_in_tree_order_Core_DOM =
  l_first_in_tree_order_Core_DOM_defs to_tree_order +
  l_first_in_tree_order_defs first_in_tree_order
  for to_tree_order :: "(_) object_ptr ⇒ ((_) heap, exception, (_) object_ptr list) prog"
  and first_in_tree_order :: "(_) object_ptr ⇒ ((_) object_ptr ⇒ ((_) heap, exception, 'result option)
prog)
                                            ⇒ ((_) heap, exception, 'result option) prog" +
assumes first_in_tree_order_impl: "first_in_tree_order = a_first_in_tree_order"
begin
lemmas first_in_tree_order_def = first_in_tree_order_impl[unfolded a_first_in_tree_order_def]
end

locale l_first_in_tree_order

interpretation i_first_in_tree_order?:
  l_first_in_tree_order_Core_DOM to_tree_order first_in_tree_order
  ⟨proof⟩
declare l_first_in_tree_order_Core_DOM_axioms[instances]
```

get_element_by

```
locale l_get_element_by_Core_DOM_defs =
  l_first_in_tree_order_defs first_in_tree_order +
  l_to_tree_order_defs to_tree_order +
  l_get_attribute_defs get_attribute get_attribute_locs
  for to_tree_order :: "(_::linorder) object_ptr ⇒ ((_) heap, exception, (_) object_ptr list) prog"
  and first_in_tree_order :: "(_) object_ptr ⇒ ((_) object_ptr
                              ⇒ ((_) heap, exception, (_) element_ptr option) prog)
                              ⇒ ((_) heap, exception, (_) element_ptr option) prog"
  and get_attribute :: "(_) element_ptr ⇒ char list ⇒ ((_) heap, exception, char list option) prog"
  and get_attribute_locs :: "(_) element_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"
begin
definition a_get_element_by_id :: "(_) object_ptr ⇒ attr_value ⇒ (_, (_) element_ptr option) dom_prog"
  where
    "a_get_element_by_id ptr iden = first_in_tree_order ptr (λptr. (case cast ptr of
      Some element_ptr ⇒ do {
        id_opt ← get_attribute element_ptr ''id'';
        (if id_opt = Some iden then return (Some element_ptr) else return None)
      }
    | _ ⇒ return None
    ))"

definition a_get_elements_by_class_name :: "(_) object_ptr ⇒ attr_value ⇒ (_, (_) element_ptr list) dom_prog"
  where
    "a_get_elements_by_class_name ptr class_name = to_tree_order ptr ≫=
      map_filter_M (λptr. (case cast ptr of
        Some element_ptr ⇒ do {
          class_name_opt ← get_attribute element_ptr ''class'';
          (if class_name_opt = Some class_name then return (Some element_ptr) else return None)
        }
      | _ ⇒ return None))"

definition a_get_elements_by_tag_name :: "(_) object_ptr ⇒ attr_value ⇒ (_, (_) element_ptr list) dom_prog"
  where
    "a_get_elements_by_tag_name ptr tag = to_tree_order ptr ≫=
      map_filter_M (λptr. (case cast ptr of
        Some element_ptr ⇒ do {
          this_tag_name ← get_M element_ptr tag_name;
          (if this_tag_name = tag then return (Some element_ptr) else return None)
        }
      | _ ⇒ return None))"
end

locale l_get_element_by_defs =
  fixes get_element_by_id :: "(_) object_ptr ⇒ attr_value ⇒ (_, (_) element_ptr option) dom_prog"
  fixes get_elements_by_class_name :: "(_) object_ptr ⇒ attr_value ⇒ (_, (_) element_ptr list) dom_prog"
  fixes get_elements_by_tag_name :: "(_) object_ptr ⇒ attr_value ⇒ (_, (_) element_ptr list) dom_prog"

global_interpretation
l_get_element_by_Core_DOM_defs to_tree_order first_in_tree_order get_attribute get_attribute_locs
defines
  get_element_by_id = "l_get_element_by_Core_DOM_defs.a_get_element_by_id first_in_tree_order get_attribute"

and
    get_elements_by_class_name = "l_get_element_by_Core_DOM_defs.a_get_elements_by_class_name
to_tree_order get_attribute"
and
  get_elements_by_tag_name = "l_get_element_by_Core_DOM_defs.a_get_elements_by_tag_name to_tree_order" ⟨proof⟩

locale l_get_element_by_Core_DOM =
  l_get_element_by_Core_DOM_defs to_tree_order first_in_tree_order get_attribute get_attribute_locs +
  l_get_element_by_defs get_element_by_id get_elements_by_class_name get_elements_by_tag_name +
  l_first_in_tree_order_Core_DOM to_tree_order first_in_tree_order +
  l_to_tree_order to_tree_order +
  l_get_attribute type_wf get_attribute get_attribute_locs
  for to_tree_order :: "(_::linorder) object_ptr ⇒ ((_) heap, exception, (_) object_ptr list) prog"
    and first_in_tree_order ::
    "(_) object_ptr ⇒ ((_) object_ptr ⇒ ((_) heap, exception, (_) element_ptr option) prog)
```

```
                                         ⇒ ((_) heap, exception, (_) element_ptr option) prog"
  and get_attribute :: "(_) element_ptr ⇒ char list ⇒ ((_) heap, exception, char list option) prog"
  and get_attribute_locs :: "(_) element_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"
    and get_element_by_id ::
    "(_) object_ptr ⇒ char list ⇒ ((_) heap, exception, (_) element_ptr option) prog"
    and get_elements_by_class_name ::
    "(_) object_ptr ⇒ char list ⇒ ((_) heap, exception, (_) element_ptr list) prog"
    and get_elements_by_tag_name ::
    "(_) object_ptr ⇒ char list ⇒ ((_) heap, exception, (_) element_ptr list) prog"
  and type_wf :: "(_) heap ⇒ bool" +
  assumes get_element_by_id_impl: "get_element_by_id = a_get_element_by_id"
  assumes get_elements_by_class_name_impl: "get_elements_by_class_name = a_get_elements_by_class_name"
  assumes get_elements_by_tag_name_impl: "get_elements_by_tag_name = a_get_elements_by_tag_name"
begin
lemmas
  get_element_by_id_def = get_element_by_id_impl[unfolded a_get_element_by_id_def]
lemmas
  get_elements_by_class_name_def = get_elements_by_class_name_impl[unfolded a_get_elements_by_class_name_def]
lemmas
  get_elements_by_tag_name_def = get_elements_by_tag_name_impl[unfolded a_get_elements_by_tag_name_def]

lemma get_element_by_id_result_in_tree_order:
  assumes "h ⊢ get_element_by_id ptr iden →r Some element_ptr"
  assumes "h ⊢ to_tree_order ptr →r to"
  shows "cast element_ptr ∈ set to"
  ⟨proof⟩

lemma get_elements_by_class_name_result_in_tree_order:
  assumes "h ⊢ get_elements_by_class_name ptr name →r results"
  assumes "h ⊢ to_tree_order ptr →r to"
  assumes "element_ptr ∈ set results"
  shows "cast element_ptr ∈ set to"
  ⟨proof⟩

lemma get_elements_by_tag_name_result_in_tree_order:
  assumes "h ⊢ get_elements_by_tag_name ptr name →r results"
  assumes "h ⊢ to_tree_order ptr →r to"
  assumes "element_ptr ∈ set results"
  shows "cast element_ptr ∈ set to"
  ⟨proof⟩

lemma get_elements_by_tag_name_pure [simp]: "pure (get_elements_by_tag_name ptr tag) h"
  ⟨proof⟩
end

locale l_get_element_by = l_get_element_by_defs + l_to_tree_order_defs +
  assumes get_element_by_id_result_in_tree_order:
          "h ⊢ get_element_by_id ptr iden →r Some element_ptr ⟹ h ⊢ to_tree_order ptr →r to
               ⟹ cast element_ptr ∈ set to"
  assumes get_elements_by_tag_name_pure [simp]: "pure (get_elements_by_tag_name ptr tag) h"

interpretation
  i_get_element_by?: l_get_element_by_Core_DOM to_tree_order first_in_tree_order get_attribute
                     get_attribute_locs get_element_by_id get_elements_by_class_name
                     get_elements_by_tag_name type_wf
  ⟨proof⟩
declare l_get_element_by_Core_DOM_axioms[instances]

lemma get_element_by_is_l_get_element_by [instances]:
  "l_get_element_by get_element_by_id get_elements_by_tag_name to_tree_order"
  ⟨proof⟩
end
```

6.3 Wellformedness (Core_DOM_Heap_WF)

In this theory, we discuss the wellformedness of the DOM. First, we define wellformedness and, second, we show for all functions for querying and modifying the DOM to what extend they preserve wellformendess.

```
theory Core_DOM_Heap_WF
  imports
    "Core_DOM_Functions"
begin

locale l_heap_is_wellformed_Core_DOM_defs =
  l_get_child_nodes_defs get_child_nodes get_child_nodes_locs +
  l_get_disconnected_nodes_defs get_disconnected_nodes get_disconnected_nodes_locs
  for get_child_nodes :: "(_::linorder) object_ptr ⇒ ((_) heap, exception, (_) node_ptr list) prog"
    and get_child_nodes_locs :: "(_) object_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"
    and get_disconnected_nodes :: "(_) document_ptr ⇒ ((_) heap, exception, (_) node_ptr list) prog"
    and get_disconnected_nodes_locs :: "(_) document_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"
begin
definition a_owner_document_valid :: "(_) heap ⇒ bool"
  where
    "a_owner_document_valid h ⟷ (∀node_ptr ∈ fset (node_ptr_kinds h).
      ((∃document_ptr. document_ptr |∈| document_ptr_kinds h
         ∧ node_ptr ∈ set |h ⊢ get_disconnected_nodes document_ptr|_r)
     ∨ (∃parent_ptr. parent_ptr |∈| object_ptr_kinds h
           ∧ node_ptr ∈ set |h ⊢ get_child_nodes parent_ptr|_r)))"

lemma a_owner_document_valid_code [code]: "a_owner_document_valid h ⟷ node_ptr_kinds h |⊆|
  fset_of_list (concat (map (λparent. |h ⊢ get_child_nodes parent|_r)
(sorted_list_of_fset (object_ptr_kinds h)) @ map (λparent. |h ⊢ get_disconnected_nodes parent|_r)
(sorted_list_of_fset (document_ptr_kinds h))))
"
  ⟨proof⟩

definition a_parent_child_rel :: "(_) heap ⇒ ((_) object_ptr × (_) object_ptr) set"
  where
    "a_parent_child_rel h = {(parent, child). parent |∈| object_ptr_kinds h
                              ∧ child ∈ cast ` set |h ⊢ get_child_nodes parent|_r}"

lemma a_parent_child_rel_code [code]: "a_parent_child_rel h = set (concat (map
  (λparent. map
    (λchild. (parent, cast_node_ptr2object_ptr child))
    |h ⊢ get_child_nodes parent|_r)
  (sorted_list_of_fset (object_ptr_kinds h)))
)"
  ⟨proof⟩

definition a_acyclic_heap :: "(_) heap ⇒ bool"
  where
    "a_acyclic_heap h = acyclic (a_parent_child_rel h)"

definition a_all_ptrs_in_heap :: "(_) heap ⇒ bool"
  where
    "a_all_ptrs_in_heap h ⟷
      (∀ptr ∈ fset (object_ptr_kinds h). set |h ⊢ get_child_nodes ptr|_r ⊆ fset (node_ptr_kinds h)) ∧
      (∀document_ptr ∈ fset (document_ptr_kinds h).
set |h ⊢ get_disconnected_nodes document_ptr|_r ⊆ fset (node_ptr_kinds h))"

definition a_distinct_lists :: "(_) heap ⇒ bool"
  where
    "a_distinct_lists h = distinct (concat (
      (map (λptr. |h ⊢ get_child_nodes ptr|_r) |h ⊢ object_ptr_kinds_M|_r)
   @ (map (λdocument_ptr. |h ⊢ get_disconnected_nodes document_ptr|_r) |h ⊢ document_ptr_kinds_M|_r)
    ))"
```

```
definition a_heap_is_wellformed :: "(_) heap ⇒ bool"
  where
    "a_heap_is_wellformed h ⟷
      a_acyclic_heap h ∧ a_all_ptrs_in_heap h ∧ a_distinct_lists h ∧ a_owner_document_valid h"
end

locale l_heap_is_wellformed_defs =
  fixes heap_is_wellformed :: "(_) heap ⇒ bool"
  fixes parent_child_rel :: "(_) heap ⇒ ((_) object_ptr × (_) object_ptr) set"

global_interpretation l_heap_is_wellformed_Core_DOM_defs get_child_nodes get_child_nodes_locs
  get_disconnected_nodes get_disconnected_nodes_locs
  defines heap_is_wellformed = "l_heap_is_wellformed_Core_DOM_defs.a_heap_is_wellformed get_child_nodes
                   get_disconnected_nodes"
    and parent_child_rel = "l_heap_is_wellformed_Core_DOM_defs.a_parent_child_rel get_child_nodes"
    and acyclic_heap = a_acyclic_heap
    and all_ptrs_in_heap = a_all_ptrs_in_heap
    and distinct_lists = a_distinct_lists
    and owner_document_valid = a_owner_document_valid
  ⟨proof⟩

locale l_heap_is_wellformed_Core_DOM =
  l_get_child_nodes type_wf known_ptr get_child_nodes get_child_nodes_locs
  + l_heap_is_wellformed_Core_DOM_defs get_child_nodes get_child_nodes_locs get_disconnected_nodes
  get_disconnected_nodes_locs
  + l_heap_is_wellformed_defs heap_is_wellformed parent_child_rel
  + l_get_disconnected_nodes type_wf get_disconnected_nodes get_disconnected_nodes_locs
  for known_ptr :: "(_::linorder) object_ptr ⇒ bool"
    and type_wf :: "(_) heap ⇒ bool"
    and get_child_nodes :: "(_) object_ptr ⇒ ((_) heap, exception, (_) node_ptr list) prog"
    and get_child_nodes_locs :: "(_) object_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"
    and get_disconnected_nodes :: "(_) document_ptr ⇒ ((_) heap, exception, (_) node_ptr list) prog"
    and get_disconnected_nodes_locs :: "(_) document_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"
    and heap_is_wellformed :: "(_) heap ⇒ bool"
    and parent_child_rel :: "(_) heap ⇒ ((_) object_ptr × (_) object_ptr) set" +
  assumes heap_is_wellformed_impl: "heap_is_wellformed = a_heap_is_wellformed"
  assumes parent_child_rel_impl: "parent_child_rel = a_parent_child_rel"
begin
lemmas heap_is_wellformed_def = heap_is_wellformed_impl[unfolded a_heap_is_wellformed_def]
lemmas parent_child_rel_def = parent_child_rel_impl[unfolded a_parent_child_rel_def]
lemmas acyclic_heap_def = a_acyclic_heap_def[folded parent_child_rel_impl]

lemma parent_child_rel_node_ptr:
  "(parent, child) ∈ parent_child_rel h ⟹ is_node_ptr_kind child"
  ⟨proof⟩

lemma parent_child_rel_child_nodes:
  assumes "known_ptr parent"
    and "h ⊢ get_child_nodes parent →_r children"
    and "child ∈ set children"
  shows "(parent, cast child) ∈ parent_child_rel h"
  ⟨proof⟩

lemma parent_child_rel_child_nodes2:
  assumes "known_ptr parent"
    and "h ⊢ get_child_nodes parent →_r children"
    and "child ∈ set children"
    and "cast_node_ptr2object_ptr child = child_obj"
  shows "(parent, child_obj) ∈ parent_child_rel h"
  ⟨proof⟩
```

```
lemma parent_child_rel_finite: "finite (parent_child_rel h)"
⟨proof⟩

lemma distinct_lists_no_parent:
  assumes "a_distinct_lists h"
  assumes "h ⊢ get_disconnected_nodes document_ptr →r disc_nodes"
  assumes "node_ptr ∈ set disc_nodes"
  shows "¬(∃parent_ptr. parent_ptr |∈| object_ptr_kinds h
                ∧ node_ptr ∈ set |h ⊢ get_child_nodes parent_ptr|r)"
  ⟨proof⟩


lemma distinct_lists_disconnected_nodes:
  assumes "a_distinct_lists h"
    and "h ⊢ get_disconnected_nodes document_ptr →r disc_nodes"
  shows "distinct disc_nodes"
⟨proof⟩

lemma distinct_lists_children:
  assumes "a_distinct_lists h"
    and "known_ptr ptr"
    and "h ⊢ get_child_nodes ptr →r children"
  shows "distinct children"
⟨proof⟩

lemma heap_is_wellformed_children_in_heap:
  assumes "heap_is_wellformed h"
  assumes "h ⊢ get_child_nodes ptr →r children"
  assumes "child ∈ set children"
  shows "child |∈| node_ptr_kinds h"
  ⟨proof⟩

lemma heap_is_wellformed_one_parent:
  assumes "heap_is_wellformed h"
  assumes "h ⊢ get_child_nodes ptr →r children"
  assumes "h ⊢ get_child_nodes ptr' →r children'"
  assumes "set children ∩ set children' ≠ {}"
  shows "ptr = ptr'"
  ⟨proof⟩

lemma parent_child_rel_child:
  "h ⊢ get_child_nodes ptr →r children ⟹
child ∈ set children ⟷ (ptr, cast child) ∈ parent_child_rel h"
  ⟨proof⟩

lemma parent_child_rel_acyclic: "heap_is_wellformed h ⟹ acyclic (parent_child_rel h)"
  ⟨proof⟩

lemma heap_is_wellformed_disconnected_nodes_distinct:
  "heap_is_wellformed h ⟹ h ⊢ get_disconnected_nodes document_ptr →r disc_nodes ⟹
distinct disc_nodes"
  ⟨proof⟩

lemma parent_child_rel_parent_in_heap:
  "(parent, child_ptr) ∈ parent_child_rel h ⟹ parent |∈| object_ptr_kinds h"
  ⟨proof⟩

lemma parent_child_rel_child_in_heap:
  "heap_is_wellformed h ⟹ type_wf h ⟹ known_ptr parent
    ⟹ (parent, child_ptr) ∈ parent_child_rel h ⟹ child_ptr |∈| object_ptr_kinds h"
  ⟨proof⟩

lemma heap_is_wellformed_disc_nodes_in_heap:
  "heap_is_wellformed h ⟹ h ⊢ get_disconnected_nodes document_ptr →r disc_nodes
   ⟹ node ∈ set disc_nodes ⟹ node |∈| node_ptr_kinds h"
  ⟨proof⟩

lemma heap_is_wellformed_one_disc_parent:
  "heap_is_wellformed h ⟹ h ⊢ get_disconnected_nodes document_ptr →r disc_nodes
   ⟹ h ⊢ get_disconnected_nodes document_ptr' →r disc_nodes'
   ⟹ set disc_nodes ∩ set disc_nodes' ≠ {} ⟹ document_ptr = document_ptr'"
  ⟨proof⟩

lemma heap_is_wellformed_children_disc_nodes_different:
  "heap_is_wellformed h ⟹ h ⊢ get_child_nodes ptr →r children
    ⟹ h ⊢ get_disconnected_nodes document_ptr →r disc_nodes
   ⟹ set children ∩ set disc_nodes = {}"
  ⟨proof⟩

lemma heap_is_wellformed_children_disc_nodes:
  "heap_is_wellformed h ⟹ node_ptr |∈| node_ptr_kinds h
   ⟹  ¬(∃parent ∈ fset (object_ptr_kinds h). node_ptr ∈ set |h ⊢ get_child_nodes parent|r)
   ⟹ (∃document_ptr ∈ fset (document_ptr_kinds h). node_ptr ∈ set |h ⊢ get_disconnected_nodes document_ptr|r)"
  ⟨proof⟩

lemma heap_is_wellformed_children_distinct:
  "heap_is_wellformed h ⟹ h ⊢ get_child_nodes ptr →r children ⟹ distinct children"
  ⟨proof⟩
end

locale l_heap_is_wellformed = l_type_wf + l_known_ptr + l_heap_is_wellformed_defs
  + l_get_child_nodes_defs + l_get_disconnected_nodes_defs +
  assumes heap_is_wellformed_children_in_heap:
    "heap_is_wellformed h ⟹ h ⊢ get_child_nodes ptr →r children ⟹ child ∈ set children
                      ⟹ child |∈| node_ptr_kinds h"
  assumes heap_is_wellformed_disc_nodes_in_heap:
    "heap_is_wellformed h ⟹ h ⊢ get_disconnected_nodes document_ptr →r disc_nodes
                      ⟹ node ∈ set disc_nodes ⟹ node |∈| node_ptr_kinds h"
  assumes heap_is_wellformed_one_parent:
    "heap_is_wellformed h ⟹ h ⊢ get_child_nodes ptr →r children
                      ⟹ h ⊢ get_child_nodes ptr' →r children'
                      ⟹ set children ∩ set children' ≠ {} ⟹ ptr = ptr'"
  assumes heap_is_wellformed_one_disc_parent:
    "heap_is_wellformed h ⟹ h ⊢ get_disconnected_nodes document_ptr →r disc_nodes
                      ⟹ h ⊢ get_disconnected_nodes document_ptr' →r disc_nodes'
                      ⟹ set disc_nodes ∩ set disc_nodes' ≠ {} ⟹ document_ptr = document_ptr'"
  assumes heap_is_wellformed_children_disc_nodes_different:
    "heap_is_wellformed h ⟹ h ⊢ get_child_nodes ptr →r children
                      ⟹ h ⊢ get_disconnected_nodes document_ptr →r disc_nodes
                      ⟹ set children ∩ set disc_nodes = {}"
  assumes heap_is_wellformed_disconnected_nodes_distinct:
    "heap_is_wellformed h ⟹ h ⊢ get_disconnected_nodes document_ptr →r disc_nodes
                      ⟹ distinct disc_nodes"
  assumes heap_is_wellformed_children_distinct:
    "heap_is_wellformed h ⟹ h ⊢ get_child_nodes ptr →r children ⟹ distinct children"
  assumes heap_is_wellformed_children_disc_nodes:
    "heap_is_wellformed h ⟹ node_ptr |∈| node_ptr_kinds h
   ⟹  ¬(∃parent ∈ fset (object_ptr_kinds h). node_ptr ∈ set |h ⊢ get_child_nodes parent|r)
   ⟹ (∃document_ptr ∈ fset (document_ptr_kinds h). node_ptr ∈ set |h ⊢ get_disconnected_nodes document_ptr|r)"
  assumes parent_child_rel_child:
    "h ⊢ get_child_nodes ptr →r children
   ⟹ child ∈ set children ⟷ (ptr, cast child) ∈ parent_child_rel h"
  assumes parent_child_rel_finite:
    "heap_is_wellformed h ⟹ finite (parent_child_rel h)"
  assumes parent_child_rel_acyclic:
    "heap_is_wellformed h ⟹ acyclic (parent_child_rel h)"
  assumes parent_child_rel_node_ptr:
    "(parent, child_ptr) ∈ parent_child_rel h ⟹ is_node_ptr_kind child_ptr"
  assumes parent_child_rel_parent_in_heap:
    "(parent, child_ptr) ∈ parent_child_rel h ⟹ parent |∈| object_ptr_kinds h"
  assumes parent_child_rel_child_in_heap:
    "heap_is_wellformed h ⟹ type_wf h ⟹ known_ptr parent
   ⟹ (parent, child_ptr) ∈ parent_child_rel h ⟹ child_ptr |∈| object_ptr_kinds h"

interpretation i_heap_is_wellformed?: l_heap_is_wellformed_Core_DOM known_ptr type_wf get_child_nodes
  get_child_nodes_locs get_disconnected_nodes get_disconnected_nodes_locs
  heap_is_wellformed parent_child_rel
  ⟨proof⟩
declare l_heap_is_wellformed_Core_DOM_axioms[instances]


lemma heap_is_wellformed_is_l_heap_is_wellformed [instances]:
  "l_heap_is_wellformed type_wf known_ptr heap_is_wellformed parent_child_rel get_child_nodes
                       get_disconnected_nodes"
  ⟨proof⟩
```

6.3.1 get_parent

```
locale l_get_parent_wf_Core_DOM =
  l_get_parent_Core_DOM
  known_ptr type_wf get_child_nodes get_child_nodes_locs known_ptrs get_parent get_parent_locs
  + l_heap_is_wellformed
  type_wf known_ptr heap_is_wellformed parent_child_rel get_child_nodes get_child_nodes_locs
  get_disconnected_nodes get_disconnected_nodes_locs
  for known_ptr :: "(_::linorder) object_ptr ⇒ bool"
    and type_wf :: "(_) heap ⇒ bool"
    and get_child_nodes :: "(_) object_ptr ⇒ ((_) heap, exception, (_) node_ptr list) prog"
    and get_child_nodes_locs :: "(_) object_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"
    and known_ptrs :: "(_) heap ⇒ bool"
    and get_parent :: "(_) node_ptr ⇒ ((_) heap, exception, (_) object_ptr option) prog"
    and get_parent_locs :: "((_) heap ⇒ (_) heap ⇒ bool) set"
    and heap_is_wellformed :: "(_) heap ⇒ bool"
    and parent_child_rel :: "(_) heap ⇒ ((_) object_ptr × (_) object_ptr) set"
    and get_disconnected_nodes :: "(_) document_ptr ⇒ ((_) heap, exception, (_) node_ptr list) prog"
    and get_disconnected_nodes_locs :: "(_) document_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"
begin
lemma child_parent_dual:
  assumes heap_is_wellformed: "heap_is_wellformed h"
  assumes "h ⊢ get_child_nodes ptr →r children"
  assumes "child ∈ set children"
  assumes "known_ptrs h"
  assumes type_wf: "type_wf h"
  shows "h ⊢ get_parent child →r Some ptr"
⟨proof⟩

lemma parent_child_rel_parent:
  assumes "heap_is_wellformed h"
    and "h ⊢ get_parent child_node →r Some parent"
  shows "(parent, cast child_node) ∈ parent_child_rel h"
  ⟨proof⟩

lemma heap_wellformed_induct [consumes 1, case_names step]:
  assumes "heap_is_wellformed h"
    and step: "⋀parent. (⋀children child. h ⊢ get_child_nodes parent →r children
              ⟹ child ∈ set children ⟹ P (cast child)) ⟹ P parent"
  shows "P ptr"
⟨proof⟩

lemma heap_wellformed_induct2 [consumes 3, case_names not_in_heap empty_children step]:
  assumes "heap_is_wellformed h" and "type_wf h" and "known_ptrs h"
    and not_in_heap: "⋀parent. parent |∉| object_ptr_kinds h ⟹ P parent"
    and empty_children: "⋀parent. h ⊢ get_child_nodes parent →r [] ⟹ P parent"
    and step: "⋀parent children child. h ⊢ get_child_nodes parent →r children
            ⟹ child ∈ set children ⟹ P (cast child) ⟹ P parent"
  shows "P ptr"
⟨proof⟩

lemma heap_wellformed_induct_rev [consumes 1, case_names step]:
  assumes "heap_is_wellformed h"
    and step: "⋀child. (⋀parent child_node. cast child_node = child
            ⟹ h ⊢ get_parent child_node →r Some parent ⟹ P parent) ⟹ P child"
  shows "P ptr"
⟨proof⟩
end

interpretation i_get_parent_wf?: l_get_parent_wf_Core_DOM known_ptr type_wf get_child_nodes
  get_child_nodes_locs known_ptrs get_parent get_parent_locs heap_is_wellformed
  parent_child_rel get_disconnected_nodes
  ⟨proof⟩
declare l_get_parent_wf_Core_DOM_axioms[instances]


locale l_get_parent_wf2_Core_DOM =
  l_get_parent_wf_Core_DOM
  known_ptr type_wf get_child_nodes get_child_nodes_locs known_ptrs get_parent get_parent_locs
  heap_is_wellformed parent_child_rel get_disconnected_nodes get_disconnected_nodes_locs
  + l_heap_is_wellformed_Core_DOM
  known_ptr type_wf get_child_nodes get_child_nodes_locs get_disconnected_nodes
  get_disconnected_nodes_locs heap_is_wellformed parent_child_rel
  for known_ptr :: "(_::linorder) object_ptr ⇒ bool"
    and type_wf :: "(_) heap ⇒ bool"
    and get_child_nodes :: "(_) object_ptr ⇒ ((_) heap, exception, (_) node_ptr list) prog"
    and get_child_nodes_locs :: "(_) object_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"
    and known_ptrs :: "(_) heap ⇒ bool"
    and get_parent :: "(_) node_ptr ⇒ ((_) heap, exception, (_) object_ptr option) prog"
    and get_parent_locs :: "((_) heap ⇒ (_) heap ⇒ bool) set"
    and heap_is_wellformed :: "(_) heap ⇒ bool"
    and parent_child_rel :: "(_) heap ⇒ ((_) object_ptr × (_) object_ptr) set"
    and get_disconnected_nodes :: "(_) document_ptr ⇒ ((_) heap, exception, (_) node_ptr list) prog"
    and get_disconnected_nodes_locs :: "(_) document_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"
begin
lemma preserves_wellformedness_writes_needed:
  assumes heap_is_wellformed: "heap_is_wellformed h"
    and "h ⊢ f →h h'"
    and "writes SW f h h'"
    and preserved_get_child_nodes:
    "⋀h h' w. w ∈ SW ⟹ h ⊢ w →h h'
                  ⟹ ∀object_ptr. ∀r ∈ get_child_nodes_locs object_ptr. r h h'"
    and preserved_get_disconnected_nodes:
    "⋀h h' w. w ∈ SW ⟹ h ⊢ w →h h'
                  ⟹ ∀document_ptr. ∀r ∈ get_disconnected_nodes_locs document_ptr. r h h'"
    and preserved_object_pointers:
    "⋀h h' w. w ∈ SW ⟹ h ⊢ w →h h'
                  ⟹ ∀object_ptr. preserved (get_M_Object object_ptr RObject.nothing) h h'"
  shows "heap_is_wellformed h'"
⟨proof⟩
end

interpretation i_get_parent_wf2?: l_get_parent_wf2_Core_DOM known_ptr type_wf get_child_nodes
  get_child_nodes_locs known_ptrs get_parent get_parent_locs
  heap_is_wellformed parent_child_rel get_disconnected_nodes
  get_disconnected_nodes_locs
```

⟨*proof*⟩

```
declare l_get_parent_wf2$_{Core\_DOM}$_axioms[instances]
locale l_get_parent_wf = l_type_wf + l_known_ptrs  + l_heap_is_wellformed_defs
  + l_get_child_nodes_defs + l_get_parent_defs +
  assumes child_parent_dual:
    "heap_is_wellformed h
    ⟹ type_wf h
    ⟹ known_ptrs h
    ⟹ h ⊢ get_child_nodes ptr →r children
    ⟹ child ∈ set children
    ⟹ h ⊢ get_parent child →r Some ptr"
  assumes heap_wellformed_induct [consumes 1, case_names step]:
    "heap_is_wellformed h
    ⟹ (⋀parent. (⋀children child. h ⊢ get_child_nodes parent →r children
    ⟹ child ∈ set children ⟹ P (cast child)) ⟹ P parent)
    ⟹ P ptr"
  assumes heap_wellformed_induct_rev [consumes 1, case_names step]:
    "heap_is_wellformed h
    ⟹ (⋀child. (⋀parent child_node. cast child_node = child
    ⟹ h ⊢ get_parent child_node →r Some parent ⟹ P parent) ⟹ P child)
    ⟹ P ptr"
  assumes parent_child_rel_parent: "heap_is_wellformed h
    ⟹ h ⊢ get_parent child_node →r Some parent
    ⟹ (parent, cast child_node) ∈ parent_child_rel h"

lemma get_parent_wf_is_l_get_parent_wf [instances]:
  "l_get_parent_wf type_wf known_ptr known_ptrs heap_is_wellformed parent_child_rel
  get_child_nodes get_parent"
  ⟨proof⟩
```

6.3.2 get_disconnected_nodes

6.3.3 set_disconnected_nodes

get_disconnected_nodes

```
locale l_set_disconnected_nodes_get_disconnected_nodes_wf$_{Core\_DOM}$ =
  l_set_disconnected_nodes_get_disconnected_nodes
  type_wf get_disconnected_nodes get_disconnected_nodes_locs set_disconnected_nodes
  set_disconnected_nodes_locs
  + l_heap_is_wellformed
  type_wf known_ptr heap_is_wellformed parent_child_rel get_child_nodes get_child_nodes_locs
  get_disconnected_nodes get_disconnected_nodes_locs
  for known_ptr :: "(_) object_ptr ⇒ bool"
    and type_wf :: "(_) heap ⇒ bool"
    and get_disconnected_nodes :: "(_) document_ptr ⇒ ((_) heap, exception, (_) node_ptr list) prog"
    and get_disconnected_nodes_locs :: "(_) document_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"
    and set_disconnected_nodes :: "(_) document_ptr ⇒ (_) node_ptr list ⇒ ((_) heap, exception, unit)
prog"
    and set_disconnected_nodes_locs :: "(_) document_ptr ⇒ ((_) heap, exception, unit) prog set"
    and heap_is_wellformed :: "(_) heap ⇒ bool"
    and parent_child_rel :: "(_) heap ⇒ ((_) object_ptr × (_) object_ptr) set"
    and get_child_nodes :: "(_) object_ptr ⇒ ((_) heap, exception, (_) node_ptr list) prog"
    and get_child_nodes_locs :: "(_) object_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"
begin

lemma remove_from_disconnected_nodes_removes:
  assumes "heap_is_wellformed h"
  assumes "h ⊢ get_disconnected_nodes ptr →r disc_nodes"
  assumes "h ⊢ set_disconnected_nodes ptr (remove1 node_ptr disc_nodes) →h h'"
  assumes "h' ⊢ get_disconnected_nodes ptr →r disc_nodes'"
  shows "node_ptr ∉ set disc_nodes'"
  ⟨proof⟩
end

locale l_set_disconnected_nodes_get_disconnected_nodes_wf = l_heap_is_wellformed
  + l_set_disconnected_nodes_get_disconnected_nodes +
  assumes remove_from_disconnected_nodes_removes:
    "heap_is_wellformed h ⟹ h ⊢ get_disconnected_nodes ptr →r disc_nodes
                          ⟹ h ⊢ set_disconnected_nodes ptr (remove1 node_ptr disc_nodes) →h h'
                          ⟹ h' ⊢ get_disconnected_nodes ptr →r disc_nodes'
                          ⟹ node_ptr ∉ set disc_nodes'"

interpretation i_set_disconnected_nodes_get_disconnected_nodes_wfCore_DOM?:
  l_set_disconnected_nodes_get_disconnected_nodes_wfCore_DOM  known_ptr type_wf get_disconnected_nodes
  get_disconnected_nodes_locs set_disconnected_nodes set_disconnected_nodes_locs heap_is_wellformed
  parent_child_rel get_child_nodes
  ⟨proof⟩
declare l_set_disconnected_nodes_get_disconnected_nodes_wfCore_DOM_axioms[instances]

lemma set_disconnected_nodes_get_disconnected_nodes_wf_is_l_set_disconnected_nodes_get_disconnected_nodes_wf
[instances]:
  "l_set_disconnected_nodes_get_disconnected_nodes_wf type_wf known_ptr heap_is_wellformed parent_child_rel
   get_child_nodes get_disconnected_nodes get_disconnected_nodes_locs set_disconnected_nodes
      set_disconnected_nodes_locs"
  ⟨proof⟩
```

6.3.4 get_root_node

```
locale l_get_root_node_wfCore_DOM =
  l_heap_is_wellformed
  type_wf known_ptr heap_is_wellformed parent_child_rel get_child_nodes get_child_nodes_locs
  get_disconnected_nodes get_disconnected_nodes_locs
  + l_get_parentCore_DOM
  known_ptr type_wf get_child_nodes get_child_nodes_locs known_ptrs get_parent get_parent_locs
  + l_get_parent_wf
  type_wf known_ptr known_ptrs heap_is_wellformed parent_child_rel get_child_nodes
  get_child_nodes_locs get_parent get_parent_locs
  + l_get_root_nodeCore_DOM
  type_wf known_ptr known_ptrs get_parent get_parent_locs get_child_nodes get_child_nodes_locs
  get_ancestors get_ancestors_locs get_root_node get_root_node_locs
  for known_ptr :: "(_::linorder) object_ptr ⇒ bool"
    and type_wf :: "(_) heap ⇒ bool"
    and known_ptrs :: "(_) heap ⇒ bool"
    and heap_is_wellformed :: "(_) heap ⇒ bool"
    and parent_child_rel :: "(_) heap ⇒ ((_) object_ptr × (_) object_ptr) set"
    and get_child_nodes :: "(_) object_ptr ⇒ ((_) heap, exception, (_) node_ptr list) prog"
    and get_child_nodes_locs :: "(_) object_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"
    and get_disconnected_nodes :: "(_) document_ptr ⇒ ((_) heap, exception, (_) node_ptr list) prog"
    and get_disconnected_nodes_locs :: "(_) document_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"
    and get_parent :: "(_) node_ptr ⇒ ((_) heap, exception, (_) object_ptr option) prog"
    and get_parent_locs :: "((_) heap ⇒ (_) heap ⇒ bool) set"
    and get_ancestors :: "(_) object_ptr ⇒ ((_) heap, exception, (_) object_ptr list) prog"
    and get_ancestors_locs :: "((_) heap ⇒ (_) heap ⇒ bool) set"
    and get_root_node :: "(_) object_ptr ⇒ ((_) heap, exception, (_) object_ptr) prog"
    and get_root_node_locs :: "((_) heap ⇒ (_) heap ⇒ bool) set"

begin
lemma get_ancestors_reads:
  assumes "heap_is_wellformed h"
  shows "reads get_ancestors_locs (get_ancestors node_ptr) h h'"
⟨proof⟩

lemma get_ancestors_ok:
  assumes "heap_is_wellformed h"
```

```
    and "ptr |∈| object_ptr_kinds h"
    and "known_ptrs h"
    and type_wf: "type_wf h"
  shows "h ⊢ ok (get_ancestors ptr)"
⟨proof⟩

lemma get_root_node_ptr_in_heap:
  assumes "h ⊢ ok (get_root_node ptr)"
  shows "ptr |∈| object_ptr_kinds h"
  ⟨proof⟩


lemma get_root_node_ok:
  assumes "heap_is_wellformed h" "known_ptrs h" "type_wf h"
    and "ptr |∈| object_ptr_kinds h"
  shows "h ⊢ ok (get_root_node ptr)"
  ⟨proof⟩


lemma get_ancestors_parent:
  assumes "heap_is_wellformed h"
    and "h ⊢ get_parent child →r Some parent"
  shows "h ⊢ get_ancestors (cast child) →r (cast child) # parent # ancestors
     ⟷ h ⊢ get_ancestors parent →r parent # ancestors"
⟨proof⟩


lemma get_ancestors_never_empty:
  assumes "heap_is_wellformed h"
    and "h ⊢ get_ancestors child →r ancestors"
  shows "ancestors ≠ []"
⟨proof⟩



lemma get_ancestors_subset:
  assumes "heap_is_wellformed h"
    and "h ⊢ get_ancestors ptr →r ancestors"
    and "ancestor ∈ set ancestors"
    and "h ⊢ get_ancestors ancestor →r ancestor_ancestors"
    and type_wf: "type_wf h"
    and known_ptrs: "known_ptrs h"
  shows "set ancestor_ancestors ⊆ set ancestors"
⟨proof⟩

lemma get_ancestors_also_parent:
  assumes "heap_is_wellformed h"
    and "h ⊢ get_ancestors some_ptr →r ancestors"
    and "cast child ∈ set ancestors"
    and "h ⊢ get_parent child →r Some parent"
    and type_wf: "type_wf h"
    and known_ptrs: "known_ptrs h"
  shows "parent ∈ set ancestors"
⟨proof⟩

lemma get_ancestors_obtains_children:
  assumes "heap_is_wellformed h"
    and "ancestor ≠ ptr"
    and "ancestor ∈ set ancestors"
    and "h ⊢ get_ancestors ptr →r ancestors"
    and type_wf: "type_wf h"
    and known_ptrs: "known_ptrs h"
  obtains children ancestor_child where "h ⊢ get_child_nodes ancestor →r children"
```

```
    and "ancestor_child ∈ set children" and "cast ancestor_child ∈ set ancestors"
⟨proof⟩

lemma get_ancestors_parent_child_rel:
  assumes "heap_is_wellformed h"
    and "h ⊢ get_ancestors child →r ancestors"
    and known_ptrs: "known_ptrs h"
    and type_wf: "type_wf h"
  shows "(ptr, child) ∈ (parent_child_rel h)* ⟷ ptr ∈ set ancestors"
⟨proof⟩

lemma get_root_node_parent_child_rel:
  assumes "heap_is_wellformed h"
    and "h ⊢ get_root_node child →r root"
    and known_ptrs: "known_ptrs h"
    and type_wf: "type_wf h"
  shows "(root, child) ∈ (parent_child_rel h)*"
  ⟨proof⟩


lemma get_ancestors_eq:
  assumes "heap_is_wellformed h"
    and "heap_is_wellformed h'"
    and "⋀object_ptr w. object_ptr ≠ ptr ⟹ w ∈ get_child_nodes_locs object_ptr ⟹ w h h'"
    and pointers_preserved: "⋀object_ptr. preserved (get_M_Object object_ptr RObject.nothing) h h'"
    and known_ptrs: "known_ptrs h"
    and known_ptrs': "known_ptrs h'"
    and "h ⊢ get_ancestors ptr →r ancestors"
    and type_wf: "type_wf h"
    and type_wf': "type_wf h'"
  shows "h' ⊢ get_ancestors ptr →r ancestors"
⟨proof⟩

lemma get_ancestors_remains_not_in_ancestors:
  assumes "heap_is_wellformed h"
    and "heap_is_wellformed h'"
    and "h ⊢ get_ancestors ptr →r ancestors"
    and "h' ⊢ get_ancestors ptr →r ancestors'"
    and "⋀p children children'. h ⊢ get_child_nodes p →r children
        ⟹ h' ⊢ get_child_nodes p →r children' ⟹ set children' ⊆ set children"
    and "node ∉ set ancestors"
    and object_ptr_kinds_eq3: "object_ptr_kinds h = object_ptr_kinds h'"
    and known_ptrs: "known_ptrs h"
    and type_wf: "type_wf h"
    and type_wf': "type_wf h'"
  shows "node ∉ set ancestors'"
⟨proof⟩

lemma get_ancestors_ptrs_in_heap:
  assumes "heap_is_wellformed h" and "type_wf h" and "known_ptrs h"
  assumes "h ⊢ get_ancestors ptr →r ancestors"
  assumes "ptr' ∈ set ancestors"
  shows "ptr' |∈| object_ptr_kinds h"
⟨proof⟩


lemma get_ancestors_prefix:
  assumes "heap_is_wellformed h" and "type_wf h" and "known_ptrs h"
  assumes "h ⊢ get_ancestors ptr →r ancestors"
  assumes "ptr' ∈ set ancestors"
  assumes "h ⊢ get_ancestors ptr' →r ancestors'"
  shows "∃pre. ancestors = pre @ ancestors'"
⟨proof⟩
```

```
lemma get_ancestors_same_root_node:
  assumes "heap_is_wellformed h" and "type_wf h" and "known_ptrs h"
  assumes "h ⊢ get_ancestors ptr →r ancestors"
  assumes "ptr' ∈ set ancestors"
  assumes "ptr'' ∈ set ancestors"
  shows "h ⊢ get_root_node ptr' →r root_ptr ⟷ h ⊢ get_root_node ptr'' →r root_ptr"
⟨proof⟩

lemma get_root_node_parent_same:
  assumes "h ⊢ get_parent child →r Some ptr"
  shows "h ⊢ get_root_node (cast child) →r root ⟷ h ⊢ get_root_node ptr →r root"
⟨proof⟩

lemma get_root_node_same_no_parent:
  assumes "heap_is_wellformed h" and "type_wf h" and "known_ptrs h"
  assumes "h ⊢ get_root_node ptr →r cast child"
  shows "h ⊢ get_parent child →r None"
⟨proof⟩

lemma get_root_node_not_node_same:
  assumes "ptr |∈| object_ptr_kinds h"
  assumes "¬is_node_ptr_kind ptr"
  shows "h ⊢ get_root_node ptr →r ptr"
  ⟨proof⟩


lemma get_root_node_root_in_heap:
  assumes "heap_is_wellformed h" and "type_wf h" and "known_ptrs h"
  assumes "h ⊢ get_root_node ptr →r root"
  shows "root |∈| object_ptr_kinds h"
  ⟨proof⟩


lemma get_root_node_same_no_parent_parent_child_rel:
  assumes "heap_is_wellformed h" and "type_wf h" and "known_ptrs h"
  assumes "h ⊢ get_root_node ptr' →r ptr'"
  shows "¬(∃p. (p, ptr') ∈ (parent_child_rel h))"
  ⟨proof⟩

end


locale l_get_ancestors_wf = l_heap_is_wellformed_defs + l_known_ptrs + l_type_wf + l_get_ancestors_defs
  + l_get_child_nodes_defs + l_get_parent_defs +
  assumes get_ancestors_never_empty:
    "heap_is_wellformed h ⟹ h ⊢ get_ancestors child →r ancestors ⟹ ancestors ≠ []"
  assumes get_ancestors_ok:
    "heap_is_wellformed h ⟹ ptr |∈| object_ptr_kinds h ⟹ known_ptrs h ⟹ type_wf h
                         ⟹ h ⊢ ok (get_ancestors ptr)"
  assumes get_ancestors_reads:
    "heap_is_wellformed h ⟹ reads get_ancestors_locs (get_ancestors node_ptr) h h'"
  assumes get_ancestors_ptrs_in_heap:
    "heap_is_wellformed h ⟹ type_wf h ⟹ known_ptrs h
                         ⟹ h ⊢ get_ancestors ptr →r ancestors ⟹ ptr' ∈ set ancestors
                         ⟹ ptr' |∈| object_ptr_kinds h"
  assumes get_ancestors_remains_not_in_ancestors:
    "heap_is_wellformed h ⟹ heap_is_wellformed h' ⟹ h ⊢ get_ancestors ptr →r ancestors
                         ⟹ h' ⊢ get_ancestors ptr →r ancestors'
                         ⟹ (⋀p children children'. h ⊢ get_child_nodes p →r children
                              ⟹ h' ⊢ get_child_nodes p →r children'
                              ⟹ set children' ⊆ set children)
                         ⟹ node ∉ set ancestors
                         ⟹ object_ptr_kinds h = object_ptr_kinds h' ⟹ known_ptrs h
                         ⟹ type_wf h ⟹ type_wf h' ⟹ node ∉ set ancestors'"
  assumes get_ancestors_also_parent:
    "heap_is_wellformed h ⟹ h ⊢ get_ancestors some_ptr →r ancestors
                         ⟹ cast child_node ∈ set ancestors
                         ⟹ h ⊢ get_parent child_node →r Some parent ⟹ type_wf h
                         ⟹ known_ptrs h ⟹ parent ∈ set ancestors"
  assumes get_ancestors_obtains_children:
    "heap_is_wellformed h ⟹ ancestor ≠ ptr ⟹ ancestor ∈ set ancestors
                         ⟹ h ⊢ get_ancestors ptr →r ancestors ⟹ type_wf h ⟹ known_ptrs h
                         ⟹ (⋀children ancestor_child . h ⊢ get_child_nodes ancestor →r children
                              ⟹ ancestor_child ∈ set children
                              ⟹ cast ancestor_child ∈ set ancestors
                              ⟹ thesis)
                         ⟹ thesis"
  assumes get_ancestors_parent_child_rel:
    "heap_is_wellformed h ⟹ h ⊢ get_ancestors child →r ancestors ⟹ known_ptrs h ⟹ type_wf h
                         ⟹ (ptr, child) ∈ (parent_child_rel h)* ⟷ ptr ∈ set ancestors"

locale l_get_root_node_wf = l_heap_is_wellformed_defs + l_get_root_node_defs + l_type_wf
  + l_known_ptrs + l_get_ancestors_defs + l_get_parent_defs +
  assumes get_root_node_ok:
    "heap_is_wellformed h ⟹ known_ptrs h ⟹ type_wf h ⟹ ptr |∈| object_ptr_kinds h
                         ⟹ h ⊢ ok (get_root_node ptr)"
  assumes get_root_node_ptr_in_heap:
    "h ⊢ ok (get_root_node ptr) ⟹ ptr |∈| object_ptr_kinds h"
  assumes get_root_node_root_in_heap:
    "heap_is_wellformed h ⟹ type_wf h ⟹ known_ptrs h
                         ⟹ h ⊢ get_root_node ptr →r root ⟹ root |∈| object_ptr_kinds h"
  assumes get_ancestors_same_root_node:
    "heap_is_wellformed h ⟹ type_wf h ⟹ known_ptrs h
                         ⟹ h ⊢ get_ancestors ptr →r ancestors ⟹ ptr' ∈ set ancestors
                         ⟹ ptr'' ∈ set ancestors
                         ⟹ h ⊢ get_root_node ptr' →r root_ptr ⟷ h ⊢ get_root_node ptr'' →r root_ptr"
  assumes get_root_node_same_no_parent:
    "heap_is_wellformed h ⟹ type_wf h ⟹ known_ptrs h
                         ⟹ h ⊢ get_root_node ptr →r cast child ⟹ h ⊢ get_parent child →r None"
  assumes get_root_node_parent_same:
    "h ⊢ get_parent child →r Some ptr
         ⟹ h ⊢ get_root_node (cast child) →r root ⟷ h ⊢ get_root_node ptr →r root"

interpretation i_get_root_node_wf?:
  l_get_root_node_wfCore_DOM known_ptr type_wf known_ptrs heap_is_wellformed parent_child_rel
  get_child_nodes get_child_nodes_locs get_disconnected_nodes get_disconnected_nodes_locs
  get_parent get_parent_locs get_ancestors get_ancestors_locs get_root_node get_root_node_locs
  ⟨proof⟩
declare l_get_root_node_wfCore_DOM_axioms[instances]

lemma get_ancestors_wf_is_l_get_ancestors_wf [instances]:
  "l_get_ancestors_wf heap_is_wellformed parent_child_rel known_ptr known_ptrs type_wf get_ancestors
  get_ancestors_locs get_child_nodes get_parent"
  ⟨proof⟩

lemma get_root_node_wf_is_l_get_root_node_wf [instances]:
  "l_get_root_node_wf heap_is_wellformed get_root_node type_wf known_ptr known_ptrs
   get_ancestors get_parent"
  ⟨proof⟩
```

6.3.5 to_tree_order

```
locale l_to_tree_order_wfCore_DOM =
  l_to_tree_orderCore_DOM +
  l_get_parent +
  l_get_parent_wf +
  l_heap_is_wellformed

begin

lemma to_tree_order_ptr_in_heap:
  assumes "heap_is_wellformed h" and "known_ptrs h" and "type_wf h"
  assumes "h ⊢ ok (to_tree_order ptr)"
  shows "ptr |∈| object_ptr_kinds h"
⟨proof⟩

lemma to_tree_order_either_ptr_or_in_children:
  assumes "h ⊢ to_tree_order ptr →r nodes"
    and "node ∈ set nodes"
    and "h ⊢ get_child_nodes ptr →r children"
    and "node ≠ ptr"
  obtains child child_to where "child ∈ set children"
    and "h ⊢ to_tree_order (cast child) →r child_to" and "node ∈ set child_to"
⟨proof⟩


lemma to_tree_order_ptrs_in_heap:
  assumes "heap_is_wellformed h" and "type_wf h" and "known_ptrs h"
  assumes "h ⊢ to_tree_order ptr →r to"
  assumes "ptr' ∈ set to"
  shows "ptr' |∈| object_ptr_kinds h"
⟨proof⟩

lemma to_tree_order_ok:
  assumes wellformed: "heap_is_wellformed h"
    and "ptr |∈| object_ptr_kinds h"
    and "known_ptrs h"
    and type_wf: "type_wf h"
  shows "h ⊢ ok (to_tree_order ptr)"
⟨proof⟩

lemma to_tree_order_child_subset:
  assumes "heap_is_wellformed h"
    and "h ⊢ to_tree_order ptr →r nodes"
    and "h ⊢ get_child_nodes ptr →r children"
    and "node ∈ set children"
    and "h ⊢ to_tree_order (cast node) →r nodes'"
  shows "set nodes' ⊆ set nodes"
⟨proof⟩

lemma to_tree_order_ptr_in_result:
  assumes "h ⊢ to_tree_order ptr →r nodes"
  shows "ptr ∈ set nodes"
  ⟨proof⟩

lemma to_tree_order_subset:
  assumes "heap_is_wellformed h"
    and "h ⊢ to_tree_order ptr →r nodes"
    and "node ∈ set nodes"
    and "h ⊢ to_tree_order node →r nodes'"
    and "known_ptrs h"
    and type_wf: "type_wf h"
  shows "set nodes' ⊆ set nodes"
⟨proof⟩

lemma to_tree_order_parent:
  assumes "heap_is_wellformed h" and "type_wf h" and "known_ptrs h"
  assumes "h ⊢ to_tree_order ptr →r nodes"
  assumes "h ⊢ get_parent child →r Some parent"
  assumes "parent ∈ set nodes"
  shows "cast child ∈ set nodes"
⟨proof⟩

lemma to_tree_order_child:
  assumes "heap_is_wellformed h" and "type_wf h" and "known_ptrs h"
  assumes "h ⊢ to_tree_order ptr →r nodes"
  assumes "h ⊢ get_child_nodes parent →r children"
  assumes "cast child ≠ ptr"
  assumes "child ∈ set children"
  assumes "cast child ∈ set nodes"
  shows "parent ∈ set nodes"
⟨proof⟩

lemma to_tree_order_node_ptrs:
  assumes "heap_is_wellformed h" and "type_wf h" and "known_ptrs h"
  assumes "h ⊢ to_tree_order ptr →r nodes"
  assumes "ptr' ≠ ptr"
  assumes "ptr' ∈ set nodes"
  shows "is_node_ptr_kind ptr'"
⟨proof⟩

lemma to_tree_order_child2:
  assumes "heap_is_wellformed h" and "type_wf h" and "known_ptrs h"
  assumes "h ⊢ to_tree_order ptr →r nodes"
  assumes "cast child ≠ ptr"
  assumes "cast child ∈ set nodes"
  obtains parent where "h ⊢ get_parent child →r Some parent" and "parent ∈ set nodes"
⟨proof⟩

lemma to_tree_order_parent_child_rel:
  assumes "heap_is_wellformed h" and "type_wf h" and "known_ptrs h"
  assumes "h ⊢ to_tree_order ptr →r to"
  shows "(ptr, child) ∈ (parent_child_rel h)* ⟷ child ∈ set to"
⟨proof⟩
end

interpretation i_to_tree_order_wf?: l_to_tree_order_wf_Core_DOM known_ptr type_wf get_child_nodes
  get_child_nodes_locs to_tree_order known_ptrs get_parent
  get_parent_locs heap_is_wellformed parent_child_rel
  get_disconnected_nodes get_disconnected_nodes_locs
  ⟨proof⟩
declare l_to_tree_order_wf_Core_DOM_axioms [instances]

locale l_to_tree_order_wf = l_heap_is_wellformed_defs + l_type_wf + l_known_ptrs
  + l_to_tree_order_defs
  + l_get_parent_defs + l_get_child_nodes_defs +
  assumes to_tree_order_ok:
    "heap_is_wellformed h ⟹ ptr |∈| object_ptr_kinds h ⟹ known_ptrs h ⟹ type_wf h
                          ⟹ h ⊢ ok (to_tree_order ptr)"
  assumes to_tree_order_ptrs_in_heap:
    "heap_is_wellformed h ⟹ type_wf h ⟹ known_ptrs h ⟹ h ⊢ to_tree_order ptr →r to
                          ⟹ ptr' ∈ set to ⟹ ptr' |∈| object_ptr_kinds h"
  assumes to_tree_order_parent_child_rel:
    "heap_is_wellformed h ⟹ type_wf h ⟹ known_ptrs h ⟹ h ⊢ to_tree_order ptr →r to
                          ⟹ (ptr, child_ptr) ∈ (parent_child_rel h)* ⟷ child_ptr ∈ set to"
  assumes to_tree_order_child2:
    "heap_is_wellformed h ⟹ type_wf h ⟹ known_ptrs h ⟹ h ⊢ to_tree_order ptr →r nodes
                          ⟹ cast child ≠ ptr ⟹ cast child ∈ set nodes
                          ⟹ (⋀parent. h ⊢ get_parent child →r Some parent
                                        ⟹ parent ∈ set nodes ⟹ thesis)
```

```
                           ⟹ thesis"
  assumes to_tree_order_node_ptrs:
    "heap_is_wellformed h ⟹ type_wf h ⟹ known_ptrs h ⟹ h ⊢ to_tree_order ptr →r nodes
                           ⟹ ptr' ≠ ptr ⟹ ptr' ∈ set nodes ⟹ is_node_ptr_kind ptr'"
  assumes to_tree_order_child:
    "heap_is_wellformed h ⟹ type_wf h ⟹ known_ptrs h ⟹ h ⊢ to_tree_order ptr →r nodes
                           ⟹ h ⊢ get_child_nodes parent →r children ⟹ cast child ≠ ptr
                           ⟹ child ∈ set children ⟹ cast child ∈ set nodes
                           ⟹ parent ∈ set nodes"
  assumes to_tree_order_ptr_in_result:
    "h ⊢ to_tree_order ptr →r nodes ⟹ ptr ∈ set nodes"
  assumes to_tree_order_parent:
    "heap_is_wellformed h ⟹ type_wf h ⟹ known_ptrs h ⟹ h ⊢ to_tree_order ptr →r nodes
                           ⟹ h ⊢ get_parent child →r Some parent ⟹ parent ∈ set nodes
                           ⟹ cast child ∈ set nodes"
  assumes to_tree_order_subset:
    "heap_is_wellformed h ⟹ h ⊢ to_tree_order ptr →r nodes ⟹ node ∈ set nodes
                           ⟹ h ⊢ to_tree_order node →r nodes' ⟹ known_ptrs h
                           ⟹ type_wf h ⟹ set nodes' ⊆ set nodes"

lemma to_tree_order_wf_is_l_to_tree_order_wf [instances]:
  "l_to_tree_order_wf heap_is_wellformed parent_child_rel type_wf known_ptr known_ptrs
                      to_tree_order get_parent get_child_nodes"
  ⟨proof⟩
```

get_root_node

```
locale l_to_tree_order_wf_get_root_node_wf_Core_DOM =
  l_get_root_node_wf_Core_DOM
  + l_to_tree_order_wf
begin
lemma to_tree_order_get_root_node:
  assumes "heap_is_wellformed h" and "type_wf h" and "known_ptrs h"
  assumes "h ⊢ to_tree_order ptr →r to"
  assumes "ptr' ∈ set to"
  assumes "h ⊢ get_root_node ptr' →r root_ptr"
  assumes "ptr'' ∈ set to"
  shows "h ⊢ get_root_node ptr'' →r root_ptr"
⟨proof⟩

lemma to_tree_order_same_root:
  assumes "heap_is_wellformed h" and "type_wf h" and "known_ptrs h"
  assumes "h ⊢ get_root_node ptr →r root_ptr"
  assumes "h ⊢ to_tree_order root_ptr →r to"
  assumes "ptr' ∈ set to"
  shows "h ⊢ get_root_node ptr' →r root_ptr"
⟨proof⟩
end

interpretation i_to_tree_order_wf_get_root_node_wf?: l_to_tree_order_wf_get_root_node_wf_Core_DOM
  known_ptr type_wf known_ptrs heap_is_wellformed parent_child_rel get_child_nodes
  get_child_nodes_locs get_disconnected_nodes get_disconnected_nodes_locs get_parent get_parent_locs
  get_ancestors get_ancestors_locs get_root_node get_root_node_locs to_tree_order
  ⟨proof⟩

locale l_to_tree_order_wf_get_root_node_wf = l_type_wf + l_known_ptrs + l_to_tree_order_defs
  + l_get_root_node_defs + l_heap_is_wellformed_defs +
  assumes to_tree_order_get_root_node:
    "heap_is_wellformed h ⟹ type_wf h ⟹ known_ptrs h ⟹ h ⊢ to_tree_order ptr →r to
                           ⟹ ptr' ∈ set to ⟹ h ⊢ get_root_node ptr' →r root_ptr
                           ⟹ ptr'' ∈ set to ⟹ h ⊢ get_root_node ptr'' →r root_ptr"
  assumes to_tree_order_same_root:
    "heap_is_wellformed h ⟹ type_wf h ⟹ known_ptrs h
```

```
                    ⟹ h ⊢ get_root_node ptr →r root_ptr
                    ⟹ h ⊢ to_tree_order root_ptr →r to ⟹ ptr' ∈ set to
                    ⟹ h ⊢ get_root_node ptr' →r root_ptr"
```

lemma *to_tree_order_wf_get_root_node_wf_is_l_to_tree_order_wf_get_root_node_wf [instances]:*
```
  "l_to_tree_order_wf_get_root_node_wf type_wf known_ptr known_ptrs to_tree_order
                                    get_root_node heap_is_wellformed"
```
  ⟨*proof*⟩

6.3.6 get_owner_document

locale *l_get_owner_document_wf*$_{Core\_DOM}$ =
```
  l_known_ptrs
  + l_heap_is_wellformed
```
  + *l_get_root_node*$_{Core\_DOM}$
```
  + l_get_ancestors
  + l_get_ancestors_wf
  + l_get_parent
  + l_get_parent_wf
  + l_get_root_node_wf
```
  + *l_get_owner_document*$_{Core\_DOM}$

begin

lemma *get_owner_document_disconnected_nodes:*
  assumes *"heap_is_wellformed h"*
  assumes *"h ⊢ get_disconnected_nodes document_ptr →*$_r$ *disc_nodes"*
  assumes *"node_ptr ∈ set disc_nodes"*
  assumes *known_ptrs: "known_ptrs h"*
  assumes *type_wf: "type_wf h"*
  shows *"h ⊢ get_owner_document (cast node_ptr) →*$_r$ *document_ptr"*
⟨*proof*⟩

lemma *in_disconnected_nodes_no_parent:*
  assumes *"heap_is_wellformed h"*
    and *"h ⊢ get_parent node_ptr →*$_r$ *None"*
    and *"h ⊢ get_owner_document (cast node_ptr) →*$_r$ *owner_document"*
    and *"h ⊢ get_disconnected_nodes owner_document →*$_r$ *disc_nodes"*
    and *known_ptrs: "known_ptrs h"*
    and *type_wf: "type_wf h"*
  shows *"node_ptr ∈ set disc_nodes"*
⟨*proof*⟩

lemma *get_owner_document_owner_document_in_heap:*
  assumes *"heap_is_wellformed h"* **and** *"type_wf h"* **and** *"known_ptrs h"*
  assumes *"h ⊢ get_owner_document ptr →*$_r$ *owner_document"*
  shows *"owner_document |∈| document_ptr_kinds h"*
  ⟨*proof*⟩

lemma *get_owner_document_ok:*
  assumes *"heap_is_wellformed h" "known_ptrs h" "type_wf h"*
  assumes *"ptr |∈| object_ptr_kinds h"*
  shows *"h ⊢ ok (get_owner_document ptr)"*
⟨*proof*⟩

lemma *get_owner_document_child_same:*
  assumes *"heap_is_wellformed h" "known_ptrs h" "type_wf h"*
  assumes *"h ⊢ get_child_nodes ptr →*$_r$ *children"*
  assumes *"child ∈ set children"*
  shows *"h ⊢ get_owner_document ptr →*$_r$ *owner_document ⟷ h ⊢ get_owner_document (cast child) →*$_r$ *owner_documen*
⟨*proof*⟩

end

```
locale l_get_owner_document_wf = l_heap_is_wellformed_defs + l_type_wf + l_known_ptrs
  + l_get_disconnected_nodes_defs + l_get_owner_document_defs
  + l_get_parent_defs +
  assumes get_owner_document_disconnected_nodes:
    "heap_is_wellformed h ⟹
     known_ptrs h ⟹
     type_wf h ⟹
     h ⊢ get_disconnected_nodes document_ptr →r disc_nodes ⟹
     node_ptr ∈ set disc_nodes ⟹
     h ⊢ get_owner_document (cast node_ptr) →r document_ptr"
  assumes in_disconnected_nodes_no_parent:
    "heap_is_wellformed h ⟹
    h ⊢ get_parent node_ptr →r None⟹
    h ⊢ get_owner_document (cast node_ptr) →r owner_document ⟹
    h ⊢ get_disconnected_nodes owner_document →r disc_nodes ⟹
    known_ptrs h ⟹
    type_wf h⟹
    node_ptr ∈ set disc_nodes"
  assumes get_owner_document_owner_document_in_heap:
    "heap_is_wellformed h ⟹ type_wf h ⟹ known_ptrs h ⟹
h ⊢ get_owner_document ptr →r owner_document ⟹
owner_document |∈| document_ptr_kinds h"
  assumes get_owner_document_ok:
    "heap_is_wellformed h ⟹ known_ptrs h ⟹ type_wf h ⟹ ptr |∈| object_ptr_kinds h
                          ⟹ h ⊢ ok (get_owner_document ptr)"

interpretation i_get_owner_document_wf?: l_get_owner_document_wf_Core_DOM
  known_ptr known_ptrs type_wf heap_is_wellformed parent_child_rel get_child_nodes
  get_child_nodes_locs get_disconnected_nodes get_disconnected_nodes_locs get_parent get_parent_locs
  get_ancestors get_ancestors_locs get_root_node get_root_node_locs get_owner_document
  ⟨proof⟩
declare l_get_owner_document_wf_Core_DOM_axioms [instances]

lemma get_owner_document_wf_is_l_get_owner_document_wf [instances]:
  "l_get_owner_document_wf heap_is_wellformed type_wf known_ptr known_ptrs get_disconnected_nodes
                          get_owner_document get_parent"
  ⟨proof⟩
```

get_root_node

```
locale l_get_owner_document_wf_get_root_node_wf_Core_DOM =
  l_get_root_node_Core_DOM +
  l_get_root_node_wf +
  l_get_owner_document_Core_DOM +
  l_get_owner_document_wf
begin

lemma get_root_node_document:
  assumes "heap_is_wellformed h" and "type_wf h" and "known_ptrs h"
  assumes "h ⊢ get_root_node ptr →r root"
  assumes "is_document_ptr_kind root"
  shows "h ⊢ get_owner_document ptr →r the (cast root)"
⟨proof⟩

lemma get_root_node_same_owner_document:
  assumes "heap_is_wellformed h" and "type_wf h" and "known_ptrs h"
  assumes "h ⊢ get_root_node ptr →r root"
  shows "h ⊢ get_owner_document ptr →r owner_document ⟷ h ⊢ get_owner_document root →r owner_document"
⟨proof⟩
end

interpretation get_owner_document_wf_get_root_node_wf?: l_get_owner_document_wf_get_root_node_wf_Core_DOM
  type_wf known_ptr known_ptrs get_parent get_parent_locs get_child_nodes get_child_nodes_locs
  get_ancestors get_ancestors_locs get_root_node get_root_node_locs heap_is_wellformed parent_child_rel
  get_disconnected_nodes get_disconnected_nodes_locs get_owner_document
  ⟨proof⟩
declare l_get_owner_document_wf_get_root_node_wf$_{Core\_DOM}$_axioms [instances]

locale l_get_owner_document_wf_get_root_node_wf = l_heap_is_wellformed_defs + l_type_wf +
  l_known_ptrs + l_get_root_node_defs + l_get_owner_document_defs +
  assumes get_root_node_document:
    "heap_is_wellformed h ⟹ type_wf h ⟹ known_ptrs h ⟹ h ⊢ get_root_node ptr →$_r$ root ⟹
is_document_ptr_kind root ⟹ h ⊢ get_owner_document ptr →$_r$ the (cast root)"
  assumes get_root_node_same_owner_document:
    "heap_is_wellformed h ⟹ type_wf h ⟹ known_ptrs h ⟹ h ⊢ get_root_node ptr →$_r$ root ⟹
h ⊢ get_owner_document ptr →$_r$ owner_document ⟷ h ⊢ get_owner_document root →$_r$ owner_document"

lemma get_owner_document_wf_get_root_node_wf_is_l_get_owner_document_wf_get_root_node_wf [instances]:
  "l_get_owner_document_wf_get_root_node_wf heap_is_wellformed type_wf known_ptr known_ptrs
get_root_node get_owner_document"
  ⟨proof⟩
```

6.3.7 Preserving heap-wellformedness

6.3.8 set_attribute

```
locale l_set_attribute_wf$_{Core\_DOM}$ =
  l_get_parent_wf2$_{Core\_DOM}$ +
  l_set_attribute$_{Core\_DOM}$ +
  l_set_attribute_get_disconnected_nodes +
  l_set_attribute_get_child_nodes
begin
lemma set_attribute_preserves_wellformedness:
  assumes "heap_is_wellformed h"
    and "h ⊢ set_attribute element_ptr k v →$_h$ h'"
  shows "heap_is_wellformed h'"
  thm preserves_wellformedness_writes_needed
  ⟨proof⟩
end
```

6.3.9 remove_child

```
locale l_remove_child_wf$_{Core\_DOM}$ =
  l_remove_child$_{Core\_DOM}$ +
  l_get_parent_wf$_{Core\_DOM}$ +
  l_heap_is_wellformed +
  l_set_disconnected_nodes_get_child_nodes
begin
lemma remove_child_removes_parent:
  assumes wellformed: "heap_is_wellformed h"
    and remove_child: "h ⊢ remove_child ptr child →$_h$ h2"
    and known_ptrs: "known_ptrs h"
    and type_wf: "type_wf h"
  shows "h2 ⊢ get_parent child →$_r$ None"
⟨proof⟩
end

locale l_remove_child_wf2$_{Core\_DOM}$ =
  l_remove_child_wf$_{Core\_DOM}$ +
  l_heap_is_wellformed$_{Core\_DOM}$
begin

lemma remove_child_parent_child_rel_subset:
  assumes "heap_is_wellformed h"
    and "h ⊢ remove_child ptr child →$_h$ h'"
    and "known_ptrs h"
    and type_wf: "type_wf h"
  shows "parent_child_rel h' ⊆ parent_child_rel h"
⟨proof⟩


lemma remove_child_heap_is_wellformed_preserved:
  assumes "heap_is_wellformed h"
    and "h ⊢ remove_child ptr child →h h'"
    and "known_ptrs h"
    and type_wf: "type_wf h"
  shows "type_wf h'" and "known_ptrs h'" and "heap_is_wellformed h'"
⟨proof⟩

lemma remove_heap_is_wellformed_preserved:
  assumes "heap_is_wellformed h"
    and "h ⊢ remove child →h h'"
    and "known_ptrs h"
    and type_wf: "type_wf h"
  shows "type_wf h'" and "known_ptrs h'" and "heap_is_wellformed h'"
  ⟨proof⟩

lemma remove_child_removes_child:
  assumes wellformed: "heap_is_wellformed h"
    and remove_child: "h ⊢ remove_child ptr' child →h h'"
    and children: "h' ⊢ get_child_nodes ptr →r children"
    and known_ptrs: "known_ptrs h"
    and type_wf: "type_wf h"
  shows "child ∉ set children"
⟨proof⟩

lemma remove_child_removes_first_child:
  assumes "heap_is_wellformed h" and "type_wf h" and "known_ptrs h"
  assumes "h ⊢ get_child_nodes ptr →r node_ptr # children"
  assumes "h ⊢ remove_child ptr node_ptr →h h'"
  shows "h' ⊢ get_child_nodes ptr →r children"
⟨proof⟩

lemma remove_removes_child:
  assumes "heap_is_wellformed h" and "type_wf h" and "known_ptrs h"
  assumes "h ⊢ get_child_nodes ptr →r node_ptr # children"
  assumes "h ⊢ remove node_ptr →h h'"
  shows "h' ⊢ get_child_nodes ptr →r children"
⟨proof⟩

lemma remove_for_all_empty_children:
  assumes "heap_is_wellformed h" and "type_wf h" and "known_ptrs h"
  assumes "h ⊢ get_child_nodes ptr →r children"
  assumes "h ⊢ forall_M remove children →h h'"
  shows "h' ⊢ get_child_nodes ptr →r []"
  ⟨proof⟩
end

locale l_remove_child_wf2 = l_type_wf + l_known_ptrs + l_remove_child_defs + l_heap_is_wellformed_defs
  + l_get_child_nodes_defs + l_remove_defs +
  assumes remove_child_preserves_type_wf:
    "heap_is_wellformed h ⟹ type_wf h ⟹ known_ptrs h ⟹ h ⊢ remove_child ptr child →h h'
                         ⟹ type_wf h'"
  assumes remove_child_preserves_known_ptrs:
    "heap_is_wellformed h ⟹ type_wf h ⟹ known_ptrs h ⟹ h ⊢ remove_child ptr child →h h'
                         ⟹ known_ptrs h'"
  assumes remove_child_heap_is_wellformed_preserved:
    "type_wf h ⟹ known_ptrs h ⟹ heap_is_wellformed h ⟹ h ⊢ remove_child ptr child →h h'
              ⟹ heap_is_wellformed h'"
  assumes remove_preserves_type_wf:
    "heap_is_wellformed h ⟹ type_wf h ⟹ known_ptrs h ⟹ h ⊢ remove child →h h'
                          ⟹ type_wf h'"
  assumes remove_preserves_known_ptrs:
    "heap_is_wellformed h ⟹ type_wf h ⟹ known_ptrs h ⟹ h ⊢ remove child →h h'
                          ⟹ known_ptrs h'"
  assumes remove_heap_is_wellformed_preserved:
    "type_wf h ⟹ known_ptrs h ⟹ heap_is_wellformed h ⟹ h ⊢ remove child →h h'
              ⟹ heap_is_wellformed h'"
  assumes remove_child_removes_child:
    "heap_is_wellformed h ⟹ h ⊢ remove_child ptr' child →h h' ⟹ h' ⊢ get_child_nodes ptr →r children
                          ⟹ known_ptrs h ⟹ type_wf h
                          ⟹ child ∉ set children"
  assumes remove_child_removes_first_child:
    "heap_is_wellformed h ⟹ type_wf h ⟹ known_ptrs h
                          ⟹ h ⊢ get_child_nodes ptr →r node_ptr # children
                          ⟹ h ⊢ remove_child ptr node_ptr →h h'
                          ⟹ h' ⊢ get_child_nodes ptr →r children"
  assumes remove_removes_child:
    "heap_is_wellformed h ⟹ type_wf h ⟹ known_ptrs h
                          ⟹ h ⊢ get_child_nodes ptr →r node_ptr # children
                          ⟹ h ⊢ remove node_ptr →h h' ⟹ h' ⊢ get_child_nodes ptr →r children"
  assumes remove_for_all_empty_children:
    "heap_is_wellformed h ⟹ type_wf h ⟹ known_ptrs h ⟹ h ⊢ get_child_nodes ptr →r children
                          ⟹ h ⊢ forall_M remove children →h h' ⟹ h' ⊢ get_child_nodes ptr →r []"

interpretation i_remove_child_wf2?: l_remove_child_wf2Core_DOM get_child_nodes get_child_nodes_locs
  set_child_nodes set_child_nodes_locs get_parent get_parent_locs get_owner_document
  get_disconnected_nodes get_disconnected_nodes_locs set_disconnected_nodes
  set_disconnected_nodes_locs remove_child remove_child_locs remove type_wf known_ptr known_ptrs
  heap_is_wellformed parent_child_rel
  ⟨proof⟩

lemma remove_child_wf2_is_l_remove_child_wf2 [instances]:
  "l_remove_child_wf2 type_wf known_ptr known_ptrs remove_child heap_is_wellformed get_child_nodes remove"
  ⟨proof⟩
```

6.3.10 adopt_node

```
locale l_adopt_node_wfCore_DOM =
  l_adopt_nodeCore_DOM +
  l_get_parent_wf +
  l_get_owner_document_wf +
  l_remove_child_wf2 +
  l_heap_is_wellformed
begin
lemma adopt_node_removes_first_child:
  assumes "heap_is_wellformed h" and "type_wf h" and "known_ptrs h"
  assumes "h ⊢ adopt_node owner_document node →h h'"
  assumes "h ⊢ get_child_nodes ptr' →r node # children"
  shows "h' ⊢ get_child_nodes ptr' →r children"
⟨proof⟩


lemma adopt_node_document_in_heap:
  assumes "heap_is_wellformed h" and "known_ptrs h" and "type_wf h"
  assumes "h ⊢ ok (adopt_node owner_document node)"
  shows "owner_document |∈| document_ptr_kinds h"
⟨proof⟩
end

locale l_adopt_node_wf2Core_DOM =
  l_adopt_node_wfCore_DOM +
  l_adopt_node_Core_DOM +
  l_get_parent_wf_Core_DOM +
  l_get_root_node +
  l_get_owner_document_wf +
  l_remove_child_wf2 +
  l_heap_is_wellformed_Core_DOM
begin

lemma adopt_node_removes_child_step:
  assumes wellformed: "heap_is_wellformed h"
    and adopt_node: "h ⊢ adopt_node owner_document node_ptr →_h h2"
    and children: "h2 ⊢ get_child_nodes ptr →_r children"
    and known_ptrs: "known_ptrs h"
    and type_wf: "type_wf h"
  shows "node_ptr ∉ set children"
⟨proof⟩

lemma adopt_node_removes_child:
  assumes "heap_is_wellformed h" and "known_ptrs h" and "type_wf h"
  assumes "h ⊢ adopt_node owner_document node_ptr →_h h'"
  shows "⋀ptr' children'.
  h' ⊢ get_child_nodes ptr' →_r children' ⟹ node_ptr ∉ set children'"
  ⟨proof⟩

lemma adopt_node_preserves_wellformedness:
  assumes "heap_is_wellformed h"
    and "h ⊢ adopt_node document_ptr child →_h h'"
    and known_ptrs: "known_ptrs h"
    and type_wf: "type_wf h"
  shows "heap_is_wellformed h'" and "known_ptrs h'" and "type_wf h'"
⟨proof⟩

lemma adopt_node_node_in_disconnected_nodes:
  assumes wellformed: "heap_is_wellformed h"
    and adopt_node: "h ⊢ adopt_node owner_document node_ptr →_h h'"
    and "h' ⊢ get_disconnected_nodes owner_document →_r disc_nodes"
    and known_ptrs: "known_ptrs h"
    and type_wf: "type_wf h"
  shows "node_ptr ∈ set disc_nodes"
⟨proof⟩
end

interpretation i_adopt_node_wf?: l_adopt_node_wf_Core_DOM get_owner_document get_parent get_parent_locs
  remove_child remove_child_locs get_disconnected_nodes get_disconnected_nodes_locs
  set_disconnected_nodes set_disconnected_nodes_locs adopt_node adopt_node_locs known_ptr
  type_wf get_child_nodes get_child_nodes_locs known_ptrs set_child_nodes set_child_nodes_locs
  remove heap_is_wellformed parent_child_rel
  ⟨proof⟩
declare l_adopt_node_wf_Core_DOM_axioms[instances]

interpretation i_adopt_node_wf2?: l_adopt_node_wf2_Core_DOM get_owner_document get_parent get_parent_locs
  remove_child remove_child_locs get_disconnected_nodes get_disconnected_nodes_locs
  set_disconnected_nodes set_disconnected_nodes_locs adopt_node adopt_node_locs known_ptr
  type_wf get_child_nodes get_child_nodes_locs known_ptrs set_child_nodes set_child_nodes_locs
  remove heap_is_wellformed parent_child_rel get_root_node get_root_node_locs
  ⟨proof⟩
declare l_adopt_node_wf2_Core_DOM_axioms[instances]


locale l_adopt_node_wf = l_heap_is_wellformed + l_known_ptrs + l_type_wf + l_adopt_node_defs
  + l_get_child_nodes_defs + l_get_disconnected_nodes_defs +
  assumes adopt_node_preserves_wellformedness:
    "heap_is_wellformed h ⟹ h ⊢ adopt_node document_ptr child →_h h' ⟹ known_ptrs h
```

```
                        ⟹ type_wf h ⟹ heap_is_wellformed h'"
  assumes adopt_node_removes_child:
    "heap_is_wellformed h ⟹ h ⊢ adopt_node owner_document node_ptr →h h2
                        ⟹ h2 ⊢ get_child_nodes ptr →r children ⟹ known_ptrs h
                        ⟹ type_wf h ⟹ node_ptr ∉ set children"
  assumes adopt_node_node_in_disconnected_nodes:
    "heap_is_wellformed h ⟹ h ⊢ adopt_node owner_document node_ptr →h h'
                        ⟹ h' ⊢ get_disconnected_nodes owner_document →r disc_nodes
                        ⟹ known_ptrs h ⟹ type_wf h ⟹ node_ptr ∈ set disc_nodes"
  assumes adopt_node_removes_first_child: "heap_is_wellformed h ⟹ type_wf h ⟹ known_ptrs h
                       ⟹ h ⊢ adopt_node owner_document node →h h'
                       ⟹ h ⊢ get_child_nodes ptr' →r node # children
                       ⟹ h' ⊢ get_child_nodes ptr' →r children"
  assumes adopt_node_document_in_heap: "heap_is_wellformed h ⟹ known_ptrs h ⟹ type_wf h
                      ⟹ h ⊢ ok (adopt_node owner_document node)
                      ⟹ owner_document |∈| document_ptr_kinds h"
  assumes adopt_node_preserves_type_wf:
    "heap_is_wellformed h ⟹ h ⊢ adopt_node document_ptr child →h h' ⟹ known_ptrs h
                        ⟹ type_wf h ⟹ type_wf h'"
  assumes adopt_node_preserves_known_ptrs:
    "heap_is_wellformed h ⟹ h ⊢ adopt_node document_ptr child →h h' ⟹ known_ptrs h
                        ⟹ type_wf h ⟹ known_ptrs h'"


lemma adopt_node_wf_is_l_adopt_node_wf [instances]:
  "l_adopt_node_wf type_wf known_ptr heap_is_wellformed parent_child_rel get_child_nodes
                  get_disconnected_nodes known_ptrs adopt_node"
  ⟨proof⟩
```

6.3.11 insert_before

```
locale l_insert_before_wf_Core_DOM =
  l_insert_before_Core_DOM +
  l_adopt_node_wf +
  l_set_disconnected_nodes_get_child_nodes +
  l_heap_is_wellformed
begin
lemma insert_before_removes_child:
  assumes "heap_is_wellformed h" and "type_wf h" and "known_ptrs h"
  assumes "ptr ≠ ptr'"
  assumes "h ⊢ insert_before ptr node child →h h'"
  assumes "h ⊢ get_child_nodes ptr' →r node # children"
  shows "h' ⊢ get_child_nodes ptr' →r children"
⟨proof⟩
end

locale l_insert_before_wf = l_heap_is_wellformed_defs + l_type_wf + l_known_ptrs
  + l_insert_before_defs + l_get_child_nodes_defs +
  assumes insert_before_removes_child:
    "heap_is_wellformed h ⟹ type_wf h ⟹ known_ptrs h ⟹ ptr ≠ ptr'
                      ⟹ h ⊢ insert_before ptr node child →h h'
                      ⟹ h ⊢ get_child_nodes ptr' →r node # children
                      ⟹ h' ⊢ get_child_nodes ptr' →r children"

interpretation i_insert_before_wf?: l_insert_before_wf_Core_DOM get_parent get_parent_locs
  get_child_nodes get_child_nodes_locs set_child_nodes
  set_child_nodes_locs get_ancestors get_ancestors_locs
  adopt_node adopt_node_locs set_disconnected_nodes
  set_disconnected_nodes_locs get_disconnected_nodes
  get_disconnected_nodes_locs get_owner_document insert_before
  insert_before_locs append_child type_wf known_ptr known_ptrs
  heap_is_wellformed parent_child_rel
  ⟨proof⟩
```

```
declare l_insert_before_wf$_{Core\_DOM}$_axioms [instances]

lemma insert_before_wf_is_l_insert_before_wf [instances]:
  "l_insert_before_wf heap_is_wellformed type_wf known_ptr known_ptrs insert_before get_child_nodes"
  ⟨proof⟩

locale l_insert_before_wf2$_{Core\_DOM}$ =
  l_insert_before_wf$_{Core\_DOM}$ +
  l_set_child_nodes_get_disconnected_nodes +
  l_remove_child +
  l_get_root_node_wf +
  l_set_disconnected_nodes_get_disconnected_nodes_wf +
  l_set_disconnected_nodes_get_ancestors +
  l_get_ancestors_wf +
  l_get_owner_document +
  l_heap_is_wellformed$_{Core\_DOM}$ +
  l_get_owner_document_wf
begin

lemma insert_before_preserves_acyclitity:
  assumes "heap_is_wellformed h" and "known_ptrs h" and "type_wf h"
  assumes "h ⊢ insert_before ptr node child →h h'"
  shows "acyclic (parent_child_rel h')"
⟨proof⟩

lemma insert_before_heap_is_wellformed_preserved:
  assumes wellformed: "heap_is_wellformed h"
    and insert_before: "h ⊢ insert_before ptr node child →h h'"
    and known_ptrs: "known_ptrs h"
    and type_wf: "type_wf h"
  shows "heap_is_wellformed h'" and "type_wf h'" and "known_ptrs h'"
⟨proof⟩

lemma adopt_node_children_remain_distinct:
  assumes "heap_is_wellformed h" and "known_ptrs h" and "type_wf h"
  assumes "h ⊢ adopt_node owner_document node_ptr →h h'"
  shows "⋀ptr' children'.
  h' ⊢ get_child_nodes ptr' →r children' ⟹ distinct children'"
  ⟨proof⟩


lemma insert_node_children_remain_distinct:
  assumes "heap_is_wellformed h" and "known_ptrs h" and "type_wf h"
  assumes "h ⊢ a_insert_node ptr new_child reference_child_opt →h h'"
  assumes "h ⊢ get_child_nodes ptr →r children"
  assumes "new_child ∉ set children"
  shows "⋀children'.
  h' ⊢ get_child_nodes ptr →r children' ⟹ distinct children'"
⟨proof⟩

lemma insert_before_children_remain_distinct:
  assumes "heap_is_wellformed h" and "known_ptrs h" and "type_wf h"
  assumes "h ⊢ insert_before ptr new_child child_opt →h h'"
  shows "⋀ptr' children'.
  h' ⊢ get_child_nodes ptr' →r children' ⟹ distinct children'"
⟨proof⟩


lemma insert_before_removes_child:
  assumes "heap_is_wellformed h" and "known_ptrs h" and "type_wf h"
  assumes "h ⊢ insert_before ptr node child →h h'"
  assumes "ptr ≠ ptr'"
  shows "⋀children'. h' ⊢ get_child_nodes ptr' →r children' ⟹ node ∉ set children'"
```

```
⟨proof⟩

lemma ensure_pre_insertion_validity_ok:
  assumes "heap_is_wellformed h" and "known_ptrs h" and "type_wf h"
  assumes "ptr |∈| object_ptr_kinds h"
  assumes "¬is_character_data_ptr_kind parent"
  assumes "cast node ∉ set |h ⊢ get_ancestors parent|_r"
  assumes "h ⊢ get_parent ref →_r Some parent"
  assumes "is_document_ptr parent ⟹ h ⊢ get_child_nodes parent →_r []"
  assumes "is_document_ptr parent ⟹ ¬is_character_data_ptr_kind node"
  shows "h ⊢ ok (a_ensure_pre_insertion_validity node parent (Some ref))"
⟨proof⟩
end

locale l_insert_before_wf2 = l_type_wf + l_known_ptrs + l_insert_before_defs
  + l_heap_is_wellformed_defs + l_get_child_nodes_defs + l_remove_defs +
  assumes insert_before_preserves_type_wf:
    "heap_is_wellformed h ⟹ type_wf h ⟹ known_ptrs h ⟹ h ⊢ insert_before ptr child ref →_h h'
                        ⟹ type_wf h'"
  assumes insert_before_preserves_known_ptrs:
    "heap_is_wellformed h ⟹ type_wf h ⟹ known_ptrs h ⟹ h ⊢ insert_before ptr child ref →_h h'
                        ⟹ known_ptrs h'"
  assumes insert_before_heap_is_wellformed_preserved:
    "type_wf h ⟹ known_ptrs h ⟹ heap_is_wellformed h ⟹ h ⊢ insert_before ptr child ref →_h h'
              ⟹ heap_is_wellformed h'"

interpretation i_insert_before_wf2?: l_insert_before_wf2_Core_DOM get_parent get_parent_locs
  get_child_nodes get_child_nodes_locs set_child_nodes
  set_child_nodes_locs get_ancestors get_ancestors_locs
  adopt_node adopt_node_locs set_disconnected_nodes
  set_disconnected_nodes_locs get_disconnected_nodes
  get_disconnected_nodes_locs get_owner_document insert_before
  insert_before_locs append_child type_wf known_ptr known_ptrs
  heap_is_wellformed parent_child_rel remove_child
  remove_child_locs get_root_node get_root_node_locs
  ⟨proof⟩
declare l_insert_before_wf2_Core_DOM_axioms [instances]

lemma insert_before_wf2_is_l_insert_before_wf2 [instances]:
  "l_insert_before_wf2 type_wf known_ptr known_ptrs insert_before heap_is_wellformed"
  ⟨proof⟩



locale l_insert_before_wf3_Core_DOM =
  l_insert_before_wf2_Core_DOM +
  l_adopt_node_Core_DOM +
  l_set_child_nodes_get_child_nodes_Core_DOM +
  l_remove_child_wf2
begin

lemma next_sibling_ok:
  assumes "heap_is_wellformed h" and "known_ptrs h" and "type_wf h"
  assumes "node_ptr |∈| node_ptr_kinds h"
  shows "h ⊢ ok (a_next_sibling node_ptr)"
⟨proof⟩

lemma remove_child_ok:
  assumes "heap_is_wellformed h" and "known_ptrs h" and "type_wf h"
  assumes "h ⊢ get_child_nodes ptr →_r children"
  assumes "child ∈ set children"
  shows "h ⊢ ok (remove_child ptr child)"
⟨proof⟩
```

```
lemma adopt_node_ok:
  assumes "heap_is_wellformed h" and "known_ptrs h" and "type_wf h"
  assumes "document_ptr |∈| document_ptr_kinds h"
  assumes "child |∈| node_ptr_kinds h"
  shows "h ⊢ ok (adopt_node document_ptr child)"
⟨proof⟩

lemma insert_node_ok:
  assumes "known_ptr parent" and "type_wf h"
  assumes "parent |∈| object_ptr_kinds h"
  assumes "¬is_character_data_ptr_kind parent"
  assumes "is_document_ptr parent ⟹ h ⊢ get_child_nodes parent →r []"
  assumes "is_document_ptr parent ⟹ ¬is_character_data_ptr_kind node"
  assumes "known_ptr (cast node)"
  shows "h ⊢ ok (a_insert_node parent node ref)"
⟨proof⟩

lemma insert_before_ok:
  assumes "heap_is_wellformed h" and "known_ptrs h" and "type_wf h"
  assumes "parent |∈| object_ptr_kinds h"
  assumes "node |∈| node_ptr_kinds h"
  assumes "¬is_character_data_ptr_kind parent"
  assumes "cast node ∉ set |h ⊢ get_ancestors parent|r"
  assumes "h ⊢ get_parent ref →r Some parent"
  assumes "is_document_ptr parent ⟹ h ⊢ get_child_nodes parent →r []"
  assumes "is_document_ptr parent ⟹ ¬is_character_data_ptr_kind node"
  shows "h ⊢ ok (insert_before parent node (Some ref))"
⟨proof⟩
end

interpretation i_insert_before_wf3?: l_insert_before_wf3Core_DOM
  get_parent get_parent_locs get_child_nodes get_child_nodes_locs set_child_nodes set_child_nodes_locs
  get_ancestors get_ancestors_locs adopt_node adopt_node_locs set_disconnected_nodes
  set_disconnected_nodes_locs get_disconnected_nodes get_disconnected_nodes_locs get_owner_document
  insert_before insert_before_locs append_child type_wf known_ptr known_ptrs heap_is_wellformed
  parent_child_rel remove_child remove_child_locs get_root_node get_root_node_locs remove
  ⟨proof⟩
declare l_insert_before_wf3Core_DOM_axioms [instances]


locale l_append_child_wfCore_DOM =
  l_adopt_nodeCore_DOM +
  l_insert_beforeCore_DOM +
  l_append_childCore_DOM +
  l_insert_before_wf +
  l_insert_before_wf2 +
  l_get_child_nodes
begin


lemma append_child_heap_is_wellformed_preserved:
  assumes wellformed: "heap_is_wellformed h"
    and append_child: "h ⊢ append_child ptr node →h h'"
    and known_ptrs: "known_ptrs h"
    and type_wf: "type_wf h"
  shows "heap_is_wellformed h'" and "type_wf h'" and "known_ptrs h'"
  ⟨proof⟩

lemma append_child_children:
  assumes "heap_is_wellformed h" and "type_wf h" and "known_ptrs h"
  assumes "h ⊢ get_child_nodes ptr →r xs"
  assumes "h ⊢ append_child ptr node →h h'"
  assumes "node ∉ set xs"
  shows "h' ⊢ get_child_nodes ptr →r xs @ [node]"
⟨proof⟩

lemma append_child_for_all_on_children:
  assumes "heap_is_wellformed h" and "type_wf h" and "known_ptrs h"
  assumes "h ⊢ get_child_nodes ptr →r xs"
  assumes "h ⊢ forall_M (append_child ptr) nodes →h h'"
  assumes "set nodes ∩ set xs = {}"
  assumes "distinct nodes"
  shows "h' ⊢ get_child_nodes ptr →r xs@nodes"
  ⟨proof⟩


lemma append_child_for_all_on_no_children:
  assumes "heap_is_wellformed h" and "type_wf h" and "known_ptrs h"
  assumes "h ⊢ get_child_nodes ptr →r []"
  assumes "h ⊢ forall_M (append_child ptr) nodes →h h'"
  assumes "distinct nodes"
  shows "h' ⊢ get_child_nodes ptr →r nodes"
  ⟨proof⟩
end

locale l_append_child_wf = l_type_wf + l_known_ptrs + l_append_child_defs + l_heap_is_wellformed_defs +
  assumes append_child_preserves_type_wf:
    "heap_is_wellformed h ⟹ type_wf h ⟹ known_ptrs h ⟹ h ⊢ append_child ptr child →h h'
                        ⟹ type_wf h'"
  assumes append_child_preserves_known_ptrs:
    "heap_is_wellformed h ⟹ type_wf h ⟹ known_ptrs h ⟹ h ⊢ append_child ptr child →h h'
                        ⟹ known_ptrs h'"
  assumes append_child_heap_is_wellformed_preserved:
    "type_wf h ⟹ known_ptrs h ⟹ heap_is_wellformed h ⟹ h ⊢ append_child ptr child →h h'
            ⟹ heap_is_wellformed h'"

interpretation i_append_child_wf?: l_append_child_wfCore_DOM get_owner_document get_parent
  get_parent_locs remove_child remove_child_locs
  get_disconnected_nodes get_disconnected_nodes_locs
  set_disconnected_nodes set_disconnected_nodes_locs
  adopt_node adopt_node_locs known_ptr type_wf get_child_nodes
  get_child_nodes_locs known_ptrs set_child_nodes
  set_child_nodes_locs remove get_ancestors get_ancestors_locs
  insert_before insert_before_locs append_child heap_is_wellformed
  parent_child_rel
  ⟨proof⟩

lemma append_child_wf_is_l_append_child_wf [instances]: "l_append_child_wf type_wf known_ptr
known_ptrs append_child heap_is_wellformed"
  ⟨proof⟩
```

6.3.12 create_element

```
locale l_create_element_wfCore_DOM =
  l_heap_is_wellformedCore_DOM known_ptr type_wf get_child_nodes get_child_nodes_locs
  get_disconnected_nodes get_disconnected_nodes_locs
  heap_is_wellformed parent_child_rel +
  l_new_element_get_disconnected_nodes get_disconnected_nodes get_disconnected_nodes_locs +
  l_set_tag_name_get_disconnected_nodes type_wf set_tag_name set_tag_name_locs
  get_disconnected_nodes get_disconnected_nodes_locs +
  l_create_elementCore_DOM get_disconnected_nodes get_disconnected_nodes_locs set_disconnected_nodes
  set_disconnected_nodes_locs set_tag_name set_tag_name_locs type_wf create_element known_ptr +
  l_new_element_get_child_nodes type_wf known_ptr get_child_nodes get_child_nodes_locs +
  l_set_tag_name_get_child_nodes type_wf set_tag_name set_tag_name_locs known_ptr
  get_child_nodes get_child_nodes_locs +
  l_set_disconnected_nodes_get_child_nodes set_disconnected_nodes set_disconnected_nodes_locs
  get_child_nodes get_child_nodes_locs +
  l_set_disconnected_nodes type_wf set_disconnected_nodes set_disconnected_nodes_locs +
  l_set_disconnected_nodes_get_disconnected_nodes  type_wf get_disconnected_nodes
  get_disconnected_nodes_locs set_disconnected_nodes set_disconnected_nodes_locs +
  l_new_element type_wf +
  l_known_ptrs known_ptr known_ptrs
  for known_ptr :: "(_::linorder) object_ptr ⇒ bool"
    and known_ptrs :: "(_) heap ⇒ bool"
    and type_wf :: "(_) heap ⇒ bool"
    and get_child_nodes :: "(_) object_ptr ⇒ ((_) heap, exception, (_) node_ptr list) prog"
    and get_child_nodes_locs :: "(_) object_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"
    and get_disconnected_nodes :: "(_) document_ptr ⇒ ((_) heap, exception, (_) node_ptr list) prog"
    and get_disconnected_nodes_locs :: "(_) document_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"
    and heap_is_wellformed :: "(_) heap ⇒ bool"
    and parent_child_rel :: "(_) heap ⇒ ((_) object_ptr × (_) object_ptr) set"
    and set_tag_name :: "(_) element_ptr ⇒ char list ⇒ ((_) heap, exception, unit) prog"
    and set_tag_name_locs :: "(_) element_ptr ⇒ ((_) heap, exception, unit) prog set"
    and set_disconnected_nodes :: "(_) document_ptr ⇒ (_) node_ptr list ⇒ ((_) heap, exception, unit)
prog"
    and set_disconnected_nodes_locs :: "(_) document_ptr ⇒ ((_) heap, exception, unit) prog set"
    and create_element :: "(_) document_ptr ⇒ char list ⇒ ((_) heap, exception, (_) element_ptr) prog"
begin
lemma create_element_preserves_wellformedness:
  assumes "heap_is_wellformed h"
    and "h ⊢ create_element document_ptr tag →h h'"
    and "type_wf h"
    and "known_ptrs h"
  shows "heap_is_wellformed h'" and "type_wf h'" and "known_ptrs h'"
⟨proof⟩
end

interpretation i_create_element_wf?: l_create_element_wf_Core_DOM known_ptr known_ptrs type_wf
  get_child_nodes get_child_nodes_locs get_disconnected_nodes
  get_disconnected_nodes_locs heap_is_wellformed parent_child_rel
  set_tag_name set_tag_name_locs
  set_disconnected_nodes set_disconnected_nodes_locs create_element
  ⟨proof⟩
declare l_create_element_wf_Core_DOM_axioms [instances]
```

6.3.13 create_character_data

```
locale l_create_character_data_wf_Core_DOM =
  l_heap_is_wellformed_Core_DOM
  known_ptr type_wf get_child_nodes get_child_nodes_locs get_disconnected_nodes
  get_disconnected_nodes_locs heap_is_wellformed parent_child_rel
  + l_new_character_data_get_disconnected_nodes
  get_disconnected_nodes get_disconnected_nodes_locs
  + l_set_val_get_disconnected_nodes
  type_wf set_val set_val_locs get_disconnected_nodes get_disconnected_nodes_locs
  + l_create_character_data_Core_DOM
  get_disconnected_nodes get_disconnected_nodes_locs set_disconnected_nodes
  set_disconnected_nodes_locs set_val set_val_locs type_wf create_character_data known_ptr
  + l_new_character_data_get_child_nodes
  type_wf known_ptr get_child_nodes get_child_nodes_locs
  + l_set_val_get_child_nodes
  type_wf set_val set_val_locs known_ptr get_child_nodes get_child_nodes_locs
  + l_set_disconnected_nodes_get_child_nodes
  set_disconnected_nodes set_disconnected_nodes_locs get_child_nodes get_child_nodes_locs
  + l_set_disconnected_nodes
  type_wf set_disconnected_nodes set_disconnected_nodes_locs
  + l_set_disconnected_nodes_get_disconnected_nodes
  type_wf get_disconnected_nodes get_disconnected_nodes_locs set_disconnected_nodes
  set_disconnected_nodes_locs
  + l_new_character_data
  type_wf
  + l_known_ptrs
  known_ptr known_ptrs
  for known_ptr :: "(_::linorder) object_ptr ⇒ bool"
    and type_wf :: "(_) heap ⇒ bool"
    and get_child_nodes :: "(_) object_ptr ⇒ ((_) heap, exception, (_) node_ptr list) prog"
    and get_child_nodes_locs :: "(_) object_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"
    and get_disconnected_nodes :: "(_) document_ptr ⇒ ((_) heap, exception, (_) node_ptr list) prog"
    and get_disconnected_nodes_locs :: "(_) document_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"
    and heap_is_wellformed :: "(_) heap ⇒ bool"
    and parent_child_rel :: "(_) heap ⇒ ((_) object_ptr × (_) object_ptr) set"
    and set_val :: "(_) character_data_ptr ⇒ char list ⇒ ((_) heap, exception, unit) prog"
    and set_val_locs :: "(_) character_data_ptr ⇒ ((_) heap, exception, unit) prog set"
    and set_disconnected_nodes ::
    "(_) document_ptr ⇒ (_) node_ptr list ⇒ ((_) heap, exception, unit) prog"
    and set_disconnected_nodes_locs :: "(_) document_ptr ⇒ ((_) heap, exception, unit) prog set"
    and create_character_data ::
    "(_) document_ptr ⇒ char list ⇒ ((_) heap, exception, (_) character_data_ptr) prog"
    and known_ptrs :: "(_) heap ⇒ bool"
begin

lemma create_character_data_preserves_wellformedness:
  assumes "heap_is_wellformed h"
    and "h ⊢ create_character_data document_ptr text →h h'"
    and "type_wf h"
    and "known_ptrs h"
  shows "heap_is_wellformed h'" and "type_wf h'" and "known_ptrs h'"
⟨proof⟩
end

interpretation i_create_character_data_wf?: l_create_character_data_wf_Core_DOM known_ptr type_wf
  get_child_nodes get_child_nodes_locs get_disconnected_nodes get_disconnected_nodes_locs
  heap_is_wellformed parent_child_rel set_val set_val_locs set_disconnected_nodes
  set_disconnected_nodes_locs create_character_data known_ptrs
  ⟨proof⟩
declare l_create_character_data_wf_Core_DOM_axioms [instances]
```

6.3.14 create_document

```
locale l_create_document_wf_Core_DOM =
  l_heap_is_wellformed_Core_DOM
  known_ptr type_wf get_child_nodes get_child_nodes_locs get_disconnected_nodes
  get_disconnected_nodes_locs heap_is_wellformed parent_child_rel
  + l_new_document_get_disconnected_nodes
  get_disconnected_nodes get_disconnected_nodes_locs
  + l_create_document_Core_DOM
  create_document
  + l_new_document_get_child_nodes
  type_wf known_ptr get_child_nodes get_child_nodes_locs
  + l_new_document
  type_wf
  + l_known_ptrs
  known_ptr known_ptrs
  for known_ptr :: "(_::linorder) object_ptr ⇒ bool"
    and type_wf :: "(_) heap ⇒ bool"
    and get_child_nodes :: "(_) object_ptr ⇒ ((_) heap, exception, (_) node_ptr list) prog"
    and get_child_nodes_locs :: "(_) object_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"
    and get_disconnected_nodes :: "(_) document_ptr ⇒ ((_) heap, exception, (_) node_ptr list) prog"
    and get_disconnected_nodes_locs :: "(_) document_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"
    and heap_is_wellformed :: "(_) heap ⇒ bool"
    and parent_child_rel :: "(_) heap ⇒ ((_) object_ptr × (_) object_ptr) set"
    and set_val :: "(_) character_data_ptr ⇒ char list ⇒ ((_) heap, exception, unit) prog"
    and set_val_locs :: "(_) character_data_ptr ⇒ ((_) heap, exception, unit) prog set"
    and set_disconnected_nodes :: "(_) document_ptr ⇒ (_) node_ptr list ⇒ ((_) heap, exception, unit)
prog"
    and set_disconnected_nodes_locs :: "(_) document_ptr ⇒ ((_) heap, exception, unit) prog set"
    and create_document :: "((_) heap, exception, (_) document_ptr) prog"
    and known_ptrs :: "(_) heap ⇒ bool"
begin

lemma create_document_preserves_wellformedness:
  assumes "heap_is_wellformed h"
    and "h ⊢ create_document →_h h'"
    and "type_wf h"
    and "known_ptrs h"
  shows "heap_is_wellformed h'"
⟨proof⟩
end

interpretation i_create_document_wf?: l_create_document_wf_Core_DOM known_ptr type_wf get_child_nodes
  get_child_nodes_locs get_disconnected_nodes
  get_disconnected_nodes_locs heap_is_wellformed parent_child_rel
  set_val set_val_locs set_disconnected_nodes
  set_disconnected_nodes_locs create_document known_ptrs
  ⟨proof⟩
declare l_create_document_wf_Core_DOM_axioms [instances]


end
```

6.4 The Core DOM (Core_DOM)

This theory is the main entry point of our formalization of the core DOM.

```
theory Core_DOM
imports
  "Core_DOM_Heap_WF"
begin


end
```

7 Test Suite

In this chapter, we present the formalized compliance test cases for the core DOM. As our formalization is executable, we can (symbolically) execute the test cases on top of our model. Executing these test cases successfully shows that our model is compliant to the official DOM standard. As future work, we plan to generate test cases from our formal model (e.g., using [6, 8]) to improve the quality of the official compliance test suite. For more details on the relation of test and proof in the context of web standards, we refer the reader to [5].

7.1 Common Test Setup (Core_DOM_BaseTest)

This theory provides the common test setup that is used by all formalized test cases.

```
theory Core_DOM_BaseTest
  imports
    "../Core_DOM"
begin

definition "assert_throws e p = do {
  h ← get_heap;
  (if (h ⊢ p →e e) then return () else error AssertException)
}"
notation assert_throws (‹assert'_throws'(_, _')›)

definition "test p h ⟷ h ⊢ ok p"


definition field_access :: "(string ⇒ (_, (_) object_ptr option) dom_prog) ⇒ string
                                ⇒ (_, (_) object_ptr option) dom_prog"  (infix ‹.› 80)
  where
    "field_access m field = m field"

definition assert_equals :: "'a ⇒ 'a ⇒ (_, unit) dom_prog"
  where
    "assert_equals l r = (if l = r then return () else error AssertException)"
definition assert_equals_with_message :: "'a ⇒ 'a ⇒ 'b ⇒ (_, unit) dom_prog"
  where
    "assert_equals_with_message l r _ = (if l = r then return () else error AssertException)"
notation assert_equals (‹assert'_equals'(_, _')›)
notation assert_equals_with_message (‹assert'_equals'(_, _, _')›)
notation assert_equals (‹assert'_array'_equals'(_, _')›)
notation assert_equals_with_message (‹assert'_array'_equals'(_, _, _')›)

definition assert_not_equals :: "'a ⇒ 'a ⇒ (_, unit) dom_prog"
  where
    "assert_not_equals l r = (if l ≠ r then return () else error AssertException)"
definition assert_not_equals_with_message :: "'a ⇒ 'a ⇒ 'b ⇒ (_, unit) dom_prog"
  where
    "assert_not_equals_with_message l r _ = (if l ≠ r then return () else error AssertException)"
notation assert_not_equals (‹assert'_not'_equals'(_, _')›)
notation assert_not_equals_with_message (‹assert'_not'_equals'(_, _, _')›)
notation assert_not_equals (‹assert'_array'_not'_equals'(_, _')›)
notation assert_not_equals_with_message (‹assert'_array'_not'_equals'(_, _, _')›)

definition removeWhiteSpaceOnlyTextNodes :: "((_) object_ptr option) ⇒ (_, unit) dom_prog"
  where
    "removeWhiteSpaceOnlyTextNodes _ = return ()"
```

7.1.1 Making the functions under test compatible with untyped languages such as JavaScript

```
fun set_attribute_with_null :: "((_) object_ptr option) ⇒ attr_key ⇒ attr_value ⇒ (_, unit) dom_prog"
  where
    "set_attribute_with_null (Some ptr) k v = (case cast ptr of
      Some element_ptr ⇒ set_attribute element_ptr k (Some v))"
fun set_attribute_with_null2 :: "((_) object_ptr option) ⇒ attr_key ⇒ attr_value option ⇒ (_, unit) dom_prog"
  where
    "set_attribute_with_null2 (Some ptr) k v = (case cast ptr of
      Some element_ptr ⇒ set_attribute element_ptr k v)"
notation set_attribute_with_null (‹_ . setAttribute'(_, _')›)
notation set_attribute_with_null2 (‹_ . setAttribute'(_, _')›)

fun get_child_nodes_Core_DOM_with_null :: "((_) object_ptr option) ⇒ (_, (_) object_ptr option list) dom_prog"
  where
    "get_child_nodes_Core_DOM_with_null (Some ptr) = do {
      children ← get_child_nodes ptr;
      return (map (Some ∘ cast) children)
    }"
notation get_child_nodes_Core_DOM_with_null (‹_ . childNodes›)

fun create_element_with_null :: "((_) object_ptr option) ⇒ string ⇒ (_, ((_) object_ptr option)) dom_prog"
  where
    "create_element_with_null (Some owner_document_obj) tag = (case cast owner_document_obj of
      Some owner_document ⇒ do {
        element_ptr ← create_element owner_document tag;
        return (Some (cast element_ptr))})"
notation create_element_with_null (‹_ . createElement'(_')›)

fun create_character_data_with_null :: "((_) object_ptr option) ⇒ string ⇒ (_, ((_) object_ptr option))
dom_prog"
  where
    "create_character_data_with_null (Some owner_document_obj) tag = (case cast owner_document_obj of
      Some owner_document ⇒ do {
        character_data_ptr ← create_character_data owner_document tag;
        return (Some (cast character_data_ptr))})"
notation create_character_data_with_null (‹_ . createTextNode'(_')›)

definition create_document_with_null :: "string ⇒ (_, ((_::linorder) object_ptr option)) dom_prog"
  where
    "create_document_with_null title = do {
      new_document_ptr ← create_document;
      html ← create_element new_document_ptr ''html'';
      append_child (cast new_document_ptr) (cast html);
      heap ← create_element new_document_ptr ''heap'';
      append_child (cast html) (cast heap);
      body ← create_element new_document_ptr ''body'';
      append_child (cast html) (cast body);
      return (Some (cast new_document_ptr))
    }"
abbreviation "create_document_with_null2 _ _ _ ≡ create_document_with_null ''''"
notation create_document_with_null (‹createDocument'(_')›)
notation create_document_with_null2 (‹createDocument'(_, _, _')›)

fun get_element_by_id_with_null :: "((_::linorder) object_ptr option) ⇒ string ⇒ (_, ((_) object_ptr option))
dom_prog"
  where
    "get_element_by_id_with_null (Some ptr) id' = do {
        element_ptr_opt ← get_element_by_id ptr id';
        (case element_ptr_opt of
          Some element_ptr ⇒ return (Some (cast_element_ptr2object_ptr element_ptr))
        | None ⇒ return None)}"
  | "get_element_by_id_with_null _ _ = error SegmentationFault"
notation get_element_by_id_with_null (<_ . getElementById'(_')>)

fun get_elements_by_class_name_with_null ::
"((_::linorder) object_ptr option) ⇒ string ⇒ (_, ((_) object_ptr option) list) dom_prog"
  where
    "get_elements_by_class_name_with_null (Some ptr) class_name =
      get_elements_by_class_name ptr class_name ≫= map_M (return ∘ Some ∘ cast_element_ptr2object_ptr)"
notation get_elements_by_class_name_with_null (<_ . getElementsByClassName'(_')>)

fun get_elements_by_tag_name_with_null ::
"((_::linorder) object_ptr option) ⇒ string ⇒ (_, ((_) object_ptr option) list) dom_prog"
  where
    "get_elements_by_tag_name_with_null (Some ptr) tag =
      get_elements_by_tag_name ptr tag ≫= map_M (return ∘ Some ∘ cast_element_ptr2object_ptr)"
notation get_elements_by_tag_name_with_null (<_ . getElementsByTagName'(_')>)

fun insert_before_with_null ::
"((_::linorder) object_ptr option) ⇒ ((_) object_ptr option) ⇒ ((_) object_ptr option) ⇒
(_, ((_) object_ptr option)) dom_prog"
  where
    "insert_before_with_null (Some ptr) (Some child_obj) ref_child_obj_opt = (case cast child_obj of
      Some child ⇒ do {
        (case ref_child_obj_opt of
          Some ref_child_obj ⇒ insert_before ptr child (cast ref_child_obj)
        | None ⇒ insert_before ptr child None);
        return (Some child_obj)}
    | None ⇒ error HierarchyRequestError)"
notation insert_before_with_null (<_ . insertBefore'(_, _')>)

fun append_child_with_null :: "((_::linorder) object_ptr option) ⇒ ((_) object_ptr option) ⇒
(_, unit) dom_prog"
  where
    "append_child_with_null (Some ptr) (Some child_obj) = (case cast child_obj of
      Some child ⇒ append_child ptr child
    | None ⇒ error SegmentationFault)"
notation append_child_with_null (<_ . appendChild'(_')>)

fun get_body :: "((_::linorder) object_ptr option) ⇒ (_, ((_) object_ptr option)) dom_prog"
  where
    "get_body ptr = do {
       ptrs ← ptr . getElementsByTagName(''body'');
       return (hd ptrs)
    }"
notation get_body (<_ . body>)

fun get_document_element_with_null :: "((_::linorder) object_ptr option) ⇒
(_, ((_) object_ptr option)) dom_prog"
  where
    "get_document_element_with_null (Some ptr) = (case cast_object_ptr2document_ptr ptr of
    Some document_ptr ⇒ do {
      element_ptr_opt ← get_M document_ptr document_element;
      return (case element_ptr_opt of
        Some element_ptr ⇒ Some (cast_element_ptr2object_ptr element_ptr)
      | None ⇒ None)})"
notation get_document_element_with_null (<_ . documentElement>)

fun get_owner_document_with_null :: "((_::linorder) object_ptr option) ⇒
(_, ((_) object_ptr option)) dom_prog"
  where
    "get_owner_document_with_null (Some ptr) = (do {
        document_ptr ← get_owner_document ptr;
        return (Some (cast_document_ptr2object_ptr document_ptr))})"
```

```
notation get_owner_document_with_null (‹_ . ownerDocument›)

fun remove_with_null :: "((_::linorder) object_ptr option) ⇒ ((_) object_ptr option) ⇒
(_, ((_) object_ptr option)) dom_prog"
  where
    "remove_with_null (Some ptr) (Some child) = (case cast child of
      Some child_node ⇒ do {
        remove child_node;
        return (Some child)}
    | None ⇒ error NotFoundError)"
  | "remove_with_null None _ = error TypeError"
  | "remove_with_null _ None = error TypeError"
notation remove_with_null (‹_ . remove'(')›)

fun remove_child_with_null :: "((_::linorder) object_ptr option) ⇒ ((_) object_ptr option) ⇒
(_, ((_) object_ptr option)) dom_prog"
  where
    "remove_child_with_null (Some ptr) (Some child) = (case cast child of
      Some child_node ⇒ do {
        remove_child ptr child_node;
        return (Some child)}
    | None ⇒ error NotFoundError)"
  | "remove_child_with_null None _ = error TypeError"
  | "remove_child_with_null _ None = error TypeError"
notation remove_child_with_null (‹_ . removeChild›)

fun get_tag_name_with_null :: "((_) object_ptr option) ⇒ (_, attr_value) dom_prog"
  where
    "get_tag_name_with_null (Some ptr) = (case cast ptr of
      Some element_ptr ⇒ get_M element_ptr tag_name)"
notation get_tag_name_with_null (‹_ . tagName›)

abbreviation "remove_attribute_with_null ptr k ≡ set_attribute_with_null2 ptr k None"
notation remove_attribute_with_null (‹_ . removeAttribute'(_')›)

fun get_attribute_with_null :: "((_) object_ptr option) ⇒ attr_key ⇒ (_, attr_value option) dom_prog"
  where
    "get_attribute_with_null (Some ptr) k = (case cast ptr of
      Some element_ptr ⇒ get_attribute element_ptr k)"
fun get_attribute_with_null2 :: "((_) object_ptr option) ⇒ attr_key ⇒ (_, attr_value) dom_prog"
  where
    "get_attribute_with_null2 (Some ptr) k = (case cast ptr of
      Some element_ptr ⇒ do {
        a ← get_attribute element_ptr k;
        return (the a)})"
notation get_attribute_with_null (‹_ . getAttribute'(_')›)
notation get_attribute_with_null2 (‹_ . getAttribute'(_')›)

fun get_parent_with_null :: "((_::linorder) object_ptr option) ⇒ (_, (_) object_ptr option) dom_prog"
  where
    "get_parent_with_null (Some ptr) = (case cast ptr of
      Some node_ptr ⇒ get_parent node_ptr)"
notation get_parent_with_null (‹_ . parentNode›)

fun first_child_with_null :: "((_) object_ptr option) ⇒ (_, ((_) object_ptr option)) dom_prog"
  where
    "first_child_with_null (Some ptr) = do {
      child_opt ← first_child ptr;
      return (case child_opt of
        Some child ⇒ Some (cast child)
      | None ⇒ None)}"
notation first_child_with_null (‹_ . firstChild›)
```

```
fun adopt_node_with_null ::
"((_::linorder) object_ptr option) ⇒ ((_) object_ptr option) ⇒(_, ((_) object_ptr option)) dom_prog"
  where
    "adopt_node_with_null (Some ptr) (Some child) = (case cast ptr of
      Some document_ptr ⇒ (case cast child of
        Some child_node ⇒ do {
          adopt_node document_ptr child_node;
          return (Some child)}))"
notation adopt_node_with_null (‹_ . adoptNode'(_')›)


definition createTestTree ::
"((_::linorder) object_ptr option) ⇒ (_, (string ⇒  (_, ((_) object_ptr option)) dom_prog)) dom_prog"
  where
   "createTestTree ref = return (λid. get_element_by_id_with_null ref id)"

end
```

7.2 Testing Document_adoptNode (Document_adoptNode)

This theory contains the test cases for Document_adoptNode.

```
theory Document_adoptNode
imports
  "Core_DOM_BaseTest"
begin

definition Document_adoptNode_heap :: heap_final where
  "Document_adoptNode_heap = create_heap [(cast (document_ptr.Ref 1), cast (create_document_obj html (Some
(cast (element_ptr.Ref 1))) [])),
    (cast (element_ptr.Ref 1), cast (create_element_obj ''html'' [cast (element_ptr.Ref 2), cast (element_ptr.Ref
8)] fmempty None)),
    (cast (element_ptr.Ref 2), cast (create_element_obj ''head'' [cast (element_ptr.Ref 3), cast (element_ptr.Ref
4), cast (element_ptr.Ref 5), cast (element_ptr.Ref 6), cast (element_ptr.Ref 7)] fmempty None)),
    (cast (element_ptr.Ref 3), cast (create_element_obj ''meta'' [] (fmap_of_list [(''charset'', ''utf-8'')])
None)),
    (cast (element_ptr.Ref 4), cast (create_element_obj ''title'' [cast (character_data_ptr.Ref 1)] fmempty
None)),
    (cast (character_data_ptr.Ref 1), cast (create_character_data_obj ''Document.adoptNode'')),
    (cast (element_ptr.Ref 5), cast (create_element_obj ''link'' [] (fmap_of_list [(''rel'', ''help''),
(''href'', ''https://dom.spec.whatwg.org/#dom-document-adoptnode'')]) None)),
    (cast (element_ptr.Ref 6), cast (create_element_obj ''script'' [] (fmap_of_list [(''src'', ''/resources/testhar
None)),
    (cast (element_ptr.Ref 7), cast (create_element_obj ''script'' [] (fmap_of_list [(''src'', ''/resources/testhar
None)),
    (cast (element_ptr.Ref 8), cast (create_element_obj ''body'' [cast (element_ptr.Ref 9), cast (element_ptr.Ref
10), cast (element_ptr.Ref 11)] fmempty None)),
    (cast (element_ptr.Ref 9), cast (create_element_obj ''div'' [] (fmap_of_list [(''id'', ''log'')]) None)),
    (cast (element_ptr.Ref 10), cast (create_element_obj ''x<'' [cast (character_data_ptr.Ref 2)] fmempty
None)),
    (cast (character_data_ptr.Ref 2), cast (create_character_data_obj ''x'')),
    (cast (element_ptr.Ref 11), cast (create_element_obj ''script'' [cast (character_data_ptr.Ref 3)] fmempty
None)),
    (cast (character_data_ptr.Ref 3), cast (create_character_data_obj ''%3C%3Cscript%3E%3E''))]"

definition Document_adoptNode_document :: "(unit, unit, unit, unit, unit, unit) object_ptr option" where
"Document_adoptNode_document = Some (cast (document_ptr.Ref 1))"
```

"Adopting an Element called 'x<' should work."

```
lemma "test (do {
  tmp0 ← Document_adoptNode_document . getElementsByTagName(''x<'');
  y ← return (tmp0 ! 0);
  child ← y . firstChild;
  tmp1 ← y . parentNode;
  tmp2 ← Document_adoptNode_document . body;
  assert_equals(tmp1, tmp2);
  tmp3 ← y . ownerDocument;
  assert_equals(tmp3, Document_adoptNode_document);
  tmp4 ← Document_adoptNode_document . adoptNode(y);
  assert_equals(tmp4, y);
  tmp5 ← y . parentNode;
  assert_equals(tmp5, None);
  tmp6 ← y . firstChild;
  assert_equals(tmp6, child);
  tmp7 ← y . ownerDocument;
  assert_equals(tmp7, Document_adoptNode_document);
  tmp8 ← child . ownerDocument;
  assert_equals(tmp8, Document_adoptNode_document);
  doc ← createDocument(None, None, None);
  tmp9 ← doc . adoptNode(y);
  assert_equals(tmp9, y);
  tmp10 ← y . parentNode;
  assert_equals(tmp10, None);
  tmp11 ← y . firstChild;
  assert_equals(tmp11, child);
  tmp12 ← y . ownerDocument;
  assert_equals(tmp12, doc);
  tmp13 ← child . ownerDocument;
  assert_equals(tmp13, doc)
}) Document_adoptNode_heap"
  ⟨proof⟩
```

"Adopting an Element called ':good:times:' should work."

```
lemma "test (do {
  x ← Document_adoptNode_document . createElement('':good:times:'');
  tmp0 ← Document_adoptNode_document . adoptNode(x);
  assert_equals(tmp0, x);
  doc ← createDocument(None, None, None);
  tmp1 ← doc . adoptNode(x);
  assert_equals(tmp1, x);
  tmp2 ← x . parentNode;
  assert_equals(tmp2, None);
  tmp3 ← x . ownerDocument;
  assert_equals(tmp3, doc)
}) Document_adoptNode_heap"
  ⟨proof⟩


end
```

7.3 Testing Document_getElementById (Document_getElementById)

This theory contains the test cases for Document_getElementById.

```
theory Document_getElementById
imports
  "Core_DOM_BaseTest"
begin

definition Document_getElementById_heap :: heap_final where
  "Document_getElementById_heap = create_heap [(cast (document_ptr.Ref 1), cast (create_document_obj html
(Some (cast (element_ptr.Ref 1))) [])),
    (cast (element_ptr.Ref 1), cast (create_element_obj ''html'' [cast (element_ptr.Ref 2), cast (element_ptr.Ref
9)] fmempty None)),
    (cast (element_ptr.Ref 2), cast (create_element_obj ''head'' [cast (element_ptr.Ref 3), cast (element_ptr.Ref
4), cast (element_ptr.Ref 5), cast (element_ptr.Ref 6), cast (element_ptr.Ref 7), cast (element_ptr.Ref
```

```
8)] fmempty None)),
    (cast (element_ptr.Ref 3), cast (create_element_obj ''meta'' [] (fmap_of_list [(''charset'', ''utf-8'')])
None)),
    (cast (element_ptr.Ref 4), cast (create_element_obj ''title'' [cast (character_data_ptr.Ref 1)] fmempty
None)),
    (cast (character_data_ptr.Ref 1), cast (create_character_data_obj ''Document.getElementById'')),
    (cast (element_ptr.Ref 5), cast (create_element_obj ''link'' [] (fmap_of_list [(''rel'', ''author''),
(''title'', ''Tetsuharu OHZEKI''), (''href'', ''mailto:saneyuki.snyk@gmail.com'')]) None)),
    (cast (element_ptr.Ref 6), cast (create_element_obj ''link'' [] (fmap_of_list [(''rel'', ''help''),
(''href'', ''https://dom.spec.whatwg.org/#dom-document-getelementbyid'')]) None)),
    (cast (element_ptr.Ref 7), cast (create_element_obj ''script'' [] (fmap_of_list [(''src'', ''/resources/testhar
None)),
    (cast (element_ptr.Ref 8), cast (create_element_obj ''script'' [] (fmap_of_list [(''src'', ''/resources/testhar
None)),
    (cast (element_ptr.Ref 9), cast (create_element_obj ''body'' [cast (element_ptr.Ref 10), cast (element_ptr.Ref
11), cast (element_ptr.Ref 12), cast (element_ptr.Ref 13), cast (element_ptr.Ref 16), cast (element_ptr.Ref
19)] fmempty None)),
    (cast (element_ptr.Ref 10), cast (create_element_obj ''div'' [] (fmap_of_list [(''id'', ''log'')]) None)),
    (cast (element_ptr.Ref 11), cast (create_element_obj ''div'' [] (fmap_of_list [(''id'', '''')]) None)),
    (cast (element_ptr.Ref 12), cast (create_element_obj ''div'' [] (fmap_of_list [(''id'', ''test1'')])
None)),
    (cast (element_ptr.Ref 13), cast (create_element_obj ''div'' [cast (element_ptr.Ref 14), cast (element_ptr.Ref
15)] (fmap_of_list [(''id'', ''test5''), (''data-name'', ''1st'')]) None)),
    (cast (element_ptr.Ref 14), cast (create_element_obj ''p'' [cast (character_data_ptr.Ref 2)] (fmap_of_list
[(''id'', ''test5''), (''data-name'', ''2nd'')]) None)),
    (cast (character_data_ptr.Ref 2), cast (create_character_data_obj ''P'')),
    (cast (element_ptr.Ref 15), cast (create_element_obj ''input'' [] (fmap_of_list [(''id'', ''test5''),
(''type'', ''submit''), (''value'', ''Submit''), (''data-name'', ''3rd'')]) None)),
    (cast (element_ptr.Ref 16), cast (create_element_obj ''div'' [cast (element_ptr.Ref 17)] (fmap_of_list
[(''id'', ''outer'')]) None)),
    (cast (element_ptr.Ref 17), cast (create_element_obj ''div'' [cast (element_ptr.Ref 18)] (fmap_of_list
[(''id'', ''middle'')]) None)),
    (cast (element_ptr.Ref 18), cast (create_element_obj ''div'' [] (fmap_of_list [(''id'', ''inner'')])
None)),
    (cast (element_ptr.Ref 19), cast (create_element_obj ''script'' [cast (character_data_ptr.Ref 3)] fmempty
None)),
    (cast (character_data_ptr.Ref 3), cast (create_character_data_obj ''%3C%3Cscript%3E%3E''))]"
```

definition *Document_getElementById_document :: "(unit, unit, unit, unit, unit, unit) object_ptr option"* **where**
"Document_getElementById_document = Some (cast (document_ptr.Ref 1))"

"Document.getElementById with a script-inserted element"

lemma
```
"test (do {
  gBody ← Document_getElementById_document . body;
  TEST_ID ← return ''test2'';
  test ← Document_getElementById_document . createElement(''div'');
  test . setAttribute(''id'', TEST_ID);
  gBody . appendChild(test);
  result ← Document_getElementById_document . getElementById(TEST_ID);
  assert_not_equals(result, None, ''should not be null.'');
  tmp0 ← result . tagName;
  assert_equals(tmp0, ''div'', ''should have appended element's tag name'');
  gBody . removeChild(test);
  removed ← Document_getElementById_document . getElementById(TEST_ID);
  assert_equals(removed, None, ''should not get removed element.'')
}) Document_getElementById_heap"
```
⟨*proof*⟩

"update ‘id‘ attribute via setAttribute/removeAttribute"

lemma
```
"test (do {
  gBody ← Document_getElementById_document . body;
  TEST_ID ← return ''test3'';
  test ← Document_getElementById_document . createElement(''div'');
  test . setAttribute(''id'', TEST_ID);
  gBody . appendChild(test);
  UPDATED_ID ← return ''test3-updated'';
  test . setAttribute(''id'', UPDATED_ID);
  e ← Document_getElementById_document . getElementById(UPDATED_ID);
  assert_equals(e, test, ''should get the element with id.'');
  old ← Document_getElementById_document . getElementById(TEST_ID);
  assert_equals(old, None, ''shouldn't get the element by the old id.'');
  test . removeAttribute(''id'');
  e2 ← Document_getElementById_document . getElementById(UPDATED_ID);
  assert_equals(e2, None, ''should return null when the passed id is none in document.'')
}) Document_getElementById_heap"
  ⟨proof⟩
```

"Ensure that the id attribute only affects elements present in a document"

```
lemma "test (do {
  TEST_ID ← return ''test4-should-not-exist'';
  e ← Document_getElementById_document . createElement(''div'');
  e . setAttribute(''id'', TEST_ID);
  tmp0 ← Document_getElementById_document . getElementById(TEST_ID);
  assert_equals(tmp0, None, ''should be null'');
  tmp1 ← Document_getElementById_document . body;
  tmp1 . appendChild(e);
  tmp2 ← Document_getElementById_document . getElementById(TEST_ID);
  assert_equals(tmp2, e, ''should be the appended element'')
}) Document_getElementById_heap"
  ⟨proof⟩
```

"in tree order, within the context object's tree"

```
lemma "test (do {
  gBody ← Document_getElementById_document . body;
  TEST_ID ← return ''test5'';
  target ← Document_getElementById_document . getElementById(TEST_ID);
  assert_not_equals(target, None, ''should not be null'');
  tmp0 ← target . getAttribute(''data-name'');
  assert_equals(tmp0, ''1st'', ''should return the 1st'');
  element4 ← Document_getElementById_document . createElement(''div'');
  element4 . setAttribute(''id'', TEST_ID);
  element4 . setAttribute(''data-name'', ''4th'');
  gBody . appendChild(element4);
  target2 ← Document_getElementById_document . getElementById(TEST_ID);
  assert_not_equals(target2, None, ''should not be null'');
  tmp1 ← target2 . getAttribute(''data-name'');
  assert_equals(tmp1, ''1st'', ''should be the 1st'');
  tmp2 ← target2 . parentNode;
  tmp2 . removeChild(target2);
  target3 ← Document_getElementById_document . getElementById(TEST_ID);
  assert_not_equals(target3, None, ''should not be null'');
  tmp3 ← target3 . getAttribute(''data-name'');
  assert_equals(tmp3, ''4th'', ''should be the 4th'')
}) Document_getElementById_heap"
  ⟨proof⟩
```

"Modern browsers optimize this method with using internal id cache. This test checks that their optimization should effect only append to 'Document', not append to 'Node'."

```
lemma "test (do {
  TEST_ID ← return ''test6'';
  s ← Document_getElementById_document . createElement(''div'');
  s . setAttribute(''id'', TEST_ID);
  tmp0 ← Document_getElementById_document . createElement(''div'');
  tmp0 . appendChild(s);
  tmp1 ← Document_getElementById_document . getElementById(TEST_ID);
  assert_equals(tmp1, None, ''should be null'')
}) Document_getElementById_heap"
  ⟨proof⟩
```

"changing attribute's value via 'Attr' gotten from 'Element.attribute'."

```
lemma "test (do {
  gBody ← Document_getElementById_document . body;
  TEST_ID ← return ''test7'';
  element ← Document_getElementById_document . createElement(''div'');
  element . setAttribute(''id'', TEST_ID);
  gBody . appendChild(element);
  target ← Document_getElementById_document . getElementById(TEST_ID);
  assert_equals(target, element, ''should return the element before changing the value'');
  element . setAttribute(''id'', (TEST_ID @ ''-updated''));
  target2 ← Document_getElementById_document . getElementById(TEST_ID);
  assert_equals(target2, None, ''should return null after updated id via Attr.value'');
  target3 ← Document_getElementById_document . getElementById((TEST_ID @ ''-updated''));
  assert_equals(target3, element, ''should be equal to the updated element.'')
}) Document_getElementById_heap"
  ⟨proof⟩
```

"update 'id' attribute via element.id"

```
lemma "test (do {
  gBody ← Document_getElementById_document . body;
  TEST_ID ← return ''test12'';
  test ← Document_getElementById_document . createElement(''div'');
  test . setAttribute(''id'', TEST_ID);
  gBody . appendChild(test);
  UPDATED_ID ← return (TEST_ID @ ''-updated'');
  test . setAttribute(''id'', UPDATED_ID);
  e ← Document_getElementById_document . getElementById(UPDATED_ID);
  assert_equals(e, test, ''should get the element with id.'');
  old ← Document_getElementById_document . getElementById(TEST_ID);
  assert_equals(old, None, ''shouldn't get the element by the old id.'');
  test . setAttribute(''id'', '''');
  e2 ← Document_getElementById_document . getElementById(UPDATED_ID);
  assert_equals(e2, None, ''should return null when the passed id is none in document.'')
}) Document_getElementById_heap"
  ⟨proof⟩
```

"where insertion order and tree order don't match"

```
lemma "test (do {
  gBody ← Document_getElementById_document . body;
  TEST_ID ← return ''test13'';
  container ← Document_getElementById_document . createElement(''div'');
  container . setAttribute(''id'', (TEST_ID @ ''-fixture''));
  gBody . appendChild(container);
  element1 ← Document_getElementById_document . createElement(''div'');
  element1 . setAttribute(''id'', TEST_ID);
  element2 ← Document_getElementById_document . createElement(''div'');
  element2 . setAttribute(''id'', TEST_ID);
  element3 ← Document_getElementById_document . createElement(''div'');
  element3 . setAttribute(''id'', TEST_ID);
  element4 ← Document_getElementById_document . createElement(''div'');
  element4 . setAttribute(''id'', TEST_ID);
  container . appendChild(element2);
  container . appendChild(element4);
  container . insertBefore(element3, element4);
  container . insertBefore(element1, element2);
  test ← Document_getElementById_document . getElementById(TEST_ID);
  assert_equals(test, element1, ''should return 1st element'');
  container . removeChild(element1);
  test ← Document_getElementById_document . getElementById(TEST_ID);
  assert_equals(test, element2, ''should return 2nd element'');
  container . removeChild(element2);
  test ← Document_getElementById_document . getElementById(TEST_ID);
  assert_equals(test, element3, ''should return 3rd element'');
  container . removeChild(element3);
  test ← Document_getElementById_document . getElementById(TEST_ID);
  assert_equals(test, element4, ''should return 4th element'');
  container . removeChild(element4)
}) Document_getElementById_heap"
  ⟨proof⟩
```

"Inserting an id by inserting its parent node"

```
lemma "test (do {
  gBody ← Document_getElementById_document . body;
  TEST_ID ← return ''test14'';
  a ← Document_getElementById_document . createElement(''a'');
  b ← Document_getElementById_document . createElement(''b'');
  a . appendChild(b);
  b . setAttribute(''id'', TEST_ID);
  tmp0 ← Document_getElementById_document . getElementById(TEST_ID);
  assert_equals(tmp0, None);
  gBody . appendChild(a);
  tmp1 ← Document_getElementById_document . getElementById(TEST_ID);
  assert_equals(tmp1, b)
}) Document_getElementById_heap"
  ⟨proof⟩
```

"Document.getElementById must not return nodes not present in document"

```
lemma "test (do {
  TEST_ID ← return ''test15'';
  outer ← Document_getElementById_document . getElementById(''outer'');
  middle ← Document_getElementById_document . getElementById(''middle'');
  inner ← Document_getElementById_document . getElementById(''inner'');
  tmp0 ← Document_getElementById_document . getElementById(''middle'');
  outer . removeChild(tmp0);
  new_el ← Document_getElementById_document . createElement(''h1'');
  new_el . setAttribute(''id'', ''heading'');
  inner . appendChild(new_el);
  tmp1 ← Document_getElementById_document . getElementById(''heading'');
  assert_equals(tmp1, None)
}) Document_getElementById_heap"
  ⟨proof⟩


end
```

7.4 Testing Node_insertBefore (Node_insertBefore)

This theory contains the test cases for Node_insertBefore.

```
theory Node_insertBefore
imports
  "Core_DOM_BaseTest"
begin

definition Node_insertBefore_heap :: heap_final  where
  "Node_insertBefore_heap = create_heap [(cast (document_ptr.Ref 1), cast (create_document_obj html (Some
(cast (element_ptr.Ref 1))) [])),
    (cast (element_ptr.Ref 1), cast (create_element_obj ''html'' [cast (element_ptr.Ref 2), cast (element_ptr.Ref
6)] fmempty None)),
    (cast (element_ptr.Ref 2), cast (create_element_obj ''head'' [cast (element_ptr.Ref 3), cast (element_ptr.Ref
4), cast (element_ptr.Ref 5)] fmempty None)),
    (cast (element_ptr.Ref 3), cast (create_element_obj ''title'' [cast (character_data_ptr.Ref 1)] fmempty
None)),
    (cast (character_data_ptr.Ref 1), cast (create_character_data_obj ''Node.insertBefore'')),
    (cast (element_ptr.Ref 4), cast (create_element_obj ''script'' [] (fmap_of_list [(''src'', ''/resources/testhar
None)),
    (cast (element_ptr.Ref 5), cast (create_element_obj ''script'' [] (fmap_of_list [(''src'', ''/resources/testhar
None)),
    (cast (element_ptr.Ref 6), cast (create_element_obj ''body'' [cast (element_ptr.Ref 7), cast (element_ptr.Ref
8)] fmempty None)),
    (cast (element_ptr.Ref 7), cast (create_element_obj ''div'' [] (fmap_of_list [(''id'', ''log'')]) None)),
    (cast (element_ptr.Ref 8), cast (create_element_obj ''script'' [cast (character_data_ptr.Ref 2)] fmempty
None)),
    (cast (character_data_ptr.Ref 2), cast (create_character_data_obj ''%3C%3Cscript%3E%3E''))]"
```

definition *Node_insertBefore_document :: "(unit, unit, unit, unit, unit, unit) object_ptr option"* **where**
"Node_insertBefore_document = Some (cast (document_ptr.Ref 1))"

"Calling insertBefore an a leaf node Text must throw HIERARCHY_REQUEST_ERR."

```
lemma "test (do {
  node ← Node_insertBefore_document . createTextNode(''Foo'');
  tmp0 ← Node_insertBefore_document . createTextNode(''fail'');
  assert_throws(HierarchyRequestError, node . insertBefore(tmp0, None))
}) Node_insertBefore_heap"
  ⟨proof⟩
```

"Calling insertBefore with an inclusive ancestor of the context object must throw HIERARCHY_REQUEST_ERR."

```
lemma "test (do {
  tmp1 ← Node_insertBefore_document . body;
  tmp2 ← Node_insertBefore_document . getElementById(''log'');
  tmp0 ← Node_insertBefore_document . body;
  assert_throws(HierarchyRequestError, tmp0 . insertBefore(tmp1, tmp2));
  tmp4 ← Node_insertBefore_document . documentElement;
  tmp5 ← Node_insertBefore_document . getElementById(''log'');
  tmp3 ← Node_insertBefore_document . body;
  assert_throws(HierarchyRequestError, tmp3 . insertBefore(tmp4, tmp5))
}) Node_insertBefore_heap"
  ⟨proof⟩
```

"Calling insertBefore with a reference child whose parent is not the context node must throw a NotFoundError."

```
lemma "test (do {
  a ← Node_insertBefore_document . createElement(''div'');
  b ← Node_insertBefore_document . createElement(''div'');
  c ← Node_insertBefore_document . createElement(''div'');
  assert_throws(NotFoundError, a . insertBefore(b, c))
}) Node_insertBefore_heap"
  ⟨proof⟩
```

"If the context node is a document, inserting a document or text node should throw a HierarchyRequestError."

```
lemma "test (do {
  doc ← createDocument(''title'');
  doc2 ← createDocument(''title2'');
  tmp0 ← doc . documentElement;
  assert_throws(HierarchyRequestError, doc . insertBefore(doc2, tmp0));
  tmp1 ← doc . createTextNode(''text'');
  tmp2 ← doc . documentElement;
  assert_throws(HierarchyRequestError, doc . insertBefore(tmp1, tmp2))
}) Node_insertBefore_heap"
  ⟨proof⟩
```

"Inserting a node before itself should not move the node"

```
lemma "test (do {
  a ← Node_insertBefore_document . createElement(''div'');
  b ← Node_insertBefore_document . createElement(''div'');
  c ← Node_insertBefore_document . createElement(''div'');
  a . appendChild(b);
  a . appendChild(c);
  tmp0 ← a . childNodes;
  assert_array_equals(tmp0, [b, c]);
  tmp1 ← a . insertBefore(b, b);
  assert_equals(tmp1, b);
  tmp2 ← a . childNodes;
  assert_array_equals(tmp2, [b, c]);
  tmp3 ← a . insertBefore(c, c);
  assert_equals(tmp3, c);
  tmp4 ← a . childNodes;
  assert_array_equals(tmp4, [b, c])
}) Node_insertBefore_heap"
  ⟨proof⟩
```

end

7.5 Testing Node_removeChild (Node_removeChild)

This theory contains the test cases for Node_removeChild.

theory *Node_removeChild*
imports
  "Core_DOM_BaseTest"
begin

definition *Node_removeChild_heap :: heap$_{final}$* **where**
```
  "Node_removeChild_heap = create_heap [(cast (document_ptr.Ref 1), cast (create_document_obj html (Some
(cast (element_ptr.Ref 1))) [])),
    (cast (element_ptr.Ref 1), cast (create_element_obj ''html'' [cast (element_ptr.Ref 2), cast (element_ptr.Ref
7)] fmempty None)),
    (cast (element_ptr.Ref 2), cast (create_element_obj ''head'' [cast (element_ptr.Ref 3), cast (element_ptr.Ref
4), cast (element_ptr.Ref 5), cast (element_ptr.Ref 6)] fmempty None)),
    (cast (element_ptr.Ref 3), cast (create_element_obj ''title'' [cast (character_data_ptr.Ref 1)] fmempty
None)),
    (cast (character_data_ptr.Ref 1), cast (create_character_data_obj ''Node.removeChild'')),
    (cast (element_ptr.Ref 4), cast (create_element_obj ''script'' [] (fmap_of_list [(''src'', ''/resources/testhar
None)),
    (cast (element_ptr.Ref 5), cast (create_element_obj ''script'' [] (fmap_of_list [(''src'', ''/resources/testhar
None)),
    (cast (element_ptr.Ref 6), cast (create_element_obj ''script'' [] (fmap_of_list [(''src'', ''creators.js'')])
None)),
    (cast (element_ptr.Ref 7), cast (create_element_obj ''body'' [cast (element_ptr.Ref 8), cast (element_ptr.Ref
9), cast (element_ptr.Ref 10)] fmempty None)),
    (cast (element_ptr.Ref 8), cast (create_element_obj ''div'' [] (fmap_of_list [(''id'', ''log'')]) None)),
    (cast (element_ptr.Ref 9), cast (create_element_obj ''iframe'' [] (fmap_of_list [(''src'', ''about:blank'')])
None)),
    (cast (element_ptr.Ref 10), cast (create_element_obj ''script'' [cast (character_data_ptr.Ref 2)] fmempty
None)),
    (cast (character_data_ptr.Ref 2), cast (create_character_data_obj ''%3C%3Cscript%3E%3E''))]"
```

definition *Node_removeChild_document :: "(unit, unit, unit, unit, unit, unit) object_ptr option"* **where** *"Node_remov*
= Some (cast (document_ptr.Ref 1))"

"Passing a detached Element to removeChild should not affect it."

lemma *"test (do {*
```
  doc ← return Node_removeChild_document;
  s ← doc . createElement(''div'');
  tmp0 ← s . ownerDocument;
  assert_equals(tmp0, doc);
  tmp1 ← Node_removeChild_document . body;
  assert_throws(NotFoundError, tmp1 . removeChild(s));
  tmp2 ← s . ownerDocument;
  assert_equals(tmp2, doc)
}) Node_removeChild_heap"
  ⟨proof⟩
```

"Passing a non-detached Element to removeChild should not affect it."

```
lemma "test (do {
  doc ← return Node_removeChild_document;
  s ← doc . createElement(''div'');
  tmp0 ← doc . documentElement;
  tmp0 . appendChild(s);
  tmp1 ← s . ownerDocument;
  assert_equals(tmp1, doc);
  tmp2 ← Node_removeChild_document . body;
  assert_throws(NotFoundError, tmp2 . removeChild(s));
  tmp3 ← s . ownerDocument;
  assert_equals(tmp3, doc)
}) Node_removeChild_heap"
  ⟨proof⟩
```

"Calling removeChild on an Element with no children should throw NOT_FOUND_ERR."

```
lemma "test (do {
  doc ← return Node_removeChild_document;
  s ← doc . createElement(''div'');
  tmp0 ← doc . body;
  tmp0 . appendChild(s);
  tmp1 ← s . ownerDocument;
  assert_equals(tmp1, doc);
  assert_throws(NotFoundError, s . removeChild(doc))
}) Node_removeChild_heap"
  ⟨proof⟩
```

"Passing a detached Element to removeChild should not affect it."

```
lemma "test (do {
  doc ← createDocument('''');
  s ← doc . createElement(''div'');
  tmp0 ← s . ownerDocument;
  assert_equals(tmp0, doc);
  tmp1 ← Node_removeChild_document . body;
  assert_throws(NotFoundError, tmp1 . removeChild(s));
  tmp2 ← s . ownerDocument;
  assert_equals(tmp2, doc)
}) Node_removeChild_heap"
  ⟨proof⟩
```

"Passing a non-detached Element to removeChild should not affect it."

```
lemma "test (do {
  doc ← createDocument('''');
  s ← doc . createElement(''div'');
  tmp0 ← doc . documentElement;
  tmp0 . appendChild(s);
  tmp1 ← s . ownerDocument;
  assert_equals(tmp1, doc);
  tmp2 ← Node_removeChild_document . body;
  assert_throws(NotFoundError, tmp2 . removeChild(s));
  tmp3 ← s . ownerDocument;
  assert_equals(tmp3, doc)
}) Node_removeChild_heap"
  ⟨proof⟩
```

"Calling removeChild on an Element with no children should throw NOT_FOUND_ERR."

```
lemma "test (do {
  doc ← createDocument('''');
  s ← doc . createElement(''div'');
  tmp0 ← doc . body;
  tmp0 . appendChild(s);
  tmp1 ← s . ownerDocument;
  assert_equals(tmp1, doc);
  assert_throws(NotFoundError, s . removeChild(doc))
}) Node_removeChild_heap"
  ⟨proof⟩
```

"Passing a value that is not a Node reference to removeChild should throw TypeError."

```
lemma "test (do {
  tmp0 ← Node_removeChild_document . body;
  assert_throws(TypeError, tmp0 . removeChild(None))
}) Node_removeChild_heap"
  ⟨proof⟩


end
```

7.6 Core DOM Test Cases (Core_DOM_Tests)

This theory aggregates the individual test cases for the core DOM.

```
theory Core_DOM_Tests
  imports
    "tests/Document_adoptNode"
    "tests/Document_getElementById"
    "tests/Node_insertBefore"
    "tests/Node_removeChild"
begin
end
```

Bibliography

[1] DOM Living Standard – Last Updated 20 October 2016 2016. URL https://dom.spec.whatwg.org/. An archived copy of the version from 20 October 2016 is available at https://git.logicalhacking.com/BrowserSecurity/fDOM-idl/.

[2] A. Bohannon and B. C. Pierce. Featherweight Firefox: Formalizing the core of a web browser. In *Usenix Conference on Web Application Development (WebApps)*, June 2010. URL http://www.cis.upenn.edu/~bohannon/browser-model/.

[3] A. D. Brucker. *An Interactive Proof Environment for Object-oriented Specifications*. PhD thesis, ETH Zurich, mar 2007. URL https://www.brucker.ch/bibliography/abstract/brucker-interactive-2007. ETH Dissertation No. 17097.

[4] A. D. Brucker and M. Herzberg. A formal semantics of the core DOM in Isabelle/HOL. In *Proceedings of the Web Programming, Design, Analysis, And Implementation (WPDAI) track at WWW 2018*, 2018. URL https://www.brucker.ch/bibliography/abstract/brucker.ea-fdom-2018.

[5] A. D. Brucker and M. Herzberg. Formalizing (web) standards: An application of test and proof. In C. Dubois and B. Wolff, editors, *TAP 2018: Tests And Proofs*, number 10889 in Lecture Notes in Computer Science, pages 159–166. Springer-Verlag, Heidelberg, 2018. ISBN 978-3-642-38915-3. doi: 10.1007/978-3-319-92994-1_9. URL http://www.brucker.ch/bibliography/abstract/brucker.ea-standard-compliance-testing-2018.

[6] A. D. Brucker and B. Wolff. Interactive testing using HOL-TestGen. In W. Grieskamp and C. Weise, editors, *Formal Approaches to Testing of Software*, number 3997 in Lecture Notes in Computer Science. Springer-Verlag, Heidelberg, 2005. ISBN 3-540-25109-X. doi: 10.1007/11759744_7. URL http://www.brucker.ch/bibliography/abstract/brucker.ea-interactive-2005.

[7] A. D. Brucker and B. Wolff. An extensible encoding of object-oriented data models in hol. *Journal of Automated Reasoning*, 41:219–249, 2008. ISSN 0168-7433. doi: 10.1007/s10817-008-9108-3. URL https://www.brucker.ch/bibliography/abstract/brucker.ea-extensible-2008-b.

[8] A. D. Brucker and B. Wolff. On theorem prover-based testing. *Formal Aspects of Computing*, 25(5):683–721, 2013. ISSN 0934-5043. doi: 10.1007/s00165-012-0222-y. URL http://www.brucker.ch/bibliography/abstract/brucker.ea-theorem-prover-2012.

[9] P. Gardner, G. Smith, M. J. Wheelhouse, and U. Zarfaty. DOM: towards a formal specification. In *PLAN-X 2008, Programming Language Technologies for XML, An ACM SIGPLAN Workshop colocated with POPL 2008, San Francisco, California, USA, January 9, 2008*, 2008. URL http://gemo.futurs.inria.fr/events/PLANX2008/papers/p18.pdf.

[10] D. Jang, Z. Tatlock, and S. Lerner. Establishing browser security guarantees through formal shim verification. In T. Kohno, editor, *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, pages 113–128. USENIX Association, 2012. URL https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/jang.

[11] J. J. Joyce and C.-J. H. Seger, editors. *Higher Order Logic Theorem Proving and Its Applications (HUG)*, volume 780 of *Lecture Notes in Computer Science*, Heidelberg, 1994. Springer-Verlag. ISBN 3-540-57826-9. doi: 10.1007/3-540-57826-9.

[12] G. Klein. Operating system verification — an overview. *Sādhanā*, 34(1):27–69, Feb. 2009.

[13] A. Raad, J. F. Santos, and P. Gardner. DOM: specification and client reasoning. In A. Igarashi, editor, *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings*, volume 10017 of *Lecture Notes in Computer Science*, pages 401–422, 2016. ISBN 978-3-319-47957-6. doi: 10.1007/978-3-319-47958-3_21.

[14] W3C. W3C DOM4, Nov. 2015. URL https://www.w3.org/TR/dom/.

[15] WHATWG. DOM – living standard, Mar. 2017. URL https://dom.spec.whatwg.org/commit-snapshots/6253e53af2fbfaa6d25ad09fd54280d8083b2a97/. Last Updated 24 March 2017.